B. Mitschang et al. (Hrsg.): BTW 2017 – Workshopband,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 269

# The Case For Change Notifications in Pull-Based Databases

Wolfram Wingerath[1], Felix Gessert[1], Steffen Friedrich[1], Erik Witt[2] and Norbert Ritter[1]

**Abstract:** Modern web applications often require application servers to deliver updates proactively to the client. These push-based architectures, however, are notoriously hard to implement on top of existing infrastructure, because today's databases typically only support pull-based access to data. In this paper, we first illustrate the usefulness of query change notifications and the complexity of providing them. We then describe use cases and discuss state-of-the-art systems that do provide them, before we finally propose a system architecture that offers query change notifications as an opt-in feature for existing pull-based databases. As our proposed architecture distributes computational work across a cluster of machines, we also compare scalable stream processing frameworks that could be used to implement the proposed system design.

**Keywords:** continuous queries, materialized view maintenance, real-time stream processing, Big Data

## 1 Introduction

OLTP databases traditionally only support pull-based access to data, i.e. they return data as a direct response to a client request. This paradigm is a good fit for domains where users work on a common data set, but isolated from one another. A variety of modern (web) applications like messengers or collaborative worksheets, on the other hand, target more interactive settings and are expected to reflect concurrent activity of other users. Ideally, clients would be able to subscribe to complex queries and receive both the initial result as well as result updates (i.e. change notifications) as soon as they happen. But only few OLTP database systems provide real-time capabilities beyond simple triggers and application developers often have to employ workarounds to compensate for the lack of functionality. For example, applications are often modeled in such a way that application servers can filter out relevant changes by monitoring specific keys instead of actually maintaining query results in real-time. This makes it possible to notify co-workers of recent changes in a shared document or invalidating caches for static resources [GBR14], but more complex scenarios that cannot be mapped to monitoring individual keys (e.g. maintaining user-defined search queries) are simply infeasible.

In this paper, we survey currently available systems that do provide query change notifications, discuss strengths and weaknesses of their respective designs and propose an alternative system architecture that offers a unique set of strong points. The envisioned system is built around a scalable stream processing framework and, compared to the current state of the art, provides the following main benefits:

[1] University of Hamburg, Information Systems, Vogt-Kölln-Straße 30, 22527 Hamburg, Germany: wingerath@informatik.uni-hamburg.de, gessert@informatik.uni-hamburg.de, friedrich@informatik.uni-hamburg.de, ritter@informatik.uni-hamburg.de
[2] Baqend GmbH, Vogt-Kölln-Straße 30, Room F-528, 22527 Hamburg, Germany: ew@baqend.com

1. **Opt-In Change Notifications:** Being a standalone system, our proposed architecture provides real-time change notifications as an additional feature to existing DBMSs.
2. **Linear Scalability:** The system is able to scale with the number of continuously maintained queries as well as the update throughput.
3. **Pluggable Query Engine:** Through a pluggable query engine, the approach is not system-specific, but applicable to a variety of different databases.

The rest of this article is structured as follows: In Section 2, we provide an example to illustrate what exactly query change notifications are and why providing them is a non-trivial task. We then explore the three use cases (1) real-time notifications for interactive (web) applications, (2) query result cache invalidation and (3) materialized view maintenance in Section 3 and survey existing systems that provide query change notifications in Section 4. Subsequently, we present our own architecture for opt-in query change notifications and discuss viable candidates for the underlying stream processing framework in Section 5. A conclusion and final thoughts are given in Section 6.

## 2 Problem Statement

In order to enable clients to define their critical data set and keep it in-sync with the server, we argue that clients should be provided with the initial result *and* updates to the result alike. For an illustration of a possible set of notifications, consider Figure 1 that shows a query for NoSQL-related blog posts and a blog post that enters and leaves the result set as it is edited.

