Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases

Dissertation with the aim of achieving a doctoral degree at the
Faculty of Mathematics, Informatics, and Natural Sciences

Submitted at the University of Hamburg
by Wolfram Wingerath, 2018

Day of oral defense: May 8th, 2019

The following evaluators recommend the admission of the dissertation:

Prof. Dr.-Ing. Norbert Ritter (Universität Hamburg)

Prof. Dr. Daniela Nicklas (Otto-Friedrich-Universität Bamberg)

Prof. Dr. Erhard Rahm (Universität Leipzig)

“Machen!”
—Anonymous

This document was created on May 18th, 2019.

This thesis and revisions, if any, will be made available at `https://invalidb.info/thesis`.

Acknowledgments

> "I love deadlines. I like the whooshing sound they make as they fly by."
>
> —Douglas Adams

So I guess this is it. After about a quarter of a century, I'm finally leaving my disciple status behind and move on to actually build things – I'm excited! But as another chapter of my life is about to close, I feel obliged to stop and think for a moment in order to acknowledge the amplitude of help I had in getting where I am now.

I'd like to put my colleagues Steffen and Anne first, because they picked up my slack more times than I dare to admit. I'm sure my studies would have been twice as hard and half the fun without you two – thank you for everything! Then I'd like to apologize to my advisor Norbert Ritter for driving him out of the pole position in these acknowledgments. More importantly, though, I want to thank him for providing me the opportunity to undertake this endeavor in the first place and for the various lessons on pragmatism he has taught me over the years. I'd also like to express my gratitude towards Daniela Nicklas for assessing my thesis and for inspiring me to put the foot down during the final months, even though I still did not manage to wrap everything up before Christmas ...or Easter ...or midsummer...I'm further supremely thankful to Felix who is not only tall and handsome, but equally inspiring and frustrating to argue with. You have been – and continue to be! – an invaluable resource of devastatingly helpful criticism! My gratitude also goes to Fabian with whom I've had many fruitful discussions and countless more that were utterly pointless: I cannot say which ones I enjoyed more – thank you for them all! Gabriel and Dirk deserve mentioning, too, not only because they provided occasional distraction through pleasant chats, but for their most valuable feedback on this dissertation during the critical phase. And I want to tip my hat to all the students with whom I have collaborated over the years, most of all Stephan, Marcel, and Randy, because they contributed to this work with code as well as clarifying discussions – and simply for being so much fun to work with.

Next, I want to acknowledge the help from the awesome people at Baqend, some of whom have put as great a deal of sweat and tears into the InvaliDB production deployment as myself. Chief amongst them is Flo as the ultimate exception handler: Thank you for taking time you didn't have to solve problems you didn't cause! Special thanks go to Erik for churning out articles and conference talks with me that earned the highest praise ("nicht soviel schlechter als manche der anderen"). But also to those not entangled in my scientific activities, thank you for making office time worthwhile! I would swipe any of you right any time – in randomized order: Gitte, Knuth, Sven, Malte, Julian, KSM, Hannes, the other Julian, Kevi, and Jörn.

Beyond my academic life, there have been several people who have contributed to this thesis in one way or another. For starters, I have to give ELke a big smooch for her unconditional trust and support: If I needed a horse, I would definitely steal it with you! Of course, all non-horse-related stealing has been and will be done with Kowa, my brother in spirit. Persistently, he has been there when I needed a bearly man, a manly beard, or mainly beer. Also of the utmost persistence, Dadi has been with me all my life and knowing that'll never change is one of the pillars of my world. Albeit a bit clichéed, I have to stress the importance of Muddi and Vaddas who both invested a lot to see me become halfway self-sustainable. Having said this, I think it's all the more noteworthy that they simply let me alone when I needed to concentrate on my work – I promise compensation beers! Ann-Kristin deserves particular distinction for the years of listening to what must've been the most boring monologues in the history of storytelling – you were a great help in structuring my thoughts before writing them up! And, naturally, I would like to thank my assistants: not only for copious amounts of delicious coffee, but more importantly for their consistent and reliable support in virtually everything I do! Last and most certainly least, however, I want to mention Kjell who has shown me the importance of keeping a positive attitude, more vividly than anyone ever could have.

Seeing that I have shamefully ignored so many of my friends and family for way too long, I feel lucky that most will still pick up the phone when I'm calling. I solemnly swear to make up for the social neglect! Specifically and in no particular order: I'll make sure that I get to see Renie not only every other year, visit Boji in her new home, mop the floor with Jule in a round of Mensch-ärgere-dich-nicht, take Muddi to an Elphi concert, play skat with Onna, visit Lotte and Mattis with Vaddas, schnüffelate with Schnüffi (*"hannahannahannahanna-hanna*"), peacefully enjoy a glass of good wine with Walli, fix that goddamn wooden board in the kitchen with Stephan, go to the theater with Dani and Kari, watch one man making a difference with Benny, watch B-movies with Souli, make Stevi try out his new PlayStation 3 with me, pull off a comeback with Joe, find out what Mr. Unbeaten Intelligence has been up to all these years, and finally have that barbecue at Jan's ("Die Wurst ist gekauft!").

I'm afraid I might have forgotten someone who deserves acknowledgment…But that's a risk I'm willing to take – this thing needs to be printed now!

Wolfram Wingerath   Hamburg, September 24th, 2018

Contents

Abstract

English

Many of today's web applications notify users of status updates and other events in real-time. But even though more and more usage scenarios revolve around the interaction between users, detecting and publishing changes remains notoriously hard even with state-of-the-art data management systems. While traditional database systems excel at complex queries over historical data, they are inherently pull-based and therefore ill-equipped to push new information to clients. Systems for data stream management and processing, on the other hand, are natively push-oriented and thus facilitate reactive behavior. However, they do not retain data indefinitely and are therefore not able to answer historical queries. The separation between these two system classes gives rise to both high complexity and high maintenance costs for applications that require persistence and real-time change notifications at the same time. How can push-based access be enabled for database queries over historical data collections in a simple and efficient manner?

In this thesis, we explore the system space between pull-oriented database systems and push-oriented stream management systems. Specifically, we focus on the novel system class of real-time databases that bridge the gap between both paradigms by providing collection-based semantics for pull-based and push-based queries alike. Through an in-depth system survey, we uncover deficiencies in existing implementations and scale-prohibitive limitations in their respective designs. In order to address these issues, we propose the system design *InvaliDB* which makes push-based real-time queries available as an opt-in feature for existing pull-based database systems. InvaliDB exhibits several substantial benefits over current real-time database architectures. First, it avoids the scalability bottlenecks that other systems are constrained by through a novel two-dimensional workload partitioning scheme. Second, our design supports more expressive queries than its peers, including sorted filter queries with limit and offset clauses, aggregations, and joins. Third, InvaliDB is database-agnostic through a pluggable query engine and can therefore be applied to existing (pull-based) application stacks in order to enable push-based data access. We provide an experimental evaluation to demonstrate that sustainable query matching throughput scales linearly with the number of servers employed for query matching, while end-to-end notification latency remains consistently low across all InvaliDB configurations. A detailed case study of our InvaliDB prototype in a production deployment further illustrates that our approach is feasible to implement, enables easy-to-use query interfaces, and is practically useful for data-intensive industry applications.

German

Heutzutage informieren viele Webapplikationen Benutzer über Status-Updates und andere Ereignisse in Echtzeit. Aber auch wenn die Interaktion zwischen Nutzern immer häufiger in den Vordergrund rückt, so sind selbst moderne Datenverwaltungssysteme nur bedingt zur Erkennung und Propagierung von Zustandsänderungen in der Lage. Während traditionelle Datenbanken für komplexe Anfragen über historische Daten konzipiert wurden, sind sie inhärent Pull-basiert und bieten daher nur eingeschränkte Unterstützung für proaktive Datenzugriffsmuster. Systeme für Datenstromverwaltung und -verarbeitung sind dagegen Push-orientiert und ermöglichen so reaktives Verhalten. Sie speichern Daten jedoch nur für begrenzte Zeit und können folglich keine historischen Anfragen beantworten. Die Trennung zwischen diesen beiden Systemklassen bedingt sowohl hohe Komplexität als auch hohe Wartungskosten bei Anwendungen, die gleichzeitig Persistenz und Echtzeitbenachrichtigungen bei Zustandsänderungen benötigen. Wie kann Push-basierter Zugriff für Anfragen über historische Daten simpel und effizient ermöglicht werden?

In dieser Arbeit untersuchen wir das Spektrum zwischen Pull-orientierten Datenbanksystemen und Push-orientierten Systemen zur Datenstromverwaltung. Insbesondere konzentrieren wir uns auf die neuartige Systemklasse der Echtzeitdatenbanken (*real-time databases*). Systeme dieser Klasse schließen die Kluft zwischen beiden Paradigmen, indem sie die für Datenbanksysteme übliche Collection-basierte Semantik für traditionelle Pull-basierte Anfragen sowie für Push-basierte Echtzeitanfragen (*real-time queries*) unterstützen. Durch eine detaillierte Analyse aktueller Systeme decken wir Mängel in konkreten Implementationen sowie konzeptionelle Limitationen in den jeweiligen Architekturen auf. Zur Lösung dieser Probleme schlagen wir das Systemdesign *InvaliDB* vor, welches Push-basierte Echtzeitanfragen als Opt-in-Feature für existierende Pull-basierte Datenbanksysteme bereitstellt. InvaliDB verfügt über mehrere wesentliche Vorteile gegenüber bestehenden Echtzeitdatenbankarchitekturen. Erstens vermeidet es Flaschenhälse, die die Skalierbarkeit anderer Systeme einschränken, durch ein neuartiges Konzept zur zweidimensionalen Lastverteilung. Zweitens unterstützt unser Design mächtigere Echtzeitanfragen als bestehende Systeme, darunter sortierte Filteranfragen mit Limit- und Offsetklauseln, Aggregationen und Joins. Drittens abstrahiert InvaliDB durch eine austauschbare Komponente zur Anfrageverarbeitung (*pluggable query engine*) von konkreten Datenbanktechnologien und kann daher auch bestehende (Pull-basierte) Anwendungsstacks um Push-basierte Datenzugriffsmechanismen erweitern. In einer experimentellen Evaluation demonstrieren wir, dass der für eine InvaliDB-Instanz tragbare Durchsatz bei der Anfrageverarbeitung (*sustainable query matching throughput*) linear mit der Anzahl der für die Anfrageverarbeitung eingesetzten Server skaliert, wobei die Ende-zu-Ende-Latenz über alle InvaliDB-Konfigurationen hinweg konstant niedrig bleibt. Eine detaillierte Fallstudie über unseren InvaliDB-Prototypen im Produktionsbetrieb zeigt darüber hinaus, dass unser Ansatz mit überschaubarem Aufwand implementierbar ist, simple Anfrageschnittstellen ermöglicht und in datenintensiven Industrieanwendungen praktisch einsetzbar ist.

1 Introduction

"Please refresh the page to get the most current information."

—from the FAQs of an online auctioning website

This work devises scalable push-based real-time queries on top of pull-based databases.

In recent years, users have come to expect reactivity from their applications, i.e. they assume that changes made by other users are immediately reflected in the interfaces they are using. Examples are shared worksheets and websites targeting social interaction. These applications require the underlying data storage to publish new and updated information as soon as it is created: Data access is *push-based*. In contrast, traditional **database management systems** [HSH07] have been tailored towards *pull-based* data access where information is only made available as a direct response to a client request. While triggers and other push-oriented mechanisms have been added to their initial design, they are outperformed by several orders of magnitude when held against natively push-based systems [SC05]. In consequence, the inadequacy of traditional database technology for handling rapidly changing data has been widely accepted as one of the fundamental challenges in database system design [SCZ05].

To warrant low-latency updates in quickly evolving domains, **data stream management systems** [GZ10] break with the idea of maintaining a persistent data repository. Instead of random access queries on static collections, they perform sequential, long-running queries over data streams. Data stream management systems generate new output whenever new data becomes available and are thus natively push-based. However, data is only available for processing in one single pass, because data streams are conceptually *unbounded* sequences of data items and therefore infeasible to retain indefinitely. Consequently, queries over streams are confined to data that arrives after query activation.

	Database Management	**Data Stream Management**
Data Access	pull-based	push-based
Data Model	persistent collection	ephemeral stream
Query Execution	ad hoc, random access	continuous, sequential

Table 1.1: A side-by-side comparison of core characteristics of database and data stream management systems.

Database and data stream management, respectively, follow fundamentally different semantics regarding the way that data is processed and accessed as Table 1.1 summarizes. The concept of persistent collections conforms to applications that require a (consistent) view of their domain, for instance to keep track of warehouse stock or do financial accounting. The stream data model, on the other hand, comes natural for domains that facilitate a notion of event sequences or demand reasoning about the relationship between events, for example to analyze stock prices or identify malicious user behavior. However, the access paradigm – pull-based or push-based – is tied to the data model: Database management systems lack support for continuous queries over collections, whereas data stream management systems only provide limited options for persistent data handling.

Acknowledging the gap between database and data stream management systems, a new class of information systems has emerged that combines collection-based semantics with a push-based access model. These systems are often referred to as **real-time databases** [Puf16, Yu15], because they keep data at the client in-sync with current database state "in realtime" [Pau15], i.e. as soon as possible after change. Like traditional database systems, they store consistent snapshots of domain knowledge. But like stream management systems, they let clients subscribe to long-running queries that push incremental updates.

1.1 Real-Time Databases

In the past, the term "real-time databases"[1] has been used as a reference to specialized pull-based database systems that produce an output within strict timing constraints [PSS+93, AH98, Eri98]. Within the scope of this work, however, **real-time databases** are systems that provide push-based access to database collections. Popular examples are Firebase and Meteor as discussed in Section 2.3. Likewise, **real-time queries** in the context of this work are push-based queries on top of database *collections*. They follow the same collection-based querying semantics as common database queries, but respond with a continuous stream of informational updates in addition to the initial query result. In the literature, contrastingly, the term "real-time query" sometimes refers to a particular form of ad hoc pull-based query [TS+09] or to a push-based query over data streams [Ros11]; we do not share this notion of real-time queries in this thesis. Correspondingly, **real-time applications** are reactive or interactive applications that make new information available to the user as soon as possible after they have been committed to storage [Eri98]. This work addresses applications with soft timing constraints. It specifically does not address security-critical or other applications that impose strict upper bounds on response times such as flight control systems or nuclear power plant controls [Sta88].

Intuitively, the information delivered by a real-time database through a real-time query has two components: the initial result and change events. Both inherit certain characteristics from data returned by database and data stream management systems, respectively.

[1] In this thesis, we use the terms "database" and "database system" synonymously, when the context makes clear whether we are referring to the base data or the system maintaining it.

The **initial result** corresponds to the data returned by a common ad hoc database query and thus captures the data items matching on request. It is assembled from persistent storage, just like a common database query result. **Change events** (also called change notifications) are sent whenever the result is altered (e.g. when a matching data item is inserted). Thus, they capture how the result evolves over time. Through change events, the client receives all information required to maintain the initial result up-to-date.

Changes in a query result are gathered through a continuous process that sifts through the database write stream, similar to a continuous query in a stream management system. In order to identify the relevant updates for a given real-time query, the system has to inspect every single write operation that might possibly affect the query result. This task is straightforward for some queries and rather complicated for others. We only address simple filter queries (i.e. select-project queries [Bad09]) in this section. More complex queries that involve ordering, joins, or aggregations are discussed in Section 3.3.

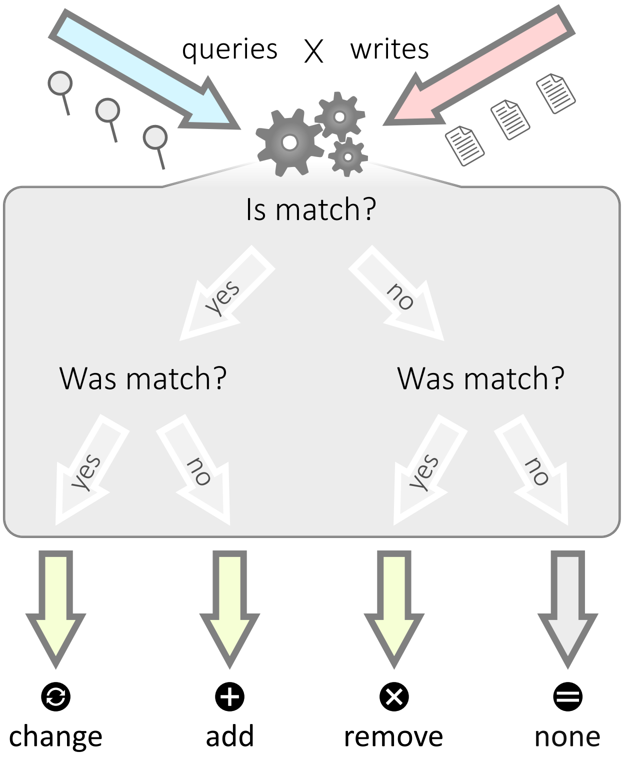


Figure 1.1: A real-time database has to match all real-time queries (blue arrow) against all incoming write operations (red arrow) to generate change notifications on result alterations (green arrows).

The decision tree in Figure 1.1 illustrates how this task can be translated to two simple questions asked for any written data item[2]:

1. Does the item match the query now? (“Is match?”)
2. Did the item match the query before? (“Was match?”)

[2] While the depicted approach does not apply to set operations, an update that affects several data items can be transformed into a set of single-item updates to make the approach applicable (cf. Section 3.1.3).

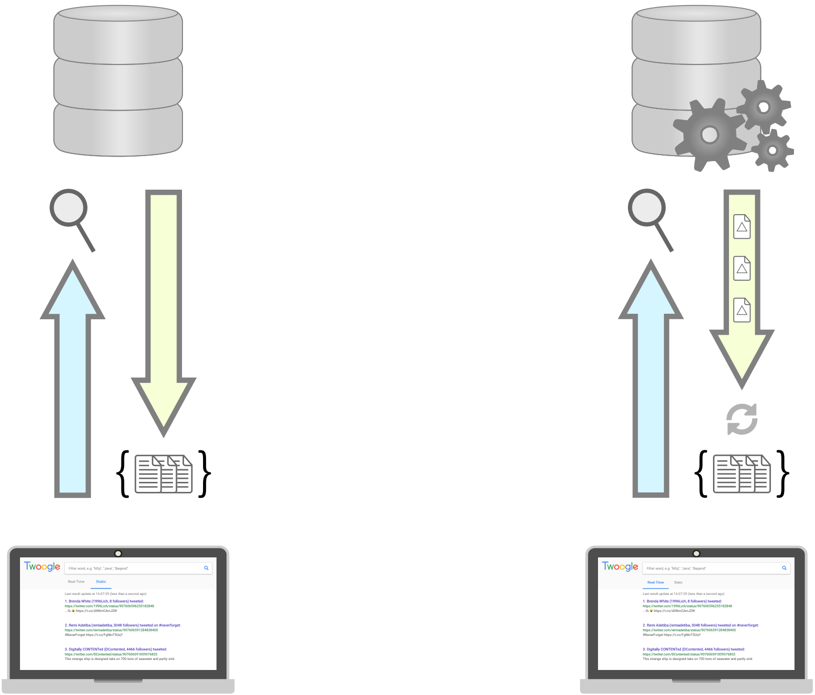
When a data item matches the query after an insert or update (left branch), it is either an already-matching item that was altered (*change*) or a former non-match that just entered the result (*add*). Similarly, whenever a data item is deleted or does not match the query after an insert or update (right branch), it either is a matching item that just left the result (*remove*) or it does not relate to the query result whatsoever (*none*).

In the presence of many concurrent real-time queries or high update throughput, this continuous monitoring process becomes very expensive. To put the resource requirements into perspective, consider an app with 1 000 concurrent users and an average throughput of 1 000 written data items per second. Given each user has only one active real-time query to filter a string attribute by some pattern, the real-time database already may have to perform 1 million matching operations – every single second. And this does not account for more complex queries: Sorted queries may require additional work to maintain result order and enforce limit or offset clauses. Similarly, queries with joins or aggregations may impose even higher **overhead**, because they necessitate maintaining counters, intermediate results, or other data structures that are implicitly required to maintain the actual query result. It is possible to apply "optimizations" (e.g. batching) that trade throughput for increased latency. Likewise, complexity may not be quadratic for query expressions that allow efficient indexing (e.g. comparisons). But if minimal latency is mandatory, there is no alternative to considering every write operation in the context of every active real-time query. To warrant feasibility, this has to be implemented in a scalable manner.

1.2 Real-Time Database Queries

There are two distinct types of real-time database queries that differ in the way they expose data to the application: **Event stream queries** simply present the raw change events to the application, so that it can maintain an up-to-date copy of the result or apply custom business logic. **Self-maintaining queries** are more abstract as they do the result maintenance in a transparent manner and provide the client with the complete (updated) query result on every change. Using the latter, reactive behavior can be implemented without any notion of data streams or change events built into the business logic. Some static applications can be transformed into real-time applications simply by switching the underlying query mechanism – without even touching application code.

For query interfaces based on callback functions, the transition between static and real-time behavior is particularly smooth. When a query is executed asynchronously, the main thread of execution does not wait for the result to return. Instead, a callback function is provided at query time to specify how the result should be processed. When it is received, the callback function is invoked with the result as an argument.
If the query is executed as a static ad hoc query (see Figure 1.2a), the callback function will only be called once. If the query is executed as a self-maintaining real-time query (see Figure 1.2b), on the other hand, the callback function will be invoked once when the initial



(a) A static ad hoc query produces a single result that represents database state at query time.

(b) A self-maintaining query yields a sequence of results, each reflecting the latest update.

Figure 1.2: While a static ad hoc query is *pull-based*, a self-maintaining (real-time) query *pushes* updates to the client and presents a new result on every change.

result is returned and then again every time when the underlying data has changed; thus, the application effectively behaves as though a new static query was executed whenever it would return a different result than the last invocation. As the only application requirement, every callback function has to be implemented in idempotent fashion, so that an invocation overrides all effects of the previous invocation. For illustration[3], consider a search website where a callback function renders the result of a user-defined query. If the callback function is executed in the context of a self-maintaining query, it has to remove any result representations that were generated earlier or else the website will not reflect result updates correctly.

1.3 Challenges

In concept, real-time databases extend traditional databases as they follow the same semantics, but provide an additional mode of access. In practice, though, there is no established scheme how to actually build a real-time database on top of a traditional database system. Currently existing real-time databases have been built from scratch and consequently do not inherit the rich feature set and stability that some pull-based systems have

[3] For a demonstration of the self-maintaining query implementation developed in the context of this work, visit `https://twoogle.info` (cf. Section 5.3.1).

gained over decades of development. To date, every push-based real-time query mechanism fails in at least one of the following challenges (see Chapter 2 for details):

C_1 Scalability. Serving real-time queries is a resource-intensive process which requires continuous monitoring of all write operations that might possibly affect query results. To sustain more demanding workloads than a single machine could handle, real-time databases typically partition the set of queries across database nodes. As each node is only responsible for a subset of all queries in this scheme, most systems can scale with the number of concurrent queries. However, we are not aware of any real-time database that supports partitioning the write stream as well. Thus, responsibility for individual queries is not shared among nodes and overall system throughput remains bottlenecked by single-machine capacity: Queries simply become intractable as soon as one node is not able to keep up with processing the entire write stream.

C_2 Expressiveness. The majority of real-time query APIs (application programming interfaces) are limited in comparison to their ad hoc counterparts. Aggregations are generally not available and sorted queries are often unsupported or have severe restrictions; for example, there are implementations that only allow ordering by a single attribute or offer a limit, but no offset clause. The lack of such basic functionality on the database side necessitates inefficient workarounds in the application code, even for moderately sophisticated data access patterns.

C_3 Legacy Support. Today's real-time databases have been designed from scratch or on top of NoSQL data stores [Win17b] that do not follow standards regarding data model or query language. They implement custom protocols for pull-based and push-based data access alike and exhibit interfaces that are incompatible among different vendors. While the complete lack of support for legacy interfaces (particularly SQL) may be acceptable in development of a new application, it complicates the adoption of push-based queries for existing applications.

C_4 Abstract API. Many real-time query APIs expose specificities of the underlying implementation and thus offer poor data independence. These interfaces reflect bottom-up design and force developers to reason about problems that lie beyond the application domain. For example, most real-time databases do not provide built-in result maintenance (cf. self-maintaining queries) and thus require knowledge of system internals (e.g. structure of change events) to handle real-time data correctly.

In this thesis, we argue that neither of these limitations is inherent to the challenge of providing push-based real-time queries over database collections. Assuming there is a demand for reactivity on the database side as established at the beginning of this chapter, we thus postulate the following research question:

Research Question: *How can expressive push-based real-time queries be implemented on top of an existing pull-based database in a scalable and generic manner?*

To substantiate our claim and address the above research question, we devise and implement a generic real-time database design that solves challenges C_1 through C_3. We then integrate the prototype implementation into an existing database middleware and, in particular, extend the middleware's existing purely pull-based query API by an abstract interface for push-based real-time queries in order to master Challenge C_4.

1.4 Primary Contributions

We present four primary contributions. In our first contribution, we provide a broad **discussion** of push-based data access mechanisms in modern data management systems. By pointing out where the current state of the art is deficient, we thus further motivate our work towards collection-based real-time queries. Our second contribution is a system **design** that provides push-based real-time queries on top of a purely pull-based database. It is highly scalable (cf. C_1), supports a wide range of query expressions (cf. C_2), and abstracts from the underlying database. Thus, it is compatible with legacy systems that do not support push-based queries (cf. C_3). We demonstrate our design's feasibility and effectiveness through our third contribution: a **prototype** implementation. As our fourth contribution, we prove tractability and usefulness of the overall approach via **integration** of our prototype into an existing pull-based database middleware. The integrated system introduces two client-facing real-time query interfaces: a sophisticated event-based API and a more abstract high-level API using self-maintaining queries (cf. C_4).

1.4.1 A Categorization of Data Management Systems by Access Paradigm

To promote an intuitive understanding of what makes real-time databases unique, we present a classification scheme for data management systems that revolves around their respective support for pull-based and push-based data retrieval. To this end, we compare the different push-based query mechanisms in database systems, real-time databases, and systems for data stream management and processing and highlight the conceptual differences between real-time databases and the other system classes. Further, we inspect the limitations of existing real-time databases and identify the design decisions that cause them.

1.4.2 InvaliDB: A Scalable Approach for Opt-in Real-Time Queries

Next, we use our findings to create a system design for collection-based real-time queries that avoids the bottlenecks present in the current state of the art: We propose *InvaliDB*, a scalable system design that provides push-based real-time queries on top of a pull-based database. InvaliDB sets itself apart from existing system designs through (1) a unique workload distribution scheme for *linear scalability*, (2) support for expressive real-time queries including sorted filter queries, joins, and aggregations, (3) a pluggable query engine to achieve *database independence*, and (4) a *separation of concerns* between the primary storage subsystem and the subsystem for real-time features, effectively decoupling failure domains and enabling independent scaling for both.

1.4.3 An Implementation for Real-Time Queries on Top of MongoDB

To demonstrate InvaliDB's implementation feasibility, we then describe an InvaliDB prototype built with the distributed stream processor Storm [TTS$^+$14] and the in-memory store Redis [San18b] to provide real-time queries on top of the NoSQL database MongoDB [Mon18e]. In order to prove both high performance and linear scalability of our prototype implementation, we then conduct an extensive performance analysis in which we measure response times and sustainable throughput for various workloads and differently sized deployments: Our experiments indicate that InvaliDB's real-time query performance scales *linearly* with the number of matching nodes, both in terms of sustainable write throughput and the number of concurrent real-time queries. Irrespective of the number of nodes, InvaliDB exhibits consistently low latency even under high per-node load.

1.4.4 Integration With the Orestes Database Middleware

To demonstrate the practical applicability of our approach, we detail the integration of our InvaliDB prototype into the Orestes database middleware [GBR14]. In this setup, InvaliDB serves two purposes. First, it provides push-based real-time queries that would be infeasible to provide for the middleware itself: Easy-to-use self-maintaining queries lend themselves to applications that merely need up-to-date views of their critical data, whereas the highly flexible event stream queries cater for more complex access patterns such as user-defined real-time joins or aggregations. Second, InvaliDB enables consistent query result caching through low-latency invalidation messages. Thus, InvaliDB improves latency and throughput of pull-based queries by more than an order of magnitude.

In comparison to existing real-time databases, our integrated architecture separates subsystems for serving OLTP (online transaction processing) workloads and real-time queries. The real-time subsystem is scaled independently from the remaining system components, has its own failure domain, and therefore does not compromise overall availability or fault tolerance. We further show it incurs only minimal overhead on the application server.

1.5 Thesis Outline

The remainder of this dissertation is structured as follows.

In Chapter 2, we discuss related work and provide background on concepts and technologies used in this thesis. First, we categorize different information system classes that account for real-time data in different ways: traditional database management systems, real-time databases, data stream management systems, and general-purpose stream processors. We then examine the available push-based mechanisms in each of these classes. We put an emphasis on two system classes that are in the focal point of this work: In order to identify scaling limitations of today's real-time query implementations, we conduct a system survey of currently existing real-time databases. We further survey the state of the art in general-purpose stream processing technology, because it is a major building block of the system design developed in this work.

In Chapter 3, we present the conceptualization of InvaliDB, a system design that enables push-based real-time queries on top of pull-based databases. We begin with a systematic overview and a brief discussion of the exhibited real-time query semantics and the provided consistency guarantees. Next, we elaborate on the event layer which subsumes the different communication paths to and from the real-time component. In particular, we detail the different kinds of messages that are exchanged between end user devices, application servers, the database, and the real-time component. Further on, we focus on InvaliDB's scheme for workload distribution across a cluster of machines. Specifically, we describe how write stream monitoring and result maintenance are performed in distributed fashion for sorted and unsorted filter queries, aggregations, and joins. We also explore the pluggable query engine and how it lets InvaliDB abstract from specificities such as the database query language or any concrete data format.

In Chapter 4, we describe and evaluate our InvaliDB implementation. We first lay out requirements to be met by the underlying processing engine and discuss reasons for choosing Storm over possible alternatives. We then expand on the Redis-based event layer implementation as well as the MongoDB-compatible query engine. As final and essential part of this chapter, we present experimental results that attest consistent latency in the realm of low milliseconds and high throughput under various workloads in deployments with few and many nodes alike.

In Chapter 5, we elucidate the Quaestor architecture that ties our InvaliDB implementation into the Orestes database middleware. First, we introduce Orestes and its characteristic approach for caching dynamic data, covering data model, access control mechanisms, and query API. The introduction closes with a description of how InvaliDB and Orestes work together in the Quaestor architecture to provide client-facing real-time queries and consistent query caching. We closely examine two different ways of accessing real-time data in the integrated system, namely self-maintaining queries and event stream queries. With several examples, we illustrate the semantics of both real-time query types and show how they can be used to implement custom real-time aggregations and even real-time join queries. Lastly, we quantify overall system performance in the context of query caching and real-time queries as the two primary features that are enabled by InvaliDB.

The final Chapter 6 summarizes the main contributions and limitations, unfolds opportunities for future work, and concludes.

1.6 Previously Published Work

This dissertation contains revised material from previous publications in Chapter 1 ([WGF$^+$17] and [Win17b]), Chapter 2 ([WGW$^+$18], [WGF$^+$17], [Win17b], and [WGFR16]), Chapter 3 ([WGF$^+$17], [GSW$^+$17], and [Win17b]), Chapter 4 ([GSW$^+$17]), and Chapter 5 ([GSW$^+$17]). Publication [WRG18] is being finished at the time of writing and contains material taken from this thesis, mainly from Chapter 1, Chapter 2, and Chapter 6. Further, the InvaliDB prototype described in this thesis has been serving customers at the company Baqend since July 2017. Chapters 3, 4, and 5 as well as Appendix D therefore include revised examples and explanations that were made available in an online usage guide[4,5] before the publication of this thesis.

The following list shows publications made in the context of the presented work:

[WRG18] Wingerath, Wolfram ; Ritter, Norbert ; Gessert, Felix: *Real-Time & Stream Data Management: Push-Based Data in Research & Practice*. Springer, book to be published in late 2018[6]

[WGW$^+$18] Wingerath, Wolfram ; Gessert, Felix ; Witt, Erik ; Friedrich, Steffen ; Ritter, Norbert: Real-Time Data Management for Big Data. In: *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, OpenProceedings.org, 2018

[GSW$^+$17] Gessert, Felix ; Schaarschmidt, Michael ; Wingerath, Wolfram ; Witt, Erik ; Yoneki, Eiko ; Ritter, Norbert: Quaestor: Query Web Caching for Database-as-a-Service Providers. In: *Proceedings of the 43rd International Conference on Very Large Data Bases* (2017)

[WGF$^+$17] Wingerath, Wolfram ; Gessert, Felix ; Friedrich, Steffen ; Witt, Erik ; Ritter, Norbert: The Case For Change Notifications in Pull-Based Databases. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany*, 2017

[GWR17] Gessert, Felix ; Wingerath, Wolfram ; Ritter, Norbert: Scalable Data Management: An In-Depth Tutorial on NoSQL Data Stores. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany* vol. P-266, GI, 2017 (LNI), pp. 399–402

[Win17b] Wingerath, Wolfram: Real-Time Databases Explained: Why Meteor, RethinkDB, Parse and Firebase Don't Scale. In: *Baqend Tech Blog* (2017). `https://medium.com/p/822ff87d2f87`

[4] Guide on Baqend Real-Time Queries: `https://www.baqend.com/guide/topics/realtime/`.

[5] Guide on the Baqend WebSocket API: `https://www.baqend.com/guide/websockets/`.

[6] The book ended up being published in early 2019 (ISBN: 978–3–030–10554–9).

[FWR17] Friedrich, Steffen ; Wingerath, Wolfram ; Ritter, Norbert: Coordinated Omission in NoSQL Database Benchmarking. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany*, 2017

[GWFR16] Gessert, Felix ; Wingerath, Wolfram ; Friedrich, Steffen ; Ritter, Norbert: NoSQL Database Systems: A Survey and Decision Guidance. In: *Computer Science - Research and Development* (2016)

[WGFR16] Wingerath, Wolfram ; Gessert, Felix ; Friedrich, Steffen ; Ritter, Norbert: Real-time stream processing for Big Data. In: *it - Information Technology* 58 (2016), no. 4, 186–194.

[Win16] Wingerath, Wolfram: The Joy of Deploying Apache Storm on Docker Swarm. In: *highscalability.com* (2016). `http://highscalability.com/blog/2016/4/25/the-joy-of-deploying-apache-storm-on-docker-swarm.html`. – Accessed: 2016-05-03

[GSW+15] Gessert, Felix ; Schaarschmidt, Michael ; Wingerath, Wolfram ; Friedrich, Steffen ; Ritter, Norbert: The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management. In: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, 2015, 53–72

[WFGR15] Wingerath, Wolfram ; Friedrich, Steffen ; Gessert, Felix ; Ritter, Norbert: Who Watches the Watchmen? On the Lack of Validation in NoSQL Benchmarking. In: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, 2015, 351–360

[RHL+15] Ritter, Norbert (ed.) ; Henrich, Andreas (ed.) ; Lehner, Wolfgang (ed.) ; Thor, Andreas (ed.) ; Friedrich, Steffen (ed.) ; Wingerath, Wolfram (ed.): *Datenbanksysteme für Business, Technologie und Web (BTW 2015) - Workshopband, 2.-3. März 2015, Hamburg, Germany*. vol. *242*. GI, 2015 (LNI). – ISBN 978–3–88579–636–7

[SRS+15] Seidl, Thomas (ed.) ; Ritter, Norbert (ed.) ; Schöning, Harald (ed.) ; Sattler, Kai-Uwe (ed.) ; Härder, Theo (ed.) ; Friedrich, Steffen (ed.) ; Wingerath, Wolfram (ed.): *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*. vol. *241*. GI, 2015 (LNI). – ISBN 978–3–88579–635–0

[WFR15] Wingerath, Wolfram ; Friedrich, Steffen ; Ritter, Norbert: BTW 2015 – Jubiläum an der Waterkant. In: *Datenbank-Spektrum* 15 (2015), no. 2, 159–162.

[GFW+14] Gessert, Felix ; Friedrich, Steffen ; Wingerath, Wolfram ; Schaarschmidt, Michael ; Ritter, Norbert: Towards a Scalable and Unified REST API for Cloud Data Stores. In: *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, 2014, 723–734

[FWGR14] Friedrich, Steffen ; Wingerath, Wolfram ; Gessert, Felix ; Ritter, Norbert: NoSQL OLTP Benchmarking: A Survey. In: *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, 2014, 693–704

2 Background & Related Work

> "We are stuck with technology when what
> we really want is just stuff that works."
>
> —Douglas Adams

In the first chapter, we introduced real-time databases as systems that take the middle ground between traditional database systems and data stream management systems. In this chapter, we further explore why real-time databases deserve distinction in a separate system class. To this end, we compare push-based real-time queries against the push-based mechanisms for data access available in other information systems. In doing so, we uncover a mismatch between functionality desired by application developers on the one side and features actually provided by state-of-the-art systems on the other.

In Section 2.1, we delineate real-time databases from traditional database management systems, data stream management systems, and general-purpose stream processing engines. We do this by considering their individual notions of data and their respectively facilitated means of data retrieval. In the following sections, we examine the different mechanisms for push-based data access available in each of these system classes, starting with traditional (relational) database systems in Section 2.2. Next, we dissect the real-time query implementations of the most prominent real-time databases in Section 2.3. Thus, we uncover both deficiencies in the individual implementations and scale-prohibitive limitations in their respective designs. Our findings provide valuable insights into what design decisions should be avoided in a scalable real-time database architecture (cf. Chapter 3). In Section 2.4, we then focus on data stream management systems which do not promote collection-based semantics, but instead view data as potentially infinite streams of information. Subsequently in Section 2.5, we survey general-purpose stream processing technology and analyze different systems with respect to their capabilities. In Section 2.6, we finally sum up our findings and conclude that all current push-based query implementations are either based on streaming semantics, are severely limited in their expressiveness, are incompatible with legacy systems, or do not scale beyond trivial workloads.

2.1 Systems for Pull-Based & Push-Based Data Access

In the first chapter, we highlighted the individual characteristics of pull-based and push-based data access: A *pull-based* query assembles data from a bounded data repository and completes by returning data once, whereas a *push-based* query processes a conceptually unbounded stream of information to generate incremental output over time. Given these fundamental differences, the design of any data management system reflects a bias towards one or the other; for example, while databases do support push-based access to a certain degree (e.g. through triggers), they are clearly geared towards efficiency for pull-based data retrieval.

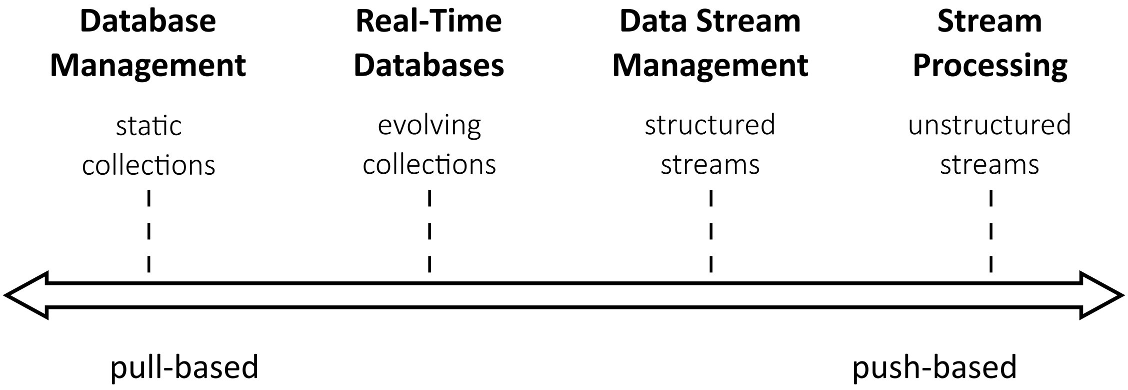


Figure 2.1: Different classes of data management systems and the access patterns they support.

We argue that systems for data management can be classified by the way they facilitate access to data as illustrated in Figure 2.1. At the one extreme, there are **traditional databases**[1] which represent snapshots of domain knowledge that are the basis of all queries. At the other extreme, there are general-purpose **stream processing** engines which are designed to generate output from conceptually unbounded and arbitrarily structured ephemeral data streams. Real-time databases and data stream management systems both stand in the middle, but adhere to different semantics: **Real-time databases** work on evolving collections that are distinguished from their static counterparts (i.e. from database collections) through continuous integration of updates over time, enabling push-based real-time queries. **Data stream management** systems provide APIs to query data streams, for example, by applying filters to incoming data or by computing rolling aggregations and joins over configurable time windows.

[1] While we focus on relational databases in our discussion, a preference for pull-based over push-based access is also evident in graph databases [JPNR17], object databases [Wie15, Ch. 9], and other databases with non-relational data models [OM10, Ch. 5, Sec. I].

2.1.1 Collections vs. Streams

While a database collection represents the current *state* of the application domain, a data stream rather encapsulates recent *change*. For an illustration of the difference between the semantics of collections and streams, consider the example data given in Table 2.1 that shows two different representations of an application for user account management. The **stream**-based representation (a) provides a sequential view on all user actions, but does not retain them indefinitely: Events are available for a certain time window (framed records), but are discarded eventually (lightly shaded records). This view on the data promotes use cases that require notifications, e.g. for individual users logging in or out. However, the system only operates on a suffix and not the entirety of event history. Therefore, queries do not reflect actions that happened long ago: For example, it is not possible to produce a list of all registered users sorted by name or by date of first login, because relevant data is beyond the query's time horizon. In order to serve historical data, the ephemeral events have to be applied to a persistent representation of application state. A database **collection** (b) reflects all data ever written and thus enables queries such as the above-mentioned one. Since collection-based ad hoc queries do not capture events that arrive after the query, though, traditional databases do not propagate informational updates to the client.

Timestamp	Name	Action
2017-05-05 07h49	Jane	login
2017-05-05 08h52	Jane	logout
2017-05-06 08h08	Jill	login
2017-05-06 09h32	Ken	login
2017-05-06 09h47	Jill	logout
2017-05-06 10h11	Bob	login
⋮	⋮	⋮

(a) A data stream primarily captures changes in application state.

Name	First Login	Last Login	Logged In
Bob	2017-01-15	2017-05-06	true
Erk	2017-01-26	2017-01-26	false
Jane	2017-02-12	2017-05-05	false
Jill	2016-08-02	2017-05-06	true
Joy	2017-03-09	2017-04-24	false
Ken	2017-05-05	2017-05-06	false
Lee	2016-03-01	2016-04-17	false
Tim	2017-02-23	2017-05-05	false

(b) A database collection provides access to the current state of the application.

Table 2.1: Streams and collections promote different perspectives on data.

User account management is just one example of an application domain that requires some form of permanent data storage to answer queries regarding the current state of the world. Given a database's limitation to mainly pull-based access, though, real-time user interfaces are hard to build: One possibility is to reevaluate a given collection-based query from time to time which is inefficient and introduces staleness in the order of the refresh interval (cf. poll-and-diff in Section 2.3.1). Another approach is to merge results from collection-based and stream-based queries; thus, the application is effectively burdened with the task of view maintenance which is complex an error-prone (cf. Section 2.2.3).

Real-time databases aim to close the gap between both paradigms by providing collection-based semantics for pull-based and push-based queries alike.

In the rest of this chapter, we provide an overview over push-based query mechanisms in data management systems available today. In doing so, we show conclusively that InvaliDB is the only generic system design that combines the push-based access paradigm with expressive collection-based queries at scale.

2.2 Database Management

The first databases were **hierarchical and network databases** [TL76], developed during the 1960s. They exposed procedural query interfaces (as opposed to descriptive ones), so that accessing specific information in one of these systems was similar to navigating to a specific file within a file system [FS76]. Early query languages were severely limited in expressiveness and relied on high-level programming languages for scanning and search [Cod71, Sec. 1.2]. Similarly, consistency checks were mostly enforced within client applications and revolved around conventions and best practices [Oll06]. In consequence, data integrity was difficult to maintain and reorganizing or scaling a database could be disruptive for existing client applications. During the 1970s, standards in database management (especially regarding query languages) [FS76] and data independence received more attention within the database community [Cod71, Sec. 2]. The proposition of the **relational model** [Cod70] then eventually led to a descriptive query language that evolved into the *Structured Query Language (SQL)* [CB74].

2.2.1 Triggers & Active Databases

Relational databases were initially designed as *passive* repositories that accept, modify, or retrieve structured data as a direct response to an explicit request [Cod82]. Acknowledging the need to model application behavior in addition to the structural aspects of a domain, database **triggers** were the first *active* mechanisms to be proposed for relational database systems [EC75] that became part of the SQL standard [CPM96]. Essentially, triggers are procedures that are implicitly invoked on database events such as insert, update, or delete operations or on system events like errors or user logins [Ora16, Sec. 3.2]. As such, they are primarily used as a means to enforce integrity or to propagate writes on specific entities to depending entities [Sto86]. In an effort to facilitate more sophisticated behavioral semantics, **event-condition-action (ECA) rules** [SKM92] were introduced to database systems during the 1980s and 1990s. ECA rules capture more complex *events* (cf. Section 2.4.4), for example with temporal components (e.g. a fixed date or a time period) or compositions (e.g. disjunctions or sequences of specific occurrences) [Cha95]. Further, execution of an action is not only tied to the occurrence of a specified event, but typically also depends on fulfillment of a corresponding *condition*, for example a predicate over database state or the triggering event's net effect (i.e. the difference between state before and after the event) [HW93, Sec. 4.1].

The usefulness of **active databases** [Mor83, PD99], i.e. database systems with advanced active features, is often illustrated with applications similar to those that motivate our work: Typical examples are materialized view maintenance [SJGP90] and real-time user interface (UI) updates [DJPQ94]. Common to all centralized active database implementations, however, is that database-internal active mechanisms quickly become performance bottlenecks [SD95]. Therefore, some approaches restrict semantics to avoid infeasible scenarios; for example, Alert [SPAM91] supports active rules exclusively over append-only tables[2]. Comparable sacrifices are made by **real-time (active) databases** [PSS$^+$93, RSS$^+$96]. The underlying notion of the term "real-time" in this context refers to compliance with time constraints, whereas the databases in the focal point of this thesis are "real-time" in the sense that they detect and propagate updated information with low latency (see term disambiguation in Section 1.1). In order to deliver predictable execution times, real-time (active) databases relax consistency guarantees, reduce concurrency, or restrict query and rule expressiveness [Eri98]. To this day, active database mechanisms remain prohibitive for scalability and thus even modern database systems are documented to display poor latency and throughput when active features are used at scale (see for example [BMH$^+$16, p. 13]). Implementing active facilities *on top of* an existing database system (as opposed to implementing them as an internal component) is usually even less efficient [GGD95] [SKM92, Sec. 4.1] and significantly more difficult, as the information required for event detection is often only available within the database [PD99, Sec. 7.1].

2.2.2 Change Data Capture, Cache Coherence & Time Series Data

Given the limitations of active mechanisms within database systems, more generic approaches have been sought to make informational updates available outside of the database system. Systems for **change data capture (CDC)** [Kle16, Ch. 11] extract data from the primary storage system and propagate it to other systems, e.g. for replication purposes, invalidation of cached views, or for custom data processing pipelines. Some systems use *trigger-based replication*, i.e. they employ active mechanisms to persist updates to auxiliary tables which, in turn, are then polled periodically by downstream systems. For instance, Databus extracts data through this pattern from Oracle databases [DBS$^+$12, Sec. 4.1] and Bucardo [Mul14, Mul11] extracts data from PostgreSQL through triggers and notification listeners (cf. page 23). However, the auxiliary tables dedicated to storing change information can become write bottlenecks and using triggers for data replication is rather error-prone in itself because of the complexity involved in replicating multiple tables or ensuring transactional visibility of updates [DBS$^+$12]. To avoid these difficulties,

[2] Ruling out the possibility of updated or deleted records makes detection of new matches significantly more efficient: In terms of the example in Section 1.1, only *add* events can occur, whereas *change* and *remove* events are impossible.

some systems attach to the database using lower-level database protocols: As one example, Databus hooks into MySQL's storage engine API to obtain a change log [DBS+12, Sec. 4.2]. There are also various products accompanying commercial databases that implement inter-database replication using proprietary protocols (e.g. Oracle's Active Data Guard [Ora15a]) or provide database change logs for external applications (e.g. Oracle's GoldenGate [Ora15b] or IBM's InfoSphere CDC [IBM11]). Even though some systems provide simple mechanisms to filter extracted data by a user-defined predicate (e.g. Databus [DBS+12, Sec. 4.3]), more complex transformations or processing is usually performed in dedicated (stream processing) systems such as the ones discussed in Section 2.4 and Section 2.5. Our real-time database design InvaliDB requires a mechanism for change data capture to make database writes available for query matching; our prototype implementation discussed in Chapter 4 employs a custom capturing mechanism, though.

Change data capture can be used to feed **distributed caches** or **memory grids** which offload read and write workload from the main database system or perform processing tasks such as data transformation. For example, systems like Oracle Cache Coherence [BMH+16], Hazelcast [L+17], or Ignite [Gri17] are deployed as server clusters which act as write-through or write-behind caches between client applications and the primary data storage. The cached entities are stored in a derived format (e.g. Java objects) and relate to the application data model rather than the underlying database schema. Use of filter queries is discouraged, because they are processed node-locally and thus require heavyweight scatter-gather patterns for result assembly (see for example Hazelcast [L+17, Sec. 5.14] or Oracle Cache Coherence [BMH+16, p. 5]). Even though these systems provide limited support for SQL or SQL-like query languages, data is usually accessed by primary key and in a programmatic fashion, i.e. through program code. Under the term **continuous queries** [Haz17] [R+11, Ch. 23] [WBL+07], some systems offer push-based filter queries with stream-based semantics[3] or even collection-based static filter predicates and result ordering (e.g. Oracle Cache Coherence [R+14a, R+14b]). Thus, these systems provide real-time database functionality like the systems in the focus of this thesis. However, we are not aware of any continuous query implementation that has a pluggable query engine or supports joins.

Time series databases [DF14] (e.g. OpenTSDB [SDSB18] or DalmatinerDB [Pro18]) are specialized in storing and querying conceptually infinite sequences of events as a function of their time of occurrence, for example sensor data indexed by time. While some of them (e.g. InfluxDB [Inf16]) are capable of rolling averages or other continuous aggregations [BD91], they are typically employed for analytic queries, or downsampling streams of information; their capabilities do not extend to change notifications akin to the real-time queries focused in this thesis.

[3] When a matching record leaves a query result, Ignite does not notify listening clients [Ign17].

2.2.3 Materialized Views

Since results of complex queries cannot be maintained up-to-date efficiently by the database-external systems discussed above, sophisticated mechanisms for database-internal query result caching have been developed. The idea behind **materialized views** [BC79] [CY12] is to precompute the result of particularly expensive queries, so that they do not have to be evaluated repeatedly, but can be served immediately when requested. **Logical views**, in contrast, are rewritten to queries that have to be evaluated on every request. Thus, materialized views are significantly faster to access than logical views. However, this read time performance advantage comes at a hefty cost: In order to guarantee freshness of the cached query result, changes[4] have to be detected and applied at write time. This challenge is equivalent to the challenge of providing real-time queries as described in Section 1.1.

View maintenance algorithms can be classified according to whether they assume full or only partial access to the underlying database tables [GM99]. **Recomputation** of a query result is conceptually straightforward, applicable to arbitrarily complex queries, and requires unrestricted access to the base data. While queries with relatively stable results can theoretically be maintained fresh and consistent by just recomputing the result after every invalidating write operation, it is difficult to distinguish invalidating writes from those that can be safely ignored: A host of literature is dedicated to recognizing updates that change the query result (*relevant updates*) [BLT86] [BCL89] [Elk90] [LS93], so that recomputation can be avoided unless required for consistency. In order to address queries that evolve more quickly or for which the base data is not accessible, **incremental view maintenance** [Vis96] avoids recomputation altogether by detecting and applying changes directly to the materialized result. Even though incremental maintenance of query results has been studied for other data models as well (e.g. object databases [Nak01]), the majority of literature refers to relational databases [CY12, GM99]. To decouple the maintenance process from access to the base data, **self-maintainability** [GJSM96] has received particular attention in the context of materialized views: It postulates that a view can be kept up-to-date using only the view contents and the incoming modifications (i.e. the database writes). In practice, many queries are not self-maintainable per se, but only with respect to specific modifications at runtime or with the help of auxiliary data [QGMW96]. For example, sorted queries are not inherently self-maintainable with respect to deletes and updates, because removing one item from the result may cause an unrelated item to enter the result from beyond limit; it is possible, however, to make a top-k query self-maintainable with respect to *a certain number* of updates or deletes by initially requesting a top-k' result where $k' > k$ [YYY$^+$03]. Thus, there is often a trade-off between the useful-

[4] In the context of view maintenance in relational (SQL) databases, it is not only important to update a materialized view whenever records in the corresponding base tables are written, but also when the view definition is altered or when one of the base table's schema is modified. We do not go into detail here in order to keep the focus on result maintenance and refer to [CY12, Sec. 2.5.1] and [NLR98] for more information.

ness of additional auxiliary data during maintenance and the costs of initially assembling it. Taking the middle ground between recomputation and incremental maintenance, some algorithms only materialize and incrementally maintain auxiliary data (e.g. a subquery result or a join index) instead of the actual result [BM90]. This kind of **partial maintenance** of a materialized view still necessitates reevaluating the query to refresh, but makes this process very cheap. At the same time, the maintained auxiliary data can be used by the database optimizer to accelerate other queries as well [Vis98]. A materialized view can thus be understood as a data structure that speeds up database reads, similar to a *database index*. Since both recomputation and incremental maintenance of a query result can be very resource-intensive, maintaining very complex or a great number of materialized views is sometimes performed under relaxed consistency guarantees to allow distributing or deferring the maintenance process [ASC+09]: When stale data is tolerable, incremental maintenance can be performed through asynchronous and throughput-optimized batch updates [SBLC00]; likewise, recomputation can be deferred to save resources. When tables are spread or sharded across different physical nodes, materialized views are typically maintained in deferred fashion, because the overhead for transferring data between different sites is significant and sometimes prohibitive for immediate change propagation [LYC+00]. Maintenance of **materialized views in distributed environments** has been researched for decades, specifically in the context of data warehousing where queries tend to be particularly expensive to compute [CBHB09] [BKS00] [SP89]. Instead of maintaining individual queries or indices, entire database partitions are sometimes mirrored to remote sites in these scenarios, so that queries can be executed locally (i.e. at the replica) that would otherwise have to be executed remotely (i.e. at the primary). In these distributed setups, the transitions between indexing, materialized view maintenance, and database replication can therefore be fluent.

While the algorithms employed for materialized view maintenance are conceptually identical to the ones applied in real-time databases, materialized views are used exclusively to improve pull-based query performance. Building push-based mechanisms on top using triggers is possible [LG+03, Sec. 16–100], but at least as complex and error-prone as an implementation on top of common database tables. Further, materialized views in monolithic databases are heavyweight processes that have to be carefully planned in accordance with overall system resources [DA09]. The real-time queries addressed by this thesis, in contrast, are executed by arbitrary end users; a real-time database must be able to support thousands of concurrent real-time queries reliably and at all times (e.g. under peak throughput). Nonetheless, the concepts behind maintaining a materialized view also apply to maintenance of real-time queries; in particular, the real-time database design described in Chapter 3 adheres to the same principles as algorithms for incremental maintenance in relational database systems. In the context of this work, specifically, we use existing approaches for top-k query maintenance [YYY+03] to enable sorted real-time queries (see Section 3.3.2). We thus only consider queries that are self-maintainable at runtime without accessing the base data. Further, we avoid irrelevant updates by identifying obvious mismatches as early and cheaply as possible (cf. [Elk90]).

2.2.4 Change Notifications

Using mechanisms related to change detection in materialized views [Mic17a], some relational database systems have the ability to send notifications when previously requested data has changed or might have changed. Most implementations exhibit occasional false positives, i.e. change notifications may be sent without an actual data change occurring [Pos17][Mic17d][MK$^+$17]. In contrast to **continuous query subscriptions** in stream management systems or real-time databases, change notification messages in relational database systems *do not carry the changed data itself*, but only identifiers (e.g. row IDs, table names or query identifiers) and some information on what happened (e.g. the type of operation or a text message). Therefore, supposedly changed data items or queries have to be requested again after a notification has been received in order to make sure the local copy is still up-to-date [Pos17, Mic17c][MK$^+$17, Sec. 15.5].

Like materialized views, change notifications serve the overall purpose of improving ad hoc query performance in domains *where updates are infrequent* [Mic17b][MK$^+$17, Sec. 15.5]. As typical use cases, vendors describe three-tier applications where data is cached in the middle tier and *only few queries* have to be monitored for changes [MK$^+$17, Sec. 15.7.7]. When receiving user requests, the middle-tier application servers respond on the basis of their local copies of the data which they refresh asynchronously when receiving a notification [Mic17e] [MK$^+$17, Sec. 15.5]. To receive change notifications, application servers can subscribe to specific data items or queries, thus launching a process at the database that monitors ongoing operations and detects relevant changes. Since monitoring a query for changes incurs additional work on write operations, more than a few concurrently active registered queries may impair OLTP throughput [Mic17b][MK$^+$17, Sec. 15.7.7]. For scenarios with high throughput or many unique queries, some vendors explicitly discourage using their notification feature and instead recommend employing workarounds, for example using triggers or implementing sophisticated middleware for change detection [Mic17b]. Several variants of this notification mechanism exist and the individual implementations exhibit partly substantial differences.

Oracle 12c offers two different change notification mechanisms: Object[5] Change Notifications and Query Result Change Notifications. When a query is registered for *Object Change Notifications (OCNs)*, a notification will be sent for every write operation in one of the query's underlying tables [MK$^+$17, Example 15–1]. Since only the target collection of a write operation (and specifically not the query predicate) is considered, the monitoring process associated with a single OCN has a low performance footprint. In consequence, though, the generated change notifications may or may not bear any relevance to the

[5] In literature and this thesis, the terms "object" and "record" both refer to an entity representation.

registered query. For example, when registering a highly selective query on a frequently updated table, the majority of change notifications might be false positives, because the updated records do not relate to the query result most of the time. In such a scenario, the receiver of an OCN therefore has to weigh potential staleness (through disregarding a notification and thus not refreshing a query result that actually has changed) against potential inefficiency (through refreshing a query result that has actually not changed).
For applications that cannot easily tolerate the high chance of false positives that comes with OCNs, the more concise *Query Result Change Notifications (QRCNs)* can be used. Depending on the concrete query and the mode of operation, QRCNs can reduce or even eliminate the risk of receiving false positives. In comparison to OCNs, however, they are also more expensive to provide, because they necessitate (1) deriving[6] before-images of relevant data items and (2) evaluating the query with respect to both *before- and after-images*[7] of relevant data items [WBL$^+$07]. There are two modes of operation for QRCNs: In *Guaranteed Mode*, QRCNs are sent when and only when the registered query's result was altered; in other words, there are no false positives in Guaranteed Mode. This mode of operation is the default. In *Best-Effort Mode*, false positives are possible, because the system simplifies the monitoring process by two aspects: First, a simplified version of the registered query is monitored instead of the registered query itself. Thus, a change of the registered query always entails a notification, but a notification may also be sent for an update that only affects the simplified query, but not the originally registered query. As a second simplification, notifications in Best-Effort Mode are generated considering individual operations and not transactional effects. Thus, a notification may be generated in response to an individual operation, even though it would never have become visible[8]. There are a range of queries that are not eligible for Guaranteed Mode, but only for Best-Effort Mode; for example, a query cannot be registered in Guaranteed Mode when it contains an `EXISTS` condition, an aggregation (e.g. `MIN`, `MAX`, or `SUM`), an arithmetic function (e.g. `ABS` or `SQRT`), a string function (e.g. `SUBSTR`), a pattern matching condition (e.g. `LIKE`), or when it contains an `ORDER BY` or `GROUP BY` statement or uses any join other than inner equi-join [VSGC10, 0025] [MK$^+$17, Sec. 15.7.5.2]. Some queries are not supported for QRCNs in either mode of operation, for example queries that include a `COUNT` aggregation or queries that reference views or columns of types other than `NUMBER` or `VARCHAR2` [MK$^+$17, Sections 15.7.5.1 and 15.7.5.3]. It is important to note that eligibility for either mode of operation does not depend on the query specified by the user, but the

[6] According to [WBL$^+$07, Sec. 4], changes are retrieved from querying transaction undo/redo logs which is infeasible unless done in large batches: In an experimental evaluation, performance increased by a factor of 35 when switching from near-realtime change computation (after every 10 transactions) to batched change computation (after every 2 000 transactions) [WBL$^+$07, Sec. 10].

[7] In the context of a write operation, we use the terms *before-image* and *after-image* to denote the fully specified representations of a written entity immediately before and after the write operation has been applied, respectively.

[8] As an example, consider a transaction that increases an employee's salary by some amount and later decreases it by the same amount. In Best-Effort Mode, two notifications might be generated, even though the transaction effectively does not change any data and thus does not warrant any notification at all.

query that is actually executed. Thus, registering an originally supported query for QRCN may still fail, for example when the query optimizer rewrites the query to use a materialized view [MK$^+$17, Sec. 15.7.5.3]. Change notifications are among the "features that are not available or are restricted for a multi-tenant container database" in Oracle databases [Ora17b, Sec. 2.2.1].

Microsoft SQL Server provides change notifications roughly comparable to Oracle's QR-CNs in Best-Effort Mode: In SQL Server, false positives cannot be excluded for any registered query [Mic17c]. The range of supported queries appears somewhat less wide, because SQL Server denies notifications for unsupported query types altogether; in contrast to Oracle databases, Microsoft SQL Server does not simplify unsupported query types to enable query change notifications. Change notifications in Microsoft SQL Server are not available for queries that reference a view, use subqueries, outer joins, or self-joins, include the `UNION` operator, the `DISTINCT` keyword, certain aggregates like `AVG`, `MAX`, and `MIN`, or use `*` as column identifier [Mic17a]. As another distinction to Oracle's notification mechanism, a subscription in Microsoft SQL Server is canceled immediately on notification, so that ongoing result maintenance requires creating a new subscription and thus reexecuting the query in question whenever a notification has been received [HG17].

PostgreSQL provides only very basic support for notifications with an expressiveness that is roughly comparable to Oracles Object Change Notifications. Change notifications in PostgreSQL are not generated for specific queries, but for changes on particular tables: Notification messages essentially tell the user that a write operation has occurred on the table that was specified on subscription. To distinguish different kinds of change, varying payload strings can be included in the notification messages [Pos17].

In summary, change notifications neither provide real-time queries out-of-the-box nor can they be used to implement client-facing real-time queries in an efficient way, because the only way to do so involves recomputing the query result on every change notification. Further, using change notifications quickly becomes prohibitive, because they impair write throughput, incur additional database round-trips for fetching invalidated results, and (depending on the concrete system) sometimes even demand reestablishing a query subscription after each change notification.

2.2.5 Database Systems: Summary & Discussion

Traditional databases offer limited capabilities to push information to clients. Triggers, ECA rules, and change notifications cannot be used for proactive data delivery, unless brittle workarounds are employed. A few systems even provide continuous query capabilities that resemble our notion of real-time queries, but these systems are limited in their expressiveness and bound to specific query languages. While materialized views employ mechanisms that are suitable for providing real-time queries, they are exclusively used to increase pull-based query performance.

2.3 Real-Time Databases

While traditional databases are targeted at providing a consistent snapshot of the application domain, real-time databases acknowledge that data may evolve. Both the architectures and client APIs of real-time databases reflect that facts can change over time and that the system may have to enhance or correct issued information. Real-time databases may allow snapshot (one-time) queries over database collections or they may provide interfaces to directly access the stream of update operations. But their defining property is that queries are formulated as though they were evaluated on static data collections, even when their response is a continuous stream of updates to the query result.

In this section, we scrutinize the current state of the art in real-time databases with respect to how the individual systems provide real-time queries. In doing so, we uncover scaling limitations inherent to their respective designs. This section contains revised material from [Win17b].

2.3.1 Meteor

Meteor [Met18] is a JavaScript app development framework that targets reactive apps and websites. It uses MongoDB as its internal data storage and therefore inherits its query expressiveness while adding self-maintaining and event stream queries on top. Interestingly, Meteor offers two different implementations to detect relevant changes to a query result: The original approach (*change monitoring + poll-and-diff*) combines monitoring local write operations within the application server and periodic query reevaluation; it is only used as fallback nowadays. The more recent (and current default) implementation (*oplog tailing*) relies on monitoring MongoDB's replication log.

Change Monitoring + Poll-and-Diff. As illustrated in Figure 2.2, Meteor's original approach towards real-time queries combines two mechanisms that detect changes received by the server itself (blue) and those received by another server (red), respectively: First, a Meteor application server performs local **change monitoring** to discover state changes that are relevant for currently active real-time queries. Thus, a newly inserted record (**1**) is not only forwarded to the database (**2**), but is also translated to a corresponding query change delta for every matching real-time query (**3**). However, change monitoring alone is not sufficient in multi-server deployments: Server-local monitoring done within the server on the left (A), for example, will not capture write operations received by the server on the right (B). To compensate for this fact, Meteor employs a second strategy called **poll-and-diff** which essentially reevaluates a real-time query periodically ("poll"), computes relative changes to the locally maintained result ("diff"), and then sends them to the client (**4**). When these two strategies are combined, the client will receive a relevant update either immediately through change monitoring (red) or after a short delay upon discovery through poll-and-diff (blue).

From the client perspective, poll-and-diff has the obvious disadvantage of potential staleness windows bounded by the polling interval (default: 10 seconds). With unfortunate timing, result updates can also go completely unnoticed; for example, an item entering and leaving the result between query polls will not be registered at all. But even if an application can tolerate multi-second lags and missed notifications, poll-and-diff becomes infeasible when many real-time queries are active concurrently. This is because each one of them induces processing overhead on the database system through periodic polling. In numbers, 1 000 query subscriptions result in an average of 100 queries per second whose results have to be (1) assembled and (2) serialized, then (3) be sent to the application server where they are (4) deserialized again, so that they can finally (5) be analyzed for relevant changes. While this may be tractable in some cases, it quickly gets prohibitively expensive when results are large. Further, it should be pointed out that this is just what happens in an otherwise idle system, i.e. when no data is being written whatsoever. The situation gets worse when write throughput increases, because the database has less spare resources to serve periodic poll queries and because each application server has to spend more CPU time on change monitoring – which becomes more demanding with an increasing number of active real-time queries as well. It should be noted that poll-and-diff works for arbitrary queries, since the pull-based query mechanisms of the underlying database are used. At the same time, though, it is important to understand that real-time queries that rely exclusively on poll-and-diff (i.e. queries that are not supported by change monitoring) implicitly expose latency in the order of the polling interval.

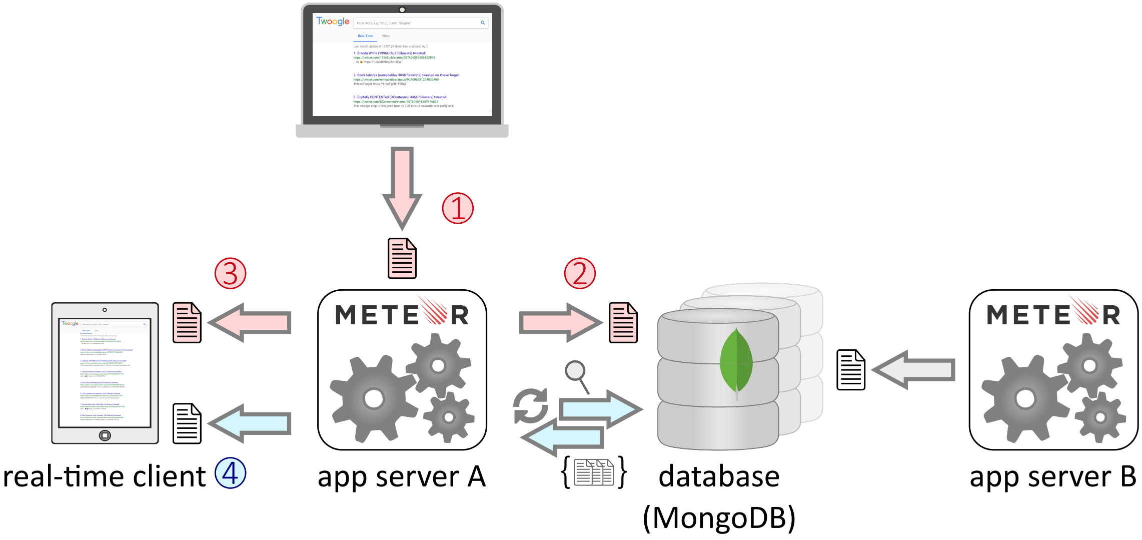


Figure 2.2: Poll-and-diff: Meteor executes a query again and again to discover changes.

Oplog Tailing. Seeing the considerable downsides of poll-and-diff, the Meteor developers came up with an alternative approach that has a different set of trade-offs. As described above, a Meteor application server is able to maintain a query result by itself, given it receives all relevant write operations. However, since a write operation is only visible to the application server that receives it, multi-server deployments require a mechanism that informs all servers about write operations received by the others. The poll-and-diff approach accomplishes this, but at the same time introduces high base load by executing the same query over and over. Acknowledging this problem, *oplog tailing* was introduced as an alternative solution that uses MongoDB's replication protocol to feed the full write stream to every application server.

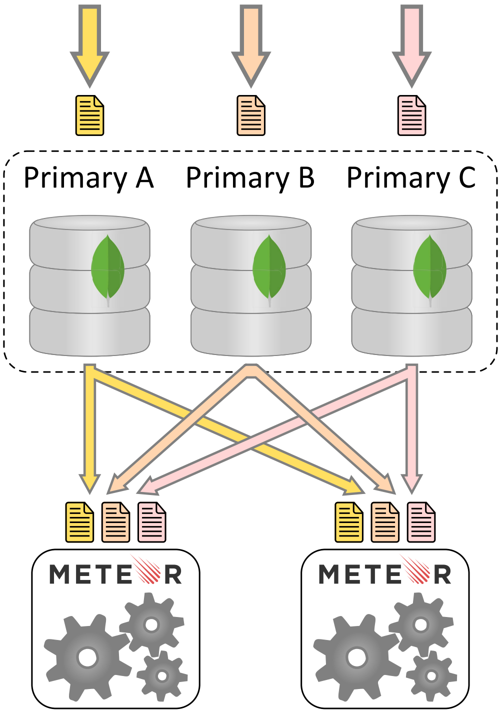


Figure 2.3: With oplog tailing, each Meteor application server receives all MongoDB writes: Thus, OLTP workload is sharded, but real-time workload is not.

MongoDB achieves write scalability by distributing data across different partitions (*shards*). In a production deployment, data within a partition is kept redundantly in a so-called **replica set**: Write operations within a partition are first applied by the *primary* and then delivered to the *secondaries*. Internally, the primary logs all writes in a *capped collection* – a ring buffer called the **oplog** – and the secondaries are following along using *tailable cursors* [Mon17b]. The basic idea of oplog tailing [Col16] is to have the Meteor application servers hook into MongoDB replication as though they were secondaries. As illustrated in Figure 2.3, each application server taps into each primary's oplog and thus will never miss a single update operation. This setup makes periodic polling obsolete and eliminates the staleness inherent to poll-and-diff. But in doing so, it introduces the application server as a bottleneck for write throughput. Because this is where Meteor deviates from the way that MongoDB uses the oplog: While each MongoDB secondary has to keep up with only one primary, each Meteor server has to keep up with the combined throughput of the *entire MongoDB cluster*. As a consequence, MongoDB can scale with write throughput, but Meteor cannot.

This is not just a theoretical limitation, but a critical problem for production deployments: It is a known issue [W+14] that load spikes can saturate and take down a Meteor application server when oplog tailing is enabled [M+14]. To address this particular issue, poll-and-diff is used as a fallback strategy [Met15] for oplog tailing whenever an application server falls behind in monitoring the change log. But this is obviously associated with the cost of potential staleness and becomes infeasible as well when there are more than a few active real-time queries [K+15]. To leverage maximum performance, it is therefore *officially recommended* to carefully evaluate – on a per-query basis – whether to enable or disable oplog tailing [Met16].

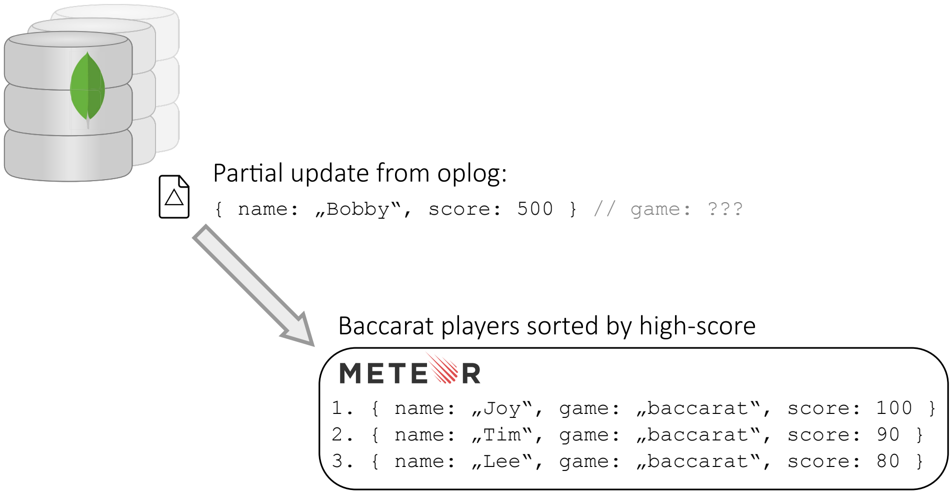

Partial update from oplog:
{ name: „Bobby", score: 500 } // game: ???
Baccarat players sorted by high-score
METEOR
1. { name: „Joy", game: „baccarat", score: 100 }
2. { name: „Tim", game: „baccarat", score: 90 }
3. { name: „Lee", game: „baccarat", score: 80 }

Figure 2.4: Information delivered through the oplog can be insufficient to decide whether or not an update has an effect on a given query's result.

As a side note, oplog tailing does not eliminate the need to contact the database completely. For an example, consider a real-time query for the top-3 baccarat players as depicted in Figure 2.4. The maintaining server receives a partial update to a previously unknown player: According to the received update, Bobby has a higher score than any baccarat player. However, it is unknown whether Bobby is a baccarat player to begin with, because the oplog does not tell: It only contains information on Bobby's new score (`500`), but not the associated game. In consequence, the Meteor application server has to query MongoDB for Bobby's associated `game` in order to determine whether the query is affected by Bobby's new high score.

In summary, Meteor offers two implementations of real-time queries: Poll-and-diff facilitates expressive real-time queries, but can be laggy and only works when few real-time queries are active. Oplog tailing is viable in the presence of many concurrent real-time queries, but is only feasible for low write throughput, because it effectively circumvents the sharding mechanism of the underlying database. Neither approach works for many users and high throughput.

2.3.2 RethinkDB

RethinkDB [Ret16] is a JSON document store with query expressiveness comparable to MongoDB's. However, RethinkDB is a completely independent project that seems more ambitious than MongoDB in some areas, offering advanced features like pull-based θ-join queries [Mar16]. Among the more interesting specialties of RethinkDB are *changefeeds*: real-time queries with an event stream query interface.

Changefeeds. RethinkDB's technique of change discovery is very similar to Meteor's oplog tailing: Clients do not communicate with the database nodes directly, but instead with the application server [Mew16]; each application server runs an instance of the **RethinkDB proxy**, a process that relays communication between clients and the RethinkDB cluster. A client registers a real-time query at a RethinkDB proxy and then receives the initial result and a stream of change deltas: the *changefeed*. The RethinkDB proxy, in turn, queries the database once on subscription and subsequently monitors all write operations from the RethinkDB cluster to maintain the real-time query result.

Just like Meteor with oplog tailing, RethinkDB nullifies all benefits of database-level sharding by burdening individual application servers with monitoring the complete cluster write stream. Due to this write bottleneck, RethinkDB is subject to the same performance limitations as oplog tailing: Changefeeds cannot scale beyond the capacity of a single application server and will saturate application server resources under pressure. In contrast to Meteor, RethinkDB does not rely on external technical artifacts such as the oplog, though. Thus, in particular, internal change propagation is more elegant and does not require additional round-trips to fill informational gaps, as is the case with oplog tailing (cf. Figure 2.4). But even though RethinkDB executes the concept behind oplog tailing in cleaner fashion than Meteor, the concept remains inherently unscalable.
There are few reports of RethinkDB users and their experiences with the scalability of changefeeds. However, CoCalc (formerly known as SageMathCloud) [CoC18] as one of the most enthusiastic RethinkDB users reverted from RethinkDB to PostgreSQL for a reactive web application, because a makeshift solution based on PostgreSQL's change notifications (cf. page 23) outperformed RethinkDB changefeeds by an order of magnitude [Ste17].

In brief, RethinkDB offers a real-time query implementation similar to Meteor's oplog tailing: Write throughput for RethinkDB's real-time queries does not scale as it is bottlenecked by single-node capacity. Unlike Meteor, RethinkDB does not have a poll-and-diff-like approach to utilize the pull-based query API for real-time queries. Thus, the pull-based interface is more expressive than the push-based one. For example, changefeeds for sorted queries support a limit, but no offset [Mar15]. Further, the client API has *no self-maintaining queries*. In order to keep a query result up-to-date, the application logic therefore has to implement result maintenance on top of an event stream query.

