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

1st Wolfram Wingerath
Baqend GmbH
22769 Hamburg, Germany
ww@baqend.com

2nd Felix Gessert
Baqend GmbH
22769 Hamburg, Germany
fg@baqend.com

3rd Norbert Ritter
Databases and Information Systems
University of Hamburg
22527 Hamburg, Germany
ritter@informatik.uni-hamburg.de

***Abstract*—Traditional databases are optimized for pull-based queries, i.e. they make information available in direct response to client requests. While this access pattern is adequate for mostly static domains, it requires inefficient and slow workarounds (e.g. periodic polling) when clients need to stay up-to-date. Acknowledging reactive and interactive workloads, modern real-time databases such as Firebase, Meteor, and RethinkDB proactively deliver result updates to their clients through push-based real-time queries. 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. To address these issues, we propose the system design InvaliDB which combines linear read and write scalability for real-time queries with superior query expressiveness and legacy compatibility. We compare InvaliDB against competing system designs to emphasize the benefits of our approach that has been serving customers at the Database-as-a-Service company Baqend since July 2017.**

***Index Terms*—database systems, real-time queries, real-time databases, Firebase, Meteor, RethinkDB, push-based data access, result change notifications, NoSQL, document stores, MongoDB**

I. Introduction

Many of today's Web applications notify users of status updates and other events in realtime. But even though more and more usage scenarios revolve around the interaction between users, building applications that detect and publish changes with low latency remains notoriously hard even with state-of-the-art data management systems. While traditional databases[1] excel at complex queries over historical data [14], they are inherently pull-based and therefore ill-equipped to push new information to clients [20]. Systems for data stream management, in contrast, are natively push-oriented and thus facilitate reactive behavior [13]. 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 high complexity for applications that require both persistence and real-time change notifications [29].

In this paper, we present the real-time database system design InvaliDB for bridging this gap: InvaliDB is built on top of a pull-based (NoSQL) database, supports the query expressiveness of the underlying datastore for pull- and push-based queries alike, and scales linearly with reads and writes through a scheme for two-dimensional workload distribution.

[1]We use "database" and "database system" synonymously, when it is clear whether we are referring to the base data or the system maintaining it.

II. Related Work

Subsumed under the term "real-time databases"[2] [22] [31], systems like Firebase, Meteor, and RethinkDB provide push-based real-time queries that do not only deliver a single result upon request, but also produce a continuous stream of informational updates thereafter. Like traditional database systems, real-time databases store consistent snapshots of domain knowledge. But like stream management systems, they let clients subscribe to long-running queries that push incremental updates. Table I sums up the query capabilities of some of the most popular systems for real-time queries in comparison with our system design InvaliDB.

	Meteor		**RethinkDB**	**Parse**	**Firebase**	**InvaliDB**
	Poll-and-Diff	Log Tailing			Proprietary	2-D Dist.
Scales With Write Throughput	✓	✗	✗	✗	✗	✓
Scales With Number of Queries	✗	✓	✓	✓	○ (100k connections)	✓
Real-Time Change Notifications	✗	✓	✓	✓	✓	✓
Composite Queries (AND/OR)	✓	✓	✓	✓	○ (AND in Firestore)	✓
Sorted Queries	✓	✓	✓	✗	○ (single attribute)	✓
Limit	✓	✓	✓	✗	✓	✓
Offset	✓	✓	✗	✗	○ (value-based)	✓

TABLE I: A direct comparison of the different collection-based real-time query implementations detailed in this paper.

Meteor [15] is the only system featuring two different real-time query implementations: Poll-and-diff relies on periodic query reexecution and scales with write throughput, whereas log tailing relies on processing changes within the application server and scales with the number of concurrent real-time queries – neither scales with both [16]. RethinkDB [2] and Parse [4] provide real-time queries with log tailing as well and therefore also collapse under heavy write load: A lack of write stream partitioning represents a scale-prohibitive bottleneck in the designs of all these systems. While the technology

[2]In the past, the term "real-time databases" has been used to reference specialized pull-based databases that produce an output within strict timing constraints [19] [1] [8]; we do not share this notion of real-time databases.

stacks behind Firebase [3] and Firestore [7] are not disclosed, hard scalability limits for write throughput and parallel client connections are documented [9] [11]. Further, it is apparent that both services mitigate 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 [25]. Even the more advanced Firestore queries lack support for disjunction of filter expressions (`OR`) and only provide limited options for filter conjunction (`AND`) [10]. 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 supports limit (but no offset) [17], and Parse does not support ordered real-time queries at all [23].

