Towards Automated Polyglot Persistence

Michael Schaarschmidt, Felix Gessert, Norbert Ritter

Database and Information Systems Group
University of Hamburg
mks40@cam.ac.uk, {gessert, ritter}@informatik.uni-hamburg.de

Abstract: In this paper, we present an innovative solution for providing automated polyglot persistence based on service level agreements defined over functional and non-functional requirements of database systems. Complex applications require polyglot persistence to deal with a wide range of database related needs. Until now, the overhead and the required know-how to manage multiple database systems prevents many applications from employing efficient polyglot persistence solutions. Instead, developers are often forced to implement one-size-fits-all solutions that do not scale well and cannot easily be upgraded. Therefore, we introduce the concept for a Polyglot Persistence Mediator (PPM), which allows for runtime decisions on routing data to different backends according to schema-based annotations. This enables applications to either employ polyglot persistence right from the beginning or employ new systems at any point with minimal overhead. We have implemented and evaluated the concept of automated polyglot persistence for a REST-based Database-as-a-Service setting. Evaluations were performed on various EC2 setups, showing a scalable write-performance increase of 50-100% for a typical polyglot persistence scenario as well as drastically reduced latencies for reads and queries.

1 Introduction

Polyglot persistence is the concept of using different database systems within a single application domain, addressing different functional and non-functional needs with each system [SF12]. While virtually any non-single-purpose application could benefit from polyglot persistence, there are currently some obvious drawbacks. Designing and implementing an application on multiple databases is considerably harder than just using one backend. Furthermore, application demands begin to exceed capabilities of single databases. At the same time, overhead of configuration, deployment and maintenance increases drastically with each database system used. Therefore today, superior polyglot persistence solutions are often abandoned for lack of know-how and resources.

Recently, cloud providers began to allow programmers to develop and deploy applications at stellar paces. However, using a state-of-the-art cloud services, tenants still have to make the choice of using a certain database [Hac02, Agr09, Cur11]. To solve this dilemma, we are going to present the PPM as a new kind of middleware service layer. Employing a service level agreement (SLA), developers can define specific requirements for their schemas. Schemas are centrally managed by the Orestes [Ges14] middleware, which maps them to individual database protocols. Even though many NoSQL systems do not employ schemas (although some like DynamoDB or Cassandra do), we assume the presence of a central schema in order to compute a database-independent evaluation of requirements.

For illustration, consider the scenario of an online newspaper: usually, there is a ranking of the most popular articles. To this end, articles have to maintain hit-counters that need to be written up to many thousand times per second. Assume the articles themselves are kept in a document store and the application manages a sorted set of hit-counters for news articles. The article itself is rarely changed after publication. Nevertheless, if hit-counters and articles were stored together, too many writes on the hit-counter would eventually slowdown reads. Using the SLA annotations for write-throughput and another annotation for sorting, the PPM could decide to split objects so that writes on the hit-counter are stored to Redis (achieving much higher throughputs), while reads are directed to MongoDB.

In this paper, we will first introduce a discrete classification of functional and non-functional requirements of database systems, especially NoSQL systems. This enables us to derive the concept of an automated choice of backends. In section 3, we explain the architecture of our prototype. In section 4, we introduce our EC2 benchmark scenario. We will discuss a number of enhancements in section 5.

2 Concept

Automated polyglot persistence requires formal decision criteria. To this end, we first need a classification of functional and non-functional requirements. Table 1 provides an overview of such requirements. On the highest level, requirements are divided into binary and continuous requirements. Binary requirements support yes-or-no decisions: either a database supports server-side joins or it does not. This might be subjective for some non-functional requirements like scalability, since there is no agreed-upon way of measuring it. Continuous requirements like write latency on the other hand can be evaluated by comparing specific values to the context of an application. For instance, low latency for an interactive website usually would not constitute low latency for high-frequency trading. Therefore, automated polyglot persistence needs to provide rich-syntax SLAs that can be parametrized with application-specific goals.