2.3.3 Parse

Similar to Meteor, Parse is an app development framework that uses MongoDB as its backing store. It was immensely popular and had one of the largest MongoDB deployments worldwide around the year 2015 [Ges17]. While Meteor and RethinkDB have provided real-time queries since 2012 and 2014, respectively, Parse announced the feature in March 2016 [Wan16b] – after they had announced their own shutdown [Lac16]. In this section, we describe the *Live Query* feature that brings event stream queries to the Parse platform.

Live Queries. Without any support for sorting [Wan16a], Parse's real-time queries are less expressive than those of its peers, even though the architecture itself is very similar. Parse's Live Query mechanism bears a strong resemblance to Meteor's oplog tailing and RethinkDB's changefeeds: Each real-time query is maintained by a single process, the **LiveQuery Server**. Likewise, single-server matching performance is the hard limit for write throughput. In contrast to Meteor and RethinkDB, though, Parse entertains separate processes for OLTP workload (application server) and for real-time query matching (LiveQuery server). Further, application servers publish their change log into a Redis-based message queue instead of handing them directly to the matching processors; this increases scalability and further decouples failure of the query matching process from the main application server. While the approach in itself remains unscalable by design, the Parse implementation appears less likely to break in an overload scenario. This statement cannot be validated, though, because we are not aware of any major projects using Parse Live Queries.

To summarize, the Parse LiveQuery architecture bears resemblance to Meteor's oplog tailing and RethinkDB's changefeeds. Thus, real-time queries do not scale with write throughput and query expressiveness is limited in comparison to pull-based queries: Sorted real-time queries are not supported at all. Like RethinkDB, Parse has *no self-maintaining queries*, but only an event-based real-time query interface.

2.3.4 Firebase

Unlike the other systems discussed in this section, **Firebase** [Fir16] is a proprietary service and only very little is known about the technology stack behind its interfaces. Certain hard limits are known, though, beyond which a single instance will not scale [Fir17f]; for example, more than 100 000 parallel client connections or more than 1 000 write operations per second are not supported [Fir17b]. Firebase has been developed since 2011 and was acquired by Google in 2014 [Tam14]. Even though the core service is often advertised as a "real-time database" [Puf16], it can arguably be better described as a service for cross-device state synchronization due to a very restrictive query interface. Over the years, additional functionality has been added to the Firebase ecosystem through other services, e.g. for authentication, asset hosting, and performance monitoring [Tam16].

Data Modeling & Querying. In contrast to other JSON-based data stores, the original Firebase data model is not represented by a collection of JSON documents, but by one single JSON document: a cloud-hosted tree structure of nested objects and lists. In order to access data, a client essentially has to navigate through the hierarchy and request specific child nodes for which it will receive immediate updates when data is modified by others. Thus, Firebase is natively push-based. However, Firebase only provides little querying capabilities beyond simple lookups by primary key, so that it can be difficult to map application requirements to the simplistic access model Firebase exhibits. The only way to deviate from access by key is to apply a *single* static filter (no logical `AND`/`OR`) or to enforce order on a *single* attribute. Filter expressiveness is limited to lookups ($=$) and range queries ($<$, $\leq$, $>$, $\geq$); in particular, content-based filtering (e.g. through regex queries) is not supported [Ric13] [Leh14] and usually implemented by employing third-party services or systems such as Elasticsearch [Ric14]. For use cases that require more sophistication (e.g. ordering by surname and then by forename), the Firebase team recommends employing workarounds like introducing artificial attributes (e.g. the composite of two other attributes) or retrieving a superset of all relevant data and doing the actual query processing in the client [Wen15]. Likewise, nesting data is officially discouraged [Fir17a], because it makes fine-grained access control impractical and data retrieval more expensive: Since Firebase only allows fetching data subtrees entirely (i.e. including their child nodes), high-level nodes in a deeply-nested tree structure become more bloated as their child count grows. However, flattening out data structures often takes away even more of the expressiveness, since one-to-one and one-to-many relationships are naturally expressed through nesting in the document data model [SF12].

Advanced Queries With Firestore. In late 2017, Firestore was introduced as a document-oriented real-time database service that aims to provide increased scalability and more advanced querying capabilities compared to the original Firebase [Duf17] [Fir17b]. In contrast to the original real-time database service, data in Firestore is organized in a collection of JSON documents rather than one single huge JSON document; thus, Firestore facilitates data retrieval in a more fine-grained fashion. Another relevant improvement is support for composite queries through filter chaining (logical `AND`). However, these improvements come with certain restrictions: First, it is only possible to combine range expressions and lookups on the same attribute or to combine lookup expressions on different attributes [Fir17e]. Put differently, range expressions on different attributes as well as filter disjunction (logical `OR`) are not supported by either the original Firebase nor Firestore. Regarding sorting, Firestore imposes some restrictions as well: If a query contains a range expression, the first sorting key must be the attribute over which the range expression is evaluated [Fir17d]. Thus, a query such as “Find all citizens older than 20 and younger than 30 years, sorted by hometown” is still not feasible. Firestore has been built upon work from

Google's Cloud Datastore [Goo18] [Tam17] and thus inherits some traits and limitations from the underlying systems. Like Megastore [BBC$^+$11], specifically, Firestore provides transactions, but also imposes harsh limits on write throughput: With only 500 writes/s per collection and even only 1 write/s per document [Fir17c], Firestore bars itself from write-heavy applications, similar to Firebase with its restriction to 1 000 writes/s across the entire data set. Firestore further exhibits latency which is several hundred milliseconds higher than Firebase's [Ker17]. Like the original Firebase, Firestore does not support regex queries or comparable content-based filters [Fir17e].

To conclude, Firebase does not suffer from the scalability bottlenecks apparent in the design of Meteor's, RethinkDB's, or Parse's real-time query implementations, but also does not feature sophisticated query mechanisms. Even though denormalizing the data model or evaluating queries in the client can compensate the lack of query expressiveness to a certain degree, sophisticated query patterns tend to be inefficient and awkward to implement [Jam16] [Ros16] [Bov17].

2.3.5 Further Systems

Above, we covered the most expressive real-time databases currently available. In the following, we will briefly survey push-based query mechanisms in other (NoSQL) data storage systems.

Realm [Rea18b] is an embedded database often compared with SQLite that provides cross-device synchronization and collection-based real-time queries, both in the form of event stream queries and self-maintaining queries [Rea17b]. Write operations are executed locally and synchronized with the *Realm Object Server* which broadcasts them to all clients [RRM17]. Since every client replicates the entire database, reads are always executed locally as well and return very fast. At the same time, however, this scheme introduces the client – e.g. a notebook, tablet, or mobile phone – as a bottleneck for processing as well as storage and the Object Server as a bottleneck for change propagation. In consequence, Realm's real-time queries are only feasible in domains with small data sets and low update throughput. **RxDB** [RxD18] is another embedded database that provides real-time queries through local change detection [RxD17b] [RxD17a]. It is written in JavaScript and supports different storage backends, using their respective replication protocol to synchronize data from the backend to the client. Like Realm, RxDB is only feasible in scenarios where change monitoring can be handled by the (mobile) client device. **OrientDB** [Ori18] allows filtering newly written objects by a query predicate through so-called Live Queries. Semantically, they appear somewhat collection-based, since they handle inserts, updates, and deletes. But in contrast to systems like Meteor, OrientDB's Live Queries only react to ongoing write operations and do not deliver initially matching items [Del15]. To maintain an up-to-date query result, it is therefore necessary to combine the pull-based query mechanism for the initial result with the push-based mechanism for updates. Self-maintaining queries are thus only possible with custom code. **CouchDB** [Apa18e] is another contender that has an API for continuous changes [ALS10, Ch. 20]. As in OrientDB, the initial result of a query has to be requested separately from the stream of changes. In contrast to many systems discussed in this thesis, though, CouchDB only pushes matching items to the client, i.e. the client will not receive delete notifications[9]. In consequence, self-maintaining queries are complex to implement and require the client to maintain query state. Similar to OrientDB's Live Queries, **Graphcool** [Gra18] subscriptions filter the write stream by custom criteria. As with OrientDB's Live Queries, the matching process does not consider a query result, but only the data item that is being written [BMS17]. As another similarity, Graphcool subscriptions also do not provide the initially matching items [BM$^+$17]. **Rapid.io** [STR18] is a proprietary database service that provides push-based real-time queries with collection-based semantics, similar to Firestore [Dro17]. As of September 2017, Rapid.io is officially in public Beta and the technology stack behind the query API is undisclosed. Since we are not aware of any case studies or customer reports, we thus cannot set its performance or scalability in perspective to that of the other systems discussed in this section. Filter queries with comparisons, prefix and suffix matching, and containment checks are supported and can be composed using logical `AND`/`OR`; however, it is not possible to use negations or more sophisticated search operators (e.g. regex predicates, wildcards, or case-insensitive search) [STR17]. **Elasticsearch** [Ela18] is a distributed NoSQL database most famously known for its sophisticated full-text search capabilities. Through so-called *percolator queries* [Ban11] [Gro16], Elasticsearch supports push-based access in the sense that clients receive notifications as soon as new matches to their queries are written to the database. However, only new matches are registered and no notifications are sent for documents that are deleted or cease matching after an update. Lastly, real-time APIs for **MongoDB** [Mon18e] and **Stitch** [Mon18h] [Dan17], a cloud backend by the MongoDB creators, have been introduced in 2017 [Mon17a]. However, only stream-based filtering semantics are supported, i.e. updates that remove data from the result cannot be detected [Dav17a] [Dav17b].

2.3.6 Real-Time Databases: Summary & Discussion

Table 2.2 sums up the capabilities of each system detailed in this thesis, comparing them against our system design InvaliDB. Meteor is the only system featuring two different real-time query implementations: Poll-and-diff scales with write throughput and oplog tailing scales with the number of concurrent real-time queries – neither scales with both. RethinkDB and Parse provide real-time queries with mechanisms similar to oplog tailing and therefore also collapse under heavy write load: The lack of write stream partitioning represents a scale-prohibitive bottleneck in the designs of all these systems. While the technology stack behind Firebase is not disclosed, hard scalability limits for both write throughput

[9] The same applies to CouchDB's commercial derivative Cloudant [IBM18].

and parallel client connections are documented. Further, it is apparent that Firebase mitigates scalability issues by simply denying complex queries to begin with: In the original Firebase model, composite queries are impossible and sorted queries are only allowed with single-attribute ordering keys. Even the more advanced Firestore queries lack support for disjunction of filter expressions (logical `OR`) and only provide limited options for filter conjunction (logical `AND`). All systems in the comparison apart from Firebase offer composite filter conditions for real-time queries, but differ in their support for ordered results: Meteor supports sorted real-time queries with limit and offset, RethinkDB only supports limit (but no offset), and Parse does not support sorting for real-time queries whatsoever. All systems covered in the comparison matrix provide an event stream query interface, but only Meteor provides an interface that hides the complexity of handling change deltas from the client (self-maintaining queries).

	Meteor		**RethinkDB**	**Parse**	**Firebase**	**InvaliDB**
	Poll-and-Diff	Oplog Tailing				
Scales With Write Throughput	✓	✗	✗	✗	✗	✓ (see Section 3.3)
Scales With Number of Queries	✗	✓	✓	✓	○ (100k connections)	✓
Composite Queries (`AND`/`OR`)	✓	✓	✓	✓	○ (`AND` in Firestore)	✓
Sorted Queries	✓	✓	✓	✗	○ (single attribute)	✓ (see Section 3.3.2)
Limit	✓	✓	✓	✗	✓	✓
Offset	✓	✓	✗	✗	○ (value-based)	✓
Joins	✗	✗	✗	✗	✗	✓ (see Section 3.3.2)
Aggregations	✗	✗	✗	✗	✗	✓ (see Section 3.3.2)
Self-Maintaining Queries	✓	✓	✗	✗	✗	✓ (see Section 5.3.1)
Event Stream Queries	✓	✓	✓	✓	✓	✓ (see Section 5.3.3)

Table 2.2: A direct comparison of the different collection-based real-time query implementations detailed in this thesis.

Summing up, we are not aware of any system that carries non-trivial pull-based query features to the push-based paradigm without severe compromises: Developers always have to weigh a lack of expressiveness against the presence of hard scalability bottlenecks. Through the system design developed in this thesis, InvaliDB, we show that expressive real-time queries and scalability can go hand-in-hand.

2.4 Data Stream Management

In some domains, data arrives so fast and in such great quantity that storing it in a database collection is simply infeasible [BBD+02]. When the incoming data relates to ongoing (real-world) events that require immediate action, persistence may further not even be useful; for example, data in electronic trading, network monitoring, or real-time fraud detection is only valuable for a short amount of time and therefore has to be utilized immediately [SCZ05]. To adapt to these circumstances, data stream management systems (DSMSs) introduce the **data stream** as an abstraction for an infinite sequence of database records that arrive over time. The raw data streams arriving at the systems are usually referred to as *base streams*, whereas those resulting from data transformations (e.g. queries) are called *derived streams* [GZ10]. Since a data stream is impossible to store entirely due to its unbounded nature, DSMSs drop the database requirement of eternal data persistence: They retain incoming records for limited time only and eventually discard them.

2.4.1 Queries Over Streams

Queries over streams are long-running and produce new output whenever new data is received; this is in contrast to ad hoc queries over collections which produce output once on user request. Thus, queries over streams generate output streams, just like queries over database relations produce output relations (i.e. result lists). In spite of these fundamental differences between queries over collections and queries over streams, however, streaming query languages are usually designed to resemble traditional database languages: While some systems for stream management expose procedural interfaces (e.g. Aurora [ACC+03] [CCC+02] or Borealis [AAB+05]), most systems employ SQL-like query languages that extend relational algebra through special operators for handling streams (e.g. STREAM [ABW06], Tapestry [TGNO92], TelegraphCQ [CCD+03], PipelineDB [Nel17], S-Store [CDK+14]); in fact, several DSMSs are built on top of pull-based databases (e.g. TelegraphCQ and PipelineDB extend PostgreSQL, S-store extends H-Store [KKN+08]). Therefore, data streams are often implemented as **time-varying collections** [GZ10, Sec. 2.3.1] where new records are inserted and old records are deleted (cf. PipelineDB [Pip17] and S-store [CDK+14, Sec. 3.2.1]).

Early systems (e.g. Tapestry) only supported **append-only** streams, i.e. they relied on the assumption that records are only added and never modified or removed. When tuples represent sensor measurements or other unchangeable facts, this assumption is valid; in fact, base stream data is final in most applications [GZ10, Sec. 2.1.1]. Query results are commonly not immutable, though, since they can evolve over time; as an example, consider a query that collects the maximum temperature of the day from a feed of sensor readings. To accommodate such non-monotonic data sources, modern DSMSs are developed with support for **mutable streams** where issued data can be revised [RMCZ06] [GAE06] [GHM+07]: Queries over streams in these systems can be used to maintain predicate-based relations up-to-date, similar to materialized views. Some DSMS query implementations even bear resemblance to the real-time query mechanisms discussed in this thesis. For example, *insert/delete streams* in STREAM [ABW06, Sec. 6.3] and *delta streams* in PipelineDB [Nel17] produce change deltas similar to event stream queries (cf. Section 5.3.3). As another example, *relation streams* in STREAM [ABW06, Sec. 6.3] [BSW04] produce fully maintained results very much like self-maintaining queries do (cf. Section 5.3.1).

It is important to note, though, that DSMSs and real-time databases operate on differently scoped data sets: Since DSMSs do not retain data indefinitely, queries over streams only reflect recent data whereas queries in a real-time database reflect all data that ever entered the system. By intuition, this means that DSMSs essentially access data that is yet to arrive, because old portions of the stream are oftentimes not available anymore. In consequence, queries over streams in DSMSs do not behave like collection-based real-time queries unless they are defined over **monotonic or quasi-monotonic attributes**[10] and only refer to recent data. Queries that reference non-monotonic attributes (e.g. username) can only produce output from the currently buffered portion of the stream; data that has been discarded is effectively lost for queries to come. Therefore, use of non-monotonic attributes for queries over streams is sometimes forbidden (see for example Calcite [Cal17]).

```
SELECT name
  FROM users
```

(a) This query cannot be answered without knowledge of the entire data stream.

```
SELECT name
  FROM users
  WHERE timestamp > now()
```

(b) This query explicitly and exclusively refers to future information and can therefore be processed based on the stream alone.

Listing 2.1: Whether or not a query can be answered on the basis of an ephemeral data stream without access to the stream history depends on the temporal scope of the query.

For illustration, consider a data management system receiving a stream of user actions as presented in Section 2.1.1. If the data management system is a DSMS, it disposes of data after a while and therefore is only aware of users that have been active recently. A real-time database, on the other hand, would be aware of all users, because it maintains a persistent data repository that reflects the entire data stream. This point is illustrated by the queries in Listing 2.1: A complete list of all registered users (a) can only be produced

[10] An attribute is *monotonic*, if all its values are either decreasing or increasing, such as arrival timestamps in a centralized DSMS. Similarly, an attribute is *quasi-monotonic*, if it is correlated to a monotonic attribute. For example, the time at which an event is registered according to a sensor's local clock (event time) is quasi-monotonic, because it typically corresponds to the time at which it is received according to the server's clock (arrival time) within a certain error margin (cf. Section 2.4.2).

by the real-time database, because the DSMS does not have access to all relevant data as soon as the first portion of the stream has been discarded. If the result is constrained to current and future users (b), however, the query result can be computed by the real-time database and the DSMS alike, because it does not require historical knowledge of the stream.

Several DSMSs account for the need to access historical data by letting users combine queries over unbounded data streams with queries over persistent database collections. While this enables both query types illustrated in Listing 2.1, executing stream queries without bounded windows is often considered infeasible for practical workloads (cf. [Wid05]) as scalability of database-rooted DSMSs is typically limited. STREAM, for example, is a centralized DSMS that does not scale horizontally at all [ABB$^+$16, Sec. 8.1], while PipelineDB does not support data sharding (i.e. every node maintains the entire data set) and is only able to distribute query processing across few nodes because of synchronization overhead for replication[11]. For systems that are not built on databases, achieving query semantics as illustrated above can further be difficult, because their query languages are sometimes hardly comparable with traditional database languages such as SQL[12].

2.4.2 Notions of Time

Since records in streams often refer to events in a complex distributed system, they can be attributed with different timestamps. For example, a record may carry the time at which it was received by the DSMS (**arrival time**) and the time at which the corresponding event actually occurred according to the measuring device's local clock (**event time**) [GZ10]. Event and arrival time are usually correlated and ideally close. In practice, however, there is often a delay between the occurrence of an event and its registration in the stream management system. For example, consider user data that is collected on a smartphone and reported sporadically through the Internet; depending on network connectivity, clock skew between the mobile device and the DSMS, and other factors, event time and arrival time for a particular record may diverge by seconds, hours, or even days [Aki15].

Since base streams are naturally ordered by the arrival timestamp, they have to be reorganized whenever application semantics revolve around a different notion of time such as event time. For **stream reordering**, the input stream is buffered for t_Δ time units and items within the buffered portion of the stream are reorganized and emitted according to the sort specification [ACC$^+$03, Sec. 5.2.2]. Records that arrive more than t_Δ time units late are typically dropped from the reordered stream [GZ10, Sec. 2.2.2]. It is also possible to revise stream output on receiving **delayed records** instead of dropping them: The basic

[11] All write operations in PipelineDB are coordinated synchronously via two-phase commit between all nodes [Pip15], so that highly distributed setups are likely to experience increased latency as well as reduced throughput and availability [Pan15, Sec. 3.1].

[12] As an example, consider the graphical user interface of Aurora/Borealis which is based on arrows and boxes rather than SQL-style declarative statements [ÇAA$^+$16].

idea is to revoke previously issued information (that has turned out to be incorrect) and to emit updated records which reflect the new information [RMCZ06]. However, providing **revised output** takes a performance toll on the upstream component, just like applying revised output can be expensive for downstream components[13].

Since system performance can degrade significantly when output has to be retained for a long time [ABB^{+}13, Sec. 8.4], different approaches limit the amount of state utilized for compensating out-of-order arrival. Some systems use a fixed value for the time or the number of records to buffer (e.g. Aurora [ACC^{+}03]). Other systems dynamically control the amount of metadata by measuring or estimating the current delay within the application stack (e.g. NiagaraCQ [CDTW00], MillWheel [ABB^{+}13], Gigascope [JMSS05]). Special records are used to propagate this kind of information from upstream components further downstream. For example, a **punctuation** [TMSF03] is a record that carries an invariant condition such as `timestamp` $\geq$ `'12h05m00s'` that is guaranteed to be true for all subsequent records; thus, all buffered records that do not fulfill the invariant can be safely abandoned (e.g. all records from before timestamp `'12h05m00s'`). So-called **heartbeats** [SW04] or **watermarks** [ABB^{+}13] serve a similar purpose. Computing or estimating these invariant conditions, however, is challenging in some settings [SW04].

2.4.3 Windowing & Approximation

Queries over streams are usually evaluated in the context of a **window**, i.e. a finite partition of the conceptually unbounded sequence of records. There are several dimensions by which a window can be described [GZ10, Sec. 2.1.2]. One dimension is the **direction** of movement, relating to the way that start and end point of the window are chosen: Either both are fixed (*fixed window*), one is fixed and one is moving (*landmark window*), or both are moving (*sliding window*) [PS06]; windows can also be *expanding* or *contracting*, depending on whether the window boundaries are moving in the same direction or whether they are moving at the same speed. Further, windows can be distinguished by the way their **contents** are defined. Most commonly, window contents are specified in terms of *time* (e.g. “all data from the last two minutes”) or *count* (e.g. “the last 1 000 records”) [ABW06, Sec. 6], but other forms of defining a stream partition are possible, for instance based on query *predicates* [GAE06] [JMS^{+}08]. Movement and **update frequency** can also be used to characterize a window. Intuitively, a *sliding window* [Gol06] is eagerly refreshed with every incoming record: The window advances as one record enters and another one leaves. When a new query result is produced lazily every n time units or every n tuples instead, the window is called a *jumping window* [MVLL05]. When the update interval equals window size, the stream is split into contiguous, non-overlapping ranges.

[13] Specifically, providing undo information requires buffering the original output [ABC^{+}15, Sec. 2.3]. Likewise, reprocessing huge amounts of data to generate updated records can lead to CPU contention and can thus significantly impair overall system performance [Kre14c].

Since these windows are often illustrated to “tumble over” from one range to the next, they are referred to as *tumbling windows*.

In comparison to sliding or jumping windows, tumbling windows are relatively easy to implement, because their **state** is reset on every move: In other words, only new tuples have to be incorporated when updating a tumbling window. Sliding and jumping windows, in contrast, are more complex (and sometimes less efficient) to realize [GZ10], because they have to reflect tuples moving out of the window as well. In consequence, the metadata required for incremental computation of a query result over a sliding window can vary significantly depending on the query type [Feg16]. For example, maintaining a counter is very straightforward, as it is incremented for every new record and decremented for every expiring record; more complex aggregations such as an average, in contrast, may require retaining all records within the window, because their contribution to the query result depends on their concrete values. In Section 5.4.2, we provide two examples for incremental maintenance of aggregate values, namely count and average.

To reduce the amount of metadata required for incremental query result computation, specific algorithms and data structures have been developed for the **approximation** of value frequencies [Gib01] [PT05] [MM02], quantiles [GK01] [LLXY04], top-k queries [MBP06] [DLOM02] [GM98] [MAEA05b], skyline queries [LYWL05, TP06], aggregate queries [DGIM02] [LMT$^+$05] [GKS01a] [CM05], range queries [BL10], and other query types [DGIM02] [FKSV03] [MAEA05a] [GKS01b]. Workloads are further bursty in many streaming applications [Kle02] and therefore input rates may (temporarily) exceed system capacity, despite optimized implementations. In order to prevent system overload in such scenarios, a DSMS may resort to **load shedding** [TCZ$^+$03]: Here, the DSMS deliberately skips records within the stream to reduce effective load. There are different strategies for selecting to-be-skipped tuples (e.g. probabilistic sampling [SH12] [BDM07] or selection by semantic criteria [AN04]). Which one fits a given scenario best depends on the optimization target (e.g. throughput, quality of service [MWA$^+$03]).

InvaliDB evaluates queries over streams of database records in order to provide incremental result updates for common database queries. While it is thus technically stream-based, InvaliDB’s query engine has to be aligned perfectly with the semantics of the underlying data store and therefore follows collection-based semantics. In consequence, reducing load by approximation or load shedding bears the risk of compromising correctness in the context of our work: Given the same query and the same input records, InvaliDB’s query engine must produce the exact same matching decisions as the query engine of the underlying database. Approximative query evaluation and write workload shedding are therefore incompatible with the collection-based real-time query semantics focused in this work. Query load shedding, in contrast, can be applied safely by rejecting or aborting individual real-time query subscriptions to reduce overall workload. Since InvaliDB’s query engine is pluggable (cf. Section 3.3.2), it could also be implemented as a true DSMS without backing database. All techniques discussed in this section would be applicable to such an InvaliDB implementation with purely stream-based query semantics (cf. future work in Section 6.2.1).

2.4.4 Complex Event Processing

In some applications, the relevant information may not be explicitly represented in the individual base stream entries, but rather implicitly encoded in the context between them. Systems for **complex event processing (CEP)** [CM12] [CVZ13] address these applications by capturing the temporal, local, or even causal relationships and dependencies between individual records. Similar to active databases (cf. Section 2.2.1), CEP systems execute business logic and thus proactively trigger actions; these may range from a simple notification of maintenance personnel to an emergency shutdown of failing hardware [ENL11]. Depending on the concrete system, the **rules** that determine application behavior are either defined through a declarative language, imperative programming, or a graphical user interface [VRR10]. By establishing a **context** between stream entries, CEP systems thus create data streams with a higher level of **abstraction** from the low-level base streams [BD15]. Typical use cases include prediction of customer behavior [AGR^{+}09], monitoring freight logistics [RRH13], routing network traffic in realtime to maximize quality of service [Ara13], and intrusion detection [FR11].

For illustration, consider an array of temperature sensors deployed for monitoring hardware in a data center. Through aggregation and correlation of different readings over time, a CEP engine can derive pieces of information which are more abstract and more relevant to the application domain than the raw temperature values [Pal13]. For example, **complex error conditions** (e.g. machines in a particular rack overheating) might be detected by considering measurements of different sensors over time. Likewise, sensor readings that appear plausible in themselves might be uncovered as faulty when they deviate significantly from output of colocated sensors.

The requirement to correlate events with one another makes CEP engines inherently difficult to scale across machine boundaries [CGH^{+}17] and therefore deployments usually do not span more than a few nodes [Esp16] [IBM14] [Pip15]. However, distribution can be achieved by application-level sharding or by building abstraction hierarchies that reduce the number of events to process on every individual machine [BD15, Sec. 2.4]. Since general-purpose stream processing systems (cf. Section 2.5) can also be used for complex event processing and sometimes even provide abstractions to facilitate this task (e.g. Flink [KW17]), the line between complex event processing and general-purpose stream processing has become blurred [Vin16].

In contrast to the CEP systems mentioned in this section, InvaliDB is neither designed for detecting complex event patterns nor for establishing temporal, causal, or other relationships between individual occurrences. As described in Section 3.2.1, we rather entertain a simplistic notion of event processing in the context of this work where different events can be distinguished by their respective match type (i.e. by the kind of query result alteration associated with an event) and by their respective operation type (i.e. the type of operation that triggered the event). Since InvaliDB's query engine is pluggable, though, more sophisticated complex event processing mechanisms could be developed in the future (see Section 6.2.1).

2.4.5 Messaging Middleware

Processing streaming data in a massively distributed system necessitates funneling information from the system periphery into a data stream and propagating it with low latency. In contrast to systems for change data capture (cf. Section 2.2.2), **message-oriented middleware (MOM)** is not primarily concerned with collecting relevant information on events as they occur, but mainly implements efficient and often reliable mechanisms for data distribution: A **producer** provides data items to the messaging middleware which then delivers them to one or many **consumers**. Various forms of propagation are common, for example *point-to-point* (single producer, single consumer) or *publish-subscribe* (single producer, multiple consumers) delivery. The provided delivery guarantees range from none (e.g. publish-subscribe in Redis [San18b] or NATS [Clo18a]) over at-least-once (e.g. in Kafka [KNR11]) to exactly-once (e.g. RabbitMQ [Piv18], ActiveMQ [Apa18a], Qpid [Apa18h], HornetQ [Gia12], or IBM WebSphere MQ [LLO$^+$12]). In order to acknowledge individual messages, some MOMs keep track of each consumer's delivery log and retain messages that have not been acknowledged by all consumers. When implemented in this fashion, exactly-once delivery may incur significant overhead and can even bring the entire system down when just one single consumer stays disconnected for a long time. To avoid this kind of failure scenario, **distributed log systems** such as Kafka archive all data to disk and allow clients to request data replays by providing a log offset of event history. Thus, data can be retained for days or even weeks with minimal processing overhead for the middleware, while *at-least*-once delivery guarantees are naturally met through the ability to replay the archived messages. *Exactly*-once delivery guarantees are still achievable, but at the cost of reduced scalability [Tre15] [Tre17] and/or significantly increased system complexity [Nar17]. While simple content-based and even rule-based filtering is supported by some systems (e.g. Siena [CRW01] or Delta [KKM13]), querying capability is typically limited in comparison to the systems discussed above.

2.4.6 Data Stream Management: Summary & Discussion

Data stream management systems are similar to real-time databases in several ways. First, they support continuous queries, i.e. they proactively deliver information as soon as new data of relevance becomes available. Second, many of them are also capable of ad hoc queries over currently buffered data; in fact, many data stream management systems are extensions of existing databases and therefore inherit some of their capabilities. As an important distinction to real-time databases, however, data streams are typically retained for only a relatively short amount of time. Seeing that data stream management systems are thus oriented towards current and future events, querying data that is rooted in the past is inefficient or impossible without a second system for persistent data management.

2.5 General-Purpose Stream Processing

Unlike data stream management systems that are mostly intended for analyzing structured information through declarative query languages, systems for stream processing expose generic and imperative (i.e. non-declarative) programming interfaces to work with structured, semi-structured, and entirely unstructured data. Rather than yet another approach for querying data, stream processing can thus be seen as the latency-oriented counterpart to batch processing. In this section, we provide an overview over some of the most popular distributed stream processing systems currently available and highlight similarities, differences, and trade-offs taken in their respective designs. In Chapter 4, we will choose a stream processor for our InvaliDB prototype based on these findings.

2.5.1 Architectural Patterns

In contrast to traditional data analytics systems that collect and periodically process huge – static – volumes of data, streaming analytics systems avoid putting data at rest and process it as it becomes available, thus minimizing the time a single data item spends in the processing pipeline. Stream processing pipelines often routinely achieve end-to-end latencies of several seconds or even subsecond latency.

Figure 2.5 illustrates typical layers of a streaming analytics pipeline. Data like user clicks, billing information, or unstructured content such as images or text messages are collected from various places inside an organization and then moved to the streaming layer (e.g. a queuing/streaming system like Kafka [KNR11] or Kinesis [Ama18]) from which it is accessible to a stream processor that performs a certain task to produce an output. This output is then forwarded to the serving layer which might for example be an analytics web GUI like trending topics at Twitter or a database where a materialized view is maintained.

In an attempt to combine the best of both worlds, an architectural pattern called the **Lambda Architecture** [MW15] has become quite popular that complements the slow batch-oriented processing with an additional real-time component and thus targets both the *Volume* and the *Velocity* challenge of Big Data [Lan01] at the same time. As illustrated in Figure 2.6a, the Lambda Architecture describes a system comprising three layers: Data is stored in a *persistence layer* like HDFS [SKRC10] from which it is ingested and processed by the *batch layer* periodically (e.g. once a day), while the *speed layer* handles the portion of the data that has not yet been processed by the batch layer, and the serving layer

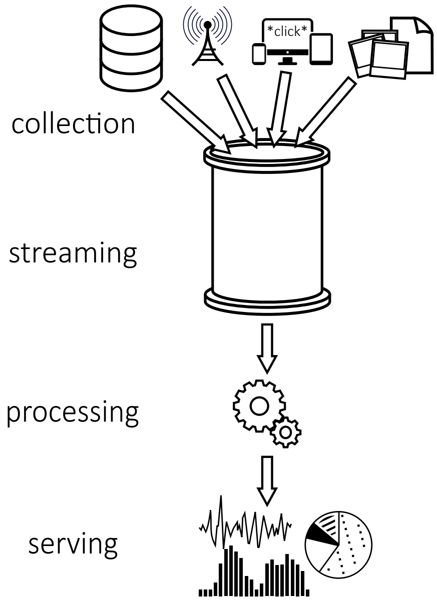
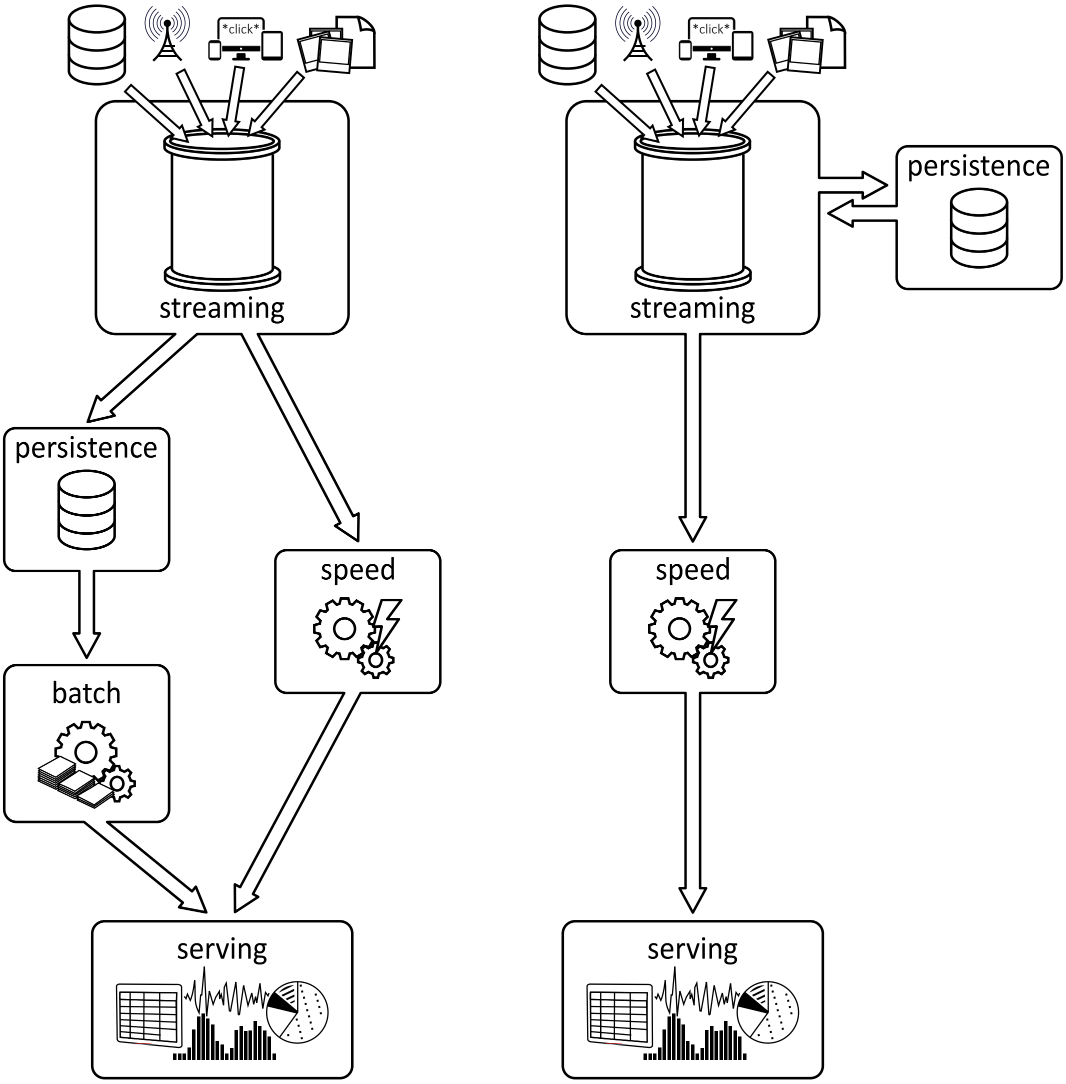


Figure 2.5: An abstract view on a streaming analytics pipeline.

consolidates both by merging the output of the batch and the speed layer. The obvious benefit of having a real-time system compensate for the high latency of batch processing is paid for by increased complexity in development, deployment, and maintenance. If the batch layer is implemented with a system that supports both batch and stream processing (e.g. Spark), the speed layer often can be implemented with minimal overhead by using the corresponding streaming API (e.g. Spark Streaming) to make use of existing business logic and the existing deployment. For Hadoop-based and other systems that do not provide a streaming API, however, the speed layer is only available as a separate system. Using an abstract language like Summingbird [BROL14] to write the business logic enables automatic compilation of code for both the batch and the stream processing system (e.g. Hadoop and Storm) and thus eases development in those cases where batch and speed layer can use (parts of) the same business logic, but the overhead for deployment and maintenance still remains.

Another approach that, in contrast, dispenses with the batch layer in favor of simplicity is known as the **Kappa Architecture** [Kre14b] and is illustrated in Figure 2.6b. The basic idea is to not periodically recompute all data in the batch layer, but to do all computation in the stream processing system alone and only perform recomputation when the business logic changes by replaying historical data. To achieve this, the Kappa Architecture employs a powerful stream processor capable of coping with data at a far greater rate than it is incoming and a scalable streaming system for data retention. An example of such a streaming system is Kafka which has been specifically designed to work with the stream processor Samza in this kind of architecture. Archiving data (e.g. in HDFS) is still possible, but not part of the critical path and often not required as Kafka, for instance, supports retention times in the order of weeks. On the downside, however, the effort required



(a) The Lambda Architecture achieves low latency by complementing a batch-oriented with a stream-oriented processing system.

(b) The Kappa Architecture relies on stream-oriented processing only.

Figure 2.6: Lambda and Kappa Architecture in comparison.

to replay the entire history increases linearly with data volume and the naive approach of retaining the entire write stream may have significantly greater storage requirements than periodically processing the new data and updating an existing database, depending on whether and how efficiently the data is compacted in the streaming layer. As a consequence, the Kappa Architecture should only be considered an alternative to the Lambda Architecture in applications that do not require unbounded retention times or allow for efficient compaction (e.g. because it is reasonable to only keep the most recent value for each given key).

Of course, the latency displayed by the stream processor (speed layer) alone is only a fraction of the end-to-end application latency due to the impact of the network or other systems in the pipeline. But it is obviously an important factor and may dictate which system to choose in applications with strict timing SLAs. In the context of this thesis, low latency is particularly important in order to provide real-time queries.

2.5.2 State-of-the-Art Systems

While all stream processors share some common ground regarding their underlying concepts and working principle, an important distinction between the individual systems that directly translates to the achievable speed of processing (i.e. latency) is the processing model as illustrated in Figure 2.7: Handling data items immediately as they arrive minimizes latency at the cost of high per-item overhead (e.g. through messaging), whereas buffering and processing them in batches yields increased efficiency, but obviously increases the time the individual item spends in the data pipeline. Purely stream-oriented systems such as Storm and Samza provide very low latency and relatively high per-item cost, while batch-oriented systems achieve unparalleled resource-efficiency at the expense of latency that is prohibitively high for real-time applications. The space between these two extremes is vast and some systems like Storm Trident and Spark Streaming employ micro-batching strategies to trade latency against throughput: Trident groups tuples into batches to relax the one-at-a-time processing model in favor of increased throughput, whereas Spark Streaming restricts batch size in a native batch processor to reduce latency. In the following, we go into more detail on the specificities of the above-mentioned systems and highlight inherent trade-offs and design decisions.

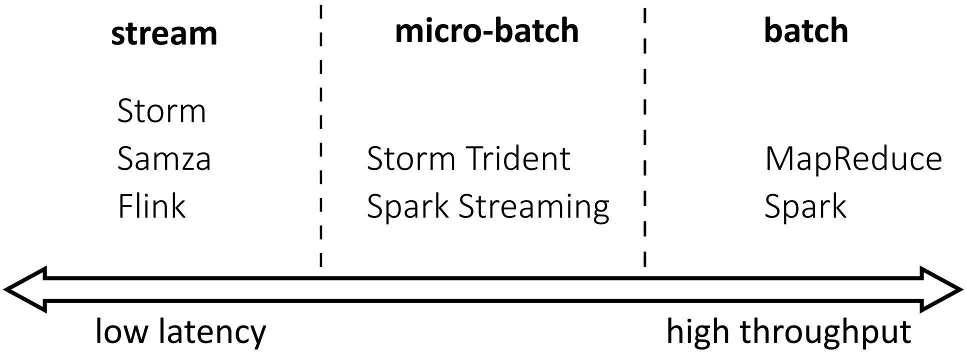


Figure 2.7: Choosing a processing model means trading off latency against throughput.

Storm has been in development since late 2010, was open-sourced in September 2011 by Twitter, and eventually became an Apache top-level project in 2014. It is the first distributed stream processing system to gain traction throughout research and practice and was initially promoted as the "Hadoop of realtime" [Mar12, Mar14], because its programming model provided an abstraction for stream processing similar to the abstraction that the MapReduce paradigm provides for batch processing. But apart from being the first of its kind, Storm also has a wide user base due to its compatibility with virtually any language: On top of the Java API, Storm is also Thrift-compatible [SAK07] and comes with adapters for numerous languages such as Perl, Python, and Ruby. Storm can run on top of Mesos

[HKZ+11], as a dedicated cluster, or even on a single machine. The vital parts of a Storm deployment are a **ZooKeeper** [HKJR10] cluster for reliable coordination, several **supervisors** for execution, and a **Nimbus** server to distribute code across the cluster and take action in case of worker failure; in order to shield against a failing Nimbus server, Storm allows having several hot-standby Nimbus instances. Storm is scalable, fault-tolerant, and even elastic as work may be reassigned at runtime. As of version 1.0.0, Storm provides reliable state implementations that survive and recover from supervisor failure. However, Storm's state management is only feasible for applications with small state, because updates are persisted *synchronously* and therefore can dominate latency when they are large. Earlier versions of Storm only provided the option of stateless processing and thus required state management at the application level to achieve fault tolerance and elasticity in stateful applications. Storm excels at speed and thus is able to perform in the realm of low double-digit milliseconds when carefully tuned (for example, see the evaluation of our InvaliDB prototype in Chapter 4). Through the impact of network latency and garbage collection, however, real-world topologies usually do not display consistent end-to-end latency below 50 ms [GMSS15, Ch. 7].

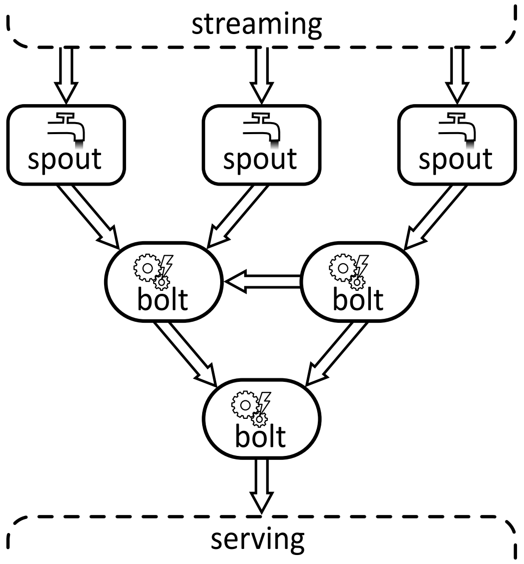


Figure 2.8: Data flow in a Storm topology: Data is ingested from the streaming layer and then passed between Storm components, until the final output reaches the serving layer.

A data pipeline or application in Storm is called a **topology**. As illustrated in Figure 2.8, a topology is a directed graph that represents data flow as directed edges between nodes which again represent the individual processing steps: The nodes that ingest data and thus initiate the data flow in the topology are called **spouts** and emit **tuples** to the nodes downstream which are called **bolts** and do processing, write data to external storage, and may send tuples further downstream themselves. Storm comes with several **groupings** that control data flow between nodes, e.g. for shuffling or hash-partitioning a stream of tuples by some attribute value, but also allows arbitrary custom groupings. By default, Storm distributes spouts and bolts across the nodes in the cluster in a round-robin fashion, although the scheduler is pluggable to account for scenarios in which a certain processing step has to be executed on a particular node, for example because of hardware dependencies. The application logic is encapsulated in a manual definition of data flow and the spouts and bolts which implement interfaces to define their behaviour during startup, and on data ingestion or on receiving a tuple, respectively.

While Storm does not provide any guarantee on the order in which tuples are processed, it does provide the option of at-least-once processing through an **acknowledgement** feature that tracks the processing status of every single tuple on its way through the topology: Storm will replay a tuple, if any bolt involved in processing it explicitly signals failure or does not acknowledge successful processing within a given timeframe. Using an appropriate streaming system, it is even possible to shield against spout failures, but the acknowledgement feature is often not used in practice, because the messaging overhead imposed by tracking tuple **lineage** (i.e. a tuple and all the tuples that are emitted on its behalf) noticeably impairs achievable system throughput [CDE$^+$15]. With version 1.0.0, Storm introduced a **backpressure** mechanism to throttle data ingestion as a last resort whenever data is ingested faster than it can be processed. If processing becomes a bottleneck in a topology without such a mechanism, throughput degrades as tuples eventually time-out and are either lost (at-most-once processing) or replayed repeatedly to possibly time-out again (at-least-once processing), thus putting even more load on an already overburdened system.

Storm Trident was released in autumn 2012 and version 0.8.0 as a high-level API with stronger ordering guarantees and a more abstract programming interface with built-in support for joins, aggregations, grouping, functions, and filters. In contrast to Storm, Trident topologies are **directed *acyclic* graphs (DAGs)** as they do not support cycles; this makes them less suitable for implementing iterative algorithms and is also a difference to plain Storm topologies which are often wrongfully described as DAGs [Apa18g], but actually can introduce cycles. Also, Trident does not work on individual tuples, but on micro-batches. Correspondingly, Trident introduces batch size as a parameter to increase throughput at the cost of latency which, however, may still be as low as several milliseconds for small batches [Eri14]. All batches are by default processed in sequential order, although Trident can also be configured to process multiple batches in parallel. On top of Storm's scalability and elasticity, Trident provides its own API for fault-tolerant state management with exactly-once processing semantics. In more detail, Trident prevents data loss by using Storm's acknowledgement feature and guarantees that every tuple is reflected only once in persistent state by maintaining additional information alongside state and by applying updates transactionally. As of writing, two variants of state management are available: One only stores the sequence number of the last-processed batch together with current state, but may block the entire topology when one or more tuples of a failed batch cannot be replayed (e.g. due to unavailability of the data source), whereas the other can tolerate this kind of failure, but is more heavyweight as it also stores the last-known state. Irrespective of whether batches are processed in parallel or one by one, state updates have to be persisted in strict order to guarantee correct semantics. As a consequence, their size and frequency can become a bottleneck and Trident can therefore only feasibly manage small state.

Samza [NPP$^+$17] [Ram15] is very similar to Storm in that it is a stream processor with a one-at-a-time processing model and at-least-once processing semantics. It was initially created at LinkedIn, submitted to the Apache Incubator in July 2013 and was granted top-level status in 2015. Samza was co-developed with the queuing system **Kafka**[14] [KNR11] and therefore relies on the same messaging semantics: Streams are partitioned and **messages** (i.e. data items) inside the same partition are ordered, whereas there is no order between messages of different partitions. Even though Samza can work with other queuing systems, Kafka's capabilities are effectively required to use Samza to its full potential and therefore it is assumed to be deployed with Samza for the rest of this section. In comparison to Storm, Samza requires a little more work to deploy as it does not only depend on a ZooKeeper cluster, but also runs on top of Hadoop YARN [Apa16h] for fault tolerance: In essence, application logic is implemented as a **job** that is submitted through the Samza YARN client which has YARN then start and supervise one or more **containers**. Scalability is achieved through running a Samza job in several parallel **tasks** each of which consumes a separate Kafka partition; the degree of parallelism, i.e. the number of tasks, cannot be increased dynamically at runtime. Similar to Kafka, Samza focuses on support for JVM-languages, particularly Java. Contrasting Storm and Trident, Samza is designed to handle *large amounts of state* in a fault-tolerant fashion by persisting state in a local database and replicating state updates to Kafka. By default, Samza employs a key-value store for this purpose, but other storage engines with richer querying capabilities can be plugged in.

As illustrated in Figure 2.9, a Samza job represents one processing step in an analytics pipeline and thus roughly corresponds to a bolt in a Storm topology. In stark contrast to Storm where data is directly sent from one bolt to another, though, output produced by a Samza job is always written back to Kafka from where it can be consumed by other Samza jobs. Although a single Samza job or a single Kafka persistence hop may delay a message by only a few milliseconds [Kre14a], latency adds up and complex analytics pipelines comprising several processing steps eventually display higher end-to-end latency than comparable Storm implementations.

On the upside, however, this design also decouples individual processing steps and thus eases development. Another advantage is that buffering data between processing steps makes (intermediate) results available to unrelated parties, e.g. other teams in the same

[14] In 2016, a native stream processor was introduced to Kafka: **Kafka Streams** [Kre16] is not only conceptually similar to Samza, but was also built by the same people, reusing portions of the Samza source code [PM16]. Kafka Streams can therefore be considered an unofficial Samza successor.

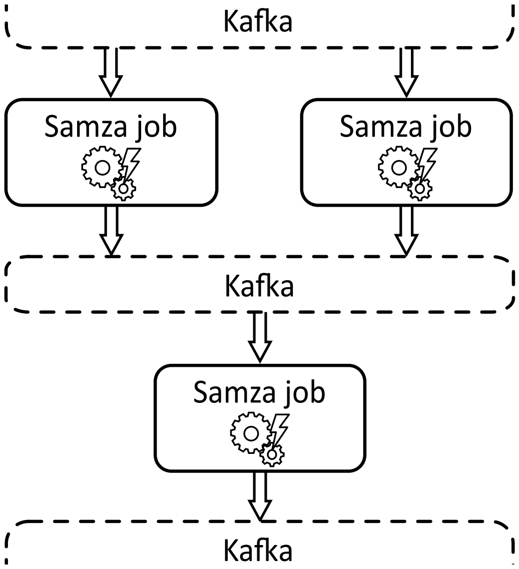


Figure 2.9: Data flow in a typical Samza analytics pipeline: Samza jobs cannot communicate directly, but have to use a queuing system such as Kafka as message broker.

company. Further, it eliminates the need for a backpressure algorithm, since there is no harm in the backlog of a particular job filling up temporarily, given a reasonably sized Kafka deployment. Since Samza processes messages in order and stores processing results durably after each step, it is able to prevent data loss by periodically checkpointing current progress and reprocessing all data from that point onwards in case of failure; in fact, Samza does not support a weaker guarantee than at-least-once processing, since there would be virtually no performance gain in relaxing this guarantee. While Samza does not provide exactly-once semantics, it allows configuring the checkpointing interval and thus offers some control over the amount of data that may be processed multiple times in an error scenario.

Spark [ZCD$^+$12] is a batch processing framework that is often mentioned as the unofficial successor of Hadoop as it offers several benefits in comparison, most notably a more concise API resulting in less verbose application logic and significant performance improvements through in-memory caching. In particular, iterative algorithms (e.g. machine learning algorithms such as k-means clustering or logistic regression) are accelerated by orders of magnitude, because data is not necessarily written to and loaded from disk between every processing step. In addition to these performance benefits, Spark provides a variety of machine learning algorithms out-of-the-box through the **MLlib** library. Originating from UC Berkeley in 2009, Spark was open-sourced in 2010 and was donated to the Apache Software Foundation in 2013 where it became a top-level project in February 2014. It is mostly written in Scala and has Java, Scala, and Python APIs. The core abstraction of Spark are distributed and immutable collections called **RDDs** (resilient distributed datasets)[15] that

[15] On top of RDDs, Spark provides DataFrames and Datasets as even more abstract APIs that impose a schema on the otherwise unstructured RDD tuples [Dat18].

can only be manipulated through deterministic operations. Spark is resilient to machine failures by keeping track of any RDD's **lineage**, i.e. the sequence of operations that created it, and checkpointing RDDs that are expensive to recompute, e.g. to HDFS [SKRC10]. A Spark deployment consists of a cluster manager for resource management (and supervision), a **driver program** for application scheduling, and several **worker** nodes to execute the application logic. Spark runs on top of Mesos, YARN, or in standalone mode in which case it may be used in combination with ZooKeeper to remove the **master node** (i.e. the cluster manager) as a single point of failure.

Spark Streaming [ZDL$^+$13] shifts Spark's batch processing approach towards real-time requirements by chunking the stream of incoming data items into small batches, transforming them into RDDs, and processing them as usual. It further takes care of data flow and distribution automatically. Spark Streaming has been in development since late 2011 and became part of Spark in February 2013. Being a part of the Spark framework, Spark Streaming had a large developer community and also a huge group of potential users from day one, since both systems share the same API and since Spark Streaming runs on top of a common Spark cluster. Thus, it can be made resilient to failure of any component [VPAU15] like Storm and Samza and further supports dynamically scaling the resources allocated for an application. Data is ingested and transformed into a sequence of RDDs which is called **DStream** (discretized stream) before processing through workers. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees. In consequence, the order in which data items are processed may diverge from the order in which they are received, roughly by the batch size. Since there is a certain job scheduling delay when processing an RDD, batch sizes below 50 ms tend to be infeasible [Apa16f, Sec. "Performance Tuning"]. Accordingly, processing an RDD takes around 100 ms in the best case, although Spark Streaming is designed for latency in the order of a few seconds [ZDL$^+$13, Sec. 2]. To prevent data loss even for unreliable data sources, Spark Streaming grants the option of using a **write-ahead log (WAL)** from which data can be replayed after failure. State management is realized through a **state DStream** that can be updated through a DStream transformation.

Flink [Apa16d] is a project that has many parallels to Spark Streaming as it also originated from research and advertises the unification of batch and stream processing in the same system, providing exactly-once guarantees for the stream programming model and a high-level API comparable to that of Trident. Formerly known as Stratosphere [ABE$^+$14], Flink entered the Apache Incubator under its current name in mid-2014, received top-level status in January 2015 [Apa15], and reached stable version 1.0.0 in March 2016 [Apa16a]. In contrast to Spark Streaming, Flink is a native stream processor and does not rely on batching internally. Apart from the batching API and the streaming API in the focus of this section, Flink also provides APIs for graph processing, complex event processing, SQL, and an executor to run Storm topologies [Sax15]. Flink can be deployed using a resource negotiator such as YARN or Mesos, but also in standalone mode directly on machines. A Flink deployment has at least one **job manager** process (with optional standbys for failover) to coordinate checkpointing and recovery and for receiving Flink **jobs**. The job manager also schedules work across the **task manager** processes which usually reside on separate machines and in turn execute the code. Resource allocation for a job was initially static [Ewe16], but dynamic scaling has been added in version 1.5 in May 2018 [Hue18].

Conceptually, Flink can be considered one of the more advanced stream processors as many of its core features were already considered in the initial design and not just added as an afterthought as opposed to Spark's streaming API or state management in Storm, for instance. However, only relatively few big players have committed to using it in production so far (cf. [Apa16g] [Apa16b]). To provide exactly-once processing guarantees, Flink uses an algorithm grounded in the Chandy-Lamport algorithm for distributed snapshots [CL85]: Essentially, **watermark items** are periodically injected into the data stream and trigger any receiving component to create a checkpoint of its local state. On success, the entirety of all local checkpoints for a given watermark comprise a **distributed global system checkpoint**. In a failure scenario, all components are reset to the last valid global checkpoint and data is replayed from the corresponding watermark. Since data items may never overtake watermark items (which are therefore also called *barriers*), acknowledgment does not happen on a per-item basis and is consequently much more lightweight than in Storm. Flink implements a back pressure mechanism [CTE15] through buffers with bounded capacity: Whenever ingestion is overtaking processing speed, the data buffers effectively behave like fixed-size blocking queues and thus slow down the rate at which new data enters the system. By making the buffering time for data items configurable, Flink promotes an explicit trade-off between latency and throughput and can sustain higher throughput than Storm. But while Flink is also able to provide consistent latency below 100 ms, it cannot satisfy as aggressive latency goals as Storm [CDE$^+$15].

Flink provides several APIs to execute *collection-based* (relational) queries: The only distinction between the functionally equivalent **Table and SQL APIs** [HWJ17] is that the first is integrated into the host programming language whereas the latter executes standardized SQL queries as the name implies. The **DataStream API** [Wal17] offers an abstraction to perform complex *stream-based* queries, including operators for windowed aggregations and stream joins. By inserting all tuples within a data stream into an ever-growing and initially empty **dynamic table** [HWJ17], collection-based queries become applicable to streaming data. While *continuous queries* over dynamic tables are push-based and follow collection-based semantics like the self-maintaining queries presented in this thesis, they do not reflect historical data, but only tuples that have arrived since table creation. Therefore, Flink's continuous queries bear similarity to PipelineDB's continuous views (see Section 2.4) rather than InvaliDB's real-time queries.

Further Systems. In the last couple of years, a great number of stream processors have emerged that all aim to provide high availability, fault tolerance, and horizontal scalability. Much like Flink, **Apex** [Apa18b] is a native stream processor that promises high performance in stream and batch processing with low latency in streaming workloads. It has been in development since 2012, was accepted as Apache Incubator project in August 2015 and was granted top-level status in April 2016 [Apa16c]. It is also complemented by a host of database, file system, and other connectors as well as pattern matching, machine learning, and more algorithms through an additional library, called Apex-Malhar. Compared to projects like Spark Streaming or Flink, Apex has only few contributors and little development activity [Apa18f]. **Heron** [KBF^{+}15] was developed to replace Storm at Twitter and is completely API-compatible to Storm, but improves on several aspects such as backpressure, efficiency, resource isolation, multi-tenancy, ease of debugging, and performance monitoring. It was open-sourced in May 2016 [Ram16]. **MillWheel** [ABB^{+}13] is an extremely scalable stream processor that offers similar qualities as Flink and Apex, e.g. state management and exactly-once semantics. Millwheel and FlumeJava [CRP^{+}10] are the execution engines behind Google's **Dataflow** cloud service for data processing. Like other Google services and unlike most other systems discussed in this section, Dataflow is fully managed and thus relieves its users of the burden of deployment and all related troubles. The Dataflow programming model [ABC^{+}15] combines batch and stream processing and is also agnostic of the underlying processing system, thus decoupling business logic from the actual implementation. The runtime-agnostic API was open-sourced in 2015 and has evolved into the Apache **Beam** [Apa18c] project (short for B̲atch and stre̲a̲m̲) to bundle it with the corresponding execution engines (*runners*): As of writing, Apex, Flink, Spark and the proprietary Google Dataflow cloud service are supported. Another fully managed stream processing system is **IBM Infosphere Streams** [BBF^{+}10]. In contrast to Google Dataflow which is documented to be highly scalable (quota limit for customers: 1 000 compute nodes [Goo16]), it is hard to find evidence for high scalability of IBM Infosphere Streams; performance evaluations made by IBM [IBM14] only indicate it performs well in small deployments with up to a few nodes. **Photon** [ABD^{+}13] is a system developed by Google to join distributed data streams under exactly-once processing semantics. In contrast to the other stream processors discussed here, Photon is designed for geographically distributed deployments. Thus, Photon exhibits relatively high end-to-end latencies in the order of several seconds on average and is specifically designed to cope with infrastructure degradation and failure (such as data center outages) in automatized fashion. **Quill** [CFG^{+}16] is a distributed platform that supports temporal and relational data analysis of historical and streaming data. It uses the analytics library **Trill** [CGB^{+}14] for relational and temporal data analysis. Since Trill relies on micro-batching to compute incremental output over data streams, latency is typically in the order of seconds, even though it is configurable through batch size. **Concord** [Bro15] is a proprietary stream processing framework designed around performance predictability and ease-of-use. To remove garbage collection as a source of possible delay, it is implemented in C++. To facilitate isolation in multi-tenant deployments, Concord is tightly integrated with the resource negotiator Mesos. Flume [Apa16e] is a system for efficient data aggregation and collection that is often used for data ingestion into Hadoop as it integrates well with HDFS and can handle large volumes of incoming data. While it is not designed for complex topologies, Flume does support simple operations such as filtering or modifying incoming data through **Flume Interceptors** [GMSS15, Ch. 7] which may be chained together to form a low-latency processing pipeline. The list of distributed stream processors goes on, but we consider systems out of scope that have been discontinued (e.g. Muppet [LLP$^+$12], S4 [NRNK10]), or focus on mobile computing (e.g. Sonora [YQC$^+$12]).

2.5.3 Design Decisions & Trade-Offs

Table 2.3 sums up the properties of those systems in direct comparison which we covered in-depth in the last section. Storm provides low latency, but does not offer ordering guarantees and is often deployed providing no delivery guarantees at all, since the per-tuple acknowledgement required for at-least-once processing effectively doubles messaging overhead. Stateful exactly-once processing is available in Trident through idempotent state updates, but has notable impact on performance and even fault tolerance in some failure scenarios. Samza is another native stream processor that has not been geared towards low latency as much as Storm and puts more focus on providing rich semantics, in particular through a built-in concept of state management. Having been developed for use with Kafka in the Kappa Architecture, Samza and Kafka are tightly integrated and share messaging semantics; thus, Samza can fully exploit the ordering guarantees provided by Kafka. Spark Streaming effectively unifies batch and stream processing and offers a high-level API, exactly-once processing guarantees, and a rich set of libraries, all of which can greatly reduce the complexity of application development. However, being a native batch processor, Spark Streaming loses to its contenders with respect to latency [CDE$^+$15].

For a discussion of the different notions of event and processing time (cf. Section 2.4.2) for the individual systems, we refer to [Aki15] [Aki16, Nie17] [Apa17]. We consider the intricacies of event and processing time to be out of scope, because InvaliDB directly uses object versions as logical event timestamps (cf. Section 3.1) and therefore does not rely on the event time semantics of the underlying stream processor.

2.5.4 Stream Processing: Summary & Discussion

With current technology, it has become feasible to build Big Data analytics pipelines that process data items as they arrive. However, processing latency is involved in a number of *trade-offs* with other desirable properties such as throughput, fault tolerance, reliability (processing guarantees), and ease of development. Throughput can be optimized by buffering data and processing it in batches to reduce the impact of messaging and other overhead per data item, whereas this obviously increases the in-flight time of individual data items. Abstract interfaces hide system complexity and ease the process of application development, but sometimes also limit the possibilities of performance tuning. Similarly,

	strictest guarantee	achievable latency	state management	processing model	backpressure mechanism	ordering guarantees	elastic scalability
Storm	at-least-once	$\ll 100$ ms	yes	one-at-a-time	yes	no	yes
Trident	exactly-once	< 100 ms	yes (small state)	micro-batch	yes	between batches	yes
Samza	at-least-once	< 100 ms	yes	one-at-a-time	not required	within stream partitions	no
Spark Streaming	exactly-once	< 1 second	yes	micro-batch	yes	between batches	yes
Flink (streaming)	exactly-once	< 100 ms	yes	one-at-a-time	yes	within stream partitions	yes

Table 2.3: Storm/Trident, Samza, Spark Streaming, and Flink’s streaming engine in direct comparison.

rich processing guarantees and fault tolerance for stateful operations increase reliability and make it easier to reason about semantics, but require the system to do additional work, e.g. acknowledgements and state replication. Exactly-once semantics are particularly desirable and can be implemented through combining at-least-once guarantees with either transactional or idempotent state updates, but they cannot be achieved for actions with side effects such as sending a notification to an administrator.

Through general-purpose stream processing frameworks, data can be accessed in the arguably most generic of ways: through writing application code. At the same time, though, neither declarative query languages nor collection-based change notifications nor self-maintaining queries are available out-of-the-box. While collection-based real-time queries can be implemented using stream processing technology (as detailed in Chapter 3 and Chapter 4), the complexity involved in doing so is prohibitive in many use cases.

2.6 Push-Based Access in Data Management: Historical Overview & Discussion

Unsynchronized access to file systems, network databases (CODASYL), and hierarchical databases (IMS) [FS76] represented the state-of-the-art mechanisms for data storage and retrieval before the advent of relational database systems. However, data management has come a long way since then: Figure 2.10 provides a coarse-grained overview over the development of data management systems from 1970 until today.

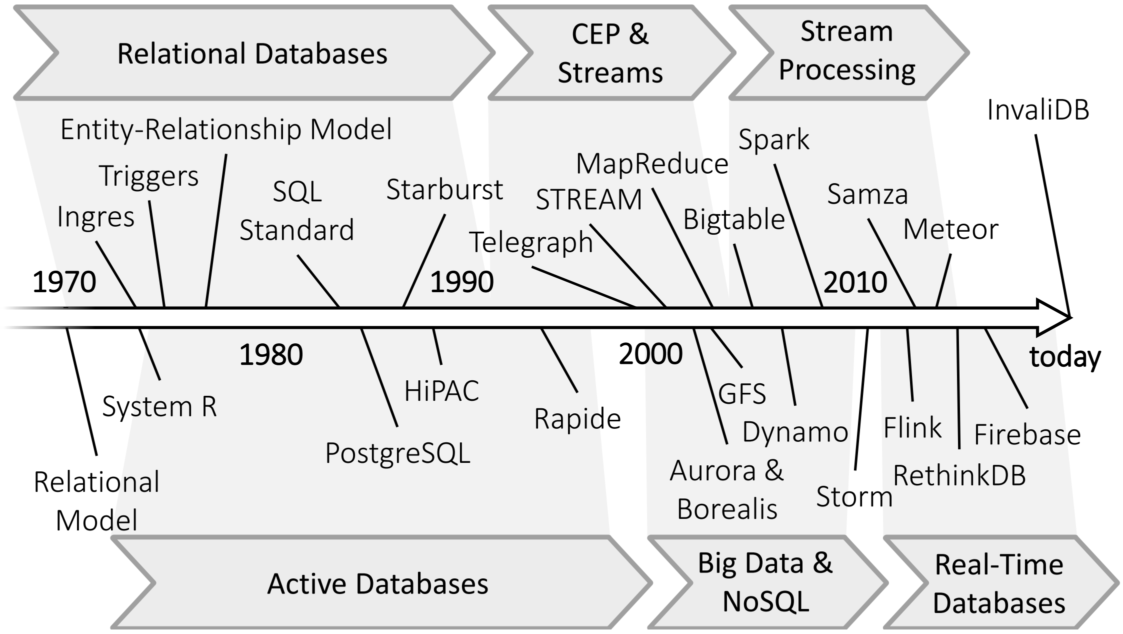


Figure 2.10: Over the last five decades, different classes of data management systems have been in the focal point of research interest.

After the introduction of the relational model in 1970 [Cod70], Ingres [SHWK76] and System R [CAB+81] followed shortly thereafter as the first implementations of relational database systems. In the following years, the formalization of data modeling (e.g. through the Entity-Relationship Model [Che75]) and standardization through both ANSI [ANS86] and ISO [Tec87] helped to increase the popularity of relational systems further. While triggers as the first active mechanisms were proposed in 1975 already [EC75], active databases such as Starburst [SCF+86], HiPAC [DBB+88], and Postgres [SR86] did not emerge before the mid-1980s. Systems like Rapide [San93], Telegraph [CCD+03], STREAM [MWA+03], and Aurora/Borealis [ÇAA+16] in the 1990s and early 2000s eventually took the acknowledgment of data in motion one step further by introducing dedicated concepts for data streams and event sequences, thus deviating from the relational model centered around static data collections. The explosion of user-generated data at companies like Google and Amazon in the early 2000s finally sparked development of several data management system classes that parted with the relational model altogether [GWFR16]. NoSQL data stores like BigTable [CDG+06] and Dynamo [DHJ+07] revolutionized the way that distributed data stores were designed, favoring high scalability and fault tolerance over query expressiveness and compliance with existing standards. Similarly, the Google File System (GFS) [GGL03] and MapReduce [DG04] pioneered storage and batch processing of semi-structured and unstructured huge data volumes and thus turned Big Data management and analytics into hot research topics. Around 2010, stream processing frameworks like Storm [TTS+14], Stratosphere (later renamed to Flink) [ABE+14], and Samza [NPP+17] shifted the research focus from maximizing throughput to also achieving low latency for data-intensive applications at scale. With the growing popularity of interactive and collaborative applications in recent times, real-time databases finally received some attention as well as, because they effectively combine the collection-based semantics of traditional databases with the push-based delivery mechanisms known from stream-based systems. However, first-generation real-time database systems like Meteor [Met18], RethinkDB [Ret16], and Firebase [Fir16] exhibited critical design flaws in query expressiveness, scalability, and fault tolerance, resulting in limited usefulness for large-scale industry applications. In this thesis, we present InvaliDB as a second-generation real-time database system design that removes critical constraints present in the first-generation systems.

Discussion

The ability to notify clients of changes to their critical data has become an important feature for both data storage systems and application development frameworks. Table 2.4 sums up the current landscape of systems that provide push-based data access in one form or another. Traditional (SQL) *database management systems* provide a wealth of features for applications based on request-response interaction, but maintaining query results on a per-user basis is not what they have been designed for. Few SQL systems provide active features beyond triggers and existing functionality for query result maintenance is almost exclusively employed for optimizing pull-based query performance (e.g. materialized views or change notifications).

	Database Management	**Real-Time Databases**	**Data Stream Management**	**Stream Processing**
Primitive	persistent collections		ephemeral streams	
Processing	one-time	one-time + continuous	continuous	
Access	random	random + sequential	sequential (single-pass)	
Data	structured			structured, unstructured

Table 2.4: An overview over system classes providing push-based data access.

Data stream management systems and *stream processing* engines are push-based, expressive, and scalable at the same time, but do not provide collection-based query semantics; instead, they rely on the notion of data streams as the basic primitive. *Real-time databases* combine the data model of traditional databases with the access model of stream-oriented systems. Current implementations, however, are either unscalable or avoid the complex queries that make them so appealing in theory: In consequence, developers often find themselves evaluating complex performance trade-offs or compensating for missing query expressiveness when building reactive applications on top of a state-of-the-art real-time database. Since existing approaches are designed to replace (rather than amend) existing pull-based systems, the benefits of using a real-time instead of a traditional database always have to be weighed against the comparative lack of pull-based features and overall maturity. In order to make the concept of real-time databases more practically relevant, it is necessary to remove these limitations.

In the remainder of this thesis, we will therefore present a real-time database design that is legacy-compatible, provides high read and write scalability, and is suitable for expressive real-time queries at the same time (cf. Chapter 3). We will then provide experimental evidence to support our claims (cf. Chapter 4) and describe a concrete application scenario to illustrate that our approach is practical for existing pull-based systems (cf. Chapter 5).

3 InvaliDB: A Scalable Design for Opt-in Real-Time Queries

"Life does not get better by chance, it gets better by change."

—Jim Rohn

In Chapters 1 and 2, we postulated four pivotal challenges of building real-time databases and surveyed the state of the art in related technology. While we found that many systems do acknowledge a need for reactivity on the database side by pushing updated information to the client, no collection-based real-time query implementation addresses all of the challenges at the same time. Throughout the rest of this thesis, we will conceive, implement, and evaluate a push-based real-time query mechanism that solves all of the aforementioned challenges in a comprehensive fashion.
In this chapter, we therefore present InvaliDB, a system design that provides push-based real-time queries over database collections on top of existing pull-based databases. Thus, we address three out of the four mentioned challenges in this chapter: scalability (C_1), query expressiveness (C_2), and support of legacy systems (C_3). An experimental evaluation of InvaliDB and the remaining challenge of an abstract query API (C_4) will be covered in Chapter 4 and Chapter 5, respectively.

In Section 3.1, we explore the context in which InvaliDB is employed and the semantics which it follows, emphasizing data model, access control, and ordering as well as consistency guarantees for real-time queries. In Section 3.2, we then detail the messaging layer that decouples communication between the pull-based database on the one side and InvaliDB on the other. To promote a clear understanding of the intended system behavior, we first lay out the different message types that are required to facilitate efficient maintenance of sorted results. Then, we discuss the different communication pipelines over which these messages are exchanged. Until this point in the chapter, we treat InvaliDB as a black box and only describe received input and generated output. Starting with Section 3.3, we focus on InvaliDB's internals and explain how computation and state management required for these messages are distributed horizontally *across machines*. In doing so, we explicate how many concurrent real-time queries and high update throughput are made feasible for sorted and unsorted filter expressions, joins, and aggregations. Last in this section, we turn to the pluggable query engine and describe which components of our design are generic and which components have to be customized for the underlying pull-based database. Finally, we provide a synoptic discussion of our approach in Section 3.4.

3.1 System Model

InvaliDB is a real-time database design that provides push-based access to data through collection-based real-time queries. Its name is derived from one of its usages[1] where it invalidates cached database queries.

Similar to some of the systems discussed in Chapter 2 (e.g. Meteor, RethinkDB, Parse), InvaliDB relies on a pull-based database system for data storage. End users do not directly interact with the database, but instead with application servers that execute queries and write operations on the users' behalf. As an important distinction to state-of-the-art real-time databases, however, InvaliDB separates the query matching process from all other system components: The real-time component (InvaliDB cluster) is deployed as a separate system, isolated from the application servers, and it can only be reached through an asynchronous message broker (the *event layer*). To enable real-time queries, an application server only runs the lightweight *InvaliDB client* which relays messages between the end users, the database, and the *InvaliDB cluster*. The expensive task of real-time query matching, on the other hand, is offloaded to the InvaliDB cluster.

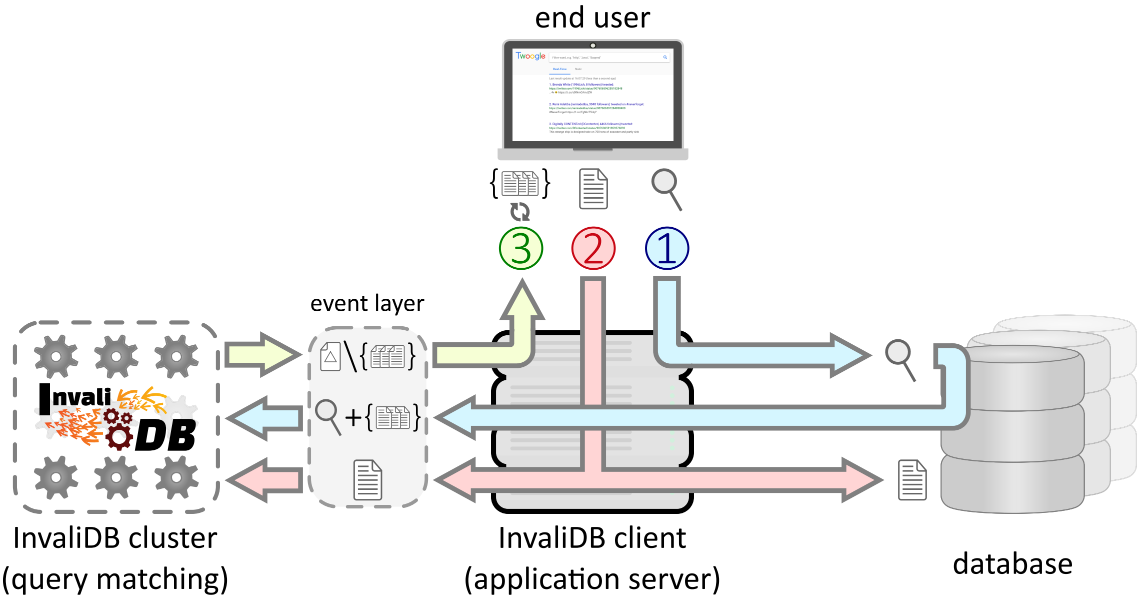


Figure 3.1: InvaliDB strictly separates responsibilities for data storage (database) from real-time query matching (InvaliDB cluster). The InvaliDB client is located at the application server and acts as a broker between these two.

Figure 3.1 sketches out the information flow within the overall architecture. In essence, messages are exchanged between end user, application server, database, and the InvaliDB cluster on the following occasions:

[1] The Quaestor architecture (cf. Section 5.2) implements a scheme for global caching of dynamic content, specifically database query results. Within the Quaestor architecture, InvaliDB is used to detect modifications to query results, so that stale caches can be invalidated in a timely manner.

- **query subscription** (1): In order to subscribe to a real-time query, a web or mobile application user sends a subscription request with a unique identifier[2] to an application server. The application server then executes the query against the database to produce the initial result, i.e. the currently matching data objects. This result and a representation of the query itself are asynchronously handed to the InvaliDB cluster which then *activates* the query. From then on, the InvaliDB cluster maintains an up-to-date representation of the query result.
- **write operation** (2): For every insert, update, or delete operation which is executed at the database, an *after-image* (i.e. a fully specified representation) of the written entity is handed to the InvaliDB cluster. The after-image is then matched against all active real-time queries to detect changes to currently maintained results. Since InvaliDB receives complete copies of the written data objects, it does not require additional database queries for query maintenance (unlike Meteor, cf. page 27).
- **change notification** (3): As a response to a real-time query subscription, the InvaliDB cluster sends out a stream of notification messages. The first notification message for any real-time query contains the initial result; this message is generated on query subscription. All subsequent notifications contain incremental result updates: Whenever a write operation changes any currently active real-time query, the InvaliDB cluster sends a notification to the subscribed application servers which, in turn, forward the notification to the subscribed clients.
- **query cancellation** (not illustrated): Similar to a real-time query subscription, a real-time query cancellation request is asynchronously passed to the real-time component (InvaliDB cluster), so that the query can be *deactivated* and does not consume further resources. No database interaction is required for query deactivation.

In the remainder of this section, we lay out the context in which InvaliDB can be deployed. By doing so, we present assumptions that our system design is based on and further clarify the semantics which it follows.

3.1.1 Fault Tolerance, Scalability & Multi-Tenancy

InvaliDB promotes a separation of concerns between pull-based OLTP workloads and push-based real-time workloads. In this section, we briefly describe consequences and implications of this characteristic of our design.

