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

Wolfram Wingerath
Baqend GmbH
Stresemannstr. 23
22769 Hamburg, Germany
ww@baqend.com

Felix Gessert
Baqend GmbH
Stresemannstr. 23
22769 Hamburg, Germany
fg@baqend.com

Norbert Ritter
University of Hamburg
Vogt-Kölln-Strae 30
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. To validate our claims of linear scalability, we further present an experimental evaluation of the InvaliDB prototype that has been serving customers at the Database-as-a-Service company Baqend since July 2017.

PVLDB Reference Format:
Wolfram Wingerath, Felix Gessert, Norbert Ritter. InvaliDB: Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases (Extended). *PVLDB*, 13(12): 3032-3045, 2020.
DOI: https://doi.org/10.14778/3415478.3415532

1. 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 database systems (or short "databases") excel at complex queries over historical data [37], they are inherently pull-based and therefore ill-equipped to push new information to clients [67]. Systems for managing data streams, on the other hand, are natively push-oriented and thus facilitate reactive behavior [35]. 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 [77].

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

1.1 Real-Time Databases: Open Challenges

Numerous system designs have been proposed to provide collection-based semantics for pull-based and push-based queries alike: Subsumed under the term "real-time databases"[1] [69] [81], systems like Firebase, Meteor, and RethinkDB provide push-based real-time queries that do not only deliver a single result upon request, but also 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.

In concept, real-time databases thus extend traditional database systems 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. 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 gained over decades of development. To date, every push-based real-time query mechanism fails in at least one of the following challenges:

C_1 Scalability. Serving real-time queries is a resource-intensive process which requires continuous monitor-

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415532

[1] 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 [60] [14] [27]; we do not share this notion of real-time databases.

ing 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 server 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 become intractable as soon as one of the nodes is not able to keep up with processing the entire write stream.

C_2 Expressiveness. The majority of real-time query APIs are limited in comparison to their ad hoc counterparts. Even relatively simple queries are often unsupported or have severe restrictions; for example, some implementations do not offer filter composition with `AND`/`OR`, do not allow ordering by multiple attributes, or support limit, but no offset clauses. 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 do not follow standards regarding data model or query language, 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 may be acceptable in the development of new applications, it complicates the adoption of push-based queries for existing software.

We argue that neither of these limitations is inherent to the challenge of providing push-based real-time queries over database collections. To prove our point, we present the design and implementation of a system for real-time queries that is built on top of the pull-based NoSQL database MongoDB, supports sorted filter queries over single collections for pull- and push-based execution alike, and scales linearly with reads and writes.

2. CONTRIBUTIONS & OUTLINE

Our contributions in this paper are threefold. First in Section 3, we present a discussion of push-based query mechanisms in modern data management to expound why real-time databases deserve distinction in a separate system class next to pull-based database and push-based data stream management systems. By identifying the limitations of the current real-time query mechanisms, we derive important insights for the design of our own real-time database system and further motivate our work. As our second contribution in Section 5, we propose the real-time database design InvaliDB that solves challenges C_1, C_2, and C_3 through a unique combination of characteristics: 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. As our third contribution in Sections 6 and 7, we present an experimental evaluation of our InvaliDB prototype that has been used in production as part of the Database-as-a-Service (DBaaS) offering at Baqend [9] since July 2017. Our experiments confirm that InvaliDB's performance scales *linearly* with the number of matching nodes, both in terms of sustainable write throughput and the number of concurrent real-time queries. The results further indicate that our InvaliDB implementation exhibits consistently low latency even under high per-node load, irrespective of the number of nodes employed for query matching. We discuss our findings in Section 8 and conclude in Section 9.

3. RELATED WORK: PUSH-BASED ACCESS IN DATA MANAGEMENT

Given the trend towards reactivity in modern applications, data management systems of all classes have come to provide push-based data access in one form or another. In Table 1, we delineate real-time databases from traditional databases on the one side and systems for stream management and processing on the other.

Table 1: An overview over data access in data management.

	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

Traditional (SQL) **database management** systems are optimized for expressive random access queries over persistent data collections. They 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 offer active features beyond *triggers* [28] or *event-condition-action (ECA) rules* [65] and existing functionality for query result maintenance is almost exclusively employed for optimizing pull-based query performance (e.g. materialized views [23] [36] or change notification mechanisms [56] [51] [59]).

To warrant low-latency updates in quickly evolving domains, systems for **data stream management** [35] [73] 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. Systems for data stream management are thus natively push-based and generate new output whenever new data becomes available. Through *complex event processing (CEP)* [24], some systems even extract information not explicitly encoded in the data items themselves, but rather in the context between them. Even for sophisticated query engines, though, 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. In consequence, queries over streams are confined to data arriving after query activation and querying data that is rooted in the past is inefficient or impossible without a second system for persistent data management.

Unlike data stream management systems that are mostly intended for analyzing structured information through

declarative query languages, systems for **stream processing** [73] expose generic and imperative 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. Some systems do provide declarative query interfaces and even continuous result updates, but they typically follow stream-based rather than collection-based query semantics (e.g. Flink's dynamic tables [39]) and are very heavyweight, since every query is deployed as a separate application in the distributed computing cluster (cf. Spark's structured streaming [52]). Serving many concurrent users in ad hoc fashion is therefore simply not feasible.

Real-time databases combine principles from both database and data stream management as they evaluate queries over historical data repositories, but also continuously update the query results. 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. In contrast to stream-based queries, however, real-time queries are formulated as though they were evaluated on static data collections, even though they deliver a continuous stream of updates to the query result.

3.1 Real-Time Query Mechanisms

Different approaches are used in practice for maintaining query results within real-time databases. As implied by the name, **poll-and-diff** relies on reevaluating a database query periodically ("poll") and comparing the newly obtained result against the last-known result ("diff") in order to compute result changes which can be sent to the subscribed clients. To the best of our knowledge, the first and only real-time database to implement this approach is Meteor [50] which uses MongoDB [54] as its internal data storage. As a great advantage, poll-and-diff inherits the query expressiveness of the underlying database for real-time queries by using it for query execution. However, it also suffers from potential staleness only bounded by the polling interval (default in Meteor: 10 seconds). But even for applications that can tolerate multi-second lags, poll-and-diff becomes infeasible when many real-time queries are active concurrently, because each one of them induces processing overhead on the database system through frequent reexecution. In numbers, 1 000 active real-time query subscriptions in Meteor result in an average of 100 pull-based queries per second executed against the underlying database system. For each of these queries, a result has to be (1) assembled and (2) serialized by the database, then (3) be sent to the Meteor server where it is (4) deserialized again, so that it can finally (5) be analyzed for relevant changes. While this may be tractable in some cases, it quickly gets prohibitive when results are large or when the database is under high load already (e.g. due to high write throughput).