2.1 Requirements

Figure 1 provides an overview of our concept of annotation-based SLAs on those requirements. On a high level, tenants define schemas consisting of databases, classes and fields. Tenants may then define annotations that can be used to annotate complete databases, classes or attributes (fields) of a class. Binary functional and non-functional requirements (capabilities) annotated at a certain hierarchy level result in constraints that have to be met by every entity on and below that level. Continuous requirements are simply pushed down to the field level, i.e., if a complete database is annotated to support 99.5% of read avail-

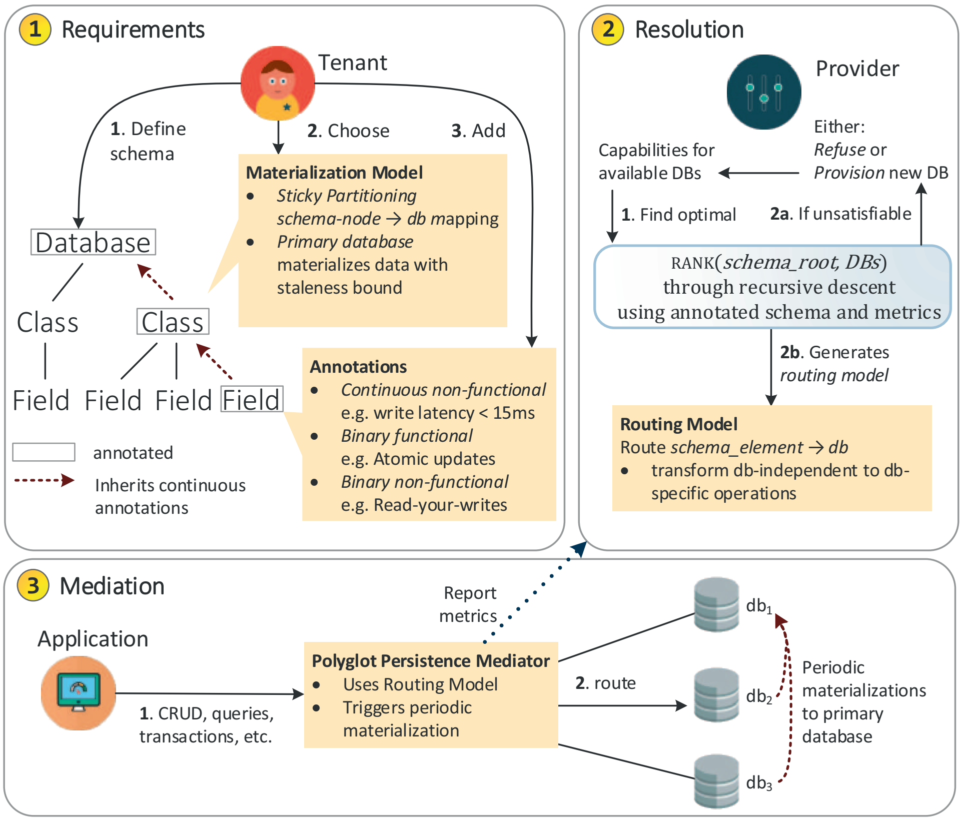

1 Requirements
Tenant
1. Define schema
2. Choose
3. Add
Materialization Model
Sticky Partitioning schema-node → db mapping
Primary database materializes data with staleness bound
Database
Class
Class
Field
Field
Field
Field
annotated
Inherits continuous annotations
Annotations
Continuous non-functional e.g. write latency < 15ms
Binary functional e.g. Atomic updates
Binary non-functional e.g. Read-your-writes
2 Resolution
Provider
Either: Refuse or Provision new DB
Capabilities for available DBs
1. Find optimal
2a. If unsatisfiable
RANK(schema_root, DBs) through recursive descent using annotated schema and metrics
2b. Generates routing model
Routing Model
Route schema_element → db
transform db-independent to db-specific operations
3 Mediation
Application
Report metrics
Polyglot Persistence Mediator
Uses Routing Model
Triggers periodic materialization
1. CRUD, queries, transactions, etc.
2. route
db1
db2
db3
Periodic materializations to primary database

Figure 1: SLA concept and persistence mediation.

ability, every field in that database inherits that annotation. Enforcing continuous requirements on higher levels (i.e., classes) is not feasible, since this would require knowledge about the distribution of this requirement on all its child entities.

An annotation consists of an arbitrary number of binary requirements and continuous non-functional requirements specified either through utility functions or specific goals on the requirement (e.g., latency below 20 milliseconds). In the following section, we are going to demonstrate how these specifications translate to routing decisions by introducing our scoring model. We will also introduce how annotations combined over different hierarchy levels can be resolved. Finally, using polyglot persistence, tenants also have to chose a materialization model: Sticky partitioning instructs the mediator to always route operations for a schema-node to the same database. The primary database defines a read-only master copy to which the mediator periodically materializes data stored in other databases. A staleness bound defines the maximum tolerable delay Δ between materializations. Under the primary copy model, applications must thus tolerate eventual consistency with Δ-atomicity, i.e. the possibility of reading a value that has been stale for at most Δ. This model allows to unite complex, slightly stale queries in the primary database with high throughput and low latency for simple updates and queries performed in other databases.