Fault Tolerance Through Isolated Failure Domains. As one of the main advantages of the strict segregation between OLTP and real-time processing in our proposed design, overload or failure of one component does not compromise availability of the other: In particular, a failing InvaliDB cluster will simply stop responding to application servers, but

[2] For every subscription request, the client generates a unique identifier which is used by the application server to tag the individual change notifications. Thus, the client knows to which real-time query subscription an incoming change notification belongs, even though all subscriptions share the same connection.

will do no further harm. This is an important distinction to systems that burden the application servers with query matching: As we found in Section 2.3, for example, Meteor and RethinkDB ultimately crash when the real-time subsystem becomes a performance bottleneck.

Resource Isolation & Scalability. Another prime benefit of our segregated system design is that resources for OLTP workloads and those for real-time workloads can be scaled independently: Pull-based OLTP performance can be improved by adding application servers or scaling out the database system and, likewise, additional processing nodes can be assigned to the InvaliDB cluster in order to increase sustainable real-time workload (i.e. the number of real-time queries or write throughput). In contrast, many of the systems discussed in Chapter 2 are inherently unscalable, because they colocate query matching with OLTP processing within the application server (e.g. Meteor, RethinkDB) or the database itself (e.g. change notification mechanisms in Oracle, SQL Server, or PostgreSQL).

Multi-Tenancy. In our proposed system design, an application does not necessarily have to be deployed with a dedicated InvaliDB cluster. Instead, different applications (tenants) can share a single, comparatively more powerful InvaliDB cluster. Therefore, real-time queries can be understood as a “Realtime-as-a-Service” offering which is provided by the InvaliDB cluster and consumed by application servers through the event layer interface. Providing real-time queries with a single shared InvaliDB cluster can be preferable over using small dedicated clusters, because it optimizes efficiency through resource pooling, increases resilience against load spikes, and is also easier to deploy and monitor.

3.1.2 Real-Time Queries

InvaliDB’s real-time queries are designed to resemble ad hoc database queries in many aspects, specifically query semantics. However, given the fundamental differences between the push-based and the pull-based access paradigm, both query types are used in different ways. In the following, we therefore elucidate how real-time queries can be defined, how user permissions can be enforced, and what correctness guarantees are provided in our conceptual framework.

Implicit Ordering by Primary Key. Evaluating database queries sometimes involves a certain degree of indeterminism. While this may be acceptable in some applications, it can have disastrous effects in others: Using InvaliDB for cache invalidation (cf. Chapter 5) is only possible, for instance, if InvaliDB’s query evaluation is perfectly aligned with the pull-based query engine underneath. As an illustration of this issue, consider a sorted SQL query: Since `ORDER BY` clauses do not guarantee stable sorting order unless the sorting key is unique for every result entry [Tec92], entities may appear in arbitrary order within the query result when they have identical sorting keys. For queries that contain the equivalent of an offset or limit clause, even result membership can thus be subject to indeterminism, whenever result boundaries are ambiguous. In order to guarantee that query evaluation in the pull-based database and within InvaliDB produce the same output, queries therefore have to be defined in such a way that their results are unambiguous. For sorted queries, in particular, we therefore assume that all sorting keys are made unique (e.g. by implicitly adding the primary key as the last component).

Immutable Query Parameters. InvaliDB's real-time queries are immutable, i.e. it is not possible to update the parameters of an active real-time query: Once a subscription request has been sent, the subscriber is only able to either cancel it or keep listening for notifications. Changing the definition of a query is thus only possible by resubscribing with different parameters. It is important to note, however, that we assume query immutability in this work to reduce overall system complexity; in reality, changes in query context during execution may contradict this assumption. For example, revoking certain user rights should be immediately reflected in the data provided through active real-time queries. In our system model, a user's active subscriptions therefore have to be canceled after permission updates (see next paragraph).

Access Control. InvaliDB's real-time queries are designed to enforce access control on the level of individual records. To this end, every real-time query subscription carries a representation of the executing user's permissions (e.g. the user ID and a list of associated roles). Likewise, access control lists are attached to every incoming write operation to restrict or explicitly allow access to certain users and user groups[3]. Irrespective of the actual matching status, a real-time query may only produce a match for a given entity representation when the user has read access. Thus, the check of permissions can be understood as an additional filter predicate that an incoming object needs to satisfy. It should be noted, however, that user permissions (unlike query predicates) may be updated during the lifetime of a real-time query. Since real-time queries (and thus also the associated permissions) cannot be updated, though, a change of user permissions will have no effect on active real-time queries. To effectively propagate updated permissions to the InvaliDB cluster, all affected real-time queries have to be canceled and resubscribed with fresh parameters.

Eventually Consistent Query Results. InvaliDB can be seen as a replicated storage, since it receives a copy of database state (the initial query result) and subsequently applies updates to it (after-images). But since all communication is purely asynchronous[4], InvaliDB may receive after-images delayed or skewed (compared with the order in which the corresponding write operations arrive at the database). In consequence, a query result maintained by InvaliDB may diverge temporarily from actual database state. When InvaliDB has applied the same write operations as the database, however, the maintained result of an active real-time query (within InvaliDB) is identical to the result of the corresponding pull-based query (run against the database). Similar to other asynchronously replicated

[3] Sections 5.1.2 and 5.2.2 further explore how access control can be implemented in a system using InvaliDB.

[4] As detailed in Section 3.2, the application server never waits for input from InvaliDB: In essence, the application server passes after-images to the InvaliDB cluster when it receives them from the database and passes change notifications to the end user as soon as it receives them from the InvaliDB cluster.

systems, InvaliDB thus facilitates a notion of correctness that corresponds to *eventual consistency* [BG13].

3.1.3 The Backing Database System

InvaliDB is designed for query expressiveness on par with that of aggregate-oriented [SF12] NoSQL document stores such as MongoDB. Since these systems rely on nesting to express relationships between entities, they typically[5] provide only very limited or no support for join queries. Accordingly, InvaliDB is designed to enable real-time queries over single collections, like the state-of-the-art real-time databases discussed in Section 2.3. In contrast to these systems, though, InvaliDB also enables real-time join queries and real-time aggregations (cf. Section 3.3.2).

Data Model. Since write operations may arrive out-of-order, InvaliDB needs to identify (and ignore) records for which newer versions have already been processed; to ensure eventual correctness of maintained query results, InvaliDB only processes new versions and filters out already-processed ones. To make this feasible, we assume all data to be *versioned*[6], so that it is always trivial to pick the most recent version among different representations of an entity. For the same reason, we assume that there is an arbitrary (but finite) upper bound for propagation delays, so that last-seen object versions do not have to be retained indefinitely. To further reduce complexity, we assume all write operations to be final and non-conflicting; we do not consider scenarios where conflicts have to be resolved or write operations have to be rolled back. We also make two assumptions regarding primary keys: First, primary keys are always *final*, meaning they cannot be changed once assigned; this characteristic is mandatory, because InvaliDB's workload distribution scheme assumes primary key hashes to be static for database entities (see Section 3.3 for details). Second, primary keys must not be reused to prevent InvaliDB from mistaking entities for one another: Since InvaliDB remembers deleted items (and ignores inserts and updates for them), inserting an entity with the primary key of a previously deleted one may lead to incorrect behavior.

Write Propagation. A real-time query entails at most one single database request on subscription (cf. Section 3.2.4) and is otherwise *self-maintainable* [QGMW96], i.e. its result can be kept up-to-date with only the incoming write stream. To avoid contacting the database during query matching, however, InvaliDB requires a complete (i.e. fully specified) copy of any written data object[7]. Producing these may incur some processing overhead at write time, depending on a variety of parameters. For example, set-oriented

[5] See our in-depth database survey [GWFR16].

[6] Versioning data records is a standard feature in many database systems (e.g. PostgreSQL [Pos18, Sec. 5.4], SQL Server [MGB17], Oracle NoSQL Database [Ora18b, Sec. 9]) and can be implemented within the application server, if the underlying database system does not support it (cf. Section 4.3).

[7] A delete operation provides the identifier of the deleted entity and `null` as after-image.

writes[8] can only be processed correctly when all affected database records are identified and assembled by the application server and provided to InvaliDB. Similarly, a write operation has to complete before the corresponding after-image can be sent to InvaliDB, if the primary key (or any auto-generated attribute) is assigned by the database or if it is executed as part of a transaction and therefore might be aborted by the database. To shield against bursty write workloads and write hotspots in general, the application server enforces a *per-object propagation rate limit* by *collapsing* after-images of the same entity. For example, a rate limit of 10 write operations per second translates to a minimum delay of 100 ms between after-images for the same entity. When a particular entity is updated three times within 50 ms under this rate limit, the application server will propagate the first update immediately and the third update 100 ms later; the second update will not be propagated at all.

Query Execution. While InvaliDB serves every subscription to an active real-time query without database interaction, it requires the initial result whenever the subscribed query is not being maintained already (see Section 3.2.4 for details). However, InvaliDB neither regulates the overall number of real-time queries in the system nor the number of distinct queries that an application server subscribes to. Instead, InvaliDB only employs rate limiting on the level of individual queries to bound the impact of runtime errors that require resubscription (cf. poll frequency rate limit in Section 3.3.2). Application servers are consequently able to incur substantial load on themselves and on the underlying database system by registering excessively many *distinct* real-time queries over a short period of time. In this work, we assume that application servers avoid such self-inflicted overload scenarios by enforcing a rate limit on pull-based queries for the initial results. Since InvaliDB is designed to serve many application servers with possibly heterogeneous hardware profiles in a multi-tenant environment, the performance characteristics of the subscribed application servers and the capabilities of the backing data store are transparent to InvaliDB.

3.2 The Event Layer: Decoupling Real-Time & OLTP Workloads

In Section 3.1, we introduced the event layer as a message broker that decouples application servers from an InvaliDB cluster. We also briefly presented its different communication pipelines for write propagation, query management, and real-time notifications. In this section, we detail the three event layer pipelines and their respective characteristics such as requirements on ordering or delivery guarantees.

We start in Section 3.2.1 by describing the intended messaging semantics. In more detail, we explicate how query result modifications are encoded in the different types of noti-

[8] As an example, consider the following SQL command which may affect arbitrarily many database records: `UPDATE employee SET income += 500 WHERE name = 'Smith'`.

fication messages and how client-side result maintenance (cf. self-maintaining[9] queries) can be implemented on their basis. In Section 3.2.2, we then address the write pipeline that propagates after-images from the application servers to the InvaliDB cluster. Next in Section 3.2.3, we focus on information flow in the opposite direction and consider the notification pipeline over which initial results and result changes are pushed from the InvaliDB cluster back to the application servers. In Section 3.2.4, we finally turn to the query pipeline and describe how query subscriptions are established, how they are canceled, and how they are related to change notification streams.

3.2.1 Messaging Semantics

As stated in Section 3.1, every real-time query produces a sequential stream of change notifications which enables the receiving party to maintain the query result: For every result change, the InvaliDB cluster sends out one or several notifications containing the information required to implement these changes. Further, every change notification relates to exactly one database entity that is or was part of the result. To make explicit how the transmitted entity relates to the query result, every change notification carries a **match type** attribute. The elementary match types are as follows:

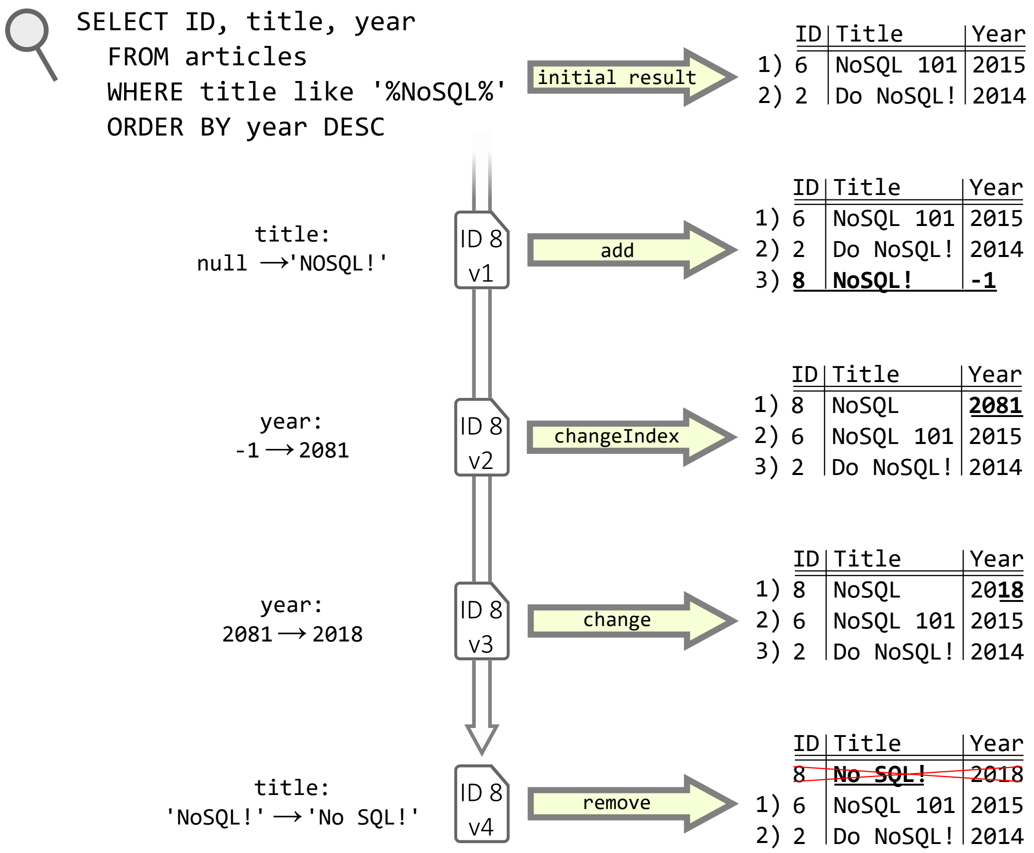
- `add`: An entity entered the result, i.e. it did not match before and is matching now.
- `change`: A matching entity was updated and remains a match.
- `changeIndex` (for sorted results only): A matching entity was updated and changed its position within the query result (subsumed by `change`).
- `match`: The entity matches the query (either `add`, `change`, or `changeIndex`).
- `remove`: The entity was a match before, but is not matching any longer.

The `add`, `change`, `changeIndex`, and `remove` match types relate to collection-based query semantics as they encode incremental result alterations. The `match` match type, in contrast, is intended for stream-oriented query semantics: Since the match type `match` does not reflect the previous matching state of the written entity, the InvaliDB cluster does not maintain result state for these queries unless required for matching (e.g. when specified with a limit or offset). It is only assigned when the client specifically requests this more generic match type; by default, the most specific match types are provided.

As an illustration of the different match types, consider the query in Figure 3.2: The query selects blog posts whose title contains the keyword `'NoSQL'` and it sorts them by publication year in descending order. The initial result in the example only contains two articles from the years 2015 and 2014, respectively. Next, a new article is entered into the system with title `'NoSQL'` and the default year of `-1`. This triggers an `add` notification and thus ap-

[9] To avoid confusion, we would like to point out that *self-maintaining* queries are an abstraction for real-time queries that hides the complexity of result maintenance from the application developer (cf. Section 5.3.1). In contrast, a real-time query is called *self-maintainable*, if its result can be kept up-to-date without access to the underlying database (cf. Section 2.2.3).

pends the article with ID 8 to the result. A later update changes the article's year to 2081 and thus provokes a `changeIndex` notification: The article is moved to the first position of the query result. However, the author notices a typo in the year value and corrects it to 2018. This alteration entails a simple `change` notification as the article is modified, but its position within the result remains unchanged. Finally, the author spots and rectifies another typo, this time relating to the title: Changing the article's title from `'NoSQL'` to `'No SQL'` causes a `remove` notification as the article does not satisfy the matching condition anymore and therefore leaves the query's result.



```
SELECT ID, title, year
  FROM articles
  WHERE title like '%NoSQL%'
  ORDER BY year DESC
```

	ID	Title	Year
1)	6	NoSQL 101	2015
2)	2	Do NoSQL!	2014

	ID	Title	Year
1)	6	NoSQL 101	2015
2)	2	Do NoSQL!	2014
3)	**8**	**NoSQL!**	**-1**

	ID	Title	Year
1)	8	NoSQL	**2081**
2)	6	NoSQL 101	2015
3)	2	Do NoSQL!	2014

	ID	Title	Year
1)	8	NoSQL	20**18**
2)	6	NoSQL 101	2015
3)	2	Do NoSQL!	2014

	ID	Title	Year
	~~8~~	~~No SQL!~~	~~2018~~
1)	6	NoSQL 101	2015
2)	2	Do NoSQL!	2014

Figure 3.2: The different match types correspond to different kinds of result alterations.

There are two different approaches for transferring the **initial result** to the real-time query subscriber. In this work, we represent the initial result as a *sequence* of `add` events, so that change notifications for the initial result as well as the incremental ones for result alterations relate to single database entities, respectively. Since initial and incremental change notifications are both scoped to individual entities here, they can be processed in the same fashion[10]. This is not the case with the second approach where the initial result is transferred in the form of an actual collection with the first change notification. While we think the second approach is useful to illustrate the general idea (see for example Figure 3.5 on page 72), we consider it less practical for two reasons. First, it requires

[10] The subscriber can discern initial and incremental change notifications through the `initial` flag (cf. parameters on page 68) which is only true for events that represent part of the initial result.

handling initial and incremental notifications separately, because the initial notification carries a *collection* of records whereas all following notifications carry *individual* records. In other words, it makes notification processing more complex for the client than the first approach where all notifications carry individual records. As a second reason, the current system design does not even allow providing the initial result as a contiguous collection for certain[11] queries, because the result maintenance procedure and, in particular, the result itself are partitioned within the InvaliDB cluster, so that no single matching node in the InvaliDB cluster has a notion of the complete result. For details, see Section 3.3.

After-Images, Before-Images & Attribute-Level Change Deltas

The InvaliDB cluster receives representations of database entities in the form of after-images. These after-images are delivered through the write stream (cf. Section 3.2.2) or as part of query results (e.g. on subscription, cf. Section 3.2.4). On receiving an entity representation, the cluster generates one or several change notifications for all affected real-time query subscriptions. However, the InvaliDB cluster only receives fully specified **after-images** of written entities (and no information on *which* attributes were updated specifically). In consequence, the naïve way of change propagation is to provide a fully specified after-image with every change notification.

When an after-image is received for an already-known entity, the already-known version reflects the entity *before* the write operation was applied and the more recent version represents the entity immediately *thereafter*. Thus, these two representations can be interpreted as *before-image* and *after-image* of the entity, respectively. A **before-image** can only be derived when the written entity is already part of the query's result; this corresponds to change notifications with match type `change`, `changeIndex`, and `remove`. In the case of `add` notifications, only the after-image (and no prior version) of the written entity is available by definition; the entity is just becoming part of the result.

The before-image can be provided to the subscriber as a change notification property in addition to the after-image. This can be convenient, for instance, when the receiving application displays the old and the new version of the written entity side-by-side. However, it is not necessary to send two fully specified records for this purpose. Instead, the same information can also be encoded in the after-image plus **attribute-level change deltas** that capture the difference between before- and after-image; these can be computed within the InvaliDB cluster before sending out the notification. Assuming the subscriber maintains the query result, it is even possible to strip the after-images from all but the `add` notifications: For all other match types, the subscriber already knows the current record version (before-image), so that attribute-level change deltas are sufficient to derive the after-image from local state. While complete records are arguably more intuitive

[11] Specifically, unsorted filter queries and aggregation queries with grouping key (cf. Section 3.3.2) are maintained in partitions.

to process than attribute-level change deltas, they also bloat payload traffic by including information that has not changed (and has already been delivered with previous notifications). When entities are large and changes are frequent, transmitting only attribute-level change deltas instead of complete records can thus reduce network traffic significantly without reducing entropy[12].

For illustration, consider an e-commerce application where product information and current warehouse stock are stored in the same entity; whenever the stock value for a particular product changes (i.e. every time it is sold or replenished), a change notification will be generated and sent to all subscribers. When fully specified after-images are included in every change notification, static information such as the product description will also be transmitted on every update. Therefore, the size of the product description and the frequency of updates may dominate the network footprint of a real-time query. However, the effective change that needs to be transmitted is only a counter update. By using attribute-level change deltas, unaltered attributes are stripped from the generated change notification before it is sent to the subscriber.

Change Notification Properties

Change notifications encode changes to the result of the underlying real-time query. Further, every change notification is related to exactly one database entity ("the related entity") and carries its characteristics and result context (e.g. position in the result). These information are encoded in the following properties:

- **id** (optional): a subscription identifier. If this attribute is present, the change notification is only processed by the application server responsible for the subscription (e.g. for changes representing an initial result that is designated for a specific subscriber or for query renewal requests, cf. page 80); a change notification without subscription ID will be processed by all application servers (specifically: incremental change notifications that are intended for all subscribers of the query).
- **match type**: The match type encodes what kind of result alteration the change notification represents. As illustrated in Figure 3.2 on page 65, it thus explicitly tells the subscriber how the maintained result needs to be updated.
- **before-image** and **after-image**: a representation of the related entity from *before* and *after* write execution, respectively.
- **change delta**: a representation of the relative change that happened to the related entity (i.e. new, updated, and removed properties).
- **operation**: the type of operation by which the entity relating to the notification was altered (`insert`, `update`, or `delete`; `none` if unknown or not applicable). For an ex-

[12] It should be noted that complete before- and after-images can be exposed to the application developer, independent of whether fully specified entity representations or only change deltas are transmitted: If desired, the InvaliDB client can assemble the complete records in transparent fashion.

ample where neither `insert`, `update`, nor `delete` can reasonably be applied to an event, consider how the last entity in a top-10 query result is pushed out when a new item enters the top-10: While one notification represents the insertion of the new entity itself (`add`), another one represents the entity leaving the result (`remove`). Since the leaving entity was neither inserted, updated, nor deleted, the corresponding notification would be delivered with operation type `none`.

- **index** (for sorted queries only): an integer representing the new position of the related entity within the result.
- **initial**: a boolean value indicating whether this notification represents part of the initial result or not.
- **error** (only on error): If present, this attribute indicates that the subscription was terminated due to an error condition. Ideally, this attribute contains hints on how the application server can handle the error (see for example subscription errors in Section 3.2.4 or query maintenance errors in Section 3.3.2).

By applying the transmitted change notifications according to their respective match types (and indices), the subscriber can maintain an up-to-date representation of the query's result. In Section 5.4.4, we provide a detailed description and example code to illustrate a straightforward algorithm for client-side result maintenance.

Throughout the rest of this section, we describe the different communication channels for message exchange between the application server and the InvaliDB cluster.

3.2.2 Write Pipeline

InvaliDB's write pipeline is similar to a queue where **after-images** are inserted by application servers on the one side and taken out by the InvaliDB cluster on the other. As described in Section 3.1.3, however, InvaliDB does not rely on first-in-first-out (FIFO) ordering; in fact, it does not assume anything at all regarding the order in which after-images arrive. In consequence, InvaliDB's write pipeline can be scaled from one single message queue to many concurrent message queues without any coordination between them.

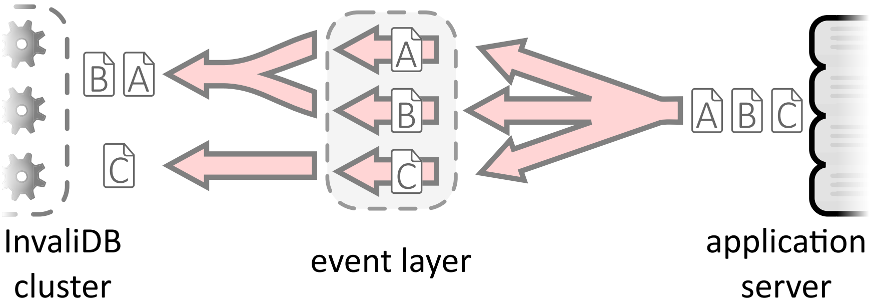


Figure 3.3: The write pipeline is horizontally scalable, because it does not have to be order-preserving: Propagation can be distributed across an arbitrary number of asynchronous message queues within the event layer.

The example in Figure 3.3 shows an event layer where the write pipeline consists of three independent message queues. For publishing write operations, an application server alternates between them (e.g. in Round-Robin fashion) to scatter the write workload evenly; thus, the after-images A, B, and C are not transferred to the event layer sequentially, but to different message queues in parallel. On the other side, consumer processes within the InvaliDB cluster draw after-images from the event layer in arbitrary order: In the example, after-image B is consumed before after-image A and after-image C is consumed concurrently.

Importantly, the number of producers (application servers), consumers (InvaliDB cluster), and message queues (write pipeline) are entirely independent from one another: To support higher write throughput, parallelism of each one can be increased individually. Therefore, (OLTP) write throughput does not become a limiting factor to the event layer. The same holds true for the overall system because of the way that the matching workload is distributed within the InvaliDB cluster (see Section 3.3).

3.2.3 Notification Pipeline

The change notification pipeline can be seen as the counterpart to the write pipeline, since it transports **change notifications** back to the application servers as an immediate response to received after-images. In the context of the write pipeline, the InvaliDB cluster can be seen as a subscriber to after-images that are provided by application servers. For the notification pipeline, roles are reversed: The InvaliDB cluster emits change notifications that are pushed towards the application servers.

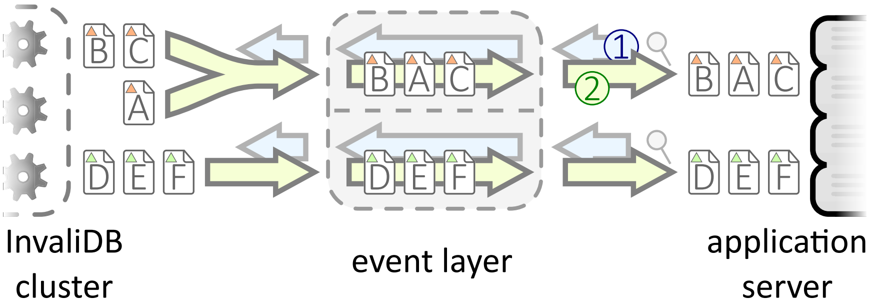


Figure 3.4: The notification pipeline delivers the initial result and incremental result changes for any registered real-time query. Since individual notification messages for the same query depend on one another, the notification pipeline must be order-preserving per query to guarantee correctness.

Figure 3.4 shows, however, that the notification pipeline is significantly more complex than the write pipeline, mainly for three reasons:

1. *Coupling with query pipeline*: Every change notification belongs to a particular real-time query. In consequence, a real-time query subscription request (**1**) is required to open the corresponding change stream, before a change notification can be received (**2**). The process of activating a real-time query is described in Section 3.2.4.

2. *Ordering guarantees*: Every change notification represents a transition of the corresponding query result from one state to another. For some queries (e.g. sorted filter queries, cf. page 68), change notifications are totally ordered (green Δ), i.e. they are emitted in sequence and have to be applied correspondingly. This is the case for D, E, and F in the illustration. Notifications for other queries (e.g. unsorted filter queries) only obey a partial order (orange Δ), so that notifications for unrelated changes can be emitted by concurrent processes within the InvaliDB cluster. In the illustration, B is emitted before C by the same process, so that B also has to be delivered before C on the side of the application servers; for example, B and C could be `add` and `remove` notifications for the same entity. Notification A, in contrast, does not have any dependency to either B or C as it is emitted by a concurrent process; for example, it may relate to a different entity. Since A is thus unrelated to all other emitted notifications, it may occur anywhere in the change stream.
3. *Persistent connections*: The notification pipeline requires a persistent connection between the event layer and the application server, so that result changes can be pushed from InvaliDB to the listening application server. If the event layer is partitioned, an application server therefore has to maintain a connection to every partition that holds a currently subscribed query. This is not required[13] for propagation of after-images.

Due to the above-described complications, notifications cannot just be scattered arbitrarily across any number of concurrent message queues as it is the case with after-images in the write pipeline. However, the change notification pipeline of the event layer can still be partitioned as indicated by the dashed separation line in the illustration: All notifications for one particular query go through the same message queue within the event layer, but different queries can be assigned to different message queues. The change notification pipeline within the event layer is thus horizontally scalable; unlike the write pipeline, though, it is not easily scalable beyond throughput sustainable by a single machine. Seeing that every change stream is consumed by a single-machine subscriber, however, this constraint does not limit overall scalability.

3.2.4 Query Pipeline

As stated earlier, the output of a real-time query is delivered through the *notification pipeline*, while every notification channel is opened and closed through specific requests sent over the *query pipeline*. In more detail, an application server first connects to the event layer and starts listening for change notifications on the corresponding notification channel. Then, it sends a real-time query **subscription request** and waits for change notifications to arrive. In this scheme, the subscription is deliberately delayed until *after*

[13] As an aside, persistent connections between application servers and event layer endpoints are generally desirable for efficiency reasons, even though not strictly required.

the notification channel has been established to exclude the possibility of losing change notifications to race conditions.

The application server keeps listening until the query's internal or external deactivation (cf. internal and external transaction aborts [GD94]). An *external query deactivation* occurs, when the last listening client explicitly unsubscribes by sending a **cancellation request** to the application server which is then forwarded to the event layer. Since the InvaliDB cluster keeps track of all current subscriptions, it will automatically deactivate the query when the last subscription for a real-time query has been canceled. Contrastingly, an *internal query deactivation* is triggered by the InvaliDB cluster itself whenever it encounters a condition that makes further maintenance impossible (cf. query maintenance errors in Section 3.3.2). In this case, an error notification (cf. `error` attribute on page 68) is sent out as the last message on the corresponding notification channel to inform all subscribers of the query's termination.

Possible reasons for implicit query deactivation include errors during query maintenance[14] and query expiration: On activation within the InvaliDB cluster, every query is assigned a *time to live (TTL)* which has to be renewed before expiration to keep the associated subscriptions active. Every application server therefore sends periodic **TTL extension requests** over the query pipeline for every active subscription. This mechanism ensures that real-time queries will be deactivated eventually when subscribers leave without sending a prior cancellation request (e.g. on application server crash). The query TTL can be static or dynamically chosen at runtime; either way, it is injected into the subscription request by the application server and transparent to end users.

Query subscriptions are more complex to handle than cancellations or TTL extensions, because they require the responsible application server to perform different actions, depending on the context at subscription time. Figure 3.5 illustrates the process of registering a real-time query in more detail. Whenever an end user sends a real-time query subscription request to an application server, the application server forwards the subscription request to the InvaliDB cluster immediately (**1**). In addition to the unique subscription ID generated by the end user, the server also injects its own server ID into the request, so that the InvaliDB cluster can keep track of the subscriptions maintained by every individual application server; this knowledge facilitates efficient handling of query maintenance errors (see page 80 for details). If the real-time query is being maintained already, the InvaliDB cluster responds with the initial result (**5**) and the subscription is established without any access to the pull-based database system (i.e. steps **2**, **3**, **4** are skipped). If the result is not already being maintained, on the other hand, the InvaliDB cluster sends a **subscription error** back (**2**), thus indicating that the initial query result is required for query registration. On receiving this error message, the application server then queries the pull-based data-

[14] Whenever the effect of an incoming write operation on a given query's result cannot be determined with certainty, the InvaliDB cluster cannot proceed with maintenance and therefore deactivates the query and informs all subscribers through an error notification. In Section 3.3.2, we use the example of sorted filter queries to discuss how this kind of *query maintenance error* can be handled transparently for end users.

base to assemble the initial result (**3**) and repeats the subscription request (**4**): Since the query result is not missing this time, the query will be successfully registered in InvaliDB and the InvaliDB cluster will send out the current[15] result (**5**). While the result will be received by all listening application servers through the notification pipeline, it will only be forwarded to end users with a pending subscription request. For subscriptions that already have been bootstrapped, the initial result is ignored since incremental notifications suffice to maintain the result in those cases. After emitting the initial result, the InvaliDB cluster only produces incremental change notifications to be received by all subscribers (**6**) until a new subscription request arrives (in which case a full result is emitted to be received by the new subscriber alone) or until the query is deactivated.

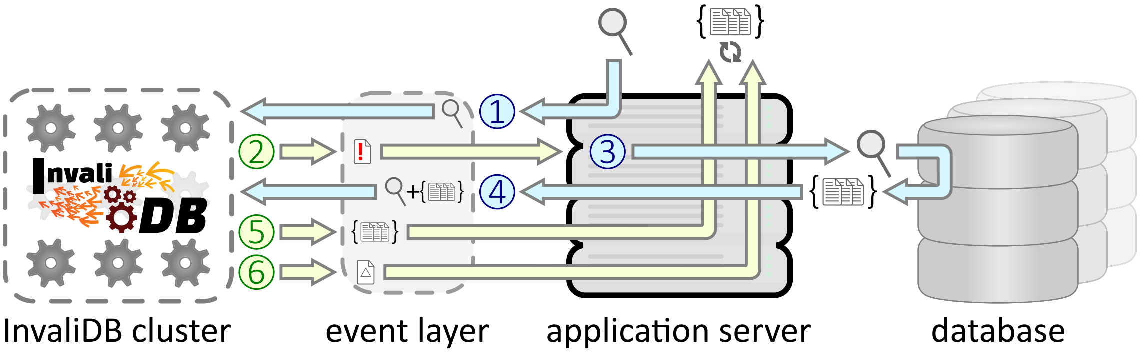


Figure 3.5: InvaliDB avoids contacting the database where possible. Thus, a subscription request is first sent to the InvaliDB cluster without the initial result (**1**), because the query can be directly served when it is already being maintained (**5**/**6**). If this is not the case (**2**), the initial result is queried from the database (**3**) and the subscription request is repeated (**4**), with the initial result attached.

When there already is an active subscription for a particular real-time query, serving an additional end user subscription for the same query is particularly efficient for two reasons. First, the initial result can be retrieved without contacting the database, since the query is already being maintained within InvaliDB. Second, no additional subscription between the application server and the change notification pipeline has to be established, because changes for the query being subscribed are already coming in. When receiving an additional subscription to an already subscribed query, the application server thus simply awaits the initial result, returns it to the new subscriber when it arrives, and delivers the incoming change notifications to the new subscriber as well from this point onwards.

Similar to the write pipeline, the query pipeline is not designed to provide the same order on output as on input. Consequently, subscription, cancellation, and TTL extension requests for the same query can arrive in arbitrary order at the InvaliDB cluster. To simplify message processing, the InvaliDB cluster only accepts messages in the correct order; for example, cancellation and TTL extension requests for yet-unregistered real-time queries

[15] At this point, the query might have been activated by a concurrent subscription issued by another application server. In this case, the result attached to the subscription is processed like a series of write operations, so that change notifications are generated to reflect the most recent version of the result.

are simply answered with error messages. Since the query pipeline does not have to obey any ordering guarantees, message transport can easily be distributed across different machines, like the write pipeline.

Subscription Parameters

Apart from the implicitly injected server ID, there are two subscription parameters that are *mandatory* for every real-time query: a representation of the query (e.g. an SQL statement serialized as a string) and a unique subscription identifier. In addition, there are many *optional* parameters conceivable to make InvaliDB filter out undesired notifications from the change stream or to remove nonessential attributes from the individual notifications.

When the incoming change notifications are directly exposed to application code through an *event stream query*, virtually any parameterization can be reasonable, depending on the use case[16]. In particular, the stream of change notifications may be narrowed down by allowing only notifications with certain match or operation types. Likewise, delivery of the initial result can be disabled when the client application is only awaiting a specific event; thus, InvaliDB can also support a purely stream-based notion of data. For a *self-maintaining query*, in contrast, all notifications are relevant by design and therefore none are filtered out. At the same time, though, before-images are ideally stripped from the individual change notifications, since only after-images are required for result maintenance.

As a final note, subscription parameters can also specify properties of the change stream that do not relate to notification properties at all. For example, they can define how to handle query maintenance errors (cf. *slack* and *poll frequency rate limit* on page 80) or how to *collapse* highly frequent updates on the same entity to reduce event stream throughput (see page 129). In Section 5.3.2, we describe real-time query subscriptions and their parameterization from the user's perspective in more detail.

3.3 Query Processing: Distributed Result Maintenance

In the last section, we presented InvaliDB's event layer as a scalable mechanism for query management and the propagation of write operations and change notifications. In this section, we describe how distributed query processing within InvaliDB is designed to master high write throughput and many concurrent real-time queries at the same time.

In Section 3.3.1, we first explain how InvaliDB achieves read and write scalability for evaluating query predicates through a two-dimensional *workload partitioning* scheme in the

[16] As an illustration, consider the different variants of query caching discussed in Section 5.2.1; they respectively depend on event stream query subscriptions with distinct match type parameterization.

filtering stage. In Section 3.3.2, we then lay out how sorted queries, joins, and aggregations are handled through additional *processing stages*. Addressing the generalizability of our approach, we finally cover the *pluggable query engine* in Section 3.3.3. In doing so, we make explicit what parts of query processing have to be implemented for every instance of the pluggable query engine and which parts can be done in generic fashion and independently of the concrete query engine.

3.3.1 Two-Dimensional Workload Partitioning for Filter Predicates

InvaliDB's real-time query engine has to match all incoming after-images (write pipeline) against all active real-time queries (query pipeline) to produce change notifications as output (notification pipeline). To enable higher input rates than a single machine could handle, InvaliDB partitions data from both the query pipeline and the write pipeline evenly across a cluster of machines: By assigning each node in the cluster to exactly one **query partition** and exactly one **write partition**, any given node is only responsible for a subset of all queries and only a fraction of all written data items.

Figure 3.6 depicts an InvaliDB cluster with three query partitions (vertical blocks) and three write partitions (horizontal blocks). When a subscription request is received by one of the **query ingestion nodes** (**1**), it is forwarded to every matching node in the corresponding query partition; while the query itself is broadcasted to all partition members, the items in the initial result are delivered according to their respective write partitions (i.e. every node receives only a partition of the result). Likewise, any incoming after-image received by one of the **write ingestion nodes** (**2**) is delivered to all nodes in the corresponding write partition as well. To detect result changes, every matching node matches any incoming after-image against all of its queries and compares the current against the former matching status of the related entity. In the example, a change notification is generated by the **matching node** (**3**) that is responsible for the intersection of query partition 2 and write partition 2. Since every matching node only holds a subset of all active queries and only maintains a partition of the corresponding results, processing or storage limitations of an individual node do not constrain overall system performance: By adding query partitions (**$+_{qp}$**) or write partitions (**$+_{wp}$**), the number of sustainable active queries and overall system write throughput can be increased, respectively. In similar fashion, the sustainable rate of data intake can be increased by adding nodes for query and write stream ingestion; these nodes are stateless (and therefore easy to scale out) as they merely receive data items from the event layer, compute their respective partitions by hashing static attributes, and forward the data items to the corresponding matching nodes.

To make workload distribution as even as possible, InvaliDB performs **hash-partitioning** for inbound data from the write and query pipelines. For after-images, the hash value is computed from the *primary key*, because it is the only attribute that is transmitted on insert, update, and delete. In contrast, there is no attribute that is present in all requests re-

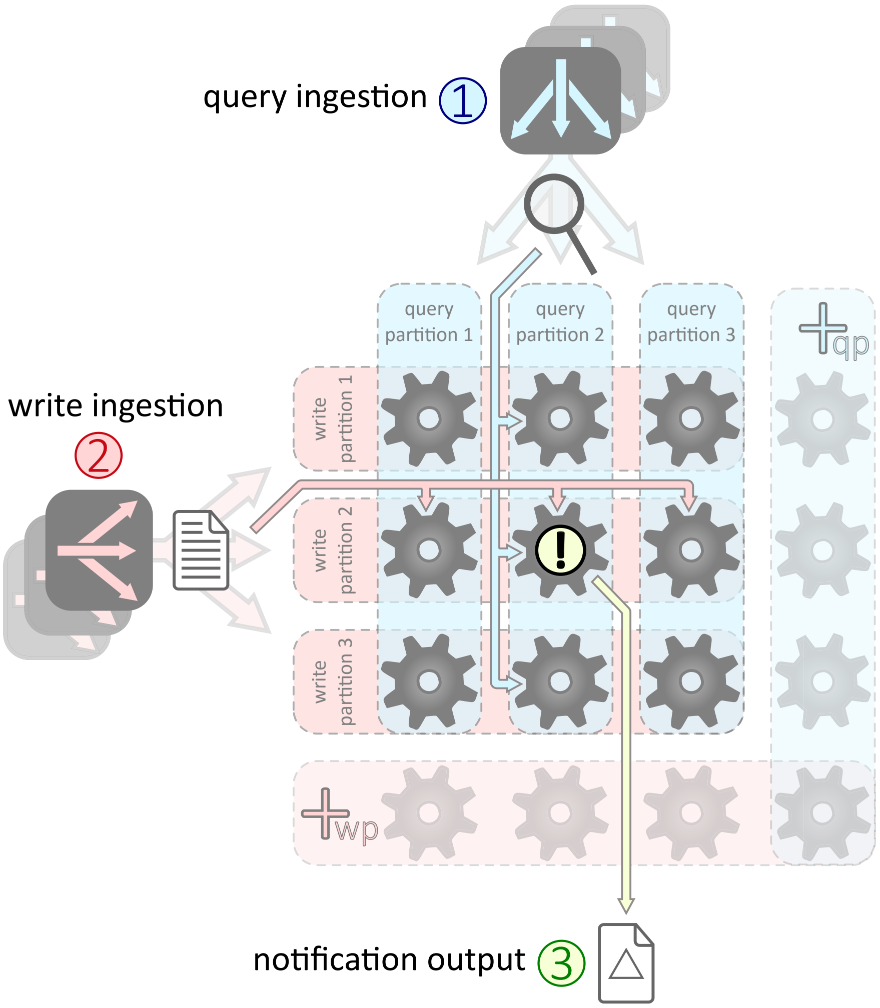


Figure 3.6: InvaliDB partitions both the query and the write pipeline, so that any given matching node is only responsible for matching few queries against some of the incoming write operations.

lated to a particular query. Using the subscription ID to generate the hash value would not violate correctness as the partitioning would be consistent throughout subscription lifetime. However, since subscription IDs are randomly generated, query partitioning would be random as well; in consequence, a single query could be assigned to several (or even all) partitions within the InvaliDB cluster, given it was associated with multiple subscriptions. For the sake of efficiency, an application server therefore computes the hash value from the query attributes when the subscription request is received. Thus, distinct subscriptions to a particular query are always assigned the same hash value and are thus routed to the same partition, even when received by different application servers. Since the hash value cannot be computed for requests other than the subscription requests, though, the application server remembers every hash value for the entire lifetime of a subscription and attaches it to every subsequent[17] request relating to the same subscription.

[17] As a side note, this scheme makes it impossible to assign cancellation and TTL extension requests to a query partition without prior subscription. However, this does not present an error scenario, since cancellations and TTL extensions are only meaningful for active subscriptions as described in Section 3.2.4.

When an application server subscribes to a real-time query, it first executes the query against the pull-based database and then forwards the result to the InvaliDB cluster via a subscription request (see Section 3.2.4). Since write operations happen asynchronously to this procedure, however, there are two **race conditions** that need to be acknowledged in InvaliDB's design. The first race condition is between the write operation and the pull-based query (**write-query race**): The initial result will only reflect the write operation, if the write is applied at the database before execution of the pull-based query. For example, a newly inserted item will only be present in the initial result, when the insert is processed before the query. The second race condition is between the write operation and the real-time query subscription (**write-subscription race**): The responsible matching node will only implicitly match the incoming after-image against the query, if the subscription request arrives before the after-image. Without special precautions, a write operation can thus be missed by a real-time query when it is (1) not reflected in an initial result and when it is also (2) processed by the responsible matching node before the query has been activated through the subscription request.

To avoid missing write operations through unfortunate timing on both these race conditions, InvaliDB employs temporary **write stream retention**: Every matching node stores received after-images and matches them against a new query on subscription. However, since every matching node has only bounded space, long-lasting network partitions can render this scheme infeasible. Likewise, write stream retention does not protect from unbounded propagation delays: In practice, write stream retention time therefore needs to be chosen according to actually observed delays (cf. Section 3.1.3). For reference, the production deployment at Baqend described in Chapter 5 enforces a retention time of few seconds, since our implementation of InvaliDB exhibits consistent end-to-end notification latencies in the realm of few milliseconds with subsecond peaks (see evaluation in Section 5.5.1); these latencies include possible delays of after-image propagation. It should be noted that write stream retention is not only exploited for after-image replay on subscription, but also crucial for **staleness avoidance**, i.e. the ability to detect (and ignore) stale *write operations*. For example, an insert or update operation for a specific item is ignored whenever a delete for the same item has already been received. Likewise, an insert or update is not applied when a higher version already has (see Section 3.1.3).

Depending on the specificities of the implementation, there are different mechanisms for handling failure of individual nodes, network, or other infrastructure components. Since this chapter provides a conceptual overview over InvaliDB and abstracts from concrete technologies, though, we do not go into more detail here. The mechanisms for *fault tolerance* devised in our InvaliDB implementation are described in Chapter 4.

3.3.2 Advanced Processing Stages: Sorted Queries, Joins & Aggregations

By partitioning incoming queries and write operations orthogonally to one another, the **filtering stage** described above distributes the predicate matching workload evenly across all nodes in the cluster. However, while this scheme avoids hotspots, it also prevents capturing the context between different items within the result: As every matching node holds result partitions, changes of an individual result can only be registered on a per-record basis. In more detail, changes relating to the sorting order cannot be detected and queries that aggregate values from different entities (e.g. to compute an average) or queries that join data collections cannot be handled, either.

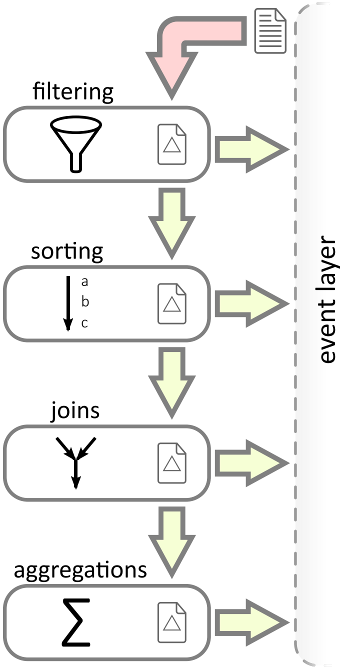


Figure 3.7: Real-time query matching in InvaliDB is organized in several stages, each of which consumes input from its predecessor and sends output either to the event layer or to its respective successor stage.

In order to make InvaliDB suitable for these kinds of real-time queries without impairing overall scalability, the process of generating change notifications for more advanced queries is performed in additional **processing stages**[18]. The individual processing stages are separated according to the way their input data is partitioned, so that workload is always spread across as many nodes as possible. The first processing stage for any query is therefore the *filtering stage* as described in Section 3.3.1. It is the only processing stage to ingest after-images; all subsequent processing stages only receive change notifications from upstream matching nodes. The filtering stage only passes down written data items when they either satisfy a query's matching condition (match types `add` and `change`) or

[18] Similar to the *staged event-driven architecture (SEDA)* model [WCB01], InvaliDB employs loosely coupled processing stages that can be scaled independently.

when they just ceased matching (match type `remove`); all other input is filtered out. Thus, throughput is greatly reduced for subsequent stages in case of queries that require more complex processing, because no change notifications are generated for obviously irrelevant write operations (similar to existing approaches for efficient view maintenance, e.g. [BLT86] [BCL89] [LS93] [Elk90]).

The different stages are illustrated in Figure 3.7. Every query is registered in the filtering stage where the static filter predicates are evaluated. Since query results are partitioned in the *filtering stage*, it only directly serves queries that can be maintained without coordination between result partitions; this is only the case for unsorted filter queries over single collections. Change notifications for these queries are directly sent to the event layer. For more advanced queries, filtering stage output is instead passed on to the subsequent stages. In the *sorting stage*, matching nodes detect positional changes of individual items within the result (match type `changeIndex`). Likewise, added and removed items based on limit and offset clauses are identified here. Queries that link data between collections are also possible: A matching node in the *join stage* essentially combines the output of different real-time queries into a continuously maintained join query result. Finally, matching nodes in the *aggregation stage* are responsible for query components relating to aggregations and groupings. This includes evaluation of matching conditions over groupings (cf. `HAVING` clauses in SQL).

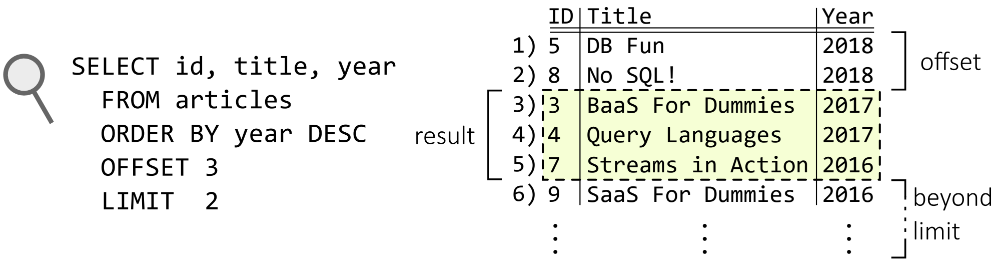
Like a physical **execution plan** in a relational database [HSH07, Sec. 4] which is determined by the executed query, the sequence of processing stages involved in result maintenance is also determined by the real-time query at hand. Depending on the query, the concrete processing stages and the order in which they are traversed can therefore diverge from the illustration. For example, maintaining a query result that is sorted by an attribute value may not involve the aggregation stage at all, whereas maintaining a result that is sorted by an aggregated value necessarily involves the aggregation stage before the sorting stage. In this thesis, we focus on queries with and without explicit ordering, aggregation queries, and simple join queries; we leave recursive queries, queries with subqueries, or other more complex constructs for future work (see Section 6.2.1).

In the following, we describe the individual advanced processing stages in more detail.

Sorted Queries With Limit & Offset

For *unsorted filter queries over single collections* (i.e. for queries without explicit ordering, limit, or offset), matching conditions are *static*: A given object is part of a given query result, if and only if the object satisfies all of the query's filter predicates. Thus, all information required for matching is encapsulated in the query and the written after-image. This does not only allow distributing workload by queries and writes at the same time, but also makes these queries inherently **self-maintainable** [QGMW96], i.e. their results can be kept up-to-date by simply applying incoming write operations.

This is not the case for explicitly *sorted filter queries*, because their change notifications also reflect result permutations (cf. `index` attribute on page 68); to capture these, a matching node in the sorting stage requires access to the full result. Moreover, even the matching status of an object can depend on its absolute **position** within the result or its relative position to other items: For a sorted query with a limit clause, adding a new item to the result can push the last item out and removing an item from the result can pull another item in. When a sorted query is specified with an offset clause, result membership further depends on the items in the offset, i.e. on items that are *not even part of the result*. To maintain a sorted real-time query in incremental fashion, having access to the full result may therefore not even suffice; for sorted queries with limit or offset clauses, a matching node requires auxiliary data.



```
SELECT id, title, year
  FROM articles
  ORDER BY year DESC
  OFFSET 3
  LIMIT  2
```

	ID	Title	Year
1)	5	DB Fun	2018
2)	8	No SQL!	2018
3)	3	BaaS For Dummies	2017
4)	4	Query Languages	2017
5)	7	Streams in Action	2016
6)	9	SaaS For Dummies	2016
	⋮	⋮	⋮

Figure 3.8: Knowledge of the items in the query's offset and beyond the specified limit is critical to enable incremental maintenance of a sorted query's result.

Figure 3.8 shows a sorted query with limit and offset clauses along with related data to illustrate the extent of the auxiliary data required for incremental result maintenance. When an article is removed from the offset by deletion or update (e.g. `'No SQL!'`), the first article in the result (`'BaaS For Dummies'`) will move into the offset, while the first article beyond limit (`'SaaS For Dummies'`) will move into the result. The other way around, when an article is added to the offset (either by insert or update), the last article in the offset will move into the result and the last article in the result will move beyond limit and thus out of the result. To be able to detect updates and deletes happening to items in the offset, the matching node responsible for a sorted query needs to be aware of all items in the query's offset. To further handle operations that remove an item from either the offset or the result, the node also needs to know at least one item beyond the specified limit; otherwise, a removed item cannot be replaced.

In order to make the query maintenance procedure more robust against these kinds of write operations, sorted real-time queries are registered with auxiliary data in InvaliDB. Similar to related work on top-k query maintenance [YYY$^+$03], the query used to retrieve the bootstrapping data (i.e. the initial result) is rewritten for this purpose: First, the offset clause is removed (i.e. `OFFSET = 0`), so that the initial result contains all elements in the query's offset. Maintaining all items in the offset is necessary, because otherwise the actual result cannot be maintained in the presence of certain update operations (see example above). Second, the limit clause is extended beyond the query's specified limit, so that the initial result contains all items in the offset, the actual result, and an additional number of items beyond limit; we refer to the number of items known beyond limit as **slack**. By definition, the slack changes dynamically, because items can enter and leave both result and offset at runtime. Therefore, the current slack also represents the number of subsequent removes that can be handled at a given time.

Whenever the slack reaches zero, removing an item from the result or offset will render the query unmaintainable, because the matching node cannot determine which effect the removal has on the query result. When such a **query maintenance error** occurs, the responsible matching node deactivates the query and sends out a change notification with an `error` attribute (cf. page 68). This particular error notification can also be seen as a **query renewal request**, because it triggers the process to retrieve a fresh result which is required for reactivating the query: On receiving a query renewal request, an application server reexecutes[19] the (rewritten) query against the database and submits the result to the InvaliDB cluster via subscription request. After receiving the up-to-date result, the responsible matching node in the sorting stage sends out incremental change notifications that reflect the evolution from the last valid to the current result representation. From there on, the query is self-maintainable again and therefore will produce change notifications until its cancellation or until the next query maintenance error.

To avoid wasting database resources during this recovery process, the matching node designates one (as opposed to all) of the subscribed application servers for query renewal. In more detail, it specifies one of the active subscriptions in the `id` attribute (cf. page 67) of the query renewal request; thus, the request is only processed by the application server responsible for the specified subscription and ignored by all others. If the request remains unanswered for a certain period of time, the node sends out cancellation notifications for all subscriptions of the unresponsive server[20] and repeats the renewal request, targeting a different application server. It repeats this process, until either the query is reactivated or until no subscribers remain (in which case the query is deactivated altogether).

The requirement of contacting the database for query renewal violates the intended isolation between real-time and OLTP workloads. For example, consider a sorted query with limit clause where items are removed from the result in rapid succession over a long period of time. Quickly after recovering from a query maintenance error (after exhausting the slack), the query will become unmaintainable again very quickly. As a result, change notifications will effectively only be sent on and immediately after recovery from maintenance errors. In order to make the query load inflicted upon the underlying database both predictable and configurable, InvaliDB controls the frequency of query renewals through a **poll frequency rate limit**. This rate limit can be specified as a globally fixed value, on the

[19] As a runtime optimization, the slack value can be adapted to the workload on reexecution, for example by using a higher slack value to increase robustness against deletes.

[20] InvaliDB maintains the mapping between servers and their respective subscriptions on the basis of the server ID which is transmitted on subscription as described in Section 3.2.4.

level of queries, or on the level of individual subscriptions[21]. To enforce a rate limit on a query, the responsible matching node delays sending a query renewal request whenever sending it immediately would result in a violation of the rate limit. For highly error-prone scenarios, the rate limit represents a trade-off between expended database resources on the one side and worst-case notification latency on the other: As long as maintenance errors are infrequent, though, the rate limit does not affect notification latency at all.

Joins

Intuitively, a real-time join can be achieved within InvaliDB through **incremental result maintenance**, similar to the query types discussed above: A node in the join stage maintains the different to-be-joined subquery results, combines them via the join conditions of the actual query, and updates the overall result whenever new data becomes available. On the upside, this approach enables real-time joins on top of databases that do not even support pull-based join queries (see Section 5.4.3 for an example). On the downside, though, it also prohibits database-internal query optimization: While a relational database avoids materializing auxiliary data where possible, evaluating the join query outside the database requires materialization of all subquery results. This is not feasible in the general case, but only when intermediate results are small.

Provided the underlying database supports pull-based joins, it can be more efficient to implement real-time joins through **result recomputation**. The basic idea is to exploit database-internal optimization mechanisms by querying the data exactly as requested by the client, specifically without rewriting the query for auxiliary data. Compared with external result maintenance of a join query, the performance penalty for query execution on subscription and query renewal is thus minimized. At the same time, though, query maintenance errors are likely to become more frequent due to the relative shortage of information. InvaliDB will generate change notifications without database interaction where possible; for example, updates to items within the result can trivially be applied when they do not alter attributes referenced in the matching or join conditions or the sorting key. However, the possibility of occasional or even frequent *query renewal* should be accounted for by choosing a reasonable poll frequency rate limit.

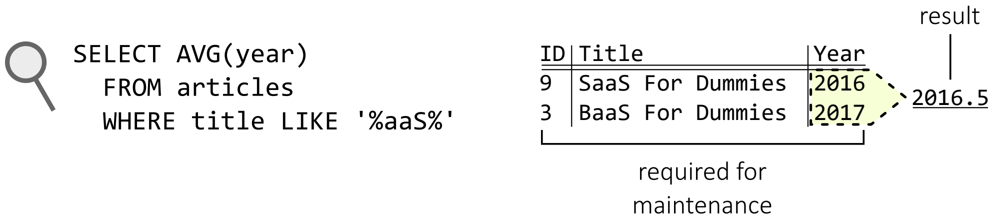
Through the deliberate fallback to pull-based query execution by repeated query renewal, this scheme resembles the *poll-and-diff* mechanism in Meteor (see Section 2.3.1). However, InvaliDB provides three significant *advantages*: First, it avoids unnecessary database queries by employing a dynamic query execution policy. Since the InvaliDB cluster monitors the entire write stream, it can maintain a query result *without any database interaction*, when only irrelevant writes are observed. Under poll-and-diff, in contrast, an application server only monitors part of the write stream and therefore relies on periodic query

[21] When different subscriptions for the same query prescribe conflicting rate limits, InvaliDB always enforces the one permitting the highest query frequency.

execution for change discovery: Even in the absence of write operations, a poll-and-diff application server must reevaluate all active real-time queries periodically to guarantee correctness. As a second advantage, InvaliDB enables better notification latency. Since InvaliDB observes all write operations as they happen, change notifications (or maintenance errors) can always be produced quickly; the only constraint is the deliberate delay introduced on error recovery through the *poll frequency rate limiting*. Latency under Meteor's poll-and-diff, on the other hand, depends on the polling interval, because some changes can only be detected through query reexecution[22]. As the third and arguably most important advantage, InvaliDB is able to support real-time *join* queries in the first place, whereas all of the real-time database concepts discussed in Chapter 2 (including Meteor's poll-and-diff) only account for real-time queries over single collections. For maintenance of real-time queries over single collections, in contrast, InvaliDB only resorts to pull-based query execution on errors which occur only infrequently.

Aggregations

Similar to join queries, aggregation queries can also be maintained either in recomputation-based or in incremental fashion. Since the recomputation-based method requests no additional information apart from the actual result, it is relatively cheap to compute for the database and also has a minimal network footprint. However, without any of the underlying data, invalidating writes are often hard to discern from write operations that have no impact on the result and can therefore be ignored.



```
SELECT AVG(year)
  FROM articles
  WHERE title LIKE '%aaS%'
```

ID	Title	Year
9	SaaS For Dummies	2016
3	BaaS For Dummies	2017

Figure 3.9: An aggregation query result can be impossible to maintain incrementally without also maintaining the base data from which the aggregate is derived.

For illustration, consider the query given in Figure 3.9 which produces the average publication year of a particular set of articles: a single numeric value (2016.5). If only this value is given to the maintenance routine (recomputation-based maintenance), any insert, update, or delete operation triggers an immediate maintenance error: On insert, the responsible matching node might be able to determine whether the written entity contributes to the average, but it cannot possibly tell by how much, since the overall number of contributing articles is unknown; for updates and deletes, the situation is even more involved,

[22] In Meteor, every real-time query is maintained by an individual application server. In a distributed Meteor setup, however, an application server is not notified of write operations performed by other application servers. Consequently, periodic query execution is necessary to find out whether and in what ways a query result has been modified through write operations received by other servers in the cluster.

because it is also unknown whether the written entity contributed to the aggregation before the write occurred. In order to implement a maintenance routine that updates the result in the presence of inserts, updates, and deletes, the result alone is therefore not sufficient: Beyond the actual `year` value, the unique identifier (`ID`) and all attributes required for determining the matching status (`title`) have to be known *for every article* that contributes to the current average value. With this information, the InvaliDB cluster can maintain the underlying set of articles (i.e. the ones with `'aaS'` in the title). In the example, only articles 9 and 3 contribute to the aggregation result, so that recomputation-based and incremental maintenance are both feasible. However, the amount of auxiliary data required for incremental maintenance (outside the database) can be prohibitive in the real world; as it is the case for joining queries, aggregation queries over large data sets tend to be infeasible to maintain in an incremental fashion.

Result approximation techniques as they are used in stream management systems (cf. Section 2.4.3) are generally not applicable to InvaliDB under the collection-based query semantics focused in this thesis, because they are dictated by the underlying pull-based database: If the maintained database query produces an exact result, InvaliDB must maintain the exact result as well. However, we see potential for optimization in the use of probabilistic or summary data structures [Cor17]. For example, an aggregation such as the one illustrated in Figure 3.9 can be maintained through the initial aggregation result (2016.5), a counter for the number of entities contributing to the aggregation result ($n = 2$), and a Bloom filter [Blo70] as a space-efficient representation of all contributing entities (all initially provided on subscription). For illustration, consider the incoming after-image of an article from 2018 which is found to comply with the query's `WHERE` clause. If the article was inserted or if the article was updated but its ID is not contained in the Bloom filter, it is certain[23] that the corresponding entity was not already contributing to the aggregation result. Thus, the counter can be incremented to $n = 3$ and the aggregation result can be updated to $2016.5 + \frac{2018-2016.5}{n} = 2016.5 + \frac{1.5}{3} = 2017$. Likewise, a delete operation can simply be ignored whenever the Bloom filter check for the deleted entity is negative. The query will only become unmaintainable, when a positive Bloom filter check indicates that the written entity might already be contributing to the aggregation result. Under certain workloads, representing the set of matching entities through a Bloom filter instead of an actual data collection could therefore significantly reduce the amount of data that needs to be transferred between the application server and the InvaliDB cluster.

As an important distinction to the sorting and the join stages, computation within the aggregation stage can be distributed across different matching nodes to a certain degree. To this end, aggregation stage input is partitioned according to the **grouping key** (if specified),

[23] A Bloom filter is a fixed-size representation of an arbitrarily large set of items that might produce false positives, but never false negatives. As an illustration of how a Bloom filter works, consider a list of name initials ("the Bloom filter") as the compressed representation of a set of names: If `'JD'` is not among the initials, `'John Doe'` cannot possibly be contained in the original set; if `'JD'` is on the list of initials, though, it is uncertain whether `'John Doe'` is in the set or whether the set contains another name with the same initials, e.g. `'James Dean'`.

so that an individual aggregation stage node for a given query only needs to maintain one partition of the result. While this does not solve the infeasibility problem for large data sets in principle, it does mitigate its severity for queries that are specified with a grouping key.

3.3.3 The Pluggable Query Engine

InvaliDB is a real-time database design that provides push-based real-time queries on top of an existing pull-based database. Since many of its components abstract from specificities such as query language or data format, InvaliDB can be understood as a generic framework where most components can be shared between implementations for different databases. In the following, we highlight the database-agnostic components of our approach and describe where database-specific development is required for an InvaliDB implementation.

Generic System Components. The event layer is database-agnostic as it handles entirely *opaque* data transmissions between end users, application servers, and the InvaliDB cluster. Likewise, the workload distribution scheme presented earlier in this chapter is not based on any concrete technology or database language; it only makes sure that all queries are matched against all incoming after-images. Finally, even the way that match types are derived[24] from the matching status abstracts from specificities of the underlying data store: Whenever an incoming after-image satisfies a query's predicates (`match`), it is either new to the result (`add`) or it was updated within the result (`change`), possibly changing its position (`changeIndex`); correspondingly, a non-matching after-image either corresponds to an item that is leaving the result (`remove`) or it does not bear any relevance to the result whatsoever.

Specialized System Components. There are only two aspects of real-time query maintenance that contain database-specific artifacts. First, an existing application server has to be adapted in such a way that fully specified and versioned after-images are produced for every write operation. As explained in Section 3.1.3, this is one of the underlying assumptions made in this thesis and required for InvaliDB to work. The exact measures to be taken here may vary depending on the employed database (see Section 4.3 for details on our implementation on top of MongoDB). The second customized component is the *pluggable query engine* which contains all logic related to (1) parsing queries according to one specific query language, (2) interpreting the incoming after-images according to the prevalent format and encoding, (3) computing the actual matching decision, and (4) sorting the result according to database semantics[25]. In addition, customized logic has to be applied on *query subscription* to extract query parameters from the subscription request and initialize possible auxiliary data structures required for matching. Since these steps are

[24] The procedure of deriving match types from matching decisions is illustrated in Figure 1.1 on page 3.

[25] While different databases usually sort values similarly, sorting order may differ in edge cases such as collections containing differently typed values or cases where different records have identical sorting keys.

executed across all of the different processing stages, the pluggable query engine defines interfaces on each one of them. By encapsulating the specialized parts of query matching behind different interfaces in a pluggable component, generic system components can be reused and support for new databases can be added with relative ease.

Collection-Based vs. Stream-Based Semantics. This thesis is focused on real-time queries with collection-based semantics: By default, a query result reflects the entire write stream history as it is bootstrapped with data from the underlying database. However, it is important to note that InvaliDB does not enforce collection-based semantics. On the contrary, the pluggable query engine explicitly enables arbitrary processing over the incoming write stream. When a query result is initialized as an empty collection, for example, it is implicitly scoped to the timeframe starting on query activation. In order to enable window semantics, the result can then be cleared periodically (tumbling window) or be maintained in such a way that old data items leave when new items arrive (sliding window). We do not go into further detail here and defer the challenge of providing stream-based semantics within InvaliDB to future lines of research (cf. Section 6.2.1).

3.4 Summary & Discussion

In Chapter 2, we found that modern real-time query mechanisms exhibit severe limitations with respect to scalability, query expressiveness, and compatibility to existing database systems. Our system design InvaliDB addresses all of these issues and thus provides a comprehensive solution to the challenge of providing push-based real-time queries over database collections. In the following, we highlight the most significant aspects by which InvaliDB separates itself from the current state of the art.

Scalability (cf. Challenge C_1). The arguably most critical of the above-mentioned problems is poor scalability, because it makes currently available real-time databases not only difficult, but practically infeasible to use: Some systems collapse when write workload exceeds single-machine capacity (e.g. Meteor with oplog tailing, RethinkDB, and Parse), while others fail in the face of many concurrent real-time queries (e.g. Meteor with poll-and-diff). Contrastingly, InvaliDB distributes computation and state for unsorted filter queries across a cluster of machines. Through its two-dimensional workload partitioning scheme, InvaliDB's overall matching performance is neither bounded by write throughput nor by the number of active real-time queries. Computation required for more complex queries (specifically sorting, joining, and aggregation) is performed in dedicated processing stages; while input for these stages is already greatly reduced by the filtering stage, it is also partitioned by query. By default, InvaliDB maintains queries in incremental fashion in order to minimize change notification latency. However, it also supports recomputation-based maintenance as a fallback for queries that are prohibitively expensive to maintain incrementally.

Query Expressiveness (cf. Challenge C_2). Real-time query languages often provide only limited or no support for sorted queries (e.g. RethinkDB, Firebase, and Parse), for content-based filtering (e.g. Firebase), or even for combining filter expressions (e.g. Firebase), while join queries are not supported by any current real-time database. InvaliDB, on the other hand, supports unsorted filter queries, sorted filter queries, join queries, and aggregations in push-based fashion. Even though collection-based query semantics are in the focal point of this thesis, InvaliDB can also support streaming query semantics. For stream-based semantics, the query engine forgoes retrieving the initial result from the database and restricts itself to the data arriving in the write stream to implement filters, joins, or aggregations over sliding, tumbling, or other variants of time windows. In this thesis, we do not provide a detailed discussion of stream-based semantics, because our focus lies on collection-based real-time queries.

Legacy Support (cf. Challenge C_3). Finally, we are not aware of even a single database-agnostic system designed to provide push-based real-time queries over database collections: Every real-time query mechanism we encountered is designed for and only applicable to a specific database system. InvaliDB, however, is designed with a *pluggable query engine* on top of an abstract data model. Through the generic design of the event layer and the different processing stages, data flow between application servers and the InvaliDB cluster as well as workload distribution for real-time query maintenance abstract from concrete database systems. At the same time, InvaliDB's capabilities are designed to match those of many popular database systems. In particular, InvaliDB provides the query expressiveness of typical aggregate-oriented NoSQL data stores through the filtering and sorting stage: These systems commonly support unsorted and sorted filter queries over single collections. Additional capabilities of relational databases, namely aggregation and join queries, are covered by dedicated processing stages.

In conclusion to this chapter, InvaliDB provides a scalable, expressive, and polyglot approach towards real-time queries over database collections. Next in Chapter 4, we will present our InvaliDB prototype and experimental evidence for its high scalability. In Chapter 5, we will then cover different industry applications for InvaliDB. Arguing for its practicality, we will describe how InvaliDB enables push-based real-time queries and consistent query caching on top of the purely pull-based database middleware Orestes and its commercial implementation Baqend.

4 InvaliDB Prototype: Implementation & Experimental Evaluation

> “However beautiful the strategy, you should occasionally look at the results.”
>
> —Winston Churchill

In Chapter 1 and Chapter 2, we argued for the usefulness of real-time databases, surveyed the different representatives available today, and conducted a qualitative comparison of these representatives with respect to different functional and non-functional requirements. In Chapter 3, we then presented InvaliDB, a system design to provide expressive real-time queries over database collections in a scalable and database-agnostic manner. In this chapter, we demonstrate that our system design is feasible to implement and highly scalable with regards to read and write workloads. To this end, we first describe our distributed InvaliDB prototype that has been running in production at Baqend since July 2017. Through an experimental evaluation, we then confirm our claims of scalability and low latency. More specifically, we provide evidence that sustainable query matching throughput scales linearly with the number of servers employed for query matching, while latency remains consistently low across different InvaliDB cluster configurations.

To start the chapter in Section 4.1, we lay out our reasons for choosing Storm as the underlying processing engine and discuss possible alternatives. We also describe how our workload partitioning scheme maps to a Storm topology. In Section 4.2, we then address the event layer that decouples the InvaliDB cluster from the pull-based system on which it is deployed. In more detail, we describe two different Redis-based event layer implementations which mainly differ in the degree to which they can be scaled horizontally and the operational complexity involved in their respective deployments. In Section 4.3, we then cover the pluggable query engine and two MongoDB-compatible implementations: our very first prototype which uses a third-party JavaScript library for query predicate evaluation and a more sophisticated implementation which is completely written in Java. Having thus overviewed our InvaliDB prototype implementation, we present experimental results in Section 4.4 to quantify sustainable matching throughput, latency characteristics, and scalability: Our prototype exhibits latency consistently below 50 ms in the 99th percentile with peak latency rarely exceeding 100 ms, while overall system throughput scales linearly with the number of employed matching nodes. In the final Section 4.5, we sum up and discuss our results.

4.1 An InvaliDB Implementation Based on Storm

By design, an InvaliDB implementation inherits some of its characteristics from the underlying stream processor which is required to implement the workload distribution scheme described in Section 3.3. For implementation, we therefore only considered horizontally scalable and fault-tolerant stream processors providing at-least-once or exactly-once delivery guarantees and low end-to-end latency: We thus immediately dismissed systems that are prone to data loss (e.g. S4 [NRNK10]), had already been abandoned (e.g. Muppet [LLP$^+$12] or Naiad [MMI$^+$13]), or were not publicly available at the time (e.g. Heron [KBF$^+$15], Apex [Apa16c], Kafka Streams [Kre16], Wallaroo [Mum17]). We also did not consider systems that cannot be deployed on-premise (e.g. Google's MillWheel [ABB$^+$13] and Photon [ABD$^+$13] or the Dataflow cloud service [ABC$^+$15], and Facebook's Puma and Stylus [CWI$^+$16]). Given the requirement for low latency, the viable choices[1] for the underlying stream processor were thus narrowed down to either Storm [TTS$^+$14] or Flink [ABE$^+$14]; by concept, Samza [Ram15] and Spark Streaming [ZDL$^+$13] cannot provide competitive latency. Even though Flink provides higher-level abstractions in comparison with Storm and therefore might have facilitated more efficient development, we finally chose Storm for the benefit of better latency [CDE$^+$15]. Like Storm, InvaliDB is written in Java to facilitate high performance: Since our InvaliDB prototype is running as a Storm topology within the same Java Virtual Machine (JVM) as the Storm supervisors, it avoids expensive data serialization and deserialization that would have been required, if we had chosen any other language.

4.1.1 Workload Distribution

In contrast to some other systems such as Samza, Storm facilitates complex data flow graphs. Thus, Storm topologies are a natural fit for InvaliDB's internal design which performs query maintenance in multiple processing stages (cf. Section 3.3). At the same time, Storm only distinguishes between two different kinds of nodes within each topology: InvaliDB's query ingestion and write ingestion nodes are implemented as Storm *spouts*, whereas all processing stages are implemented using Storm *bolts*. The ingestion nodes (spouts) receive data from the event layer and dispatch it to the associated matching nodes (bolts) in the filtering stage, targeting all matching nodes in a write partition (write ingestion) or query partition (query ingestion) at a time. To enable this kind of multicast message flow within the topology, we implemented a custom Storm grouping (cf. [TTS$^+$14, Sec. 2]). Following InvaliDB's workload distribution scheme, the processing bolts in the filtering stage are the only ones to receive input from spouts; all subsequent processing stages are fed from upstream bolts via intra-topology messaging, i.e. they receive change notifications generated in the preceding processing stages. The processing bolts

[1] We provide a more detailed comparison of these systems in Section 2.5.

in the final processing stage of any real-time query type send the finalized change notifications to the event layer. Since performance and scalability of InvaliDB's recomputation-based real-time queries (cf. Section 3.3.2) are limited by the underlying database rather than InvaliDB itself, we only implemented incremental real-time queries in our prototype.

4.1.2 Elasticity, State Management & Fault Tolerance

Our InvaliDB implementation receives its scalability and **elasticity** properties from Storm: More machines can be added to an InvaliDB cluster at runtime in order to increase sustainable matching throughput on-the-fly. For our InvaliDB prototype, we did not make use of Storm's built-in **state management**, since it is prohibitive for low latency (cf. page 45). Consequently, all state of a running spout or bolt will be lost, when a node fails or when workload is transferred from one query matching process to another (i.e. on elastic scale-out or scale-in at runtime). Given an event layer implementation that is able to replay data, the lack of state management is not an issue for spouts (ingestion nodes), because they are completely stateless apart from metadata collected during bootstrapping: The combination of at-least-once processing in Storm and data replay in the event layer guarantees correctness in the presence of spout failures. If the event layer is *not* able to replay data (as is the case in our implementation, see Section 4.2), there are different scenarios to consider.

Regardless of the event layer's replay capability, a **query spout failure** can always be tolerated, because it can only trigger three different kinds of data loss, all of which result in non-critical error conditions. First, a *subscription request* might be dropped, before it is processed by the InvaliDB cluster. In this case, the subscribing client will receive a timeout[2] and needs to repeat the subscription attempt. Second, a *cancellation request* can be missed, so that InvaliDB might keep on maintaining a real-time query that has no subscribed clients. Since every real-time query is registered with a time to live (TTL) that is not extended without live subscribers, though, the query will be removed from InvaliDB eventually and bound resources will thus be freed. As a third error condition, losing TTL extension requests can lead to premature query deactivation: But without renewing a query's TTL, it will eventually be canceled and error messages will be sent out to all listening clients (timeout). Like other runtime errors such as connection loss or query maintenance errors (cf. page 80), a premature deactivation has to be addressed by the client, e.g. through resubscribing the failed query. For our prototype, premature deactivation does not occur, unless several TTL extension requests for the same query are dropped in direct succession; our implementation uses 120 seconds as the default TTL and sends extension requests every 30 seconds.

[2] Even if no initial result is requested (see subscription parameters in Section 3.2.4), our InvaliDB prototype emits an initial message to inform the subscribers of the successful query activation.

While query-related data loss only affects those query subscriptions for which data has been dropped, losing even a single write operation may corrupt any currently maintained query result: If InvaliDB does not match all incoming writes against all active real-time queries, it cannot guarantee convergence of real-time query results towards the underlying database's state as a liveness guarantee (cf. Section 3.1.2). Since any write operation might thus potentially affect any query result, all currently maintained real-time queries have to be canceled on **write spout failure**, unless missed write operations can be replayed by the event layer. As every subscriber receives an error notification, canceled queries can be resubscribed[3].

Since our implementation relies on a Storm cluster without state management and since every bolt (processing node) has critical state such as the maintained query result for every active subscription, **bolt failure** cannot be tolerated regardless of the event layer implementation: Whenever a subscription's state is lost because of workload reassignment or server outage, it is therefore irredeemably broken and has to be canceled. The **bolt startup procedure** guarantees that such irrecoverable state loss cannot occur unnoticed: A starting bolt always retrieves all currently active queries in its query partition from the event layer[4]. All subscriptions found to be active are then canceled through error messages, because the state required for their maintenance is not available in the starting bolt; in the absence of error conditions, however, a starting bolt will not find any active queries and will therefore not send out any error message. While this startup procedure guarantees correct behavior in the presence of bolt failure and workload migration, it also exposes runtime errors to end users on these events. For self-maintaining queries that deliver complete query results, these errors can be handled transparently by implicit query resubscription (cf. `reconnect` option on page 129). For event stream queries that deliver incremental changes, on the other hand, explicit error handling within the client application is required, because a simple resubscription may have undesired side effects: Depending on whether or not the initial result is requested on resubscription, events might be reflected more than once in the event stream (resubscription with initial result) or events might be missed when they occurred in the short time between the runtime error and resubscription (resubscription without initial result).

