ADVANCED DATABASES PROJECT:
REAL-TIME DATABASES AND FIREBASE

Submitted to
Ph.D. Esteban Zimányi
INFO-H-415 Advanced Databases

Submitted by
Gabriela Martínez (000474221)
Pablo López (000471769)

Big Data Management and Analytics Master Program
École Polytéchnique Bruxelles
Université Libre de Bruxelles
Brussels
December 2018

Table of contents

Introduction

Databases in the era of reactiveness

The broadening of the Internet and hardware advancements during the last decade have given birth to what is nowadays known as the reactive programming paradigm in the computer science field. Behind it, a single powerful reason appears and is challenging the ways of working of the software development industry around the world: *today, a single website handles as much traffic as the entire Internet just a few years ago* and this has implied that end users rely each time more on applications and also expect millisecond response times, 100% uptime (Bonér, Farley, Kuhn, & Thompson, 2016, p.2)

This new era of reactiveness, of course, has its own predecessors. The different programming frameworks that can be observed in below graph, were also answers to the very first challenges that modern technologies began to present to application developers. Just to illustrate one of them, back in 1999, the management of concurrency was already an issue for expert developers that needed to deal with threads. In this sense, Kevin Webber was accurate to express this difficulty that was intended to be addressed with J2EE (now Java EE): "*... the main selling feature of Java was the ability to "write once, run anywhere", but anywhere was in the context of which operating system the JVM was installed on, not concepts like the cloud or the massive number of concurrent connections that we're designing for in the age of the Internet of Things*" (Webber, 2014, p.6). Later in time, Akka was developed, one of the most mature technologies available on the JVM for building reactive applications, whose creators, Jonas Bonér and Roland Kuhn, were also co-authors of the Reactive Manifesto that was declared a few years ago (Bernhardt, 2016, p.1).

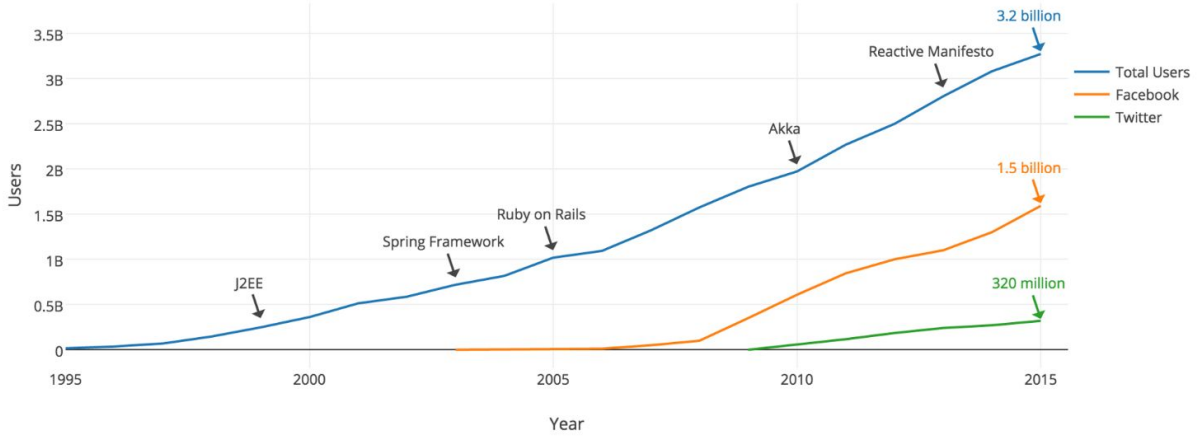

Figure 1. Evolution of the number of users on the Internet, Facebook and Twitter from 1995 to 2015. Source: taken from (Webber, 2014)

From those times back in the nineties, it may have been expected that a change was going to be needed in order to improve users application experience. Yesterday's technologies used to support applications deployed in several servers that took seconds to respond and consumed gigabytes of data, while nowadays applications are run in several devices (many

of them cloud-based with multi-core processors) and may need to load petabytes of data in a matter of milliseconds. It is not a specific-framework issue anymore, but a need for change in the whole sense new applications are perceived and programmed (Bonér, Farley, Kuhn, & Thompson, 2016, p.2). Indeed, this is why the reactive programming paradigm was born: to build systems that are ***Responsive, Resilient, Elastic (or scalable) and Message-Driven***. Moreover, this paradigm also seeks global and individual recognition of these four features and the creation of a common architecture approach to develop reactive systems. (Bonér, Farley, Kuhn, & Thompson, 2016, p.3).

The "Reactive Manifesto" is an approach to effectively manage user experience and meet users' new requirements with regards to fast response and permanent access availability and refresh. However, databases as known today, which are behind the scenes for the users, also need to be adapted to comply with the expected storage capacities they may require for the order of petabytes. Furthermore, reactive applications demand that the data they consume can be published and refreshed immediately after its creation. Therefore, databases in this new scenario need to deal with two major challenging aspects, as they need to be scalable enough while ensuring a type of data access better known as *push-based* (Stonebraker, Hamilton, & Hellerstein, 2007, p.141-259). Given the previous, the concept of real-time databases has appeared to close the existing gap between the way of working of classical databases and the need for a more suitable data storing logic able to deal with evolving collections.

Objective and report structure

The following report aims to present the state-of-the-art of the concept of real-time databases and deep technical details about one of the most widely-used products using this new technology in real-world applications: Firebase. Altogether with the previous, a use case will also be presented to show a real example that implements Firebase as a database layer. Specifically, Firebase will be used to store tweets information coming directly from the Twitter API.

The report is structured as follows: first, an overall view of the traditional database management systems is presented, which will introduce the main weaknesses of these classical tools to deal with evolving collections, the main concept behind push-based databases. After a comparison between both approaches, real-time databases are introduced and some of the main vendors will be discussed. Later, an in-depth conceptualization will be made for Firebase database. Finally, a technical Firebase implementation will be shown followed by a short discussion and conclusion section.

Pull-based data management systems

The limits of traditional database management systems

Opposite to the goal of being reactive, traditional database management systems (DBMS) operate under a *pull-based* type of data access, which enables information as long as a client request is made. According to Wolfram et al, this type of database architecture used to fit well a variety of domains where users work on a common data set but in isolated ways, which is the case of most of the Online Transactional Processing (OLTP) engines. However, specific types of applications such as online websites creation portals need to act on an immediate concurrent basis in order to keep data up to date (Wingerath, Gessert, Friedrich, Witt, & Ritter, 2017, p.1). As observed in Figure 2, *Wingerath et al* provide a simple example to show how a traditional SQL query would ineffectively retrieve the state of a blog post that has been edited several times.

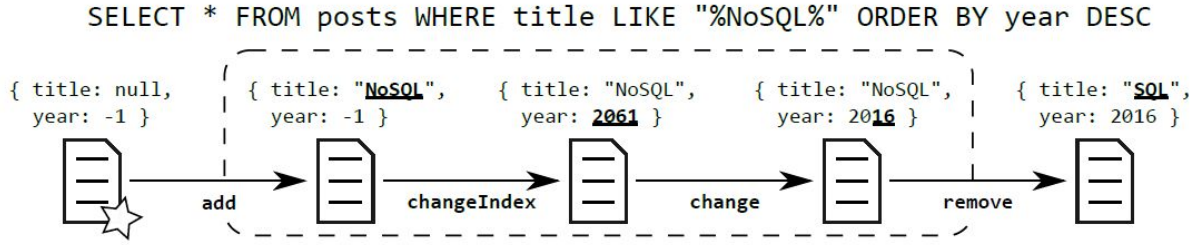

Figure 2. Notifications that occur while a blog post is being edited. Source: taken from (Wingerath, Gessert, Friedrich, Witt, & Ritter, 2017, p.1)

When the referred blog is initially created, it does not have any title or year of creation and therefore does not match the SQL query presented. Later, the author of the blog decided to change its title to “NoSQL” and then the SQL query will return a result (dashed region) and all the blog viewers will be notified of the event. Note that at this time, the blog is still missing a year and will appear in the last position of the result set. Later on, the author decided to change this figure and wanted to write “2016” as the year of creation but accidentally wrote “2061”. This fact will make the blog to have the largest value in the field *year*, and the query will move the blog position to the first one in the result set while notifies the rest of the users of this event through a ChangeIndex notification. Furthermore, when the author realized the made mistake, also decided to change the year again by “2016”. This change will generate a notification to the users of the blog but the index position of the blog will remain the same (last position), as no other new entry has been created. The problem arrives when the author finally decided to change the name of the blog to “SQL”, a scenario under which the SQL query does not match anymore and gets out of the results boundary that allowed to notify other users about the recent changes in the document. Thus, it is clear now that detecting query result changes may be more complex than detecting individual data item changes as standalone.