2.2 Resolution and scoring

Annotations are resolved by the provider by first comparing the specified binary requirements with all currently available systems. If no system can provide the desired combination, the provider can either reject the annotation right away or try to provision another type of database. In this context, a system is one specific deployment of a database, i.e., an IaaS provider might manage a number of different configurations of the same database. All databases capable of delivering the binary requirements are then scored to find an optimal setup for a specific tenant. The evaluation of annotations is performed recursively over all hierarchy levels of the schema, as shown in algorithm 1.

Annotation	Type	Annotated at
Read Availability	Continuous	*
Write Availability	Continuous	*
Read Latency	Continuous	*
Write Latency	Continuous	*
Write Throughput	Continuous	*
Data Vol. Scalability	Non-Functional	Field/Class/DB
Write Scalability	Non-Functional	Field/Class/DB
Read Scalabilty	Non-Functional	Field/Class/DB
Elasticity	Non-Functional	Field/Class/DB
Durability	Non-Functional	Field/Class/DB
Replicated	Non-Functional	Field/Class/DB
Consistency*	Non-Functional	Field/Class
Scans	Functional	Field
Sorting	Functional	Field
Range Queries	Functional	Field
Point Lookups	Functional	Field
ACID Transactions	Functional	Class/DB
Conditional Updates	Functional	Field
Joins	Functional	Class/DB
Analytics Integration	Functional	Field/Class/DB
Fulltext Search	Functional	Field
Atomic Updates	Functional	Field/Class

Table 1: Proposed annotations.
*Multiple levels of consistency.

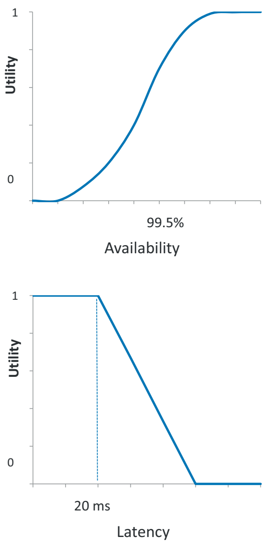


Figure 2: Utility functions

We start at the database (root) level of the schema. First, we exclude all databases incompatible with the current node's constraints (i.e., its binary requirements), so child nodes cannot choose databases their parent nodes do not support. Field nodes simply calculate their scores according to the scoring model below. Other nodes recursively calculate mappings of databases to scores for their child nodes. We then intersect among the resulting databases (line 7) to find systems that can support all requirements of the child nodes and average over the resulting scores of each database of each child node. When the recursion returns to the current node and it is annotated (line 8), it adds the resulting optimal mapping to the routing model. Finally, (aggregated) scores are returned.

Consider the following example: a class is annotated towards object-level atomic updates. One field of this class requires a certain level of write-throughput, another one has an annotation for read-latency. At the root level, there is nothing to do, so the algorithm turns to class nodes. For the specific class, all databases incapable of ensuring atomicity are removed. Individual field level annotations are now evaluated and each returns a set of databases capable of supporting both their binary and continuous requirements. The recursion then returns and computes the intersection of each field node, determining which databases can best support the required combination of throughput and latency. Finally, the class node adds the result to its routing model and all data items will be stored to the according database. Annotating a node with a binary requirement at database or class level means that all items of this database/class will be stored to the same node. The algorithm ensures compatibility of annotations along the schema hierarchy. It should be noted that schema or annotation changes may require repartitioning, the details of which we leave to future work.

Algorithm 1 Scoring algorithm for input schema node

```
1:  procedure RANK(node, DBs) returns $\{db \to score\}$
2:      drop $db \in DB$ if not $node.annotations \subseteq db.capabilities$
3:      if $node$ is $field$ then
4:          $scores \leftarrow \{db \in DBs \to score(db, node)\}$
5:      else
6:          $childScores \leftarrow \{(child, db, score) \mid$
                $child \in node.children$ and
                $(db, score) \in$ RANK$(child, DBs)\}$
7:          $scores \leftarrow db$, avg$(score)$ from $childScores$
                group by $db$
                having count$(child) = |node.children|$
8:      if $node$ is $annotated$ then
9:          add $(node \to argmax_{db}\ scores)$ to $routingModel$
10:     return $scores$
```

The score of a database is calculated by adding individual scores for each continuous non-functional requirement $cn \in CN$. Tenants can also assign arbitrary weights to each requirement to model relative importance. The total score is then normalized by the sum of weights.

$$score(db) = \frac{\sum_{i=1}^{|CN|} w_i * f_i(metric(cn_i))}{\sum_{i=1}^{|CN|} w_i} \tag{1}$$

For our scoring model, we propose two alternatives. First, we consider requirement-specific, normalized utility functions. A utility function maps values of a particular metric to the utility this requirement has for a specific use-case: $f(metric) \to utility \in [0,1]$.