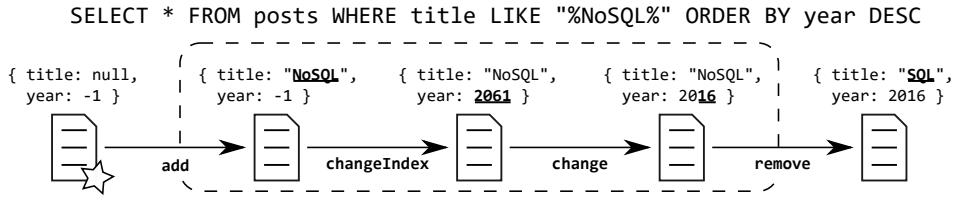


Fig. 1: An example of notifications that occur while a blog post is edited.

Initially, the blog post is created without title and without year and therefore does not match the query. When the author chooses the title to be “NoSQL”, the blog post enters the query result (dashed box) and all query subscribers have to be notified of this event through an `add` notification. Since the publication year is still set to the default value of `-1`, the blog post currently has the last position in the result set. Next, the author intends to set the publication year to the current year, 2016, but accidentally sets it to `2061`. Irrespective of this typo, the blog post moves from the last to the first position in the result set, because it now has the largest `year` value; subscribers receive a `changeIndex` notification and are thus aware of an update to the blog post that changed its position. The author becomes aware of her mistake and subsequently corrects it by setting the year to `2016`. Since there is no other more recent article, the blog post does not change its position (no `changeIndex` notification), but subscribers still receive a `change` notification. After a change of mind, the author updates the title to “SQL” and the blog post correspondingly ceases to match the

query predicate, provoking a `remove` notification. None of the following blog post updates will lead to a notification as long as the matching condition is not satisfied.

Detecting query result changes is significantly more complex than detecting updates on individual data items, because any written object (here: blog post) may or may not satisfy the matching condition of any maintained query. Further, the previous matching status of an object with respect to a query has to be known in order to be able to tell the difference between `add`, `change` and `changeIndex` events and to recognize a `remove` event.

## 3 Use Cases

There are many applications that rely on server-side notifications of state updates. They can be classified as follows:

1. **Notifications for Clients**: The obvious use case for change notification is forwarding them to clients, so that they are informed whenever relevant state changes occur.
2. **Cache Invalidations**: The capability of detecting result changes opens up the possibility of caching dynamic data, namely query results, with minimal staleness windows. Whenever a result is updated (i.e. becomes stale), the corresponding caches can be invalidated. Since queries are not only served by the database itself, but also by caches located near the clients, response times are reduced significantly and read workload is taken off the database.
3. **Materialized Views**: Frequently requested queries can be kept up-to-date in a separate data store to further relieve the primary database. It is even possible to amend the querying capabilities of the primary database to enable access patterns that would otherwise not be available: for example, a materialized MongoDB view could be maintained on top of a key-value store like Riak by loading all data in a bucket initially and subsequently only processing incoming updates.

## 4 State of the Art

The inability of traditional pull-based database systems to cope with streaming data well has been identified as a critical and mostly open challenge years ago [SC05, ScZ05] and the integration of static and streaming data has been studied for decades [BLT86, BW01, MWA+03]. While early prototypes required append-only databases [TGNO92], modern systems also consider updated and removed data and thus target more practical applications.
**Complex Event Processing (CEP) engines** [ACc+03, AAB+05] are software systems specifically designed to derive complex events like a sensor malfunction or an ongoing fraud from low-level events such as individual sensor inputs or login attempts. Queries do not only constrain data properties, but also temporal, local or even causal relationships between events. In contrast to databases that permanently store and subsequently update information, though, CEP engines work on *ephemeral data streams* and only retain derived state such as aggregates in memory for a relatively short amount of time.
**Timeseries databases** [DF14] are specialized to store and query infinite sequences of events as a function of the time at which they occurred, for example sensor data indexed by time. While they store data permanently and some of them do also have continuous

querying capabilities (e.g. InfluxDB[3]), they are typically employed for analytic queries, materialized view maintenance or downsampling streams of information and do not extend to change notifications.

