Real-Time Data Management for Big Data

Extended Abstract

Wolfram Wingerath
University of Hamburg
Hamburg, Germany
wingerath@informatik.uni-hamburg.de

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

Erik Witt
Baqend GmbH
Hamburg, Germany
ew@baqend.com

Steffen Friedrich
University of Hamburg
Hamburg, Germany
friedrich@informatik.uni-hamburg.de

Norbert Ritter
University of Hamburg
Hamburg, Germany
ritter@informatik.uni-hamburg.de

ABSTRACT

Users have come to expect reactivity from mobile and web applications, i.e. they assume that changes made by other users become visible immediately. However, developers are challenged with building reactive applications on top of traditional pull-oriented databases, because they are ill-equipped to push new information to the client. Systems for data stream management and processing, on the other hand, are natively push-oriented and thus facilitate reactive behavior, but they do not follow the same collection-based semantics as traditional databases: Instead of database collections, stream-oriented systems are based on a notion of potentially unbounded sequences of data items.
In this tutorial, we survey and categorize the system space between pull-oriented databases and push-oriented stream management systems, using their respectively facilitated means of data retrieval as a reference point. A particular emphasis lies on the novel system class of real-time databases which combine the push-based access paradigm of stream-oriented systems with the collection-based query semantics of traditional databases. We explore why real-time databases deserve distinction in a separate system class and dissect their different architectures to highlight issues, derive open challenges, and discuss avenues for addressing them.

1 INTRODUCTION

Reactive applications require the underlying data storage to publish new and updated information as soon as it is created; data access is *push-based.* In contrast, traditional **database management** systems [6] have been tailored towards *pull-based* data access where information is only made available as a direct response to a client request. While triggers and other push-oriented mechanisms have been added to their initial design, they are outperformed by several orders of magnitude when held against natively push-based systems [9]. In consequence, the inadequacy of traditional database technology for handling rapidly changing data has been widely accepted as one of the fundamental challenges in database design [8].

To warrant low-latency updates in quickly evolving domains, **data stream management** systems [5] break with the idea of maintaining a persistent data repository. Instead of random access queries on static collections, they perform sequential, long-running queries over data streams. Data stream management systems generate new output whenever new data becomes available and are thus natively push-based. However, data is only available for processing in one single pass, because data streams are conceptually *unbounded* sequences of data items and therefore infeasible to retain indefinitely. Consequently, queries over streams are confined to data that arrives after query activation.

Database Management	Data Stream Management
pull-based	push-based
persistent collection	ephemeral stream
ad hoc, random access	continuous, sequential

Table 1: A side-by-side comparison of core characteristics of database and data stream management systems.

Database and data stream management, respectively, follow fundamentally different semantics regarding the way that data is processed and accessed as Table 1 summarizes. The concept of **persistent collections** conforms to applications that require a (consistent) view of their domain, for instance to keep track of warehouse stock or do financial accounting. The **data stream** model, on the other hand, comes natural for domains that entertain a notion of event sequences or need to reason about the relationship between events, for example to analyze stock prices or identify malicious user behavior. However, the access paradigm – pull-based or push-based – is tied to the data model: Database management systems lack support for **continuous queries** over collections, whereas data stream management systems only provide limited options for persistent data handling.

Acknowledging the gap between traditional databases on the one side and data stream management and stream processing systems on the other, a new class of information systems has emerged that combines collection-based semantics with a push-based access model. These systems are often referred to as **real-time databases** [10, 13], because they keep data at the client in-sync with current database state "in realtime" [7], i.e. as soon as possible after change. Like traditional databases, they store consistent snapshots of domain knowledge. But like stream management systems, they allow clients to subscribe to long-running queries that push incremental updates.