Query maintenance: the problem statement

In short, the main difficulty that traditional databases face dealing with examples as the previous one has to be with the capacity of keeping an eye on the result of any real-time query in order to capture the new information and update the data. This is a hard task because results to be retrieved are not a list of identifiers easily trackable. Moreover, in every written operation, a real-time database has to inspect every single object involved (Wingerath, 2017, p.15). Specifically, each database update has to be analyzed with respect to each continuous query, as can be observed in the following schema:

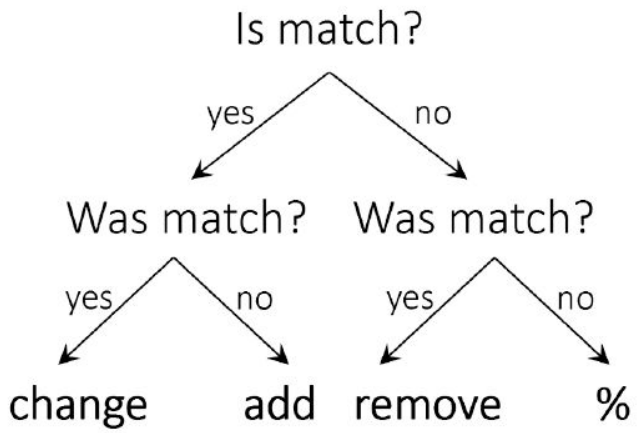

Figure 3. Updates to be analyzed by a real-time database. Source: taken from (Wingerath, 2017, p.16)

There are two main questions to answer in a continuous way for every single real-time query. First, the database needs to check if the object that was written is a match with regards to the query and second, the database needs to check if that object made a match before. If the answer to those both questions is a *no*, then the update suggested by the query is disregarded. Otherwise, the result gets updated and the end user should receive an update (Wingerath, 2017, p.17). In order to bring this problem to the real world, Wingerath explains the following: "*Let's say you have an app with 1,000 concurrent users and an average throughput of 1,000 operations per second. Given each user has only one active real-time query, the real-time database already has to perform 1 million matching operations — every single second. And this is just for plain filtering and does not account for more complex queries with even higher overhead. You could apply "optimizations" (e.g. batching) that trade throughput for increased latency, but if you want immediate change notifications, there are no other options*" (Wingerath, 2017, p.18).

The previous is linked to what Michael Stonebraker says about the BDMS, pointing them as "*one size fits all*" that no longer match modern database storage and software development needs, which are oriented towards the updating of *fastly* incoming data. The term "fast" of course, will not imply the same data processing velocity for the blog post application than for a financial stock market feed analysis in real-time, but in general, the use of the same relational model row-by-row, B-trees for indexing, cost-based optimizers and ACID transaction properties fit well for OLTP engines and are insufficient to properly address two main appliances: data warehouses, and data streaming processing (both structured and

unstructured), where the forehead mentioned examples fit (Stonebraker & Cetintemel, 2005, p.1-6).

Were triggers a solution?

The inability of conventional databases to deal with non-static data, however, is not new and has been addressed since some decades ago as a critical and as one of the fundamental challenges in modern databases design. To deal with this, OLTP vendors started to implement alternatives such as *triggers* as part of their products. The idea was to mix the existing database features with a desired reactive functionality, but this was still much behind the expectations. Example 1 illustrates how one of these enhanced databases was outperformed by several orders of magnitude when compared against one *push-oriented* data management system.

Example 1. Financial-feed processing business case

In 2005, Michael Stonebraker and Ugur Centitemel were pioneers in spreading the advantages of the push-oriented paradigm, which at the same time explicitly showed the limitations of pull-based systems for certain business domains (Stonebraker & Cetintemel, 2005, p.1-6).

The authors of this example worked with a financial company that used to deploy real-time analytics from financial subscription feeds such as Bloomberg or Infodyne, in a traditional OLAP tool. The business purpose was to alert commercial traders if information from any of these sources was delayed so they do not completely trust that feed for any decision-making. Expectedly, this is a business case in which the velocity of data processing is highly demanded and the KPI to maximize was the number of messages processed in a single window time defined and in a single machine (no parallelization was considered).

Specifically, the referred financial company sought to read information about securities from two different real-time feed providers. Securities could be *slow* or *fast-moving* in the market, and the time for detecting late signals from each type of security was different (for the slow-moving securities, 60 seconds after the last tick for the same security would imply a late tick, but this number will be reduced to 5 seconds for the fast-moving ones). Additionally, all the late ticks were needed to be counted by feed supplier for historical analysis purposes, so if a feed provider summed up to 100 late ticks, it was fully disregarded by the analysts. The complete prototype application is described in Figure 4.

This implementation was replicated in both a traditional relational database management system (RDBMS), that used triggers to insert the streaming events, and also in StreamBase, a framework used for event processing and which started to implement push-oriented features to efficiently capture streaming data on top of the Aurora MIT project back in 2003 (Kochovsky, 2017, p.3). Although the authors do not mention which database management system was specifically used, they do mention that both tools worked under the same machine conditions (2.8Ghz Pentium processor with 512 Mbytes of memory and a single SCSI disk). As a result, StreamBase could execute 160.000 messages per second and the RDBMS tool could coax only 900 per second.

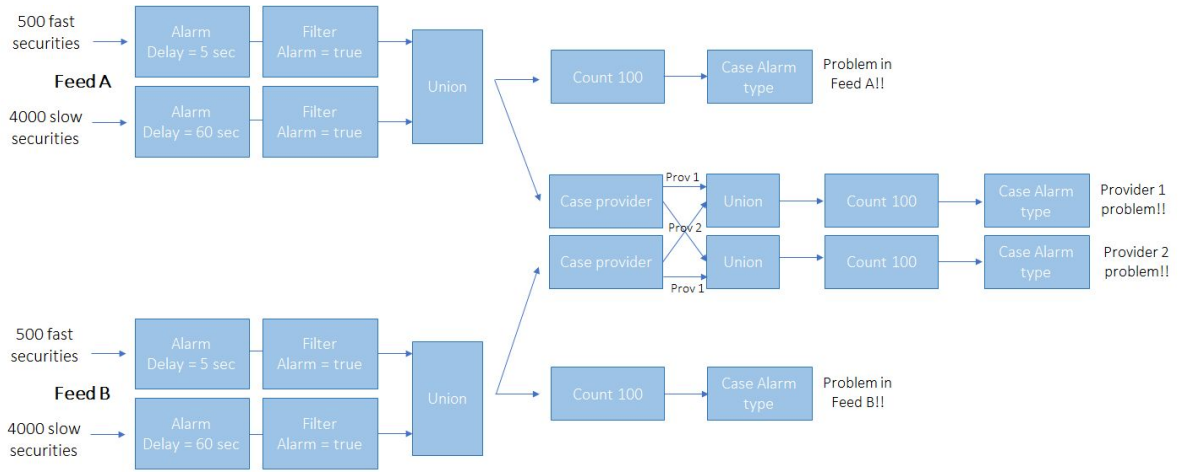

Figure 4. The feed alarm application in StreamBase. Source: taken from (Stonebraker & Cetintemel, 2005, p.3)

Business cases out of pull-based systems boundaries

The previous section illustrates for a couple of applications and in a simple way the types of limitations that classical databases present when dealing with dynamic query result sets. However, it is also possible to find a variety of business domains that require server-side notifications on data state updates, such as (Stonebraker & Cetintemel, 2005, p.3):

- **Notifications for customers:** instant messaging applications, courier services that allow online tracking of shipments, online collaborative portals (e.g. Google Drive Documents, project management tools), online university admission platforms, and online visa application portals are common implementations that usually notify users about the status of a process.
- **Cache invalidations:** catching dynamic data make possible that caches can be replaced or removed within significantly reduced times when a result is updated, as the responsible queries do not point directly to a static database.
- **Materialized views:** dynamic queries can allow traditional database views to be uploaded only when incoming data is found. This is how incremental Extraction, Transformation, and Load (ETL) processes would work.

State of the art

The arising of applications and use cases that were beginning to demand a push-based data solutions since the last decade, together with the lack of support showed by traditional databases to support more sophisticated reactive user interfaces on top of pullable systems led to finding further solutions that were farther from the relational world. They can be summarized in two main branches:

Complex event processing (CEP) engines

Were software systems specially designed for reading complex events updatings such as sensors malfunctioning or fraudulent login accounts online. Opposite to a traditional

database, these systems were able to read data streams in memory and aggregate the state results for short time periods and not persistently. But still, they seemed to be a very specific domain solution not applicable to a wider range of business applications (Wingerath, Gessert, Friedrich, Witt, & Ritter, 2017, p.3).

.