In recent years, new database systems have emerged that aim to provide real-time change notifications in a scalable manner, but they provide only vendor-specific solutions. Existing applications working on a purely pull-based database have to either switch the underlying data storage system to gain real-time change notifications or have to employ workarounds to compensate for the lack of them.

**Meteor**[4] is a web development platform backed by MongoDB that provides real-time query change notifications using two different techniques. In principle, a Meteor server reevaluates every continuous query periodically and compares the last and the current result to detect recent changes. This "poll-and-diff" approach allows a complete coverage of the MongoDB feature set, but also adds latency of several seconds. More importantly, it puts load on the database and the application server for computing, serializing, sending and deserializing query results that is proportional to their size. Whenever possible, Meteor applies a more light-weight strategy called oplog tailing where a Meteor server subscribes to the MongoDB oplog (the replication stream) and tries to extract relevant changes from it. While oplog tailing greatly reduces notification latency and processing overhead, it still requires querying MongoDB when the information provided by the oplog is incomplete. The approach is further hard-limited by the maximum of replica set members allowed by MongoDB [Inc16], is only feasible when overall update throughput is low and prohibits horizontal scaling [Das16, met14]. **Parse**[5] is a development framework with MongoDB-like querying capabilities and change notifications for queries. The involved computation can be distributed across several machines, but is ultimately limited by a single-node Redis instance employed for messaging [Par16b]. Parse's hosted database service is going to shutdown in January 2017 and the number of people contributing to the code base has been decreasing over the last months [Par16a]. Even though other vendors have announced support of the Parse SDK [Gai16], future support for the Parse platform is uncertain. **Oracle 11g** is a distributed SQL database with complex query change notifications that supports streaming joins with certain restrictions [WBL+07]. Materialized views of the continuous queries are maintained by applying committed change operations periodically, on-demand or on transaction commit [M+08]. Due to the strict consistency requirements and the underlying shared-disk architecture, scalability is limited. **PipelineDB**[6] extends PostgreSQL by change notifications for complex queries. While the open-sourced version can only run on a single node, the enterprise version supports a clustering mode that shards continuous views and associated computation across several machines. However, since all write operations (insert, update, delete) are coordinated synchronously via two-phase commit between all nodes [Pip15], PipelineDB Enterprise is only scalable up to moderate cluster sizes. **RethinkDB**[7] is a NoSQL database that does support rich continuous query semantics, but is currently subject to a hard scalability limit [Ret16] and does not provide

[3] https://www.influxdata.com/time-series-platform/influxdb/
[4] https://www.meteor.com/
[5] https://parse.com/
[6] https://www.pipelinedb.com/
[7] https://www.rethinkdb.com/

streaming joins. It is the underlying data store for the Horizon[8] development framework. **Firebase**[9] is a cloud database that delivers notifications for changed data, but provides only limited query expressiveness due to the very restrictive underlying data model that requires information to be organized in a tree of lists and objects.

## 5 Vision: Scalable Opt-In Query Change Notifications

To enable query change notifications on top of systems that by themselves do not provide them, we propose amending these systems by an additional real-time subsystem.

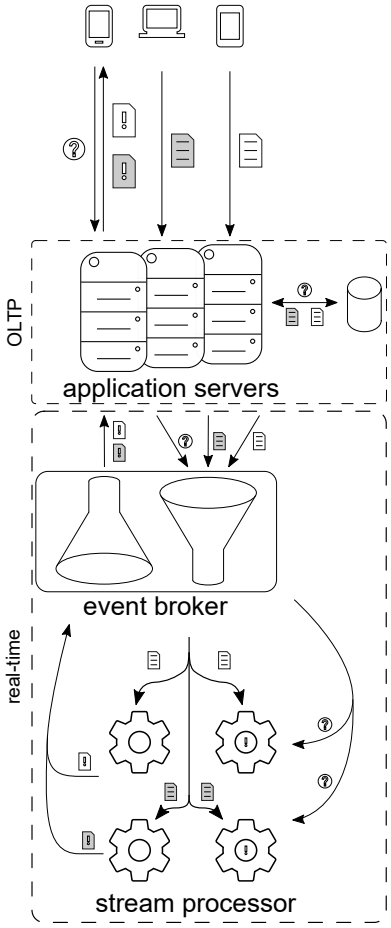


Fig. 2: An architecture that provides opt-in query change notifications on top of purely pull-based databases.