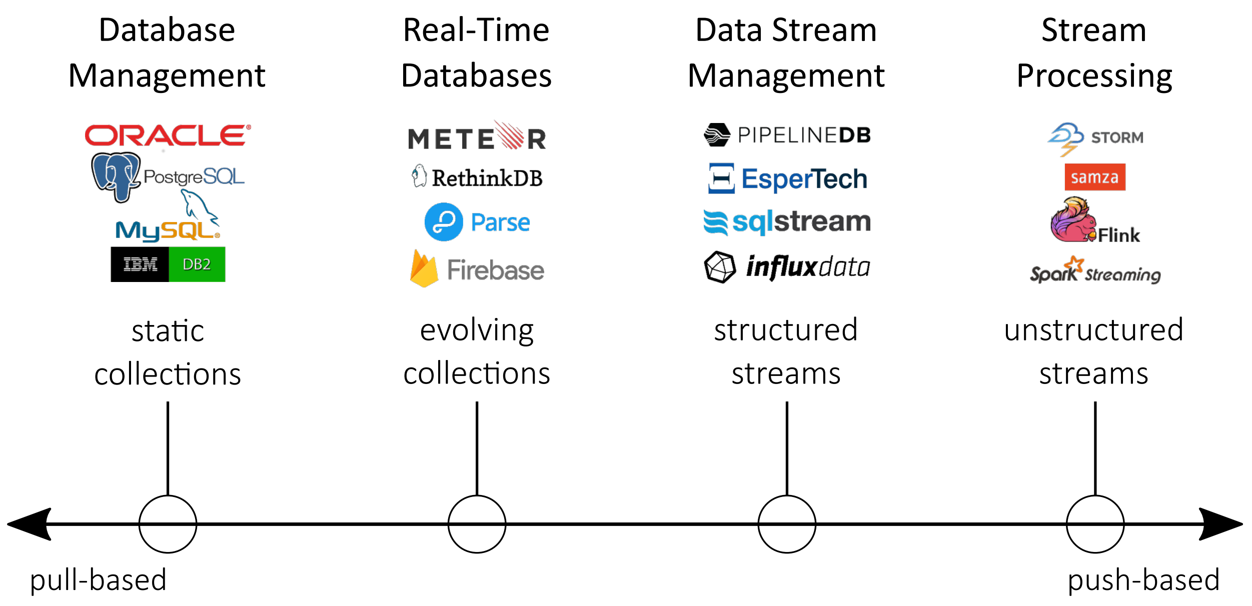

Figure 1: Different classes of data management systems and the access patterns they support.

2 SYSTEM LANDSCAPE OVERVIEW: PULL VS. PUSH

We think systems for data management can be classified by the way they facilitate access to data as illustrated in Figure 1. At the one extreme, there are **traditional databases**which represent snapshots of domain knowledge as the basis of all queries. At the other extreme, there are general-purpose **stream processing** engines which are designed to generate output from conceptually unbounded and arbitrarily structured ephemeral data streams. Real-time databases and data stream management systems both stand in the middle, but adhere to different semantics: **Real-time databases** work on evolving collections that are distinguished from their static counterparts (i.e. from database collections) through continuous integration of updates over time, enabling continuous (real-time) queries over database collections. **Data stream management** systems, as the name implies, provide APIs to query data streams, for example by filtering specific data and computing rolling aggregations and joins over to-be-specified time windows. In contrast to most general-purpose stream processors, datastream management systems usually support pull-based access to some degree, e.g. in the form of common ad hoc database queries over the set of currently retained records.

2.1 Static Queries vs. Continuous Queries

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

2.2 Collections vs. Streams

While a database collection represents the current *state* of the application domain, a data stream rather encapsulates recent *change.*

A **stream**-based representation of an application domain provides a sequential view on events as they occur, but does not retain them indefinitely: Data items are available for a certain time window and are discarded eventually. This view on the data promotes use cases that require notifications, but queries do not reflect actions that happened long ago, since the system only operates on a suffix and not the entirety of event history. In order to serve historical data, the ephemeral events have to be applied to a persistent representation of application state.

A database **collection** reflects all data ever written and thus enables queries that take all events into account. Since collection-based ad hoc queries only generate one single output, though, traditional databases do not propagate informational updates to the client.

2.3 Real-Time Queries Over Database Collections

Given a database's limitation to mainly pull-based access, reactive user interfaces are hard to build on top of an ordinary database. One possibility is to reevaluate a given collection-based query from time to time which is inefficient and introduces staleness on the order of the refresh interval. Another approach is to merge results from collection-based and stream-based queries; thus, the application is effectively burdened with the task of view maintenance which is complex and error-prone. Real-time databases aim to close the gap between both paradigms by providing collection-based semantics for pull-based and push-based queries alike.