Push-oriented databases

Subsequent alternatives were able to reach different "*pushability*" levels. However, the main idea behind all them is simply to combine a collection-based semantics with an ability to read sequences of events. The following section will be dedicated to deeply explore these called push-oriented databases.

Gap analysis: pull vs push-based propositions

In general terms, traditional pullable data is different from streams of data, and this is the reason why management systems for both types of data are fundamentally and semantically opposite in the way they access and process information. A summary of these essential differences can be observed in Table 1.

	Database management	**Data stream management**
Data processing	*outbound, persistent collection*	*inbound, ephemeral stream*
Data orientation	*pull-based, ad-hoc*	*push-based, sequential*

Table 1. Comparison of core characteristics of database and stream management systems. Source: based on information presented in (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2018)

Collections vs. Streams

As expressed by Wingertah et al. "*While a database collection represents the current state of the application domain, a data stream rather encapsulates recent change*" (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2018, p.525). Classical databases for OLTP systems are driven by the concept of "*persistent collections*" which reflects the total data stored in a system or all its events. These databases serve applications that need to access consistent views of the stored data and do not need to provide information updated to end users. Some applications such as data warehouses, OLAP tools, and retrieving data for financial accounting balances are compliant with this model, as they take "*pictures*" of specific data in a given moment t, what creates a consistent or persistent view that will serve future analysis. On the other hand, there is the concept of "*data stream*" or "*ephemeral stream*", which represents a sequential view of events as they occur without retaining them indefinitely. This concept is inherently connected to event-based applications that fundamentally seek relationships between those sequences of data and do not intend to notify actions that happened long ago. As relationships are not static, they cannot be captured in a single point in time, but instead, need to be defined under a window-time constraint and considering a flow of information. Business applications such as analyzing stock prices, online language checkers or detecting electronic fraudulent behaviors are

examples of event-based programs that need to rapidly react to new incoming data in order to provide useful information.

Statis queries vs. Continuous queries

Given the previous, the fundamental difference between database and stream management approaches is not given by the business context itself but by the data layer behind. While traditional databases are unable to effectively support *continuous queries*, data streaming management systems lack support to handle persistent data.

Moreover, this essential difference between both approaches reflects a bias towards one or the other when the design comes to the table, as a static or pull-based queries work over bounded repositories of data to return information once at a time, opposite to the continuous or push-based queries that generate incremental output from unbounded data over a lapse. Figure 5 illustrates better this outbound/inbound difference. Conventional databases follow a *process-after-store* or *outbound* model that first stores data, which after indexing and committing the transactions, will be available for query processing under pulling requests that will be later presented to final users. However, it is widely known that the storage step is highly expensive and one of the major contributors to the total latency of applications and this is one of the reasons why the outbound database systems are mostly unable to deal with highly demanded and changing data. As an alternative to this storage operation bottleneck, real-time applications are *inbound-based* and push the data streams into the system, while at the same time process them in memory and then push the results to the application client for their consumption.

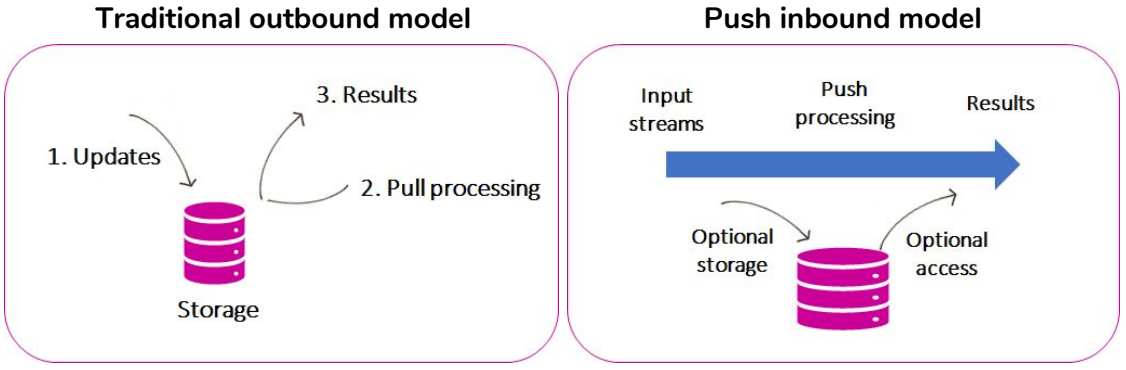

Figure 5. The outbound/inbound data access models. Source: taken from (Stonebraker & Cetintemel, 2005, p.6)

Vendors' systems landscape

Figure 6 makes a representation of different data management systems according to how they ease access to data. In the left extreme, the traditional databases appear to provide data snapshots at specific moments as the base of the queries. At the right extreme, there are general purpose stream processing tools that generate data output from both structured and unstructured ephemeral streams of events. In the middle, it is possible to find real-time databases and data streaming tools, which, however, have different types of semantics. On one hand, real-time databases work with the concept of "*evolving collection*" that allows

continuous integration of updates over database collections in real time and on the other hand, data stream management systems appear to provide a certain degree of pull-based flexibility through offering APIs to query structured retained streams over a time window.

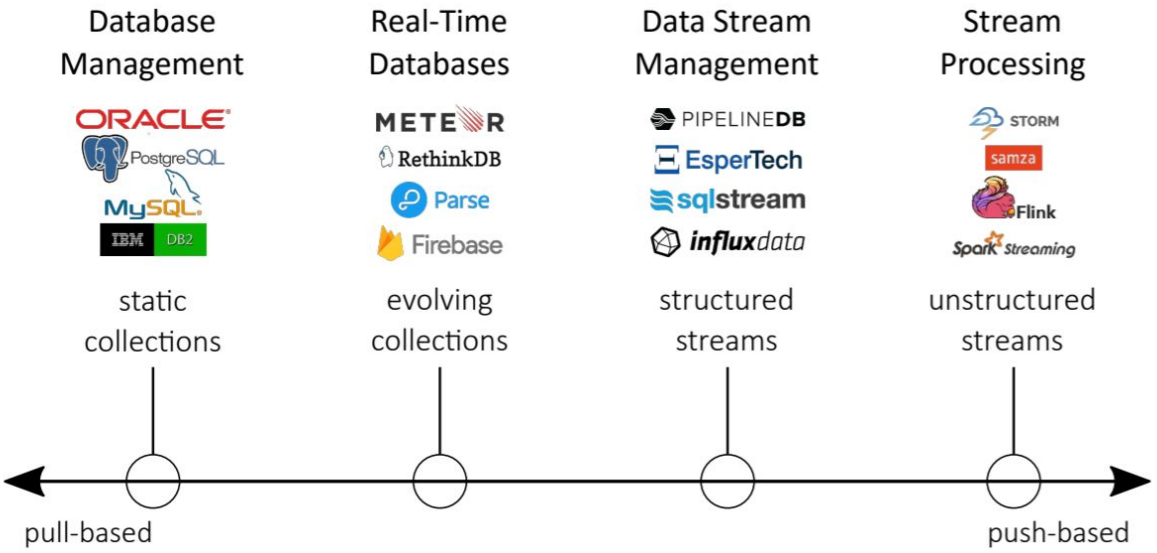

Figure 6. Different types of data management systems and access patterns supported by them. Source: taken from (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2018, p.525)

An overview of real-time databases

The gap closer

As stated before, one of the major challenges faced today by traditional OLTP systems is related to the management of evolving data needed to be captured in more than a snapshot. The systems that can efficiently handle data that changes at very fast rates require low-latency updates and to warrant this, it is necessary to break the rule of maintaining a persistent data repository. Instead of running queries over *static collections* of data, reactive systems perform sequential and long-running requests over evolving collections of data. As a result, new outputs are generated as long as the data changes and updates itself. This behavior is what we would call a native *push-based* one.

It is also known that different alternatives have been developed to overcome the static behavior of traditional databases. These new solutions have reached different levels of push-based orientation so that their domain flows from managing *evolving collections* of data to structured or unstructured streams of events (as presented in Figure 6). In this context, the introduction of the *real-time databases* concept appears to close the gap between traditional databases and more general purpose stream processing systems by including "*collection-based*" semantics and features of a push-based access model. The general idea behind this new type of information systems is that they keep the data synchronized for each client in a real-time state, what means, as soon as possible after changes were made. Furthermore, the existing gap is now closed as long as they are capable of storing consistent snapshots of data for specific domain purposes, just as a

traditional RDBMS would do, and at the same time clients are allowed to run long-lasting queries over streams to generate, for instance, incremental loads or updates of information.

Main real-time vendors

The following section will aim to provide technical specifications of the main real-time databases described above.