Figure 2 shows two examples of requirement-specific utility. For instance, an interactive application may consider any latency below 20 milliseconds to be acceptable, with a linear decrease to zero utility at, for instance, 50 milliseconds. On the other hand, availability might be scored by a sigmoid function, indicating that availability below a certain threshold is of zero utility and then drastically increases in utility up to some point of saturation (e.g., 98% vs 99.99% of availability). Practically, users could interactively manipulate these functions in a service dashboard. Normalization of utility functions helps computing a unified score and to define SLA violations. For instance, the SLA may include multiple thresholds with different consequences. First, monitoring such thresholds helps providers to better understand their setups and the impact of changes on its performance. Falling under a certain threshold could also result in auto-scaling the respective database. Second, violations may trigger compensations for tenants in the form of refunds or service credit.

The second scoring model does not require users to specify a mapping of values to utilities. Instead, they simply specify goal values for each requirement. Goals are then compared against current metrics of the system in a manner of performance indexing [LS13]: $f(cn_i) = goal(cn_i)/metric(cn_i)$. For instance, a goal of 50 milliseconds in latency compared to an actual average latency of 20 milliseconds would result in a score of 2.5. Thanks to the collected metrics, arbitrary SLA models may be defined (e.g. pricing models based on deviations [Bas12]). After computing scores, the database with the maximum score will be selected to store the annotated field. This decision is made based on both the current and historic values, using a weighted moving average of all the metrics collected by the provider (i.e., calculating either performance indices or current values of all utility functions). If annotations are changed later, the provider would have to support live data migration between different databases.

2.3 Mediation

The Polyglot Persistence Mediator acts as a broker between applications and backend databases. Applications use a defined interface, e.g., a REST API, to issue queries, CRUD operations, transactions and other operations to the mediator in a database-agnostic fashion. Based on the routing model the mediator selects the appropriate database and transforms the incoming operation to database-specific operations:

$$transform(agnosticOperation, db) \rightarrow dbOperations \tag{2}$$

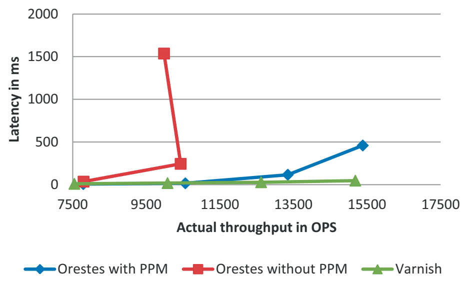
As an example, consider the addition of a value to an array-valued field, for which the resolution step determined MongoDB as most appropriate for the tenant's annotations. The mediator looks up the affected field in the routing model which yields *MongoDB*. Next, the mediator queries its transformation rule repository to map the incoming operation *push(obj, field, val)* to the specific operation {*$push:* {*field: val* }} and forwards it to the selected MongoDB cluster. In general, an operation can be transformed into a set of operations to account for potential data model transformations and denormalization (e.g., secondary index maintenance).

The mediator is stateless towards clients and can thus be replicated arbitrarily for linear scalability. Updates to the routing model (schema changes) can be shared through coordination services (e.g., Zookeeper [Hun10]), consensus protocols (e.g., Paxos [Lam01], Raft

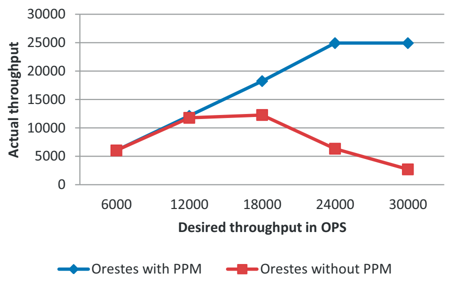
[OO14]) or classic 2PC with different availability/consistency trade-offs [Bak11, Bai12]. The mediator also manages the aforementioned materialization model.