3 IN-DEPTH SURVEY: REAL-TIME DATABASES

Our real-time database survey will concentrate on the systems we perceive as the most popular. Due to space limitations, we do

	Meteor		RethinkDB	Parse	Firebase
	poll-and-diff	oplog tailing			
scales with write throughput	✓	✗	✗	✗	?
scales with number of queries	✗	✓	✓	✓	?
composite queries (AND/OR)	✓	✓	✓	✓	✗
sorted queries	✓	✓	✓	✗	○ (single attribute)
limit	✓	✓	✓	✗	✓
offset	✓	✓	✗	✗	✓
aggregations	✗	✗	✗	✗	✗
joins	✗	✗	✗	✗	✗
event stream queries	✓	✓	✓	✓	✓
self-maintaining queries	✓	✓	✗	✗	✗

Table 2: A direct comparison of the different real-time query implementations covered in the in-depth survey.

not discuss these systems in the extended tutorial abstract and refer to our written survey for reference [11]. Table 2 sums up the respective capabilities of each system detailed in our discussion: Meteor, RethinkDB and Parse provide complex real-time queries, but present scale-prohibitive bottlenecks in their respective architectures. While the technology stack behind Firebase is not disclosed, it is apparent that Firebase avoids scalability issues by simply not offering complex queries to begin with.

3.1 Open Challenges

In concept, real-time databases extend traditional databases as they follow the same semantics, but provide an additional mode of access. In practice, though, there is no established scheme how to build a practically useful real-time database system. As will be shown in the tutorial, every push-based real-time query mechanism is deficient in at least one of the following characteristics:

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

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

(3) **legacy support**: Today's real-time databases have been designed from scratch or on top of NoSQL datastores [11] that do not follow standards regarding data model or query language. They implement custom protocols for pull-based and push-based data access alike and exhibit interfaces that are incompatible among different vendors. While the complete lack of support for legacy interfaces (particularly SQL) may be acceptable in development of a new application, it complicates the adoption of push-based queries in the context of existing technology stacks.

(4) **abstract API**: Many real-time query APIs expose specificities of the underlying implementation and thus offer poor data independence. As such, these interfaces reflect bottom-up design and force developers to reason about problems that lie beyond the application domain. For example, most real-time databases do not provide interfaces that can be used without knowledge of system internals. Instead, they mostly require an understanding of internal mechanisms or the structure of change events.

During the talk, we will illustrate how the above-mentioned limitations present themselves in practice. We also identify the underlying issues in the respective system architectures and discuss possibilities to avoid them in future designs. In this context, we will discuss related technology (e.g. distributed stream processing engines [12]) and use them as a source of inspiration for resolving the apparent challenges.

4 DIFFERENTIATION FROM OTHER VERSIONS OF THE TUTORIAL

The survey of stream processing engines and the overview over real-time databases have already been presented at different occasions, e.g. at BTW 2017 [4]. Some of the use cases that will be presented have been discussed in our VLDB 2017 industry paper [2]. However, since two of the authors (Wolfram Wingerath and Felix Gessert) are just now finishing their Ph.D. theses on real-time big data management, the tutorial intended for March 2018 will incorporate significant updates and extensions. In particular, the scientific portion of the talk will be amended by recent developments in the space of real-time databases. Further, we will present our experiences in building and using a real-time database in customer-facing applications at the Backend-as-a-Service company Baqend. Thus, the tutorial will provide a unique combination of broad scientific research and real-world experiences.

5 SCOPE, LENGTH & INTENDED AUDIENCE

The tutorial in the form outlined here is intended for 90 minutes and will concentrate on push-based systems, namely real-time databases and stream processing engines. We can also extend this tuotrial to 180 minutes by including our previous tutorials on NoSQL database systems [1, 3, 4] and discussing them in the light of real-time and stream processing requirements. This tutorial is intended for anybody interested in novel database technology; there are no prerequisites, even though a certain technical understanding of databases will be helpful in following the in-depth discussion.