Meteor

Meteor is a Javascript framework to develop interactive apps and websites. Its data layer is built on top of the open source schemaless database MongoDB that is document-oriented, which provides real-time updating capabilities, query expressiveness and more architecture flexibility to store a variety of data types and fit different web or mobile applications. The timeline for the Meteor development included three main phases (Meteor, n.d.):

Figure 7. Meteor development timeline. Source: (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017)

Live queries in Meteor are carried out in the app servers of the database. They register the data updates and send notifications to the interested end users. However, a Meteor architecture can consist of one or more app servers, depending on the desired level of scalability (number of users). When more than one app servers coexist, they are all connected to a MongoDB instance and they do not interact directly. This implies that an app server will become aware of a change made from another app server only through reading the database and retrieving the state of that server queries, and this is the reason why each app server repeats each query every 10 seconds, so it can get to knot the real state of the system almost in real time. This execution mode is called *poll-and-diff* and represents a good way of dealing with data updating, but at the cost of generating staleness windows of time (as 10 seconds may be an unacceptable latency for many applications) and therefore lack of scalability (cannot efficiently execute thousands of queries for thousands of users every 10 seconds)(Meteor, n.d.).

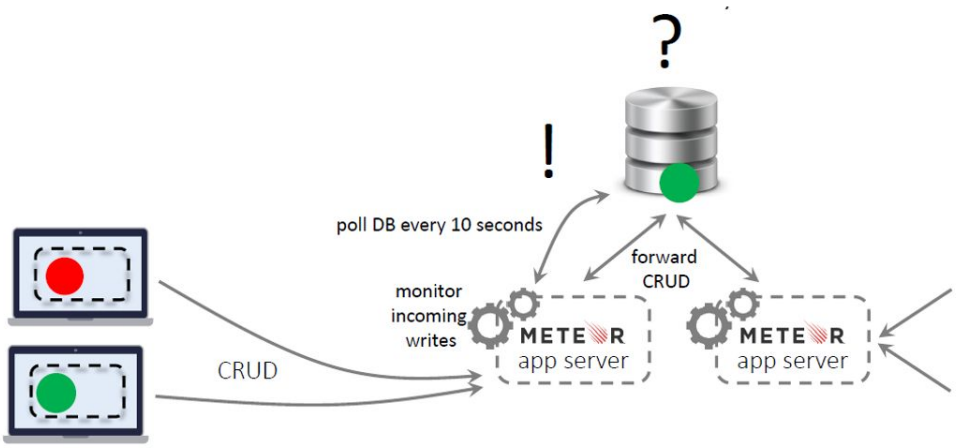

Figure 8. Poll-and-diff data querying process in Meteor. Source: taken from (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017)

The previous issue evolved in a new way of dealing with live queries. As Meteor is powered by the replication capabilities of MongoDB, it can work with *oplog tailing* querying mode. Under this scenario, each Meteor app server will act as a secondary cluster of each shard MongoDB primary cluster and each data update will reach the app servers first, which immediately after will send the updates to their corresponding primary clusters and those clusters will broadcast that written operation and will send it in the form of an oplog to all the communicated app servers. Hence, none of the app servers have to query again and again the same data because they are already aware of the changes. However, it sometimes can happen that *oplogs* do not capture completely the information contained in a written operation, and this is why each app server of Meteor has also to perform an oplog monitoring process that re-query data updates that may not be ready to be notified. The problem with this downside is that when numerous written objects are needed to be re-queried, it causes bottlenecks in the app servers.

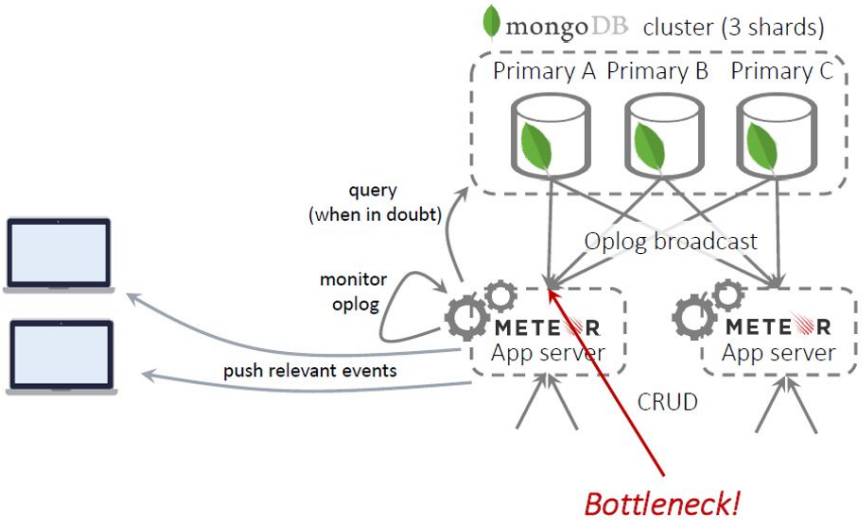

Figure 9. Oplog tailing querying approach in Meteor. Source: taken from (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017)

RethinkDB

RethinkDB is an open-source JSON-based database built in C++ and licensed on Apache 2.0. Its essence is similar to the Meteor's one, but according to Wingerath (2017), this database supports better push-based queries and joins. Furthermore, like Meteor, it also has its own managing service built on top of Javascript, which is called Horizon (RethinkDB, n.d.).

RethinkDB employes a *changefeed architecture* that allows implementing a sort of *oplog tailing* querying. However, the storage cluster is not on top of MongoDB but a proprietary development for this database and the app servers are better known as proxies. In general terms, the working model of this database presents the same scalability issue for a very high number of users as was seen in Meteor (RethinkDB, n.d.).

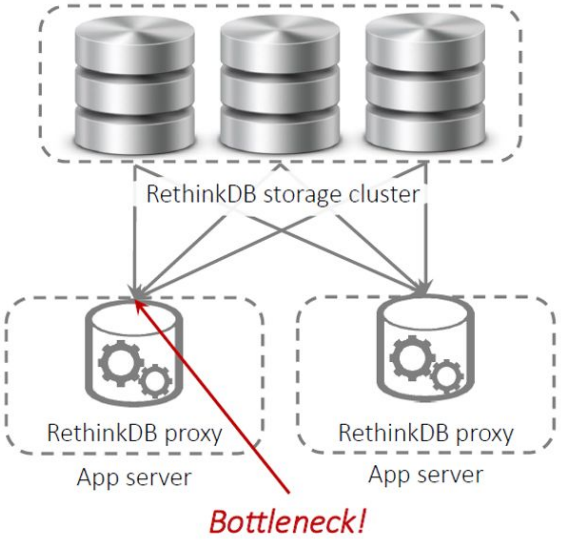

Figure 10. Changefeed architecture and querying process in RethinkDB. Source: taken from (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017)

Parse

Is one of the most popular *backend-as-a-service* tools to build mobile applications. Also open source and on top of MongoDB, Parse got one of the largest deployment of this database around the world. It supports user authentication, push notifications and has robust documentation with regards to these functionalities (Wingerath, 2017).

Nowadays, even though Parse has been discontinued (as can be seen in the timeline below), it remains to be valid and widely used by several mobile applications. This is the reason why different vendors such as Back4app, Stamplay, and AWS still continue to host Parse applications providing migration alternatives.

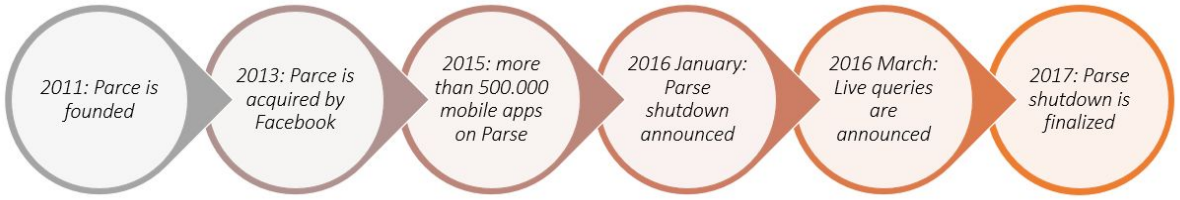

Figure 11. Parse development timeline. Source: (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017)

Firebase

The *Firebase real-time database* is a Google cloud-hosted database. Data is stored as JSON objects and is synchronized in real-time to every connected client (web, iOS, Android, amongst others). This database can be accessed directly from the clients and there is no need for an intermediary application server (Google: Firebase Real-time Database, 2018).

Data can also be persisted locally if the client application is offline, which allows real-time events to be constantly triggered, to reflect the updated state of the data to the user. When the device regains connection, Firebase synchronizes the local data changes with the remote updates that occurred while the client was offline, merging any conflicts automatically (Google: Firebase Real-time Database, 2018).

As a NoSQL database, building a properly structured database in Firebase requires to plan how the data is going to be saved and later retrieved, in order to take advantage of the optimizations that Firebase has for non-structured data. A real-time database API is available in Firebase to only allow operations that can be executed quickly (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017)(Google: Firebase Real-time Database, 2018).