Seeing the considerable downsides of poll-and-diff, many systems employ an alternative approach with a different set of trade-offs: **Log tailing** is the current default mechanism for real-time queries in Meteor and variants of it are used for real-time queries in Parse [7] (based on MongoDB as well) and RethinkDB [1] (custom-built database with MongoDB-like capabilities). The idea behind log tailing is to use the underlying database's replication log for change discovery: To make sure that no relevant write operation is missed, every application server subscribes to the complete database change log, computes result changes, and pushes them to subscribed clients. Since changes are immediately propagated in this setup, log tailing eliminates the staleness inherent to poll-and-diff and makes periodic polling obsolete. At the same time, though, it introduces the application server as a bottleneck for writes, since each application server has to keep up with the combined throughput of *all database partitions*. As a consequence, the underlying database can be partitioned to scale with write throughput, but change monitoring within the application server cannot.

Table 2: A direct comparison of the collection-based real-time query implementations detailed in this paper.

	Poll-and-Diff		**Log Tailing**		**Firebase / Firestore**	**InvaliDB**
	Meteor		RethinkDB	Parse		Baqend
Scales With Write TP	✓	✗	✗	✗	✗	✓
Scales With # Queries	✗	✓	✓	✓	○ (100k connections)	✓
Lag-Free Notifications	✗	✓	✓	✓	✓	✓
Composition (`AND`/`OR`)	✓	✓	✓	✓	○ (no `OR` in Firebase)	✓
Ordering	✓	✓	✓	✗	○ (single attribute)	✓
Limit	✓	✓	✓	✗	✓	✓
Offset	✓	✓	✗	✗	○ (value-based)	✓

Some proprietary systems use undisclosed mechanisms for query result maintenance. Prominently, **Firebase** and its successor **Firestore** [6] are Google services for developing web and mobile applications with real-time query functionality. Even though Firestore provides slightly more advanced querying capabilities compared to the original Firebase [25] [29] (e.g. chaining filter conditions through a logical `AND`), both services only support simplistic queries (e.g. there is no logical `OR` [32]). Firebase only supports sorting by a single attribute, whereas Firestore allows multi-attribute sorting for certain (albeit not all) query types [31]. Neither service supports regex expressions or comparable content-based filters [32]. In consequence, denormalizing the data model or evaluating queries in the client is often the only way to compensate the lack of expressiveness [40] [63] [20].

Practical Considerations. Under log tailing, high write throughput is known to cause extreme CPU load [66] and load spikes reportedly saturate and even take down the entire application [47] [79]. To address this particular issue, Meteor uses poll-and-diff as a fallback strategy whenever log tailing becomes infeasible [49], but poll-and-diff becomes infeasible as well when there are more than a few active real-time queries [41]. Google's real-time database services similarly bar themselves from write-heavy applications: Firebase allows at most 100 000 parallel client connections and 1 000 writes/s across the entire data set [29], while Firestore maxes out at only 500 writes/s per collection and even only 1 write/s per document [30]. We refer to our data management survey in [78] for details.

Direct Comparison. Table 2 sums up the query capabilities of the approaches described above, 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 log tailing scales with the number of concurrent real-time queries, but

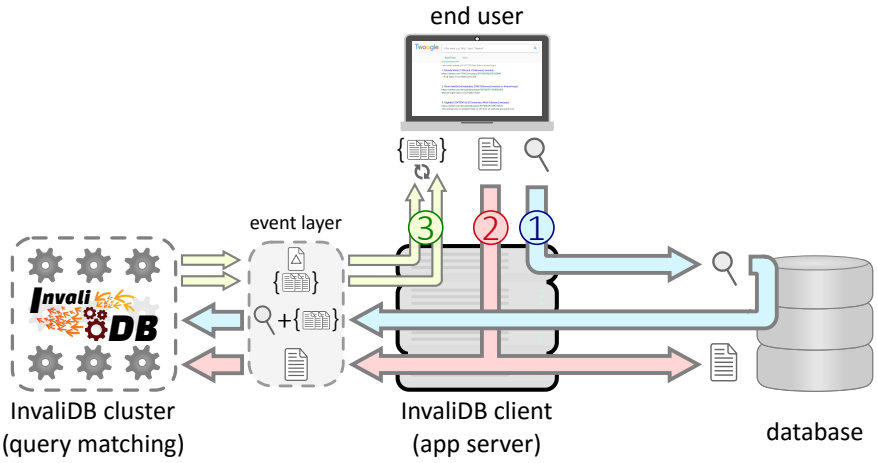


Figure 1: InvaliDB separates responsibilities for data storage (database) from real-time query matching (InvaliDB cluster). The InvaliDB client is located at the application server and acts as a broker between these two and subscribed clients.

neither scales with both. RethinkDB and Parse provide real-time queries with log tailing as well 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 stacks behind Firebase and Firestore are not disclosed, hard scalability limits for write throughput and parallel client connections are documented. 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. Even the more advanced Firestore queries lack support for disjunction of filter expressions (`OR`) and only provide limited options for filter conjunction (`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 supports limit (but no offset) [48], and Parse does not support ordered real-time queries at all [70].

In summary, 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 InvaliDB proposed in this paper, we show that expressive real-time queries and scalability can go hand-in-hand.

4. PRIOR WORK & FURTHER READING

To provide pointers for additional reading and to avoid the allegation of self-plagiarism, we want to make explicit where this paper draws content from and in what ways it relates to our earlier written work. First, this paper extends the ICDE 2020 poster abstract in which InvaliDB was initially presented [76]. All sections of this paper contain revised material from the PhD thesis [72] in the context of which InvaliDB has been developed and which has spawned several other publications on related topics. The introduction in Section 1, the related work part in Section 3, the description of our MongoDB-based real-time query engine in Section 5.4, the discussion in Section 8, and the conclusion in Section 9 contain revised material from tutorial and demo abstracts [74] [75] [77]. We present a vastly extended version of our related work (cf. Section 3) in [78]. Our VLDB 2017 industry paper [33] further details how InvaliDB is used at Baqend to enable a consistent query caching scheme that improves throughput and latency for common pull-based database queries by more than an order of magnitude. While the VLDB paper addresses our approach to improving query performance of a traditional *pull*-based database, this paper explores how we also provide *push*-based real-time queries on top.

5. INVALIDB: A SCALABLE SYSTEM DESIGN FOR REAL-TIME 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 [33] for consistent caching of query results, InvaliDB is used to invalidate cached database queries once they become stale, i.e. it detects result changes and purges the corresponding result caches in timely fashion.

Similar to some of the systems discussed above (e.g. Meteor and Parse), InvaliDB relies on a pull-based database system for data storage. Client applications only interact with the application servers that execute writes as well as pull- and push-based queries on their behalf (see [75] for details on the query interface that unifies pull- and push-based query execution). 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 incoming writes, on the other hand, is offloaded to the InvaliDB cluster. By thus decoupling the real-time query workload from the main application logic, even overburdening the real-time component cannot take down the OLTP system: In the worst-case scenario, the InvaliDB cluster is taken down and requests sent against the event layer remain unanswered.