Summing up, our InvaliDB prototype preserves **correctness** under both unexpected worker failure and workload migration by proactively canceling real-time query subscriptions whenever eventual correctness of the maintained results cannot be guaranteed. Handling these situations in a transparent fashion (i.e. without the client noticing) would have required an efficient, scalable, and reliable state management. We could have implemented this on top of Storm, but opted against this option, because it would have consumed con-

[3] As stated in Section 3.1.3, our work assumes that application servers implement a rate limiting for pull-based queries. Therefore, resubscription of many or even all active real-time queries does not overload application servers nor the underlying database system.

[4] In order to make this bolt startup procedure feasible and efficient, our event layer implementation (see Section 4.2) needs to maintain all active real-time queries in every partition of the notification pipeline.

siderable resources during development. As another option, we could have used a stream processing framework with viable state management (e.g. Flink [ABE+14]) in combination with a durable event layer implementation capable of data replay (e.g. built on Kafka [KNR11]). However, we chose Storm and Redis for the benefit of reduced latency during normal operation over their more reliable peers that would have enabled more convenient error handling.

4.2 A Redis-Backed Event Layer Prototype

The implementation of InvaliDB's event layer was guided by several **requirements**. First, InvaliDB's event layer consists of three different communication pipelines for write propagation, transmission of change notifications, and query subscription management (cf. Section 3.2). For each of these communication pipelines, the event layer needs to act as a *message queue*, with slight variations regarding the way that data is provided and delivered as well as the ordering guarantees that are required. As a second requirement, the event layer has to support the bolt startup procedure detailed on page 90 which guarantees correctness despite the lack of state management in our stream processing engine: To enable efficient access to all currently active real-time queries per partition at any given time, our event layer implementation therefore must enable pull-based storage and retrieval of data, like a *key-value store*. As a final and non-functional requirement, *low latency* is necessary to enable interactive real-time applications.

We considered various **messaging middleware systems** (cf. Section 2.4.5) as the basic building block for our event layer implementation, but excluded software libraries without built-in standalone server (e.g. ZeroMQ [Hin13] [iMa18]) and systems that were not publicly available when we started development in 2014 (e.g. Moquette [Sel18]). To keep development and operational overhead low, we further decided against a polyglot implementation and therefore also discarded systems that did not meet all of the aforementioned requirements (e.g. Kafka[5] [KNR11]). The remaining systems can be separated into two classes: *durable* message queues with exactly-once semantics (e.g RabbitMQ [Piv18], ActiveMQ [Apa18a], Qpid [Apa18h]) and *transient* systems with a focus on low latency (i.e. Redis [San18b], NATS [Clo18b]). Even though the ability for data retention and replay would have facilitated a more robust event layer implementation, we decided to use a system that provided minimal response times [Tre16]. We eventually chose Redis over alternatives like NATS for our event layer implementation, because it appeared easier to operate[6].

[5] We started developing our InvaliDB prototype in 2014, but queryable state in Kafka did not become available before the second half of 2016 [The16]. Therefore, Kafka did not lend itself to maintaining the currently active real-time queries as required for the bolt startup procedure described on page 90.

[6] First, we already had experience with Redis from running it in production and, second, there were numerous managed Redis offerings available to simplify deployment and maintenance (e.g. Azure Redis Cache [Pan14] and Amazon ElastiCache [Bar13]); to the best of our knowledge, there were no managed service offerings for NATS.

Redis (Remote Dictionary Server) is an in-memory key-value store that holds data in specialized data structures [DST15, Ch. 1 and Ch. 2], specifically strings, lists, sets, hashes, and sorted sets. Streams as native data structures were introduced as an experimental feature into Redis in 2017 [San17], but are still in development as of March 2018 [San18f]. Therefore, our implementation does not use them and instead relies on the originally available data structures alone. Similar to stored procedures in relational databases, Redis supports server-side atomic execution of Lua scripts [DST15, Ch. 4]. Redis is single-threaded and therefore unable to exploit multi-threaded processors[7]. By avoiding parallel processing, on the other hand, Redis is able to provide optimistic batch transactions without locking, latching, and other processing overhead that is inherent to concurrent database designs [HAMS08] [SMA+07]. In consequence, Redis is extremely efficient and able to achieve single-node throughput in the order of hundreds of thousands of read and write operations per second [San18a].

Redis facilitates high availability through *Redis Sentinel* [San18e], a management component that orchestrates different Redis instances in a master-slave replication scheme without data partitioning. To achieve horizontal scalability, **Redis Cluster** [San18c] divides the data into several partitions, each of which can be configured for master-slave replication as well. While Redis Cluster provides the obvious benefits of higher availability and horizontal scalability in comparison with a single-node Redis deployment, it is also more complex to operate. More importantly for this work, Redis Cluster did not provide a scalable publish-subscribe mechanism at the time of writing[8] [GCS+15] [GC15] which also made it significantly more challenging to develop an event layer implementation on top of it.

4.2.1 Implementation & Correctness

To explore the different trade-offs of the single-node Redis and the distributed Redis Cluster in practice, we created two different implementations of the event layer.

The **centralized implementation** relies on a single Redis instance that manages all communication between application servers and the InvaliDB cluster. The *write pipeline* is implemented as a list where after-images are inserted on the one side (`RPUSH` operation) and taken out on the other (`LPOP` operation): Every write operation is thus extracted and processed by exactly one write spout. Since write operations are propagated to all matching nodes in the corresponding write partitions, this scheme effectively implements a multi-

[7] For completeness, we want to point out that Redis provides the option to execute specific commands concurrently to the main thread of execution (but without concurrency control) to avoid prolonged blocking, e.g. `FLUSHDB` [San16].

[8] The Redis Cluster specification allows any client to subscribe to any publish-subscribe channel, even if the client is not connected to the Redis Cluster node responsible for the subscribed channel. To make sure that a subscribed client never misses a published message regardless, Redis Cluster implements a very inefficient strategy: When a producer publishes a message, the receiving Redis Cluster node broadcasts the message to all other Redis Cluster nodes, so that they can deliver it to any subscribed client. While this scheme is convenient to use, it also causes sustainable throughput to deteriorate (not increase) with growing cluster size, because the number of sent messages grows with the number of Redis Cluster nodes.

cast distribution. The *query pipeline* for delivering subscription, cancellation, and TTL extension requests works by the same principle for message delivery as the write pipeline, but involves additional processing for maintaining the set of active queries in Redis (cf. bolt startup procedure on page 90). In more detail, the active queries are represented as a map (i.e. Redis hash) that contains the number of active subscriptions for every real-time query; a query (i.e. hash key) is added whenever no counter is found on subscription and removed whenever the corresponding counter reaches zero on cancellation. To make sure that no query is activated in the InvaliDB cluster without being added to the set of active queries, the query pipeline further uses Redis' optimistic batch transactions to update the query map atomically with every query subscription. The *notification pipeline* is implemented using a Redis publish-subscribe channel [DST15, Ch. 4] per active real-time query: Both initial results and change notifications are published by the responsible InvaliDB matching nodes and then broadcasted to the subscribed application servers by the Redis publish-subscribe mechanism. Spreading the workload over more than three Redis instances (i.e. one per pipeline) is not trivially possible with the centralized implementation, because it would require application-level sharding (i.e. sharding on the level of producers and consumers) not only across different message queues, but across different Redis instances.

To remove this scalability limitation, we also created a **distributed implementation** of the event layer on top of Redis Cluster; we only provide a brief overview here and refer to [Suc17] for details. To facilitate even load distribution, the individual communication pipelines are hash-partitioned by ID (write pipeline) or query (query and notification pipelines) across data partitions. Thus, the distributed *query and write pipelines* are essentially sharded implementations of their centralized counterparts, i.e. each of them employs a single message queue per partition to implement multi-cast message propagation as described above. In contrast, we had to design the distributed *notification pipeline* from scratch, because the publish-subscribe mechanism built into Redis Cluster turned out to be infeasible at scale as described above. In more detail, we implemented our own broadcasting mechanism through Lua scripting: Whenever an InvaliDB matching node sends an initial result or change notification to Redis Cluster for publication, the published message is atomically duplicated and inserted into every message queue that corresponds to a subscription for the real-time query. In contrast to the write and query pipelines, the notification pipeline thus employs one message queue per real-time query *subscription* instead of only one message queue per partition. Unlike the native publish-subscribe mechanism, clients of our implementation only receive a keyspace notification [San18d] for every published message and then have to retrieve the published message itself in pull-based fashion. Thus, our approach incurs an additional round-trip between the application server and a Redis Cluster node for every published message; however, seeing that both these components are colocated, the imposed latency overhead is in the order of single-digit milliseconds only.

Neither Redis nor Redis Cluster provide mechanisms for acknowledgment or data replay, so that messages may be lost in the presence of connectivity issues. In order to guarantee **correctness** in spite of this drawback, all possibly affected queries are therefore invalidated as soon as an InvaliDB component or application server gets disconnected. In more detail, a connectivity issue between the event layer and an InvaliDB data ingestion or matching node is treated like a failure of the component itself (see Section 4.1.2). To avoid losing change notifications, an application server therefore cancels all its real-time query subscriptions and notifies the subscribed end user devices accordingly, after losing its connection to the *change notification pipeline*. As explained earlier, losing a write operation has the potential to compromise any maintained real-time query result on the same collection. One approach for handling disconnects between an application server and the *write pipeline* is therefore to have the application server cancel all possibly affected active queries as soon as a write operation might have gotten lost; our event layer implementations introduce dedicated components to serve this purpose (cf. state controller in [Suc17]). Since the InvaliDB cluster ignores already-processed after-images (cf. Section 3.1.3), however, an arguably more practical approach for scenarios with moderate write throughput is to have the application server buffer all write operations for the maximum propagation delay[9] and have it replay all possibly lost writes after reconnect. Since query subscription, cancellation, and TTL extension requests can be lost without compromising correctness (cf. query spout failure on page 89), disconnects between the application server and the *query pipeline* do not have to be addressed at all: In the worst case, query subscriptions will simply abort or expire.

Since InvaliDB's event layer only handles transient data that is discarded on reconnect in our Redis-based implementations, replication is disabled for both prototypes.

4.2.2 Horizontal Scalability & Deployment Considerations

Figure 4.1 presents experimental evidence for the **horizontal scalability** of our distributed event layer implementation. The experiments were executed on Redis 3.2.9 instances, running on Docker 1.10.2 with 2 vCPUs per Redis instance (AMD FX-8350 at 4.0 GHz, 1 thread per core). All measurement points are averaged over five runs. In our experiments, we did not measure absolute event layer performance, because even the centralized implementation was easily able to saturate the network link of our benchmarking client. Therefore, we throttled the Docker containers hosting the Redis instances in these experiments to 2 % CPU time in order to quantify the relative effect of adding nodes to the Redis Cluster. The line plot shows that a throttled single-node event layer deployment was able to sustain about 8 800 writes per second, while doubling the number of nodes increased sustainable write throughput by 75 % to 80 % each time. Even though the absolute numbers differ for change notification throughput, the relative performance increase

[9] As stated in Section 3.1.3, our system model assumes a finite upper bound for message propagation delays.

is comparable: Given a single subscribed application server, roughly 3 700 change notifications per second were feasible for the throttled single-node deployment, while doubling the number of nodes also increased sustainable throughput by at least 75 %. As was to be expected, adding more subscribed application servers reduced sustainable change notification throughput slightly, but did not affect linear scalability. We do not provide measurements of sustainable query subscription throughput, because it was limited by sustainable querying throughput of the underlying database. We further do not provide latency measurements here, because event layer latency is subsumed by the end-to-end latencies measured in the experiments presented in Section 4.4 and Section 5.5.1. For details on the experimental setup of the event layer evaluation and for additional results, we refer to [Suc17, Ch. 6].

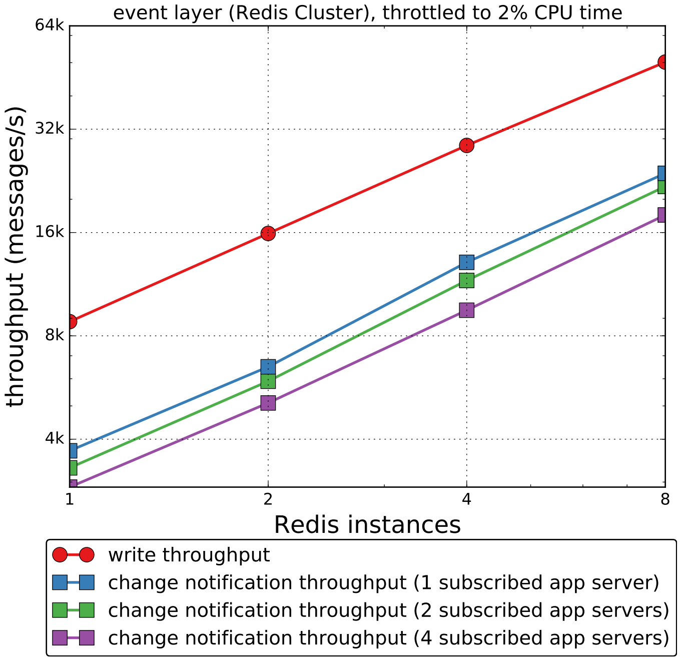


Figure 4.1: Sustainable write and change notification throughput scale linearly with the number of nodes in the distributed event layer implementation based on Redis Cluster. Note that (1) the experiments were conducted on throttled hardware (2 % CPU time), because unthrottled deployments could not be saturated, and that (2) both axes are on a logarithmic scale. (Data taken from [Suc17].)

In summary of the above, our distributed event layer implementation is unlikely to become a bottleneck in the overall system for two reasons. First, throughput achieved by an unrestricted **production deployment** can be assumed to be significantly higher than the throughput achieved during the experiments described above, since our experiments were executed on hardware throttled to only 2 % CPU time: Without this artificial bottleneck, we were not able to saturate even the single-node event layer deployment. In other words, our experiments are designed to quantify *scalability rather than absolute performance* and should not be taken as hints on the upper performance limit of the tested event layer implementation. Second, our results confirm that efficiency remains high in deployments with many nodes. While sustainable change notification throughput decreases with an increasing number of subscribed application servers, it is important to note that no application server subscribes more than once to any individual real-time query: Since end user subscriptions are coalesced (cf. Section 3.2.4), the maximum number of subscribers for a single real-time query to be served by the event layer is thus equal to the number of application servers. Since any application server can potentially serve thousands of end users (cf. performance evaluation in Section 5.5.1), the relative efficiency reduction for multiple subscribers therefore only has very limited impact on scalability of end user subscriptions.

While the distributed event layer implementation is more scalable and might thus be superior in deployments with high-performance networks, it does not necessarily provide an actual benefit over the centralized implementation in practice. Incidentally, we have not been able to identify a scenario where the performance of our centralized implementation becomes a limitation: Neither the workloads executed during the experiments described in this thesis nor any workload we have seen in production at **Baqend** [Baq18a] so far have put the single-node event layer implementation under significant pressure. Given the benefit of operational simplicity, the production deployment at Baqend therefore uses the centralized event layer implementation at the time of writing.

4.3 A MongoDB-Compatible Real-Time Query Engine

Since InvaliDB is database-agnostic, we could have built our prototype on top of any database system conforming to the requirements laid out in Section 3.1.3. However, we decided to build upon **MongoDB**, because this choice warranted a better comparison between our prototype and the current state of the art: As discussed in Chapter 2, many of today's real-time database systems are also built on MongoDB (e.g. Meteor, Parse) or display a strong similarity regarding data model and query expressiveness (e.g. RethinkDB, Firestore, Rapid.io). MongoDB is an aggregate-oriented document store [SF12] that natively supports horizontal scalability through data sharding. Conveniently, MongoDB provides a native change data capture mechanism that makes retrieving after-images for the write pipeline trivial: Our prototype uses the `findAndModify` [Mon18b] operation for inserts and updates[10], because it directly returns the after-images which are then simply forwarded to the InvaliDB cluster. Since InvaliDB requires versioning which was not supported by MongoDB at the time, every record carries a version number which the application server initializes on insert and which it increments with every subsequent write.

4.3.1 Prototype Iterations & Query Expressiveness

The pluggability of our real-time query engine allowed us to separate development of the database-specific matching component from development of InvaliDB's generic compo-

[10] The after-image of a deleted entity is `null` and therefore does not have to be retrieved from the database.

nents such as result state management and the interaction with the event layer. Although its source code is openly available [Mon18f], we did not use MongoDB directly for query matching because of two reasons. First, the C++ MongoDB code for matching queries against after-images would have been difficult to isolate and integrate into our Java-based Storm topology. Second and more importantly, though, including MongoDB in our InvaliDB prototype would have required us to adopt an AGPL licensing [Mon18g]; since we intended to use InvaliDB as part of a closed-source commercial offering at Baqend (cf. Chapter 5), this was not an option. To avoid these issues, we opted for a custom-built query engine running within the InvaliDB cluster's Java runtime (i.e. within the Storm topology). We started with a minimalistic real-time query engine to create a working proof-of-concept implementation quickly and to concentrate on implementing the generic InvaliDB components first. After the generic components had been completed, we reengineered the real-time query engine to improve performance and query expressiveness.

The first iteration was a **JavaScript-based query engine** that used the third-party JavaScript library sift.js [Con18] for evaluating MongoDB queries. In principle, we matched every incoming after-image against any active real-time query by invoking a function within the sift.js library. For this purpose, we used the Nashorn JavaScript engine [Ora18a] built into Java 8. The great advantage of this approach was its simplicity, since it allowed us to create a running version of the InvaliDB cluster without implementing the actual matching logic ourselves. However, the real-time query engine based on sift.js had three critical flaws. First, it was slow, because the context switch between the Java and the JavaScript runtime environments took longer than the actual matching procedure. Even when micro-batching after-images, the cost of merely invoking the JavaScript matching routine dominated processing latency. Second, advanced queries such as full-text search and geo queries were not supported. As the third and arguably most severe problem, sift.js only approximated MongoDB's query semantics and hence led to incorrect query results (e.g. for nested queries with dot notation [YCP17]).

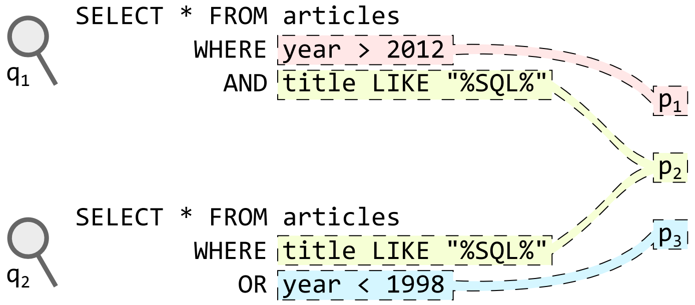
In order to eliminate query matching as a performance bottleneck, to enhance expressiveness, and to satisfy our requirement of eventual correctness (cf. Section 3.1.2), we created a **Java-based query engine** from scratch in the second iteration. By removing the JavaScript runtime from the critical processing path and thus the necessity for micro-batching, average processing latencies were reduced by one order of magnitude from double-digit to single-digit milliseconds and peak matching throughput even increased by two to three orders of magnitude, depending on the complexity of the queries used for comparison[11]. Further, InvaliDB's Java-based real-time query engine closely follows the syntax and semantics of MongoDB's query language for sorted filter queries over single collections. It supports query operators for content-based filtering through regular expressions (`$regex`), comparisons (e.g. `$eq`, `$ne`, `$gt`, `$gte`), logical combination of filter expres-

[11] We do not provide a detailed performance comparison between both implementations, because the JavaScript-based real-time query engine has been abandoned and has therefore become irrelevant.

sions (e.g. `$and`, `$or`, `$not`), evaluating matching conditions over array values (e.g. `$in`, `$elemMatch`, `$all`, `$size`), full-text search (`$text`), geo queries (e.g. `$geoIntersects`, `$geoWithin`, `$nearSphere`), and various others (e.g. `$exists`, `$mod`). *Full-text search* [Sch18] and *geo queries* [Pat18] have been particularly complex to emulate, since they are domain-specific and have sophisticated semantics. For example, they can produce results that are ordered *implicitly* by metadata such as a similarity score (full-text search) or the distance between a matching entity and a reference point (e.g. `$nearSphere` geo queries). As of writing, our real-time query engine does not support aggregations or joins[12], but extending it to support these kinds of queries is planned for future work (cf. Section 6.2.1). In Section 5.4, we show how both real-time aggregations and real-time joins can be achieved through client-side application code.

4.3.2 Multi-Query Optimization & Computational Complexity

Internally, every real-time query is represented as a logical composition of filter predicates. By decomposing all queries into their individual predicates, our Java-based real-time query engine identifies common subexpressions and achieves **predicate coalescence** across queries as illustrated in Figure 4.2: Even though both queries in the example have two predicates each, only three distinct predicates have to be evaluated overall, since predicate p_2 is shared between query q_1 and query q_2.



```
SELECT * FROM articles
        WHERE year > 2012
          AND title LIKE "%SQL%"
```

```
SELECT * FROM articles
        WHERE title LIKE "%SQL%"
           OR year < 1998
```

Figure 4.2: Query $q_1 = p_1 \wedge p_2$ and query $q_2 = p_2 \vee p_3$ have predicate p_2 in common.

For computing change notifications, the Java-based real-time query engine matches every incoming after-image against the query predicates[13] of the currently active real-time queries. Given a write throughput m and active real-time queries with overall n query predicates, the **computational complexity** of this naïve approach is thus $O(m \cdot n)$, i.e. the computational requirements for query matching grow linearly with both write throughput and the number of predicates. By using indices for predicate evaluation, overall computational complexity could be reduced to $O(m \cdot \log\, n)$ for certain predicates such as comparisons, equality checks, or anchored regex expressions (cf. [Mon18d]). Required matching

[12] MongoDB does not support joins except left outer equi-joins as part of its aggregation pipeline [Mor15].

[13] The query engine skips predicate evaluation where possible and thus does not necessarily evaluate all predicates for every after-image. For example, the query engine will not evaluate predicates in a disjunction that is already known to be true and it will not evaluate predicates in a conjunction that is already known to be false.

resources would thus still grow linearly with write throughput, but only logarithmically with the number of predicates. Due to time constraints during the development of our prototype, however, we did not implement indexing within our real-time query component and therefore leave this optimization to future work (cf. Section 6.2.2).

While overall matching node performance is typically dominated by query matching, it can also be influenced by other factors. For example, **overhead** for serializing and deserializing after-images can also bind substantial computing resources and even becomes prohibitive under write-heavy workloads unless write stream partitioning is applied (cf. evaluation in Section 4.4.3). As another example, queries with low selectivity are more expensive to maintain than queries with high selectivity, because they produce change notifications more frequently and thus incur a higher overhead for serialization and messaging.

4.4 Performance Evaluation

In this section, we quantify the throughput and latency characteristics as well as the scalability of our InvaliDB implementation. To this end, we present measurements of sustainable matching throughput and change notification latency for differently sized InvaliDB clusters. Through our experimental evaluation, we confirm that InvaliDB exhibits the following characteristics:

- **linear read scalability**: In Section 4.4.2, we show that the number of concurrently maintainable real-time queries is proportional to the number of employed query partitions.
- **linear write scalability**: In Section 4.4.3, we demonstrate that sustainable write throughput of an InvaliDB cluster increases linearly with the number of employed write partitions.
- **multi-tenancy**: In Section 4.4.4, we demonstrate that InvaliDB is well-suitable to serve multi-tenant deployments. To this end, we compare different workloads with the same number of matching operations per second (matches/s) as the number of queries times the number of writes per second. Thus, we show that InvaliDB achieves comparable performance in scenarios with many cheap tenants (workload spread across many database collections) as well as scenarios with few heavy hitters (all queries and writes on a small number of collections).

4.4.1 Experimental Setup

All experiments were executed in a private cloud environment (OpenStack 2013.2 Havana, Docker 17.09.0-ce, Ubuntu 14.04). The underlying hardware comprised five identical machines with six-core CPUs (Intel Xeon E5-2620 v2 at 2.1 GHz, 64 GB RAM), connected over a 1 Gbit/s LAN. Our experimental setup consisted of the following components:

- 1 InvaliDB client (2 vCPUs) for inserting records and measuring latency
- 1 Redis 3.0 server (1 vCPU) for communication between InvaliDB client and cluster
- 1 Storm 1.1.0 cluster (1 vCPU per node) running InvaliDB:
  - 1 nimbus for cluster management
  - 5 spouts for query ingestion (1 spout) and write ingestion (4 spouts)
  - $n \in \{1, 2, 4, 8, 16\}$ bolts for query matching (varied with experiments)

It should be noted that 4 write spouts were only required to saturate the most powerful InvaliDB cluster; the less powerful InvaliDB clusters with few matching bolts displayed virtually identical performance, irrespective of the number of employed write spouts. However, we deliberately chose the same number of spouts for all experiments, so that workload partitioning was the only difference between different configurations.

When assigning more than 4 matching nodes to a physical server, we observed inconsistent performance under high load, because matching nodes were contending for CPU time with the underlying virtualization stack. However, we were unable to deploy large InvaliDB clusters with less matching nodes per physical server with our limited resources[14]. Therefore, we throttled all InvaliDB matching nodes to 80 % of single-core CPU time, so that all of them received approximately equal computing resources regardless.

		query partitions				
		1	2	4	8	16
write partitions	1	1	2	4	8	16
	2	2	4	8	16	32
	4	4	8	16	32	64
	8	8	16	32	64	128
	16	16	32	64	128	256

Table 4.1: The workload partitioning determines the number of matching nodes in the InvaliDB cluster. With the given hardware, we could only evaluate configurations with no more than 16 matching nodes (green background).

To investigate InvaliDB's scalability, we doubled the number of query or write partitions between certain runs and compared achieved throughput and notification latency. Table 4.1 shows the number of matching nodes required for this **partitioning** scheme with up to 16 query and write partitions. With the hardware available to us, we were only able to deploy InvaliDB clusters with up to 16 matching nodes (green background). In this section, we describe the experiments where either query or write partition count was scaled from 1 to 16 (first row and first column, respectively). We present results of the experiments conducted with the remaining feasible configurations in Appendix A (read scalability), Appendix B (write scalability), and Appendix C (latency histograms).

[14] For the experiments described in Section 5.5.1, we needed to deploy an application server in addition to the components described here.

Workload

For every InvaliDB cluster configuration, we performed a series of experiments, each of which consisted of two phases: In the **preparation phase**, any still-active queries from earlier experiments were removed and queries for the upcoming one were activated. In the subsequent 1-minute **measurement phase**, the client machine performed a steady number of insert operations per second against the event layer (Redis server). The client also measured change notification latency as the time from *before* inserting an item until *after* receiving the corresponding notification; we thus measured end-to-end latency values that subsumed both processing times in the InvaliDB cluster and message propagation delays through the event layer.

```
SELECT * FROM test WHERE sequence >= i AND sequence < j
```

Listing 4.1: The queries used in our evaluation corresponded to the shown SQL query, with varying instantiations of i and j.

Each document had five 10-literal string attributes and five integer attributes, one of which was a monotonically increasing sequence number. The queries were defined with comparison predicates on the sequence number field and matched one inserted document each as illustrated in Listing 4.1. Since queries were added during the preparation phase, the number of active queries remained constant for the duration of each experiment; thus, InvaliDB's matching nodes were exclusively occupied with matching queries against incoming updates, but not with other tasks such as adding or canceling query subscriptions. To minimize the effect of network congestion and message (de-)serialization, we configured change notification throughput to be steady at roughly 17 matches per second (i.e. 1 000 per 1-minute experiment). To this end, we inserted documents in random order (i.e. specifically not sorted by sequence number) and only allowed 1 000 matching queries per experiment; abundant queries matched non-existent sequence numbers, so that they would not produce change notifications.

Coping With Garbage Collection

Providing consistently low response times requires minimizing delays in every stage of the data processing pipeline. In particular, alleviating the impact and maximizing the predictability of stop-the-world garbage collection (GC) pauses [GTSS13] has proven to be the single most important aspect of catering for predictable real-time performance in our Java-based software stack. Fortunately, the object life-cycle in our real-time data-processing system allows for efficient and low-pause GC, since there are only few long-lived objects (queries, results, etc.), but many objects that are dereferenced shortly after creation (specifically after-images). Since further only relatively few queries and objects can be matched in a single thread of execution in a given timeframe, the memory footprint of the individual matching nodes was often minimal: Spawning one JVM per core and using the generational **G1 garbage collector** [Ora17a], the CPU became a bottleneck before

JVM heap space exceeded 100 MB under write-heavy workloads. In contrast, read-heavy workloads with many active real-time queries per matching node incurred higher memory overhead and consequently also longer garbage collection pauses (see Section 4.4.4).

We experienced occasional GC pauses during some experimental runs that completely dominated latency and therefore spoiled the results. To prevent this implementation artifact from affecting the evaluation of our approach, we repeated individual runs when latency measurements were significantly distorted due to GC. This does not interfere with the validity of our results, because comparable results could be achieved using a Java Runtime Environment (JRE) that avoids garbage collection altogether [CTW05]. We used the Oracle JRE with G1 garbage collection instead of a pause-free garbage collector, because the only freely available one we are aware of (Shenandoah [Ora18c]) was not usable [Ric15] when we developed our prototype, while commercial alternatives such as Azul's Zing [Azu18] were prohibitively expensive.

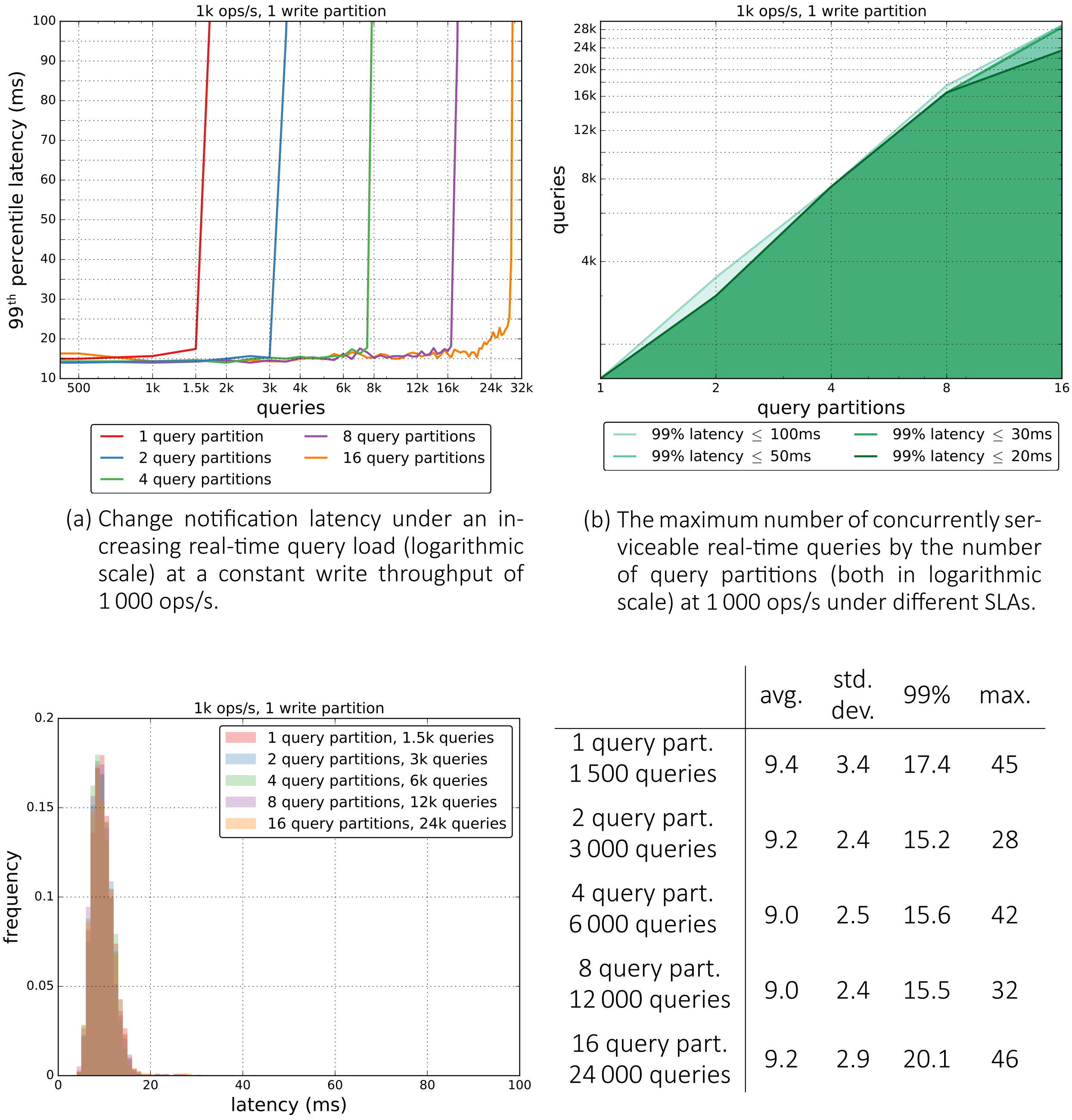
Apart from GC within InvaliDB, our measurements were also affected by GC within the benchmarking client. To further prevent client-side GC from distorting our latency measurements, we provided the client Java runtime with 30 GB RAM and configured it to perform garbage collection only in idle phases between measurements.

4.4.2 Scaling With Query Load

To assess InvaliDB's read scalability, we compared the number of concurrently sustainable real-time queries for InvaliDB clusters with varying numbers of query partitions. Figure 4.3 illustrates experimental results for clusters with 1, 2, 4, 8, and 16 query partitions at a fixed write throughput of 1 000 operations per second. Since we employed a single write partition in all of these experiments, the number of query partitions was equal to the number of matching nodes for each deployment. Please note that we plotted the number of active queries (a) and the number of query partitions (b) on a logarithmic scale.

The line plot of 99th percentile notification latency by query load (a) shows that response times were consistently below 30 ms for all InvaliDB deployments until at least 1 500 active real-time queries per matching node: As the individual InvaliDB clusters were overwhelmed at loads beyond 1 500 (1 node) and 1 800 (16 nodes) queries per node, matching workload started queueing up and latency spiked as a consequence. Since we increased workload in increments of 500 queries, per-node throughput measurements were more accurate (and therefore appear to be slightly better) for larger clusters: The single-node deployment could manage 1 500 and failed at 2 000 queries, whereas the 16-node deployment could sustain 29 000 and failed at 29 500 concurrent queries (i.e. between 1 812 and 1 844 queries per node under the assumption of perfectly even workload distribution).

The comparison of sustainable throughput under different SLAs (b) further makes explicit that sustainable query load scales linearly with cluster size and that even strict latency requirements can be satisfied near peak load. The latency histogram (c) and the provided



(a) Change notification latency under an increasing real-time query load (logarithmic scale) at a constant write throughput of 1 000 ops/s.

(b) The maximum number of concurrently serviceable real-time queries by the number of query partitions (both in logarithmic scale) at 1 000 ops/s under different SLAs.

(c) Latency distribution at 1 000 ops/s with 1 500 real-time queries per query partition (about 80 % of system capacity).

	avg.	std. dev.	99%	max.
1 query part. 1 500 queries	9.4	3.4	17.4	45
2 query part. 3 000 queries	9.2	2.4	15.2	28
4 query part. 6 000 queries	9.0	2.5	15.6	42
8 query part. 12 000 queries	9.0	2.4	15.5	32
16 query part. 24 000 queries	9.2	2.9	20.1	46

(d) Measured latency in milliseconds (average, standard deviation, 99th percentile, max.) at 1 000 ops/s with 1 500 queries per query partition (about 80 % of system capacity).

Figure 4.3: Read scalability: sustainable number of concurrent real-time queries for InvaliDB deployments with a single write partition and varying query partitions.

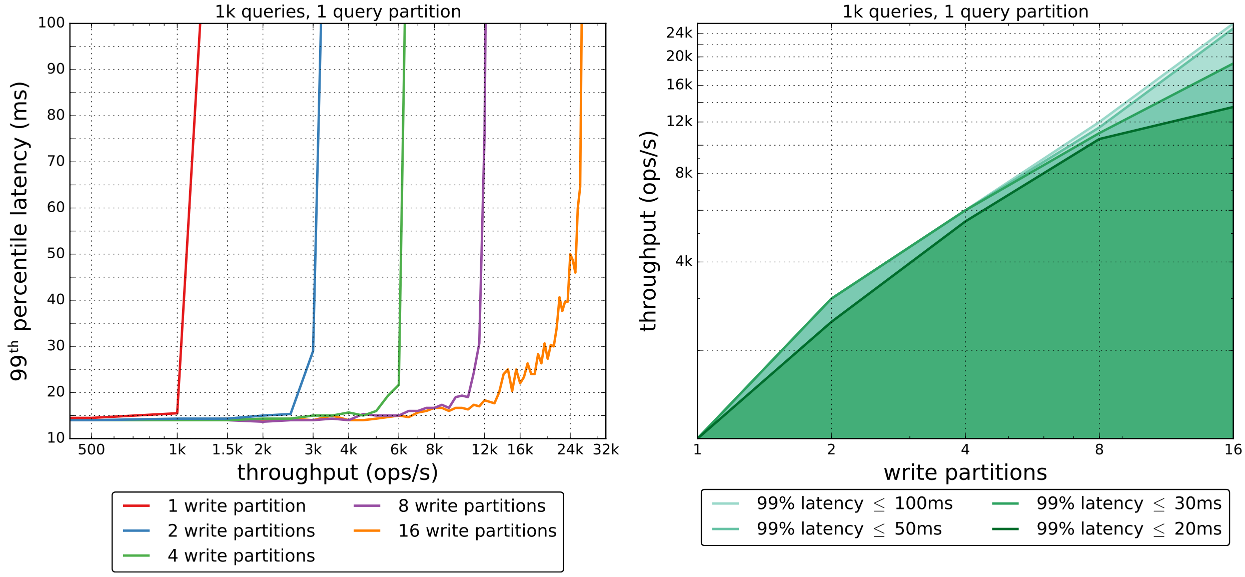
latency values (d) finally illustrate that all InvaliDB clusters displayed highly similar latency distributions under identical relative load of 1 500 queries per node: Average notification latency was measured between 9.0 ms and 9.4 ms with standard deviations ranging from 2.4 ms to 3.4 ms, while outliers never exceeded 50 ms. The largest InvaliDB configurations exhibited a slight performance degradation compared to the other configurations (99th percentile). As reported in [GSW+17, Table 2], however, we did not experience comparable issues with equally sized configurations on more potent hardware. Since processes of our test setup's virtualization layer had to share CPU cores with InvaliDB matching nodes in the InvaliDB configuration with 16 matching nodes (cf. page 100), we consider this to be a measurement artifact rather than an issue intrinsic to InvaliDB.

Appendices A and C.1 contain experimental results on the read performance and scalability of our InvaliDB prototype with 2, 4, 8, and 16 write partitions at various throughputs.

4.4.3 Scaling With Write Throughput

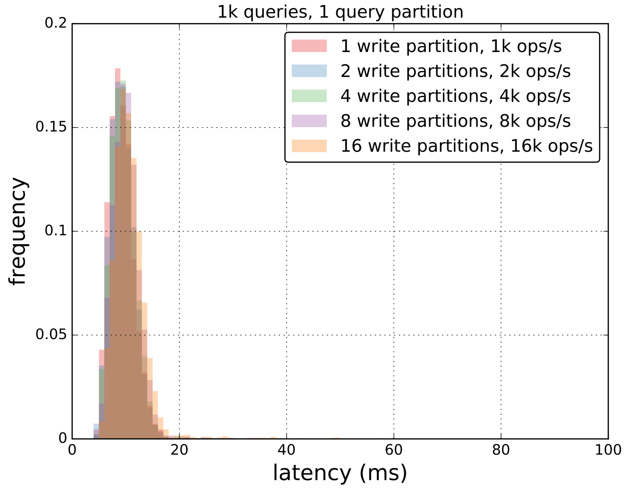
Our evaluation of InvaliDB's write scalability was conducted in similar fashion to the above-described read scalability evaluation. As shown in Figure 4.4, we contrasted sustainable throughput for InvaliDB clusters with a single query partition and 1 to 16 write partitions, using a fixed read workload of 1 000 active real-time queries. Here, write throughput in ops/s (a) and the number of write partitions (b) are on a logarithmic scale.

Considering the sustainable matching operations per second (matches/s) as the number of active real-time queries multiplied by write operations per second, our results show that InvaliDB achieves lower overall matching performance under write-heavy workloads than under the read-heavy workloads discussed above. In more detail, the 99th percentile latencies achieved under increasing write throughput (a) were measured below 30 ms until about 75 % and below 50 ms until about 95 % of system capacity, while latencies under the read-heavy workloads did not exceed 30 ms before reaching the performance limit. Similarly, sustainable write throughput (b) was feasible until 26 000 ops/s with 1 000 active queries (26 million matches/s) for the largest InvaliDB cluster, while 29 000 queries could be served at 1000 ops/s (29 million matches/s) with the same configuration under the read-heavy workload. Thus, our experiments show that sustainable matching throughput is slightly reduced under write-heavy workloads with 1.0 million (1 write partition) to 1.625 million (16 write partitions) matches/s per node, compared to 1.5 to 1.8 million matches/s per node under the read-heavy workloads. The latency histogram (c) and the tabular summary (d) illustrate that latency characteristics are nonetheless consistent across all InvaliDB configurations for the same relative workload: At roughly two thirds of system capacity, the average notification latencies ranged from 8.8 ms to 10.3 ms with standard deviations between 2.3 ms and 3.5 ms and outliers that were always well below 100 ms. Similar to the experiments under read-heavy workloads, the largest InvaliDB cluster displayed the worst outliers and therefore slightly deteriorated average latency



(a) Change notification latency under an increasing write throughput (logarithmic scale) serving 1 000 concurrent real-time queries.

(b) Sustainable write throughput by the number of write partitions (both in logarithmic scale) serving 1 000 active real-time queries under different SLAs.



(c) Latency distribution with 1 000 active real-time queries at 1 000 ops/s per write partition (about 66 % of system capacity).

	avg.	std. dev.	99%	max.
1 write part. 1 000 ops/s	8.8	2.4	15.5	34
2 write part. 2 000 ops/s	8.9	2.3	15.0	27
4 write part. 4 000 ops/s	9.0	2.3	15.6	30
8 write part. 8 000 ops/s	9.5	2.4	16.8	32
16 write part. 16 000 ops/s	10.3	3.5	21.9	79

(d) Measured latency in milliseconds (average, standard deviation, 99th percentile, max.) with 1 000 active real-time queries at 1 000 ops/s per write partition (about 66 % of system capacity).

Figure 4.4: Write scalability: sustainable write throughput for InvaliDB deployments with a single query partition and varying write partitions.

and standard deviation compared to the clusters with fewer nodes. We think there are two likely explanations for this occurrence. First, given that our test setup was only barely sufficient to support a 16-node InvaliDB cluster (cf. Section 4.4.1), CPU contention might have affected matching node performance for the largest InvaliDB configuration, but not the smaller ones. As a second possibility, garbage collection in the Storm spouts for write ingestion might have caused occasional latency stragglers at high throughput[15].

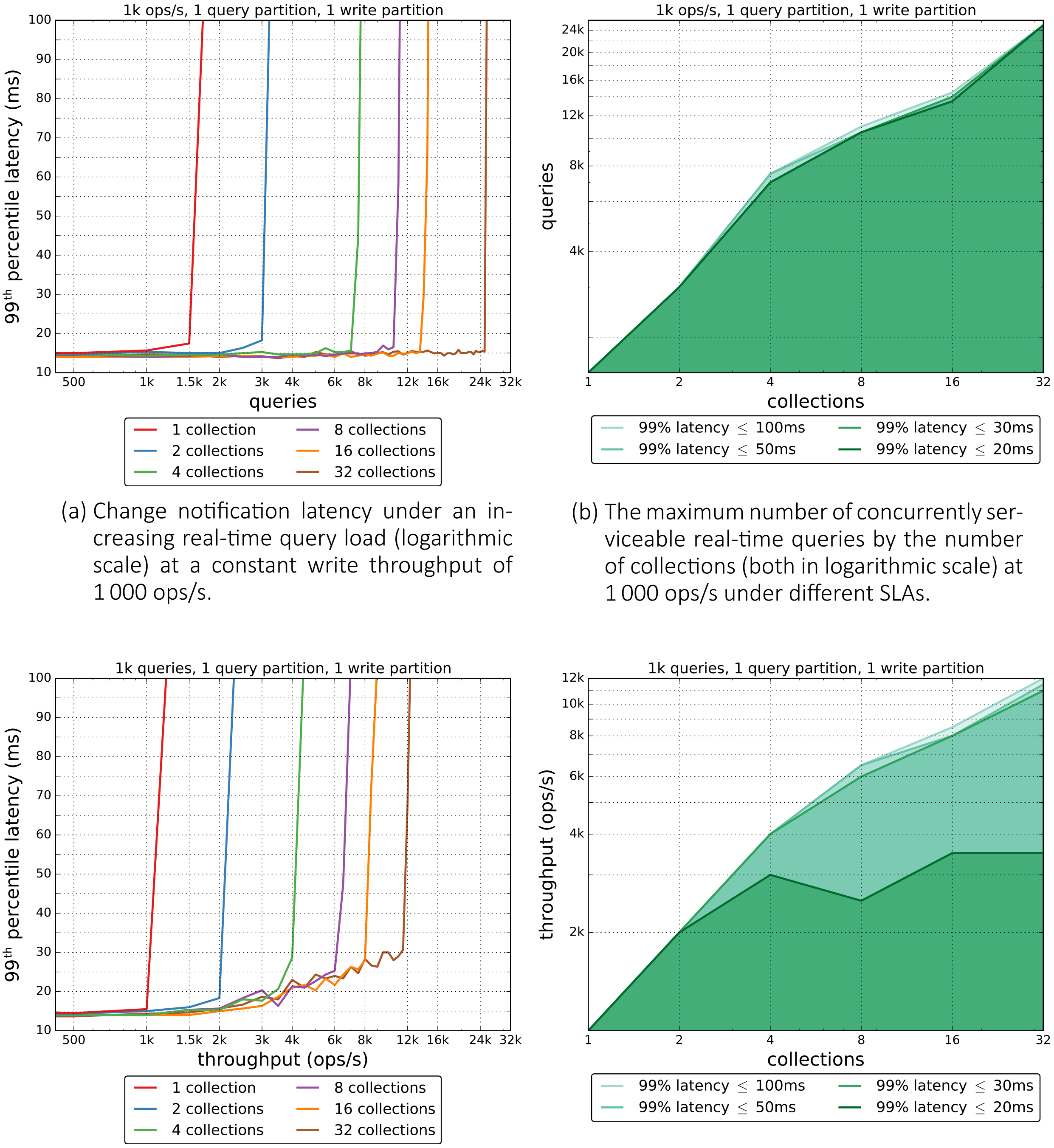
While our experiments confirm that InvaliDB scales linearly with write throughput, they also reveal that latency characteristics and sustainable throughput degrade noticeably under write-heavy workloads when compared to measurements made under read-heavy workloads. This can be explained by the overhead for (de-)serializing after-images which grows with write throughput (cf. Section 4.3.2), but not with the number of active real-time queries. We refer to Appendix B and Appendix C.2 for experimental results regarding write performance and scalability of InvaliDB configurations with 2, 4, 8, and 16 query partitions under various read workloads.

4.4.4 Efficiency of Multi-Tenant Setups

In Section 4.4.2 and Section 4.4.3, we demonstrated that even extreme workloads with thousands of concurrent real-time queries and thousands of write operations per second are feasible for InvaliDB. For industry applications at Baqend, though, we observed that real-time queries are often used for small websites or web applications with only a few hundred concurrent real-time query subscriptions and moderate write throughput. Setting up and maintaining a dedicated InvaliDB cluster for each of these lightweight use cases would be prohibitive because of the involved operational complexity and the associated costs. To support small-scale applications efficiently, an InvaliDB cluster therefore needs to be capable of multi-tenancy (cf. Section 3.1.1), so that different applications can share the same InvaliDB deployment. The plots in Figure 4.5 show read and write performance of an InvaliDB cluster with only a single matching node in a multi-tenant scenario where overall matching throughput is spread across many database collections. As before, we used a constant write throughput of 1 000 ops/s for our read workloads and a fixed number of 1 000 active real-time queries for the write workloads. We provide experimental results on multi-tenant read performance and scalability at higher throughput and on multi-tenant write performance and scalability under workloads with more active real-time queries in Appendix A.2 and Appendix B.2, respectively. Please note that the number of active queries, the number of query partitions, write throughput, and the number of write partitions are again plotted on a logarithmic scale for better readability.

When an InvaliDB cluster serves only a single tenant, all incoming write operations have to be matched against all active real-time queries. In a multi-tenant setup, in contrast, an incoming write operation only has to be matched against the real-time queries active

[15] We only monitored garbage collection in the matching nodes.



(a) Change notification latency under an increasing real-time query load (logarithmic scale) at a constant write throughput of 1 000 ops/s.

(b) The maximum number of concurrently serviceable real-time queries by the number of collections (both in logarithmic scale) at 1 000 ops/s under different SLAs.

(c) Change notification latency under an increasing write throughput (logarithmic scale) serving 1 000 real-time queries.

(d) Sustainable write throughput by the number of collections (both in logarithmic scale) serving 1 000 real-time queries under different SLAs.

Figure 4.5: Multi-tenant performance: sustainable matching performance for an InvaliDB deployment with a single matching node where real-time queries and write operations are distributed equally across varying numbers of collections.

on the written collection: When all real-time queries are evenly distributed across n tenants, matching workload is thus reduced to $\frac{1}{n}$ compared to a single-tenant workload with the same write rate and the same overall number of active real-time queries. The plots of notification latency (a) and concurrently sustainable real-time queries (b) confirm that our prototype exhibits this behavior in practice. The sustainable query load scales linearly with the number of tenants until about 7 000 queries and then flattens off slightly from there. The decline of efficiency is caused by the increased memory usage and the associated management overhead which, in turn, drains additional processing power (leaving less CPU time for the actual query matching) and which also causes more frequent, more expensive, and more time-consuming garbage collections (thus provoking occasional latency stragglers).

For write-heavy workloads, 99th percentile latencies below 20 ms were never achieved under throughput higher than 3 500 ops/s (c). This can be explained by the overhead for after-image parsing which also grows linearly with throughput, thus increasing base latency and reducing the computing resources left for query matching. As another consequence of this parsing overhead, sustainable throughput grew linearly with the number of tenants until around 4 000 ops/s, but not as steeply under higher throughput (d).

4.5 Summary

We focused on unsorted filter queries in our evaluation, because the filtering stage implements InvaliDB's unique two-dimensional partitioning scheme and is therefore the critical component for enabling InvaliDB's scalability: As discussed in Section 3.3, the filtering stage processes all queries and all write operations, while all subsequent processing stages only have to cope with the output of the respectively previous stage partitioned by query and are therefore unlikely to become a bottleneck. As the only exception, the join stage can be scale-prohibitive, since certain join queries are infeasible for incremental maintenance (cf. Section 3.3.2). However, arbitrary θ-join queries are neither supported by any current real-time database nor by MongoDB, the underlying database for our InvaliDB implementation. Therefore, we did not consider join queries in this evaluation.

Summing up the above experimental results, our Java-based InvaliDB implementation provides predictably low latency even under high load. Under various workloads, all InvaliDB deployments displayed notification latencies consistently below 30 ms in the 99th percentile when under moderate load and still below 50 ms when approaching system capacity. Through our experimental evaluation, we have thus shown that it is feasible to build an InvaliDB implementation that scales linearly with read and write workloads and is also capable of serving real-time queries in a multi-tenant environment.

In the next chapter, we will illustrate InvaliDB's practical usefulness by describing the integration of our prototype into the existing database middleware Orestes where it serves two different purposes. First, InvaliDB extends the functionality of the base system by adding push-based real-time queries to the otherwise pull-based query interface. Second, it improves both throughput and latency of the existing pull-based query mechanism by enabling consistent query caching through low-latency result change notifications.

5 Integrating InvaliDB With the Orestes Database Middleware

> "Everything should be made as simple as possible, but not simpler."
>
> —Albert Einstein

We presented four challenges of providing push-based real-time queries over database collections in Chapter 1 and discussed limitations of the current state of the art in Chapter 2. We then addressed three of these four challenges in Chapter 3 and Chapter 4 by designing and implementing a real-time query mechanism that is scalable (cf. Challenge C_1), able to support a wide range of query expressions (cf. Challenge C_2), and compatible with legacy databases (cf. Challenge C_3). In this chapter, we finally address the remaining Challenge C_4 by devising an abstract, yet expressive client API for push-based real-time queries: The API is *abstract* in the sense that it hides complexity where possible and *expressive* in the sense that it provides powerful queries and fine-grained control over what information is pushed to the client. To achieve this goal, we integrate our InvaliDB implementation into an existing database middleware, Orestes, and extend its purely pull-based query interface to support push-based real-time queries.

In Section 5.1, we provide an overview over the Orestes database middleware and its unique caching approach through which it achieves globally low read latency. We also review its data model, access control mechanisms, and query API. Next in Section 5.2, we describe how InvaliDB is integrated with Orestes in the *Quaestor* architecture and how it enables two otherwise infeasible use cases: The first use case is consistent caching of database *query results* which becomes effectively possible through InvaliDB's ability to detect result changes and thus to invalidate cached query results once they become stale. The second use case consists in collection-based *real-time queries* for end users which directly leverage InvaliDB's ability to generate result change deltas. In Section 5.3, we introduce the client API extension for push-based real-time queries and illustrate how self-maintaining queries provide an abstraction for developing real-time functionality that imposes little to no complexity overhead in comparison to the original pull-based API. We further present event stream queries as an advanced abstraction that provides more sophisticated control over which data is delivered to the client. In Section 5.4, we then detail how these abstractions can be used to implement complex real-time queries, such as aggregations and joins. Lastly, we conduct a performance evaluation of the overall system for both use cases in Section 5.5 and conclude the chapter in Section 5.6.

5.1 Orestes: A Database Middleware for Globally Low Latency

Relational database systems have been considered the one-size-fits-all solution to many data management problems over decades [SC05]. In recent years, however, a plethora of so-called NoSQL databases have emerged that are optimized for specific usage scenarios and often sacrifice strong consistency or query expressiveness in favor of increased scalability, error resilience, or other non-functional properties [GWFR16] [SF12]. In the context of the web, systems that optimize read latency have received particular attention [VV16] [LVA+15] [BVF+12] [GLS11], because end-to-end latency directly translates to page load times and thus to user satisfaction and business success [Gri13]. Since the speed by which information can be transmitted is ultimately limited by physical network latency, cross-site replication is often used as a means to bring data closer to the clients [TPK+13] [CDE+13] [BBC+11] [CDG+06] [DHJ+07].

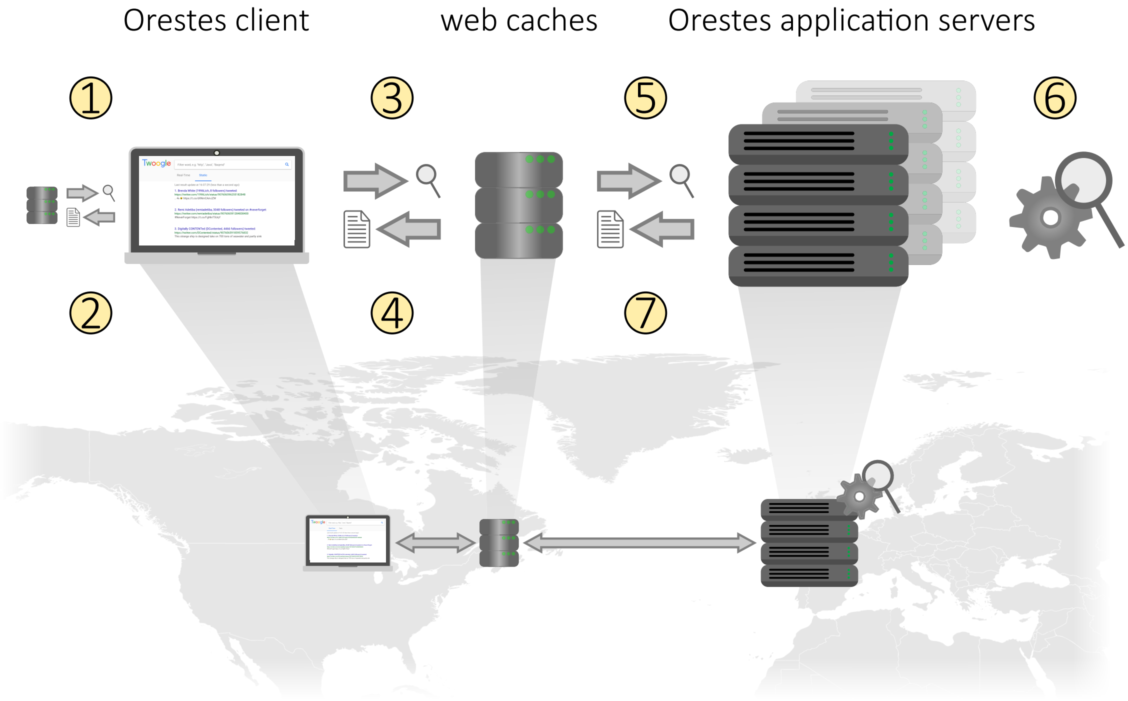


Figure 5.1: The Orestes database middleware uses the readily available web caching infrastructure to replicate data across the globe.

Orestes (Objects RESTfully Encapsulated in Standard Formats) [Ges18] [GBR14] [GB12] [GB10] is a database middleware that takes another approach by employing common HTTP [NMM+99] web caches to accelerate database reads. It follows the Database-as-a-Service (DBaaS) [HIM02] model and communicates directly with clients through an HTTP/REST [Fie00] interface. Orestes does not require setting up proprietary and globally distributed database replicas in order to scale out read performance; instead, it serves data copies from the web's generic caching infrastructure and thus implements a mechanism that can be seen as lazy on-demand geo-replication.

In order to keep clients from accessing stale copies, the Orestes application server keeps track of all cached data items that are written during their respective cache lifetime. When a cached data item is updated, the application server then actively removes it from all **invalidation-based caches** such as the content delivery network (CDN) edge nodes, so that subsequent requests to those caches will not yield old data. However, there are also purely **expiration-based caches** that cannot be purged by the application server; these caches retain any data item until its respective time to live (TTL) runs out. An example for a purely expiration-based cache is the browser cache that is located in the user device. To leverage expiration-based caches without introducing the possibility of stale reads, the application server provides the clients with the **Cache Sketch** [GSW$^+$15], a fixed-size encoding of all currently stale cached resources. Being closely related to Bloom filters [Blo70], the Cache Sketch is a probabilistic representation of a set of items that supports containment checks: While it may identify actually fresh resources as stale (occasional false positives), it is guaranteed to never identify an actually stale resource as fresh (no false negatives). By renewing the client's Cache Sketch periodically, staleness is effectively bounded[1] by the refresh interval. Using the Cache Sketch to identify stale caches, an Orestes client can reliably determine whether (1) a given database record can safely be fetched from a nearby expiration-based cache or whether (2) it needs to be retrieved from the invalidation-based CDN or the original database (in case of a CDN cache miss).

Figure 5.1 illustrates the basic idea of how an Orestes client performs a read operation. Initially, an application issues an HTTP request for a database record using the Orestes client library (**1**). If the requested resource is present in the expiration-based browser cache *and* is known to be fresh, it is directly delivered from within the client's device (**2**). Otherwise, the resource is either not present in the browser cache *or* it is present, but also possibly stale. In this case, the request is revalidated and thus forwarded to the CDN (**3**). Since the CDN is invalidation-based, the revalidation request is answered by the CDN, if the requested data item is available (**4**). Otherwise, the revalidation request is forwarded to the original backend (**5**) where the application server evaluates the request, performing database lookups and otherwise required computation in the process (**6**). The generated response is then sent back to the requesting client, updating all invalidation-based caches along the way (**7**). Thus, the client finally receives the up-to-date database record.

In order to promote a clear understanding of the Orestes cache coherence scheme, we contrast conventional HTTP caching with Orestes' caching approach in Figure 5.2. In the depicted setup, a database record (document) is served by the application server on the left to two reading clients on the right: One is a conventional HTTP client (notebook) and the other one uses the Orestes protocol for cache coherence (tablet). Both clients have already accessed the database record before, so it is available at the invalidation-based CDN and also in the purely expiration-based browser cache for each client. Staleness be-

[1] For more details on the consistency guarantees provided by the Cache Sketch, see [GSW$^+$15, Sec. 2] and [GSW$^+$17, Sec. 3].

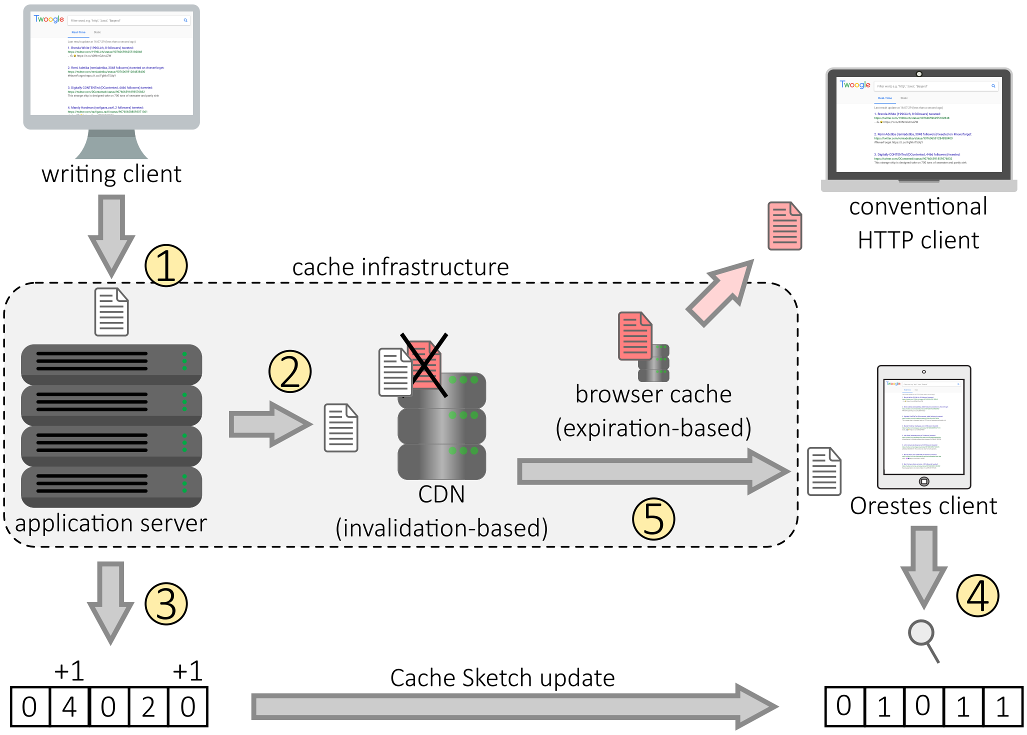


Figure 5.2: The Orestes cache coherence scheme: While conventional HTTP clients will observe stale data when accessing purely expiration-based caches, Orestes clients know which caches are stale through the Cache Sketch data structure. Thus, Orestes clients only retrieve cached data when it is known to be fresh and they issue revalidation requests otherwise.

comes an issue when the database record is updated by the client on the top left (**1**). Even though the application server purges the invalidation-based CDN cache (**2**), the outdated copy is still present in the browser caches and will be returned to the conventional HTTP client on request (red arrow). As explained above, however, the Orestes application server does not only invalidate caches where possible, but also updates its Cache Sketch (**3**) and transmits it to the Orestes client. Since the Orestes client can uncover the record as stale using the Cache Sketch, it does not use the browser cache copy and instead requests the record from the CDN. Since outdated copies are invalidated by the application server, the client knows that cache hits produce fresh data[2]. In case of a cache miss, the request is automatically upgraded to a revalidation request and will thus also deliver a fresh record.

It should be noted that the Orestes application server and client store the Cache Sketch in slightly different ways: The application server's Cache Sketch is based on a *counting Bloom filter* [BMP$^+$06] which supports removing items that are no longer stale[3]. The client

[2] In our description, we use a simplified model that abstracts from the latency of propagating updates from the origin server to CDN edge nodes. For details, we refer to [Ges18].

[3] Since expired data items are implicitly evicted from all caches, cached data items do not have to be kept beyond their TTL. Removing them decreases the risk of false positives and thus the chance of unnecessary invalidations.

Cache Sketch, in contrast, is a more space-efficient Bloom filter [Blo70] that only supports additive changes; item removal is reflected by replacing the client's Cache Sketch with a newly generated one.

5.1.1 Data Model & Query Expressiveness

Acknowledging the difficulty of choosing the right system for a given set of application requirements, Orestes is designed as a **polyglot database middleware** that unifies various backends behind a single HTTP/REST interface [GFW$^+$14]. Orestes hides implementation details of the underlying data storage systems and provides additional guarantees and functionalities on top, for example multi-object ACID[4] transactions [Wit16] and the cache coherence scheme for globally low latency detailed above. In order to leverage the benefits of different databases, Orestes allows application developers to declare functional and non-functional requirements through schema annotations. The choice of a suitable combination of data storage systems is then delegated to an Orestes subsystem, the **Polyglot Persistence Mediator (PPM)** [SGR15]. Through the PPM, Orestes effectively decouples application requirements from concrete technologies: Since the exposed interface is independent of the underlying databases, the PPM can adapt to different requirements transparently. Depending on the materialization model, data may either be partitioned across distinct databases according to a *sticky partitioning* scheme or all data may be stored in a single *primary database* that is used for queries, while updates are only periodically applied through write-behind caching [SGR15].

Orestes enforces an object-oriented and strongly typed schema that supports CRUD[5] operations. The query API supports boolean filter expressions over single aggregate-oriented collections without joins. While arbitrary relationship types can be implemented using object references that are automatically resolved by the client library, the denormalized data model promotes expressing 1:1 and 1:n relationships through nesting entities. Even though there is no hard dependency on any underlying database technology, the Orestes research prototype and the commercial Backend-as-a-Service (BaaS) product that originated from it, **Baqend** [Baq18a], use MongoDB as the primary database. In this thesis, we also assume MongoDB as the primary database on top of which we deploy InvaliDB to provide collection-based real-time queries for Orestes.

5.1.2 Access Control

Orestes features a permission system [Baq18c] that enforces access on the granularity of collections and entities through access control lists (ACLs). Collection-level permissions are defined in the schema, while object-level permissions are defined in each object: Every

[4] ACID: atomicity, consistency, isolation, durability [HR83].

[5] CRUD: create, read, update, delete.

data item has an `acl` attribute that is used to explicitly whitelist (allow) or blacklist (deny) individual users or user groups for read and write access. For every registered user, there is a corresponding record stored in the **user collection**. A special user record is associated with anonymous users (i.e. users that are currently not logged in). User groups (cf. *role-based access control (RBAC)* [SFK00]) are represented by **role collection** records, each of which stores its respective group members as a set of user IDs. By default, there are two groups to represent administrators and logged-in users, but custom groups can be created.

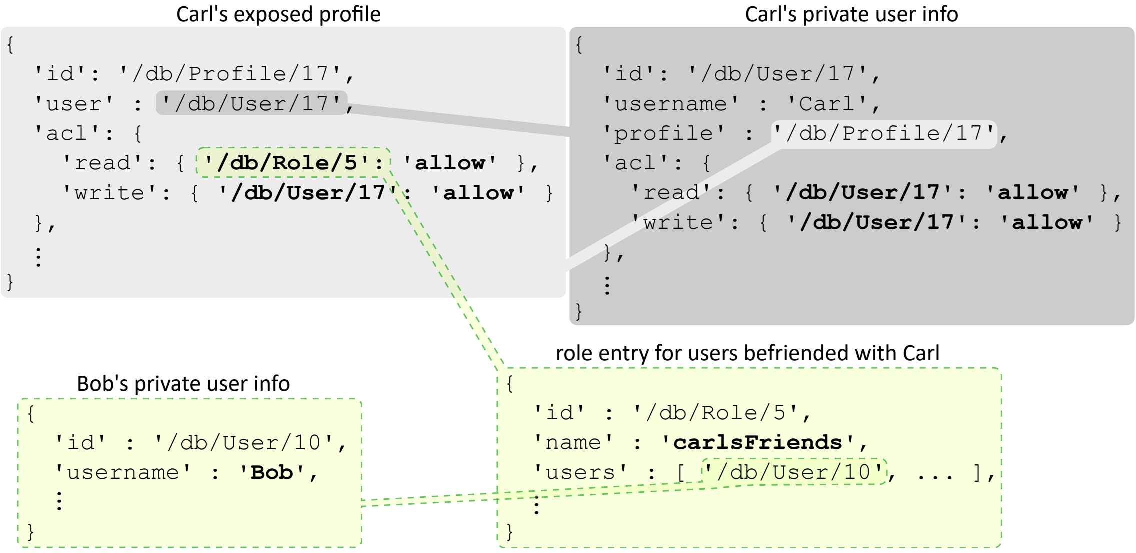
Whether or not a user may read, update, or delete a given data item is determined through the following checks[6]:

1. *Is the user a superuser?*
   Members of the administrator group are granted access immediately, without any further check.
2. *Is the user blacklisted?*
   If the user or one of her associated groups is blacklisted, access is denied.
3. *Is the user whitelisted?*
   If the object in question has an empty whitelist, it is implicitly accessible to the public (and thus to any user). Otherwise, the accessing user is only approved, if the user herself or an associated group is whitelisted.

Since its characteristic caching approach works on the level of entities, Orestes does not support a more fine-grained **permission granularity** (e.g. attribute-level access control). On the one hand, a rather coarse permission granularity may improve caching efficiency by increasing the chance of different clients accessing the same data. But on the other hand, it also has implications on data modeling, because access to a data item can only be granted either entirely or not at all. As illustrated in Figure 5.3, one approach to modeling entities with both private and public attributes is to partition them into separate database collections. In the example, the sensitive information on Carl is encoded in the standard `User` record (dark gray) which can only be accessed by himself (and implicitly by administrators). All shareable information is externalized into a referenced `Profile` record (light gray) with a separate set of permissions. Thus, only Carl's profile may be queried by another user such as Bob who is in Carl's friends list (role: `carlsFriends`), but Carl remains the only user whitelisted for write access on his private as well as his public user data. Since the client library can resolve the profile reference automatically, it appears to the application developer as though Carl's profile were nested inside the private user object.

While the specificities of permission semantics are out of scope of this section, it is important to note that the information required for a check of permissions is bounded and

[6] Corresponding rules are applied on the schema level to determine whether a user is allowed to *insert* data into a given collection.



Carl's exposed profile

```
{
  'id': '/db/Profile/17',
  'user' : '/db/User/17',
  'acl': {
    'read': { '/db/Role/5': 'allow' },
    'write': { '/db/User/17': 'allow' }
  },
  ⋮
}
```

Carl's private user info

```
{
  'id': '/db/User/17',
  'username' : 'Carl',
  'profile' : '/db/Profile/17',
  'acl': {
    'read': { '/db/User/17': 'allow' },
    'write': { '/db/User/17': 'allow' }
  },
  ⋮
}
```

Bob's private user info

```
{
  'id' : '/db/User/10',
  'username' : 'Bob',
  ⋮
}
```

role entry for users befriended with Carl

```
{
  'id' : '/db/Role/5',
  'name' : 'carlsFriends',
  'users' : [ '/db/User/10', ... ],
  ⋮
}
```

Figure 5.3: Carl's user information is partitioned according to visibility, so that public data (light gray) can be queried by Carl's friends (e.g. Bob), while the private data (dark gray) remains visible only to Carl himself and administrators.

can be assembled efficiently for any given object: It essentially amounts to (1) the ID of the accessing user, (2) the IDs of the user's groups, and (3) a blacklist and a whitelist for the accessed object itself and its database collection, respectively. All this is available at the application server and can be piggybacked to revalidation requests, so that CDN edge nodes are able to enforce access control and thus directly serve not only public, but also non-public data to clients. The permission check itself can further be expressed as a query predicate (cf. page 61). Thus, InvaliDB supports Orestes' access control mechanisms through *permission-sensitive real-time queries*.

5.2 Quaestor: Extending Orestes With InvaliDB

In order to protect clients from stale reads, the Orestes application server purges all cached copies of a data item from the invalidation-based CDN whenever the original data item is updated or deleted. Since original and copy share the same primary key, invalidation detection is trivial in the context of single-record caching. For query caching, however, invalidation detection is more involved and very similar to computing real-time query change deltas: In order to reliably identify all cached query results that are invalidated by a given update, the matching stati of the updated entity before and after the write operation have to be taken into account with respect to each cached query (cf. Section 1.1). In addition, it might be necessary to consider data that is not part of the actual result, for example items in the offset of a sorted query's result (cf. Section 3.3.2). When burdened with the task of invalidation detection, the application server becomes a scalability bottleneck (see our discussion of Meteor, RethinkDB, and Parse in Section 2.3). In order to generalize the Orestes caching approach from single-record caching to query result caching in a scalable way, we developed an architecture named **Quaestor** [GSW+17] that integrates Orestes with InvaliDB. In this architecture, InvaliDB serves two main purposes: First, it detects stale query results and thus makes query caching feasible for Orestes. Second, it provides low-latency query result updates to enable real-time queries for Orestes clients. This section describes the general system design of Quaestor and the interaction between Orestes and InvaliDB. In Section 5.3 and Section 5.4, we assume a more user-oriented perspective and discuss the client API for real-time queries in detail.

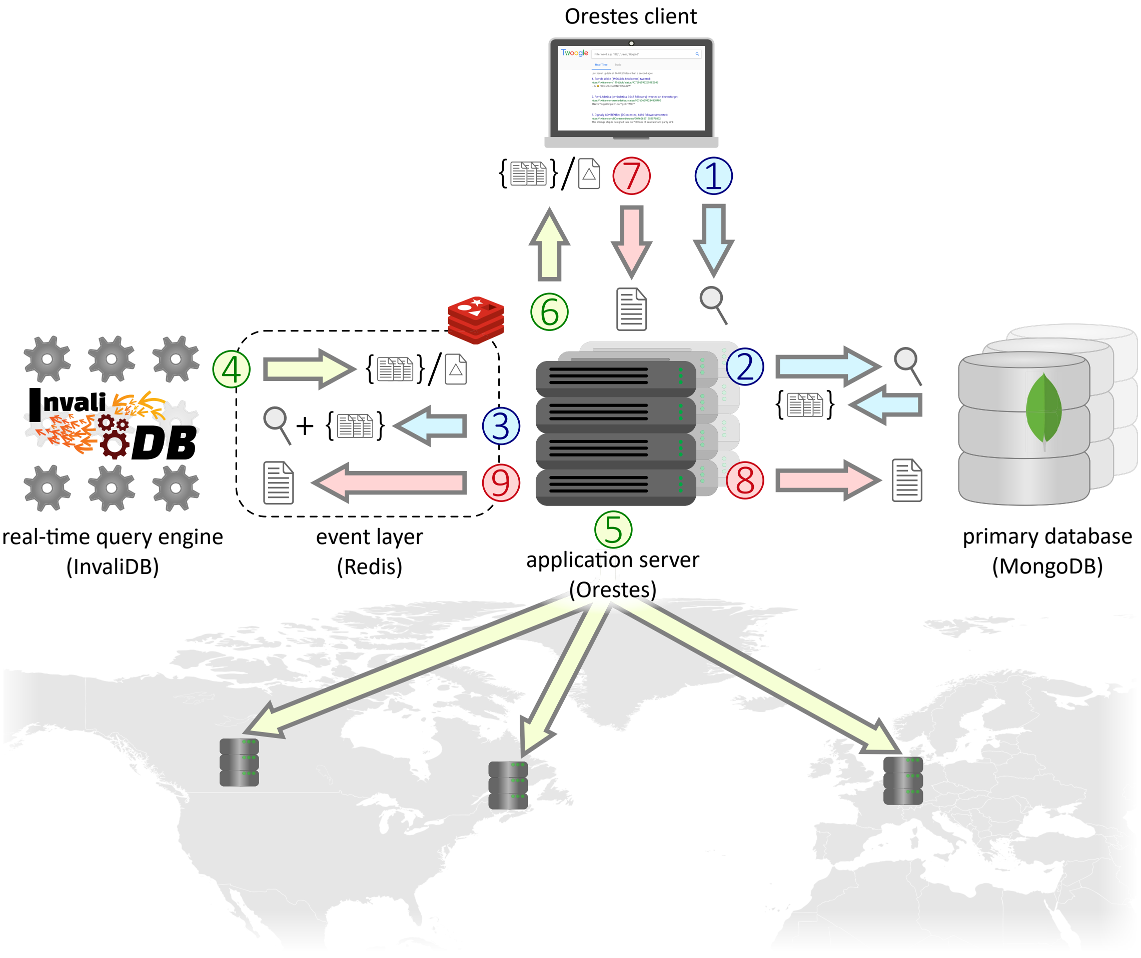


Figure 5.4: The Quaestor architecture ties InvaliDB into Orestes in order to make query caching feasible and provide client-facing real-time queries.

Figure 5.4 shows a high-level overview over the integrated system and the information flow for write operations (red arrows), ad hoc and real-time queries (blue arrows), and invalidations and real-time change notifications (green arrows). When a client poses an **ad hoc query** (1), the application server executes the query against the primary database (2) and sends the result back to the client. If the query is considered cacheable by the internal decision model, the query is also asynchronously propagated to InvaliDB (3), together with the query result, the TTL, and other information required for query matching[7].

[7] To monitor the result of a sorted query with offset, for example, InvaliDB requires all data items in the offset in addition to the actual result (see Section 3.3.2).