We are not aware of any system that carries 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 InvaliDB, we show that expressive real-time queries and scalability can go hand-in-hand.

III. InvaliDB: Scalable Real-Time Queries for Traditional Databases

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: Within the Quaestor architecture [12] for consistent caching of query results, InvaliDB is used to <u>invali</u>date cached <u>d</u>ata<u>b</u>ase queries once they become stale, i.e. it detects result alterations and purges the corresponding result caches in a timely manner.

Similar to some current systems (e.g. Meteor and 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 (**event layer**). To enable real-time queries, an application server only runs a lightweight process (**InvaliDB client**) which relays messages between the end users, the database, and the InvaliDB cluster. The expensive task of matching active real-time queries against ongoing writes, on the other hand, is offloaded to the InvaliDB cluster. By thus decoupling real-time query workload from the main application logic, even overburdening the real-time component cannot take down the OLTP system: As the InvaliDB cluster becomes unresponsive, only requests sent against the event layer remain unanswered, but pull-based queries continue to work.

In order to subscribe to a real-time query, a Web or mobile application sends a **subscription** request to one of potentially many application servers. The application server then executes the query against the database to produce the initial result, i.e. the data objects currently matching the query. 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. Similar to a real-time query subscription, a request for real-time query *cancellation* is asynchronously passed to the InvaliDB cluster, so that the given query can be deactivated and does not consume further resources. To make zombie queries expire eventually, subscriptions are further registered with a time to live (TTL) that is extended by every live subscriber through a periodic *TTL extension* request. For every **write** which is executed at the database, the *after-images* (i.e. fully specified representations) of the written entities are handed to the InvaliDB cluster. Every after-image is then matched against all active real-time queries to detect changes to currently maintained results. As a response to a real-time query subscription, the InvaliDB cluster sends out a stream of **change notifications** each of which represents a transition of the corresponding query result from one state to another. Every notification carries the information required to implement the corresponding result change, e.g. an after-image of the written entity and a *match type* that encodes the exact kind of result change: `add` (new result member), `change` (result member was updated), `changeIndex` (sorted queries only: result member was updated and changed its position in the query result), `remove` (item left the result). 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. Since communication over the event layer is asynchronous, InvaliDB may receive after-images delayed or skewed (compared with the order in which the corresponding write operations arrive at the database). While a query result maintained by InvaliDB may thus diverge temporarily from database state, it will synchronize once InvaliDB has applied the same write operations as the database. InvaliDB thus follows **eventual consistency** [6].

A. Two-Dimensional Workload Distribution

To enable higher input rates than a single machine could handle, the InvaliDB cluster partitions both the query subscriptions and incoming writes 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 1 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

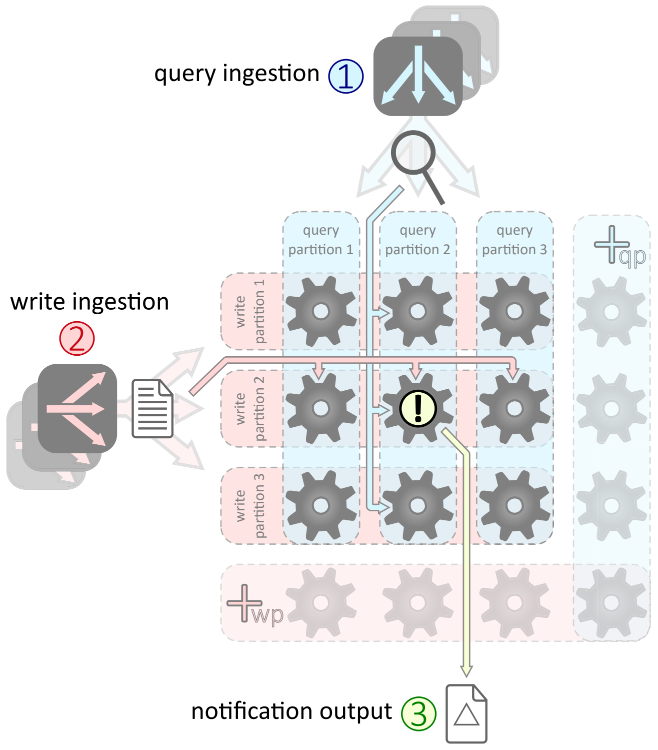

Fig. 1: InvaliDB partitions both queries and writes, so that any given matching node is only responsible for matching few queries against some of the incoming writes.