Figure 1 sketches the message flow between end user, application servers, database, and the InvaliDB cluster. Even

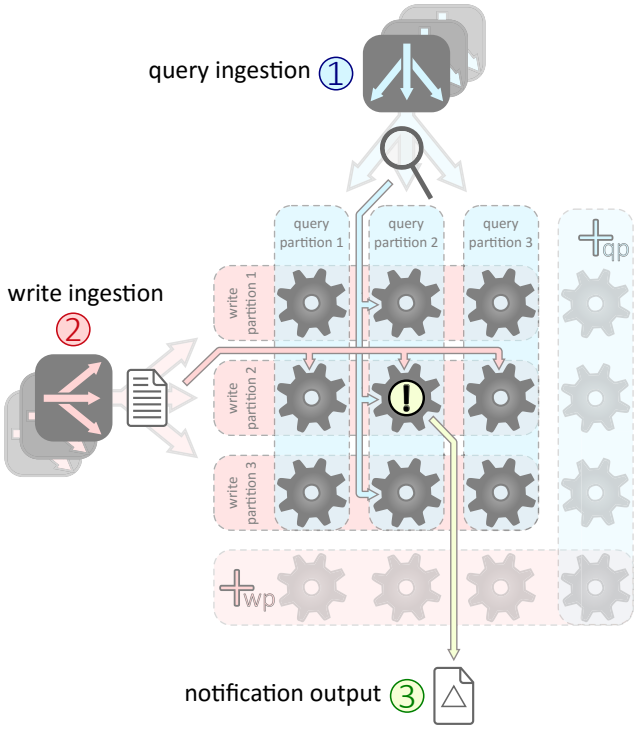


Figure 2: InvaliDB partitions both queries and writes, so that any given matching node is only responsible for matching few queries against some of the incoming writes.

though only a single application server is illustrated, it should be noted that InvaliDB is multi-tenant, so that an InvaliDB cluster can serve many application servers at the same time. In order to subscribe to a real-time query, a web or mobile application sends a **subscription** (1) 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 and sends out the initial result (see below). 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** (not illustrated) is asynchronously passed to the InvaliDB cluster, so that the given query can be deactivated and does not consume further resources. Conversely, TTL extension requests are periodically issued by the application server to avoid expiration of still-active queries. For every **write** (2) 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** (3) 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), `remove` (item left the result). The first notification message for any real-time query contains the initial result that 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 writes delayed or skewed and change notifications may be generated out-of-order (compared with the order in which the corresponding writes arrived at the database). While real-time query results may thus diverge temporarily from database state, they are **eventually consistent** [17] in the sense that they synchronize once InvaliDB has applied the same write operations as the database.

5.1 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 2 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 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

[2]The client generates a unique subscription 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.

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 related 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, the hash value is therefore computed 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, an application server remembers every hash value for the entire lifetime of a subscription and attaches it to every subsequent[3] request relating to the same subscription.

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 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, write stream retention time therefore needs to be chosen according to actually observed delays. For reference, the production deployment at Baqend enforces a retention time of few seconds, since our InvaliDB prototype exhibits consistent end-to-end notification latencies in the realm of few milliseconds with subsecond peaks (see Section 6). During normal operation, InvaliDB thus provides subsecond **data freshness** consistently, which is bounded by a configurable heartbeat interval: In the absence of heartbeat messages, an application server terminates an affected subscription with an error that can be handled by the subscribed clients (e.g. by re-subscribing to the real-time query or falling back to pull-based queries). 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*. Since write operations are versioned, the after-image associated with an insert or update operation for a specific item can thus be ignored whenever a delete (or more recent version) for the same item has already been received.

[3]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.

5.2 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. 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 (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 more advanced queries is performed in loosely coupled **processing stages** that can be scaled independently (cf. staged event-driven architecture (SEDA) [71]). The first processing stage for any query is therefore the **filtering stage** as described in Section 5.1. It is the only processing stage to ingest after-images; all subsequent processing stages 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 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 writes (similar to existing approaches for efficient view maintenance, e.g. [19] [18] [46] [26]).

Since query results are partitioned in the filtering stage, it directly serves queries that can be maintained without coordination between result partitions; this is 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. Through the filtering and sorting stages alone, InvaliDB gains the query expressiveness of aggregate-oriented document stores such as MongoDB which subsumes the expressiveness of state-of-the-art real-time databases (cf. Section 3). Adding support for joins or aggregations through additional processing stages is conceivable and planned for future work (cf. Section 8.1). We discuss the challenges and trade-offs introduced by these additional stages in [72, Ch. 3].

Sorted Filter Queries. 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,

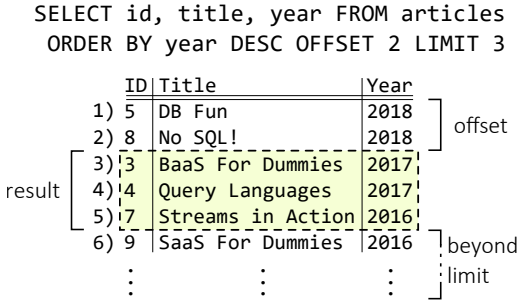


Figure 3: Knowledge of the items in the query's offset and beyond the specified limit is critical to enable incremental maintenance of a sorted query's result.

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 unsorted filter queries over single collections inherently **self-maintainable** [61], i.e. their results can be kept up-to-date by only considering incoming writes. This is not the case for explicitly *sorted filter queries*, because their change notifications also reflect result permutations; 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[4].

Figure 3 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 (i.e. out of the result). To detect updates and deletes of items in the offset, the matching node responsible for a sorted query needs to be aware of all items in the 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 [80], 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. 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 the (rewritten[5]) 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. 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**. For details, see [72, Sec. 3.3.2].

5.3 Implementation

While we use SQL for illustration in this paper, InvaliDB is a **database-agnostic design** and most of its components are generic enough to be shared between implementations for different databases. The event layer abstracts from the query language and data format as it handles data transmissions with entirely *opaque* payloads; routing and partitioning only rely on primary keys (write operations) and the server-generated query identifiers (change notifications, query subscriptions, etc.). Likewise, the two-dimensional workload distribution scheme is not based on any specific technology or database language; it only prescribes an abstract mechanism to ensure that all queries are matched against all incoming after-images. Finally, even the way that match types are derived from the matching status abstracts from specificities of the underlying data store: Whenever an incoming after-image matches a query, 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.

[4]In addition, the sorting key must be unambiguous to ensure that InvaliDB's real-time query engine and the pull-based database engine are aligned. Our prototype therefore adds the primary key as final attribute to the sorting key.