The PPM continuously monitors operations issued to backend databases to report metrics for throughput, latency and availability, aggregating them into means, modes, weighted moving averages, percentiles and standard deviations. The metrics are used for scoring in the resolution step as well as the detection of SLA violations. Scoring thus reflects the actual system state. Of course, when a new database is provisioned, metrics are unknown. They can either be estimated using a system performance model of the database or by running a synthetic workload [Coo10, Sob08] or historic traces. The mediator allows two deployment models: The PPM can be deployed on-premise (e.g., in a private cloud) relying on local database deployments and Database-as-a-Service offers of public cloud providers (e.g. Amazon RDS/DynamoDB/S3, Windows Azure Table Storage).

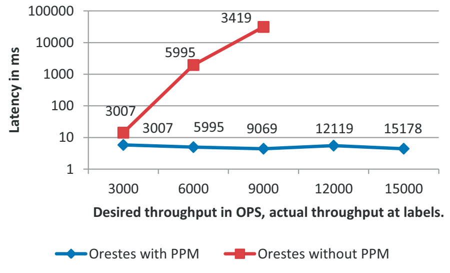
3 Architecture for a Polyglot Persistence Mediator



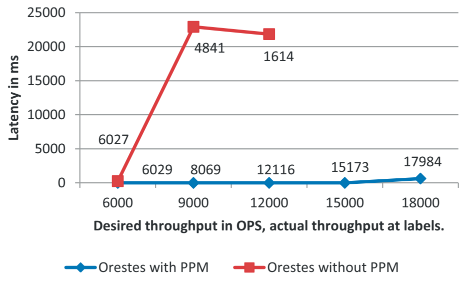
(a) Latency behaviour for a single client/single server set-up.



(c) Write-only benchmark results, 12 clients and 4 servers



(b) Read-only benchmark results, 12 clients and 4 servers.



(d) Mixed benchmark results, 12 clients and 4 servers, 95% reads.

Figure 3: Evaluation of the Polyglot Persistence Mediator.

We implemented a prototype of the PPM as part of a middleware called Orestes, which provides standardised, REST-based access to different databases [Ges14]. More specifically, applications can use a universal interface which maps all client-side operations to the appropriate server-side protocols, depending on the database an application prefers. Orestes defines a data model consisting of objects and buckets that correspond to, for instance, MongoDB documents and collections. We can thus achieve polyglot persistence on a field level by adding annotations as schema meta data.

In our experiments, we used MongoDB as the principal storage facility (i.e., the primary database materialization model) and Redis as a caching layer to accelerate writes, which constitutes a typical polyglot persistence scenario. The PPM analyses annotations (using simple performance indices) and routes partial updates on objects annotated towards write-throughput, write-latency etc. to Redis. The PPM contains a module that schedules materialisation between databases while providing strong guarantees: strictly ordered materialisation and an upper bound on the materialisation time for any particular object. Similar measures are taken to ensure delete operations are carried out for all databases containing parts of an object.

4 Evaluation

Evaluations were performed using various Amazon EC2 set-ups. To this end, we implemented our own benchmark client, collecting metrics similar to the Yahoo! Cloud Serving Benchmark (YCSB) [Coo10], i.e., averages, min/max and percentiles on latency as well as average throughput.

The first test scenario describes a single client/single server setup for the aforementioned scenario of hit-counters in online newspapers. Access patterns were generated according to a Zipf-distribution. Materialisation intervals were set to 60 seconds. Figure 3a demonstrates typical latency behaviour for medium throughputs. Notably, the actual throughput of Orestes without PPM even decreases on higher loads (backwards curve). MongoDB and Redis each ran on a separate *m1.large* instance, whereas the benchmark client and the Orestes server with/without PPM each ran on a *c3.4xlarge* since the main overhead occurs between client and Orestes server. MongoDB instances were equipped with 1,000 provisioned IOPS for its main storage volume and the write-concern was set to *acknowledged*.