When InvaliDB detects a change that modifies the cached query's result, an *invalidation message* is sent to the application server (**4**) which, in turn, purges the query from the invalidation-based CDN caches (**5**). A cached query is also implicitly removed from InvaliDB on invalidation or – when no invalidation occurs – eventually removed when its TTL expires. When the registered query is a **real-time query**, it is *always* sent to InvaliDB with all corresponding information (**3**). In contrast to an invalidation query, though, a real-time query does not only generate a single message for the CDN, but a continuous stream of *change events* for the client (**6**) and it stays active until it is deregistered explicitly by the client or implicitly, e.g. through connection loss. InvaliDB is able to detect changes, because it receives an **after-image** (i.e. a full entity representation)[8] for any executed write operation: Whenever a client inserts, updates, or deletes a data item (**7**), the receiving application server applies the write operation (**8**) and sends the after-image of the modified entity to InvaliDB (**9**) where it is matched against all active queries.

5.2.1 Result Representation & TTL Estimation

When caching a query, an internal decision model has to choose a result representation to store in the caches. Listing 5.1 shows how the same query result can be cached as a list of references (a) or as a list of fully specified objects (b). The **ID list** representation is only invalidated when a new item enters the result (`add` match type), when a matching item leaves the result (`remove` match type), or when an item changes its position within the result (`changeIndex` match type). A fully specified **object list** is invalidated on the same events and also whenever a matching object is updated (`change` match type), even if the modification does not even relate to the matching criterion or sorting key. Since the `change` match type subsumes the `changeIndex` match type, object lists (`add`/`remove`/`change` match types) tend to be invalidated more often than ID lists (`add`/`remove`/`changeIndex` match types). On the other hand, loading a query result by ID list incurs an additional round-trip, because the individual object references have to be resolved client-side, i.e. the client has to fetch every object in the result individually *after* the ID list has been retrieved.

```
[ 'obj1', 'obj2', 'obj3' ]
```

(a) An *ID list* only contains object references that need to be resolved in an additional round-trip.

```
[ { id : 'obj1', name : 'Alice' },
  { id : 'obj2', name : 'Bob' },
  { id : 'obj3', name : 'Carl' } ]
```

(b) An *object list* contains fully specified objects (no additional round-trips), but is invalidated whenever a result member is updated.

Listing 5.1: We consider two different ways to represent a query result.

[8] As described in Section 3.1, we assume that the application server only provides after-images and no before-images.

For each cached query result, Quaestor's **TTL estimator** [GSW$^+$17, Sec. 4.2] assigns a TTL that defines when the result is going to be evicted from expiration-based caches. However, since the actual expiration time cannot be predicted accurately, the TTL estimator can only approximate the actual time until the next invalidating write occurs. In doing so, it has to weigh the risk of evicting data too early (**underestimating**) against the risk of retaining data too long (**overestimating**): When actually fresh data is removed from the cache, potential cache hits become revalidation requests that are forwarded to the original database. This increases both request latency and processing overhead in the backend. Choosing longer TTLs avoids premature expiration, but also necessitates tracking more stale resources in the Cache Sketch and thus raises the chance of false positives in the staleness check (cf. page 113).

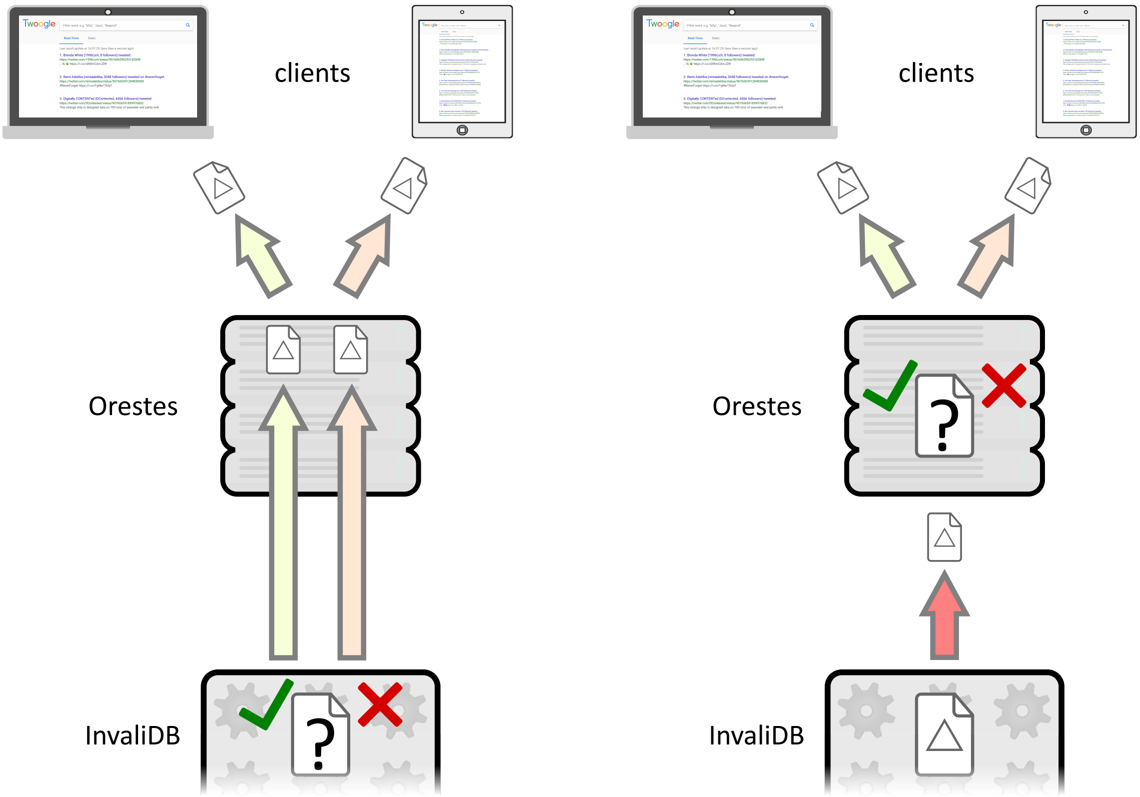
Since this chapter focuses on Quaestor's real-time component, we refer to [GSW$^+$17, Sec. 4.2] for details on the TTL estimator and the decision model. Further, [Sch15] and [SGDY16] provide background on the stochastic processes involved in estimating cache expiration times, choosing query result representations, and deciding whether or not a query is considered cacheable.

5.2.2 Enforcing Access Control

Access control as described in Section 5.1.2 is enforced at different levels within the Quaestor architecture. **Write permissions** are always validated in the application server, because it is the only component that handles inserts, updates, and deletes. **Read permissions**, in contrast, are either validated in the application server when the query is initially executed or in a CDN edge node when a client is accessing cached data. Even though **read permissions for real-time queries** could be enforced within InvaliDB always, it is sometimes more efficient to enforce them in the application server.

For illustration, consider the different scenarios depicted in Figure 5.5 where two clients subscribe to the same real-time query at the same application server. If InvaliDB is responsible for enforcing access control (a), the application server registers an individual real-time query for each client subscription and simply passes on the change events received from InvaliDB. If the data emitted for each subscription is highly dependent on the individual client's permissions, this approach minimizes traffic between InvaliDB and application server. For example, consider a real-time query on the `user` collection: Since clients without administrator privileges are only allowed to read their own `user` object, subscribing to all changes on the `user` collection will generate distinct change events for different non-administrator clients.

If, in contrast, the application server itself validates client permissions before serving the change events (b), it can coalesce both client subscriptions (green and orange) into a single InvaliDB subscription with elevated access rights (red). The application server makes sure to receive all relevant change notifications (*completeness*) by registering the shared



(a) When InvaliDB is responsible for checking permissions, every client subscription corresponds to a distinct real-time query in InvaliDB.

(b) When the application server validates access rights before delivering real-time events, client subscriptions for the same query can be coalesced.

Figure 5.5: Depending on the query, access control can be enforced in the application server or within the InvaliDB cluster during query matching.

InvaliDB subscription with access rights that subsume those of the subscribed clients[9]. At the same time, the application server filters out denied data for each client individually, so that users only receive data they are allowed to see (*correctness*). This approach is inefficient when the subscribed query exhibits high permission-based selectivity as the above example query on the `user` collection: Events for any change on the user table will be transferred from InvaliDB to the application server, even though most of them will be filtered out. On the other hand, in scenarios where different clients often receive *the same* events (e.g. when most data in the subscribed collection is public), coalescing subscriptions can reduce traffic between InvaliDB and application server, because every change event is emitted by InvaliDB only once and duplicated in the application server.

The above examples show that many factors influence what the optimal strategy for access control is. It should be noted, though, that the choice between application server and InvaliDB for the point of access control is only available for unsorted queries: For explicitly sorted queries, access control is necessarily enforced within InvaliDB, because indices can-

[9] In the extreme case, the shared subscription is registered with administrator privileges, so that permission checks within InvaliDB are effectively disabled.

not be generated without determining the matching status first (see Section 3.3.2) which is dependent on the subscriber's permissions. In consequence, applying access rules in InvaliDB is the more expressive option, while the approach illustrated in Figure 5.5b can rather be seen as a performance optimization that is only applied to specific queries.

5.2.3 A Language-Agnostic Real-Time Communication Protocol

Similar to the HTTP/REST interface that enables language-agnostic handling of CRUD operations and queries, we implemented a **WebSocket**-based [MF11] message exchange protocol for real-time queries that can be used independently from client libraries. We chose WebSockets over competing protocols or API definitions, because they are supported by every major browser vendor [Can18] and because they enable bi-directional duplex communication between the client and the server; this feature is critical to establish multiple customized real-time query subscriptions over a single connection. In contrast, *WebRTC* [HHE15] is mainly designed for browser-to-browser communication and *Server-Sent Events (SSE)* [H$^+$15] does not allow sending data from the client to the server after the initial subscription; therefore, SSE is not appropriate for handling multiple subscriptions over the same connection[10]. *Web Push* [TDR16] and the *native push mechanisms* in iOS and Android would also not have been feasible, because they only support uni-directional messaging from the server to the client and because they incur prohibitive message delivery times in the order of many seconds or even minutes [YAD14].

Using our protocol, an application can subscribe to a real-time query and subsequently receive the corresponding notifications through a cleartext or TLS-encrypted [Ris15] WebSocket connection with an Orestes application server. In the following, we overview which messages are exchanged between the Orestes client and server, but we omit details on the individual message types. We refer to Appendix D for a description of the individual message attributes and examples. The high-level client API for real-time queries will be covered in-depth in Section 5.3.

A real-time query is always initialized through a **subscribe** message sent from the client to the server. On subscription, the client provides a universally unique identifier (UUID) that is associated with the subscription; all subsequent messages relating to it will carry this identifier. After subscription, the client only listens for messages sent by the server, until the query is terminated. A real-time query can be terminated in two ways: explicitly through an **unsubscribe** message (issued by the client) or implicitly through an *error* message (issued by the server or triggered by connection loss). Between subscription (i.e. initialization) and unsubscription (i.e. termination) of a real-time query, messages are only sent from the server to the client. To guarantee correct semantics, the client has to process messages in strict order of their occurrence.

[10] Specifically, establishing new subscriptions and canceling active ones would require sending requests over a separate connection.

The first message sent by the server is a **result** message which contains all currently matching items in the specified order (if any) and serves as an indicator that the subscription has been activated successfully. Each message from this point on is either a **match** message (indicating a result update) or an **error** message (indicating query termination). By default, the client will receive a match message for every result update, so that the result can be maintained up-to-date locally by taking the corresponding action for each event (see result maintenance example in Section 5.4.4).

5.3 Extending the Pull-Based Query API by Push-Based Queries

In order to make real-time queries available to developers, we extended the existing purely pull-based Orestes query API by push-based mechanisms; unless stated otherwise in the text, all extensions discussed in this section have been implemented in the Baqend JavaScript SDK [Baq18b] and are therefore eligible for public use. Since the Orestes/Baqend **JavaScript client** is geared towards the web-centric application scenarios that motivate our work (cf. Chapter 1), we deliberately restrict all code examples and the discussion to the context of JavaScript. Please note, however, that client libraries written in other languages[11] can be implemented in similar fashion, since client-server communication is language-agnostic: CRUD operations and queries use the generic REST/HTTP interface and real-time queries rely on the WebSocket-based communication protocol described in the previous section.

In the remainder of this section, we discuss the API for real-time queries in detail. First in Section 5.3.1, we describe how our API extension for **self-maintaining queries** can be used, illustrating the simplicity of our design and the similarity to common pull-based queries with an example. In Section 5.3.2, we then explain how **observables** [Hus17] are leveraged in our API extension to provide low-latency result updates. Finally in Section 5.3.3, we turn towards the API extension for **event stream queries**, before we provide usage examples in Section 5.4 and an experimental evaluation in Section 5.5.

5.3.1 Self-Maintaining Queries

As illustrated in the introductory Section 1.2, self-maintaining queries behave very similar to regular pull-based queries, the most significant distinction being that they *update themselves* and thus can prevent application state from becoming stale. In more detail, both the pull-based ad hoc query and the push-based self-maintaining query accept a callback function that specifies how a query result will be processed once it is received. But while this callback function is only executed once in case of the regular pull-based que-

[11] In fact, the performance evaluations of InvaliDB (Section 4.4) and Quaestor (Section 5.5) have been carried out using a Java implementation of our real-time query API.

ry, the self-maintaining query also invokes the callback function with the **updated result** whenever a relevant change occurs.

For illustration, consider a shared task list that is frequently updated by different instances of a client application. All clients synchronize their interaction through a shared database that holds all tasks on the task list. An example query and a callback function that displays a given result are defined in Listing 5.2. The query refers to all records in the `Task` collection (line 1) that are not completed (line 2). Tasks with an early deadline are ranked higher (line 3) and result size is limited to 10 data items (line 4). The callback function (`console.log`) simply logs the query result to the console, when a result is received.

```
1 var query = DB.Task.find()
2   .equal('completed', false)
3   .ascending('deadline')
4   .limit(10);
5 query.resultList(console.log); // pull-based ad hoc query
6 query.resultStream(console.log); // self-maintaining query
```

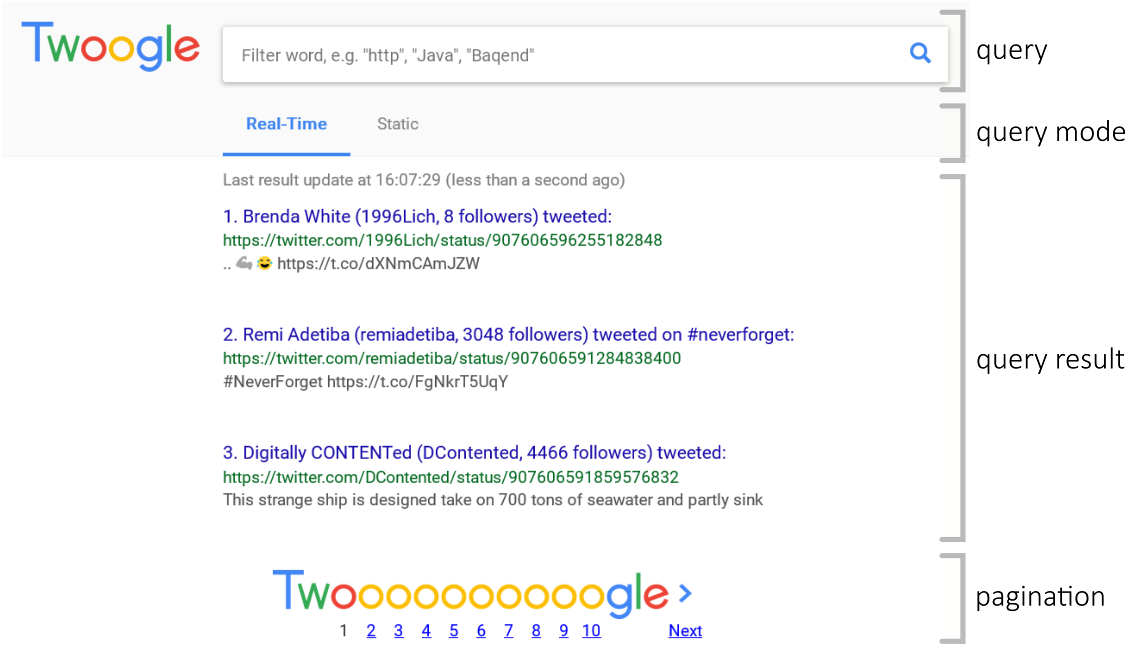
Listing 5.2: The query selects the 10 most urgent open tasks in the `Task` collection and the callback function logs the result to the console. From a developer's point of view, executing the query as a self-maintaining real-time query is very similar to executing it as a regular pull-based query.

In order to display the 10 most urgent open tasks, an instance of the application (i.e. an Orestes client) has to execute the query. The basic Orestes query API only provides the `resultList` method for pull-based queries (line 5). Since a **pull-based query** does not update the result by itself, however, the application's view of the data has to be actively refreshed again and again to make sure that no updates are missed. This pattern corresponds to Meteor's poll-and-diff mechanism that emulates real-time queries by periodically reevaluating a query. As discussed in Section 2.3.1, it has several critical problems. First, it introduces client-side staleness, because there is no indicator for the client to know when the result has changed. Second, this pattern is inefficient from the client's point of view, because the entire query result is transferred over the network repeatedly – even when nothing has changed. Third, it is hard to scale, because load on the application server and the database increases linearly with the number of clients, irrespective of whether the result actually changes or even whether data is written at all.

Acknowledging the inadequacy of pull-based access patterns in reactive domains, our API extension adds `resultStream` as a method for push-based self-maintaining queries (line 6). With a **self-maintaining query**, the top-10 list is not only logged to console once or periodically (as would be the case with a regular query), but every time and *immediately* when a task enters the top-10, is updated within the top-10, or leaves the top-10. Even though the callback function is invoked with a complete result on every call, only *incremental change deltas* are transmitted after the initial result; the updated result is produced by applying the incremental change information to the outdated result. This scheme ensures a small network footprint and also facilitates high scalability, because server-side processing overhead is very low. Computational overhead for query matching does not become a bottleneck, because it is handled by InvaliDB in a scalable fashion.

Example Application: Twoogle

The presented extension to the pull-based query API makes it possible to introduce real-time semantics to an existing purely pull-based application with ease. To this end, a developer has to replace calls to `resultList` with calls to `resultStream` first[12]. If the query changes dynamically (e.g. in case of a user-defined search query as illustrated below), the developer further has to make sure that the current real-time query subscription is canceled, before the updated real-time query is subscribed. As a showcase application, we implemented a social media search app named **Twoogle**[13] that provides an interface for searching Twitter messages. A continuous process running on the Orestes application server receives live tweet messages as they are created by users anywhere on the world, each of which is inserted into the `Tweet` database collection. Users can then search all tweets in the continuously evolving database by typing a filter expression into the web interface. The currently active real-time search is canceled whenever the search query is updated.



Last result update at 16:07:29 (less than a second ago)

1. Brenda White (1996Lich, 8 followers) tweeted:
https://twitter.com/1996Lich/status/907606596255182848
.. 💪😂 https://t.co/dXNmCAmJZW

2. Remi Adetiba (remiadetiba, 3048 followers) tweeted on #neverforget:
https://twitter.com/remiadetiba/status/907606591284838400
#NeverForget https://t.co/FgNkrT5UqY

3. Digitally CONTENTed (DContented, 4466 followers) tweeted:
https://twitter.com/DContented/status/907606591859576832
This strange ship is designed take on 700 tons of seawater and partly sink

Figure 5.6: Twoogle is a social media application that enables user-defined real-time queries over live Twitter messages.

[12] To guarantee that result updates are reflected correctly in application state, query callback functions have to be implemented in idempotent fashion (cf. Section 1.2).

[13] To see a running version of Twoogle, visit `https://twoogle.info`.

```
 1 function search(filter, limit, offset, realtime) {
 2   if (subscription) {// cancel real-time query, if active
 3     subscription.unsubscribe();
 4   }
 5   //formulate query:
 6   query = DB.Tweet.find()
 7     .matches('text', new RegExp(filter))
 8     .descending('createdAt')
 9     .offset(offset)
10     .limit(limit);
11   //execute query:
12   if (realtime) {
13     subscription = query.resultStream(renderResult);
14   } else {
15     query.resultList(renderResult);
16   }
17 }
```

Listing 5.3: The `search` function is executed whenever the user modifies one of the query parameters, for example by typing in the search field.

The Twoogle user interface (UI) is depicted in Figure 5.6. Similar to other search engines, Twoogle displays the result of a user-defined filter query, newest matches first. The result is structured in pages, so that users can retrieve older messages by selecting a later page. A feature that is probably unique to Twoogle is the ability to toggle the **mode of query execution**: Users can choose between push-based real-time queries and pull-based static queries by clicking the corresponding button. When the query is executed in pull-based fashion, the search result is generated on page load and whenever the user modifies the query, for example by typing into the search field or navigating to another page. With a self-maintaining query, in contrast, the user-defined result also refreshes itself whenever a new matching tweet is inserted into the `Tweet` collection or when a matching tweet is updated or leaves the result.

Listing 5.3 shows how the switch between real-time and static behavior is implemented in the `search` function which is triggered whenever one of the query parameters (line 1) is modified. First, the current real-time subscription is canceled, if there is one (lines 2 to 4). Subsequently, the current query parameters are used to create a query object (lines 6 to 10). Finally, the query is executed in pushed-based fashion (line 13) or in pull-based fashion (line 15), depending on whether the boolean `realtime` parameter equates to `true` or `false` (line 12). The only additional complexity that is introduced by using the push-based query (as opposed to using the pull-based query) consists in the subscription object that is required to cancel the current subscription before creating a new one. Simply resubscribing to a new real-time query without deactivating the obsolete subscription would result in faulty behavior, because both the old and the new real-time query would update the client view. The `renderResult` callback function (invoked in lines 13/15, definition omitted for clarity) generates an HTML representation of the query result and assigns it to a specific child in the Domain Object Model (DOM). An invocation of the rendering function overrides all effects of previous invocations by simply replacing the rendered search result, so that a Twoogle app instance in real-time mode exhibits virtually the same behavior as an app instance in static mode that is refreshed repeatedly.

Twoogle exemplifies how client-side staleness can be reduced with minimal development effort by employing a self-maintaining query. To highlight the congruence between pull-based and push-based query expressiveness as a distinguishing feature of our real-time query implementation, we implemented pagination in Twoogle using limit and offset clauses: This solution illustrates that even an application with sophisticated queries can be transitioned easily from static to real-time behavior. As discussed in Section 2.3, many other real-time databases only provide limited support for sorted queries: For example, the original Firebase only supports sorting by a single attribute, while RethinkDB supports sorted real-time queries with limit, but not with an offset clause. Parse does not support sorted real-time queries at all. We refer to the official Twoogle announcement [Win17a] for implementation details.

5.3.2 Observables & Subscriptions

In the previous section, we presented a use case for self-maintaining queries and illustrated how real-time query subscriptions can be established and canceled[14]. Now, we go into more detail on subscriptions and explain how we harness the expressiveness of real-time queries through the **observer pattern** [Osm12].

Every real-time query produces an **event stream** (i.e. a sequence of query result updates) that is represented by an abstraction called **observable**. An observable maintains a list of so-called **observers**, each of which is a collection of callback functions. Whenever new data becomes available in the stream, the observable notifies each observer, so that they can apply their callback functions to the new data. To add a new observer to an observable, it is necessary to create a **subscription**. This subscription can be canceled (*unsubscribed*) to remove its respective observer from the observable.

In simple words, a client has to subscribe to an observable and provide an observer in order to define real-time behavior. The following three **handler callback functions** can be wrapped by an observer:

- **next**: *Defines what to do when an update arrives in the stream.*
  For self-maintaining queries, this callback function receives the complete updated query result. For event stream queries, it receives individual change events.

[14] Listing 5.3 shows that the `resultStream` method returns an object (line 13) that can be used to unsubscribe from the query (line 3).

- **error** (optional): *Defines what to do when there is an error.*
  This callback receives a server-side error, for example when the client issues a real-time query with insufficient access rights. There are no possibilities for client-side errors apart from connection aborts which are addressed by the complete handler (see below). An `error` will implicitly cancel the corresponding subscription. An error event contains the subscription ID, the name of the problem (`reason`) and a more elaborate problem description (`message`) that should point the user towards the problem. For more details, see Appendix D.
- **complete** (optional): *Defines what to do when the network connection is closed.*
  On a `complete` event, the corresponding subscription will automatically be canceled. *Self-maintaining queries* will transparently reconnect by default (see `reconnect` option below), so this handler can usually be ignored for them. *Event stream queries*, on the other hand, do not support automatic reconnects: They will just silently stop working when disconnected, unless a `complete` function is provided.

Introducing the `error` and `complete` handler functions as second and third arguments allows explicit handling of maintenance errors and disconnects. Since they are optional, though, our real-time query interface can also be used like a simple callback-based query interface: In the Twoogle example (Listing 5.3, line 13), we thus only provide the `next` handler for processing result updates. When a subscription is created like this, the underlying observable is created *implicitly* and cannot be accessed by the application.

```
1 // one single observable:
2 var stream = query.resultStream();
3 // Multiple subscriptions on the same observable:
4 var sub = stream.subscribe(onNextA, onErrorA, onCompleteA);
5 var otherSub = stream.subscribe(onNextB, onErrorB, onCompleteB);
```

Listing 5.4: Different subscriptions can be created on top of a single observable.

Listing 5.4 shows that an alternative approach is to first *explicitly* create the real-time query's observable by calling `resultStream` without arguments (line 2) and then create subscriptions on top of it (lines 4/5). Sharing the same observable between different subscriptions on the same query is more efficient, because real-time updates are only transferred once from server to client for all subscriptions and not for each one individually. However, the semantics are also slightly different, because only the first `subscribe` call (line 4) triggers delivery of an initial result: When the second subscription (`otherSub`, line 5) is created after the initial result has been emitted by the `stream` observable, the handler function `onNextB` will not be called until a new result is emitted (e.g. because a change occurs or because the stream is resubscribed after a connection error).

Options for Self-Maintaining Query Subscriptions

By design, self-maintaining queries are straightforward to use and do not require any configuration. However, clients can customize their behavior by providing an `options` argument as the first parameter to the `resultStream` function. There are two optional parameters available for self-maintaining queries:

- **reconnect** (default: `-1`): *Determines how often the query is resubscribed after connection loss or runtime error (negative values indicating infinite retries).*
  By default, a self-maintaining query will be resubscribed and the full initial result will be delivered again whenever the WebSocket connection drops or when a query maintenance error is received. Since the full query result (and not just changed records) will be transmitted on resubscription, however, reconnecting can impose significant communication overhead for large results. To shield against this kind of performance leak, clients can specify a non-negative integer to override this behavior. In this case, they should also provide a `complete` handler which is going to be called after the specified number of reconnect attempts has been exhausted.
- **collapse** (default: `-1`):[15] *Prescribes the minimal delay in milliseconds between two subsequent result updates referring to the same entity.*
  Without explicit configuration, the Orestes server will propagate every write operation that has any effect on the subscribed query's result. This may be undesirable in the case of high-frequency updates on individual objects in the query result. As an example, consider an object with a counter attribute that is incremented several hundred times a second. To prevent a subscribed client from being overwhelmed by an abundance of updates (or simply to save network traffic), the `collapse` parameter can be set to a positive integer to indicate the number of milliseconds over which updates on the same entity should be collapsed (cf. collapsing on page 63). With a value of `200`, for instance, the server will only send the current entity representation every 200 ms and skip intermediate updates.

5.3.3 Event Stream Queries

Calling the `eventStream` method on a query object creates an observable that encapsulates all **data modifications** relevant to the query's result. In contrast to a self-maintaining query, an event stream query does not invoke the callback function with a complete result on every change, but instead with a **change notification** containing details on how exactly the result was transformed; among other attributes (see Appendix D), it carries a database record (`data`), a `matchType`, and the triggering `operation`. The syntax for creating an observable, subscribing, and unsubscribing is identical to that for self-maintaining queries. Once subscribed to the event stream of a query, the client will essentially receive

[15] This parameter has not been implemented for the JavaScript client at the time of writing.

an event for every database entity matching at subscription time and for every entity that enters the result, leaves the result, or is updated while in the result. Like a pull-based filter query, an event stream **filter query** returns all entities in a collection that match the filter criterion. An event stream query with an empty filter condition is called **simple query**, because it does not entail any query processing and simply returns all entities in the collection. A query that is defined with `limit`, `offset`, `ascending`, `descending`, or `sort` is called **sorted query**[16], because it reflects that the client is expecting an ordered result: Only a *sorted* event stream query will deliver events with an `index` attribute.

While simple queries can be very useful (for example to monitor the entirety of all operations on a particular collection), they should be handled with care, because they can produce a great number of events within a short time for high-throughput collections. Likewise, sorted queries should be avoided when filter queries would be sufficient, because enforcing order on huge results is relatively expensive. To bound the resources that can be tied up by a single subscription, the application server rewrites both pull-based and push-based queries, so that they respect a fixed upper limit (500 items at the time of writing). Further, every query is *implicitly ordered by ID* to guarantee that the output of ordering the result and capping it to a given size is deterministic (cf. Section 3.1.2). If the unique ID was not implicitly included in every query's sorting key, different objects in a result could receive identical sorting keys in which case they would be ordered randomly.

```
1 var explicitLimit = query
2   .offset(5)
3   .limit(495); // explicit limit
4 var implicitLimit = query
5   .offset(5); // implicit limit: 495 (= 500 - offset)
6 var cappedLimit = query
7   .offset(5)
8   .limit(500); // limit is capped to 495 (= 500 - offset)
```

Listing 5.5: Because of the implicit limit of at most 500 items per result, these three queries are effectively equivalent.

Since the maximum **limit** is implicitly enforced, the three queries given in Listing 5.5 are equivalent. With an explicit limit within the permitted range, the result will contain no more than the specified number of items (lines 1 to 3). However, a sorted query without the optional limit clause will be registered with the maximum permitted limit (lines 4/5): The underlying rule is that `offset + limit <= 500` must always hold, because items in the offset have to be maintained by InvaliDB just like items in the actual result (cf. Section 3.3.2). In consequence, the limit can never assume values greater than `500 - offset` (lines 6 to 8). Correspondingly, queries with an **offset** greater than 499 are invalid and will cause an error.

[16] Full-text search and some geo queries are sorted, but have not become part of the public Baqend API, yet.

Options for Event Stream Query Subscriptions

Similar to self-maintaining queries, event stream queries can be configured by providing an `options` object as the first argument on an `eventStream` call. But while a self-maintaining query necessarily receives the initial result and all events that are required to maintain it, an event stream query explicitly allows restricting events, for example to those with specific match types or to those that were triggered by specific operations (cf. subscription parameters in Section 3.2.4). In addition[17] to the options available for self-maintaining queries, event stream queries adhere to the following parameters:

- **matchTypes** (default: `['all']`): *Restricts the delivered events to those with the specified match types.*
  The default delivers all events with the most specific applicable match type (`add`, `change`, `changeIndex`, or `remove`). If only a specific subset of match types are relevant, a client can choose any combination of match types to listen for. In cases where the difference between new and updated items is irrelevant, clients can also use match type `match`: Subscribing to events with match type `match` will yield the same events as the combination of `add`, `change`, and `changeIndex`, but the match type of the received events will always be `match`.
- **operations** (default: `['any']`): *Restricts the delivered events to those that were triggered by the specified operations.*
  Per default behavior, events will *not* be sorted out based on their operation, but clients can choose any combination of `insert`, `update`, `delete`, and `none` to narrow down the kind of matches they receive.
- **initial** (default: `true`): *Specifies whether or not the initial result is to be returned.*
  If set to `true`, every entity matching the query at subscription time will be delivered with match type `add`, irrespective of whether and which restrictions are imposed on operations and match types (see the other options above). If set to `false`, clients will only receive an event when the result changes.
- **beforeImages** (default: `false`):[18] *Specifies whether or not before-images are to be delivered when available.*
  If set to `true`, change notifications are delivered with before-images, if they are available (cf. page 66 in Section 3.2.1). If set to `false` (default), change notifications only contain after-images.

Using the `matchTypes` and `operations` options correctly requires an understanding of the involved semantics. In the next section, we will provide several examples to illustrate how

[17] The *reconnect* parameter for event stream queries should be used with caution, because reconnecting event stream queries exhibit non-intuitive semantics: As mentioned in Section 4.1.2, automatic resubscription of an event stream query can lead to duplicating events (when replaying the initial result) or losing events (without initial result), so that a reconnect may not be transparent to the application logic.

[18] This parameter has not been implemented for the JavaScript client at the time of writing.

event stream queries and self-maintaining queries operate and how they can be used to implement custom data access patterns.

5.4 Real-Time Query Semantics

The main purpose of self-maintaining queries is to introduce real-time semantics with minimal complexity. Event stream queries, in contrast, expose a high level of complexity as they provide fine-grained control over which information should be delivered to the client. This section sheds more light on the intricacies of event stream query semantics by several examples.

5.4.1 Semantics by Example: Sorted Event Stream Queries

For an illustration of how a sorted event stream query behaves, consider the following timeline where two users are working concurrently on the same data. User 1 subscribes to a sorted event stream query over the `Task` collection and listens for the corresponding events, while User 2 is working on the data:

Timestamp 0: **User 1** and **User 2** connect to the same database.
Timestamp 1: **User 2** inserts `task1`:

```
var task1 = new DB.Task({title: 'My Task 1'});
task1.insert();

//actual result: [ task1 ]
```

Timestamp 2: **User 1** subscribes to an event stream query and immediately receives an event for `task1`, because it is contained in the initial result:

```
var stream = DB.Task.find()
    .matches('title', new RegExp('My Task'))
    .ascending('title')
    .limit(3)
    .eventStream();
var subscription = stream.subscribe(event => {
    console.log(event.matchType + '/'
        + event.operation
        + (event.initial ? ' (initial): ' : ': ')
        + event.data.title + ' is at index '
        + event.index);
    });
// ... one round-trip later:
//'add/none (initial): My Task 1 is at index 0'
```

Timestamp 3: **User 2** inserts `task2`:

```
var task2 = new DB.Task({title: 'My Task 2'});
task2.insert();

//actual result: [ task1, task2 ]
```

Timestamp 4: **User 1** receives a new event regarding `task2`:

```
//'add/insert: My Task 2 is at index 1'
```

Timestamp 5: **User 2**: inserts `task3`:

```
var task3 = new DB.Task({title: 'My Task 3'});
task3.insert();

//actual result: [ task1, task2, task3 ]
```

Timestamp 6: **User 1** receives a new event regarding `task3`:

```
//'add/insert: My Task 3 is at index 2'
```

Timestamp 7: **User 2** updates `task3` in such a way that its position in the ordered result changes from index 2 to index 1:

```
task3.title = 'My Task 1b (former 3)';
task3.update();

//actual result: [ task1, task3, task2 ]
```

Timestamp 8: **User 1** is notified of `task3`'s new position through an event with match type `changeIndex`, carrying the new version of `task3`. Note that there is no event to indicate `task2`'s positional change as it follows implicitly from the update on `task3`:

```
//'changeIndex/update: My Task 1b (former 3) is at index 1'
```

Timestamp 9: **User 2** inserts `task0` which sorts before all other items in the result and therefore is assigned index 0:

```
var task0 = new DB.Task({title: 'My Task 0'});
task0.insert();

//actual result: [ task0, task1, task3 ], task2
//               <--- within limit --->  <--- beyond limit
```

At this point, all four records inserted by User 2 comply with the matching criterion. Because of the limit clause, however, only the first three of all four matching entities appear in the result: The last one (`task2`) is *pushed beyond the limit*.

Timestamp 10: **User 1** receives two events that correspond to the two relevant changes in the result:

```
//'remove/none: My Task 2 is at index undefined'
//'add/insert: My Task 0 is at index 0'
```

Note that there is no triggering operation for the first event (operation type `none`), because `task2` was not updated directly – the event is an *indirect change event*.

Timestamp 11: **User 2** updates `task3` again, so that it assumes its original title. Through this update, `task2` and `task3` swap places again (cf. Timestamp 7):

```
task3.title = 'My Task 3';
task3.update();

//actual result: [ task0, task1, task2 ], task3
//               <--- within limit --->
```

Timestamp 12: **User 1** receives the corresponding events and is thus notified that `task3` leaves the result, whereas `task2` moves back into the result:

```
//'remove/update: My Task 3 is at index undefined'
//'add/none: My Task 2 is at index 2'
```

Timestamp 13: **User 2** deletes `task3`:

```
task3.delete();

//actual result: [ task0, task1, task2 ], ~~task3~~
```

Note that the deleted entity was not part of the result.

Timestamp 14: **User 1** receives no change event, because deleting `task3` had no effect on the query result.

```
//no event is received
```

The above example illustrates that operation-related semantics are complex for sorted queries: For example, `insert` and `update` operations may trigger an item to *leave* the result (cf. Timestamps 9/10 and 11/12). Similarly (even though not shown in the example), an `add` event can be triggered by a `delete` operation, when an item is removed from the offset or the result and another item enters the result from beyond limit. When triggered by an operation on a different entity, an event may therefore be delivered with operation type `none`.

5.4.2 Real-Time Aggregations

The Orestes JavaScript client is shipped with basic support for ES7 [ECM17] observables, so that real-time query functionality can be used without requiring external dependencies.

In this section, however, we use the extended Orestes API in combination with the feature-rich *RxJS* [Rea17a] client library. RxJS is the JavaScript implementation of the **ReactiveX** [Rea18a] framework which provides easy-to-use operators for transforming or joining observable data streams. ReactiveX clients are also available in various other languages such as Java, C++, C#, Scala, Python, Go, and PHP.

Count

A common use case for event stream queries is to compute and maintain aggregates in realtime. The basic idea is to keep all information required for maintenance in an **accumulator** data structure and to recompute and output the updated aggregate value whenever an event is received; the following example is a client-side implementation of incremental maintenance of an aggregation query.

```
 1  var maintainCardinality = (counter, event) => {
 2      if (event.matchType === 'add') {
 3          counter++;
 4      } else if (event.matchType === 'remove') {
 5          counter--;
 6      }
 7      return counter;
 8  };
 9  var sub = query.eventStream({matchTypes: ['add', 'remove']})
10    .scan(maintainCardinality, 0)//update counter
11    .subscribe(console.log);//output counter
```

Listing 5.6: An event stream query can be used to implement custom real-time aggregations, e.g. a counter for the number of items in the query result.

One of the simpler aggregates over a collection of entities is the *cardinality* or *count*, i.e. the number of entities in a query result. The code in Listing 5.6 will compute and maintain the cardinality of the query result. The `maintainCardinality` function is invoked with two arguments: the current `counter` value (the accumulator) and an incoming `event`. If the event adds a new item to the result, the counter is incremented (line 3). If the event removes an item from the result, the counter is decremented (line 5). Since only `add` and `remove` events are relevant, the client subscribes to those events exclusively and *not* to `all` match types as per default (line 9). The RxJS `scan` operator (line 10) provides the initial counter value (0) and invokes the `maintainCardinality` function on every incoming event to compute the new `counter` value that will be used on the next invocation. The subscription's callback function (line 11) receives the updated counter value and prints it to the console whenever a change occurs.

Average

As an example for maintaining a more complex aggregate, consider the *average* number of goals defined for each of the tasks that match a given query. When using an accumulator data structure like in the previous example, the maintenance logic becomes significantly more complex, because the value of interest (i.e. the average) cannot be maintained on its own: Even if every change notification comes with before-images (cf. page 131) and after-images, it is still necessary to maintain the number of matching tasks (cf. `count` above) separately.

```
 1 var computeAverage = (result) => {
 2   if (result.length === 0) {
 3     return 0;
 4   }
 5   var overallGoals = result.map(task => task.goals.length)
 6     .reduce((a, b) => a + b);
 7   return overallGoals / result.length;
 8 };
 9 var subscription = query.resultStream()
10   .map(computeAverage)
11   .subscribe(console.log);
```

Listing 5.7: Using the abstraction of self-maintaining queries, keeping an average up-to-date can be done by simply recomputing it on every result change.

As an alternative approach that uses neither event stream queries nor an accumulator, Listing 5.7 shows how the specified average value can be kept up-to-date through self-maintaining queries. The procedure for computing the average from a query result is encapsulated in the `computeAverage` function (lines 1 to 8): It essentially returns `0` for an empty result (line 3) and the computed average for a non-empty result (line 7). Using a self-maintaining query (`resultStream`, line 9) guarantees that the average is derived from scratch both on subscription and whenever the result changes. The `map` operator (line 10) applies the `computeAverage` function to every emitted result and the provided callback function (line 11) logs the average value to the console, as in earlier examples.

5.4.3 Real-Time Joins

As covered in Section 3.3.2, InvaliDB supports real-time joins across collections. Even though this feature has not been implemented for our InvaliDB prototype, a real-time join query can be realized within client-side application code, even on top of databases that do not support joins at all: The basic idea is to maintain an up-to-date view of the involved subqueries and to update or recompute the joined query result whenever new data becomes available.

```sql
SELECT task.title
  FROM task
  INNER JOIN issue ON task.id = issue.relatedTask
  WHERE task.completed = false
    AND task.title LIKE '%InvaliDB%'
    AND issue.open = true
  ORDER BY task.title ASC
```

Listing 5.8: An SQL query is declarative and evaluated entirely in the database system, so that its execution does not involve client-side processing.

As an example, consider a `task` collection as in the previous examples which holds work assignments of a software development team. Further, consider an `issue` collection that contains bug reports and problem descriptions submitted by users; every issue can reference a specific task that addresses it. Listing 5.8 shows an SQL representation of a join query that returns the title of every in-progress task related to an open issue. As such, the query connects records from the `task` and the `issue` collections. Specifically, the query *joins* the results of two subqueries that describe the following data sets, respectively:

1. all uncompleted tasks that have a title containing the string `'InvaliDB'` and
2. all open issues.

A relational database system will execute this join query very efficiently using sophisticated query optimization techniques [HSH07, Sec. 4]: Before its execution, the query will be rewritten into an equivalent representation that avoids expensive operations and materialization of intermediate (subquery) results where possible.

When executed as a **client-side nested loop join** as illustrated in Listing 5.9, in contrast, there is less room for optimization. In particular, evaluating the join condition requires materializing the results of both subqueries: `invalidbTasks` is the sorted list of InvaliDB-related tasks and `issues` holds the ID of every task that is referenced by an open issue (both line 1). These two collections are kept up-to-date using two different real-time queries: `taskSubquery` updates `invalidbTasks` (lines 2 to 8) and `issueSubquery` refreshes `issues` (lines 9 to 13). These two real-time query observables are combined using the `merge` operator (line 15), so that the join condition (line 16) is reevaluated whenever an update is received for either of the two subquery results. Finally, the task title is projected from the joined records (line 17). The subscription call specifies that the query result (i.e. the ordered list of task titles) is to be printed to the console.

It should be noted that both real-time queries maintain lists of fully specified records, because the Orestes query API does not support projection at the time of writing. Thus, all tasks and issues are transferred to the client with attributes that should be removed at the server for better efficiency, but are instead removed in the application (lines 13 and 17). Even though this particular issue can be fixed by introducing projection to the query API,

```
var invalidbTasks = [], issues = [];
var taskSubquery = DB.Task
  .find()
  .matches('title', /InvaliDB/)
  .equal('completed', false)
  .ascending('title')
  .resultStream()
  .map(result => invalidbTasks = result);
var issueSubquery = DB.Issue
  .find()
  .equal('open', true)
  .resultStream()
  .map(result => issues = result.map(i => i.relatedTask.id));
var joinQuery = taskSubquery
  .merge(issueSubquery)
  .scan(() => invalidbTasks.filter(t => issues.includes(t.id)));
  .map(result => result.map(t => t.title));
  .subscribe(projectionResult => console.log(projectionResult));
```

Listing 5.9: The query corresponds to the SQL query from Listing 5.8 which also retrieves the titles of all InvaliDB-related uncompleted `task` entries that are associated with an open `issue`.

performing client-side real-time joins will remain inefficient by concept when subquery results are large or when the join condition has a high selectivity. For illustration, imagine a scenario with only one single InvaliDB-related task, but 100 open issues referencing it: `issues` would contain 100 duplicates of the same ID, even though the overall result would only hold an individual task title.

5.4.4 Implementing Self-Maintaining Queries

Self-maintaining queries are another use case for real-time functionality which we already explored: Instead of considering every incremental change on its own, the client subscribes to a stream of fresh query results. Since an event stream query intuitively provides all information required to maintain an up-to-date view of the corresponding query's result, a self-maintaining query is straightforward to implement with an event stream query. One possibility to realize a self-maintaining query without using the built-in `resultStream` method is illustrated in Listing 5.10.

The actual maintenance logic is encapsulated in the `maintainResult` function (lines 1 to 18). Whenever an event is processed, the relevant information is extracted from the event first (lines 2/3), then the result is updated (lines 5 to 16) and, finally, a copy[19] of the result is returned (line 17). Depending on the event's match type (`type`), the extracted `object` is

[19] By returning a copy instead of the original result, we shield result maintenance from side effects; since variable scope is limited to the maintenance function, the result cannot be modified externally.

processed in different ways: While `remove` and `add` events can be applied by simply deleting (line 8) or inserting (line 15) the `object`, respectively, `change` and `changeIndex` events make it necessary to replace it: To this end, the `object` is first located and then removed (cf. while-loop, lines 6 to 12); in the process, its current position in the result is stored in the `oldPosition` variable (line 11). After removal, the up-to-date `object` is inserted again (line 15), either at the position specified by the event, at the previous position, or as the first element. Similar to earlier examples, the `scan` operator (line 20) once again maintains an accumulator data structure by invoking `maintainResult` on every event[20]. The subscription callback (line 21) finally prints the updated result to the console initially and on change.

The illustrated `eventStream` query subscription behaves exactly like a self-maintaining `resultStream` query subscription as described in Section 5.3.1. In fact, the internal implementation of `resultStream` queries is very similar to the provided code example; there are only minor differences, e.g. for the implementation of automatic reconnects.

```
 1 var maintainResult = (result, event) => {
 2   var object = event.data;
 3   var type = event.matchType;
 4   var oldPosition = 0;
 5   if (['change', 'changeIndex', 'remove'].includes(type)) {
 6     while (oldPosition < result.length) {
 7       if (result[oldPosition].id == event.data.id) {
 8         result.splice(oldPosition, 1);
 9         break;
10       }
11       oldPosition++;
12     }
13   }
14   if (['change', 'changeIndex', 'add'].includes(type)) {
15     result.splice(event.index || oldPosition, 0, object);
16   }
17   return result.slice();
18 };
19 var subscription = query.eventStream()
20   .scan(maintainResult, [])
21   .subscribe(console.log);
```

Listing 5.10: Similar to a numeric aggregate, a query result can be maintained fresh by applying all updates that are delivered by an event stream query.

[20] Like in the count aggregation example (cf. page 135), the query result itself is the accumulator that is updated incrementally; thus, there is no additional `map` step required to extract the relevant information as in the real-time join example (cf. page 136).

5.5 Experimental Evaluation of Quaestor

In this section, we present experimental results to confirm that Quaestor provides low latency and high throughput for push-based and pull-based queries alike. To this end, we first measure Quaestor's real-time query performance to demonstrate that Quaestor inherits InvaliDB's scalability while adding only minimal latency overhead. Second, we contrast Quaestor's pull-based query performance against CDN-cached and traditional uncached database access. Thus, we illustrate the significance of the latency and throughput benefits gained from the query caching scheme that is enabled by InvaliDB.

5.5.1 Push-Based Real-Time Query Performance

In the Quaestor architecture, clients subscribe to real-time queries at Orestes application servers which, in turn, subscribe the corresponding real-time queries at InvaliDB's event layer. An application server thus basically acts as a proxy between the clients and InvaliDB. Since the application servers and InvaliDB are decoupled by the event layer, both can be scaled separately. The application servers on the one side do not become a bottleneck for real-time queries, because different real-time query subscriptions can be managed independently from one another: Additional application servers can always be spawned to take care of further client subscriptions. InvaliDB on the other side can scale with read and write workloads as is evident from the experimental results presented in Section 4.4.

Since an in-depth evaluation of Quaestor's overall scalability is out of scope for this chapter, we restrict the performance evaluation in this section to a quantification of the latency overhead and throughput limitations entailed by processing within the Orestes application server.

Setup & Workload

We conducted our experiments using the same basic setup as described in Section 4.4.1 (InvaliDB-only deployment), but we added an Orestes application server (6 vCPUs, 4 GB RAM) between the benchmark client and InvaliDB's event layer: In the Quaestor deployment, the benchmark client communicated exclusively with the Orestes application server (and not with the event layer directly), like an end user (see Figure 3.1 on page 58). Compared to the experiments from Section 4.4, we thus effectively introduced an additional network hop for all messages sent between the benchmark client and the InvaliDB cluster. For testing Quaestor's scalability by query and write workload, we used the respectively most potent InvaliDB deployment: We configured InvaliDB with 16 query partitions and 1 write partition for the read-heavy workload and used the inverse deployment with only a single query partition and 16 write partitions for the write-heavy workload.

To put the overhead and limitations of the Quaestor architecture into perspective, we contrast Quaestor's latency and throughput characteristics with InvaliDB performance measurements from Section 4.4, using identical workloads: The benchmarking client inserted data items at a fixed rate for one minute, measuring change notification latency for each received change event. Notification latency was again measured as the time before inserting an item until after receiving the corresponding event. We also registered all real-time queries before each measuring phase and limited the number of matching items per experiment run to 1 000 ($\approx$ 17 matches per second) to bound messaging overhead, like in earlier experiments.

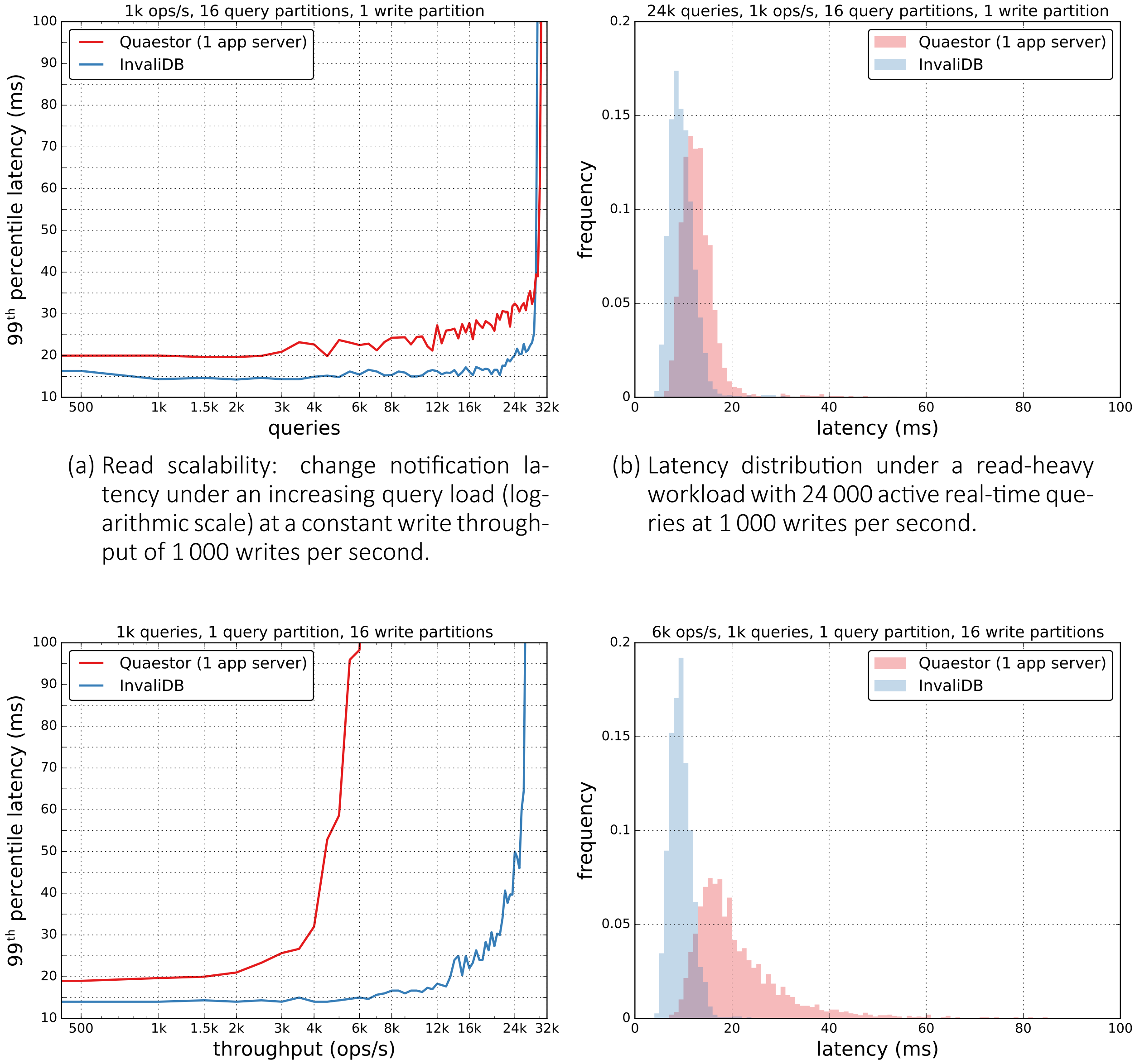
To capture matching performance with as little noise as possible, we used a single WebSocket connection for all subscriptions. The cost of handling client connections is not relevant in the context of this evaluation, because it can be offloaded from the application server to a dedicated component: At Baqend, every application server only maintains a single WebSocket connection to a client-facing proxy server which is only responsible for handling the immediate client connections (and for nothing else). Thus, the number of real-time query subscriptions can be fanned out with only one single WebSocket connection maintained by each application server.

Results

Figure 5.7 shows our results of comparing Quaestor's with InvaliDB's real-time query performance. As is evident from the line plot of 99th percentile latency during the read-heavy workload (a), Quaestor essentially adds a fixed overhead of about 5 ms to InvaliDB's raw change notification latency. At the same time, Quaestor's application server does not represent a bottleneck under the read-heavy workload as it is only limited by InvaliDB's capabilities[21]. The corresponding line plot for the write-heavy workload (c), in contrast, shows that write throughput is limited by Quaestor's application server just before 8 000 operations per second. A comparison of the latency distributions from the read-heavy (b) and the write-heavy (d) workloads at roughly 80 % of capacity paints a similar picture: During the read-heavy workload, Quaestor's latency distribution is shifted to the right by about 5 ms and displays a slightly longer tail, but otherwise corresponds to InvaliDB's latency characteristics. While Quaestor's latency distribution receives a noticeable right skew under write pressure, performance deteriorates gracefully[22] and remains consistently below 100 ms even near full capacity. We did not include measurements of the impact of query subscription, because it depends on the performance of pull-based queries for fetching the initial query result.

[21] Counterintuitively, the Quaestor deployment was able to support slightly more queries than the InvaliDB-only deployment during our experiments as shown in Figure 5.7a. We would like to point out that this is no measurement error, but rather a consequence of pushing InvaliDB to the performance limit: Near system capacity, 99th percentile latency simply becomes unstable and InvaliDB's performance may therefore vary slightly between experiments.

[22] See Appendix C.3 for additional latency histograms taken at other points during the experiments.



(a) Read scalability: change notification latency under an increasing query load (logarithmic scale) at a constant write throughput of 1 000 writes per second.

(b) Latency distribution under a read-heavy workload with 24 000 active real-time queries at 1 000 writes per second.

(c) Write scalability: change notification latency under an increasing write load (logarithmic scale) at a fixed query load of 1 000 active real-time queries.

(d) Latency distribution under a write-heavy workload with 6 000 write operations per second and 1 000 active queries.

Figure 5.7: Change notification latency exhibited by Quaestor in comparison to the corresponding InvaliDB-only deployments under read-heavy and write-heavy workloads.

When receiving a notification from InvaliDB, the application server essentially just passes the notification on, thus incurring fixed costs per received notification. Given the constant match throughput across all experiments, it is therefore plausible that subscribing more real-time queries does not raise computational load for the application server. In contrast, the observation that performance degrades and ultimately collapses under heavy write workload is explained by OLTP-related CPU contention and increased frequency and duration of garbage collection pauses within the Java-based application server.

To sum up, the implementation of our real-time database design exhibits predictable and consistently low latency, even under high OLTP workload. The experimental evidence further confirms our claims of high scalability by showing that real-time query subscriptions do not impose significant overhead on the application server: While serving a real-time query will become more expensive as match throughput grows, the mechanism itself is very lightweight and facilitates many concurrent real-time queries.

5.5.2 The Effect of Query Caching on Pull-Based Query Performance

Quaestor is not only able to provide fast push-based access to data, but it also accelerates common pull-based database queries through query caching. InvaliDB is the enabling component for query caching, because detecting stale query results is infeasible for the application server (cf. Section 5.2). In the following, we provide an experimental quantification of the significant throughput and latency improvements gained through Quaestor's query caching. The experimental data is taken from our dedicated Quaestor publication [GSW$^+$17] which provides additional details on the use case of query caching and an extended evaluation.

Setup & Workload

For a better illustration of Quaestor's benefits, we chose a geographically distributed setup for our experiments: Three Orestes application servers and an InvaliDB cluster with 16 matching cores (8 nodes with 2 vCPUs each) were hosted in Northern Ireland, while the clients executing the workload were located in Northern California. The database workload was executed over MongoDB collections which were hash-sharded by primary key. The MongoDB cluster comprised two shard servers and a single configuration server and was powerful enough to not become a bottleneck during the experiments. The clients, application servers, and all MongoDB servers were Amazon Elastic Compute Cloud (EC2) `m3.xlarge` instances with 4 vCPUs, 15 GB RAM and two 40 GB SSDs each. For InvaliDB, we used `c3.large` instances with 2 vCPUs and 3.75 GB RAM each.

For each experiment, we generated 10 MongoDB collections with 10 000 documents each and 100 distinct queries per collection. To generate the workload, all clients executed asynchronous requests that were divided into 1 % write operations (inserts, updates, and deletes), 49.5 % primary key lookups, and 49.5 % database queries. The queries were designed in such a way that each one had exactly 10 matching records at the beginning of an experiment. The records, queries, and collections to use for each client request were sampled from a Zipfian distribution. The client connections were warmed up for 30 seconds before each experiment by writing data into a separate collection. The Quaestor configurations that used caching had the clients refresh their Cache Sketches once every second to bound staleness.

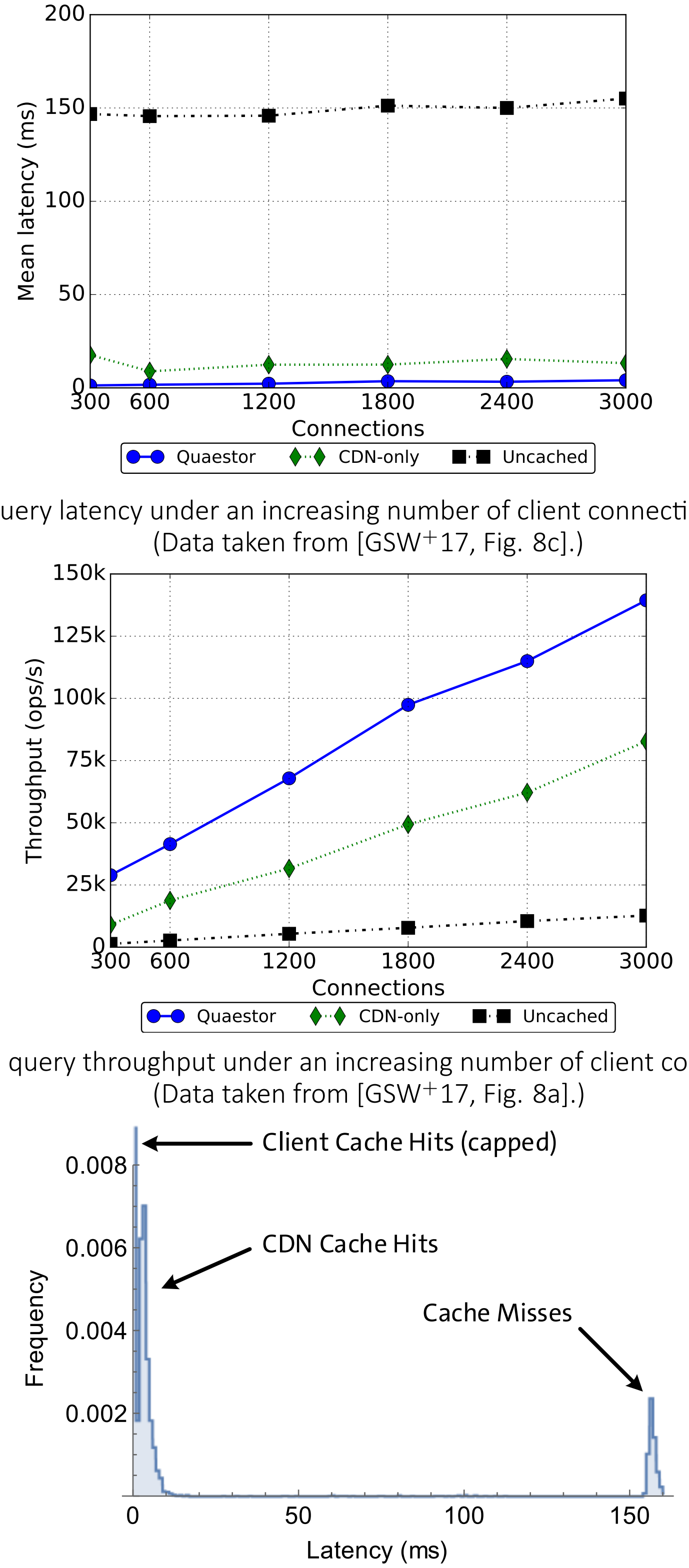
Results

Figure 5.8 compares read performance achieved under an increasing number of client connections by different Quaestor configurations:

- **uncached** Quaestor (black line) using no caches whatsoever
- **CDN-only** Quaestor (green line) using only the CDN for caching
- full-fledged **Quaestor** (blue line) using both CDN and client cache (browser cache)

Throughout the experimental evaluation of latency (a), the baseline setup with uncached database access exhibited a mean latency of about 150 ms, whereas the CDN-only and the full-fledged Quaestor setup consistently achieved less than 20 ms ($> 7\times$) and less than 5 ms ($> 30\times$), respectively. A similar observation can be made with regards to throughput (b): The setup without any caching was consistently outperformed by the CDN-only Quaestor ($> 5\times$) and Quaestor with full caching ($> 11\times$). The query latency histogram (c) shows the distribution of response times for the full-fledged Quaestor setup that used all available caches: Most requests were answered by the client cache ($<$ 1 ms latency) or the CDN ($<$ 5 ms latency), while only few requests hit the database ($\approx$ 150 ms latency).

In summary, our results demonstrate that Quaestor is able to achieve significant latency and throughput improvements compared to uncached database access and even compared to a setup that exploits CDN caches for delivery. Although not shown here, it should be noted that the corresponding non-Quaestor Orestes deployment (without InvaliDB) would perform significantly worse, because it does not support query caching: All queries (i.e. 49.5 % of all requests) would therefore have to be processed by the database system.



(a) Mean query latency under an increasing number of client connections. (Data taken from [GSW+17, Fig. 8c].)

(b) Achievable query throughput under an increasing number of client connections. (Data taken from [GSW+17, Fig. 8a].)

(c) Distribution of query latency exhibited by full-fledged Quaestor with client and CDN caching. (Diagram taken from [GSW+17, Fig. 8f].)

Figure 5.8: Query latency and throughput exhibited by full-fledged Quaestor with client and CDN caching, Quaestor with only CDN caching, and completely uncached database access.

5.6 Summary & Discussion

In this chapter, we demonstrated the practicality of our real-time database design InvaliDB by integrating it into the Orestes database middleware.

Orestes' characteristic caching approach for dynamic data relies on the ability to detect when data becomes stale. While the Orestes application server can detect when individual records are updated by CRUD operations, we identified change detection over query results as infeasible for the base system. In consequence, Orestes is only able to cache individual database records, but not query results. To address this issue, we presented the Quaestor architecture as an extension to Orestes that employs InvaliDB for query result monitoring. Within the Quaestor architecture, InvaliDB is used in two different ways: First, InvaliDB makes **query result caching** feasible by providing low-latency invalidation messages for stale query results. As demonstrated in our experimental evaluation, the query caching scheme enabled by InvaliDB can improve throughput and latency of pull-based queries by more than an order of magnitude. As a second use case, InvaliDB generates change deltas for query results which are delivered to end users through push-based **real-time queries**. In contrast to competing real-time database architectures that burden the application server with query matching (cf. Section 2.3), Quaestor delegates result maintenance to InvaliDB and thus removes the application server as a possible bottleneck. As verified by our experiments, Quaestor thus provides real-time queries in a scalable fashion with little overhead for the application server, supporting both high write throughput and many concurrent real-time queries with low double-digit latencies.

Aside from performance considerations, the design of the client APIs for push-based data access received particular attention within this chapter. We created two different query interfaces that are geared towards simplicity and expressiveness, respectively. As illustrated with Twoogle and other examples, **self-maintaining queries** are easy to reason about as they effectively behave like common pull-based queries that are refreshed on every result change. Thus, they provide a smooth transition from pull-based to push-based query semantics for application developers. Instead of the complete list of matching records, **event stream queries** deliver incremental *change deltas* that capture how query results evolve over time. While event stream queries do expose increased complexity to the application developer, they also provide more fine-grained control over information flow, for example through filtering by match type or the type of the triggering operation.

To conclude, we have demonstrated that InvaliDB can enhance an existing database architecture in two ways: First, InvaliDB facilitates significant performance improvements for existing pull-based interfaces and, second, it promotes a way to access data in push-based fashion. In contrast to other real-time database designs, InvaliDB exposes greater query expressiveness, is compatible with legacy database systems, and has further been demonstrated to scale with read and write workloads alike.

6 Conclusion

"Just one more thing..."

—Lieutenant Columbo

The goal of this thesis has been to devise a scalable design for expressive push-based real-time queries on top of existing pull-based databases. After we divided this goal into four distinct challenges in Chapter 1, we surveyed and classified related work in Chapter 2. We then set out to create the system design InvaliDB within Chapter 3 and evaluated a prototypical implementation thereof in Chapter 4. In Chapter 5, we demonstrated the practicality of our approach by integrating the InvaliDB prototype with an existing pull-based database middleware. We further presented database query caching and client-facing real-time queries as two use cases that are supported by the integrated system, but would be infeasible without InvaliDB.

In this chapter, we conclude this dissertation by reflecting on what we have achieved and projecting what remains to be done. We start with a summary of our main contributions and our most significant learnings in Section 6.1. We then discuss limitations of our work and derive possible lines of future research in Section 6.2. Finally, we provide closing thoughts in Section 6.3.

6.1 Best Practices for Designing a Scalable Real-Time Database

Within this thesis, we have conceived and implemented the system design InvaliDB for collection-based real-time queries on top of existing database systems. In doing so, we demonstrated that real-time queries can be provided in a scalable (cf. Challenge C_1) and expressive (cf. Challenge C_2) manner, while still being compatible with legacy databases (cf. Challenge C_3). Our integration with an existing database architecture further illustrated how an existing pull-based query interface can be turned into a push-based interface with minimal complexity overhead for application developers (cf. Challenge C_4).

Over the course of our work, we identified critical design flaws in existing real-time databases and derived a number of best practices to avoid them. To validate our research in a real-world setting, we deployed our implementation in production at the Backend-as-a-Service company Baqend (cf. Chapter 5) where it has been serving customers since July 2017. In this section, we recapitulate important lessons we learned and outline essential features of a real-time database system as well as ways to achieve them.

6.1.1 Scalable Workload Distribution

As illustrated in Section 1.1, a real-time database has to analyze the effect of every single write operation on every single real-time query. In consequence, both an increase in write throughput and a growing number of query subscriptions lead to a proportional increase in matching workload. Since a system design that only scales with one of these two dimensions will necessarily become a bottleneck when single-machine capacity is reached (see Section 2.3), real-time database architects must plan ahead for either dimension to grow. In Section 3.3, we presented a *two-dimensional partitioning* scheme that scales out with write operations as well as queries across several worker processes. However, cross-machine distribution is difficult to realize for operations beyond the mere checking of static filter conditions. Therefore, some components of a query may still have to be evaluated in a centralized fashion. In our architecture, the computation related to result order, joins, and aggregations is done in processing stages that are mostly partitioned by query; these do not become bottlenecks in practice, though, because the serial processing stages are downstream of the highly distributed filtering stage and therefore only receive a fraction of overall system throughput.

6.1.2 Isolated Failure Domains

Unless the process of query matching is effectively *decoupled* from the main application server, an outage of the real-time component can take down the entire system. In order to make a real-time database deployment justifiable in a production setting, overall system availability must therefore not depend on the health of the real-time subsystem. This requirement is related to the requirement of scalability, but ultimately differs in its focus on failure scenarios: While scalability of the real-time component postulates that heavy load is sustainable in principle, the isolation of failure domains ultimately demands basic availability even under partial system failure. The above-mentioned scheme for workload partitioning does not exclude the possibility of real-time subsystem failure, e.g. through a sudden load burst, a hardware malfunction, or another error. Therefore, InvaliDB separates the application server and the real-time subsystem through *asynchronous communication* in the event layer detailed in Section 3.2; thus, the worst possible impact of query matching breakdown essentially amounts to delayed real-time notifications or cancellation of active subscriptions (cf. Section 3.1.1).

6.1.3 Polyglot Data Model

Even though InvaliDB is based on several assumptions regarding the context into which it is embedded (see Section 3.1), we abstracted from implementation details where possible. In more detail, we created a *generic message exchange protocol* based on JSON as the external interface, so that InvaliDB clients can be written in any programming language (see Section 5.2.3). Further, we embedded the query engine into the overall architecture in such a way that it can be switched out with relative ease (cf. Section 3.3.3). Since our architecture is thus agnostic of the underlying data model, InvaliDB naturally supports polyglot backends like Orestes. The *pluggable query engine* also enabled us to quickly move from our initial JavaScript-based query engine to the Java-based implementation that is used in production at Baqend (cf. Section 4.3). Looking ahead, InvaliDB's *data model independence* will be crucial for adding support for new database languages.

6.1.4 Balanced Interfaces

When designing Quaestor's real-time query interface, we found one of the major challenges in striking a balance between expressiveness and complexity. *Event stream queries* (cf. Section 5.3.3) are very powerful as they combine database query predicates with event-based selection criteria such as match or operation type. However, this mix of collection-based and stream-based semantics is non-intuitive to work with, because it requires an understanding of InvaliDB's semantics as well as the event message format. *Self-maintaining queries* (cf. Section 5.3.1) produce complete query results instead of change deltas and are therefore significantly easier to use, given the concept of a pull-based query over database collections is already understood. There are two key aspects to their simplicity: First, self-maintaining queries only expose the collection-based component of a real-time query, i.e. they do not reference the match type or other event properties. Second, self-maintaining queries extend common ad hoc queries in the sense that they add functionality without imposing restrictions. In order to utilize a self-maintaining query, users thus only have to understand that they will receive updated results in addition to the initial result they would expect for the corresponding pull-based query. Since the public release of our real-time query API in mid-2017, we have seen both query interfaces in production at Baqend customers. We consider the acceptance by actual users to be a validation of our approach of exposing two distinct query interfaces with varying degrees of complexity.

6.2 Open Challenges

We firmly believe that our work provides valuable insights for real-time database architects, but we are also aware of some limitations. In this section, we outline challenges we encountered and establish points of reference for future work.

6.2.1 Extending Semantics

Our work is focused on sorted filter queries over single collections as they are supported by many document-oriented NoSQL databases. With respect to query expressiveness, consistency guarantees, and also regarding the diversity of supported database systems, we therefore see multiple opportunities to extend our approach.

Joins & Aggregations. By concept, InvaliDB provides support for groupings, aggregations, and queries that connect multiple collections through joins (cf. Section 3.3.2). However, we only implemented sorted real-time filter queries over single collections in the context of this work. Therefore, real-time aggregations and joins have to be maintained within the client application (see Section 5.4.2 and Section 5.4.3 for examples). Future work could extend our InvaliDB implementation by real-time aggregations and joins to remove the necessity for these kinds of client-side workarounds. Full support for real-time joins and aggregations would further enable quantifying the trade-off between incremental and recomputation-based maintenance for these types of real-time queries.

Additional Database Languages. As described in Section 4.3, we implemented real-time queries using MongoDB's query language, concentrating on basic filtering functionality (e.g. comparison and regex expressions). In follow-up work, the sophistication of query processing could be advanced in two different directions: First, the existing MongoDB query engine could be amended by additional operators, e.g. `$expr` for evaluating conditions over fields in the same record [Mon18c]. Second, support for additional database languages and query types[1] could be added through entirely new query engines. Depending on the concrete query engine, the complexity of this endeavor can range from being trivial to being prohibitive. For example, systems lend themselves to an InvaliDB integration when they already use a pluggable query engine (e.g. Calcite [Apa18d]) or when their source code is available (e.g. MongoDB). In these cases, the query engine can either be

[1] For example, we do not address recursive queries or queries with subqueries in this work.

used as-is or it is at least feasible to rebuild in distributed fashion, because matching behavior is transparent. Many commercial database systems, however, are effectively black boxes for which only the compiled binaries are available. Reengineering a query engine for such a system is extremely cumbersome and requires meticulous analysis of intended and actual matching behavior to assure correctness of real-time query results: Unless the distributed query engine (running within InvaliDB) and the original query engine (running within the database) are perfectly aligned, real-time query results and actual results may diverge. Future work could therefore also develop systematic approaches for defining test cases and spotting behavioral differences between query engines. Support for similar database languages or different dialects of the same language (e.g. SQL) could thus be added more efficiently.

Transactions. InvaliDB promotes a notion of eventual consistency that resembles the guarantees provided by many document-oriented NoSQL databases: In the absence of write operations, a real-time query result converges to the result that the corresponding pull-based query would yield when executed against the underlying database. In the context of query engines that support multi-key transactions, it should be considered whether and in what ways transactional guarantees can be reflected by InvaliDB's push-based real-time queries. To achieve transactional visibility of write operations, for example, events that were triggered by the same transaction could be delayed until all of them can be published to the client atomically. Likewise, serviceability of monotonic reads, read-your-writes, or other session consistency guarantees could be investigated.

Embracing Streams & Complex Events. In this thesis, we argue that collection-based semantics are natural for many applications and thus sometimes preferable over stream-based semantics. However, stream-based semantics undoubtedly have their merits and we do not see any reason why a data management system should be restricted to either streams *or* collections as basic primitives. While InvaliDB's default semantics are based on database collections, its pluggable query engine supports collection-based and stream-based semantics alike (cf. Section 3.3.3). In a purely stream-based context, InvaliDB's query engine would not have to be aligned with query evaluation in a backing database, so that arbitrary semantics could be adopted and mechanisms for approximation and load shedding (cf. Section 2.4.3) would be eligible without compromising correctness. Future work could thus develop a stream-based query engine that is tailored to *time series* or other forms of sequential data. InvaliDB further only supports simple event processing based on match and operation types. Another powerful extension to our approach would therefore be the integration of more sophisticated *complex event processing* to establish temporal, causal, or other relationships between events.

6.2.2 Exploring Trade-Offs & Optimizations

Through our implementation, we have shown that InvaliDB is scalable and capable of providing low latency even under heavy load. In the following, we present considerations regarding both InvaliDB and Quaestor to improve performance and efficiency, availability, and usability.

Failure Transparency. In our current implementation, several types of failure result in query cancellation and an error message being sent to the client. This fail-fast strategy does not impede behavioral correctness, but it causes inconvenience for end users and is sometimes relatively inefficient compared to other viable options. To avoid service disruption and save client resources, the application server could be redesigned to handle those errors transparently which are currently resolved by client-initiated resubscription. Specifically, the history of any subscription could be retained to enable compensation of *connection drops* between client and application server: On reconnection, a client would provide an identifier for every supposedly active subscription (and possibly a history offset), so that the application server could then only deliver the events that were missed since connection loss. To make this possible, parts of the event layer would have to be reimplemented[2] on top of a message queue that supports data retention and replay (e.g. Kafka); our Redis-based implementation does not persistently store any data apart from the currently active queries. Using a system like Kafka underneath, the event layer would also become natively reliable: Since Redis does not provide any reliability guarantees, loss-free communication can only be ensured through application-side measures like query invalidation on every connection loss (cf. Section 4.2). Replayability would also allow a bootstrapping procedure for InvaliDB's worker nodes that picks up and continues active subscriptions instead of canceling them: Since the processing engine Storm does not support fault-tolerant state management, a worker node failure currently results in deactivation of the worker's active subscriptions when it comes back up (see Section 4.1). There are also numerous mostly performance-related aspects of our implementation that could be improved. For example, highly frequent write operations on the same entity (and resulting change events) could be merged, so that resources are not wasted and clients are not overwhelmed by incoming messages (cf. `collapse` parameter in Section 5.3.2).

Self-Maintainability. InvaliDB makes query maintenance feasible by taking the matching process out of the database. However, since the database remains the single source of truth, queries have to be registered with enough information to make the result self-maintainable. As discussed for sorted queries in Section 3.3.2, however, only partial self-maintainability is feasible for some queries; when InvaliDB lacks required information, the database has to be contacted or else the query must be deactivated. To enforce strict isolation between InvaliDB and the database, we opted to cancel the query subscription

[2] In the Quaestor architecture, the application servers cannot be used for retaining the subscription history, because they are stateless by design in order to facilitate high scalability.

and have the client resubscribe whenever the result might have become stale. This procedure can be optimized by avoiding the detour over the client and letting InvaliDB directly interact with the database. As a possibly more significant optimization, InvaliDB could further be allowed to retrieve missing information in finer granularity than the entire result, e.g. by requesting individual records to fill up an ordered query's slack (cf. page 80) or by resolving references in an equi-join[3]. This would reduce the rate of maintenance errors for sorted queries and enable incremental maintenance for real-time queries that are purely recomputation-based in our current scheme such as certain join queries (cf. Section 3.3.2). Subsequent work could shed light on the opportunities and risks that lie in weakening the separation between InvaliDB and the database to improve self-maintainability of active real-time queries.