In the following section, more technical specifications will be given regarding the internal settings and ways of working of Firebase.

Data Structure

Firebase is a NoSQL database that stores data as JSON objects, as can be seen in Example 1. Unlike a SQL database, there are no tables or records, but each piece data becomes a JSON node within a predefined structure that can be accessed through a key, which can be user IDs, semantic names or surrogate keys generated by Firebase (Google, Estructura tu base de datos: Google Firebase, 2018). Firebase architecture can be thought of as a cloud-hosted JSON tree (Google, Estructura tu base de datos: Google Firebase, 2018). In order to access data, it is possible to navigate through the JSON hierarchy and request specific child nodes for which there is interest in receiving immediate updates when other users modify data (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017).

```
{
  "users": {
    "alovelace": {
      "name": "Ada Lovelace",
      "contacts": { "ghopper": true },
    },
    "ghopper": { ... },
    "eclarke": { ... }
  }
}
```

Example 1. Firebase real-time database data structure. Source: taken from (Google, Estructura tu base de datos: Google Firebase, 2018)

Firebase allows to nest data until 32 levels deep. When designing the data structure in the database, it needs to be considered that querying information for one node, will automatically imply fetch data of all its child nodes as well (Google, Estructura tu base de datos: Google Firebase, 2018). Therefore, when we grant someone to read or write access at a node in the database, we also grant them access to all data under that node, the reason why in practice, it is more appropriate to keep the data structure as flat as possible (Google, Estructura tu base de datos: Google Firebase, 2018). Example 2 shows a JSON structure data object difficult to query, as retrieving information about one particular chat message requires iterating over the whole tree data. As opposite, Example 3 presents a better way to store in JSON format this type of information, which separates chats metadata, members involved and conversations.

```
{
  // This is a poorly nested data architecture because iterating the children
  // of the "chats" node to get a list of conversation titles requires
  // potentially downloading hundreds of megabytes of messages
  "chats": {
    "one": {
      "title": "Historical Tech Pioneers",
      "messages": {
        "m1": { "sender": "ghopper", "message": "Relay malfunction found. Cause:
moth." },
        "m2": { ... },
        // a very long list of messages
      }
    },
    "two": { ... }
  }
}
```

Example 2. Listing the titles of chat conversations requires the entire chats tree, including all members and messages, to be downloaded to the client. Source: taken from (Google, Estructura tu base de datos: Google Firebase, 2018)

```
{
  // Chats contains only meta info about each conversation
  // stored under the chats's unique ID
  "chats": {
    "one": {
      "title": "Historical Tech Pioneers",
      "lastMessage": "ghopper: Relay malfunction found. Cause: moth.",
      "timestamp": 1459361875666
    },
    "two": { ... },
    "three": { ... }
  },

  // Conversation members are easily accessible
  // and stored by chat conversation ID
  "members": {
    // we'll talk about indices like this below
    "one": {
      "ghopper": true,
      "alovelace": true,
      "eclarke": true
    },
    "two": { ... },
    "three": { ... }
  },

  // Messages are separate from data we may want to iterate quickly
  // but still easily paginated and queried, and organized by chat
  // conversation ID
  "messages": {
    "one": {
      "m1": {
        "name": "eclarke",
        "message": "The relay seems to be malfunctioning.",
        "timestamp": 1459361875337
      },
      "m2": { ... },
      "m3": { ... }
    },
    "two": { ... },
    "three": { ... }
  }
}
```

Example 3. A JSON tree more flattened structure that allows iterating through the list of chats by downloading only a few bytes per conversation. Source: taken from (Google, Estructura tu base de datos: Google Firebase, 2018)

Firebase is a real-time database that allows scalability, as opposed to some of the same nature tools such as RethinkDB or Parse. To illustrate this, a messaging chat application can be considered, in which bidirectional relationships between users and groups exist. Users can belong to a group, and groups comprise a list of users. If the database needs to retrieve the groups that a member belongs to, it would not be a good idea to iterate over the whole list of groups in order to fetch only information pointing to that specific user. Hence, in order to accomplish this, Firebase implements the concept of *group indexes*, as can be seen below:

```
// An index to track Ada's memberships
{
  "users": {
    "alovelace": {
      "name": "Ada Lovelace",
      // Index Ada's groups in her profile
      "groups": {
         // the value here doesn't matter, just that the key exists
         "techpioneers": true,
         "womentechmakers": true
      }
    },
    ...
  },
  "groups": {
    "techpioneers": {
      "name": "Historical Tech Pioneers",
      "members": {
        "alovelace": true,
        "ghopper": true,
        "eclarke": true
      }
    },
    ...
  }
}
```

Example 4. Use the concept of group indexes to relate two kinds of groups. Source: taken from (Google, Estructura tu base de datos: Google Firebase, 2018)

This structure implies that we need to keep the nodes updated according to the changes in the different groups, for example, to delete “Ada” from the group “techpionneers”, it has to be updated in two places. This is a necessary redundancy for two-way relationships, but it allows you to quickly and efficiently fetch Ada's memberships, even when the list of users or groups scales into the millions (Google, Estructura tu base de datos: Google Firebase, 2018).

Reading and writing data

Getting a database reference

To read or write data from the database, we need a reference to the firebase database, that can be called like this:

```
// Get a reference to the database service
var database = firebase.database();
```

Example 5. Getting a reference to a firebase database. Source: taken from (Google, Lee y escribe datos en la Web: Google Firebase, 2018)

Firebase data is retrieved by attaching an asynchronous listener to the Firebase reference. The listener is triggered once for the initial state of the data and again anytime the data changes.

Callbacks are removed by calling the off() method on the Firebase database reference. It is possible to remove a single listener by passing it as a parameter to off(). Otherwise, calling off() on the location with no arguments removes all listeners at that location. Calling off() on a parent listener does not automatically remove listeners registered on its child nodes; off() must also be called on any child listeners to remove the callback (Google, Lee y escribe datos en la Web: Google Firebase, 2018).

Basic write operations

The command set() can be used to save data to a specified reference, replacing any existing data at that path. Using set() overwrites data at the specified location, including any child nodes, as stated below:

```
function writeUserData(userId, name, email, imageUrl) {
  firebase.database().ref('users/' + userId).set({
    username: name,
    email: email,
    profile_picture : imageUrl
  });
}
```

Example 6. Add a user to a firebase database. Source: taken from (Google, Lee y escribe datos en la Web: Google Firebase, 2018)

Listen for value events

To observe events at a path, methods on() or once() need to be used. The former method, on(), will read a static snapshot of the contents at a given path and is triggered when the listener is attached and again every time the data changes (including children). If there is no data, the snapshot will return false when the method exists() is called, and null when val() is

executed on it, as can be seen in Example 7 (Google, Lee y escribe datos en la Web: Google Firebase, 2018).

```
var starCountRef = firebase.database().ref('posts/' + postId + '/starCount');
starCountRef.on('value', function(snapshot) {
  updateStarCount(postElement, snapshot.val());
});
```

Example 7. The listener receives a snapshot that contains the star count of a post at the specified location in the database at the time of the event. val() method will retrieve the snapshot. Source: taken from (Google, Lee y escribe datos en la Web: Google Firebase, 2018).

Reading data once

On the other hand, the method *once()* can be used to obtain a snapshot of the data without listening for changes. This is useful for data that only needs to be loaded once and isn't expected to change frequently or require active listening, and corresponds to a more pull-based feature in Firebase. Example 8 illustrates an example of this method (Google, Lee y escribe datos en la Web: Google Firebase, 2018).

```
var userId = firebase.auth().currentUser.uid;
return firebase.database().ref('/users/' +
userId).once('value').then(function(snapshot) {
  var username = (snapshot.val() && snapshot.val().username) || 'Anonymous';
  // ...
});
```

Example 8. Load an user's profile when they begin authoring a new post. Source: taken from (Google, Lee y escribe datos en la Web: Google Firebase, 2018)

Updating data

To simultaneously write to specific children of a node without overwriting other child nodes, update() method needs to be used. Example 9 shows the code to perform simultaneous updates to multiple locations in the JSON tree with a single call to update(). Simultaneous updates are atomic: either all updates succeed or all updates fail.

```
function writeNewPost(uid, username, picture, title, body) {
  // A post entry.
  var postData = {
    author: username,
    uid: uid,
    body: body,
    title: title,
    starCount: 0,
    authorPic: picture
```

```
  };

  // Get a key for a new Post.
  var newPostKey = firebase.database().ref().child('posts').push().key;

  // Write the new post's data simultaneously in the posts list and the user's
post list.
  var updates = {};
  updates['/posts/' + newPostKey] = postData;
  updates['/user-posts/' + uid + '/' + newPostKey] = postData;

  return firebase.database().ref().update(updates);
}
```

Example 9. A social blogging app might create a post and simultaneously update it to the recent activity feed and the posting user's activity feed using. Source: taken from (Google, Lee y escribe datos en la Web: Google Firebase, 2018)

Regarding this method, it is also possible to add a completion callback to know when the data has been committed. Both set() and update() take an optional completion callback that is called when the write has been committed to the database. If the call was unsuccessful, the callback receives an error object indicating why the failure occurred, as can be observed in Example 10 (Google, Lee y escribe datos en la Web: Google Firebase, 2018).

```
firebase.database().ref('users/' + userId).set({
    username: name,
    email: email,
    profile_picture : imageUrl
  }, function(error) {
    if (error) {
      // The write failed...
    } else {
      // Data saved successfully!
    }
  });
}
```

Example 10. Callback function. Source: taken from (Google, Lee y escribe datos en la Web: Google Firebase, 2018)

Deleting data

The simplest way to delete data is to call remove() on a reference to the location of the data. Deletion is also possible by specifying null as the value for another write operation such as set() or update(). Using this technique with update() to delete multiple children in a single API call is also feasible (Google, Lee y escribe datos en la Web: Google Firebase, 2018).