6 PRESENTER BIOGRAPHIES

Wolfram Wingerath is a Ph.D. student under supervision of Norbert Ritter teaching and researching at the University of Hamburg. He was co-organiser of the BTW 2015 conference and has held workshop and conference talks on his published work on several occasions. Wolfram is part of the databases and information systems group and his research interests evolve around real-time databases and related technology such as scalable stream processing, NoSQL database systems, cloud computing, and Big Data analytics. His Ph.D. thesis explores a scalable design for push-based real-time queries on top of pull-based databases.

Felix Gessert is a Ph.D. student at the databases and information systems group at the University of Hamburg. His main research fields are scalable database systems, transactions, and web technologies for cloud data management. His thesis addresses caching and transaction processing for low-latency mobile and web applications. He is also founder and CEO of the startup Baqend that implements these research results in a cloud-based backend-as-a-service platform. Since their product is based on a polyglot, NoSQL-centric storage model, he is very interested in both the research and practical challenges of leveraging and improving these systems. He is frequently giving talks on different NoSQL topics.

Erik Witt is a Full Stack developer and perforamance engineer at Baqend where he builds and optimizes scalable web applications for the cloud. As the highlight of his master's degree at the university and in cooperation with Baqend, he developed a web-caching-based transaction concept for distributed cloud databases. Erik has talked about his work at numerous conferences and also regularly authors articles on the Baqend company blog and related media in order to present the intricacies of web performance to a broader audience.

Steffen Friedrich is a Ph.D. student working under supervision of Norbert Ritter at the University of Hamburg. He has taken part in several workshops and conferences, both as presenter and as co-organiser (BTW 2015). Being a member of the databases and information systems group, Steffen is interested in large-scale data management and data-intensive computing. Furthermore, in his Master thesis, he also dealt with data quality issues, specifically with duplicate detection in probabilistic data. His research project is primarily concerned with benchmarking of non-functional characteristics (e.g. consistency and availability) in distributed NoSQL database systems.

Norbert Ritter is a full professor of computer science at the University of Hamburg, where he heads the databases and information systems group. He received his Ph.D. from the University of Kaiserslautern in 1997. His research interests include distributed and federated database systems, transaction processing, caching, cloud data management, information integration, and autonomous database systems. He has been teaching NoSQL topics in various courses for several years. Seeing the many open challenges for NoSQL systems, he and Felix Gessert have been organizing the annual Scalable Cloud Data Management Workshop (www.scdm.cloud) to promote research in this area.

REFERENCES

[1] Felix Gessert and Norbert Ritter. 2016. Scalable Data Management: NoSQL Data Stores in Research and Practice. In *32nd IEEE International Conference on Data Engineering, ICDE 2016.*

[2] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. 2017. Quaestor: Query Web Caching for Database-as-a-Service Providers. *Proceedings of the 43rd International Conference on Very Large Data Bases* (2017), 12.

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

[4] Felix Gessert, Wolfram Wingerath, and Norbert Ritter. 2017. Scalable Data Management: An In-Depth Tutorial on NoSQL Data Stores. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany.*

[5] Lukasz Golab and M. Tamer Zsu. 2010. *Data Stream Management.* Morgan & Claypool Publishers.

[6] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends databases* 1, 2 (Feb. 2007), 141–259. https://doi.org/10.1561/1900000002

[7] Ryan Paul. 2015. Build a realtime liveblog with RethinkDB and PubNub. *RethinkDB Blog* (May 2015). https://rethinkdb.com/blog/rethinkdb-pubnub/ Access: 2017-05-20.

[8] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. https://doi.org/10.1145/1107499.1107504

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

[10] Frank van Puffelen. 2016. Have you met the Realtime Database? *The Firebase Blog* (July 2016). https://firebase.googleblog.com/2016/07/have-you-met-realtime-database.html Accessed: 2017-05-20.

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

[12] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. 2016. Real-time stream processing for Big Data. *it - Information Technology* 58, 4 (2016), 186–194. https://doi.org/10.1515/itit-2016-0002

[13] Alice Yu. 2015. What does it mean to be a real-time database? — Slava Kim at Devshop SF May 2015. *Meteor Blog* (June 2015). Accessed: 2017-05-20.