In our proposed architecture as illustrated in Figure 2, common OLTP workloads are still handled by **application servers** that interact with the **database** on behalf of clients. To cope with additional real-time workload, we introduce a new subsystem comprising an **event broker** (i.e. a streaming system like Kafka) to buffer data between application servers and a scalable **stream processor** (e.g. Storm) that maintains continuous queries and generates notifications whenever results change.

To make continuous query maintenance stand on its own, the application server has to provide the real-time subsystem with all required information, namely initial query results and complete data objects on every write. To this end, each continuous query is evaluated once upfront and then sent to the event broker along with all matching objects. Every write operation, i.e. each insert, update and delete, is sent to the event broker together with a complete after-image of the written object, i.e. with the complete data object after the operation has been executed. Besides, an application server subscribes to notifications for the continuous queries of its clients and forwards them correspondingly.

The task of matching the stream of incoming operations against all continuous queries is executed in distributed fashion and partitioned both by written objects and maintained queries. Thus, each processing node is only responsible for a subset of all queries and a subset of all operations. Changes are detected based on whether an object *used to be* a match and whether it still *is* a match for a query. For every change, a notification is sent upstream.

[8] https://horizon.io/
[9] https://firebase.google.com/

The proposed system design decouples resource requirements as well as failure domains for primary storage (persistent data, pull-based access) on the one hand and real-time features (change notifications, push-based access) on the other. Since the real-time workload is handled in a separate subsystem, resources for continuously maintaining query results can be scaled out, while persistent data may be kept in a strongly consistent single-node system. Using a shared-nothing architecture and asynchronous communication throughout the critical processing path, we avoid bottlenecks and thus achieve linear scalability and low latency.

### 5.1 Scalable Stream Processing Frameworks

Over the last years, a number of scalable and fault-tolerant stream processors have emerged. In the following, we briefly discuss systems that appear as viable candidates for implementing the sketched system design. We therefore do not go into detail on systems that are prone to data loss (e.g. S4 [NRK10]), have been abandoned (e.g. Muppet[10] [LLP+12] or Naiad[11] [MMI+13]), are not publicly available (e.g. Google's Photon [ABD+13] and MillWheel [ABB+13], Facebook's Puma and Stylus [CWI+16] or Microsoft's Sonora [YQC+12]) or cannot be deployed on-premise (e.g. Google's Dataflow cloud service[12] which is built on the eponymous programming model [ABC+15]). For a more detailed overview over the stream processing landscape and a discussion of the trade-offs made in the individual systems' designs, see our stream processing survey [WGFR16].

One of the oldest stream processors used today is **Storm**[13] [TTS+14]. It exposes a very low-level programming interface for processing individual events in a directed acyclic graph, the topology, with at-least-once processing guarantees and is generally geared towards low latency more than anything else. It also provides a more abstract API, Trident, that comes with additional functionality (e.g. aggregations) and guarantees (e.g. exactly-once state management), but also displays higher end-to-end processing latency than plain Storm, because it buffers events and processes them in micro-batches. Being built on the native batch processor Spark[14] [ZCD+12], **Spark Streaming**[15] [ZDL+13] also works on small batches, but usually displays even higher latency on the order of seconds. Through its integration with Spark, Spark Streaming probably has the widest user and developer base and is part of a very diverse ecosystem. **Samza**[16] [Ram15] and **Kafka Streams** [Kre16] are stream processors that are tightly integrated with the data streaming system Kafka [KNR11] for data ingestion and output. Data flow is based on individual events, but since neither Samza nor Kafka Streams have a concept of complex topologies, data has to be persisted between processing steps and latency thus adds up quickly. **Flink**[17] (formerly known as Stratosphere [ABE+14]) tries to combine the speed of a native stream processor with a rich