Committing transactions

The *transaction()* method is used to update values while ensuring there are no conflicts with other clients writing to the same location at the same time. If another client writes to the location before our new value is successfully written, the update function will be called again with the new current value, and the write will be retried, as can be seen in Example 11 (Google, Lee y escribe datos en la Web: Google Firebase, 2018).

```
function toggleStar(postRef, uid) {
  postRef.transaction(function(post) {
    if (post) {
      if (post.stars && post.stars[uid]) {
        post.starCount--;
        post.stars[uid] = null;
      } else {
        post.starCount++;
        if (!post.stars) {
          post.stars = {};
        }
        post.stars[uid] = true;
      }
    }
    return post;
  });
}
```

Example 11. Update of stars in a post with transaction. Source: taken from (Google, Lee y escribe datos en la Web: Google Firebase, 2018)

Reading and writing lists

Append to a list of data

The *push()* method is a proper one of the real-time databases. Firebase implements it to append data to a list in multi-user applications. The push() method generates a unique key every time a new child is added to the specified Firebase reference. By using the auto-generated keys for each new element in the list, several clients can add children to the same location at the same time without write conflicts. The unique key generated by push() is based on a timestamp, so list items are automatically ordered chronologically.

It is possible to use the reference to the new data returned by the push() method to get the value of the child's auto-generated key or set data for the child. The "key" property of a push() reference contains the auto-generated key (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018).

Listening for child events

Child events are triggered in response to specific operations that happen to the children of a node (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018). Examples of child events are: *child_added*, *child_changed*, *child_removed*, *child_moved*, as can be seen in Example 12.

```
var commentsRef = firebase.database().ref('post-comments/' + postId);
commentsRef.on('child_added', function(data) {
  addCommentElement(postElement, data.key, data.val().text, data.val().author);
});

commentsRef.on('child_changed', function(data) {
  setCommentValues(postElement, data.key, data.val().text, data.val().author);
});

commentsRef.on('child_removed', function(data) {
  deleteComment(postElement, data.key);
});
```

Example 12. Managing comments on a post. Source: taken from (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018)

Attaching a value observer to a list of data will return the entire list of data as a single snapshot, which can be accessed to query individual children.

```
ref.once('value', function(snapshot) {
  snapshot.forEach(function(childSnapshot) {
    var childKey = childSnapshot.key;
    var childData = childSnapshot.val();
    // ...
  });
});
```

Example 13. Access to the entire list throughout the "value" event. Source: taken from (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018)

Sorting data

To sort data, the following methods exist in Firebase (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018):

- ***orderByChild():*** Order results by the value of a specified child key or nested child path.
- ***orderByKey():*** Order results by child keys.
- ***orderByValue():*** Order results by child values.

Filtering data

To filter data, it is possible to combine any of the limit or range methods with an order-by method when constructing a query.

The allowed filtering behaviors are listed in Table 2. An example of the *limitToLast()* method is shown in Example 14 (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018).

Method	***Description***
limitToFirst()	Sets the maximum number of items to return from the beginning of the ordered list of results.
limitToLast()	Sets the maximum number of items to return from the end of the ordered list of results.
startAt()	Return items greater than or equal to the specified key or value, depending on the order-by method chosen.
endAt()	Return items less than or equal to the specified key or value, depending on the order-by method chosen.
equalTo()	Return items equal to the specified key or value, depending on the order-by method chosen.

Table 2. Filtering methods in Firebase Realtime Database. Source: taken from (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018)

```
var recentPostsRef = firebase.database().ref('posts').limitToLast(100);
```

Example 14. Retrieve a list of the 100 most recent posts. Source: taken from (Google, Trabaja con listas de datos en la Web: Google Firebase, 2018)

Cloud functions

This type of functions allows handling events within the Firebase real-time database, such as:

1. Waiting for changes to a particular database location.
2. Executing triggers when an event occurs and performs its tasks.
3. Receiving a data object that contains a snapshot of the data stored in the specified document.

To control when the function triggers, one of the event handlers needs to be specified as well as the database path. Functions handle database events at two levels of specificity: either listen specifically for only creation, update, or deletion events or listen for any change of any kind in a path (Google, Amplía Real-time Database con Cloud Functions: Google Firebase, 2018).

Real-time database limits

The following are restrictions on data storage and operations in Firebase real-time database. To scale beyond any of these limits, it would be necessary to use multiple databases (Google, Límites de Real-time Database: Google Firebase, 2018).

Operation	***Limit***	***Description***
Simultaneous connections	100,000	A simultaneous connection is equivalent to one mobile device, browser tab, or server app connected to the database. This isn't the same as the total number of users of an app, because users do not all connect at once.
Simultaneous responses sent from a single database.	~100,000/ second	Responses include simultaneous broadcast and read operations sent by the server from a single database at a given time.
Number of cloud functions triggered by a single write	1000	Cloud Functions are triggered by write operations, and each function can also trigger more write operations that trigger more functions (each with their own 1000-function limit).
Size of a single event triggered by a write	1 MB	The size of an event consists of the following values: ● The existing data at the writing location. ● The update value, or the delta in data necessary to write the new data to the location. ● Write operations larger than 1MB succeed on the database, but they do not trigger a function invocation.
Data transfer to cloud functions	10MB/sec sustained	The rate of event data that can be forwarded to cloud functions.

Table 3. Global limits of storage and operations of Firebase real-time databases. Source: taken from (Google, Límites de Real-time Database: Google Firebase, 2018)

Property	***Limit***	***Description***
Maximum depth of child nodes	32	Each path in data trees must be less than 32 levels deep.
Length of a key	768 Bytes	Keys are UTF-8 encoded and cannot contain newlines or any of the following characters: . $ # [ ] / or any ASCII control characters (0x00 - 0x1F and 0x7F)
The maximum size of a string	10 MB	Data is UTF-8 encoded.

Table 4. Data tree limits. Source: taken from (Google, Límites de Real-time Database: Google Firebase, 2018)

Description	Limit	Notes
Size of a single response served by the database	256 MB	The size of data downloaded from the database at a single location should be less than 256 MB for each read operation.
Total nodes in a path with listeners or queries on it	75 million*	Firebase does not allow to listen to query paths with more than 75 million nodes, cumulative. However, it would be possible to listen to query child nodes by drilling down deeper into the path or creating separate listeners or queries for more specific portions of the path.
Length of time a single query can run	15 minutes*	A single query can run for up to 15 minutes before failing. *A single query performed in the Firebase console can only run for up to 5 seconds before failing.

Table 5. Real-time Firebase reading limits. Source: taken from (Google, Límites de Real-time Database: Google Firebase, 2018)

Description	Limit	Notes
Size of a single write request to the database	256 MB from the REST API; 16 MB from the SDKs.	The total data in each write operation should be less than 256 MB. Multi-path updates are subject to the same size limitation.
Written bytes	64 MB/minute	The total written bytes through simultaneous write operations on the database at any given time.

Table 6. Real.time Firebase writing limits. Source: taken from (Google, Límites de Real-time Database: Google Firebase, 2018)

Issues with Firebase

Besides the limits stated above, there are some difficulties that need to be addressed when using Firebase in some use cases. According to experiences of some expert developers, data structure and database as a service approach may face challenges such as (Jamin, 2016; Bover, 2017):

- Client applications interact directly with the Firebase database, which can cause the server logic to be run right in each mobile or web client. This can be mitigated with the Firebase cloud functions that allow running serverless functions that are attached to data updates events (Google, Amplía Real-time Database con Cloud Functions: Google Firebase, 2018). Currently, cloud functions can be written only in TypeScript or JavaScript (Google, Primeros pasos: Cómo escribir y también implementar tus primeras funciones: Google Firebase, 2018).
- Is difficult to deal with data migration.