[5]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.

There are only two aspects of real-time query maintenance that contain database-specific artifacts. First, the application server might have to be adapted, e.g. for delivering fully specified and versioned after-images to InvaliDB on every write (see beginning of Section 5) or for rewriting queries (see discussion of sorted queries with non-empty offset on page 6). Second, the **pluggable query engine** is tailored to the underlying database's query engine, since both query engines have to produce the same output, given the same input of queries and writes. In more detail, the pluggable query engine 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.

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.

5.4 Prototype & Production Deployment

Our InvaliDB prototype is built with the distributed stream processor Storm [68] (workload distribution) and the in-memory store Redis [8] (event layer) to provide real-time queries on top of the NoSQL database MongoDB [54] with sharded collections. We monitor query latencies and execution times to ensure the pull-based part of our architecture does not become a bottleneck for providing initial query results. For details on pull-based query performance, see [33].

Workload Distribution. For managing the distributed execution of our query matching workload, we 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 [57]), had already been abandoned (e.g. Muppet [45]), or were not publicly available when we started development in 2014 (e.g. Heron [44], Apex [2], Kafka Streams [42], Wallaroo [55]). We also did not consider systems without on-premise deployment option (e.g. Google's MillWheel [10] and Photon [13] or the Dataflow cloud service [11], and Facebook's Puma and Stylus [21]). Given the requirement for low latency, the viable choices for the underlying stream processor were thus narrowed down to either Storm [68] or Flink [12]; by concept, Samza [62] and Spark Streaming [82] cannot provide competitive latency [73]. 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 (cf. [22]).

Event Layer. When choosing the underlying messaging system for our event layer implementation, we excluded software libraries without built-in standalone server (e.g. ZeroMQ [38] [5]) and systems that were not publicly available when we started development in 2014 (e.g. Moquette [3]). To minimize overall change notification latency for our InvaliDB implementation, we chose Redis over system alternatives with the ability to retain and replay data, even though they would have facilitated easier fault recovery (e.g RabbitMQ [4], ActiveMQ [15], Qpid [16], Kafka [43]).

MongoDB-Compatible Query Engine. The real-time query engine is custom-built in Java and supports sorted MongoDB queries with limit and offset. It can handle arbitrarily nested JSON documents and it supports query operators for content-based filtering through regular expressions (`$regex`), comparisons (e.g. `$eq`, `$ne`, `$gt`, `$gte`), logical combination of filter expressions (e.g. `$and`, `$or`, `$not`), evaluating matching conditions over array values (e.g. `$in`, `$elemMatch`, `$all`, `$size`), full-text search (`$text`), geo queries (e.g. `$geoWithin`, `$nearSphere`), and various others (e.g. `$exists`, `$mod`). As of writing, *full-text search* [64] and *geo queries* [58] have not been rolled out into production.

Application Server. To retrieve after-images on every write, our implementation uses MongoDB's `findAndModify` [53] operation for inserts and updates, because it directly returns the after-images which are then simply forwarded to the InvaliDB cluster; the after-image of a deleted entity is `null` and therefore does not have to be retrieved from the database. The application server initializes every record with a version number and increments it on every write.

Please refer to the InvaliDB PhD thesis [72, Ch. 4] for a discussion of fault tolerance under different error scenarios, an experimental evaluation of event layer scalability, multi-query optimizations as well as computational complexity of our real-time query engine, and Java Virtual Machine (JVM) tuning that was necessary to alleviate the impact and maximize the predictability of stop-the-world garbage collection (GC) pauses [34] in our Java-based software stack.

6. INVALIDB CLUSTER PERFORMANCE

To quantify the scalability of our InvaliDB prototype, we measured change notification latency and sustainable matching throughput for InvaliDB clusters under different workloads and configurations. Our evaluation focuses on the scalability of the filtering stage which implements InvaliDB's unique two-dimensional workload distribution scheme: While the filtering stage processes all queries and all write operations, the subsequent sorting stage only has to cope with filtering stage output partitioned by query, so that it does not become a bottleneck in practice before the subscribed client application does: In the web-centered use cases that motivate our work, subscriptions typically revolve around UI updates in websites or mobile apps and clients are significantly less powerful than InvaliDB's matching nodes.

6.1 Experiment 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 2.1GHz, 64 GB RAM), connected over a 1Gbit/s LAN. Our experimental setup consisted of the following virtualized 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 (event layer), and a Storm 1.1.0 cluster (1 vCPU per node) running InvaliDB. Please note that the event layer (Redis) did not become a bottleneck. While the number of InvaliDB nodes employed for query matching varied with the experiments ($n \in \{1, 2, 4, 8, 16\}$), there were always 4 nodes for write ingestion and 1 single node for query ingestion. By choosing the same number of data ingestion nodes for all experiments, we ensured that workload partitioning was the only difference between InvaliDB configurations.

Given our limited resources, we had to deploy large InvaliDB clusters with relatively many matching nodes per server which led to CPU contention between matching nodes

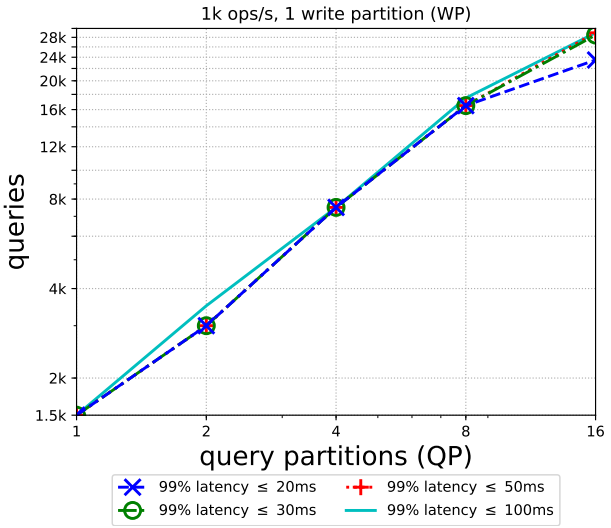


Figure 4: Read scalability: the number of serviceable real-time queries by the number of query partitions (both in logarithmic scale) at 1 000 ops/s under different SLAs.

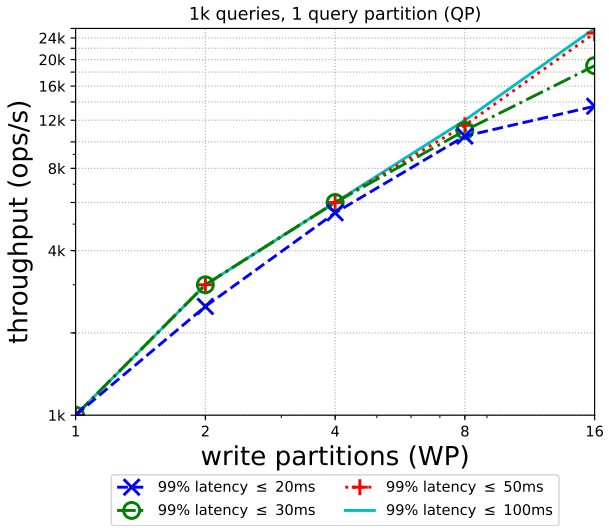


Figure 5: Write scalability: sustainable write throughput by the number of write partitions (both in logarithmic scale) serving 1 000 active real-time queries under different SLAs.

and the underlying virtualization stack. To alleviate this effect, we throttled InvaliDB matching nodes in all experiments to 80% of physical single-core CPU time.