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 items to the corresponding matching nodes.

To make workload distribution as even as possible, InvaliDB performs **hash-partitioning** for inbound writes and queries. 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. For queries, the corresponding hash value is computed from the query attributes when a subscription request is received, so that 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 only the subscription request contains the query attributes, though, the hash value cannot be computed for requests other than the initial subscription. To make sure that TTL extensions and cancellations can be routed correctly, an application server remembers every hash value for the entire lifetime of a subscription and attaches it to every subsequent request relating to the same subscription.

To avoid losing result changes to race conditions between the initial query result and incoming write operations, 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. In practice, retention time therefore needs to be chosen according to actually observed delays. For reference, the production deployment at Baqend retains writes for few seconds, since our InvaliDB prototype exhibits consistent notification latencies in the realm of double-digit milliseconds with subsecond peaks (see [26, Ch. 5]). Since communication is purely asynchronous, write stream retention is crucial for **staleness avoidance** as write operations that might have been missed can always be replayed safely: Since InvaliDB relies on data to be versioned, write operations are simply ignored whenever a more recent insert, update, or delete for the same item has already been received.

B. *Advanced Queries With Processing Stages*

By partitioning queries and writes orthogonally to one another, the task of evaluating query predicates is evenly distributed across all nodes in the cluster. While this scheme avoids hotspots, however, it also prevents capturing the context between items in a result: As every matching node holds only result partitions, result changes can only be registered on a per-record basis. In more detail, changes relating to the sorting order (match type `changeIndex`) 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. In order to make InvaliDB suitable for these kinds of real-time queries without impairing overall scalability, the process of generating change notifications for sorted queries, joins, and aggregations is performed in loosely coupled **processing stages** that can be scaled independently (cf. staged event-driven architecture (SEDA) [24]). While the InvaliDB implementation at Baqend supports sorted queries with limit and offset, real-time joins and aggregations are still ongoing work. We refer to [26, Ch. 3] for details.

IV. Prototype & Industry Applications

Since July 2017, an implementation of the system design described in this paper has been used in production at the company Baqend. Our InvaliDB prototype is built with the distributed stream processor Storm [21] (workload distribution) and the in-memory store Redis [5] (event layer) to provide real-time queries on top of the NoSQL database MongoDB [18]. While we chose MongoDB for our initial prototype, we implemented a **pluggable query engine**, so that support for additional database languages (e.g. SQL) could be added later with relative ease. This is possible, because most aspects of our system design are database-agnostic. For example, the event layer only handles opaque data transmissions between the application server and the InvaliDB cluster. Similarly, the two-dimensional workload distribution scheme does not assume any concrete database language to be used, either.

At Baqend, InvaliDB serves two different purposes. First, it adds push-based **real-time queries** to the otherwise pull-based query interface of the MongoDB-based Database-as-a-Service. As a second use case, InvaliDB makes **query result caching** feasible by providing low-latency invalidation messages for stale query results and thereby improves throughput and latency of the underlying pull-based query mechanisms by more than an order of magnitude [12].

V. Related Publications & Further Reading

This paper contains revised material from the PhD thesis [26] in the context of which InvaliDB has been developed and which has spawned several other publications on related topics. The introduction in Section I, the related work part in Section II, the system design description in Section III, the overview of our industry prototype in Section IV, and the conclusion in Section VI further contain revised material from earlier tutorial and demo abstracts [27] [28] [29]. In our other publications, we present a vastly extended version of our related work [30], an extensive performance evaluation [26, Ch. 4 & 5], and details on our industry use cases [12].

VI. Conclusion

While the rising popularity of push-based datastores like Firebase and Meteor indicates a public demand for databases with real-time queries, no current implementation combines expressiveness, high scalability, and fault tolerance. In this paper, we propose the novel real-time database design InvaliDB that adds push-based real-time queries as an opt-in feature to existing pull-based databases, but avoids the limitations other state-of-the-art systems are constrained by. InvaliDB sets itself apart from existing system designs through (1) a novel two-dimensional workload partitioning scheme for *linear scalability*, (2) support for *expressive* real-time queries including sorted filter queries with limit and offset, (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. The production deployment at Baqend supports MongoDB expressiveness and has been serving customers for several years in different applications. These facts confirm that our design is feasible to implement and can service a wide range of use cases. We hope that our work sparks new confidence in the practicality of real-time databases and that it inspires further research on the topic within the database community.