- Relations amongst data is difficult to maintain. Is not possible to declare “one to many” or “many to many” relationships. In order to represent relations in your data, you will incur in data duplications among the nodes of the JSON tree structure, making difficult to maintain the data consistency.
- Complex queries are not possible. As the Firebase real-time database guide states “*The real-time Database API is designed to only allow operations that can be executed quickly*” (Google: Firebase Real-time Database, 2018), querying capabilities are limited, so more complex filtering or sorting will be generally performed client-side.
- As Firebase is a proprietary tech and is exclusively cloud-based, there is a dependency on the vendor.
- Related to the previous point, as Firebase is entirely cloud-based, it is not possible to develop and test an application only with a local installation.

Future of Firebase

Google is developing a new database solution with real-time capabilities which is called Cloud Firestore. It makes improvements on the data structure, it also features richer and faster queries, and will provide better and automatic scalability. Cloud Firestore is currently in beta version, which means is not a fully stable product yet. More details about the new Cloud Firestore can be found in (Google, Cómo elegir tu base de datos: Cloud Firestore o Real-time Database, 2018).

Vendors general comparison

The following chart presents a general side-by-side comparison of different features and their availability in the previously mentioned real-time databases.

	Meteor: poll-and-diff	**Meteor: oplog failing**	**RethinkDB**	**Parse**	**Firebase**
Scales with write throughput	✓	✗	✗	✗	✓
Scales with the number of queries	✗	✓	✓	✓	✓
Composite queries (AND/OR)	✓	✓	✓	✓	✗
Sorted queries	✓	✓	✓	✗	✓
Limit	✓	✓	✓	✗	✓
Offset	✓	✓	✗	✗	✓
Aggregations	✗	✗	✗	✗	✗
Joins	✗	✗	✗	✗	✗
Event stream queries	✓	✓	✓	✓	✓
Self-maintaining queries	✓	✓	✗	✗	✗

Table 7. Comparison of different real-time query implementations. Source: (Wingerath, Gessert, Witt, Friedrich, & Ritter, 2017)

Firebase use case

The objective of this section is to present a real-time application implementation using Firebase as the database layer.

Description

The application uses Google Natural Language Processing (NLP) API to perform analysis of keywords and sentiments expressed in tweets obtained from the Twitter stream API. The processed data is stored in the firebase database which triggers updates in real-time in all connected clients devices. This implementation was developed based on the work done by the Google developer Sara Robinson (Robinson, 2017).

Application architecture

The architecture of the application (see Figure 12) consists on the following components: a client of Twitter stream API, a client of the NLP Google API, a Node server to perform the integration among the two external APIs and the Firebase database, and a web client which can be accessed by desktop and mobile devices to watch the updates in real-time.

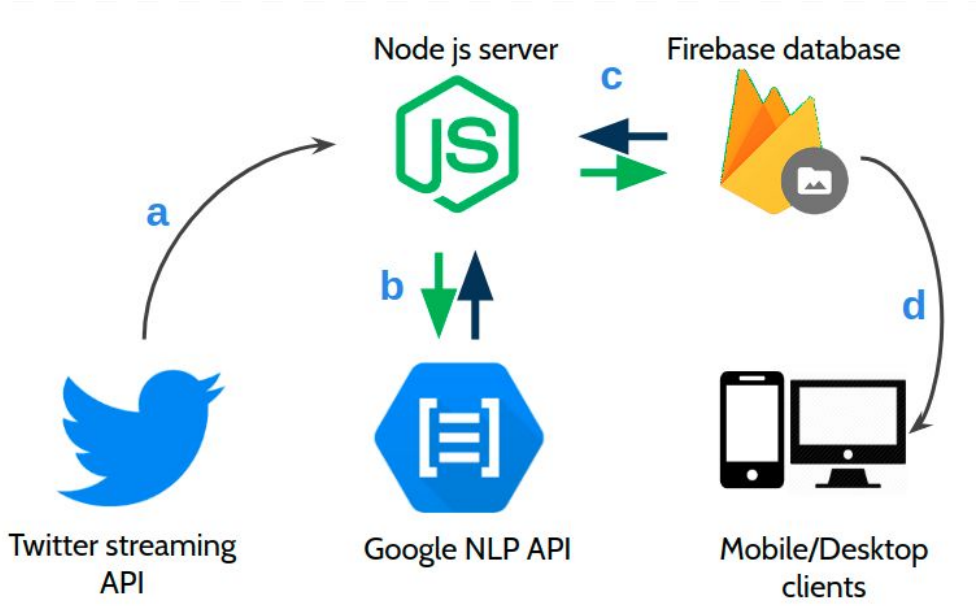

Figure 12. Real-time twitter demo application architecture. Source: based on (Robinson, 2017).

Sentiment analysis

The Node server will permanently receive new tweets from the Twitter stream API (Figure 12.a) taking into account a set of specified filters. These filters could be a list of keywords which we desire to keep track of, the language of the tweets, etc.

After a new tweet that complies with the filter's criteria arrives, the Node server extracts the text from the tweet and sends it to the Google NLP service (Figure 12.b). The NLP service returns a set of properties related to the outcome of the tweet text analysis. Example of such properties are:

- The score of the prevailing emotion in the tweet (negative, positive or neutral) given a scale from -1 to 1.
- The list of symbols or tokens found after performing a syntactic analysis of the tweet text, describing if the identified token is a symbol, an adjective, verb, etc.

More details about the Google NLP API can be found in:
cloud.google.com/natural-language/docs/basics

After obtaining the result of the tweet analysis through the NLP service, the resulting set of properties are saved into the Firebase database (Figure 12.c). In our example application, the properties are written in a child node of the root of the Firebase database, with the key “latest” which indicates the latest tweet that was processed, see Figure 13.

Figure 13. Example of the Firebase database in our example application. The child-node ‘latest’ indicates the last tweet that was analyzed by the Google NLP service.

In order to show the real-time capabilities of Firebase, the Node server also performs an active listening on the “latest” node in our database (Figure 12. c). This means that every time this node is updated in the Firebase database with a new tweet information, a function is triggered in our Node server. In this function, the Node server retrieves the properties of the latest tweet, examines which of the tracked keywords are present in the tweet text, and the emotional scale (-1 to 1) associated with the tweet. Then, the Node server, writes a new child node with the name of the corresponding keyword inside the path “root”->“keywordData” in the database, with the following properties (Figure 14):

- numMentions: Number of times the keyword was found in the analyzed tweets until now.
- totalScore: The sum of the sentiment scores associated with the keyword in all the tweets that contained the keyword.

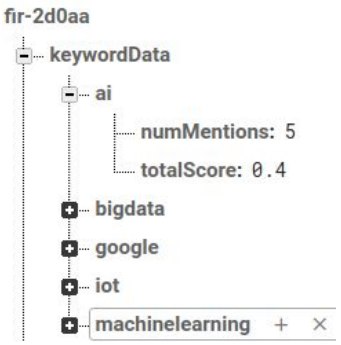

Figure 14. The keyword “ai” was found in 5 tweets already, and the sum of all the sentiments scores is 0.4 so far.

Visualization

Finally, mobiles and desktop clients can access the web application to visualize a dashboard related to the tweets (Figure 12.d). The web application view is divided into 3 sectors.

In the upper part of the view, the latest tweet that matched one or more of our selected keywords is displayed with the corresponding adjectives that were found. The sentiment score of the tweet is displayed in a blue bar where the left extreme (sad face) represent negative sentiments (-1 in sentiment score), whereas the right extreme (happy face) represents positive sentiments (+1 in sentiment score). This section is updated instantly every time a new tweet is saved in the Firebase database.

The bottom left section is a graphic that represents the keywords found and the number of matches in all the analyzed tweets. The bottom right section is a graphic that represents the average of the sentiment score per keyword. Both graphics are updated in real-time when the Firebase database is updated, the graphics are sorted by the top 10 most mentioned keyword in the analyzed tweets. The web client view is shown in Figure 15.