[10] https://github.com/walmartlabs/mupd8
[11] https://github.com/MicrosoftResearch/Naiad
[12] https://cloud.google.com/dataflow/
[13] http://storm.apache.org/
[14] https://spark.apache.org/
[15] https://spark.apache.org/streaming/
[16] https://samza.apache.org/
[17] https://flink.apache.org/

feature set comparable to that of Spark/Spark Streaming, but is not as widely adopted, yet (cf. [Fou16b, Fou16a]). Even though Flink allows to configure buffering time of individual events, it cannot be tuned as aggressively towards latency as Storm (see for example [CDE+15] or [Met16, slide 71]). **Apex**[18] is another native stream processor with similar design goals as Flink. Being relatively new on the market, Apex is still getting traction. **Heron**[19] [KBF+15] was developed by Twitter to replace Storm which had proven inefficient in multi-tenant deployments, among other reasons due to poor resource isolation. It was open-sourced recently, but has not found wide-spread use as of writing. **IBM Infosphere Streams** [HAG+13, BBF+10] is a proprietary stream processor that is bundled with its own IDE and programming language. It reportedly achieves very low latency, but performance evaluations made by IBM [Cor14] indicate it only performs well in small deployments with up to a few nodes. **Concord**[20] is a proprietary stream processing framework designed around performance predictability and ease-of-use that has just very recently been released. To remove garbage collection as a source of possible delay, it is implemented in C++. To facilitate isolation in multi-tenant deployments, Concord is tightly integrated with the resource negotiator Mesos[21] [HKZ+11].

## 6 Conclusion

The ability to notify clients of data changes as they happen has become an important feature for both data storage systems and application development frameworks. However, since established OLTP databases have been designed to work with static data sets, they typically do not feature real-time change notifications. The few systems that do are limited in their expressiveness, difficult to scale or they enforce a strong coupling between processing static and streaming data.

In this paper, we propose a scalable system architecture for providing change notifications on top of pull-based databases that sets itself apart from existing designs through a shared-nothing architecture for *linear scalability*, coordination-free processing on the critical path for *low latency*, a pluggable query engine to achieve *database-independence* and a *separation of concerns* between the primary storage system and the system for real-time features, effectively decoupling failure domains and enabling independent scaling for both. We are not aware of any other system or system design that makes complex query change notifications available as an opt-in feature.

While this paper only introduces the conceptual design and contrasts it to existing technology, we already have implemented a prototype that supports the MongoDB query language. So far, our prototype has been used in combination with MongoDB as primary storage system for two use cases: first, providing real-time change notifications for users of a web app and, second, invalidating cached query results as soon as they become stale. We will provide details on the implementation and performance of our prototype in future work.

[18] https://apex.apache.org/
[19] https://twitter.github.io/heron/
[20] http://concord.io/
[21] http://mesos.apache.org/

## References

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Very Large Data Bases*, pages 734–746, 2013.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[ABD+13] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, et al. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In *SIGMOD '13*, 2013.

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 2014.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, August 2003.

[BBF+10] Alain Biem, Eric Bouillet, Hanhua Feng, et al. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[BLT86] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[BW01] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.

[CDE+15] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. Benchmarking Streaming Computation Engines at Yahoo! *Yahoo! Engineering Blog*, December 2015. Accessed: 2016-01-11.

[Cor14] IBM Corporation. Of Streams and Storms. Technical report, IBM Software Group, 2014.

[CWI+16] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat 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.

[Das16] Dan Dascalescu. What framework should I choose to build a web app? *Dan Dascalescu's Homepage*, May 2016. https://wiki.dandascalescu.com/essays/why_meteor#BONUS_-_Scalability.

[DF14] Ted Dunning and Ellen Friedman. *Time Series Databases: New Ways to Store and Access Data*. O'Reilly Media, November 2014.