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 inserts 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; the measured end-to-end latency values thus subsumed both processing times in the InvaliDB cluster and message propagation delays through the event layer. We increased the workload in each experiment series until 99$^{\text{th}}$ percentile latency exceeded a given threshold (**latency SLA**). This marks system saturation as it indicates that incoming operations cannot be worked off immediately and therefore start queueing up, causing latency spikes and even subscription failures (when delays are so high that heartbeat messages time out).

Each written document had five 10-literal string attributes and five integer attributes, one of which was a unique random number. The queries were defined with comparison predicates on the random number field, corresponding to the following SQL query: `SELECT * FROM test WHERE random` $\geq$ `i AND random` $<$ `j`. To minimize (de-)serialization overhead for change notifications, we made sure only 1 000 of the queries would match exactly one written item each. Thus, we achieved a steady notification throughput of roughly 17 matches per second (1 000 per 1-minute experiment). Since queries were added during the preparation phase, the number of active queries remained constant for the duration of each experiment, so that the matching nodes were exclusively occupied with matching queries against incoming writes, but not with adding or canceling query subscriptions.

6.2 Linear Read Scalability

Figure 4 shows the number of concurrently sustainable real-time queries for clusters with 1, 2, 4, 8, and 16 query partitions (QP) and a single write partition (WP) under different SLAs, at a fixed write throughput of 1 000 operations per second (note logarithmic scale). 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. around 1 800 queries per node). Since doubling the number of query partitions essentially doubled the number of sustainable queries every time, our measurements thus confirm that InvaliDB scales linearly with query load. As the only exception, the largest InvaliDB cluster merely supported 23 500 concurrent queries with 99$^{\text{th}}$ percentile latency below 20ms, but more than 28 500 under all other SLAs. We attribute this anomaly to the CPU contention issue described in Section 6.1, because it made change notification latency unstable when physical servers (virtualization hosts) approached capacity.

6.3 Linear Write Scalability

Figure 5 depicts sustainable throughput for InvaliDB clusters with a single query partition and 1 to 16 write partitions, under a fixed read workload of 1 000 active real-time queries and varying SLAs (note 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 implementation achieved lower overall matching performance under the write-heavy workloads than under the read-heavy workloads discussed above. In more detail, the 99$^{\text{th}}$ percentile latencies under increasing write throughput were measured below 30ms until about 75% and below 50ms until about 95% of system capacity, while latencies under the read-heavy workloads did not exceed 30ms before reaching the performance limit. Similarly, sustainable write throughput 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 number of nodes under the read-heavy workload. This effect is caused by the overhead for (de-)serial-

Table 3: Measured latency in milliseconds (average, standard deviation, 99th percentile, max.) for different InvaliDB clusters.

(a) Read-heavy workloads at 1 000 ops/s (fixed): 1 500 queries per query partition (about 80% of system capacity).

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

(b) Write-heavy workloads with 1 000 real-time queries (fixed): 1 000 ops/s per write partition (about 66% of system capacity).

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

izing and parsing after-images which grows with increasing write throughput; since queries were activated before the experimental measurement phase, no comparable effect could be observed during the read scalability experiments.

6.4 Consistently Low Latency

To illustrate that InvaliDB's latency characteristics are stable across all configurations, we compare notification latency for different InvaliDB cluster sizes under identical *relative* load in Table 3. At about 80% of system capacity under the read-heavy workload (a), notification latency was measured between 9.0ms and 9.4ms with standard deviations ranging from 2.4ms to 3.4ms, while outliers never exceeded 50ms. At roughly two thirds of system capacity under the write-heavy workload (b), the average notification latencies ranged from 8.8ms to 10.3ms with standard deviations between 2.3ms and 3.5ms and outliers that were always well below 100ms. The largest InvaliDB cluster experienced some outliers and therefore slightly deteriorated latency in average, standard deviation, and 99th percentile compared to the clusters with fewer nodes. We think there are two likely explanations for this occurrence. As one possible explanation, CPU contention incurred by our test setup might have affected matching node performance for the largest InvaliDB configurations, but not the smaller ones (cf. Section 6.1). Another possibility is that garbage collection in the write ingestion nodes could have caused occasional latency stragglers at high throughput that were insignificant under the write workload during the read scalability experiments.

7. QUAESTOR SERVER PERFORMANCE

Within the Quaestor architecture (cf. Section 5), InvaliDB is used in two different ways: First, InvaliDB makes **query result caching** feasible by providing low-latency invalidation messages for stale query results. In our VLDB 2017 paper [33], we already demonstrated that this InvaliDB-enabled query caching scheme can improve throughput and latency of pull-based queries by more than an order of magnitude. As the second and primary use case at Baqend, however, 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 3.1), result maintenance is delegated to the InvaliDB cluster and the application server is thereby removed as a bottleneck.

In this section, we demonstrate that InvaliDB's opt-in result maintenance is very lightweight and enables high write throughput, many concurrent real-time queries, and low double-digit latencies even for single-server deployments.

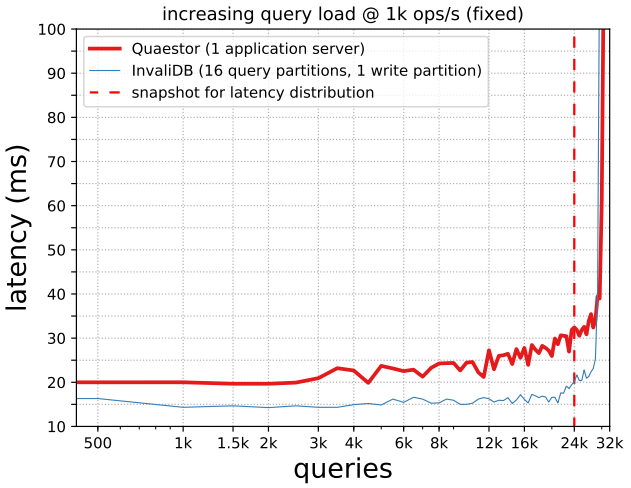
7.1 Experimental Scope

In our architecture, clients subscribe to real-time queries at 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. InvaliDB's scalability with read and write workloads is evident from the experimental results presented in Section 6. Additional application servers can always be spawned to take care of further client subscriptions, because different real-time query subscriptions are managed independently from one another. This characteristic follows from our system design and has been observed by us in production. Due to space limitations, we therefore restrict the performance evaluation in this section to a quantification of the latency overhead and throughput limitations of deployments with a single application server. To this end, we measure change notification latency exposed to Quaestor clients and compare it to the latency achieved by a standalone InvaliDB deployment (i.e. without application server).