References

[1] *Active, Real-Time, and Temporal Database Systems*, 1998.
[2] RethinkDB, 2016. https://www.rethinkdb.com/, accessed: 2019-09-21.
[3] Firebase, 2019. https://firebase.google.com/, accessed: 2019-04-15.
[4] Parse, 2019. https://parseplatform.org/, accessed: 2019-09-21.
[5] Redis, 2019. https://redis.io/, accessed: 2019-10-05.
[6] P. Bailis and A. Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Queue*, 11(3):20:20–20:32, Mar. 2013.
[7] A. Dufetel. Introducing Cloud Firestore: Our New Document Database for Apps. *Firebase Blog*, Oct. 2017. https://firebase.googleblog.com/2017/10/introducing-cloud-firestore.html, accessed: 2017-12-19.
[8] J. Eriksson. Real-Time and Active Databases: A Survey. In *Active, Real-Time, and Temporal Database Systems: Second International Workshop, ARTDB-97 Como, Italy, September 8–9, 1997 Proceedings*, 1998.
[9] Firebase. *Choose a Database: Cloud Firestore or Realtime Database*, Dec. 2017. https://firebase.google.com/docs/firestore/rtdb-vs-firestore, accessed: 2017-12-19.
[10] Firebase. *Order and Limit Data with Cloud Firestore*, Dec. 2017. https://firebase.google.com/docs/firestore/query-data/order-limit-data, accessed: 2017-12-19.
[11] Firebase. *Realtime Database Limits*, 2017. https://firebase.google.com/docs/database/usage/limits, accessed: 2017-11-17.
[12] F. Gessert, M. Schaarschmidt, W. Wingerath, E. Witt, E. Yoneki, and N. Ritter. Quaestor: Query web caching for database-as-a-service providers. *Proceedings of the 43rd International Conference on Very Large Data Bases, VLDB 2017*, 2017.
[13] L. Golab and M. T. Zsu. *Data Stream Management*. Morgan & Claypool Publishers, 2010.
[14] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a Database System. *Found. Trends databases*, 1(2):141–259, Feb. 2007.
[15] S. Hochhaus and M. C. Schoebel. *Meteor in Action*. Manning Publications Co., 2016.
[16] J. Katzen et al. Oplog Tailing Too Far Behind Not Helping, 2015. https://forums.meteor.com/t/oplog-tailing-too-far-behind-not-helping/2235, accessed: 2017-07-09.
[17] W. Martin. Changefeeds in RethinkDB. *RethinkDB Docs*, 2015. rethinkdb.com/docs/changefeeds/javascript/, accessed: 2017-07-09.
[18] MongoDB Inc., 2019. https://mongodb.com, accessed: 2019-09-15.
[19] B. Purimetla, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. A Study of Distributed Real-Time Active Database Applications. Technical report, Amherst, MA, USA, 1993.
[20] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.
[21] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.
[22] F. van Puffelen. Have you Met the Realtime Database? *Firebase Blog*, July 2016. accessed: 2017-05-20.
[23] M. Wang. Parse LiveQuery Protocol Specification. *GitHub*, 2016. https://github.com/parse-community/parse-server, accessed: 2019-10-05.
[24] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 2001.
[25] J. Wenger. List Chat Group in Order of Most Recently Posted. *Firebase Google Group*, June 2015. accessed: 2017-07-09.
[26] W. Wingerath. *Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases*. PhD thesis, University of Hamburg, 2019. https://invalidb.info/thesis.
[27] W. Wingerath, F. Gessert, and N. Ritter. NoSQL & Real-Time Data Management in Research & Practice. In *Proceedings of the 18th Conference on Business, Technology, and Web, BTW 2019*, 2019.
[28] W. Wingerath, F. Gessert, and N. Ritter. Twoogle: Searching Twitter With MongoDB Queries. In *Proceedings of the 18th Conference on Business, Technology, and Web, BTW 2019*, 2019.
[29] W. Wingerath, F. Gessert, E. Witt, S. Friedrich, and N. Ritter. Real-Time Data Management for Big Data. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018*, 2018.
[30] W. Wingerath, N. Ritter, and F. Gessert. *Real-Time & Stream Data Management: Push-Based Data in Research & Practice*. Springer International Publishing, 2019.
[31] A. Yu. What Does It Mean to Be a Real-Time Database? — Slava Kim at Devshop SF May 2015. *Meteor Blog*, June 2015. accessed: 2017-05-20.