[Fou16a] The Apache Software Foundation. Apache Flink: Powered By Flink. *Apache Flink website*, 2016. Accessed: 2016-09-23.

[Fou16b] The Apache Software Foundation. Powered By Spark – Spark – Apache Software Foundation. *Apache Spark Website*, 2016. Accessed: 2016-09-23.

[Gai16] Glenn Gailey. Azure welcomes Parse developers. *Microsoft Azure-Blog*, February 2016. Accessed: 2016-09-19.

[GBR14] F. Gessert, F. Bucklers, and N. Ritter. Orestes: A scalable Database-as-a-Service architecture for low latency. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, pages 215–222, March 2014.

[HAG+13] M. Hirzel, H. Andrade, B. Gedik, et al. IBM Streams Processing Language: Analyzing Big Data in Motion. *IBM J. Res. Dev.*, 57(3-4):1:7–1:7, May 2013.

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[Inc16] MongoDB Inc. Replica Sets. *MongoDB 3.0 Documentation*, 2016. https://docs.mongodb.com/manual/release-notes/3.0/#replica-sets, accessed: October 1, 2016.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[KNR11] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *NetDB'11*, 2011.

[Kre16] Jay Kreps. Introducing Kafka Streams: Stream Processing Made Simple. *Confluent Blog*, March 2016. Accessed: 2016-09-19.

[LLP+12] Wang Lam, Lu Liu, Sts Prasad, et al. Muppet: MapReduce-style Processing of Fast Data. *VLDB 2012*, 2012.

[M+08] S. Moore et al. Using Continuous Query Notification. In *Oracle Database Advanced Application Developer's Guide, 11g Release 1 (11.1)*. Oracle, 2008.

[met14] Large number of operations hangs server. https://github.com/meteor/meteor/issues/2668, 2014. https://github.com/meteor/meteor/issues/2668, accessed: October 1, 2016.

[Met16] Robert Metzger. Stream Processing with Apache Flink. *QCon London*, March 2016. https://qconlondon.com/london-2016/system/files/presentation-slides/robertmetzger.pdf.

[MMI+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR*, 2003.

[NRK10] Leonardo Neumeyer, Bruce Robbins, and Anand Kesari. S4: Distributed stream computing platform. In *In Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud*, 2010.

[Par16a] Parse. GitHub: contributors to ParsePlatform/parse-server. 2016. Accessed: 2016-09-19.

[Par16b] Parse. Scalability. *Parse LiveQuery documentation*, 2016. Accessed: 2016-09-20.

[Pip15] PipelineDB. Two Phase Commits. *PipelineDB Enterprise documentation*, 2015. Accessed: 2016-09-18.

[Ram15] Navina Ramesh. Apache Samza, LinkedIn's Framework for Stream Processing. *thenewstack.io*, January 2015. http://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing/.

[Ret16] RethinkDB. Limitations in RethinkDB. *RethinkDB documentation*, 2016. Accessed: 2016-09-19.

[SC05] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.*, 34(4):42–47, December 2005.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-only Databases. *SIGMOD Rec.*, 21(2):321–330, June 1992.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[WBL+07] Andrew Witkowski, Srikanth Bellamkonda, Hua-Gang Li, Vince Liang, Lei Sheng, Wayne Smith, Sankar Subramanian, James Terry, and Tsae-Feng Yu. Continuous Queries in Oracle. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1173–1184. VLDB Endowment, 2007.

[WGFR16] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. Real-time stream processing for Big Data. *it - Information Technology*, 58(4):186–194, June 2016. http://www.degruyter.com/view/j/itit.2016.58.issue-4/issue-files/itit.2016.58.issue-4.xml.

[YQC+12] Fan Yang, Zhengping Qian, Xiuwei Chen, Ivan Beschastnikh, Li Zhuang, Lidong Zhou, and Guobin Shen. Sonora: A Platform for Continuous Mobile-Cloud Computing. Technical Report MSR-TR-2012-34, Microsoft Research, March 2012.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan 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.