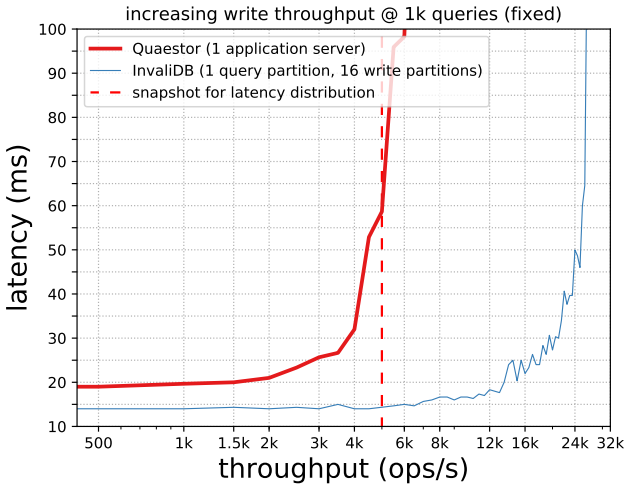
7.2 Setup & Workload

We conducted our experiments using the same basic setup as described in Section 6 (standalone InvaliDB deployment), but we added an 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 application server (and not with the event layer directly), like an end user (see Figure 1 on page 4). Compared to the experiments from Section 6, we thus effectively introduced an additional network hop for all messages sent between the benchmark client and the InvaliDB cluster. For testing Quaestor's real-time query performance under read- and write-heavy workloads, 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.

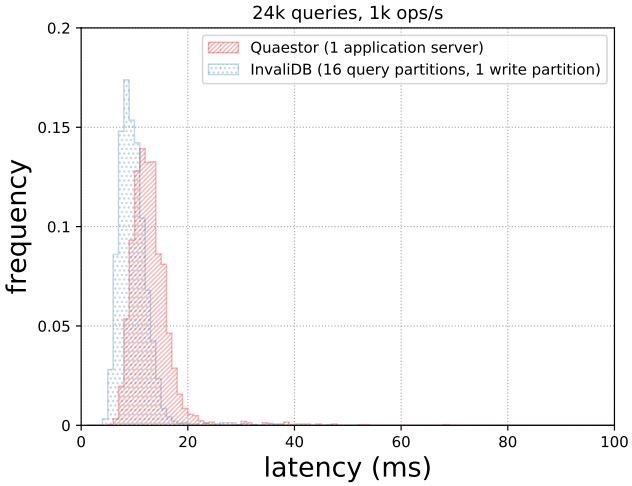
To put the overhead and limitations of the Quaestor architecture into perspective, we contrast Quaestor's latency and throughput characteristics with InvaliDB performance measurements from Section 6, using identical workloads: The benchmarking client inserted data items at a fixed rate for one minute, measuring change notification latency for each received change event. Notification latency was again measured as the time from right before inserting an item until after receiving the corresponding event. We also registered all real-time queries before the measurement phase and limited the matching items per run to 1 000 ($\approx$ 17 matches/s) to bound messaging overhead, like in earlier experiments.



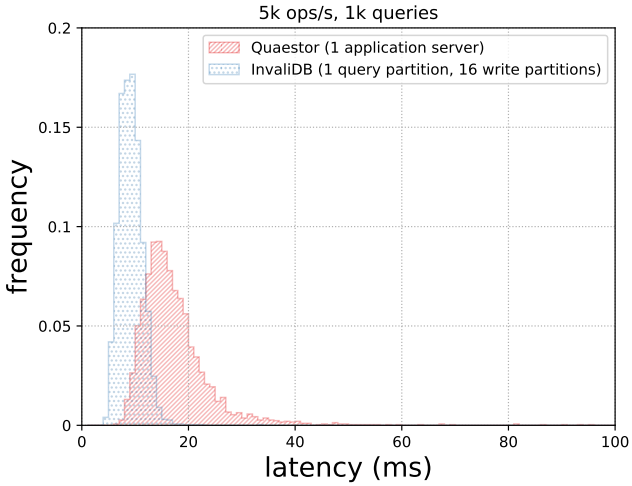
(a) Read scalability: change notification latency under an increasing query load (logarithmic scale) at a constant write throughput of 1 000 writes per second.



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



(c) Read-heavy workload: latency distribution with 24 000 active real-time queries at 1 000 writes per second.



(d) Write-heavy workload: latency distribution with 1 000 active real-time queries at 5 000 writes per second.

Figure 6: Quaestor vs. standalone InvaliDB: change notification latency under read- and write-heavy workloads.

At Baqend, every application server 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. Since the number of actual client connections is thus transparent to the application server, it is not relevant in the context of this evaluation. We therefore used a single WebSocket connection between application server and our benchmarking client for all real-time query subscriptions.

7.3 Low Application Server Overhead

Figure 6 shows our results of comparing Quaestor's with InvaliDB's real-time query performance. As is evident from the line plot of 99$^{\text{th}}$ percentile latency during the read-heavy workload (a), Quaestor essentially adds a fixed overhead of about 5ms 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. Counterintuitively, though, the Quaestor deployment was able to support slightly more queries than the InvaliDB-only deployment during the read-heavy workload in our experiments as shown in Figure 6a. We would like to point out that this is no measurement error, but a consequence of pushing InvaliDB to the performance limit: Near system capacity, (i.e. around 30 000 concurrent real-time queries in these experiments), 99$^{\text{th}}$ percentile latency becomes unstable and peak performance therefore varies slightly between experiments.

The corresponding line plot for the write-heavy workload (b) shows that write throughput is limited by Quaestor's application server at roughly 6 000 operations per second[6]. For comparison, write throughput on a single collection is capped at 500 and 1 000 writes per second in Firestore and Firebase, respectively (cf. Section 3). Without even scaling out, our experiment deployment thus outperformed Google's commercial real-time database offerings by factors 6 to 12.

[6]An application server without the real-time query feature does not provide significantly higher peak throughput. This illustrates that our mechanism imposes very little overhead.

A comparison of the latency distributions from the read-heavy (c) and the write-heavy (d) workloads at roughly 80% capacity further illustrates our mechanism's low overhead regarding notification latency: During the read-heavy workload, Quaestor's latency distribution is shifted to the right by about 5ms 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 and remains consistently below 100ms even near full capacity.

We did not include measurements of the impact of query subscription, because it is not limited by InvaliDB. Rather, the time it takes to activate a real-time query (or several real-time queries concurrently) critically depends on the performance of the underlying pull-based storage for fetching the initial query results.