Indexing & Workload Partitioning. As described in Section 4.3.2, our current implementation of InvaliDB employs a brute force comparison to match all queries against all incoming after-images. The computational complexity of continuously matching n query predicates at a given write throughput m is therefore $O(m \cdot n)$. By employing *indices* for comparisons and other predicates (e.g. anchored regex expressions), the computational complexity of the matching process could be reduced to $O(m \cdot \log\ n)$ in some cases, thus increasing single-node efficiency. Further, our current query partitioning scheme distributes entire queries by their hashes. In consequence, query predicates are evaluated on multiple nodes in the InvaliDB cluster, whenever they are part of queries in different query partitions. A *containment-aware distribution* scheme could increase efficiency by assigning queries to the same query partition whenever one of them subsumes the other. Since deviating from our current hash-based distribution scheme could also lead to an uneven load distribution, though, a key challenge of this approach would be to preserve scalability in the presence of arbitrary query workloads.

Client Performance. In the web-centered use cases that motivate our work, real-time query subscribers are often mobile phones or other devices with limited processing power, weak network links, and/or bounded (monthly) data allowance. Heavy hitter queries are therefore likely to saturate these end user devices (or the network between them and the application servers) before the InvaliDB cluster becomes a bottleneck. Even though we focussed on the scalability issues in the backend in this thesis, we also discussed three different approaches to cope with real-time query load that is prohibitive for clients. First, write operations on the same entity can be collapsed in order to mitigate hotspots and thereby not only reduce the matching load on the InvaliDB cluster, but also the frequency of emitted result updates (cf. write propagation in Section 3.1.3). Second, the frequency by which result updates are emitted can similarly be reduced by buffering and collapsing

[3] The following query illustrates that reference resolution can be more efficient than query recomputation: `SELECT * FROM score s INNER JOIN user u ON s.uid = u.id ORDER BY s.value DESC LIMIT 10`. Whenever a new user enters the top-10 high score list (i.e. the query result), our current scheme would require resubscribing (and thereby reevaluating) the query, because the new user's reference cannot be resolved. Granting InvaliDB access to the database and allowing exploratory queries would allow resolving the user reference with a simple lookup and specifically without invalidating the query's result.

change notifications themselves (cf. subscription parameters in Section 5.3.2). Third, the fallback of our maintenance procedure to periodic pull-based query execution (cf. result recomputation in Section 3.3.2) also represents an effective mechanism to control the data flow towards the client as it limits the frequency of emitted change notifications to the polling period[4]. Future work could quantify and compare the effectiveness of these different approaches for saving client resources, derive best practices, and possibly develop more sophisticated schemes to minimize client resource usage for real-time query subscriptions.

Production Deployment. The system we built in the context of this thesis does not only serve as a proof of concept, but has already been used in production at Baqend since July 2017. By exposing our system to paying customers, we created a strong incentive to optimize for metrics that would not have received much attention in a purely academic context. In contrast to an experimental setup, a production deployment is running continuously over weeks without interruption and therefore must tolerate or autonomously recover from network outages, random server restarts, fluctuating workloads, and other complications. Also, maintenance tasks such as releasing new versions on a regular basis become a necessity. Rolling out a new version of the application server or database has to be coupled with the rollout of the corresponding InvaliDB version. Consequently, safety measures have to be taken in order to enable rolling back any unsuccessful deployments. To further minimize the total cost of ownership while retaining sufficient resources for sudden load spikes, it is necessary to monitor the entire application stack on various levels and scale subsystems out or in, depending on their load. Many of these tasks are currently manual and we consider *automating cluster management* of InvaliDB and related components a huge challenge for ourselves or practitioners who are following our example.

6.2.3 Building Applications

We believe the emergence of *scalable real-time databases* enables countless ways to improve existing applications and create genuinely new ones. As a final note in this outlook, we would like to outline potential for innovation in different application areas.

Reactive & Collaborative Mobile Apps. By providing a push-based access mode for queries with traditional database semantics, InvaliDB simplifies the development of reactive social media websites, collaborative worksheets, and other applications that promote interaction between users. While we touched on several use cases in this work, we focused on a display of API functionality rather than actual applications (see Section 5.3 and Section 5.4). Qualitative studies to evaluate the usefulness of our real-time query abstractions and comparisons with peer systems would provide valuable feedback for us to improve

[4] Since this approach relies on the underlying database (and not the InvaliDB cluster) for query evaluation, it further enables real-time queries that are only supported by the underlying database, but not by the real-time query engine.

existing and add missing features. Apart from efforts to extend the existing JavaScript API, client libraries for other programming languages and especially native SDKs for mobile platforms (e.g. Android and iOS) would further extend the reach and impact of our approach towards push-oriented data access.

Upgrading Legacy Interfaces. Applications that have been developed on top of pull-based databases often work in a pull-based fashion themselves, i.e. the user has to take action to refresh or update her view of the data. However, rather than a deliberate design decision, the need to refresh manually is mostly a mere side effect of using a pull-based database for data storage. Since InvaliDB makes real-time features available on top of existing databases, we see a tremendous opportunity in upgrading existing *static interfaces* with self-maintaining queries as presented in Section 5.3.1: Those portions of the user interface that display dynamic information (e.g. user-defined search results as in Twoogle) could thus be made interactive to augment the user experience. Interfaces could also be served in two different versions. For instance, the static version of a website could be served to low-bandwidth clients or the interactive version could be served to premium customers only.

Augmenting Current Cache Coherence Schemes. Even though one of the defining properties of relational database systems is strong consistency, many approaches to scale out read capacity reflect a trade-off between consistency and query performance: Due to the significant costs associated with eager incremental *query maintenance*, in particular, result caches are commonly lagging behind, because they are refreshed asynchronously through batch updates or periodic recomputation. Similar to its role within the Quaestor architecture, InvaliDB could perform asynchronous *change detection* to trigger updates or recomputation of invalidated copies of the data. Having the database servers only subscribe to InvaliDB instead of monitoring the write stream themselves promises several benefits: First, this would significantly increase fault tolerance of the overall system, because OLTP workloads would be decoupled from the performance impact of query maintenance. As a second advantage, view maintenance could further be used in a much wider scope than before, because InvaliDB (unlike a monolithic database server) is able to scale with the number of concurrently maintained queries and with write throughput at the same time. As a third benefit, InvaliDB removes the necessity to weigh consistency against data freshness (e.g. through refresh rates), because it performs incremental and immediate result maintenance with low latency by default. Seeing that current schemes for *query offloading* deliberately expose stale data to clients in favor of increased performance, we believe InvaliDB bears the prospect of combining improved query performance with reduced data staleness, even though it currently does not provide transactional guarantees.

6.3 Closing Thoughts

Collection-based query semantics have been bound to pull-based access patterns for decades. But even though the first active database features emerged as early as the 1970s, modern databases still do not provide scalable and expressive push-based query mechanisms. Stream-oriented systems do address this lack of reactivity through continuous queries, but they also require developers to abandon the concept of a database collection and to reason about data streams instead. As a compromise between these two system classes, real-time databases attempt the conceptual unification of database query semantics with a push-based access model. However, current implementations are only of limited practical relevance, since they are incompatible with existing technology stacks, fail under heavy load, or do not support complex queries to begin with.

In this work, we proposed the real-time database design InvaliDB to overcome these limitations. While our prototype is built on top of MongoDB, the concept itself is database-agnostic and can be applied to other NoSQL and even relational database systems. In an extensive experimental evaluation, we presented compelling evidence for its high scalability and consistently low latency under various workloads and system configurations. We integrated our prototype with a purely pull-based database middleware in the Quaestor architecture to further illustrate the practicality of our approach with an end-to-end example. In doing so, we also exemplified two convenient interfaces for real-time queries: Whenever a subscribed query's result is altered, the integrated system proactively delivers either complete results through self-maintaining queries or incremental change deltas through event stream queries. For a display of potential gains beyond reactive behavior, we finally showed how InvaliDB makes consistent query caching possible by providing low-latency result invalidations. In the Quaestor architecture, InvaliDB thus enables an order-of-magnitude improvement of throughput and latency for pull-based queries over the base system without InvaliDB.

In the past, practitioners have been rightfully cautious in adopting real-time databases, because they are hard to integrate into existing applications and mostly intractable to use at scale. The system design presented in this thesis addresses frequent concerns through a combination of characteristics that, to the best of our knowledge, is unique among real-time databases: First, it abstracts from the underlying data model and is therefore applicable to virtually any database system. Second, InvaliDB scales with both the number of concurrent queries and with write throughput. Third and arguably most important for production settings, it is designed as an opt-in component with an isolated failure domain to make its adoption a low-risk endeavor. In conclusion to this dissertation, we hope that our work sparks new confidence in the practicality of real-time databases and inspires further research on the topic within the database community.

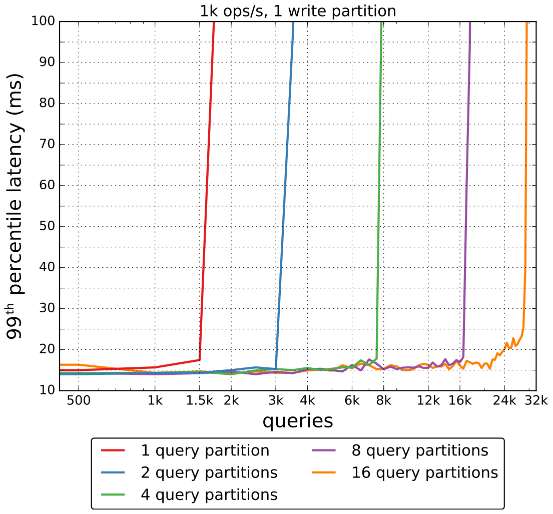
Appendix

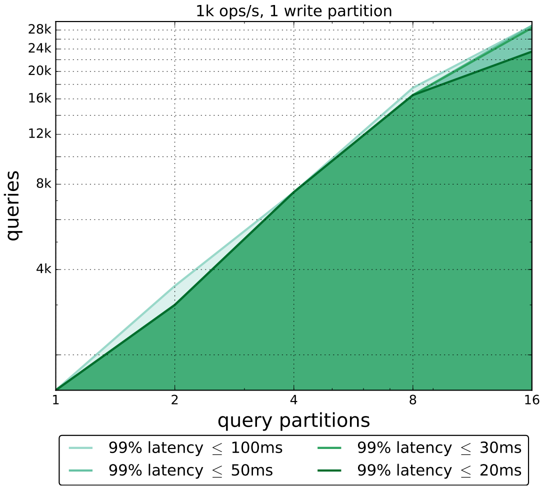
A InvaliDB Performance: Read Scalability

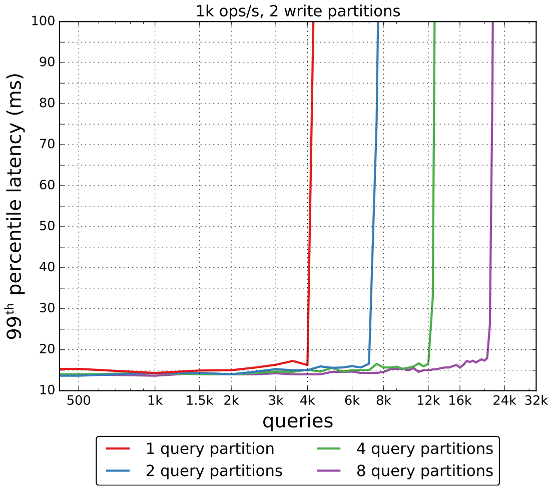
Please note that the following performance plots are scaled logarithmically.

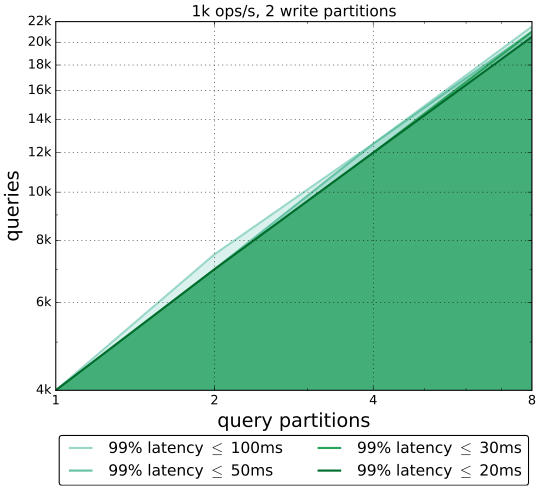
A.1 Sustainable Queries Under Varying Query Partitions

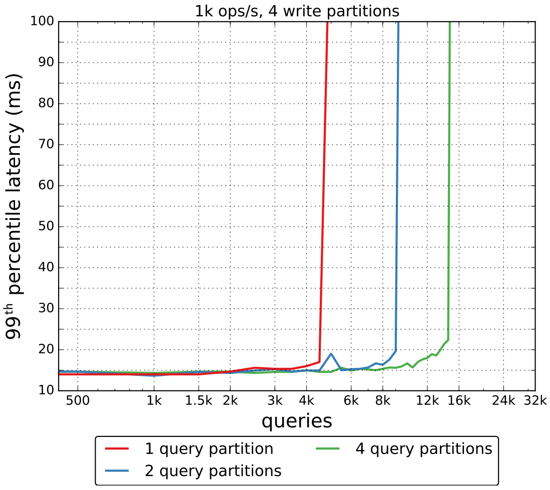
1 000 ops/s Fixed, Increasing Query Load

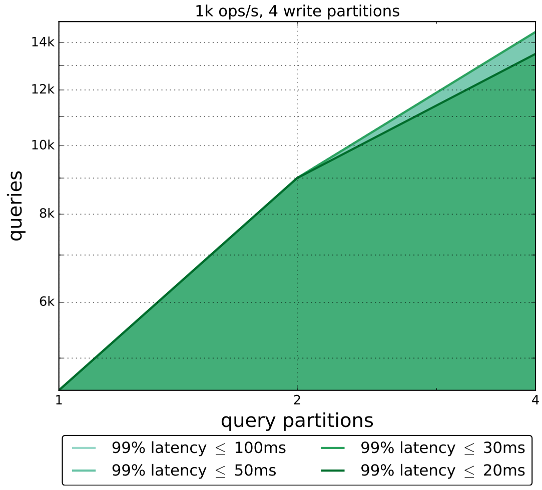


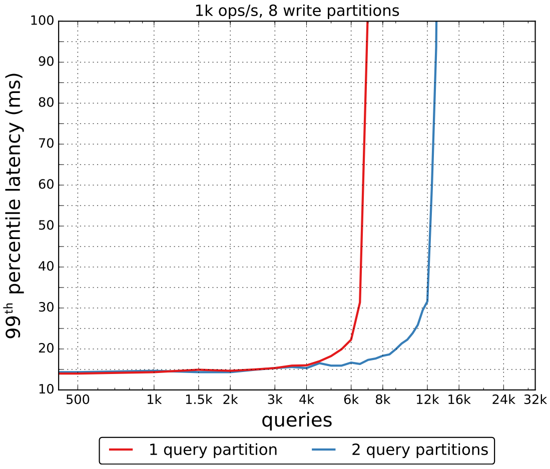


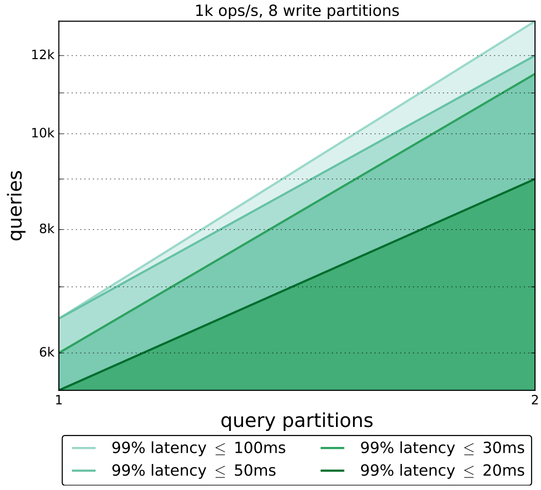


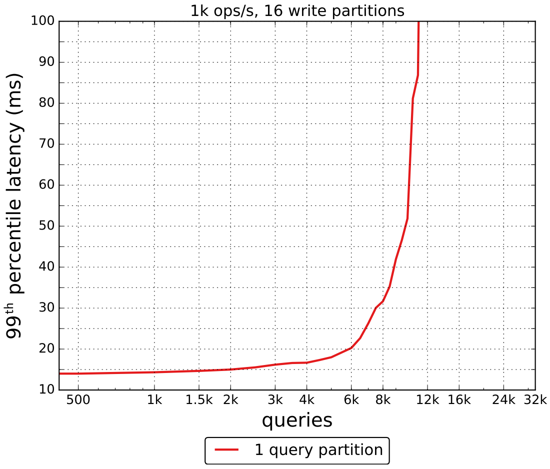



1k ops/s, 4 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions
4 query partitions

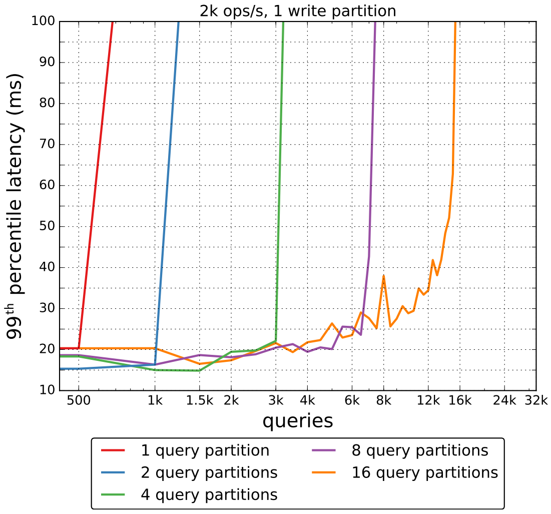

1k ops/s, 4 write partitions
queries
14k
12k
10k
8k
6k
1
2
4
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

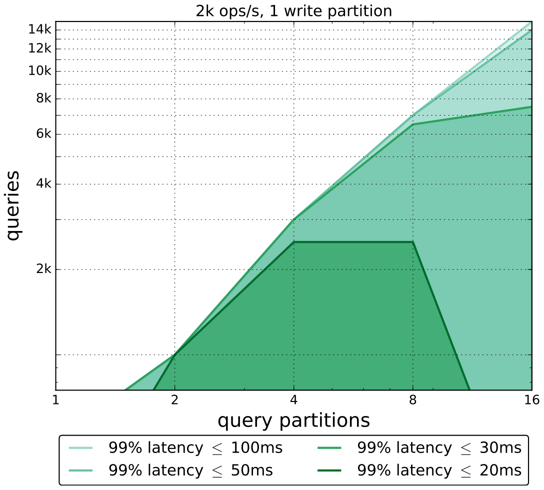

1k ops/s, 8 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions

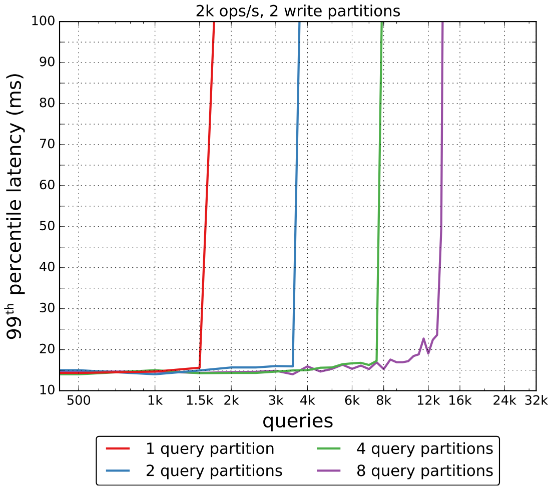

1k ops/s, 8 write partitions
queries
12k
10k
8k
6k
1
2
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

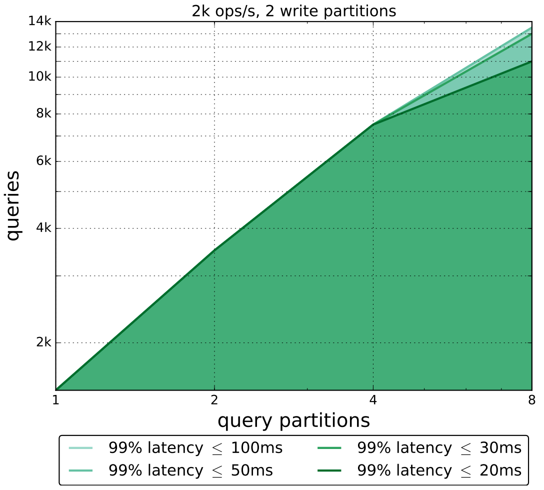

1k ops/s, 16 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition

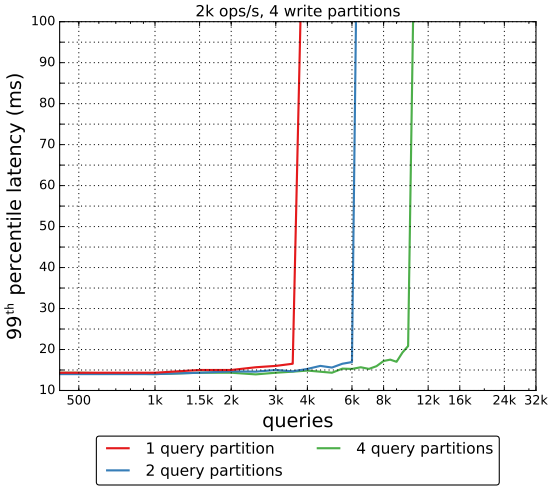
2 000 ops/s Fixed, Increasing Query Load

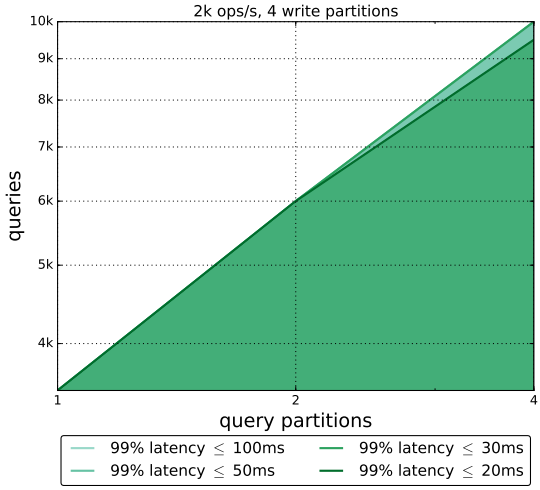


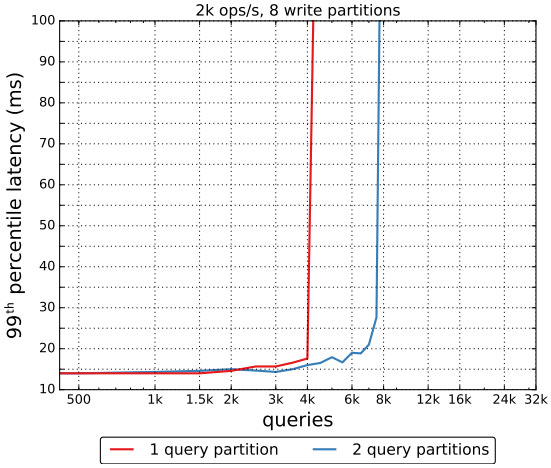


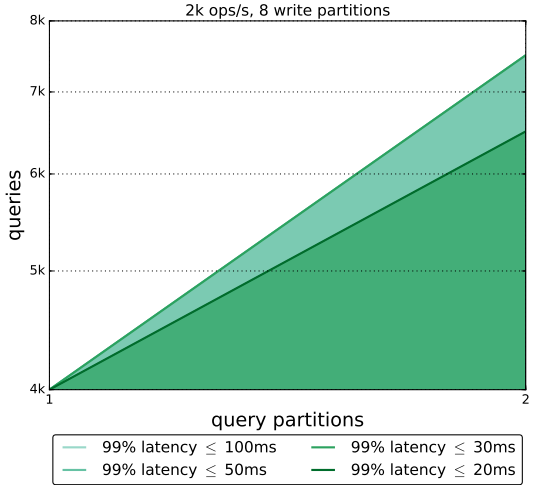


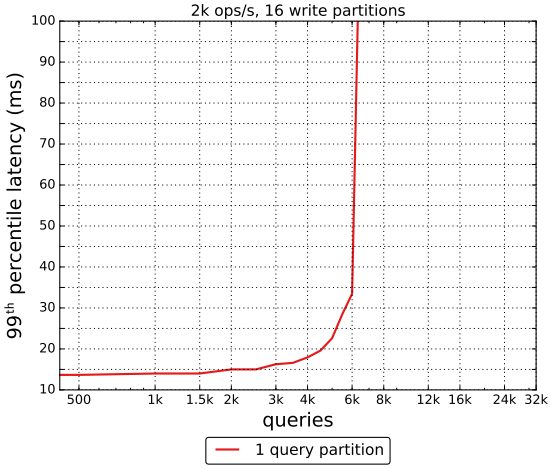



2k ops/s, 4 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions
4 query partitions

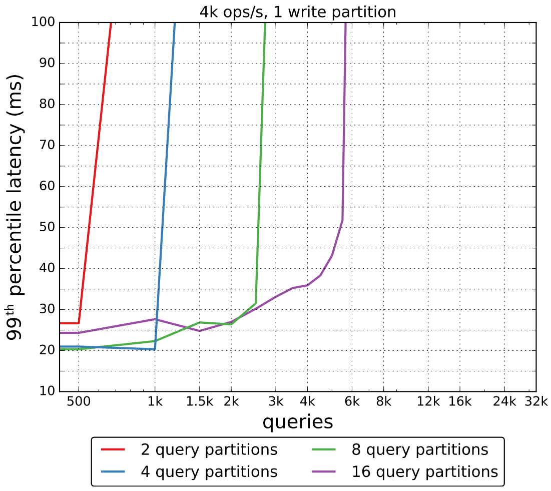

2k ops/s, 4 write partitions
queries
10k
9k
8k
7k
6k
5k
4k
1
2
4
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

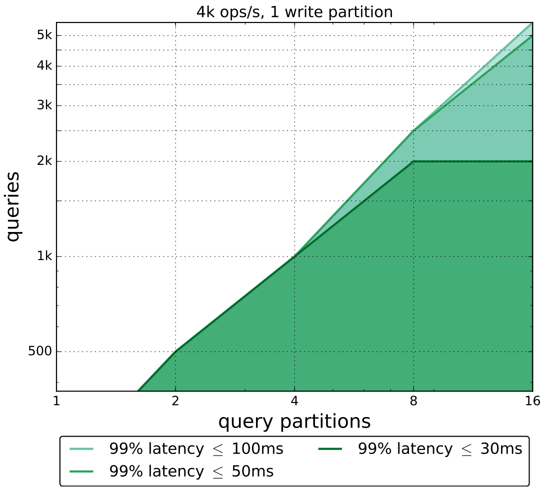

2k ops/s, 8 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions

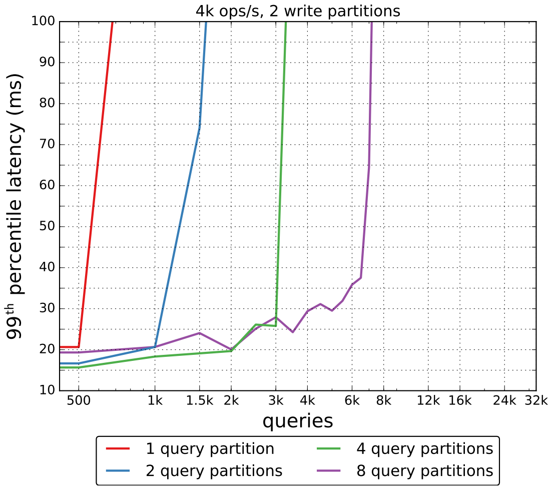

2k ops/s, 8 write partitions
queries
8k
7k
6k
5k
4k
1
2
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

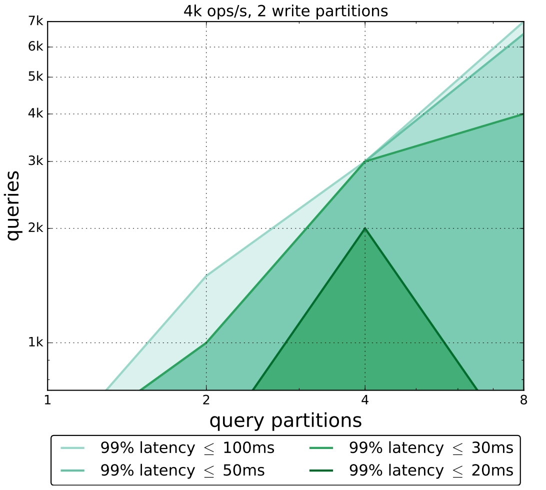

2k ops/s, 16 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition

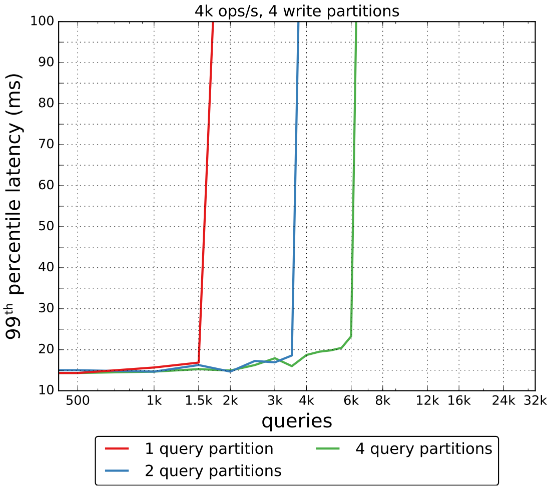
4 000 ops/s Fixed, Increasing Query Load

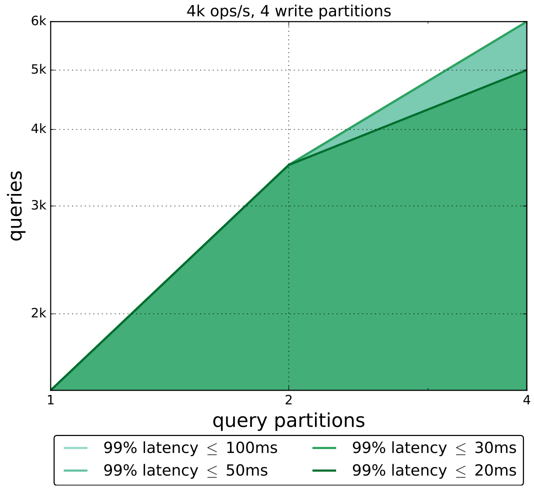


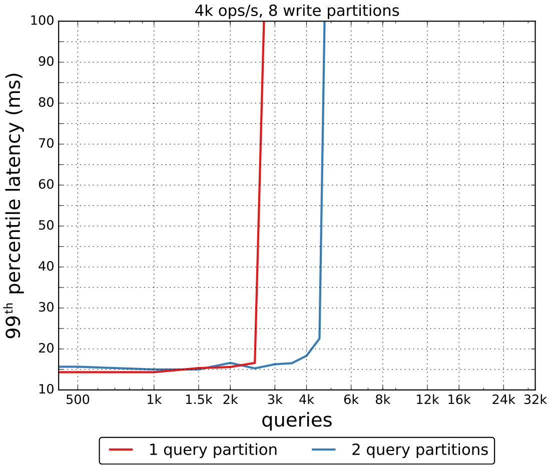


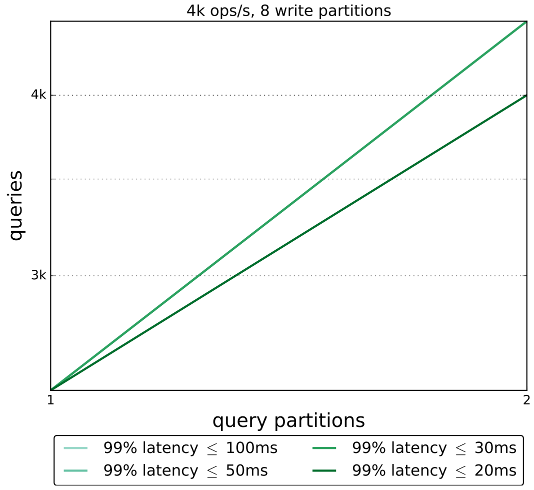


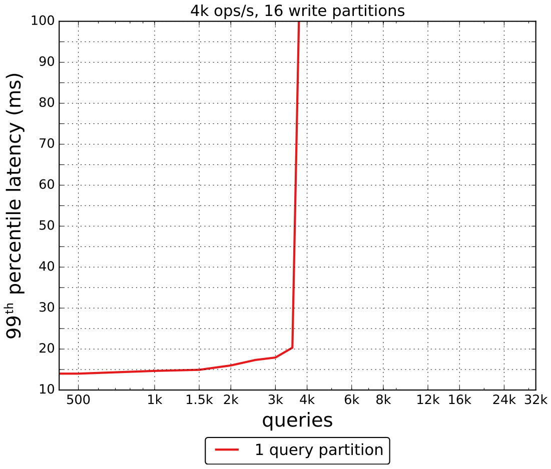



4k ops/s, 4 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions
4 query partitions

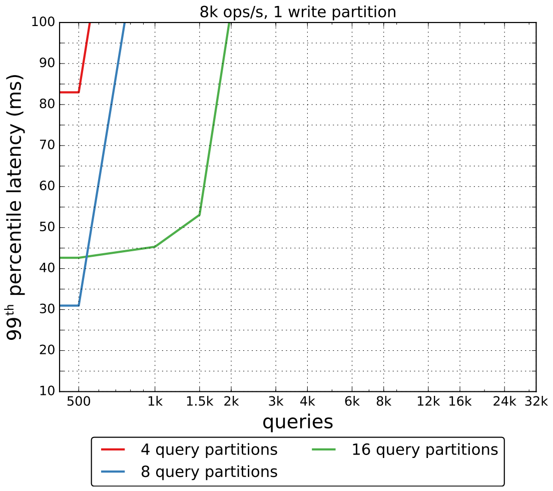

4k ops/s, 4 write partitions
queries
6k
5k
4k
3k
2k
1
2
4
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

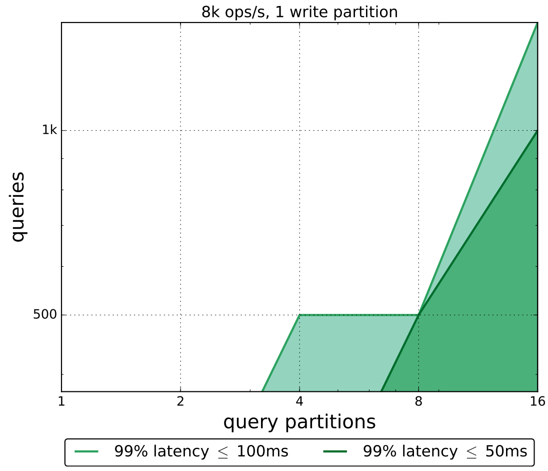

4k ops/s, 8 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions

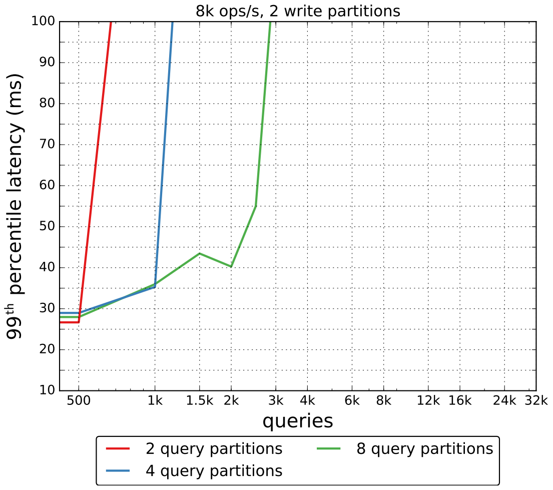

4k ops/s, 8 write partitions
queries
4k
3k
1
2
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

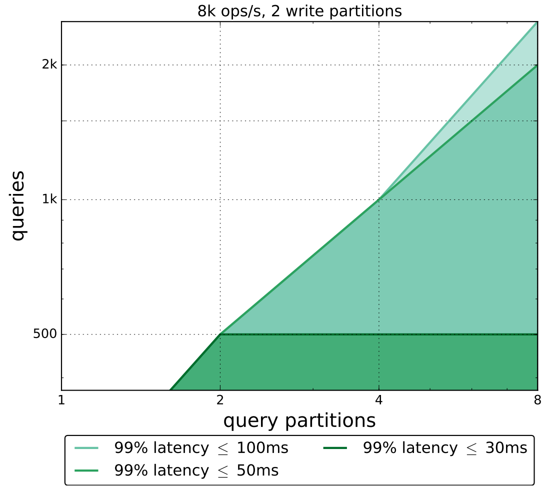

4k ops/s, 16 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition

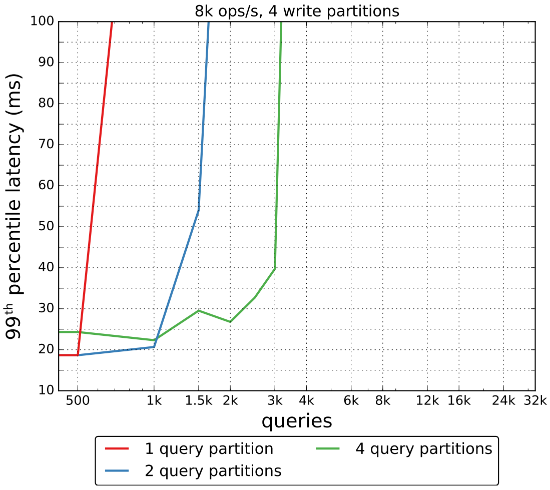
8 000 ops/s Fixed, Increasing Query Load

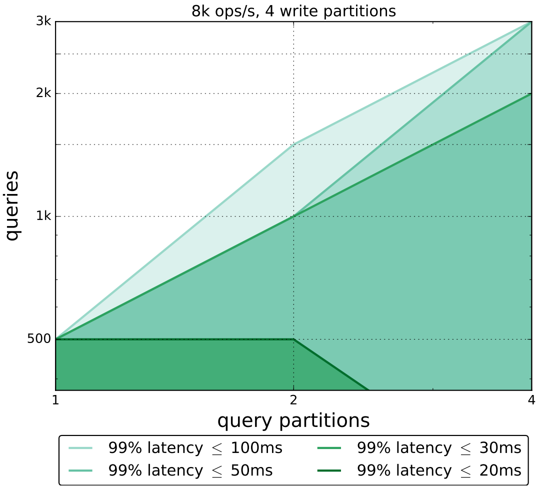


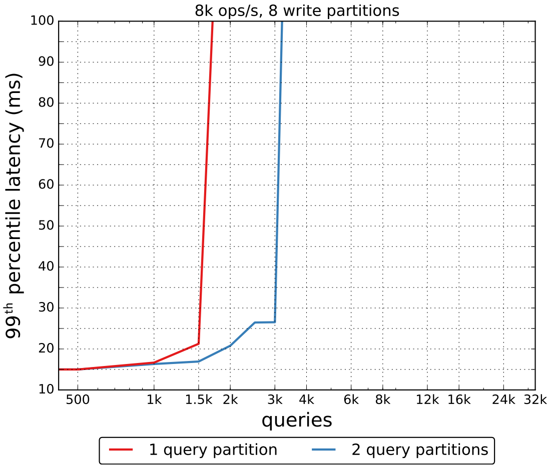


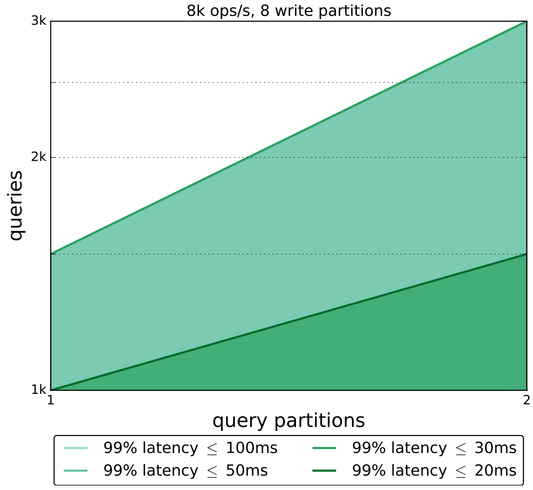


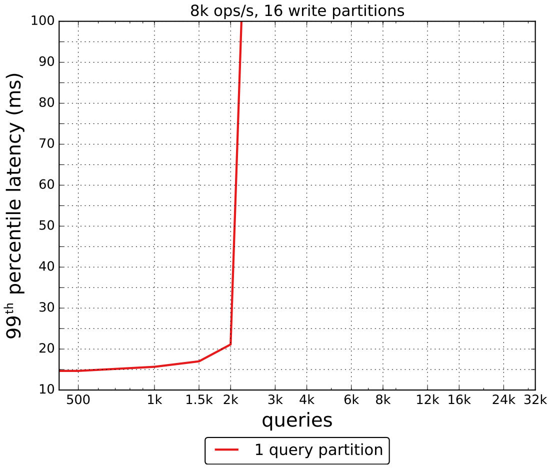



8k ops/s, 4 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions
4 query partitions

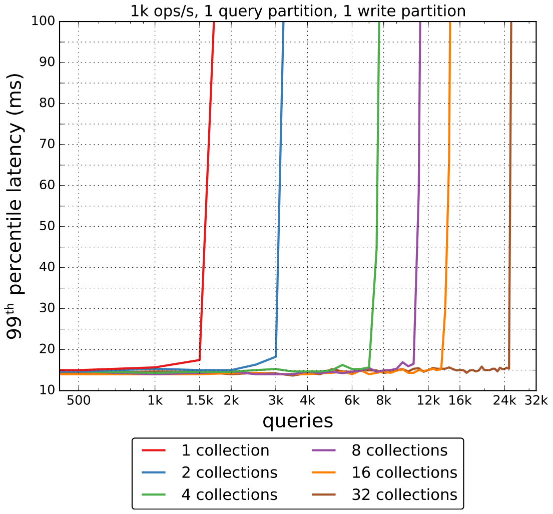

8k ops/s, 4 write partitions
queries
3k
2k
1k
500
1
2
4
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

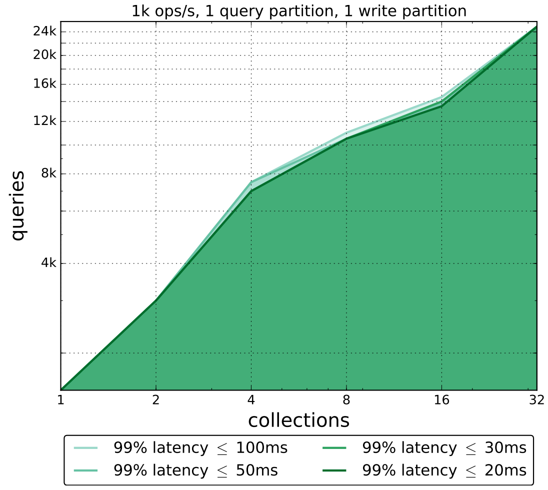

8k ops/s, 8 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition
2 query partitions

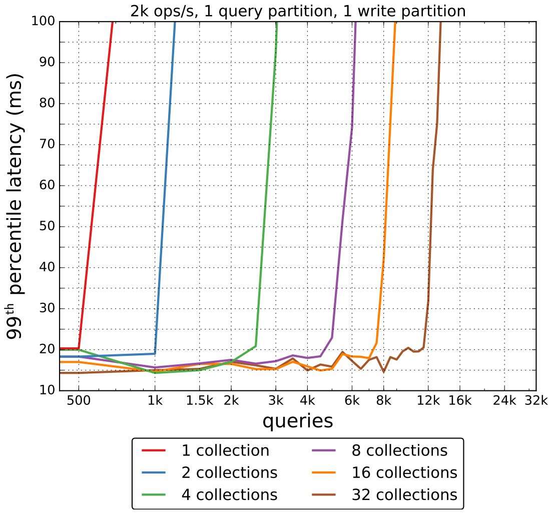

8k ops/s, 8 write partitions
queries
3k
2k
1k
1
2
query partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

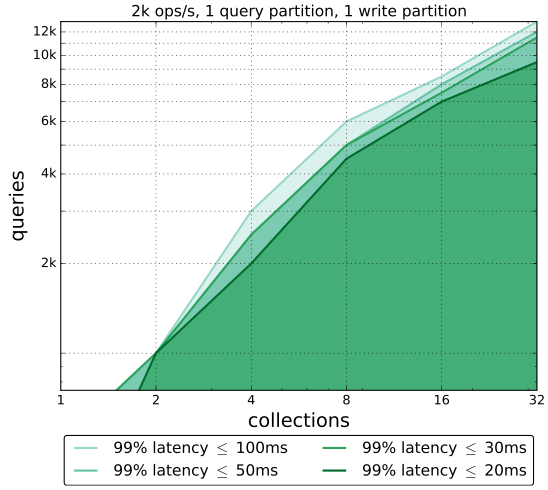

8k ops/s, 16 write partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
1 query partition

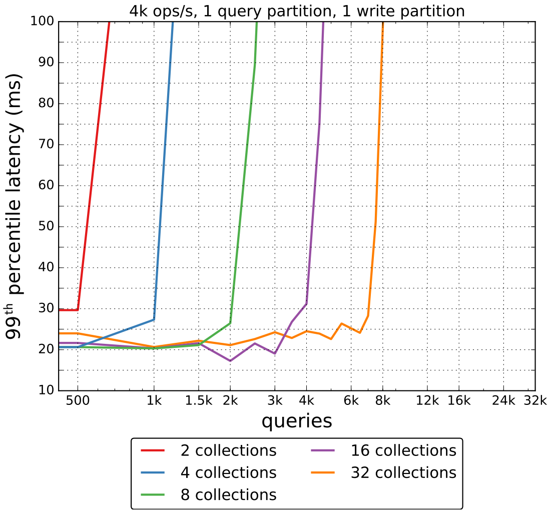
A.2 Sustainable Queries Under Varying Number of Collections

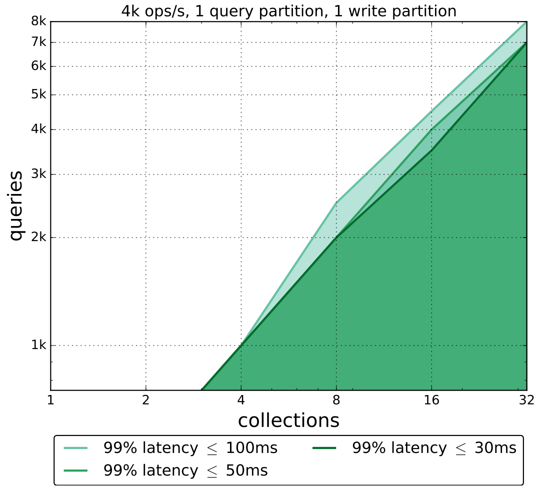


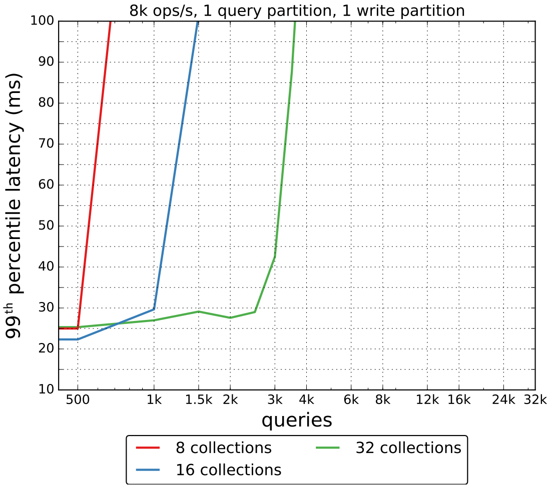


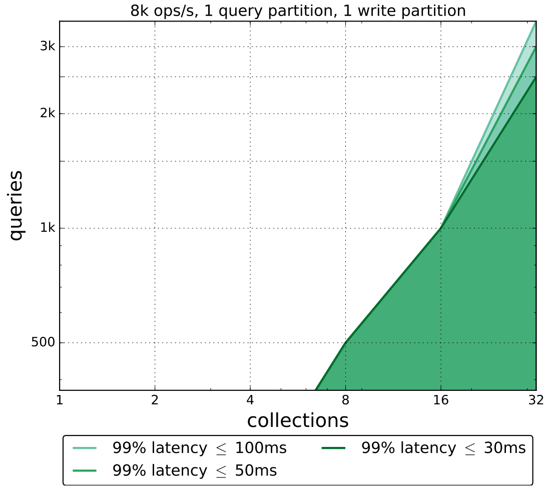





4k ops/s, 1 query partition, 1 write partition
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
2 collections
4 collections
8 collections
16 collections
32 collections


4k ops/s, 1 query partition, 1 write partition
queries
8k
7k
6k
5k
4k
3k
2k
1k
1
2
4
8
16
32
collections
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms


8k ops/s, 1 query partition, 1 write partition
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
queries
8 collections
16 collections
32 collections

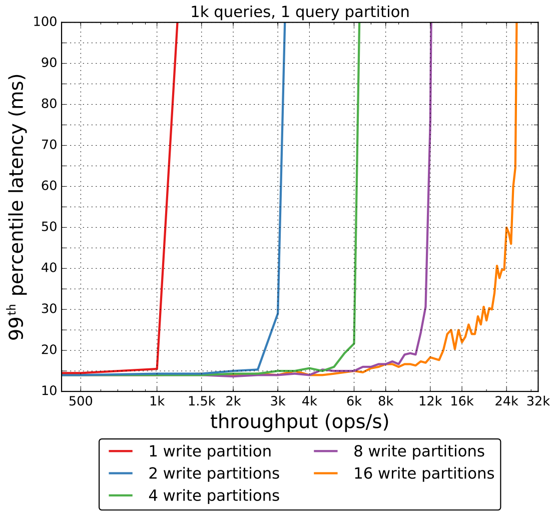

8k ops/s, 1 query partition, 1 write partition
queries
3k
2k
1k
500
1
2
4
8
16
32
collections
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms

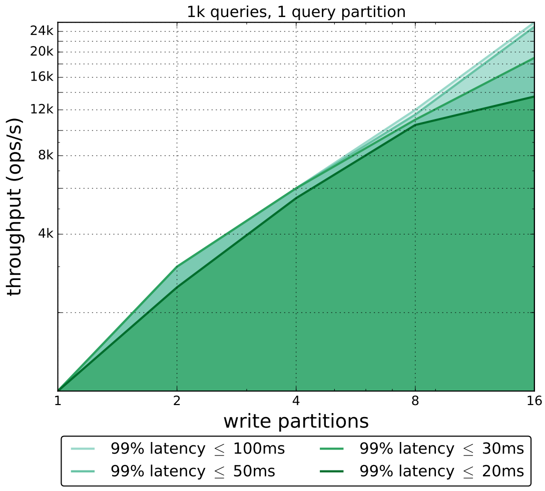
B InvaliDB Performance: Write Scalability

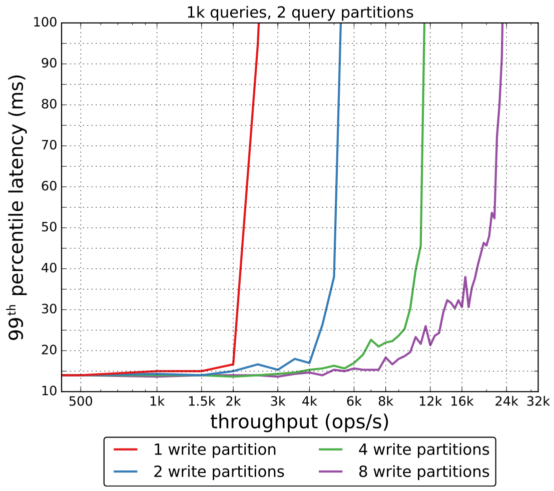
Please note that the following performance plots are scaled logarithmically.

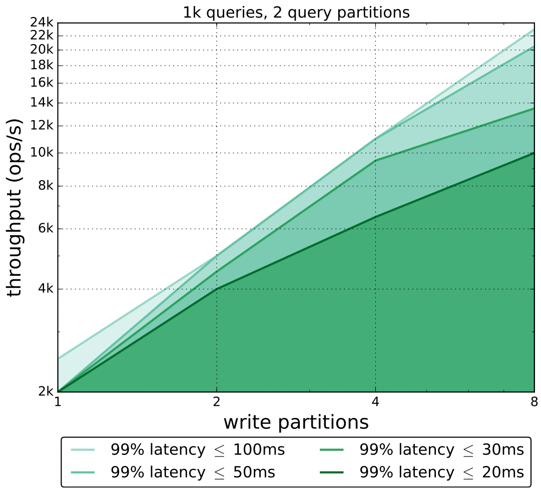
B.1 Sustainable Throughput Under Varying Write Partitions

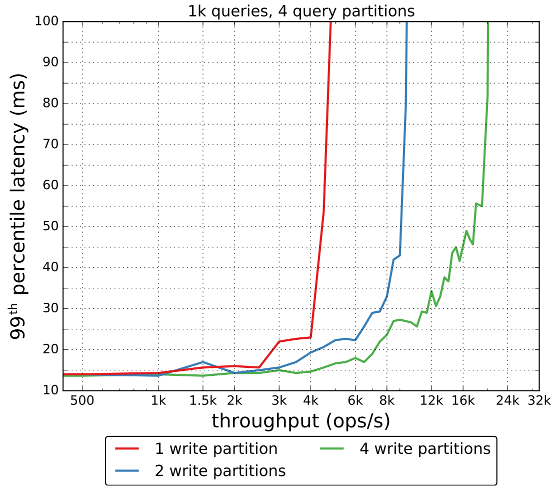
1 000 Queries Fixed, Increasing Throughput

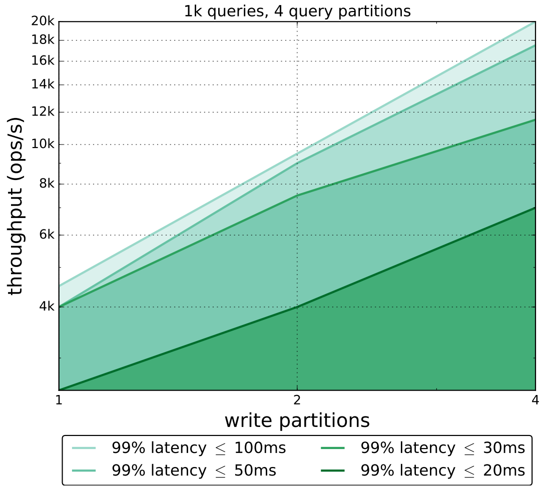


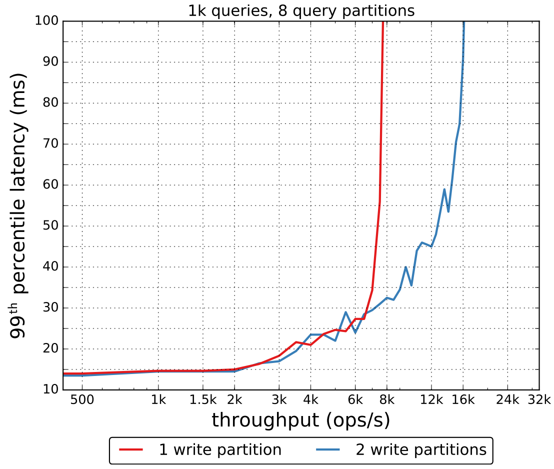


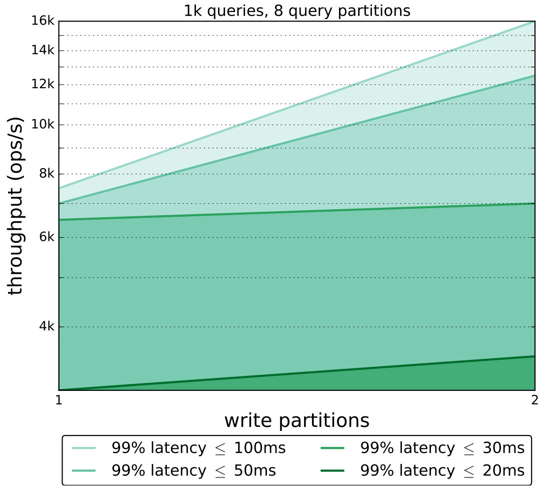


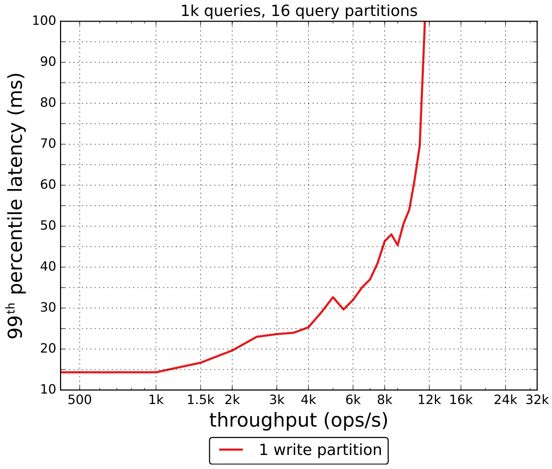



1k queries, 4 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions
4 write partitions

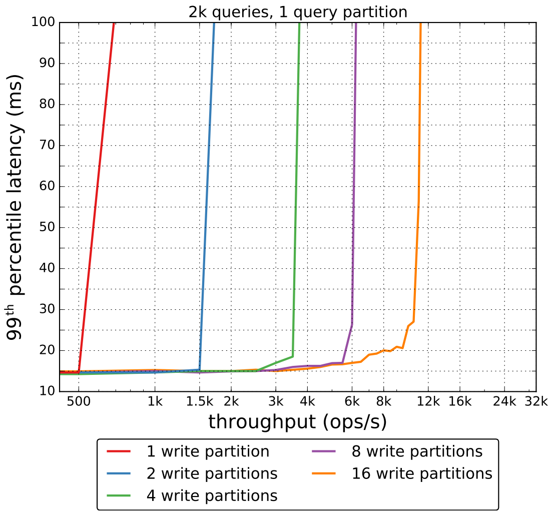

1k queries, 4 query partitions
throughput (ops/s)
20k
18k
16k
14k
12k
10k
8k
6k
4k
1
2
4
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

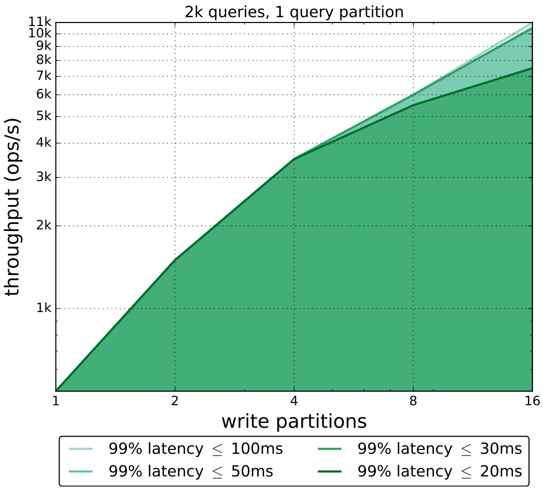

1k queries, 8 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions

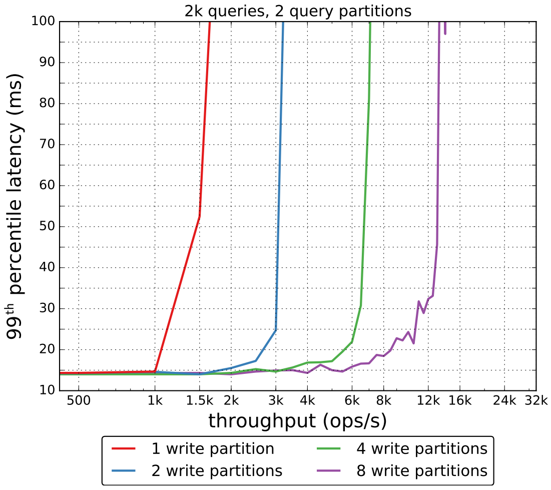

1k queries, 8 query partitions
throughput (ops/s)
16k
14k
12k
10k
8k
6k
4k
1
2
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

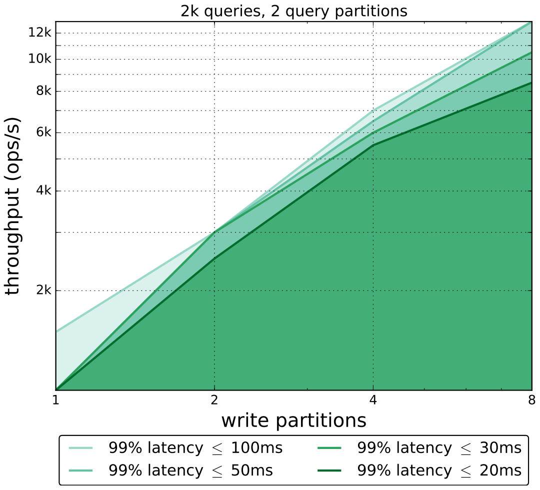

1k queries, 16 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition

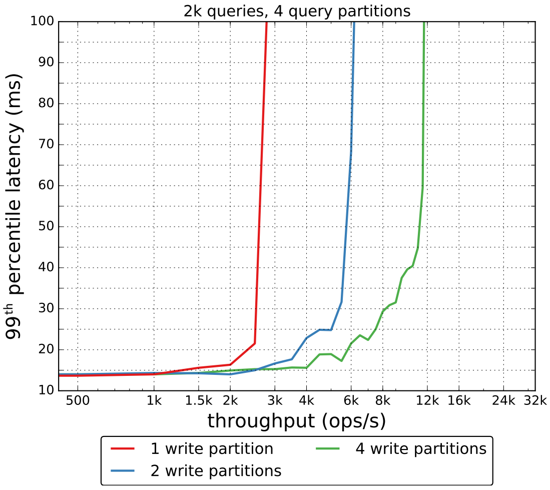
2 000 Queries Fixed, Increasing Throughput

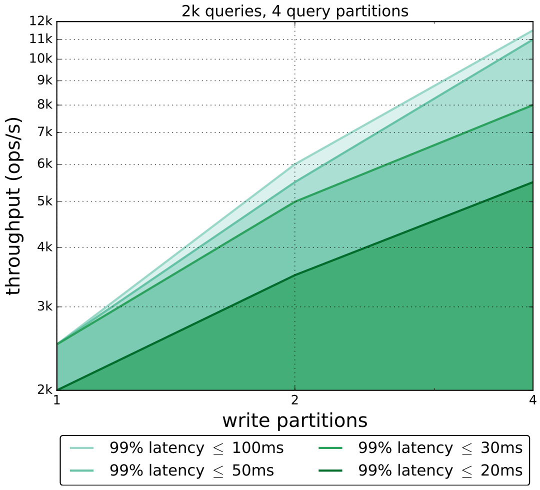


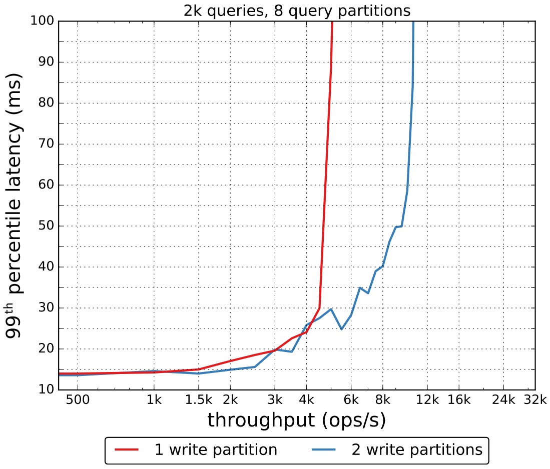


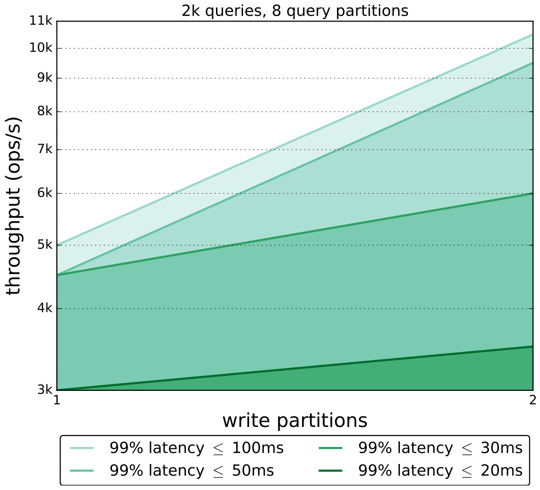


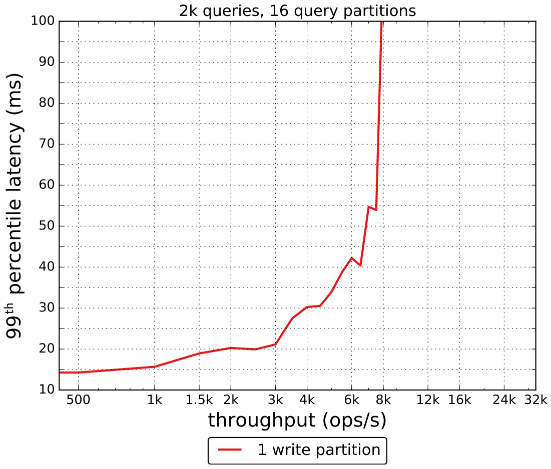



2k queries, 4 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions
4 write partitions

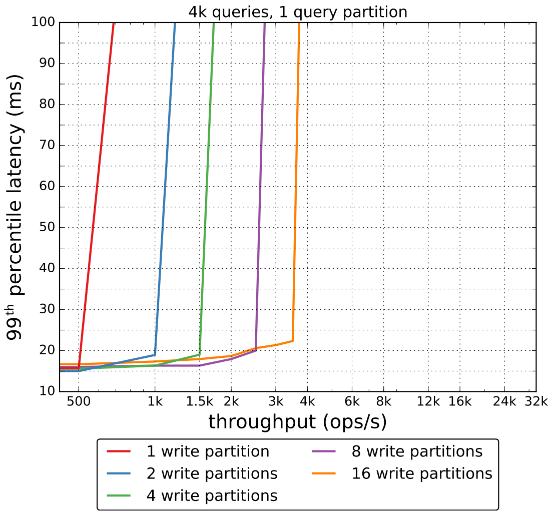

2k queries, 4 query partitions
throughput (ops/s)
12k
11k
10k
9k
8k
7k
6k
5k
4k
3k
2k
1
2
4
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

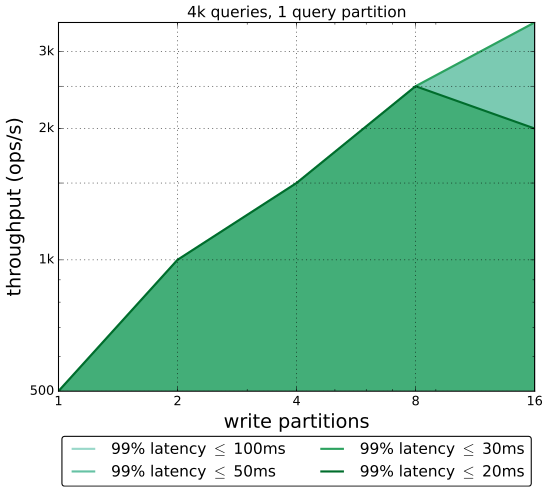

2k queries, 8 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions

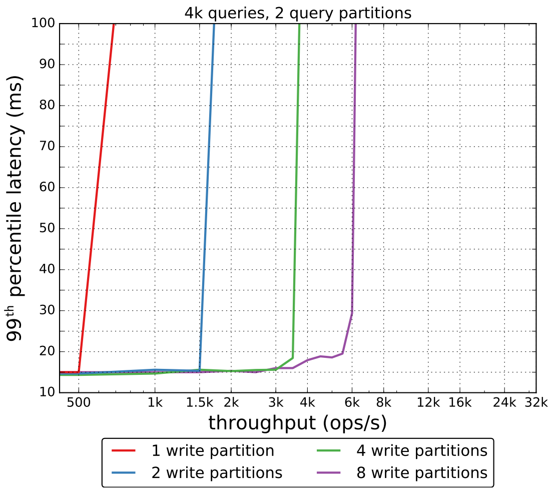

2k queries, 8 query partitions
throughput (ops/s)
11k
10k
9k
8k
7k
6k
5k
4k
3k
1
2
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

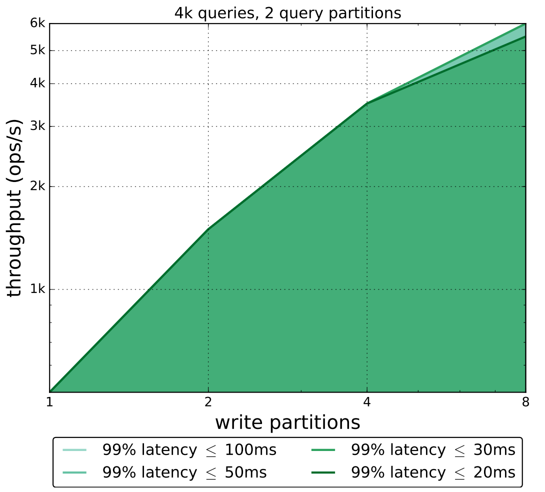

2k queries, 16 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition

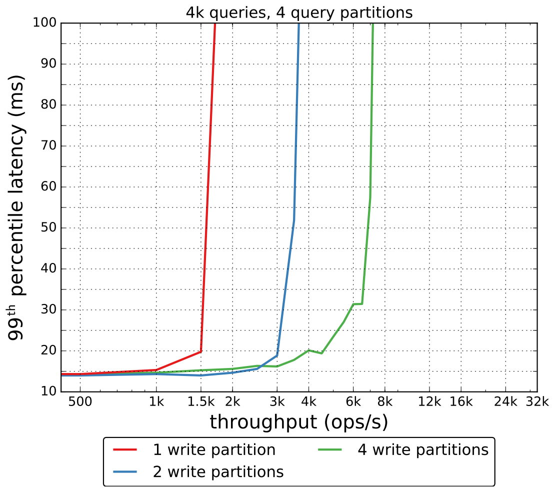
4 000 Queries Fixed, Increasing Throughput

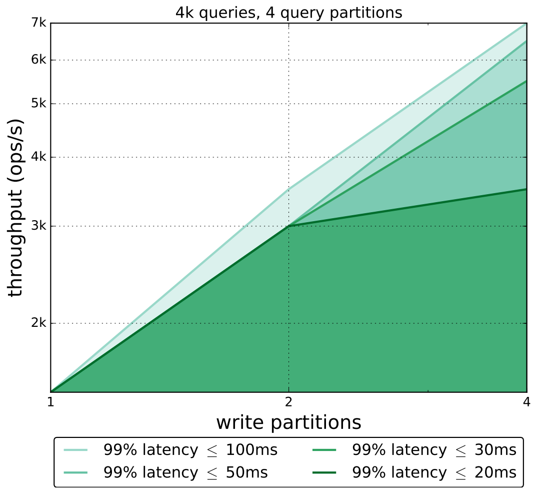


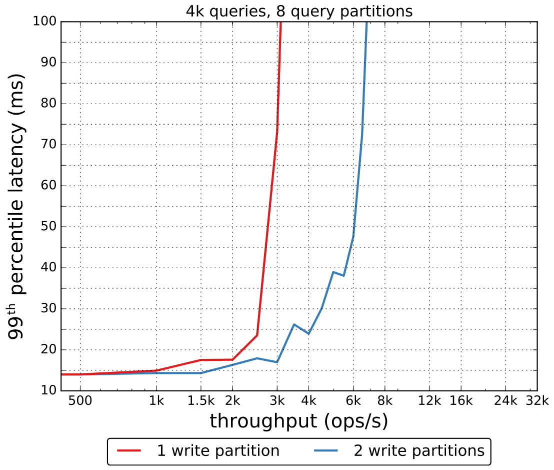


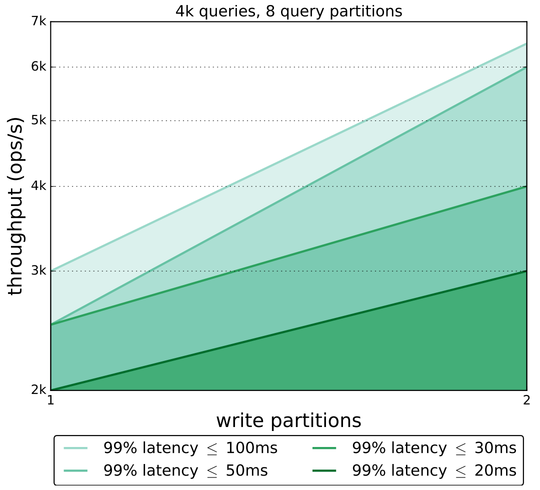


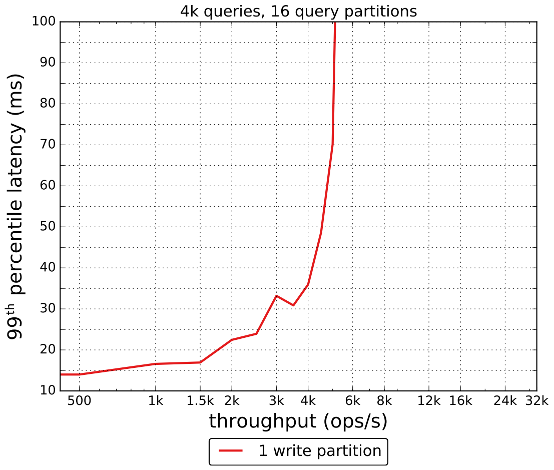



4k queries, 4 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions
4 write partitions

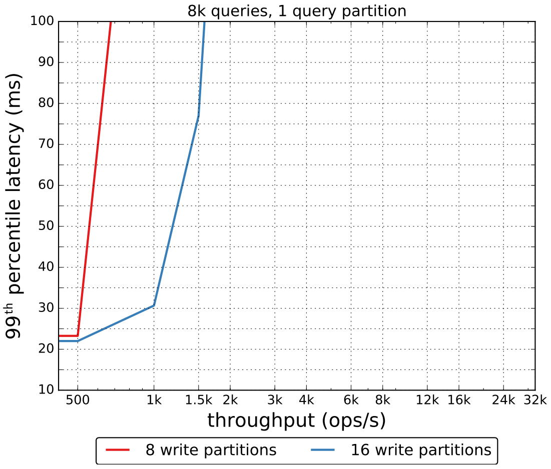

4k queries, 4 query partitions
throughput (ops/s)
7k
6k
5k
4k
3k
2k
1
2
4
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

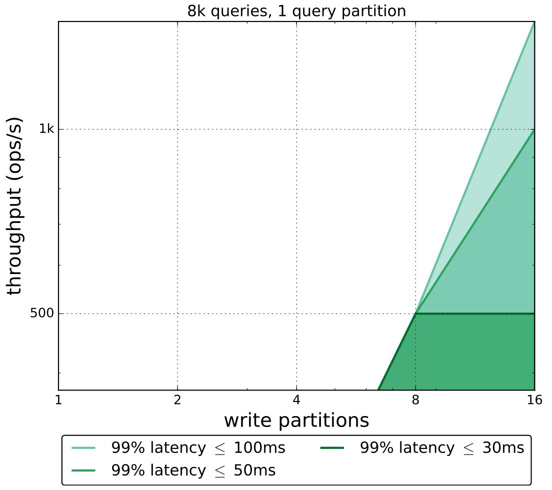

4k queries, 8 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions

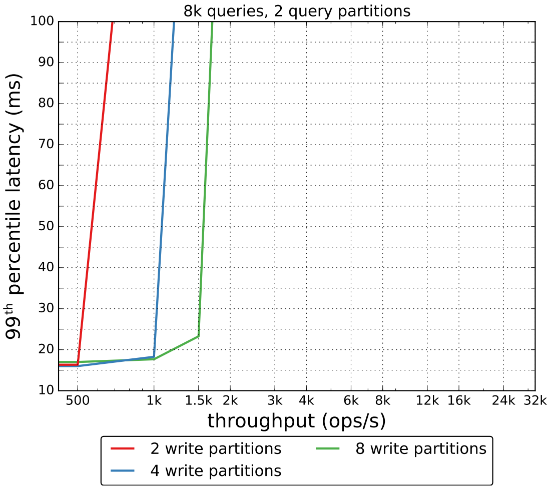

4k queries, 8 query partitions
throughput (ops/s)
7k
6k
5k
4k
3k
2k
1
2
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

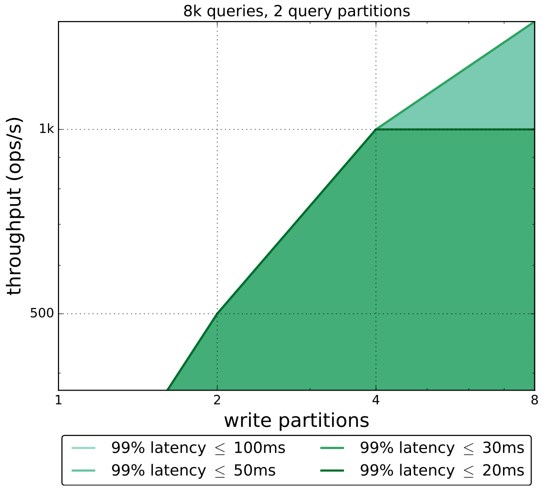

4k queries, 16 query partitions
99th percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition

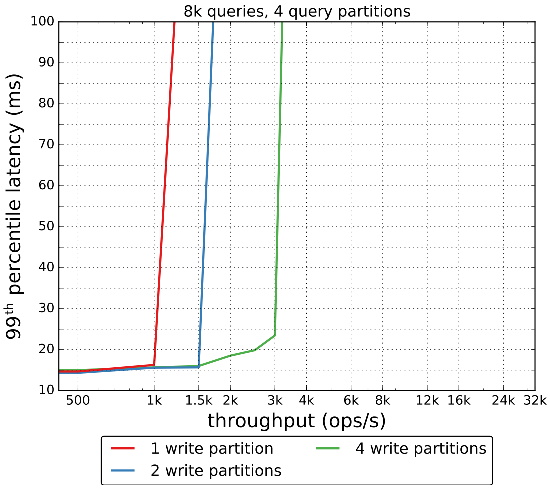
8 000 Queries Fixed, Increasing Throughput

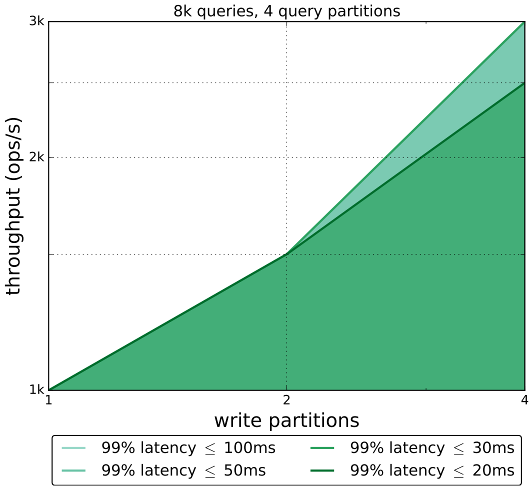


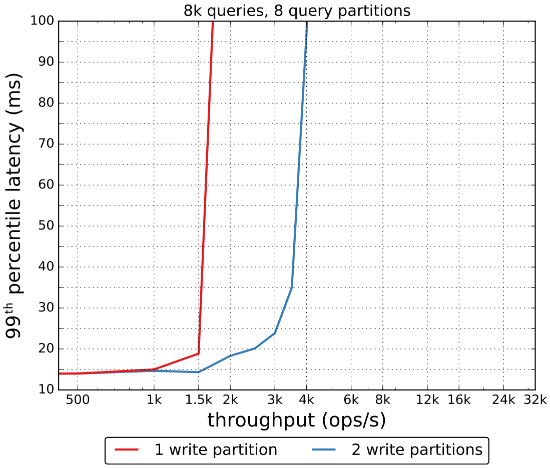


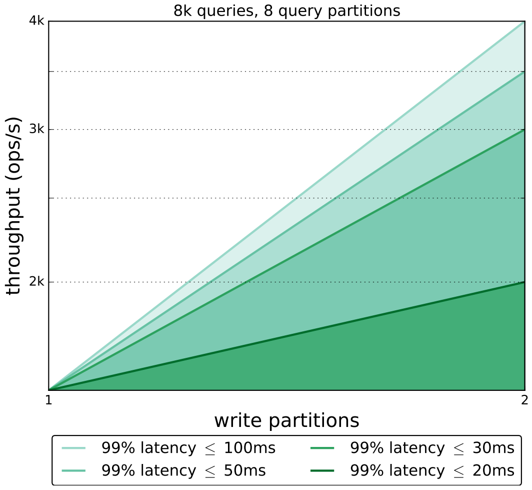


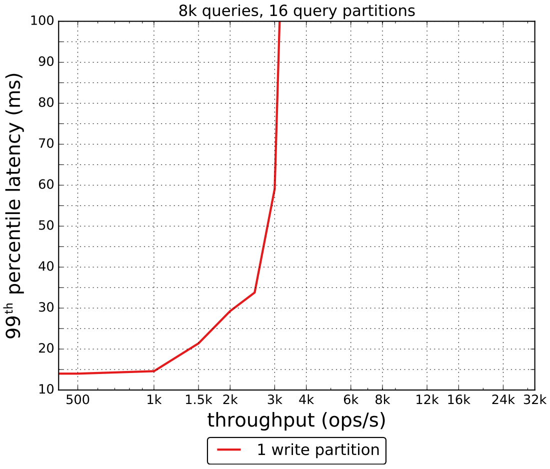



8k queries, 4 query partitions
99^{th} percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions
4 write partitions

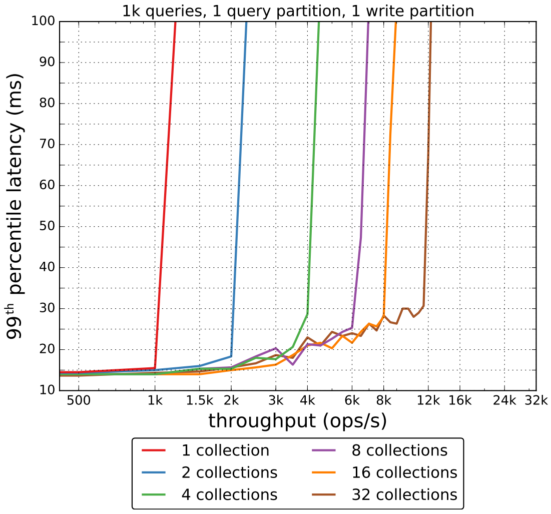

8k queries, 4 query partitions
throughput (ops/s)
3k
2k
1k
1
2
4
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

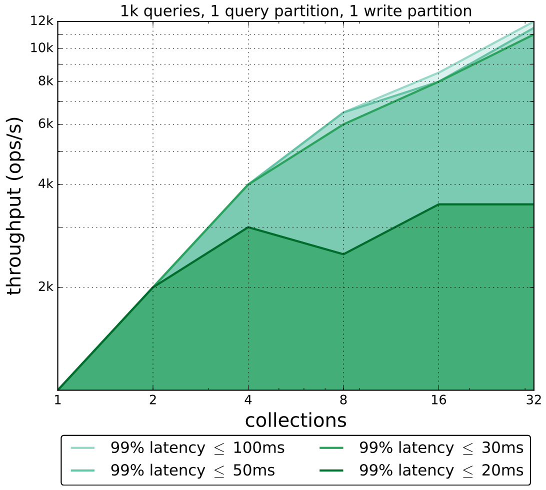

8k queries, 8 query partitions
99^{th} percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition
2 write partitions

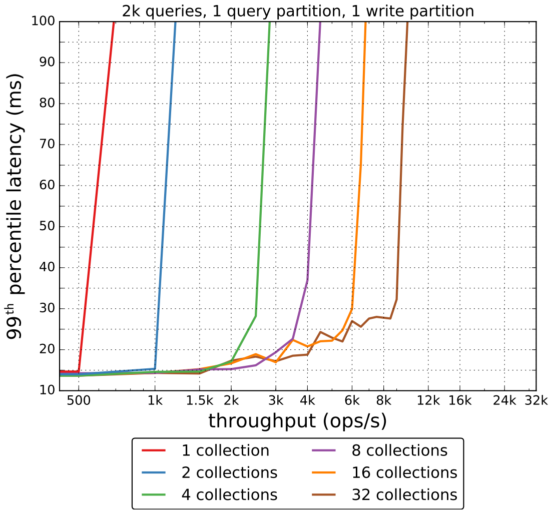

8k queries, 8 query partitions
throughput (ops/s)
4k
3k
2k
1
2
write partitions
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

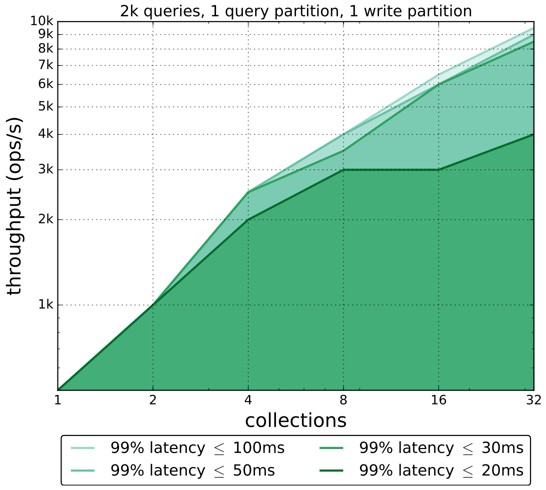

8k queries, 16 query partitions
99^{th} percentile latency (ms)
100
90
80
70
60
50
40
30
20
10
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 write partition

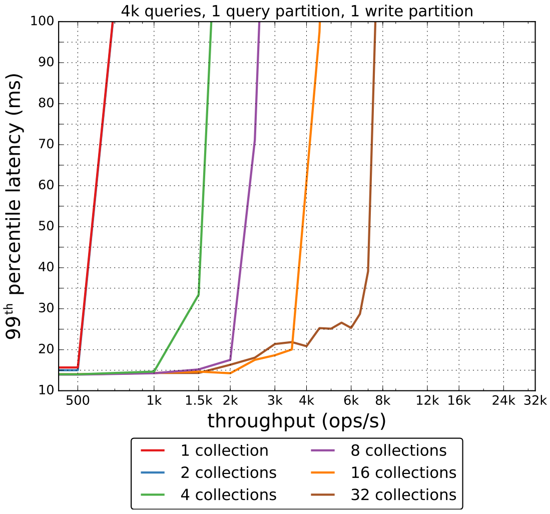
B.2 Sustainable Throughput Under Varying Number of Collections

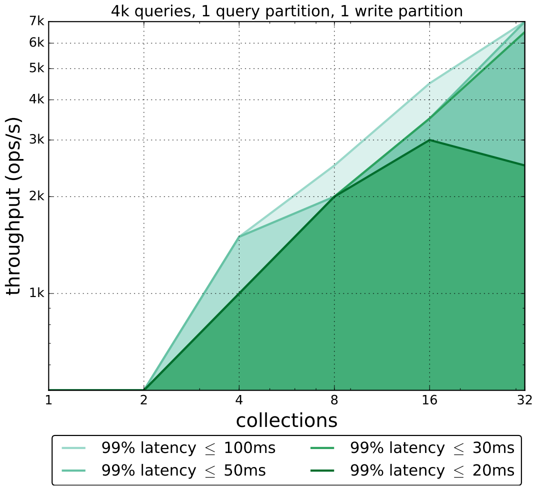


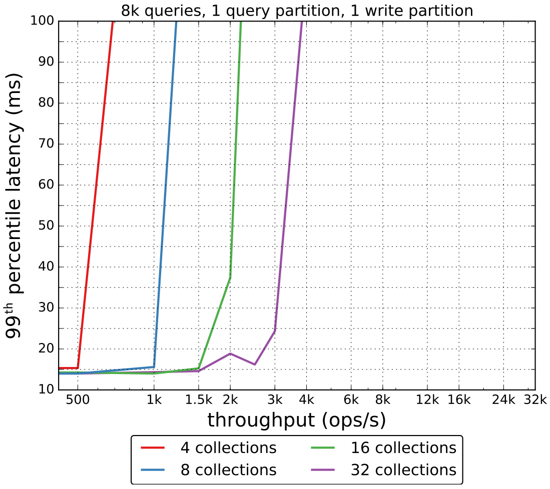


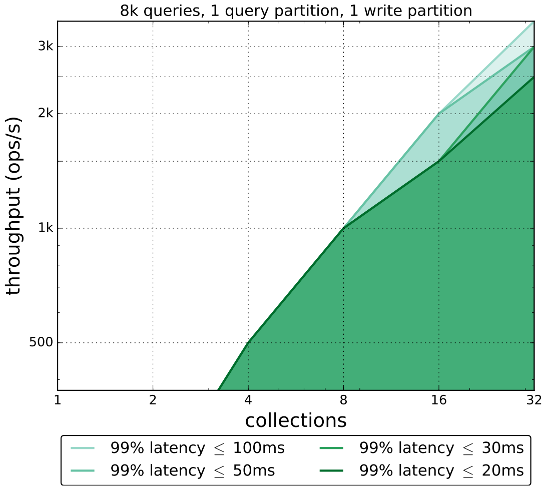





4k queries, 1 query partition, 1 write partition
99th percentile latency (ms)
10
20
30
40
50
60
70
80
90
100
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
1 collection
2 collections
4 collections
8 collections
16 collections
32 collections


4k queries, 1 query partition, 1 write partition
throughput (ops/s)
1k
2k
3k
4k
5k
6k
7k
1
2
4
8
16
32
collections
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

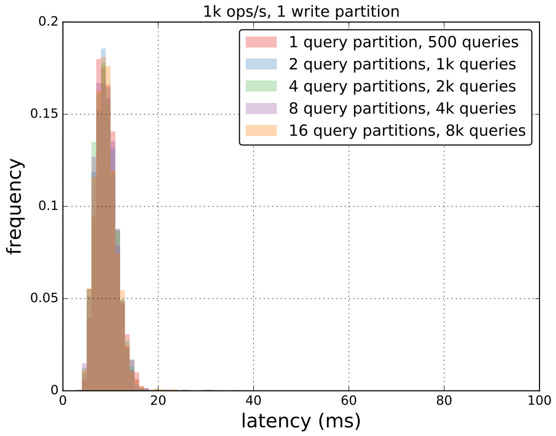

8k queries, 1 query partition, 1 write partition
99th percentile latency (ms)
10
20
30
40
50
60
70
80
90
100
500
1k
1.5k
2k
3k
4k
6k
8k
12k
16k
24k
32k
throughput (ops/s)
4 collections
8 collections
16 collections
32 collections

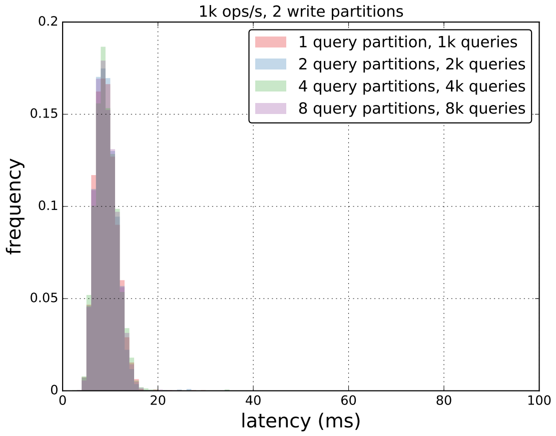

8k queries, 1 query partition, 1 write partition
throughput (ops/s)
500
1k
2k
3k
1
2
4
8
16
32
collections
99% latency ≤ 100ms
99% latency ≤ 50ms
99% latency ≤ 30ms
99% latency ≤ 20ms

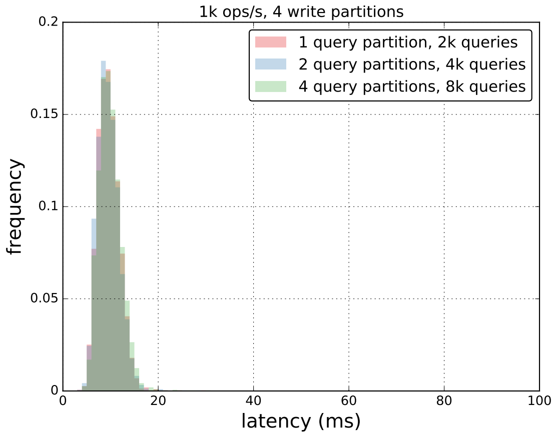
C InvaliDB Performance: Latency Distribution

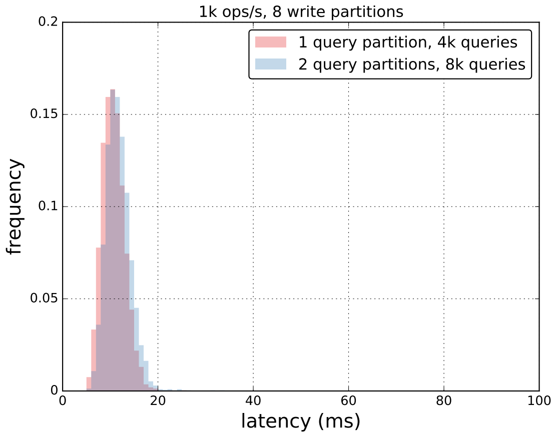
C.1 Read Scalability: Latency Distribution Under Varying Query Partitions

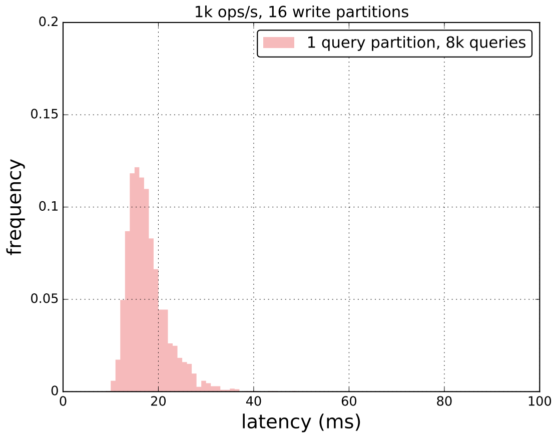
Fixed Relative Load: 1 000 ops/s, 500 Queries per Node





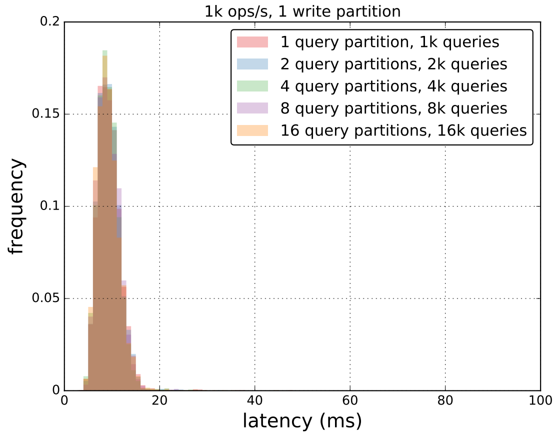


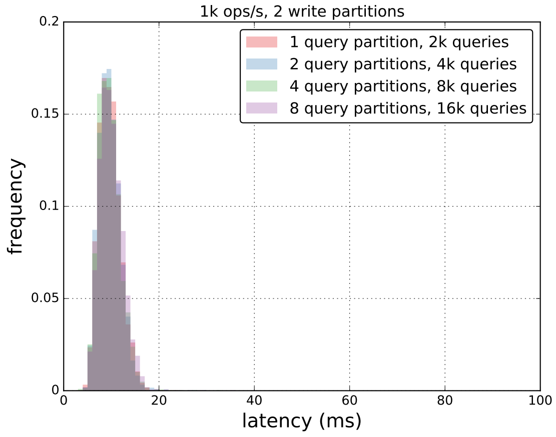


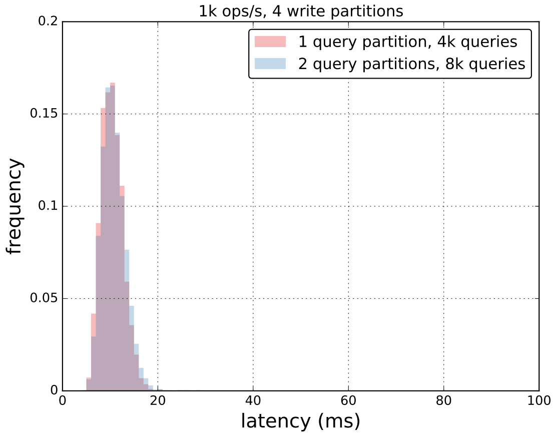


Fixed Relative Load: 1 000 ops/s, 1 000 Queries per Node

There are no histograms for InvaliDB configurations with 4 write partitions and 4 query partitions, 8 write partitions, or 16 write partitions below, because none of them was able to sustain 1 000 queries per matching node at 1 000 ops/s (cf. page 160).

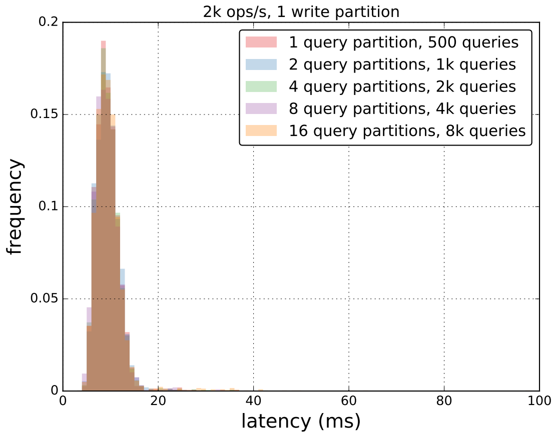


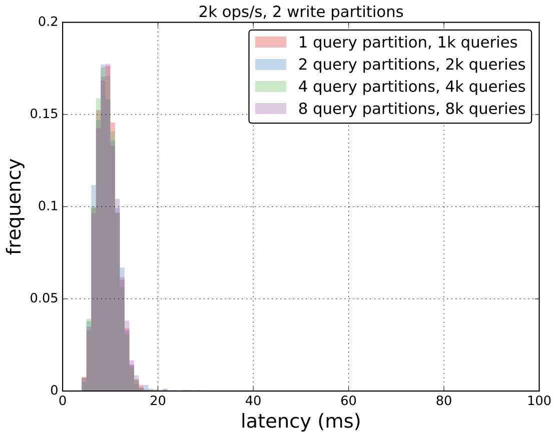


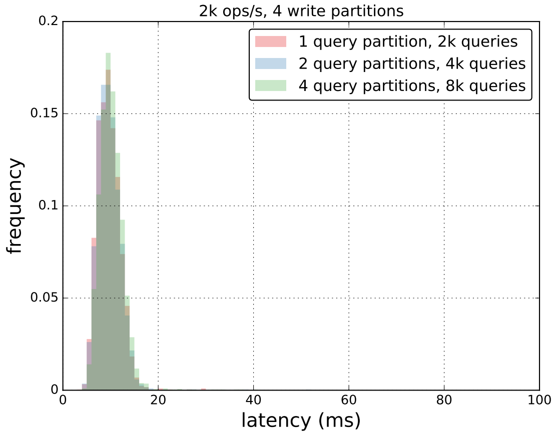


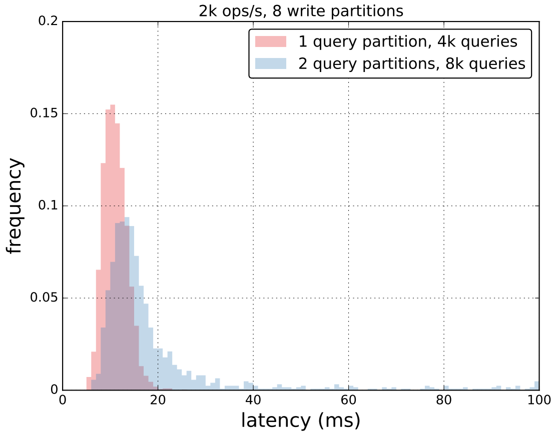
Fixed Relative Load: 2 000 ops/s, 500 Queries per Node

There is no histogram for the InvaliDB configuration with 16 write partitions below, because it was not able to sustain 8 000 queries at 2 000 ops/s (cf. page 162).



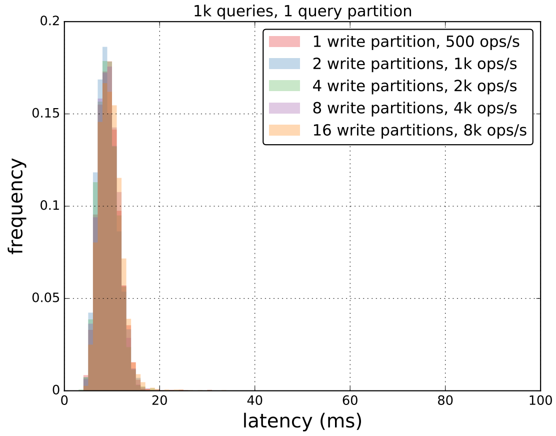


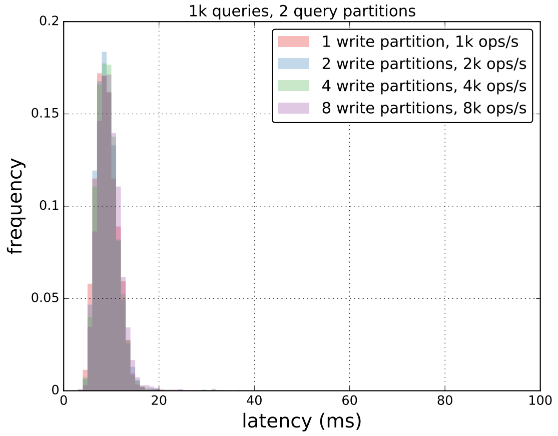


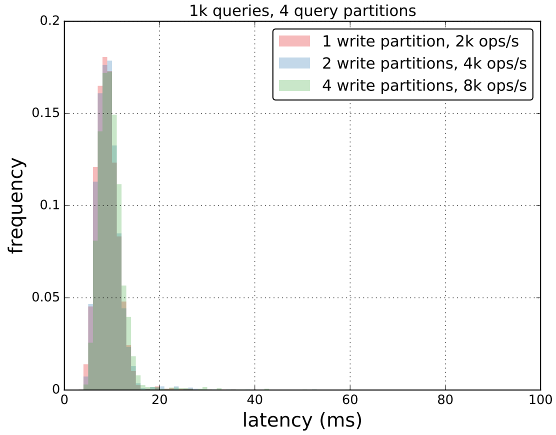


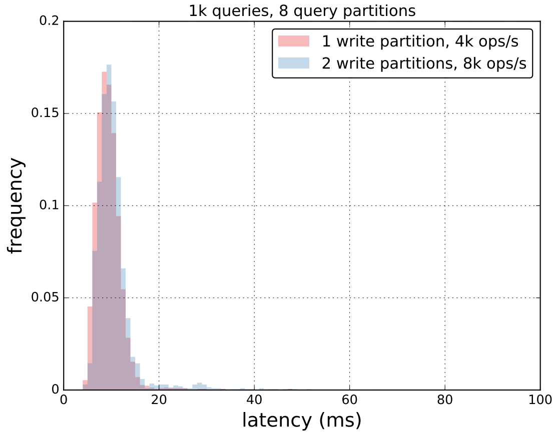
C.2 Write Scalability: Latency Distribution Under Varying Write Partitions

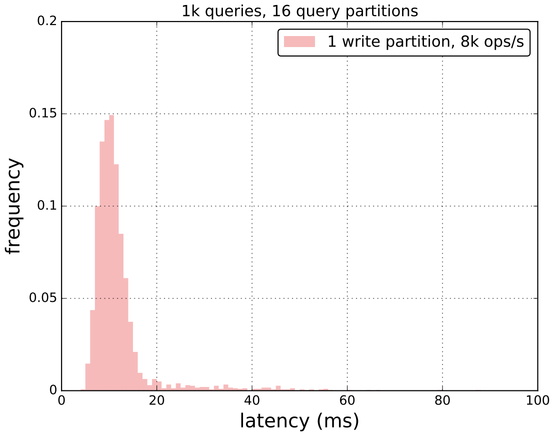
Fixed Relative Load: 1 000 queries, 500 ops/s per Node





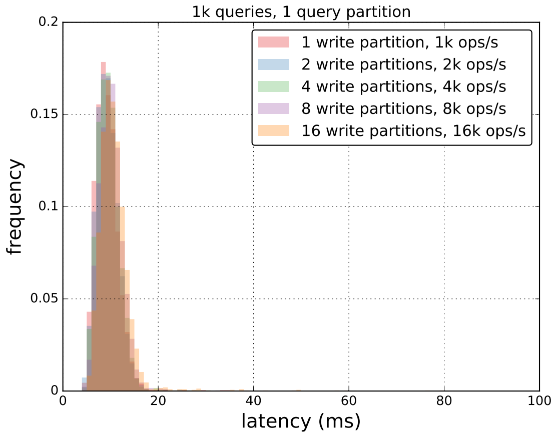


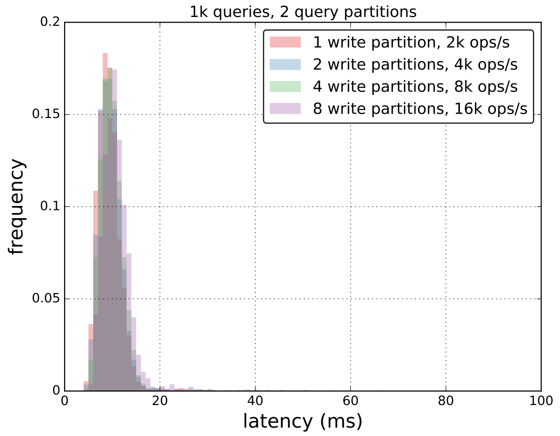


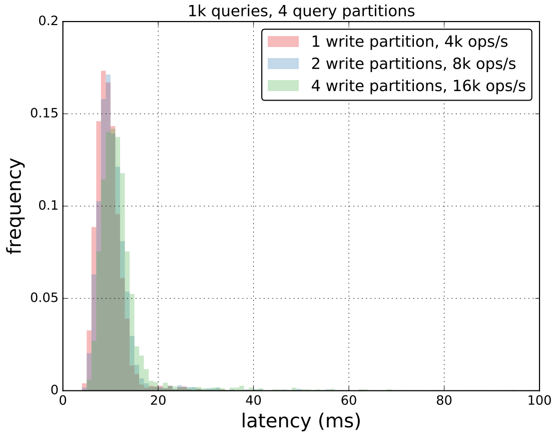


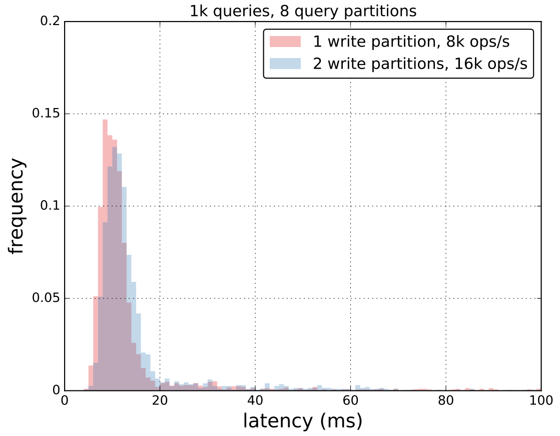
Fixed Relative Load: 1 000 queries, 1 000 ops/s per Node

There is no histogram for the InvaliDB configuration with 16 query partitions below, because it was not able to sustain 1 000 queries at 16 000 ops/s (cf. page 170).

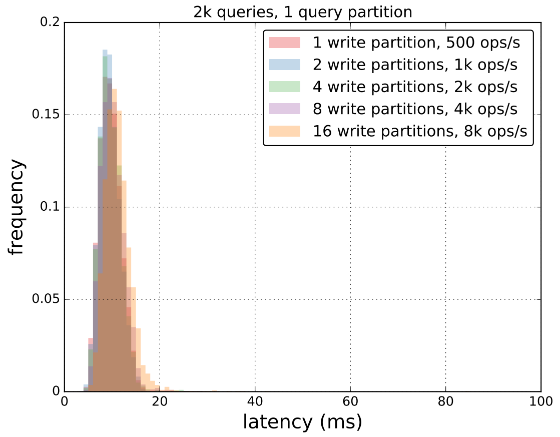


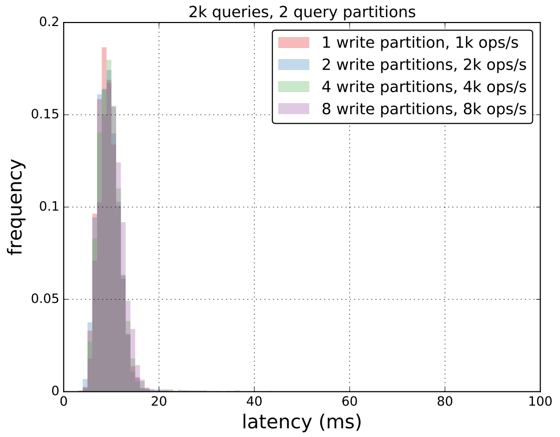


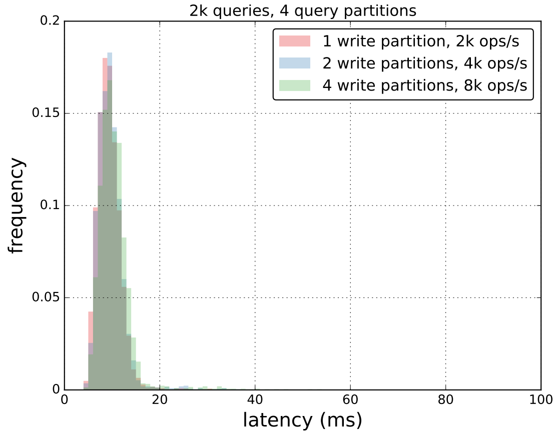


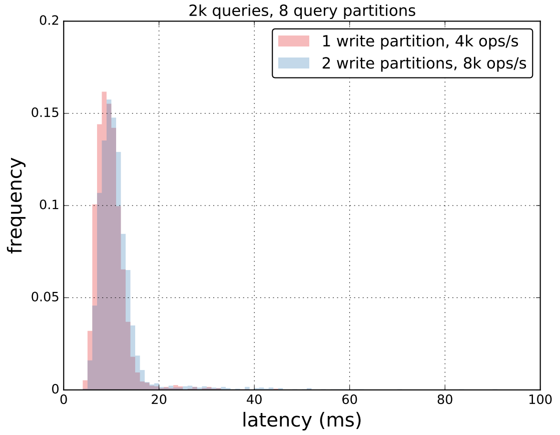


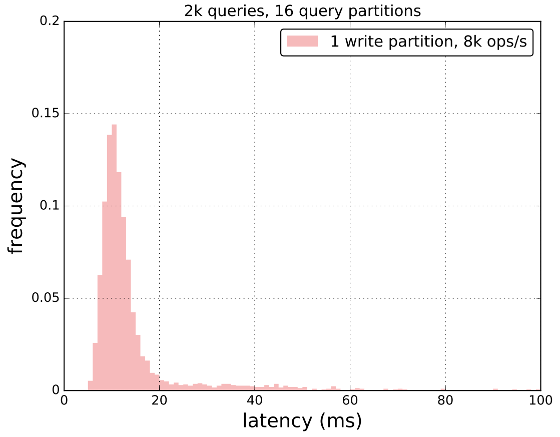
Fixed Relative Load: 2 000 queries, 500 ops/s per Node


2k queries, 1 query partition


2k queries, 2 query partitions

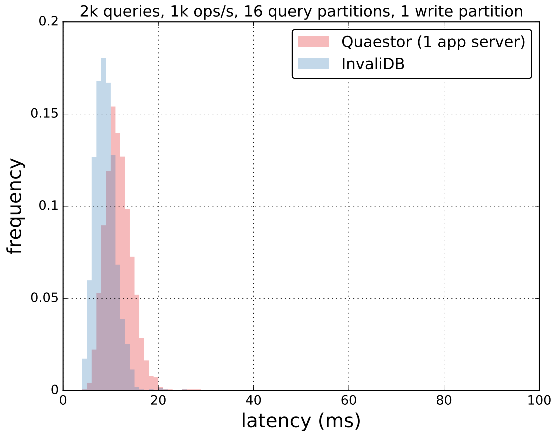

2k queries, 4 query partitions

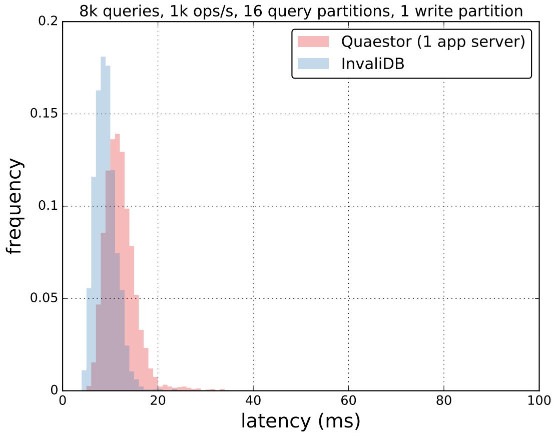

2k queries, 8 query partitions

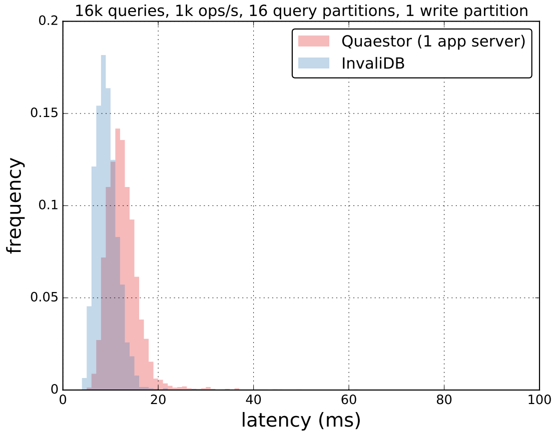

2k queries, 16 query partitions

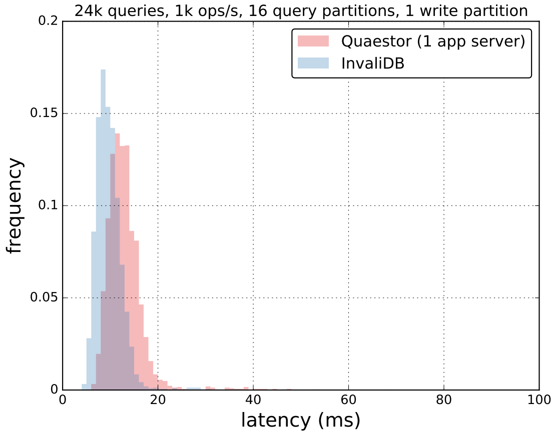
C.3 Comparison With Quaestor

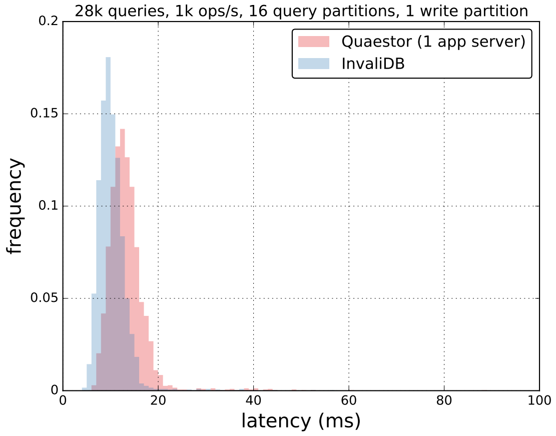
Latency Distribution Under Increasing Query Load

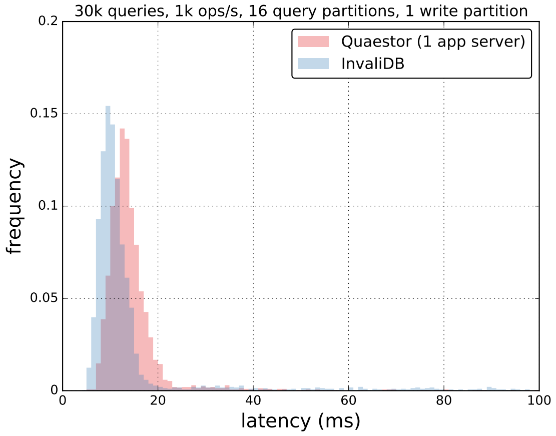




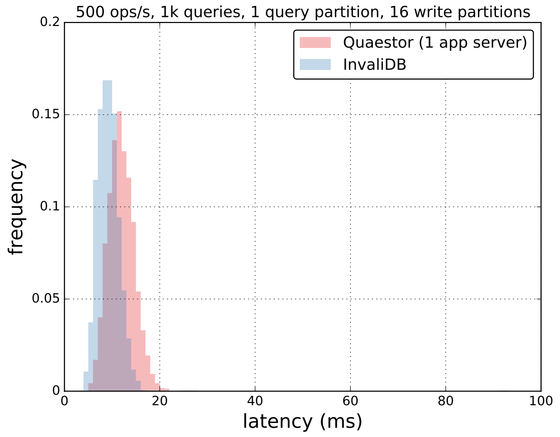


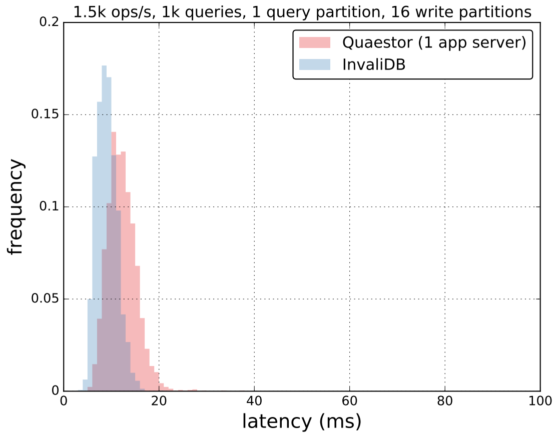


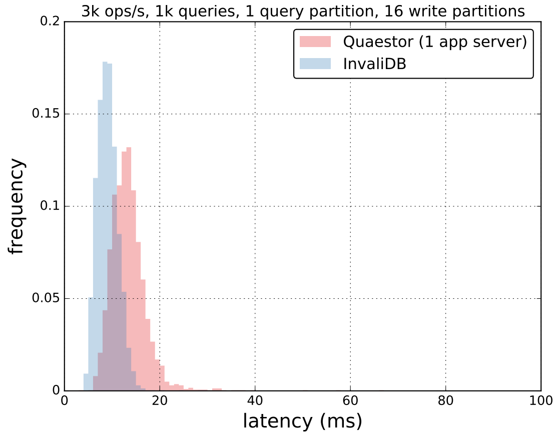


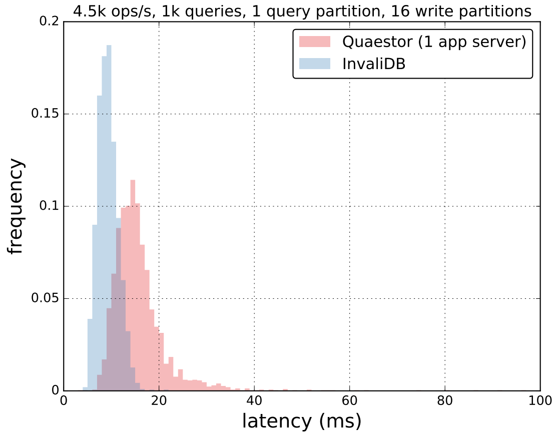


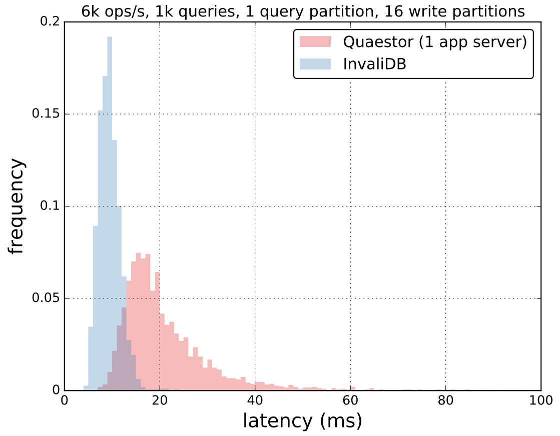
Latency Distribution Under Increasing Write Load

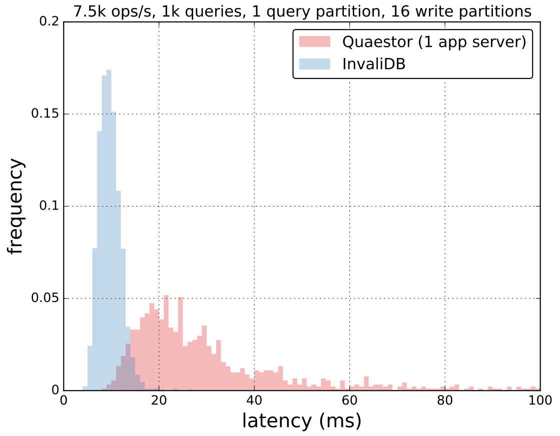












D Websocket Messaging Protocol for Quaestor's Real-Time Queries

Section 5.2.3 provides an overview over the messages that are exchanged between Quaestor's client and server to provide real-time queries. The following provides examples and a detailed description of the attributes each message type carries.

Essentially, the messages can be separated into two basic types:

- **Client-to-server (request)** messages are sent from the client towards the server to initialize or terminate a real-time query. A request message is always one of the following:
  - *subscribe message*: initializes a real-time query.
  - *unsubscribe message*: terminates a real-time query.
- **Server-to-client (response)** messages are sent from the server to the client to keep the client up-to-date on the subscribed query's result. A response message is always one of the following.
  - *result message*: indicates that the real-time query subscription has been activated. If requested on subscription, the result message carries the initial query result, i.e. all data items matching the query on subscription.
  - *match message*: carries a result change delta.
  - *error message*: terminates a real-time query because of some problem.

Every request and response message contains the following attributes:

- `id`: a universally unique identifier (UUID) generated by the client and initially provided with the subscription message; required to distinguish between different subscriptions transmitted over the same websocket connection.
- `type`: required to distinguish between the different kinds of messages (`subscribe`, `unsubscribe`, `result`, `match`, `error`).

Request Messages: From Client to Server

There are only two types of messages that are sent from the client to the server via websocket: subscribe messages and unsubscribe messages.

A **subscribe message** initializes a real-time query and therefore encapsulates what information the client is interested in. A subscribe message may contain the following specific attributes:

- `token`: authorization user token or `null` for queries executed anonymously.
- `initial`: a boolean value to indicate whether or not the initial result is requested.

- `bucket`: the collection on which the query is performed.
- `query`: a MongoDB query as string (will be parsed into a JSON document server-side).
- `sort`: a MongoDB sort document [Mon18a] as string (will be parsed into a JSON document server-side)
- `offset`: an integer indicating the number of items to skip; e.g. an offset of 20 in a sorted query result means the result will only contain items with index 20, 21, 22 and so forth.
- `limit`: an integer indicating the maximum number of items in the result.
- `operations`: the operations that can trigger an event.
- `matchTypes`: the match types that the client is interested in.

As illustrated in Listing D.1, the client has to provide a unique string (`UUID`) for the `id` attribute when creating a subscribe message.

```
{
  'id': 'be22181e-8561-445f-99a1-97ec2d4e253f',
  'type': 'subscribe',
  'token': null,
  'initial': true,
  'bucket': 'User',
  'query': '{\'value\' : 42}',
  'sort': '{\'name\': 1}',
  'offset': 20,
  'limit': 10,
  'operations': [ 'any' ],
  'matchTypes': [ 'all' ]
}
```

Listing D.1: An exemplary subscribe message.

An **unsubscribe message** terminates the specified subscription. It does not have any specific attributes (see Listing D.2).

```
{
  'id': 'be22181e-8561-445f-99a1-97ec2d4e253f',
  'type': 'unsubscribe'
}
```

Listing D.2: An exemplary unsubscribe message.

Response Messages: From Server to Client

In addition to the mandatory `id` and `type` attributes, every message issued by the server has a `date` attribute that designates the server time at which the message was generated.

A **result message** is always the first message sent to the client as it indicates the activation of the real-time query subscription. It contains the complete initial query result when and only when `initial` was specified as `true` on subscription. If present, the initial result is given as a list of JSON documents under the `data` attribute; for sorted queries, item order obeys the specified order. The result message shown in Listing D.3 carries an initial result with only a single item.

```
{
  'id': 'be22181e-8561-445f-99a1-97ec2d4e253f',
  'type': 'result',
  'date': '2017-08-17T05:02:46.467Z',
  'data': [ {
    'id': '/db/User/23edf763-031a-42c1-b841-0d5a470f117d',
    'version': 1,
    'acl': null,
    'createdAt': '2017-08-17T05:02:45.54Z',
    'updatedAt': '2017-08-17T05:02:45.54Z',
    'date': null,
    'geo': null,
    'name': 'Tim',
    'ref': null,
    'value': 42
  } ]
}
```

Listing D.3: An exemplary result message.

A **match message** encapsulates a change delta, i.e. a write operation that affected the query result. A match message contains the attributes shown in Listing D.4:

- `matchType`: the match type as described in Section 3.2.1, for example `add` for an entity that entered the result or `remove` for a deleted entity.
- `operation`: the time of operation that triggered the event; `insert`, `update`, `delete`, or `none` (please refer to the parameter description on page 67 and the example in Section 5.4.1 for details).
- `index`: for sorted queries only; the position of the modified entity within the result as a list index (`-1` for a deleted entity).

- `data`: a JSON document representing the after-image of the updated entity.

```
{
  'id': 'be22181e-8561-445f-99a1-97ec2d4e253f',
  'type': 'match',
  'date': '2017-08-17T05:02:46.848Z',
  'matchType': 'add',
  'operation': 'insert',
  'index': 0,
  'data': {
    'id': '/db/User/4ab39068-de17-45c2-b9b3-c571d7542c8b',
    'version': 1,
    'acl': null,
    'createdAt': '2017-08-17T05:02:46.829Z',
    'updatedAt': '2017-08-17T05:02:46.829Z',
    'date': null,
    'geo': null,
    'name': 'Al',
    'ref': null,
    'value': 42
  }
}
```

Listing D.4: An exemplary match message.

An **error message** terminates a real-time query subscription and informs the client that there was a problem. An error message always contains the `error` attribute with a JSON document that holds details on the problem that occurred (see Listing D.5 for an example).

```
{
  'id': 'be22181e-8561-445f-99a1-97ec2d4e253f',
  'type': 'error',
  'date': '2017-08-17T05:07:38.008Z',
  'error': {
    'message': 'Too many active subscriptions (only 20
       allowed)!',
    'status': 429,
    'reason': 'Too Many Requests'
  }
}
```

Listing D.5: An exemplary error message.

Bibliography

[AAB+05] Abadi, Daniel J. ; Ahmad, Yanif ; Balazinska, Magdalena ; Cetintemel, Ugur ; Cherniack, Mitch ; Hwang, Jeong-Hyon ; Lindner, Wolfgang ; Maskey, Anurag S. ; Rasin, Alexander ; Ryvkina, Esther ; Tatbul, Nesime ; Xing, Ying ; Zdonik, Stan: The Design of the Borealis Stream Processing Engine. In: *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. Asilomar, CA, January 2005

[ABB+13] Akidau, Tyler ; Balikov, Alex ; Bekiroglu, Kaya et al.: MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In: *Very Large Data Bases*, 2013, pp. 734–746

[ABB+16] *Chapter* STREAM: The Stanford Data Stream Management System. In: Arasu, Arvind ; Babcock, Brian ; Babu, Shivnath ; Cieslewicz, John ; Datar, Mayur ; Ito, Keith ; Motwani, Rajeev ; Srivastava, Utkarsh ; Widom, Jennifer: *Data Stream Management: Processing High-Speed Data Streams*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2016. – ISBN 978–3–540–28608–0, 317–336

[ABC+15] Akidau, Tyler ; Bradshaw, Robert ; Chambers, Craig ; Chernyak, Slava ; Fernández-Moctezuma, Rafael J. ; Lax, Reuven ; McVeety, Sam ; Mills, Daniel ; Perry, Frances ; Schmidt, Eric ; Whittle, Sam: The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. In: *Proceedings of the VLDB Endowment* 8 (2015), pp. 1792–1803

[ABD+13] Ananthanarayanan, Rajagopal ; Basker, Venkatesh ; Das, Sumit ; Gupta, Ashish et al.: Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In: *SIGMOD '13*, 2013

[ABE+14] Alexandrov, Alexander ; Bergmann, Rico ; Ewen, Stephan et al.: The Stratosphere Platform for Big Data Analytics. In: *The VLDB Journal* (2014). `http://dx.doi.org/10.1007/s00778-014-0357-y`. – DOI 10.1007/s00778–014–0357–y. – ISSN 1066–8888

[ABW06] Arasu, Arvind ; Babu, Shivnath ; Widom, Jennifer: The CQL Continuous Query Language: Semantic Foundations and Query Execution. In: *The VLDB Journal* 15 (2006), June, no. 2, 121–142. `http://dx.doi.org/10.1007/s00778-004-0147-z`. – DOI 10.1007/s00778–004–0147–z. – ISSN 1066–8888

[ACC+03] Abadi, Daniel J. ; Carney, Don ; Cetintemel, Ugur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stan: Aurora: A New Model and Architecture for Data Stream Management. In: *The VLDB Journal* 12 (2003), August, no. 2, 120–139. http://dx.doi.org/10.1007/s00778-003-0095-z. – DOI 10.1007/s00778–003–0095–z. – ISSN 1066–8888

[AGR+09] Ali, M. H. ; Gerea, C. ; Raman, B. S. ; Sezgin, B. ; Tarnavski, T. ; Verona, T. ; Wang, P. ; Zabback, P. ; Ananthanarayan, A. ; Kirilov, A. ; Lu, M. ; Raizman, A. ; Krishnan, R. ; Schindlauer, R. ; Grabs, T. ; Bjeletich, S. ; Chandramouli, B. ; Goldstein, J. ; Bhat, S. ; Li, Ying ; Di Nicola, V. ; Wang, X. ; Maier, David ; Grell, S. ; Nano, O. ; Santos, I.: Microsoft CEP Server and Online Behavioral Targeting. In: *Proc. VLDB Endow.* 2 (2009), August, no. 2, 1558–1561. http://dx.doi.org/10.14778/1687553.1687590. – DOI 10.14778/1687553.1687590. – ISSN 2150–8097

[AH98] Andler, Sten F. (ed.) ; Hansson, Jörgen (ed.): *Active, Real-Time, and Temporal Database Systems*. Springer Berlin Heidelberg, 1998 (Lecture Notes in Computer Science 1553)

[Aki15] Akidau, Tyler: The world beyond batch: Streaming 101. In: *O'Reilly Media* (2015), August. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101. – Accessed: 2017-05-21

[Aki16] Akidau, Tyler: The world beyond batch: Streaming 102. In: *O'Reilly Media* (2016), January. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102. – Accessed: 2017-05-21

[ALS10] Anderson, J. C. ; Lehnardt, Jan ; Slater, Noah: *CouchDB: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2010. – ISBN 0596155891, 9780596155896

[Ama18] Amazon Kinesis: *Amazon Kinesis*. https://aws.amazon.com/kinesis/. version: 2018. – Accessed: 2018-08-19

[AN04] Ayad, Ahmed M. ; Naughton, Jeffrey F.: Static Optimization of Conjunctive Queries with Sliding Windows over Infinite Streams. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2004 (SIGMOD '04). – ISBN 1–58113–859–8, 419–430

[ANS86] ANSI: X3.135-1986: Information Systems – Database Language – SQL / American National Standards Institute. 1986. – Standard

[Apa15] Apache Software Foundation: The Apache Software Foundation Announces Apache®Flink™as a Top-Level Project. In: *Apache Software Foundation Blog* (2015), January. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces69. – Accessed: 2016-11-25

[Apa16a] Apache Software Foundation: Announcing Apache Flink 1.0.0. In: *Apache Flink Blog* (2016), March. https://flink.apache.org/news/2016/03/08/release-1.0.0.html. – Accessed: 2017-01-15

[Apa16b] Apache Software Foundation: Apache Flink: Powered By Flink. In: *Apache Flink website* (2016). https://flink.apache.org/poweredby.html. – Accessed: 2016-10-17

[Apa16c] Apache Software Foundation: The Apache Software Foundation Announces Apache®Apex™as a Top-Level Project. In: *Apache Software Foundation Blog* (2016), April. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces90. – Accessed: 2016-11-25

[Apa16d] Apache Software Foundation: *Flink*. https://flink.apache.org/. version: 2016. – Accessed: 2016-09-18

[Apa16e] Apache Software Foundation: *Flume*. https://flume.apache.org/. version: 2016. – Accessed: 2016-10-17

[Apa16f] Apache Software Foundation: Level of Parallelism in Data Processing. In: *Spark Streaming – 2.0.0 Documentation* (2016). https://spark.apache.org/docs/2.0.0/streaming-programming-guide.html#level-of-parallelism-in-data-receiving. – Accessed: 2016-09-23

[Apa16g] Apache Software Foundation: Powered By Spark. In: *Apache Spark Website* (2016). https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark. – Accessed: 2016-10-17

[Apa16h] Apache Software Foundation: *YARN*. http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html. version: 2016. – Accessed: 2016-10-17

[Apa17] Apache Software Foundation: *Apache Beam Compatibility Matrix*. https://beam.apache.org/documentation/runners/capability-matrix/. version: 2017. – Accessed: 2017-11-18

[Apa18a] Apache Software Foundation: *ActiveMQ*. https://activemq.apache.org/. version: 2018. – Accessed: 2018-05-10

[Apa18b] Apache Software Foundation: *Apex*. http://apex.apache.org/. version: 2018. – Accessed: 2018-08-18

[Apa18c] Apache Software Foundation: *Beam*. https://beam.apache.org/. version: 2018. – Accessed: 2018-05-10

[Apa18d] Apache Software Foundation: *Calcite*. https://calcite.apache.org/. version: 2018. – Accessed: 2018-05-10

[Apa18e] Apache Software Foundation: *CouchDB*. `http://couchdb.apache.org/`. version: 2018. – Accessed: 2018-05-10

[Apa18f] Apache Software Foundation: *GitHub: Apache Apex Core*. `https://github.com/apache/apex-core`. version: 2018. – Accessed: 2018-08-18

[Apa18g] Apache Software Foundation (ed.): *Guaranteeing Message Processing*. Apache Software Foundation, 2018. `http://storm.apache.org/releases/1.2.2/Guaranteeing-message-processing.html`. – Accessed: 2018-08-19

[Apa18h] Apache Software Foundation: *Qpid*. `https://qpid.apache.org/`. version: 2018. – Accessed: 2018-05-10

[Ara13] Arango, Mauricio: Mobile QoS Management Using Complex Event Processing. In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. New York, NY, USA : ACM, 2013 (DEBS '13). – ISBN 978–1–4503–1758–0, 115–122

[ASC$^+$09] Agrawal, Parag ; Silberstein, Adam ; Cooper, Brian F. ; Srivastava, Utkarsh ; Ramakrishnan, Raghu: Asynchronous View Maintenance for VLSD Databases. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2009 (SIGMOD '09). – ISBN 978–1–60558–551–2, 179–192

[Azu18] Azul Systems, Inc.: *Zing: Java for the Real Time Business*. `https://www.azul.com/products/zing/whatisit/`. version: 2018. – Accessed: 2018-05-10

[Bad09] *Chapter* Extensions. In: Badia, Antonio: *Quantifiers in Action: Generalized Quantification in Query, Logical and Natural Languages*. Boston, MA : Springer US, 2009. – ISBN 978–0–387–09564–6, 127–147

[Ban11] Banon, Shay: Percolator. In: *Elastic Blog* (2011), February. `https://www.elastic.co/blog/percolator`. – Accessed: 2017-11-17

[Baq18a] Baqend: *Baqend*. `https://www.baqend.com/`. version: 2018. – Accessed: 2018-05-10

[Baq18b] Baqend: *Baqend JavaScript SDK and CLI for High-Performance Websites*. `https://github.com/Baqend/js-sdk`. version: 2018. – Accessed: 2018-05-10

[Baq18c] Baqend: Users, Roles, and Permissions. In: *Baqend Guide* (2018). `https://www.baqend.com/guide/topics/user-management/`. – Accessed: 2018-05-10

[Bar13] Barr, Jeff: Amazon ElastiCache – Now With a Dash of Redis. In: *AWS News Blog* (2013), September. `https://aws.amazon.com/blogs/aws/amazon-elasticache-now-with-a-dash-of-redis/`. – Accessed: 2018-05-28

[BBC$^+$11] Baker, Jason ; Bond, Chris ; Corbett, James C. ; Furman, J. J. ; Khorlin, Andrey ; Larson, James ; Léon, Jean-Michel ; Li, Yawei ; Lloyd, Alexander ; Yushprakh, Vadim: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: *CIDR* vol. 11, 2011, pp. 223–234

[BBD$^+$02] Babcock, Brian ; Babu, Shivnath ; Datar, Mayur ; Motwani, Rajeev ; Widom, Jennifer: Models and Issues in Data Stream Systems. In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : ACM, 2002 (PODS ’02). – ISBN 1–58113–507–6, 1–16

[BBF$^+$10] Biem, Alain ; Bouillet, Eric ; Feng, Hanhua et al.: IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010. – ISBN 978–1–4503–0032–2

[BC79] Buneman, O. P. ; Clemons, Eric K.: Efficiently Monitoring Relational Databases. In: *ACM Trans. Database Syst.* 4 (1979), September, no. 3, 368–382. http://dx.doi.org/10.1145/320083.320099. – DOI 10.1145/320083.320099. – ISSN 0362–5915

[BCL89] Blakeley, José A. ; Coburn, Neil ; Larson, Per-Ake: Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. In: *ACM Trans. Database Syst.* 14 (1989), September, no. 3, 369–400. http://dx.doi.org/10.1145/68012.68015. – DOI 10.1145/68012.68015. – ISSN 0362–5915

[BD91] Brockwell, Peter J. ; Davis, Richard A.: *Time Series: Theory and Methods*. 2nd Edition. Springer Science & Business Media, 1991

[BD15] Bruns, Ralf ; Dunkel, Jürgen: *Complex Event Processing: Komplexe Analyse von massiven Datenströmen mit CEP*. Springer Vieweg, 2015

[BDM07] Babcock, Brian ; Datar, Mayur ; Motwani, Rajeev: Load Shedding in Data Stream Systems. In: *Data Streams – Models and Algorithms* vol. 31. Springer, 2007, pp. 127–147

[BG13] Bailis, Peter ; Ghodsi, Ali: Eventual Consistency Today: Limitations, Extensions, and Beyond. In: *Queue* 11 (2013), March, no. 3, 20:20–20:32. http://dx.doi.org/10.1145/2460276.2462076. – DOI 10.1145/2460276.2462076. – ISSN 1542–7730

[BKS00] Bellatreche, Ladjel ; Karlapalem, Kamalakar ; Schneider, Michel: On Efficient Storage Space Distribution Among Materialized Views and Indices in Data Warehousing Environments. In: *Proceedings of the Ninth International Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2000 (CIKM ’00). – ISBN 1–58113–320–0, 397–404

[BL10] Buccafurri, Francesco ; Lax, Gianluca: Approximating Sliding Windows by Cyclic Tree-like Histograms for Efficient Range Queries. In: *Data Knowl. Eng.* 69 (2010), September, no. 9, 979–997. http://dx.doi.org/10.1016/j.datak.2010.05.002. – DOI 10.1016/j.datak.2010.05.002. – ISSN 0169–023X

[Blo70] Bloom, Burton H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. In: *Commun. ACM* 13 (1970), July, no. 7, 422–426. http://dx.doi.org/10.1145/362686.362692. – DOI 10.1145/362686.362692. – ISSN 0001–0782

[BLT86] Blakeley, José A. ; Larson, Per-Ake ; Tompa, Frank W.: Efficiently Updating Materialized Views. In: *SIGMOD Rec.* 15 (1986), June, no. 2, 61–71. http://dx.doi.org/10.1145/16856.16861. – DOI 10.1145/16856.16861. – ISSN 0163–5808

[BM90] Blakeley, José A. ; Martin, Nancy L.: Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis. In: *Proceedings of the Sixth International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 1990. – ISBN 0–8186–2025–0, 256–263

[BM+17] Burk, Nikolas ; Marktanner, Nilan et al.: GraphQL vs. Firebase. In: *Graphcool Docs* (2017). https://www.graph.cool/docs/tutorials/graphql-vs-firebase-chi6oozus1/. – Accessed: 2017-07-18

[BMH+16] Blitz, Craig ; Middleton, Tim ; Howes, Jason ; Falco, Mark ; Raja, Harvey ; Stafford, Randy ; Purdy, Jon ; Peralta, Patrick: Oracle Coherence 12c: Planning a Successful Deployment / Oracle Corporation. 2016. – technical report

[BMP+06] Bonomi, Flavio ; Mitzenmacher, Michael ; Panigrahy, Rina ; Singh, Sushil ; Varghese, George: An improved construction for counting bloom filters. In: *Algorithms–ESA 2006*. Springer, 2006, pp. 684–695

[BMS17] Burk, Nikolas ; Marktanner, Nilan ; Schickling, Johannes: How to build a Real-Time Chat with GraphQL Subscriptions and Apollo? In: *Graphcool Docs* (2017). https://www.graph.cool/docs/tutorials/worldchat-subscriptions-example-ui0eizishe/. – Accessed: 2017-07-18

[Bov17] Bover, Pier: Firebase: the great, the meh, and the ugly. In: *freeCodeCamp Blog* (2017), January. https://medium.freecodecamp.com/firebase-the-great-the-meh-and-the-ugly-a07252fbcf15. – Accessed: 2017-05-21

[Bro15] Brown, Cole: Introducing Concord. In: *Concord Blog* (2015), November. http://concord.io/posts/introducing_concord. – Accessed: 2016-09-21

[BROL14] Boykin, Oscar ; Ritchie, Sam ; O'Connell, Ian ; Lin, Jimmy: Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. In: *Proc. VLDB Endow.* 7 (2014), August, no. 13, 1441–1451. http://dx.doi.org/10.14778/2733004.2733016. – DOI 10.14778/2733004.2733016. – ISSN 2150–8097

[BSW04] Babu, Shivnath ; Srivastava, Utkarsh ; Widom, Jennifer: Exploiting K-constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. In: *ACM Trans. Database Syst.* 29 (2004), September, no. 3, 545–580. http://dx.doi.org/10.1145/1016028.1016032. – DOI 10.1145/1016028.1016032. – ISSN 0362–5915

[BVF$^+$12] Bailis, Peter ; Venkataraman, Shivaram ; Franklin, Michael J. ; Hellerstein, Joseph M. ; Stoica, Ion: Probabilistically bounded staleness for practical partial quorums. In: *Proceedings of the VLDB Endowment* 5 (2012), no. 8, 776–787. http://dl.acm.org/citation.cfm?id=2212359

[ÇAA$^+$16] Çetintemel, Uğur ; Abadi, Daniel ; Ahmad, Yanif ; Balakrishnan, Hari ; Balazinska, Magdalena ; Cherniack, Mitch ; Hwang, Jeong-Hyon ; Madden, Samuel ; Maskey, Anurag ; Rasin, Alexander ; Ryvkina, Esther ; Stonebraker, Mike ; Tatbul, Nesime ; Xing, Ying ; Zdonik, Stan: The Aurora and Borealis Stream Processing Engines. In: Garofalakis, Minos (ed.) ; Gehrke, Johannes (ed.) ; Rastogi, Rajeev (ed.): *Data Stream Management: Processing High-Speed Data Streams*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2016. – ISBN 978–3–540–28608–0, pp. 337–359

[CAB$^+$81] Chamberlin, Donald D. ; Astrahan, Morton M. ; Blasgen, Michael W. ; Gray, James N. ; King, W. F. ; Lindsay, Bruce G. ; Lorie, Raymond ; Mehl, James W. ; Price, Thomas G. ; Putzolu, Franco ; Selinger, Patricia G. ; Schkolnick, Mario ; Slutz, Donald R. ; Traiger, Irving L. ; Wade, Bradford W. ; Yost, Robert A.: A History and Evaluation of System R. In: *Commun. ACM* 24 (1981), October, no. 10, 632–646. http://dx.doi.org/10.1145/358769.358784. – DOI 10.1145/358769.358784. – ISSN 0001–0782

[Cal17] Calcite (ed.): *Streaming*. Calcite, November 2017. https://calcite.apache.org/docs/stream.html. – Accessed: 2017-11-26

[Can18] *Can I use WebSockets?* https://caniuse.com/#feat=websockets. version: May 2018. – Accessed: 2018-05-21

[CB74] Chamberlin, Donald D. ; Boyce, Raymond F.: SEQUEL: A Structured English Query Language. In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. New York, NY, USA : ACM, 1974 (SIGFIDET '74), 249–264

[CBHB09] Chaves, Leonardo Weiss F. ; Buchmann, Erik ; Hueske, Fabian ; Böhm, Klemens: Towards Materialized View Selection for Distributed Databases. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. New York, NY, USA : ACM, 2009 (EDBT '09). – ISBN 978–1–60558–422–5, 1088–1099

[CCC+02] Carney, Don ; Cetintemel, Uğur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Seidman, Greg ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stan: Monitoring Streams: A New Class of Data Management Applications. In: *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB Endowment, 2002 (VLDB '02), 215–226

[CCD+03] Chandrasekaran, Sirish ; Cooper, Owen ; Deshpande, Amol ; Franklin, Michael J. ; Hellerstein, Joseph M. ; Hong, Wei ; Krishnamurthy, Sailesh ; Madden, Samuel R. ; Reiss, Fred ; Shah, Mehul A.: TelegraphCQ: Continuous Dataflow Processing. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2003 (SIGMOD '03). – ISBN 1–58113–634–X, 668–668

[CDE+13] Corbett, James C. ; Dean, Jeffrey ; Epstein, Michael ; Fikes, Andrew ; Frost, Christopher ; Furman, J. J. ; Ghemawat, Sanjay ; Gubarev, Andrey ; Heiser, Christopher ; Hochschild, Peter ; Hsieh, Wilson ; Kanthak, Sebastian ; Kogan, Eugene ; Li, Hongyi ; Lloyd, Alexander ; Melnik, Sergey ; Mwaura, David ; Nagle, David ; Quinlan, Sean ; Rao, Rajesh ; Rolig, Lindsay ; Saito, Yasushi ; Szymaniak, Michal ; Taylor, Christopher ; Wang, Ruth ; Woodford, Dale: Spanner: Google's Globally Distributed Database. In: *ACM Trans. Comput. Syst.* 31 (2013), August, no. 3, pp. 8:1–8:22. `http://dx.doi.org/10.1145/2491245`. – DOI 10.1145/2491245. – ISSN 0734–2071

[CDE+15] Chintapalli, Sanket ; Dagit, Derek ; Evans, Bobby ; Farivar, Reza ; Graves, Tom ; Holderbaugh, Mark ; Liu, Zhuo ; Nusbaum, Kyle ; Patil, Kishorkumar ; Peng, Boyang J. ; Poulosky, Paul: Benchmarking Streaming Computation Engines at Yahoo! In: *Yahoo! Engineering Blog* (2015), December. `http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at`. – Accessed: 2016-10-17

[CDG+06] Chang, Fay ; Dean, Jeffrey ; Ghemawat, Sanjay ; Hsieh, Wilson C. ; Wallach, Deborah A. ; Burrows, Mike ; Chandra, Tushar ; Fikes, Andrew ; Gruber, Robert E.: Bigtable: A Distributed Storage System for Structured Data. In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. Berkeley, CA, USA : USENIX Association, 2006 (OSDI '06), 15–15

[CDK+14] Cetintemel, Ugur ; Du, Jiang ; Kraska, Tim ; Madden, Samuel ; Maier, David ; Meehan, John ; Pavlo, Andrew ; Stonebraker, Michael ; Sutherland, Erik ; Tatbul, Nesime ; Tufte, Kristin ; Wang, Hao ; Zdonik, Stanley: S-Store: A Streaming NewSQL System for Big Velocity Applications. In: *Proc. VLDB Endow.* 7 (2014), August, no. 13, 1633–1636. http://dx.doi.org/10.14778/2733004.2733048. – DOI 10.14778/2733004.2733048. – ISSN 2150–8097

[CDTW00] Chen, Jianjun ; DeWitt, David J. ; Tian, Feng ; Wang, Yuan: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2000 (SIGMOD '00). – ISBN 1–58113–217–4, 379–390

[CFG+16] Chandramouli, Badrish ; Fernandez, Raul C. ; Goldstein, Jonathan ; Eldawy, Ahmed ; Quamar, Abdul: Quill: Efficient, Transferable, and Rich Analytics at Scale. In: *International Conference on Very Large Databases (PVLDB Vol. 9, Issue. 14)*, 2016

[CGB+14] Chandramouli, Badrish ; Goldstein, Jonathan ; Barnett, Mike ; DeLine, Robert ; Fisher, Danyel ; Platt, John C. ; Terwilliger, James F. ; Wernsing, John: Trill: A High-performance Incremental Query Processor for Diverse Analytics. In: *Proc. VLDB Endow.* 8 (2014), December, no. 4, 401–412. http://dx.doi.org/10.14778/2735496.2735503. – DOI 10.14778/2735496.2735503. – ISSN 2150–8097

[CGH+17] Carbone, Paris ; Gévay, Gábor E. ; Hermann, Gábor ; Katsifodimos, Asterios ; Soto, Juan ; Markl, Volker ; Haridi, Seif: Large-Scale Data Stream Processing Systems. In: *Handbook of Big Data Technologies*. Springer, 2017, pp. 219–260

[Cha95] Chakravarthy, Sharma: Early Active Database Efforts: A Capsule Summary. In: *IEEE Trans. on Knowl. and Data Eng.* 7 (1995), December, no. 6, 1008–1010. http://dx.doi.org/10.1109/69.476505. – DOI 10.1109/69.476505. – ISSN 1041–4347

[Che75] Chen, Peter Pin-Shan: The Entity-relationship Model: Toward a Unified View of Data. In: *SIGIR Forum* 10 (1975), December, no. 3, 9–9. http://dx.doi.org/10.1145/1095277.1095279. – DOI 10.1145/1095277.1095279. – ISSN 0163–5840

[CL85] Chandy, K. M. ; Lamport, Leslie: Distributed Snapshots: Determining Global States of Distributed Systems. In: *ACM Trans. Comput. Syst.* 3 (1985), February, no. 1, 63–75. http://dx.doi.org/10.1145/214451.214456. – DOI 10.1145/214451.214456. – ISSN 0734–2071

[Clo18a] Cloud Native Computing Foundation (ed.): *Does NATS guarantee message delivery?* Cloud Native Computing Foundation, 2018. `https://nats.io/documentation/faq/#gmd`. – Accessed: 2018-05-28

[Clo18b] Cloud Native Computing Foundation: *NATS*. `https://nats.io/`. version: 2018. – Accessed: 2018-05-28

[CM05] Cormode, Graham ; Muthukrishnan, S.: An Improved Data Stream Summary: The Count-min Sketch and Its Applications. In: *J. Algorithms* 55 (2005), April, no. 1, 58–75. `http://dx.doi.org/10.1016/j.jalgor.2003.12.001`. – DOI 10.1016/j.jalgor.2003.12.001. – ISSN 0196–6774

[CM12] Cugola, Gianpaolo ; Margara, Alessandro: Processing Flows of Information: From Data Stream to Complex Event Processing. In: *ACM Comput. Surv.* 44 (2012), June, no. 3, 15:1–15:62. `http://dx.doi.org/10.1145/2187671.2187677`. – DOI 10.1145/2187671.2187677. – ISSN 0360–0300

[CoC18] CoCalc by SageMath, Inc.: *CoCalc*. `https://cocalc.com/`. version: 2018. – Accessed: 2018-05-10

[Cod70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. In: *Commun. ACM* 13 (1970), June, no. 6, 377–387. `http://dx.doi.org/10.1145/362384.362685`. – DOI 10.1145/362384.362685. – ISSN 0001–0782

[Cod71] Codd, E. F.: A Data Base Sublanguage Founded on the Relational Calculus. In: *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. New York, NY, USA : ACM, 1971 (SIGFIDET '71), 35–68

[Cod82] Codd, E. F.: Relational Database: A Practical Foundation for Productivity. In: *Commun. ACM* 25 (1982), February, no. 2, 109–117. `http://dx.doi.org/10.1145/358396.358400`. – DOI 10.1145/358396.358400. – ISSN 0001–0782

[Col16] Coleman, Tom: The Oplog Observe Driver. In: *Meteor Documentation* (2016). `https://github.com/meteor/docs/blob/cc3f8fe99b3db72c21ea2c0e8d7e574bca860ec6/long-form/oplog-observe-driver.md`. – Accessed: 2017-10-16

[Con18] Condon, Craig: *sift.js*. `https://github.com/crcn/sift.js`. version: 2018. – Accessed: 2018-06-24

[Cor17] Cormode, Graham: Data Sketching. In: *Commun. ACM* 60 (2017), August, no. 9, 48–55. `http://dx.doi.org/10.1145/3080008`. – DOI 10.1145/3080008. – ISSN 0001–0782

[CPM96] Cochrane, Roberta ; Pirahesh, Hamid ; Mattos, Nelson M.: Integrating Triggers and Declarative Constraints in SQL Database Sytems. In: *Proceedings of the 22th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1996 (VLDB '96). – ISBN 1–55860–382–4, 567–578

[CRP+10] Chambers, Craig ; Raniwala, Ashish ; Perry, Frances ; Adams, Stephen ; Henry, Robert ; Bradshaw, Robert ; Nathan: FlumeJava: Easy, Efficient Data-Parallel Pipelines. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010, 363-375

[CRW01] Carzaniga, Antonio ; Rosenblum, David S. ; Wolf, Alexander L.: Design and Evaluation of a Wide-area Event Notification Service. In: *ACM Trans. Comput. Syst.* 19 (2001), August, no. 3, 332–383. `http://dx.doi.org/10.1145/380749.380767`. – DOI 10.1145/380749.380767. – ISSN 0734–2071

[CTE15] Celebi, Ufuk ; Tzoumas, Kostas ; Ewen, Stephan: How Apache Flink™handles backpressure. In: *data Artisans Blog* (2015), August. `http://data-artisans.com/how-flink-handles-backpressure/`. – Accessed: 2017-09-12

[CTW05] Click, Cliff ; Tene, Gil ; Wolf, Michael: The Pauseless GC Algorithm. In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. New York, NY, USA : ACM, 2005 (VEE '05). – ISBN 1–59593–047–7, 46–56

[CVZ13] Carbone, P. ; Vandikas, K. ; Zaloshnja, F.: Towards Highly Available Complex Event Processing Deployments in the Cloud. In: *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, 2013. – ISSN 2161–2889, pp. 153–158

[CWI+16] Chen, Guoqiang J. ; Wiener, Janet L. ; Iyer, Shridhar ; Jaiswal, Anshul ; Lei, Ran ; Simha, Nikhil ; Wang, Wei ; Wilfong, Kevin ; Williamson, Tim ; Yilmaz, Serhat: Realtime Data Processing at Facebook. In: *Proceedings of the 2016 International Conference on Management of Data*. New York, NY, USA : ACM, 2016 (SIGMOD '16). – ISBN 978–1–4503–3531–7, 1087–1098