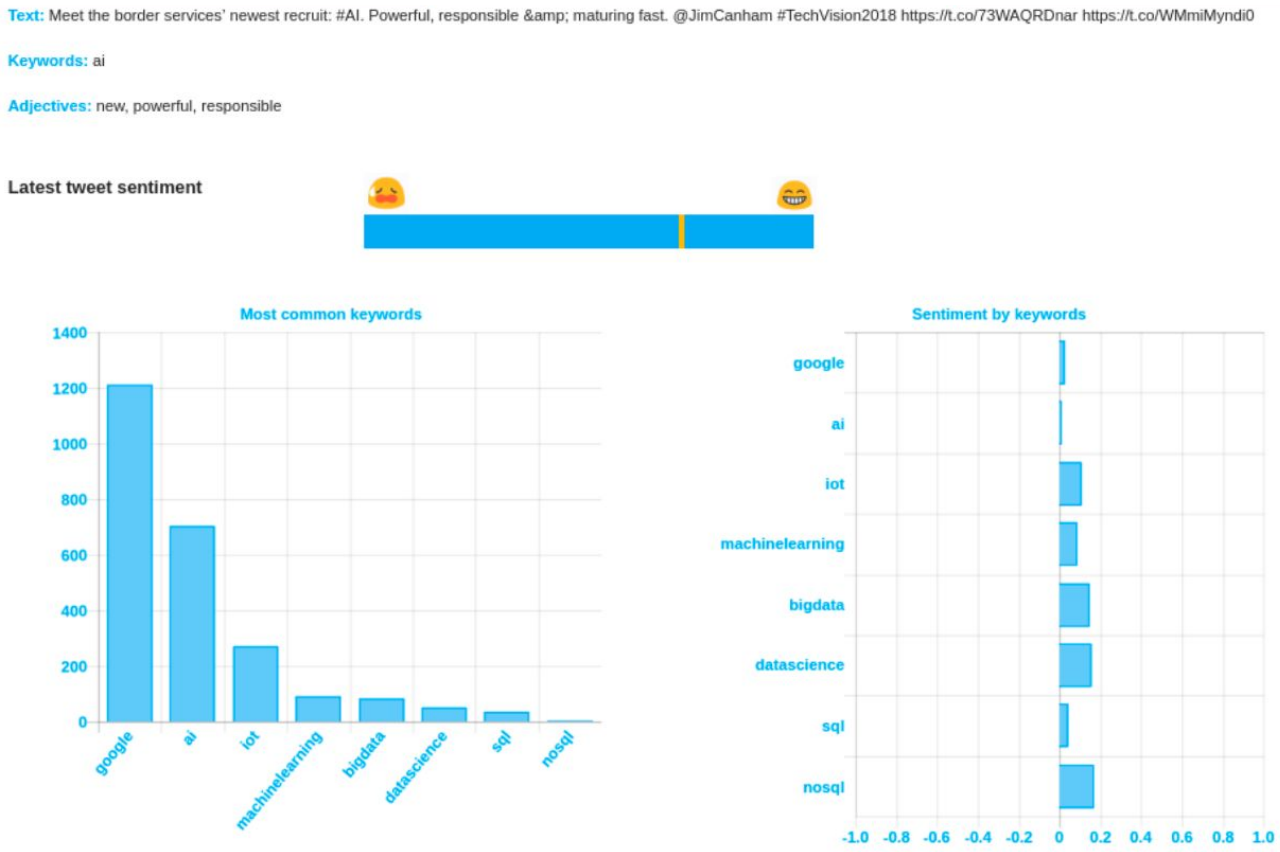

Figure 15. Web application updated in real-time with every update in the Firebase database.

The complete code with the corresponding instruction on how to execute the application can be found in the following git repository: https://github.com/srpablino/ml-talk-demos

Conclusions

In today's rapidly changing world, reactiveness has become a major need within the software development field. Users rely each time more on 100% uptime responses from applications that need to deliver real-time updates to millions of concurrent users that increase exponentially every day. Several frameworks have been designed to approach this new requirement and it is expected from databases that they also become capable of supporting fast queries over growing orders of magnitude.

In this sense, the concept of push-based databases appears to gather all the storing approaches that have been designed to deal with streams of data (evolving, structured and unstructured), which are generated by today's applications and are each time farther from the traditionally fixed schema that relational databases use to support.

Within the push-based oriented group of databases, real-time databases are a particular subgroup that aims to work with evolving collections of data. They are more flexible than a classical relational database to deal with rapidly changing data coming from web or mobile applications but are still not considered as schemaless engines able to work with unstructured streams. Firebase has been one of the major real-time databases players. Widely used in the industry, this database engine stores streams of data as single JSON documents and provides a simplistic data access model that is scalable for a large number of users when compared to other competitors such as Meteor os RethinkDB. However, some of its downsides are related to the fact that it cannot support complex and self-maintaining queries because data access is only available through keys. Moreover, Firebase database is inefficient for very nested trees because the JSON hierarchy of a database instance needs to be completely scanned to retrieve single data points.

References

Bernhardt, M. (2016, January 09). *The Swiss army knife of reactive systems on the JVM*. Retrieved November 14, 2018, from Jaxenter: https://jaxenter.com/swiss-army-knife-reactive-systems-jvm-130820.html

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2016, September 16). *The Reactive Manifesto*. Retrieved November 14, 2018, from The Reactive Manifesto: https://www.reactivemanifesto.org/

Bover, P. (2017, January 12). *Firebase: the great, the meh, and the ugly*. Retrieved December 1, 2018, from Medium Corporation: https://medium.freecodecamp.org/firebase-the-great-the-meh-and-the-ugly-a07252fbcf15

Google. (2018, October 29). *Amplía Real-time Database con Cloud Functions: Google Firebase*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/database/extend-with-functions

Google. (2018, May 29). *Cómo elegir tu base de datos: Cloud Firestore o Real-time Database*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/firestore/rtdb-vs-firestore

Google. (2018, November 19). *Estructura tu base de datos: Google Firebase*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/database/web/structure-data

Google. (2018, November 26). *Firebase Real-time Database*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/database/

Google. (2018, November 19). *Lee y escribe datos en la Web: Google Firebase*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/database/web/read-and-write

Google. (2018, November 19). *Límites de Real-time Database: Google Firebase*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/database/usage/limits

Google. (2018, November 29). *Primeros pasos: Cómo escribir y también implementar tus primeras funciones: Google Firebase*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/functions/get-started

Google. (2018, November 19). *Trabaja con listas de datos en la Web: Google Firebase*. Retrieved December 1, 2018, from Google Firebase: https://firebase.google.com/docs/database/web/lists-of-data

Jamin, B. (2016, September 18). *Reasons Not To Use Firebase*. Retrieved December 1, 2018, from Crisp: https://crisp.chat/blog/why-you-should-never-use-firebase-realtime-database/

Kochovsky, L. (2017, February 17). *TIBCO StreamBase*. Retrieved November 14, 2018, from InterWorks: https://interworks.com.mk/tibco-streambase/

Meteor. (n.d.). *What Is MongoDB?* Retrieved November 15, 2018, from Meteor: https://www.meteor.com/articles/what-is-mongodb

RethinkDB. (n.d.). *Frequently asked questions*. Retrieved November 15, 2018, from RethinkDB: https://rethinkdb.com/faq/

Robinson, S. (2017, June 28). *Building a real-time Twitter sentiment dashboard with Firebase and NLP*. Retrieved December 1, 2018, from Medium Corporation: https://codeburst.io/building-a-realtime-twitter-sentiment-dashboard-with-firebase-and-nlp-7064bb30f5ab

Stonebraker, M., & Cetintemel, U. (2005). "One size fits all": an idea whose time has come and gone. *21st International Conference on Data Engineering (ICDE'05)*. Tokyo: IEEE. doi:10.1109/ICDE.2005.1

Stonebraker, M., Hamilton, J., & Hellerstein, J. M. (2007). *Architecture of a Database System (Foundations and Trends in Databases)*. Hanover, United States: Now Publishers Inc. Retrieved November 14, 2018

Webber, K. (2014, August 19). *What is Reactive Programming?* Retrieved November 14, 2018, *from RedElastic: https://blog.redelastic.com/what-is-reactive-programming-bc9fa7f4a7fc*

Wingerath, W. (2017, July 16). A Real-Time Database Survey: The Architecture of Meteor, RethinkDB, Parse & Firebase. Retrieved November 14, 2018, from Medium baqend: https://medium.baqend.com/real-time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87

Wingerath, W. (2017, March 06). Real-Time Databases Explained: Why Meteor, RethinkDB, Parse & Firebase Don't Scale. Stuttgart. Retrieved November 16, 2018, from https://www.reddit.com/r/devel/comments/7ahop5/codetalks_2017_realtime_databases_explained_why/

Wingerath, W., Gessert, F., Witt, E., Friedrich, S., & Ritter, N. (2018). Real-Time Data Management for Big Data. *Open Proceedings*, (pp. 524-527). doi:10.5441/002/edbt.2018.63

Wingerath, W., Gessert, F., Friedrich, S., Witt, E., & Ritter, N. (2017). Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017: The Case For Change Notifications in Pull-Based Databases. Retrieved November 14, 2018, from http://btw2017.informatik.uni-stuttgart.de/slidesandpapers/E4-15-103/paper_web.pdf