7.4 Result Summary & Interpretation

To sum up, the implementation of our real-time database design exhibits predictable and consistently low latency, even under demanding OLTP workloads. The presented experimental results demonstrate high read and write scalability and further show 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. Since overhead for the application server depends on the number of change events (and not the number of concurrent query subscriptions), serving many users at the same time is very efficient. At the same time, even a single-server deployment can handle write workloads beyond documented peak throughput of competing commercial offerings.

8. DISCUSSION & OUTLOOK

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 and evaluate a novel real-time database design that adds push-based real-time queries as an opt-in feature to existing pull-based databases, without the limitations of other state-of-the-art systems. The fact that InvaliDB supports the query expressiveness of NoSQL (document) stores such as MongoDB illustrates that our design can service a wide range of use cases. The presented experimental results further confirm that our Java-based InvaliDB prototype scales linearly with write throughput and with the number of concurrent users (i.e. query subscriptions). Across all cluster sizes and under various read- and write-heavy workloads, our implementation displayed 99^{th} percentile latencies consistently below 30ms when under moderate load and still within 100ms when approaching system capacity. Our experiments thus show that InvaliDB is feasible to implement and can be efficiently operated.

To provide additional evidence for InvaliDB's practicality, the prototype described in this paper has been used in production at the company Baqend since July 2017. Here, it serves two different purposes. First, it enables consistent query caching by generating low-latency result change notifications used for query cache invalidation, thus improving both throughput and latency of the existing pull-based query mechanism by more than an order of magnitude. Second, InvaliDB extends the functionality of the MongoDB-based Database-as-a-Service by adding push-based real-time queries to the otherwise pull-based query interface.

8.1 Open Challenges & Future Work

We firmly believe that our work provides valuable insights for real-time database architects, but we see ample opportunities for follow-up research and development. As a final note, we would like to exemplify open challenges and possible future work with in three areas.

Aggregations & Joins. Our practical work so far has been focused on sorted filter queries over single collections as they are supported by many document-oriented NoSQL databases. Future research could extend our work by additional query types (e.g. aggregation and join queries) or add support for database languages such as SQL. As a related effort, the consolidation of InvaliDB's consistency model with transactional guarantees could be pursued, e.g. by implementing transactional visibility for write operations.

Client Performance. Since we have not considered performance for end user devices in our work so far, future research could also examine InvaliDB's real-time query subscriptions from the consumer perspective, e.g. to develop schemes for saving client resources by compressing messages or by collapsing write operations and change notifications to mitigate write hotspots. This seems particularly important in the context of web and mobile applications, because smartphones and other devices with limited processing power, weak network links, and/or bounded (monthly) data allowance appear to be more likely bottlenecks in certain usage scenarios (e.g. under heavy hitter queries) than the application servers or InvaliDB's real-time query engine.

Application Development. We see a tremendous opportunity for innovation in upgrading existing *pull-based interfaces* with push-based query features, since InvaliDB makes real-time queries available on top of *existing* databases. For example, certain portions of a website could be made interactive to augment the user experience. InvaliDB could also be integrated into current cache coherence schemes to improve efficiency or reduce staleness. Similar to its role within the Quaestor architecture, it could perform asynchronous *change detection* to trigger updates or recomputation of materialized views or other query caches.

9. CONCLUSION

In the past, practitioners have been rightfully cautious in adopting real-time databases, because they have been hard to integrate into existing applications and mostly intractable to use at scale. The system design presented in this paper 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. Second, InvaliDB scales with both the number of concurrent queries and with write throughput. Third and arguably most important for productive settings, it is designed as an opt-in component with an isolated failure domain to make its adoption a low-risk endeavor. In conclusion, 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.

10. REFERENCES

[1] RethinkDB, 2016. https://www.rethinkdb.com/, accessed: 2019-09-21.

[2] Apex, 2018. https://apex.apache.org/, accessed: 2018-08-18.

[3] Moquette, 2018. https://github.com/moquette-io/moquette, accessed: 2018-05-27.

[4] RabbitMQ, 2018. https://www.rabbitmq.com/, accessed: 2018-05-10.

[5] ZeroMQ, 2018. http://zeromq.org/, accessed: 2018-03-26.

[6] Firebase, 2019. https://firebase.google.com/, accessed: 2019-04-15.

[7] Parse, 2019. https://parseplatform.org/, accessed: 2019-09-21.

[8] Redis, 2019. https://redis.io/, accessed: 2019-10-05.

[9] Baqend, 2020. https://www.baqend.com/, accessed: 2020-06-10.

[10] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, Aug. 2013.

[11] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *PVLDB*, 8(12):1792–1803, Aug. 2015.

[12] A. Alexandrov, R. Bergmann, S. Ewen, et al. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 2014.

[13] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, et al. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In *SIGMOD '13*, 2013.

[14] S. F. Andler and J. Hansson, editors. *Active, Real-Time, and Temporal Database Systems*, number 1553 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998.

[15] Apache Software Foundation. ActiveMQ, 2018. https://activemq.apache.org/, accessed: 2018-05-10.

[16] Apache Software Foundation. Qpid, 2018. https://qpid.apache.org/, accessed: 2018-05-10.

[17] P. Bailis and A. Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Queue*, 11(3):20:20–20:32, Mar. 2013.

[18] J. A. Blakeley, N. Coburn, and P.-A. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. Database Syst.*, 14(3):369–400, Sept. 1989.

[19] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[20] P. Bover. Firebase: the great, the meh, and the ugly. *freeCodeCamp Blog*, Jan. 2017. https://www.freecodecamp.org/news/a07252fbcf15/, accessed: 2017-05-21.

[21] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1087–1098, New York, NY, USA, 2016. ACM.

[22] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, et al. Benchmarking Streaming Computation Engines at Yahoo! *Yahoo! Engineering Blog*, Dec. 2015. http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at, accessed: 2016-10-17.

[23] R. Chirkova and J. Yang. Materialized Views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[24] G. Cugola and A. Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[25] 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.

[26] C. Elkan. Independence of Logic Database Queries and Update. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 154–160, New York, NY, USA, 1990. ACM.

[27] 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.

[28] K. P. Eswaran and D. D. Chamberlin. Functional Specifications of a Subsystem for Data Base Integrity. *PVLDB*, pages 48–68, 1975.

[29] Firebase. *Choose a Database: Cloud Firestore or Realtime Database*, Dec. 2017. https://firebase.google.com/docs/firestore/rtdb-vs-firestore, accessed: 2017-12-19.

[30] Firebase. *Firestore: Quotas and Limits*, Dec. 2017. https://firebase.google.com/docs/firestore/quotas, accessed: 2017-12-19.

[31] 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.