[CY12] Chirkova, Rada ; Yang, Jun: Materialized Views. In: *Foundations and Trends in Databases* 4 (2012), no. 4, 295-405. `http://dx.doi.org/10.1561/1900000020`. – DOI 10.1561/1900000020. – ISSN 1931–7883

[DA09] Dhote, C.A. ; Ali, M.S.: Materialized View Selection in Data Warehousing: A Survey. In: *Materialized View Selection in Data Warehousing: A Survey* 9 (2009), no. 3, pp. 401–414

[Dan17] DANIELS, Eric: MongoDB Stitch – Backend as a Service (commentary). In: *Hacker News* (2017). https://news.ycombinator.com/item?id=14595456. – Accessed 2017-11-17

[Dat18] DATABRICKS INC.: Resilient Distributed Dataset (RDD). In: *Databricks Glossary* (2018). https://databricks.com/glossary/what-is-rdd. – Accessed: 2018-07-22

[Dav17a] DAVIS, A. Jesse J.: New Driver Features for MongoDB 3.6 (commentary). In: *emptysquare Blog* (2017), October. https://emptysqua.re/blog/driver-features-for-mongodb-3-6/#comment-3574381334. – Accessed: 2017-11-17

[Dav17b] DAVIS, A. Jesse J.: New Driver Features for MongoDB 3.6: Notification API. In: *emptysquare Blog* (2017), June. https://emptysqua.re/blog/driver-features-for-mongodb-3-6/. – Accessed: 2018-04-23

[DBB+88] DAYAL, U. ; BLAUSTEIN, B. ; BUCHMANN, A. ; CHAKRAVARTHY, U. ; HSU, M. ; LEDIN, R. ; MCCARTHY, D. ; ROSENTHAL, A. ; SARIN, S. ; CAREY, M. J. ; LIVNY, M. ; JAUHARI, R.: The HiPAC Project: Combining Active Databases and Timing Constraints. In: *SIGMOD Rec.* 17 (1988), March, no. 1, 51–70. http://dx.doi.org/10.1145/44203.44208. – DOI 10.1145/44203.44208. – ISSN 0163–5808

[DBS+12] DAS, Shirshanka ; BOTEV, Chavdar ; SURLAKER, Kapil ; GHOSH, Bhaskar ; VARADARAJAN, Balaji ; NAGARAJ, Sunil ; ZHANG, David ; GAO, Lei ; WESTERMAN, Jemiah ; GANTI, Phanindra ; SHKOLNIK, Boris ; TOPIWALA, Sajid ; PACHEV, Alexander ; SOMASUNDARAM, Naveen ; SUBRAMANIAM, Subbu: All Aboard the Databus!: Linkedin's Scalable Consistent Change Data Capture Platform. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. New York, NY, USA : ACM, 2012 (SoCC '12). – ISBN 978–1–4503–1761–0, 18:1–18:14

[Del15] DELL'AQUILA, Luigi: LiveQuery. In: *OrientDB Blog* (2015), October. http://orientdb.com/livequery/. – Accessed: 2017-07-09

[DF14] DUNNING, Ted ; FRIEDMAN, Ellen ; LOUKIDES, Mike (ed.): *Time Series Databases: New Ways to Store and Access Data*. O'Reilly Media, 2014

[DG04] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. Berkeley, CA, USA : USENIX Association, 2004 (OSDI'04), 10–10

[DGIM02] DATAR, Mayur ; GIONIS, Aristides ; INDYK, Piotr ; MOTWANI, Rajeev: Maintaining Stream Statistics over Sliding Windows. In: *SIAM Journal on Computing* 31 (2002), no. 6, pp. 1794–1813

[DHJ+07] DeCandia, G. ; Hastorun, D. ; Jampani, M. ; Kakulapati, G. ; Lakshman, A. ; Pilchin, A. ; Sivasubramanian, S. ; Vosshall, P. ; Vogels, W.: Dynamo: Amazon's highly available key-value store. In: *ACM SOSP* vol. 14, 2007 (17), pp. 205–220

[DJPQ94] Díaz, Oscar ; Jaime, Arturo ; Paton, Norman W. ; Qaimari, Ghassan al: Supporting Dynamic Displays Using Active Rules. In: *SIGMOD Rec.* 23 (1994), March, no. 1, 21–26. `http://dx.doi.org/10.1145/181550.181555`. – DOI 10.1145/181550.181555. – ISSN 0163–5808

[DLOM02] Demaine, Erik D. ; López-Ortiz, Alejandro ; Munro, J. I.: Frequency Estimation of Internet Packet Streams with Limited Space. In: *Proceedings of the 10th Annual European Symposium on Algorithms*. London, UK, UK : Springer-Verlag, 2002 (ESA '02). – ISBN 3–540–44180–8, 348–360

[Dro17] Drobik, David: Will Google Build Your Product? In: *David Drobik – Blog* (2017), October. `https://medium.com/@daviddrobik/will-google-build-your-product-56508d19524a`. – Accessed: 2017-12-23

[DST15] Da Silva, Maxwell D. ; Tavares, Hugo L.: *Redis Essentials*. Packt Publishing, 2015. – ISBN 1784392456, 9781784392451

[Duf17] Dufetel, Alex: Introducing Cloud Firestore: Our New Document Database for Apps. In: *Firebase Blog* (2017), October. `https://firebase.googleblog.com/2017/10/introducing-cloud-firestore.html`. – Accessed: 2017-12-19

[EC75] Eswaran, Kapali P. ; Chamberlin, Donald D.: Functional Specifications of a Subsystem for Data Base Integrity. In: *Proceedings of the 1st International Conference on Very Large Data Bases*. New York, NY, USA : ACM, 1975 (VLDB '75). – ISBN 978–1–4503–3920–9, 48–68

[ECM17] ECMA International: *ECMAScript 2017 Language Specification*. 8th Edition, June 2017

[Ela18] Elasticsearch: *Elasticsearch*. `https://www.elastic.co/products/elasticsearch/`. version: 2018. – Accessed: 2018-05-10

[Elk90] Elkan, Charles: Independence of Logic Database Queries and Update. In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : ACM, 1990 (PODS '90). – ISBN 0–89791–352–3, 154–160

[ENL11] Etzion, Opher ; Niblett, Peter ; Luckham, David C. ; Stirling, Sebastian (ed.): *Event processing in action*. Manning Greenwich, 2011

[Eri98] Eriksson, Joakim: Real-Time and Active Databases: A Survey. In: Andler, Sten F. (ed.) ; Hansson, Jörgen (ed.): *Active, Real-Time, and Temporal*

Database Systems: Second International Workshop, ARTDB-97 Como, Italy, September 8–9, 1997 Proceedings. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. – ISBN 978–3–540–49151–4, 1–23

[Eri14] Ericsson: Trident – benchmarking performance. In: *Ericsson Research Blog* (2014). http://www.ericsson.com/research-blog/data-knowledge/trident-benchmarking-performance/. – Accessed: 2016-01-12

[Esp16] EsperTech (ed.): *How does Esper scale?* EsperTech, 2016. http://www.espertech.com/esper/faq_esper.php#scaling. – Accessed: 2016-09-19

[Ewe16] Ewen, Stephan: FLIP-6- Flink Deployment and Process Model- Standalone, Yarn, Mesos, Kubernetes, etc. In: *Flink Improvement Proposals* (2016), August. https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=65147077. – Accessed: 2017-11-17

[Feg16] Fegaras, Leonidas: Incremental Query Processing on Big Data Streams. In: *IEEE Trans. on Knowl. and Data Eng.* 28 (2016), November, no. 11, 2998–3012. http://dx.doi.org/10.1109/TKDE.2016.2601103. – DOI 10.1109/TKDE.2016.2601103. – ISSN 1041–4347

[Fie00] Fielding, Roy T.: *REST: Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Doctoral dissertation, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[Fir16] Firebase: *Firebase*. https://firebase.google.com/. version: 2016. – Accessed: 2016-09-18

[Fir17a] Firebase: Best practices for data structure: Avoid nesting data. In: *Firebase Docs* (2017). https://firebase.google.com/docs/database/web/structure-data#avoid_nesting_data. – Accessed: 2017-05-21

[Fir17b] Firebase (ed.): *Choose a Database: Cloud Firestore or Realtime Database*. Firebase, December 2017. https://firebase.google.com/docs/firestore/rtdb-vs-firestore. – Accessed: 2017-12-19

[Fir17c] Firebase (ed.): *Firestore: Quotas and Limits*. Firebase, December 2017. https://firebase.google.com/docs/firestore/quotas. – Accessed: 2017-12-19

[Fir17d] Firebase (ed.): *Order and Limit Data with Cloud Firestore*. Firebase, December 2017. https://firebase.google.com/docs/firestore/query-data/order-limit-data. – Accessed: 2017-12-19

[Fir17e] Firebase (ed.): *Perform Simple and Compound Queries in Cloud Firestore*. Firebase, December 2017. https://firebase.google.com/docs/firestore/query-data/queries. –

Accessed: 2017-12-19

[Fir17f] FIREBASE (ed.): *Realtime Database Limits*. Firebase, November 2017. https://firebase.google.com/docs/database/usage/limits. – Accessed: 2017-11-17

[FKSV03] FEIGENBAUM, Joan ; KANNAN, Sampath ; STRAUSS, Martin J. ; VISWANATHAN, Mahesh: An Approximate L1-Difference Algorithm for Massive Data Streams. In: *SIAM J. Comput.* 32 (2003), January, no. 1, 131–151. http://dx.doi.org/10.1137/S0097539799361701. – DOI 10.1137/S0097539799361701. – ISSN 0097–5397

[FR11] FICCO, M. ; ROMANO, L.: A Generic Intrusion Detection and Diagnoser System Based on Complex Event Processing. In: *2011 First International Conference on Data Compression, Communications and Processing*, 2011, pp. 275–284

[FS76] FRY, James P. ; SIBLEY, Edgar H.: Evolution of Data-Base Management Systems. In: *ACM Comput. Surv.* 8 (1976), March, no. 1, 7–42. http://dx.doi.org/10.1145/356662.356664. – DOI 10.1145/356662.356664. – ISSN 0360–0300

[GAE06] GHANEM, Thanaa M. ; AREF, Walid G. ; ELMAGARMID, Ahmed K.: Exploiting Predicate-window Semantics over Data Streams. In: *SIGMOD Rec.* 35 (2006), March, no. 1, 3–8. http://dx.doi.org/10.1145/1121995.1121996. – DOI 10.1145/1121995.1121996. – ISSN 0163–5808

[GB10] GESSERT, Felix ; BÜCKLERS, Florian: *Performanz- und Reaktivitätssteigerung von OODBMS vermittels der Web-Caching-Hierarchie unter Einsatz und Adaption offener Standards*. University of Hamburg, bachelor's thesis, 2010

[GB12] GESSERT, Felix ; BÜCKLERS, Florian: *Kohärentes Web-Caching von Datenbankobjekten im Cloud Computing*, University of Hamburg, master's thesis, 2012

[GBR14] GESSERT, Felix ; BÜCKLERS, Florian ; RITTER, Norbert: Orestes: A scalable Database-as-a-Service architecture for low latency. In: *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, 2014, pp. 215–222

[GC15] GESSERT, Felix ; CARLSON, Josiah: Proposal for scalable PubSub in Redis Cluster. In: *Redis-dev Mailing List* (2015), July. https://groups.google.com/d/msg/redis-dev/GaCzMEOQ1b4/PaYcZ5whYy4J. – Accessed: 2018-06-09

[GCS+15] GESSERT, Felix ; CARLSON, Josiah ; SANFILIPPO, Salvatore et al.: Redis Cluster Pub/Sub – Scalability Issues. In: *Redis GitHub Issues* (2015), July. https://github.com/antirez/redis/issues/2672. – Accessed: 2018-06-09

[GD94] Geppert, Andreas ; Dittrich, Klaus R.: Rule-Based Implementation of Transaction Model Specifications. In: Paton, Norman W. (ed.) ; Williams, M. H. (ed.): *Rules in Database Systems*. London : Springer London, 1994. – ISBN 978–1–4471–3225–7, pp. 127–142

[Ges17] Gessert, Felix: The AWS and MongoDB Infrastructure of Parse: Lessons Learned. In: *Baqend Tech Blog* (2017), January. `https://medium.baqend.com/parse-is-gone-a-few-secrets-about-their-infrastructure-91b3ab2fcf71`. – Accessed: 2017-11-29

[Ges18] Gessert, Felix: *Low Latency for Cloud Data Management*. University of Hamburg, to be published in late 2018. – PhD thesis

[GFW⁺14] Gessert, Felix ; Friedrich, Steffen ; Wingerath, Wolfram ; Schaarschmidt, Michael ; Ritter, Norbert: Towards a Scalable and Unified REST API for Cloud Data Stores. In: *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, 2014, 723–734

[GGD95] Geppert, Andreas ; Gatziu, Stella ; Dittrich, Klaus R.: A designer's benchmark for active database management systems: 007 meets the BEAST. In: Sellis, Timos (ed.): *Rules in Database Systems: Second International Workshop, RIDS '95 Glyfada, Athens, Greece, September 25–27, 1995 Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1995. – ISBN 978–3–540–45137–2, pp. 309–323

[GGL03] Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google File System. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2003 (SOSP '03). – ISBN 1–58113–757–5, 29–43

[GHM⁺07] Ghanem, T. M. ; Hammad, M. A. ; Mokbel, M. F. ; Aref, W. G. ; Elmagarmid, A. K.: Incremental Evaluation of Sliding-Window Queries over Data Streams. In: *IEEE Transactions on Knowledge and Data Engineering* 19 (2007), Jan, no. 1, pp. 57–72. `http://dx.doi.org/10.1109/TKDE.2007.250585`. – DOI 10.1109/TKDE.2007.250585. – ISSN 1041–4347

[Gia12] Giacomelli, Piero ; Shashi, Ankita (ed.): *Hornetq messaging developer's guide*. Packt Publishing Ltd., 2012

[Gib01] Gibbons, Phillip B.: Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001 (VLDB '01). – ISBN 1–55860–804–4, 541–550

[GJSM96] GUPTA, Ashish ; JAGADISH, H. V. ; SINGH MUMICK, Inderpal: Data integration using self-maintainable views. In: APERS, Peter (ed.) ; BOUZEGHOUB, Mokrane (ed.) ; GARDARIN, Georges (ed.): *Advances in Database Technology – EDBT '96: 5th International Conference on Extending Database Technology Avignon, France, March 25–29, 1996 Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1996. – ISBN 978–3–540–49943–5, 140–144

[GK01] GREENWALD, Michael ; KHANNA, Sanjeev: Space-efficient Online Computation of Quantile Summaries. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2001 (SIGMOD '01). – ISBN 1–58113–332–4, 58–66

[GKS01a] GEHRKE, Johannes ; KORN, Flip ; SRIVASTAVA, Divesh: On Computing Correlated Aggregates over Continual Data Streams. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2001 (SIGMOD '01). – ISBN 1–58113–332–4, 13–24

[GKS01b] GUHA, Sudipto ; KOUDAS, Nick ; SHIM, Kyuseok: Data-streams and Histograms. In: *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*. New York, NY, USA : ACM, 2001 (STOC '01). – ISBN 1–58113–349–9, 471–475

[GLS11] GOLAB, Wojciech ; LI, Xiaozhou ; SHAH, Mehul A.: Analyzing consistency properties for fun and profit. In: *ACM PODC*, ACM, 2011, 197–206

[GM98] GIBBONS, Phillip B. ; MATIAS, Yossi: New Sampling-based Summary Statistics for Improving Approximate Query Answers. In: *SIGMOD Rec.* 27 (1998), June, no. 2, 331–342. `http://dx.doi.org/10.1145/276305.276334`. – DOI 10.1145/276305.276334. – ISSN 0163–5808

[GM99] GUPTA, Ashish ; MUMICK, Iderpal S.: *Materialized views: techniques, implementations, and applications*. MIT press, 1999. – ISBN 0–262–57122–6

[GMSS15] GROVER, Mark ; MALASKA, Ted ; SEIDMAN, Jonathan ; SHAPIRA, Gwen: *Hadoop Application Architectures*. Beijing : O'Reilly, 2015 `http://my.safaribooksonline.com/9781491900086`. – ISBN 978–1–4919–0008–6

[Gol06] GOLAB, Lukasz: *Sliding Window Query Processing over Data Streams*, University of Waterloo, diss., August 2006

[Goo16] GOOGLE (ed.): *Google Cloud Dataflow: Resource Quotas*. Google, 2016. `https://cloud.google.com/dataflow/quotas`. – Accessed: 2016-10-17

[Goo18] GOOGLE: *Google Cloud Datastore*. `https://cloud.google.com/datastore/`. version: 2018. – Accessed: 2018-05-10

[Gra18] Graphcool, Inc.: *Graphcool*. https://www.graph.cool/. version: 2018. – Accessed: 2018-05-10

[Gri13] Grigorik, Ilya: *High performance browser networking*. [S.l.] : O'Reilly Media, 2013. – ISBN 1449344763 9781449344764

[Gri17] GridGain Systems Inc. (ed.): *Introducing Apache Ignite™*. GridGain Systems Inc., 2017

[Gro16] Groningen, Martijn van: Elasticsearch Percolator Continues to Evolve. In: *Elastic Blog* (2016), June. https://www.elastic.co/blog/elasticsearch-percolator-continues-to-evolve. – Accessed: 2017-11-17

[GSW+15] Gessert, Felix ; Schaarschmidt, Michael ; Wingerath, Wolfram ; Friedrich, Steffen ; Ritter, Norbert: The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management. In: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, 2015, 53–72

[GSW+17] Gessert, Felix ; Schaarschmidt, Michael ; Wingerath, Wolfram ; Witt, Erik ; Yoneki, Eiko ; Ritter, Norbert: Quaestor: Query Web Caching for Database-as-a-Service Providers. In: *Proceedings of the 43rd International Conference on Very Large Data Bases* (2017)

[GTSS13] Gidra, Lokesh ; Thomas, Gaël ; Sopena, Julien ; Shapiro, Marc: A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In: *SIGARCH Comput. Archit. News* 41 (2013), March, no. 1, 229–240. http://dx.doi.org/10.1145/2490301.2451142. – DOI 10.1145/2490301.2451142. – ISSN 0163–5964

[GWFR16] Gessert, Felix ; Wingerath, Wolfram ; Friedrich, Steffen ; Ritter, Norbert: NoSQL Database Systems: A Survey and Decision Guidance. In: *Computer Science - Research and Development* (2016)

[GZ10] Golab, Lukasz ; Zsu, M. T.: *Data Stream Management*. Morgan & Claypool Publishers, 2010. – ISBN 1608452727, 9781608452729

[H+15] Hickson, Ian et al.: *Server-Sent Events*. W3C Recommendation. https://www.w3.org/TR/eventsource/. version: February 2015. – Accessed: 2018-05-21

[HAMS08] Harizopoulos, Stavros ; Abadi, Daniel J. ; Madden, Samuel ; Stonebraker, Michael: OLTP Through the Looking Glass, and What We Found There. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2008 (SIGMOD '08). – ISBN 978–1–60558–102–6, 981–992

[Haz17] Hazelcast (ed.): *Hazelcast: Continuous Query Cache*. Hazelcast, 2017. `http://docs.hazelcast.org/docs/latest-development/manual/html/Distributed_Query/Continuous_Query_Cache.html`. – Accessed: 2017-11-12

[HG17] Hubbard, Jennifer ; Guyer, Craig: Working with Query Notifications. In: *Microsoft SQL Documentation: Database Features* (2017), March. – Accessed: 2017-05-12

[HHE15] Holmberg, Christer ; Hakansson, Stefan ; Eriksson, Goran: *Web Real-Time Communication Use Cases and Requirements*. RFC 7478. `http://dx.doi.org/10.17487/RFC7478`. version: March 2015 (Request for Comments). – Accessed: 2018-05-21

[HIM02] Hacigumus, Hakan ; Iyer, Bala ; Mehrotra, Sharad: Providing Database As a Service. In: *Proceedings of the 18th International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 2002 (ICDE '02), 29–38

[Hin13] Hintjens, Pieter ; Oram, Andy (ed.) ; Gulick, Maria (ed.): *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013

[HKJR10] Hunt, Patrick ; Konar, Mahadev ; Junqueira, Flavio P. ; Reed, Benjamin: ZooKeeper: Wait-free Coordination for Internet-scale Systems. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2010 (USENIXATC'10)

[HKZ+11] Hindman, Benjamin ; Konwinski, Andy ; Zaharia, Matei ; Ghodsi, Ali ; Joseph, Anthony D. ; Katz, Randy ; Shenker, Scott ; Stoica, Ion: Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2011 (NSDI'11), 295–308

[HR83] Härder, Theo ; Reuter, Andreas: Principles of Transaction-oriented Database Recovery. In: *ACM Comput. Surv.* 15 (1983), December, no. 4, 287–317. `http://dx.doi.org/10.1145/289.291`. – DOI 10.1145/289.291. – ISSN 0360–0300

[HSH07] Hellerstein, Joseph M. ; Stonebraker, Michael ; Hamilton, James: Architecture of a Database System. In: *Found. Trends databases* 1 (2007), February, no. 2, 141–259. `http://dx.doi.org/10.1561/1900000002`. – DOI 10.1561/1900000002. – ISSN 1931–7883

[Hue18] Hueske, Fabian: Apache Flink 1.5.0 Release Announcement. In: *Apache Flink Blog* (2018), May. `https://flink.apache.org/news/2018/05/25/release-1.5.0.html`. –

Accessed: 2018-08-18

[Hus17] HUSAIN, Jafar: *Observables for ECMAScript*. https://tc39.github.io/proposal-observable/. version: 2017. – Accessed: 2017-08-05

[HW93] HANSON, E. ; WIDOM, J.: An Overview of Production Rules in Database Systems / Stanford University. version: June 1993. http://ilpubs.stanford.edu:8090/25/. Stanford InfoLab, June 1993 (1993-18). – Technical Report

[HWJ17] HUESKE, Fabian ; WANG, Shaoxuan ; JIANG, Xiaowei: Continuous Queries on Dynamic Tables. In: *Flink Blog* (2017), April. https://flink.apache.org/news/2017/04/04/dynamic-tables.html. – Accessed: 2017-10-27

[IBM11] IBM (ed.): *Overview of InfoSphere CDC (IBM Infosphere Change Data Capture, Version 6.5.2)*. IBM, 2011. https://www.ibm.com/support/knowledgecenter/en/SSTRGZ_10.1.3/com.ibm.cdcdoc.mcadminguide.doc/concepts/overview_of_cdc.html. – Accessed: 2017-11-12

[IBM14] IBM CORPORATION: Of Streams and Storms / IBM Software Group. 2014. – technical report

[IBM18] IBM (ed.): *Cloudant NoSQL DB Docs: Filter Functions*. IBM, May 2018. https://console.bluemix.net/docs/services/Cloudant/api/design_documents.html#filter-functions. – Accessed: 2018-05-2

[Ign17] Continuous Queries: Continuously obtain real-time query results. In: *Apache Ignite™Docs* (2017). https://apacheignite.readme.io/docs/continuous-queries. – Accessed: 2017-11-12

[iMa18] IMATIX CORPORATION: *ZeroMQ*. http://zeromq.org/. version: 2018. – Accessed: 2018-03-26

[Inf16] INFLUXDATA INC.: *InfluxDB*. https://www.influxdata.com/time-series-platform/influxdb/. version: 2016. – Accessed: 2016-09-18

[Jam16] JAMIN, Baptiste: Reasons Not To Use Firebase. In: *Chris Blog* (2016), September. https://crisp.im/blog/why-you-should-never-use-firebase-realtime-database/. – Accessed: 2017-05-21

[JMS+08] JAIN, Namit ; MISHRA, Shailendra ; SRINIVASAN, Anand ; GEHRKE, Johannes ; WIDOM, Jennifer ; BALAKRISHNAN, Hari ; CETINTEMEL, Uğur ; CHERNIACK, Mitch ; TIBBETTS, Richard ; ZDONIK, Stan: Towards a Streaming SQL Standard. In: *Proc. VLDB Endow.* 1 (2008), August, no. 2, 1379–1390.

http://dx.doi.org/10.14778/1454159.1454179. – DOI 10.14778/1454159.1454179. – ISSN 2150–8097

[JMSS05] JOHNSON, Theodore ; MUTHUKRISHNAN, S. ; SHKAPENYUK, Vladislav ; SPATSCHECK, Oliver: A Heartbeat Mechanism and Its Application in Gigascope. In: *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB Endowment, 2005 (VLDB '05). – ISBN 1–59593–154–6, 1079–1088

[JPNR17] *Chapter* Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In: JUNGHANNS, Martin ; PETERMANN, André ; NEUMANN, Martin ; RAHM, Erhard: *Handbook of Big Data Technologies*. Springer International Publishing, 2017. – ISBN 978–3–319–49340–4, 457–505

[K+15] KATZEN, Jacob et al.: *Oplog tailing too far behind not helping*. https://forums.meteor.com/t/oplog-tailing-too-far-behind-not-helping/2235. version: 2015. – Accessed: 2017-07-09

[KBF+15] KULKARNI, Sanjeev ; BHAGAT, Nikunj ; FU, Maosong ; KEDIGEHALLI, Vikas ; KELLOGG, Christopher ; MITTAL, Sailesh ; PATEL, Jignesh M. ; RAMASAMY, Karthik ; TANEJA, Siddarth: Twitter Heron: Stream Processing at Scale. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2015 (SIGMOD '15). – ISBN 978–1–4503–2758–9, 239–250

[Ker17] KERPELMAN, Todd: Cloud Firestore for Realtime Database Developers. In: *Firebase Blog* (2017), October. https://firebase.googleblog.com/2017/10/cloud-firestore-for-rtdb-developers.html. – Accessed: 2017-12-23

[KKM13] KARANASOS, Konstantinos ; KATSIFODIMOS, Asterios ; MANOLESCU, Ioana: Delta: Scalable Data Dissemination Under Capacity Constraints. In: *Proc. VLDB Endow.* 7 (2013), December, no. 4, 217–228. http://dx.doi.org/10.14778/2732240.2732241. – DOI 10.14778/2732240.2732241. – ISSN 2150–8097

[KKN+08] KALLMAN, Robert ; KIMURA, Hideaki ; NATKINS, Jonathan ; PAVLO, Andrew ; RASIN, Alexander ; ZDONIK, Stanley ; JONES, Evan P. C. ; MADDEN, Samuel ; STONEBRAKER, Michael ; ZHANG, Yang ; HUGG, John ; ABADI, Daniel J.: H-store: A High-performance, Distributed Main Memory Transaction Processing System. In: *Proc. VLDB Endow.* 1 (2008), August, no. 2, 1496–1499. http://dx.doi.org/10.14778/1454159.1454211. – DOI 10.14778/1454159.1454211. – ISSN 2150–8097

[Kle02] KLEINBERG, Jon: Bursty and Hierarchical Structure in Streams. In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM, 2002

(KDD '02). – ISBN 1–58113–567–X, 91–101

[Kle16] Kleppmann, Martin: *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2016

[KNR11] Kreps, Jay ; Narkhede, Neha ; Rao, Jun: Kafka: a Distributed Messaging System for Log Processing. In: *NetDB'11*, 2011

[Kre14a] Kreps, Jay: Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). In: *LinkedIn Engineering Blog* (2014), April. https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines. – Accessed: 2016-10-17

[Kre14b] Kreps, Jay: Questioning the Lambda Architecture. In: *O'Reilly Media* (2014), 7. http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html. – Accessed: 2015-12-17

[Kre14c] Kreps, Jay: Why local state is a fundamental primitive in stream processing. In: *O'Reilly Media* (2014), July. https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing. – Accessed: 2017-11-30

[Kre16] Kreps, Jay: Introducing Kafka Streams: Stream Processing Made Simple. In: *Confluent Blog* (2016), March. http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/. – Accessed: 2016-09-19

[KW17] Kloudas, Kostas ; Ward, Chris: Complex Event Processing with Flink: An Update on the State of Flink CEP. In: *data Artisans Blog* (2017), November. https://data-artisans.com/blog/complex-event-processing-flink-cep-update. – Accessed: 2017-12-26

[L+17] Luck, Greg et al.: Mastering Hazelcast IMDG / Hazelcast. 2017. – technical report

[Lac16] Lacker, Kevin: Moving On. In: *Parse Blog* (2016), January. http://blog.parseplatform.org/announcements/moving-on/. – Accessed: 2017-11-18

[Lan01] Laney, Douglas: 3D Data Management: Controlling Data Volume, Velocity, and Variety / META Group. version: February 2001. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf. 2001. – technical report

[Leh14] Lehenbauer, Michael: Firebase: Now with more querying! In: *Firebase Blog* (2014), November. https://firebase.googleblog.com/2014/11/firebase-now-with-more-querying.html. – Accessed: 2017-12-23

[LG+03] Lorentz, Diana ; Gregoir, Joan et al. ; Oracle (ed.): *Oracle Database SQL Reference, 10g Release 1 (10.1)*. Oracle, December 2003

[LLO+12] Lampkin, Valerie ; Leong, Weng T. ; Olivera, Leonardo ; Rawat, Sweta ; Subrahmanyam, Nagesh ; Xiang, Rong ; Kallas, Gerald ; Krishna, Neeraj ; Fassmann, Stefan ; Keen, Martin et al.: *Building smarter planet solutions with MQTT and IBM WebSphere MQ Telemetry*. IBM Redbooks, 2012

[LLP+12] Lam, Wang ; Liu, Lu ; Prasad, Sts et al.: Muppet: MapReduce-style Processing of Fast Data. In: *VLDB 2012* (2012). `http://dx.doi.org/10.14778/2367502.2367520`. – DOI 10.14778/2367502.2367520. – ISSN 2150–8097

[LLXY04] Lin, X. ; Lu, H. ; Xu, J. ; Yu, J. X.: Continuously maintaining quantile summaries of the most recent N elements over a data stream. In: *Proceedings. 20th International Conference on Data Engineering*, 2004. – ISSN 1063–6382, pp. 362–373

[LMT+05] Li, Jin ; Maier, David ; Tufte, Kristin ; Papadimos, Vassilis ; Tucker, Peter A.: Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2005 (SIGMOD '05). – ISBN 1–59593–060–4, 311–322

[LS93] Levy, Alon Y. ; Sagiv, Yehoshua: Queries Independent of Updates. In: *Proceedings of the 19th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993 (VLDB '93). – ISBN 1–55860–152–X, 171–181

[LVA+15] Lu, Haonan ; Veeraraghavan, Kaushik ; Ajoux, Philippe ; Hunt, Jim ; Song, Yee J. ; Tobagus, Wendy ; Kumar, Sanjeev ; Lloyd, Wyatt: Existential consistency: measuring and understanding consistency at Facebook. In: Miller, Ethan L. (ed.) ; Hand, Steven (ed.): *SOSP*, ACM, 2015, pp. 295–310

[LYC+00] Labio, Wilburt ; Yang, Jun ; Cui, Yingwei ; Garcia-Molina, Hector ; Widom, Jennifer: Performance Issues in Incremental Warehouse Maintenance. In: *Proceedings of the 26th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000 (VLDB '00). – ISBN 1–55860–715–3, 461–472

[LYWL05] Lin, Xuemin ; Yuan, Yidong ; Wang, Wei ; Lu, Hongjun: Stabbing the sky: efficient skyline computation over sliding windows. In: *21st International Conference on Data Engineering (ICDE'05)*, 2005. – ISSN 1063–6382, pp. 502–513

[M+14] Mao, Andrew et al.: *My experience hitting limits on Meteor performance*. `https://groups.google.com/forum/#!topic/meteor-talk/Y547Hh2z39Y`.

version: 2014. – Accessed: 2017-07-09

[MAEA05a] Metwally, Ahmed ; Agrawal, Divyakant ; El Abbadi, Amr: Duplicate Detection in Click Streams. In: *Proceedings of the 14th International Conference on World Wide Web*. New York, NY, USA : ACM, 2005 (WWW '05). – ISBN 1–59593–046–9, 12–21

[MAEA05b] Metwally, Ahmed ; Agrawal, Divyakant ; El Abbadi, Amr: Efficient Computation of Frequent and Top-k Elements in Data Streams. In: *Proceedings of the 10th International Conference on Database Theory*. Berlin, Heidelberg : Springer-Verlag, 2005 (ICDT'05). – ISBN 3–540–24288–0, 978–3–540–24288–8, 398–412

[Mar12] Marz, Nathan: Preview of Storm: The Hadoop of Realtime Processing. In: *BackType Technology Blog* (2012), 5. http://web.archive.org/web/20120509023348/http://tech.backtype.com/preview-of-storm-the-hadoop-of-realtime-processing. – Accessed: 2015-12-17

[Mar14] Marz, Nathan: History of Apache Storm and lessons learned. In: *Thoughts from the Red Planet* (2014), 10. http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html. – Accessed: 2015-12-17

[Mar15] Martin, Watts: Changefeeds in RethinkDB. In: *RethinkDB Docs* (2015). https://rethinkdb.com/docs/changefeeds/javascript/#changefeeds-with-filtering-and-aggregation-queries. – Accessed: 2017-07-09

[Mar16] Martin, Watts: Table joins in RethinkDB. In: *RethinkDB Docs* (2016). https://www.rethinkdb.com/docs/table-joins/. – Accessed: 2017-11-17

[MBP06] Mouratidis, Kyriakos ; Bakiras, Spiridon ; Papadias, Dimitris: Continuous Monitoring of Top-k Queries over Sliding Windows. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2006 (SIGMOD '06). – ISBN 1–59593–434–0, 635–646

[Met15] Meteor Development Group: Livequery. In: *Meteor Change Log v1.0.4* (2015), March. http://docs.meteor.com/changelog.html#livequery-1. – Accessed: 2017-07-09

[Met16] Meteor Development Group: Tuning Meteor Mongo Livedata for Scalability. In: *Meteor Blog* (2016), May. https://blog.meteor.com/tuning-meteor-mongo-livedata-for-scalability-13fe9deb8908. – Accessed: 2017-05-12

[Met18] Meteor Development Group: *Meteor*. https://www.meteor.com/. version: 2018. – Accessed: 2018-05-10

[Mew16] MEWES, Daniel: Scaling, sharding and replication: Running a proxy node. In: *RethinkDB Docs* (2016). https://rethinkdb.com/docs/sharding-and-replication/#running-a-proxy-node. – Accessed: 2017-07-09

[MF11] MELNIKOV, Alexey ; FETTE, Ian: *The WebSocket Protocol*. RFC 6455. http://dx.doi.org/10.17487/RFC6455. version: December 2011 (Request for Comments). – Accessed: 2018-05-21

[MGB17] MACAULEY, Ed ; GUYER, Craig ; BYHAM, Rick: Row Version (Transact-SQL). In: *SQL Server 2017 Documentation* (2017), July. https://docs.microsoft.com/en-us/sql/t-sql/data-types/rowversion-transact-sql?view=sql-server-2017. – Accessed: 2018-05-19

[Mic17a] MICROSOFT (ed.): *SQL Server 2008 R2 Books Online: Creating a Query for Notification*. Microsoft, 2017. https://msdn.microsoft.com/en-us/library/ms181122.aspx. – Accessed: 2017-05-12

[Mic17b] MICROSOFT (ed.): *SQL Server 2008 R2 Books Online: Planning for Notifications*. Microsoft, 2017. https://technet.microsoft.com/en-us/library/ms187528(v=sql.105).aspx#Anchor_1. – Accessed: 2017-05-12

[Mic17c] MICROSOFT (ed.): *SQL Server 2008 R2 Books Online: Query Notification Messages*. Microsoft, 2017. https://msdn.microsoft.com/en-us/library/ms189308(v=sql.105).aspx. – Accessed: 2017-05-13

[Mic17d] MICROSOFT (ed.): *SQL Server 2008 R2 Books Online: Understanding When Query Notifications Occur*. Microsoft, 2017. https://msdn.microsoft.com/en-us/library/ms188323(v=sql.105).aspx. – Accessed: 2017-05-15

[Mic17e] MICROSOFT (ed.): *SQL Server 2008 R2 Books Online: Using Query Notifications*. Microsoft, 2017. https://technet.microsoft.com/en-us/library/ms175110(v=sql.105).aspx. – Accessed: 2017-05-13

[MK+17] MURRAY, Chuck ; KYTE, Tom et al.: Using Continuous Query Notification (CQN). In: *Oracle Database Development Guide, 12c Release 1 (12.1)*. Oracle, May 2017

[MM02] MANKU, Gurmeet S. ; MOTWANI, Rajeev: Approximate Frequency Counts over Data Streams. In: *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB Endowment, 2002 (VLDB ’02), 346–357

[MMI+13] MURRAY, Derek G. ; MCSHERRY, Frank ; ISAACS, Rebecca ; ISARD, Michael ; BARHAM, Paul ; ABADI, Martín: Naiad: A Timely Dataflow System. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2013 (SOSP ’13). – ISBN

978–1–4503–2388–8, 439–455

[Mon17a] MONGODB INC.: MongoDB 3.6.0-rc0 is released. In: *MongoDB Blog* (2017), October. https://www.mongodb.com/blog/post/mongodb-360-rc0-is-released. – Accessed: 2017-11-17

[Mon17b] MONGODB INC. (ed.): *MongoDB CRUD Concepts: Tailable Cursor*. MongoDB Inc., 2017. https://docs.mongodb.com/manual/core/tailable-cursors/. – Accessed: 2017-11-13

[Mon18a] MONGODB INC. (ed.): *Cursor Methods:* `cursor.sort()`. MongoDB Inc., 2018. https://docs.mongodb.com/v3.6/reference/method/cursor.sort/. – Accessed: 2018-09-12

[Mon18b] MONGODB INC. (ed.): *db.collection.findAndModify()*. MongoDB Inc., 2018. https://docs.mongodb.com/v3.6/reference/method/db.collection.findAndModify/. – Accessed: 2018-06-23

[Mon18c] MONGODB INC. (ed.): *Evaluation Query Operators:* `$expr`. MongoDB Inc., 2018. https://docs.mongodb.com/v3.6/reference/operator/query/expr/. – Accessed: 2018-09-08

[Mon18d] MONGODB INC. (ed.): *Evaluation Query Operators:* `$regex`*, Index Use*. MongoDB Inc., 2018. https://docs.mongodb.com/v4.0/reference/operator/query/regex/#index-use. – Accessed: 2018-06-30

[Mon18e] MONGODB INC.: *MongoDB*. https://www.mongodb.com. version: 2018. – Accessed: 2018-05-10

[Mon18f] MONGODB INC.: *The MongoDB Database*. GitHub repository. https://github.com/mongodb/mongo. version: July 2018. – Accessed: 2018-08-07

[Mon18g] MONGODB INC. (ed.): *MongoDB Licensing*. MongoDB Inc., July 2018. https://www.mongodb.com/community/licensing. – Accessed: 2018-08-07

[Mon18h] MONGODB INC.: *MongoDB Stitch*. https://mongodb.com/cloud/stitch. version: 2018. – Accessed: 2018-05-10

[Mor83] MORGENSTERN, Matthew: Active Databases As a Paradigm for Enhanced Computing Environments. In: *Proceedings of the 9th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1983 (VLDB '83). – ISBN 0–934613–15–X, 34–42

[Mor15] MORGAN, Andrew: Joins and Other Aggregation Enhancements Coming in MongoDB 3.2 (Part 1 of 3) – Introduction. In: *MongoDB Blog* (2015), October. https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-1-of-3-introduction. – Accessed: 2018-04-23

[Mul11] MULLANE, Greg S.: NOTIFY vs. Prepared Transactions in Postgres (the Bucardo solution). In: *End Point Blog* (2011), May. https://www.endpoint.com/blog/2011/05/03/notify-vs-prepared-transactions-in. – Accessed: 2017-11-12

[Mul14] MULLANE, Greg S.: Version 5 of Bucardo database replication system. In: *End Point Blog* (2014), June. https://www.endpoint.com/blog/2014/06/23/bucardo-5-multimaster-postgres-released. – Accessed: 2017-11-12

[Mum17] MUMM, John: How Wallaroo Scales Distributed State. In: *Wallaroo Labs Blog* (2017), October. https://blog.wallaroolabs.com/2017/10/how-wallaroo-scales-distributed-state/. – Accessed: 2017-12-29

[MVLL05] MA, Lisha ; VIGLAS, Stratis D. ; LI, Meng ; LI, Qian: Stream Operators for Querying Data Streams. In: FAN, Wenfei (ed.) ; WU, Zhaohui (ed.) ; YANG, Jun (ed.): *Advances in Web-Age Information Management: 6th International Conference, WAIM 2005, Hangzhou, China, October 11 – 13, 2005. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – ISBN 978–3–540–32087–6, 404–415

[MW15] MARZ, Nathan ; WARREN, James: *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1st. Greenwich, CT, USA : Manning Publications Co., 2015. – ISBN 1617290343, 9781617290343

[MWA$^+$03] MOTWANI, Rajeev ; WIDOM, Jennifer ; ARASU, Arvind ; BABCOCK, Brian ; BABU, Shivnath ; DATAR, Mayur ; MANKU, Gurmeet S. ; OLSTON, Chris ; ROSENSTEIN, Justin ; VARMA, Rohit: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In: *CIDR*, 2003

[Nak01] NAKAMURA, Hiroaki: Incremental Computation of Complex Object Queries. In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA : ACM, 2001 (OOPSLA '01). – ISBN 1–58113–335–9, 156–165

[Nar17] NARKHEDE, Neha: Exactly-once Semantics are Possible: Here's How Kafka Does it. In: *Confluent Blog* (2017), June. https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/. – Accessed: 2017-11-18

[Nel17] Nelson, Derek: PipelineDB 0.9.7 – Delta Streams and Towards a PostgreSQL Extension. In: *PipelineDB Blog* (2017), March. https://www.pipelinedb.com/blog/pipelinedb-0-9-7-delta-streams-and-towards-a-postgresql-extension. – Accessed: 2017-11-25

[Nie17] Niehoff, Matthias: Event time processing in Apache Spark and Apache Flink. In: *Codecentric Blog* (2017), April. https://blog.codecentric.de/en/2017/04/event-time-processing-apache-spark-apache-flink/. – Accessed: 2017-11-18

[NLR98] Nica, Anisoara ; Lee, Amy J. ; Rundensteiner, Elke A.: The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In: *Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology*. Berlin, Heidelberg : Springer-Verlag, 1998 (EDBT '98). – ISBN 3–540–64264–1, 359–373

[NMM+99] Nielsen, Henrik F. ; Mogul, Jeffrey ; Masinter, Larry M. ; Fielding, Roy T. ; Gettys, Jim ; Leach, Paul J. ; Berners-Lee, Tim: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. http://dx.doi.org/10.17487/RFC2616. version: June 1999 (Request for Comments). – Accessed: 2018-05-21

[NPP+17] Noghabi, Shadi A. ; Paramasivam, Kartik ; Pan, Yi ; Ramesh, Navina ; Bringhurst, Jon ; Gupta, Indranil ; Campbell, Roy H.: Samza: Stateful Scalable Stream Processing at LinkedIn. In: *Proc. VLDB Endow.* 10 (2017), August, no. 12, 1634–1645. http://dx.doi.org/10.14778/3137765.3137770. – DOI 10.14778/3137765.3137770. – ISSN 2150–8097

[NRNK10] Neumeyer, Leonardo ; Robbins, Bruce ; Nair, Anish ; Kesari, Anand: S4: Distributed Stream Computing Platform. In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. Washington, DC, USA : IEEE Computer Society, 2010 (ICDMW '10). – ISBN 978–0–7695–4257–7, 170–177

[Oll06] Olle, T. W.: Nineteen Sixties History of Data Base Management. In: Impagliazzo, John (ed.): *History of Computing and Education 2 (HCE2): IFIP 19th World Computer Congress, WG 9.7, TC 9: History of Computing, Proceedings of the Second Conference on the History of Computing and Education, August 21–24, 2006, Santiago, Chile*. Boston, MA : Springer US, 2006. – ISBN 978–0–387–34741–7, 67–75

[OM10] O'Brien, James A. ; Marakas, George M.: *Management Information Systems*. 10th Edition. McGraw-Hill/Irwin, 2010

[Ora15a] Oracle (ed.): *Oracle Active Data Guard: Real-Time Data Protection and Availability*. Oracle, October 2015

[Ora15b] ORACLE (ed.): *Oracle GoldenGate 12c: Real-Time Access to Real-Time Information*. Oracle, March 2015

[Ora16] ORACLE (ed.): *Oracle Database Development Guide, 12c Release 1 (12.1)*. Oracle, May 2016

[Ora17a] ORACLE (ed.): *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide: Garbage-First Garbage Collector*. Oracle, 2017. `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html`. – Accessed: 2017-11-18

[Ora17b] ORACLE (ed.): *Oracle Database Readme, 12c Release 1 (12.1)*. Oracle, March 2017

[Ora18a] ORACLE (ed.): *Java Platform, Standard Edition Nashorn User's Guide*. Oracle, 2018. `https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/`. – Accessed: 2018-06-23

[Ora18b] ORACLE (ed.): *Oracle NoSQL Database: Python Driver Getting Started Guide (12.2.4.5)*. Oracle, February 2018

[Ora18c] ORACLE CORPORATION: *Shenandoah GC*. `https://wiki.openjdk.java.net/display/shenandoah/Main`. version: 2018. – Accessed: 2018-05-10

[Ori18] ORIENTDB LTD.: *OrientDB*. `https://orientdb.com/`. version: 2018. – Accessed: 2018-05-10

[Osm12] OSMANI, Adnan: The Observer Pattern. In: *Learning JavaScript Design Patterns* (2012). `https://www.safaribooksonline.com/library/view/learning-javascript-design/9781449334840/ch09s05.html`. – Accessed: 2018-05-10

[Pal13] PALMER, Mark: How To Analyze Sensor Data In Real-Time With CEP. In: *The StreamBase Event Processing Blog* (2013), April. `http://streambase.typepad.com/streambase_stream_process/2013/04/time-windowing.html`. – Accessed: 2018-02-14

[Pan14] PANT, Saurabh: Lap around Azure Redis Cache. In: *Microsoft Azure Blog* (2014). `https://azure.microsoft.com/de-de/blog/lap-around-azure-redis-cache-preview/`. – Accessed: 2018-05-28

[Pan15] PANG, Gene: *Scalable Transactions for Scalable Distributed Database Systems*, EECS Department, University of California, Berkeley, diss., Jun 2015. `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-168.html`

[Pat18] PATZWAHL, Marcel: *Inkrementelle Auswertung geobasierter MongoDB-Anfragen*, University of Hamburg, master's thesis, 2018

[Pau15] Paul, Ryan: Build a realtime liveblog with RethinkDB and PubNub. In: *RethinkDB Blog* (2015), May. https://rethinkdb.com/blog/rethinkdb-pubnub/. – Accessed: 2017-05-20

[PD99] Paton, Norman W. ; Díaz, Oscar: Active Database Systems. In: *ACM Comput. Surv.* 31 (1999), March, no. 1, 63–103. http://dx.doi.org/10.1145/311531.311623. – DOI 10.1145/311531.311623. – ISSN 0360–0300

[Pip15] PipelineDB (ed.): *Two Phase Commits*. PipelineDB, 2015. http://enterprise.pipelinedb.com/docs/two-phase.html#two-phase. – Accessed: 2016-10-17

[Pip17] PipelineDB (ed.): *Streams*. PipelineDB, 2017. http://docs.pipelinedb.com/streams.html. – Accessed: 2017-11-25

[Piv18] Pivotal Software, Inc.: *RabbitMQ*. https://www.rabbitmq.com/. version: 2018. – Accessed: 2018-05-10

[PM16] Patterson, Pat ; Malaska, Ted: Ingest & Stream Processing – What Will You Choose? In: *QCon* (2016), August. https://www.infoq.com/presentations/ingest-stream-processing. – Accessed: 2018-05-25

[Pos17] The PostgreSQL Global Development Group (ed.): *PostgreSQL 9.6 Documentation: Notify*. The PostgreSQL Global Development Group, 2017. https://www.postgresql.org/docs/9.6/static/sql-notify.html. – Accessed: 2017-05-13

[Pos18] PostgreSQL Global Development Group (ed.): *PostgreSQL 9.4.18 Documentation*. PostgreSQL Global Development Group, 2018. https://www.postgresql.org/docs/9.4/static/ddl-system-columns.html. – Accessed: 2018-05-19

[Pro18] Project FiFo: *DalmatinerDB*. https://dalmatiner.io/. version: 2018. – Accessed: 2018-05-10

[PS06] Patroumpas, Kostas ; Sellis, Timos: Window Specification over Data Streams. In: *Proceedings of the 2006 International Conference on Current Trends in Database Technology*. Berlin, Heidelberg : Springer-Verlag, 2006 (EDBT'06). – ISBN 3–540–46788–2, 978–3–540–46788–5, 445–464

[PSS$^+$93] Purimetla, B. ; Sivasankaran, R. ; Stankovic, J. ; Ramamritham, K. ; Towsley, D.: A Study of Distributed Real-Time Active Database Applications. Amherst, MA, USA : University of Massachusetts, 1993. – technical report

[PT05] Pavan, A. ; Tirthapura, Srikanta: Range-Efficient Computation of F" over Massive Data Streams. In: *Proceedings of the 21st International Conference* *on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 2005 (ICDE '05). – ISBN 0–7695–2285–8, 32–43

[Puf16] PUFFELEN, Frank van: Have you met the Realtime Database? In: *Firebase Blog* (2016), July. https://firebase.googleblog.com/2016/07/have-you-met-realtime-database.html. – Accessed: 2017-05-20

[QGMW96] QUASS, Dallan ; GUPTA, Ashish ; MUMICK, Inderpal S. ; WIDOM, Jennifer: Making Views Self-maintainable for Data Warehousing. In: *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*. Washington, DC, USA : IEEE Computer Society, 1996 (DIS '96). – ISBN 0–8186–7475–X, 158–169

[R$^+$11] RUZZI, Joseph et al.: *Oracle Coherence Developer's Guide, Release 3.7.1*, 2011

[R$^+$14a] RUZZI, Joseph et al.: Querying Data In a Cache. In: *Oracle Fusion Middleware: Developing Applications with Oracle Coherence, 12c (12.1.2)*. Oracle, May 2014

[R$^+$14b] RUZZI, Joseph et al.: Using Continuous Query Caching. In: *Oracle Fusion Middleware: Developing Applications with Oracle Coherence, 12c (12.1.2)*. Oracle, May 2014

[Ram15] RAMESH, Navina: Apache Samza, LinkedIn's Framework for Stream Processing. In: *thenewstack.io* (2015), January. http://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing/. – Accessed: 2016-09-21

[Ram16] RAMASAMY, Karthik: Open Sourcing Twitter Heron. In: *Twitter Blog* (2016), May. https://blog.twitter.com/2016/open-sourcing-twitter-heron. – Accessed: 2017-01-15

[Rea17a] REACTIVEX (ed.): *A reactive programming library for JavaScript*. ReactiveX, 2017. http://reactivex.io/rxjs/. – Accessed: 2017-08-05

[Rea17b] REALM: Realm Java 3.0: Collection Notifications, Snapshots and Sorting Across Relationships. In: *Realm Blog* (2017), February. https://news.realm.io/news/realm-java-3-0-collection-notifications/. – Accessed: 2017-07-09

[Rea18a] REACTIVEX: *ReactiveX*. http://reactivex.io/. version: 2018. – Accessed: 2018-05-10

[Rea18b] REALM: *Realm*. https://realm.io/. version: 2018. – Accessed: 2018-05-10

[Ret16] RETHINKDB: *RethinkDB*. https://www.rethinkdb.com/. version: 2016. – Accessed: 2016-09-18

[Ric13] Richardson, Kato: Queries, Part 1: Common SQL Queries Converted for Firebase. In: *Firebase Blog* (2013), October. `https://firebase.googleblog.com/2013/10/queries-part-1-common-sql-queries.html`. – Accessed: 2017-12-23

[Ric14] Richardson, Kato: Queries, Part 2: Advanced Searches with Firebase, made Plug-and-Play Simple. In: *Firebase Blog* (2014), January. `https://firebase.googleblog.com/2014/10/firebase-is-joining-google.html`. – Accessed: 2017-12-23

[Ric15] Richter, Jan: *Garbage Collector Shenandoah: desktop applications*, Masaryk University, master's thesis, 2015

[Ris15] Ristić, Ivan ; Girić-Ristić, Jelena (ed.) ; Rankin, Melinda (ed.): *Bulletproof SSL and TLS*. Feisty Duck, 2015

[RMCZ06] Ryvkina, E. ; Maskey, A. S. ; Cherniack, M. ; Zdonik, S.: Revision Processing in a Stream Processing Engine: A High-Level Design. In: *22nd International Conference on Data Engineering (ICDE'06)*, 2006. – ISSN 1063–6382, pp. 141–141

[Ros11] Rose, Ian T.: *Real-Time Query Systems for Complex Data Sources*, Harvard University Cambridge, Massachusetts, diss., 2011

[Ros16] Rose, Alex: Firebase: The Good, Bad, and the Ugly. In: *Raizlabs Developer Blog* (2016), December. `https://www.raizlabs.com/dev/2016/12/firebase-case-study/`. – Accessed: 2017-05-21

[RRH13] Rüdiger, David ; Roidl, Moritz ; Hompel, Michael ten: Towards Agile and Flexible Air Cargo Processes with Localization Based on RFID and Complex Event Processing. In: Kreowski, Hans-Jörg (ed.) ; Scholz-Reiter, Bernd (ed.) ; Thoben, Klaus-Dieter (ed.): *Dynamics in Logistics: Third International Conference, LDIC 2012 Bremen, Germany, February/March 2012 Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–35966–8, 235–246

[RRM17] Raboy, Nic ; Rajagopal, Priya ; Maxwell, Eric: NDP Episode #19: Mobile Development with Realm. In: *Couchbase Blog* (2017), June. `https://blog.couchbase.com/ndp-episode-19-mobile-development-realm/`. – audio podcast (explanation by Eric Maxwell, starting at 10:57); Accessed: 2017-09-13

[RSS+96] Ramamritham, Krithi ; Sivasankaran, Raju ; Stankovic, John A. ; Towsley, Don T. ; Xiong, Ming: Integrating Temporal, Real-time, an Active Databases. In: *SIGMOD Rec.* 25 (1996), March, no. 1, 8–12. `http://dx.doi.org/10.1145/381854.381868`. – DOI

10.1145/381854.381868. – ISSN 0163–5808

[RxD17a] RXDB (ed.): *Custom Build*. RxDB, 2017. https://pubkey.github.io/rxdb/custom-build.html. – Accessed: 2017-10-16

[RxD17b] RXDB (ed.): *Query Change Detection*. RxDB, 2017. https://pubkey.github.io/rxdb/query-change-detection.html. – Accessed: 2017-10-16

[RxD18] RXDB: *RxDB*. https://github.com/pubkey/rxdb. version: 2018. – Accessed: 2018-05-10

[SAK07] SLEE, Mark ; AGARWAL, Aditya ; KWIATKOWSKI, Marc: Thrift: Scalable Cross-Language Services Implementation / Facebook Inc. version: April 2007. http://thrift.apache.org/static/files/thrift-20070401.pdf. 2007. – technical report. – Accessed: 2018-08-19

[San93] SANTORO, Alexandre: Case Study in Prototyping With Rapide: Shared Memory Multiprocessor System / Stanford University. 1993. – technical report

[San16] SANFILIPPO, Salvatore: The first release candidate of Redis 4.0 is out. In: *Antirez.com* (2016), December. http://antirez.com/news/110. – Accessed: 2018-06-10

[San17] SANFILIPPO, Salvatore: Streams: a new general purpose data structure in Redis. In: *Antirez.com* (2017), October. http://antirez.com/news/114. – Accessed: 2018-05-12

[San18a] SANFILIPPO, Salvatore: *How Fast is Redis?*, 2018. https://redis.io/topics/benchmarks. – Accessed: 2018-05-31

[San18b] SANFILIPPO, Salvatore: *Redis*. https://redis.io/. version: 2018. – Accessed: 2018-05-10

[San18c] SANFILIPPO, Salvatore: *Redis Cluster Specification*, 2018. https://redis.io/topics/cluster-spec. – Accessed: 2018-05-30

[San18d] SANFILIPPO, Salvatore: *Redis Keyspace Notifications*, 2018. https://redis.io/topics/notifications. – Accessed: 2018-06-10

[San18e] SANFILIPPO, Salvatore: *Redis Sentinel Documentation*, 2018. https://redis.io/topics/sentinel. – Accessed: 2018-06-04

[San18f] SANFILIPPO, Salvatore: An update on Redis Streams development. In: *Antirez.com* (2018), January. http://antirez.com/news/116. – Accessed: 2018-05-12

[Sax15] SAX, Matthias J.: Storm Compatibility in Apache Flink: How to run existing Storm topologies on Flink. In: *Apache Flink Blog* (2015), December

[SBLC00] Salem, Kenneth ; Beyer, Kevin ; Lindsay, Bruce ; Cochrane, Roberta: How to Roll a Join: Asynchronous Incremental View Maintenance. In: *SIGMOD Rec.* 29 (2000), May, no. 2, 129–140. http://dx.doi.org/10.1145/335191.335393. – DOI 10.1145/335191.335393. – ISSN 0163–5808

[SC05] Stonebraker, Michael ; Cetintemel, Ugur: "One Size Fits All": An Idea Whose Time Has Come and Gone. In: *Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 2005 (ICDE '05). – ISBN 0–7695–2285–8, 2–11

[SCF+86] Schwarz, P. ; Chang, W. ; Freytag, J. C. ; Lohman, G. ; McPherson, J. ; Mohan, C. ; Pirahesh, H.: Extensibility in the Starburst Database System. In: *Proceedings on the 1986 International Workshop on Object-oriented Database Systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1986 (OODS '86). – ISBN 0–8186–0734–3, 85–92

[Sch15] Schaarschmidt, Michael: *Towards Latency: An Online Learning Mechanism for Caching Dynamic Query Content*, University of Cambridge, master's thesis, 2015

[Sch18] Schütt, Randy: *Inkrementelle Auswertung von MongoDB-Volltextsuchanfragen*, University of Hamburg, master's thesis, 2018

[SCZ05] Stonebraker, Michael ; Cetintemel, Uğur ; Zdonik, Stan: The 8 Requirements of Real-time Stream Processing. In: *SIGMOD Rec.* 34 (2005), December, no. 4, 42–47. http://dx.doi.org/10.1145/1107499.1107504. – DOI 10.1145/1107499.1107504. – ISSN 0163–5808

[SD95] Simon, Eric ; Dittrich, Angelika K.: Promises and Realities of Active Database Systems. In: *Proceedings of the 21th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1995 (VLDB '95). – ISBN 1–55860–379–4, 642–653

[SDSB18] Sigoure, Benoît ; Demir, Berk D. ; Smith, Mark ; Barr, Dave: *OpenTSDB*. http://opentsdb.net/. version: 2018. – Accessed: 2018-05-10

[Sel18] Selva, Andrea: *Moquette*. https://andsel.github.io/moquette/. version: 2018. – Accessed: 2018-05-27

[SF12] Sadalage, Pramod J. ; Fowler, Martin: *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 1st. Addison-Wesley Professional, 2012. – ISBN 0321826620, 9780321826626

[SFK00] Sandhu, Ravi ; Ferraiolo, David ; Kuhn, Richard: The NIST Model for Role-based Access Control: Towards a Unified Standard. In: *Proceedings of the Fifth ACM Workshop on Role-based Access Control*. New York, NY, USA :

ACM, 2000 (RBAC ’00). – ISBN 1–58113–259–X, 47–63

[SGDY16] Schaarschmidt, Michael ; Gessert, Felix ; Dalibard, Valentin ; Yoneki, Eiko: Learning Runtime Parameters in Computer Systems with Delayed Experience Injection. In: *CoRR* abs/1610.09903 (2016). http://arxiv.org/abs/1610.09903

[SGR15] Schaarschmidt, Michael ; Gessert, Felix ; Ritter, Norbert: Towards Automated Polyglot Persistence. In: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs ”Datenbanken und Informationssysteme”*, 2015

[SH12] Senthamilarasu, S. ; Hemalatha, M.: Load shedding techniques based on windows in data stream systems. In: *2012 International Conference on Emerging Trends in Science, Engineering and Technology (INCOSET)*, 2012, pp. 68–73

[SHWK76] Stonebraker, Michael ; Held, Gerald ; Wong, Eugene ; Kreps, Peter: The Design and Implementation of INGRES. In: *ACM Trans. Database Syst.* 1 (1976), September, no. 3, 189–222. http://dx.doi.org/10.1145/320473.320476. – DOI 10.1145/320473.320476. – ISSN 0362–5915

[SJGP90] Stonebraker, Michael ; Jhingran, Anant ; Goh, Jeffrey ; Potamianos, Spyros: On Rules, Procedure, Caching and Views in Data Base Systems. In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 1990 (SIGMOD ’90). – ISBN 0–89791–365–5, 281–290

[SKM92] Simon, Eric ; Kiernan, Jerry ; Maindreville, Christophe d.: Implementing High Level Active Rules on Top of a Relational DBMS. In: *Proceedings of the 18th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1992 (VLDB ’92). – ISBN 1–55860–151–1, 315–326

[SKRC10] Shvachko, Konstantin ; Kuang, Hairong ; Radia, Sanjay ; Chansler, Robert: The Hadoop Distributed File System. In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Washington, DC, USA : IEEE Computer Society, 2010 (MSST ’10). – ISBN 978–1–4244–7152–2, 1–10

[SMA$^+$07] Stonebraker, Michael ; Madden, Samuel ; Abadi, Daniel J. ; Harizopoulos, Stavros ; Hachem, Nabil ; Helland, Pat: The End of an Architectural Era: (It’s Time for a Complete Rewrite). In: *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB Endowment, 2007 (VLDB ’07). – ISBN 978–1–59593–649–3, 1150–1160

[SP89] Segev, Arie ; Park, Jooseok: Maintaining Materialized Views in Distributed Databases. In: *Proceedings of the Fifth International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 1989. – ISBN 0–8186–1915–5, 262–270

[SPAM91] Schreier, Ulf ; Pirahesh, Hamid ; Agrawal, Rakesh ; Mohan, C.: Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In: *Proceedings of the 17th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1991 (VLDB '91). – ISBN 1–55860–150–3, 469–478

[SR86] Stonebraker, Michael ; Rowe, Lawrence A.: The Design of POSTGRES. In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 1986 (SIGMOD '86). – ISBN 0–89791–191–1, 340–355

[Sta88] Stankovic, John A.: Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. In: *Computer* 21 (1988), October, no. 10, 10–19. `http://dx.doi.org/10.1109/2.7053`. – DOI 10.1109/2.7053. – ISSN 0018–9162

[Ste17] Stein, William: RethinkDB versus PostgreSQL: my personal experience. In: *CoCalc Blog* (2017), February. `https://blog.sagemath.com/2017/02/09/rethinkdb-vs-postgres.html`. – Accessed: 2017-07-09

[Sto86] Stonebraker, Michael: Object Management in POSTGRES Using Procedures. In: *Proceedings on the 1986 International Workshop on Object-oriented Database Systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1986 (OODS '86). – ISBN 0–8186–0734–3, 66–72

[STR17] STRV s.r.o. (ed.): *Rapid Docs: Collection*. STRV s.r.o., 2017. `https://www.rapidrealtime.com/docs/api-reference/javascript/collection`. – Accessed: 2017-10-06

[STR18] STRV s.r.o.: *Rapid – Realtime Database Services*. `https://www.rapidrealtime.com/`. version: 2018. – Accessed: 2018-05-10

[Suc17] Succo, Stephan: *Skalierbare, echtzeitnahe Kommunikation von Änderungen an Datenbankobjekten und Abfrageresultaten innerhalb einer Backend-as-a-Service-Architektur*, University of Hamburg, master's thesis, 2017

[SW04] Srivastava, Utkarsh ; Widom, Jennifer: Flexible Time Management in Data Stream Systems. In: *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*.

New York, NY, USA : ACM, 2004 (PODS ’04). – ISBN 158113858X, 263–274

[Tam14] TAMPLIN, James: Firebase is Joining Google! In: *Firebase Blog* (2014), October. https://firebase.googleblog.com/2014/10/firebase-is-joining-google.html. – Accessed: 2017-11-17

[Tam16] TAMPLIN, James: Firebase expands to become a unified app platform. In: *Firebase Blog* (2016), May. https://firebase.googleblog.com/2016/05/firebase-expands-to-become-unified-app-platform.html. – Accessed: 2017-12-23

[Tam17] TAMPLIN, James: Cloud Firestore: A New Document Database for Apps (commentary). In: *Hacker News* (2017). https://news.ycombinator.com/item?id=15393499. – Accessed 2017-12-19

[TCZ+03] TATBUL, Nesime ; CETINTEMEL, Uğur ; ZDONIK, Stan ; CHERNIACK, Mitch ; STONEBRAKER, Michael: Load Shedding in a Data Stream Manager. In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB Endowment, 2003 (VLDB ’03). – ISBN 0–12–722442–4, 309–320

[TDR16] THOMSON, Martin ; DAMAGGIO, Elio ; RAYMOR, Brian: *Generic Event Delivery Using HTTP Push*. RFC 8030. http://dx.doi.org/10.17487/RFC8030. version: December 2016 (Request for Comments). – Accessed: 2018-05-21

[Tec87] TECHNICAL COMMITTEE: ISO/IEC JTC 1 INFORMATION TECHNOLOGY: ISO 9075:1987: Information processing systems – Database language – SQL / International Organization for Standardization. 1987. – Standard

[Tec92] TECHNICAL COMMITTEE: ISO/IEC JTC 1/SC 32 DATA MANAGEMENT AND INTERCHANGE: ISO/IEC 9075:1992: Information technology – Database languages – SQL / International Organization for Standardization. 1992. – Standard

[TGNO92] TERRY, Douglas ; GOLDBERG, David ; NICHOLS, David ; OKI, Brian: Continuous Queries over Append-only Databases. In: *SIGMOD Rec.* 21 (1992), June, no. 2, 321–330. http://dx.doi.org/10.1145/141484.130333. – DOI 10.1145/141484.130333. – ISSN 0163–5808

[The16] THERESKA, Eno: KIP-67: Queryable state for Kafka Streams. In: *Kafka Improvement Proposals* (2016). https://cwiki.apache.org/confluence/display/KAFKA/KIP-67%3A+Queryable+state+for+Kafka+Streams. – Accessed: 2018-05-12

[TL76] TSICHRITZIS, D. C. ; LOCHOVSKY, F. H.: Hierarchical Data-Base Management: A Survey. In: *ACM Comput. Surv.* 8 (1976), March, no. 1, 105–123. http://dx.doi.org/10.1145/356662.356667. – DOI

10.1145/356662.356667. – ISSN 0360–0300

[TMSF03] TUCKER, P. A. ; MAIER, D. ; SHEARD, T. ; FEGARAS, L.: Exploiting punctuation semantics in continuous data streams. In: *IEEE Transactions on Knowledge and Data Engineering* 15 (2003), May, no. 3, pp. 555–568. http://dx.doi.org/10.1109/TKDE.2003.1198390. – DOI 10.1109/TKDE.2003.1198390. – ISSN 1041–4347

[TP06] TAO, Yufei ; PAPADIAS, Dimitris: Maintaining sliding window skylines on data streams. In: *IEEE Transactions on Knowledge and Data Engineering* 18 (2006), March, no. 3, pp. 377–391. http://dx.doi.org/10.1109/TKDE.2006.48. – DOI 10.1109/TKDE.2006.48. – ISSN 1041–4347

[TPK+13] TERRY, Douglas B. ; PRABHAKARAN, Vijayan ; KOTLA, Ramakrishna ; BALAKRISHNAN, Mahesh ; AGUILERA, Marcos K. ; ABU-LIBDEH, Hussam: Consistency-based service level agreements for cloud storage. In: *SOSP* ACM, 2013, pp. 309–324

[Tre15] TREAT, Tyler: You Cannot Have Exactly-Once Delivery. In: *Brave New Geek* (2015), March. https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/. – Accessed: 2018-07-22

[Tre16] TREAT, Tyler: Benchmarking Message Queue Latency. In: *DZone* (2016), February. https://dzone.com/articles/benchmarking-message-queue-latency. – Accessed: 2018-05-27

[Tre17] TREAT, Tyler: You Cannot Have Exactly-Once Delivery Redux. In: *Brave New Geek* (2017), June. https://bravenewgeek.com/you-cannot-have-exactly-once-delivery-redux/. – Accessed: 2018-07-22

[TS+09] TO, Lawrence ; SCHUPMANN, Viv et al. ; ORACLE (ed.): *Oracle Database High Availability Best Practices 11g Release 1 (11.1)*. Oracle, December 2009. https://docs.oracle.com/cd/B28359_01/server.111/b28282/glossary.htm#CHDIDADC. – Accessed: 2017-05-13

[TTS+14] TOSHNIWAL, Ankit ; TANEJA, Siddarth ; SHUKLA, Amit ; RAMASAMY, Karthik ; PATEL, Jignesh M. ; KULKARNI, Sanjeev ; JACKSON, Jason ; GADE, Krishna ; FU, Maosong ; DONHAM, Jake ; BHAGAT, Nikunj ; MITTAL, Sailesh ; RYABOY, Dmitriy: Storm@Twitter. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2014 (SIGMOD '14). – ISBN 978–1–4503–2376–5, 147–156

[Vin16] VINCENT, Paul: CEP Tooling Market Survey 2016. In: *complexevents.com* (2016), May. http://www.complexevents.com/2016/05/12/cep-tooling-market-survey-2016/. – Accessed: 2017-12-14

[Vis96] VISTA, Dimitri: *Optimizing incremental view maintenance expressions in relational databases*, University of Toronto, diss., 1996

[Vis98] *Chapter* EDBT 1998: Advances in Database Technology — EDBT'98. In: VISTA, Dimitra: *Integration of incremental view maintenance into query optimizers*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. – ISBN 978–3–540–69709–1, 374–388

[VPAU15] VENKAT, Bharat ; PADMANABHAN, Prasanna ; AROKIASAMY, Antony ; UPPALAPATI, Raju: Can Spark Streaming survive Chaos Monkey? In: *Netflix Tech Blog* (2015), Marriage. http://techblog.netflix.com/2015/03/can-spark-streaming-survive-chaos-monkey.html. – Accessed: 2016-01-11

[VRR10] VIDAČKOVIĆ, Krešimir ; RENNER, Thomas ; REX, Sascha: Marktübersicht Real-Time Monitoring Software: Event Processing Tools im Überblick / Fraunhofer Verlag, Fraunhofer-Informationszentrum Raum und Bau IRB. 2010. – technical report

[VSGC10] VEMURI, S.S. ; SINHA, B. ; GANESH, A. ; CHITTI, S.B.: *Generating continuous query notifications*. https://www.google.com/patents/US20100036831. version: February 2010. – US Patent App. 12/189,078

[VV16] VIOTTI, Paolo ; VUKOLIC, Marko: Consistency in Non-Transactional Distributed Storage Systems. In: *ACM Comput. Surv.* 49 (2016), no. 1, pp. 19:1–19:34. http://dx.doi.org/10.1145/2926965. – DOI 10.1145/2926965

[W⁺14] WORKMAN, David et al.: Large number of operations hangs server. In: *Meteor GitHub Issues* (2014). https://github.com/meteor/meteor/issues/2668. – Accessed: 2016-10-01

[Wal17] WALTHER, Timo: From Streams to Tables and Back Again: An Update on Flink's Table & SQL API. In: *Flink Blog* (2017), March. https://flink.apache.org/news/2017/03/29/table-sql-api-update.html. – Accessed: 2017-10-27

[Wan16a] WANG, Mengyan: Parse LiveQuery Protocol Specification. In: *GitHub: ParsePlatform/parse-server* (2016), March. https://github.com/parse-community/parse-server/wiki/Parse-LiveQuery-Protocol-Specification. – Accessed: 2017-11-18

[Wan16b] WANG, Mengyan: Parse Server Goes Realtime with Live Queries. In: *Parse Blog* (2016), March. http://blog.parseplatform.org/announcements/parse-server-goes-realtime-with-live-queries/. – Accessed: 2017-11-18

[WBL+07] Witkowski, Andrew ; Bellamkonda, Srikanth ; Li, Hua-Gang ; Liang, Vince ; Sheng, Lei ; Smith, Wayne ; Subramanian, Sankar ; Terry, James ; Yu, Tsae-Feng: Continuous Queries in Oracle. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB Endowment, 2007 (VLDB '07). – ISBN 978–1–59593–649–3, 1173–1184

[WCB01] Welsh, Matt ; Culler, David ; Brewer, Eric: SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In: *SIGOPS Oper. Syst. Rev.* 35 (2001), October, no. 5, 230–243. `http://dx.doi.org/10.1145/502059.502057`. – DOI 10.1145/502059.502057. – ISSN 0163–5980

[Wen15] Wenger, Jacob: List chat group in order of most recently posted. In: *Firebase Google Group* (2015), June. `https://groups.google.com/forum/#!msg/firebase-talk/d-XjaBVL2Ko/TmkIep44lGgJ`. – Accessed: 2017-07-09

[WGF+17] Wingerath, Wolfram ; Gessert, Felix ; Friedrich, Steffen ; Witt, Erik ; Ritter, Norbert: The Case For Change Notifications in Pull-Based Databases. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany*, 2017

[WGFR16] Wingerath, Wolfram ; Gessert, Felix ; Friedrich, Steffen ; Ritter, Norbert: Real-time stream processing for Big Data. In: *it - Information Technology* 58 (2016), no. 4, 186–194. `http://dx.doi.org/10.1515/itit-2016-0002`. – DOI 10.1515/itit–2016–0002

[WGW+18] Wingerath, Wolfram ; Gessert, Felix ; Witt, Erik ; Friedrich, Steffen ; Ritter, Norbert: Real-Time Data Management for Big Data. In: *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, OpenProceedings.org, 2018

[Wid05] Widom, Jennifer: The Stanford Data Stream Management System. In: *Microsoft Research Lectures* (2005), July. `https://www.microsoft.com/en-us/research/video/the-stanford-data-stream-management-system/`. – lecture video (relevant part: 25m45s to 26m48s); Accessed: 2018-07-30

[Wie15] Wiese, Lena: *Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases*. DeGruyter, 2015 `http://wiese.free.fr/adm.html`. – ISBN 978–3–11–044140–6

[Win17a] Wingerath, Wolfram: Going Real-Time Has Just Become Easy: Baqend Real-Time Queries Hit Public Beta. In: *Baqend Tech Blog* (2017), September. `http://announcement.twoogle.info`. – Accessed: 2017-09-26

[Win17b] Wingerath, Wolfram: Real-Time Databases Explained: Why Meteor, RethinkDB, Parse and Firebase Don't Scale. In: *Baqend Tech Blog* (2017).

https://medium.com/p/822ff87d2f87

[Wit16] WITT, Erik: *Distributed Cache-Aware Transactions for Polyglot Persistence*, University of Hamburg, master's thesis, August 2016

[WRG18] WINGERATH, Wolfram ; RITTER, Norbert ; GESSERT, Felix: *Real-Time & Stream Data Management: Push-Based Data in Research & Practice*. Springer, book to be published in late 2018

[YAD14] YILMAZ, Yavuz S. ; AYDIN, Bahadir I. ; DEMIRBAS, Murat: Google cloud messaging (GCM): An evaluation. In: *IEEE Global Communications Conference, GLOBECOM 2014, Austin, TX, USA, December 8-12, 2014*, 2014, 2807–2812

[YCP17] YUSEIN, Fedail ; CONDON, Craig ; PFLAUM, Pascal: Doesn't act as expected! In: *sift.js GitHub Issues* (2017), March. https://github.com/crcn/sift.js/issues/117. – Accessed: 2018-08-05

[YQC+12] YANG, Fan ; QIAN, Zhengping ; CHEN, Xiuwei ; BESCHASTNIKH, Ivan ; ZHUANG, Li ; ZHOU, Lidong ; SHEN, Guobin: Sonora: A Platform for Continuous Mobile-Cloud Computing / Microsoft Research. version: March 2012. http://research.microsoft.com/apps/pubs/default.aspx?id=161446. 2012 (MSR-TR-2012-34). – technical report

[Yu15] YU, Alice: What does it mean to be a real-time database? — Slava Kim at Devshop SF May 2015. In: *Meteor Blog* (2015), June. – Accessed: 2017-05-20

[YYY+03] YI, Ke ; YU, Hai ; YANG, Jun ; XIA, Gangqiang ; CHEN, Yuguo: Efficient Maintenance of Materialized Top-k Views. In: *Proceedings of the 19th International Conference on Data Engineering* (2003)

[ZCD+12] ZAHARIA, Matei ; CHOWDHURY, Mosharaf ; DAS, Tathagata et al.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2012 (NSDI'12), 2–2

[ZDL+13] ZAHARIA, Matei ; DAS, Tathagata ; LI, Haoyuan et al.: Discretized Streams: Fault-tolerant Streaming Computation at Scale. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2013 (SOSP '13). – ISBN 978–1–4503–2388–8, 423–438

List of Figures

List of Tables

Listings

Statutory Declaration / Eidesstattliche Erklärung

English: Statutory Declaration

I hereby declare, on oath, that I have written the present dissertation entitled

“Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases”

by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such.

This thesis was not submitted in the same or in a substantially similar version, not even partially, to any other authority to achieve an academic grading and was not published elsewhere.

I agree that a copy of this thesis may be made available in the Informatics Library of the University of Hamburg.

Wolfram Wingerath Hamburg, September 24th, 2018

German: Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorstehende Arbeit mit dem Titel

„Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases“

selbständig und ohne fremde Hilfe angefertigt und mich anderer als der angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Wolfram Wingerath Hamburg, 24. September 2018