For the benchmark, 100,000 update operations were executed on 100 different articles in MongoDB with hit-counters annotated to be stored in Redis. The line chart shows that the Orestes middle-ware supported by the PPM performs significantly better than Orestes without a PPM. In this context, Orestes without a PPM means that all database operations were routed to a single database (i.e., MongoDB). We found that the throughput-limit corresponds to an average latency of approximately 500 milliseconds. In this setup, the PPM achieved a 50% increase in write-throughput while maintaining lower latencies throughout. For a HTTP baseline, we used Varnish to benchmark performing GET requests against a static resource. At low throughputs, the PPM constantly achieved better latencies than both Varnish and Orestes without a PPM. Subfigures b to d demonstrate results for a set-up of 12 benchmark clients, 4 servers and 1 server for each database (all *m1.large*). Figure 3b shows results for a read-only benchmark. In our use-case of top-listed articles, we were interested only in the top 10 articles in the sorted set. While articles were stored in MongoDB, PPM-supported reads were routed to the Redis set. Reads without the PPM were simply executed on MongoDB. Visibly, the persistence mediator provides a consistent latency under 10 ms up to 15,000 reads on the sorted set per second. Note that the persistence mediator would still route reads on other fields or the complete object to MongoDB (since we only annotated a specific attribute).

Figure 3c demonstrates the same setup in a write-only scenario. Comparing desired and actual throughputs, we can see that MongoDB can serve up to 12,000 writes per second while the persistence mediator reaches 25,000. Hence, employing the PPM, we used MongoDB as our default storage, but doubled write-throughput for a common scenario.

5 Future Work

Scoring and Database selection. The ability of a cloud provider to maintain its SLA guarantees heavily depends on the scoring. To this end, it is crucial for the provider to select database configurations that indeed fulfill the requirements, i.e., consistently achieve high scores. One line of further research therefore is the estimation of future scores based on historic metrics. The scoring could then be adapted to prefer selections that have expected high scores in the future. There are many potential statistical and machine learning techniques to achieve this. For instance reinforcement learning (e.g., Q-learning [Dut11]) could be used to learn from past selection decisions based on the utility achieved.

Workload Management and Multi-Tenancy. The persistence mediator could improve performance by actively scheduling requests. Requests that pertain to throughput-oriented annotations can be improved by batching while latency-sensitive requests can be scheduled to experience minimum queuing delays. This workload management has to take place at a per-database-level and depends on the multi-tenancy strategy employed by the mediator to ensure sufficient isolation between tenants.

Polyglot Setups. As a practical question, future work needs to consider which system setups might provide optimal polyglot persistence functionality. For our experiments, Redis was used as an on-demand caching layer. Many other combinations are useful, too. For instance, object stores like S3 can be used to store blob data, while shared-nothing file systems such as HDFS could be used for fields with analytic potential, leveraged through platforms such as Hadoop and Spark. Wide-column stores (e.g., HBase) and table stores (e.g., DynamoDB) can similarly be used to store analyzable, structured, write-heavy data.

6 Conclusion

In this paper, we introduced our Polyglot Persistence Mediator approach. It enables tenants to leverage automated polyglot persistence on a declarative basis using schema-annotations. By defining utility functions or performance indices for non-functional requirements like availability and latency as well as demanded binary properties like object-level atomicity, tenants specify their requirements. The mediator scores available backend databases and selects the optimal database for each part of the tenant's schema and automatically routes data and operations. Exploiting the proposed annotations, providers are free to define which requirements they provide SLAs on and users can employ annotations in an opt-in fashion, maximizing flexibility on both ends. We provided evidence that tremendous performance improvements can be expected and outlined the future challenges for providing a general-purpose Polyglot Persistence Mediator.

References

[Agr09] Agrawal, Divyakant et al. Database management as a service: Challenges and opportunities. In *ICDE*, page 1709–1716. IEEE, 2009.

[Bai12] Bailis, Peter et al. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[Bak11] Baker, J. et al. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, page 223–234, 2011.

[Bas12] Salman A. Baset. Cloud SLAs: Present and Future. *SIGOPS Oper. Syst. Rev.*, 46(2):57–66, July 2012.

[Coo10] Cooper, Brian F et al. Benchmarking cloud serving systems with YCSB. In *SOCC*, page 143–154, 2010.

[Cur11] Curino, Carlo et al. Relational cloud: A database-as-a-service for the cloud. In *Proc. of CIDR*, 2011.

[Dut11] Dutreilh, Xavier et al. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *ICAS*, page 67–74, 2011.

[Ges14] Gessert, Felix et al. ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency. In *CloudDB 2014*, 2014.

[Hac02] Hacigumus, H. et al. Providing database as a service. In *ICDE*, page 29–38, 2002.

[Hun10] Hunt, Patrick et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[LS13] Wolfgang Lehner and Kai-Uwe Sattler. *Web-Scale Data Management for the Cloud*. Springer, New York, auflage: 2013 edition, April 2013.

[OO14] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC 14*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.

[Sob08] Sobel, Will et al. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.