[32] Firebase. *Perform Simple and Compound Queries in Cloud Firestore*, Dec. 2017. https://firebase.google.com/docs/firestore/query-data/queries, accessed: 2017-12-19.

[33] F. Gessert, M. Schaarschmidt, W. Wingerath, E. Witt, E. Yoneki, and N. Ritter. Quaestor: Query Web Caching for Database-as-a-Service Providers. *PVLDB*, 10(12):16701681, Aug. 2017.

[34] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. *SIGARCH Comput. Archit. News*, 41(1):229–240, Mar. 2013.

[35] L. Golab and M. T. Zsu. *Data Stream Management*. Morgan & Claypool Publishers, 2010.

[36] A. Gupta and I. S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT press, 1999.

[37] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a Database System. *Found. Trends Databases*, 1(2):141–259, Feb. 2007.

[38] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.

[39] F. Hueske, S. Wang, and X. Jiang. Continuous Queries on Dynamic Tables. *Flink Blog*, Apr. 2017. https://flink.apache.org/news/2017/04/04/dynamic-tables.html, accessed: 2017-10-27.

[40] B. Jamin. Reasons Not to Use Firebase. *Chris Blog*, Sept. 2016. https://crisp.im/blog/why-you-should-never-use-firebase-realtime-database/, accessed: 2017-05-21.

[41] 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.

[42] J. Kreps. Introducing Kafka Streams: Stream Processing Made Simple. *Confluent Blog*, March 2016. http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/, accessed: 2016-09-19.

[43] J. Kreps, N. Narkhede, and J. Rao. Kafka: a Distributed Messaging System for Log Processing. In *NetDB'11*, 2011.

[44] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings*

of the 2015 ACM SIGMOD International Conference on Management of Data, 2015.

[45] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-Style Processing of Fast Data. *PVLDB.*, 5(12):1814–1825, Aug. 2012.

[46] A. Y. Levy and Y. Sagiv. Queries Independent of Updates. *PVLDB*, pages 171–181, 1993.

[47] A. Mao et al. My Experience Hitting Limits on Meteor Performance, 2014. `https://groups.google.com/forum/#!topic/meteor-talk/Y547Hh2z39Y`, accessed: 2017-07-09.

[48] W. Martin. Changefeeds in RethinkDB. *RethinkDB Docs*, 2015. `https://rethinkdb.com/docs/changefeeds/javascript/`, accessed: 2017-07-09.

[49] Meteor Development Group. Livequery. *Meteor Change Log v1.0.4*, Mar. 2015. `http://docs.meteor.com/changelog.html#livequery-1`, accessed: 2017-07-09.

[50] Meteor Development Group. Meteor, 2018. `https://www.meteor.com/`, accessed: 2018-05-10.

[51] Microsoft. *SQL Server 2008 R2 Books Online: Creating a Query for Notification*, 2017. `https://msdn.microsoft.com/en-us/library/ms181122.aspx`, accessed: 2017-05-12.

[52] MongoDB Inc. *Structured Streaming Programming Guide: Caveats*, 2016. `https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#caveats-1`, accessed: 2019-06-15.

[53] MongoDB Inc. *db.collection.findAndModify()*, 2018. `https://docs.mongodb.com/v3.6/reference/method/db.collection.findAndModify/`, accessed: 2018-06-23.

[54] MongoDB Inc., 2019. `https://mongodb.com`, accessed: 2019-09-15.

[55] J. Mumm. How Wallaroo Scales Distributed State. *Wallaroo Labs Blog*, Oct. 2017. accessed: 2017-12-29.

[56] C. Murray, T. Kyte, et al. Using Continuous Query Notification (CQN). In *Oracle Database Development Guide, 12c Release 1 (12.1)*. Oracle, May 2017.

[57] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, 2010.

[58] M. Patzwahl. Inkrementelle Auswertung geobasierter MongoDB-Anfragen. Master's thesis, University of Hamburg, 2018.

[59] The PostgreSQL Global Development Group. *PostgreSQL 9.6 Documentation: Notify*, 2017. `https://www.postgresql.org/docs/9.6/static/sql-notify.html`, accessed: 2017-05-13.

[60] 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.

[61] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-maintainable for Data Warehousing. In *Proc. of the 4th International Conference on Parallel and Distributed Information Systems*, 1996.

[62] N. Ramesh. Apache Samza, LinkedIn's Framework for Stream Processing. *thenewstack.io*, January 2015. `https://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing/`, accessed: 2016-09-21.

[63] A. Rose. Firebase: The Good, Bad, and the Ugly. *Raizlabs Developer Blog*, Dec. 2016. `https://www.raizlabs.com/dev/2016/12/firebase-case-study/`, accessed: 2017-05-21.

[64] R. Schütt. Inkrementelle Auswertung von MongoDB-Volltextsuchanfragen. Master's thesis, University of Hamburg, 2018.

[65] E. Simon, J. Kiernan, and C. d. Maindreville. Implementing High Level Active Rules on Top of a Relational DBMS. *PVLDB*, 1992.

[66] W. Stein. RethinkDB versus PostgreSQL: My Personal Experience. *CoCalc Blog*, Feb. 2017. `https://blog.sagemath.com/2017/02/09/rethinkdb-vs-postgres.html`, accessed: 2017-07-09.

[67] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.

[68] 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.

[69] F. van Puffelen. Have you Met the Realtime Database? *Firebase Blog*, July 2016. accessed: 2017-05-20.

[70] M. Wang. Parse LiveQuery Protocol Specification. *GitHub*, 2016. `https://github.com/parse-community/parse-server`, accessed: 2019-10-05.

[71] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 2001.

[72] W. Wingerath. *Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases*. PhD thesis, University of Hamburg, 2019. `https://invalidb.info/thesis`.

[73] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter. Real-Time Stream Processing for Big Data. *it - Information Technology*, 58(4), 2016.

[74] 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.

[75] 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.

[76] W. Wingerath, F. Gessert, and N. Ritter. InvaliDB: Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases. In *36th IEEE ICDE 2020, Dallas, Texas*, 2020.

[77] 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.

[78] W. Wingerath, N. Ritter, and F. Gessert. *Real-Time & Stream Data Management: Push-Based Data in Research & Practice*. Springer International Publishing, 2019.

[79] D. Workman et al. Large Number of Operations Hangs Server. *Meteor GitHub Issues*, 2014. `https://github.com/meteor/meteor/issues/2668`, accessed: 2016-10-01.

[80] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient Maintenance of Materialized Top-k Views. *Proceedings of the 19th International Conference on Data Engineering*, 2003.

[81] A. Yu. What Does It Mean to Be a Real-Time Database? — Slava Kim at Devshop SF May 2015. *Meteor Blog*, June 2015. `https://blog.meteor.com/aa0b671c8ab5/`, accessed: 2017-05-20.

[82] M. Zaharia, T. Das, H. Li, et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.