Performance Analysis of Continuous Binary Data Processing using Distributed Databases within Stream Processing Environments

Manuel Weißbach, Hannes Hilbert and Thomas Springer
Faculty of Computer Science, Technische Universität Dresden, Germany

Keywords: Stream Processing, Benchmarking, Database Benchmark, Big Data, Performance.

Abstract: Big data applications must process increasingly large amounts of data within ever shorter time. Often a stream processing engine (SPE) is used to process incoming data with minimal latency. While these engines are designed to process data quickly, they are not made to persist and manage it. Thus, databases are still integrated into streaming architectures, which often becomes a performance bottleneck. To overcome this issue and achieve maximum performance, all system components used must be examined in terms of their throughput and latency, and how well they interact with each other. Several authors have already analyzed the performance of popular distributed database systems. While doing so, we focus on the interaction between the SPEs and the databases, as we assume that stream processing leads to changes in the access patterns to the databases. Moreover, our main focus is on the efficient storing and loading of binary data objects rather than typed data, since in our use cases the actual data analysis is not to be performed by the database, but by the SPE. We've benchmarked common databases within streaming environments to determine which software combination is best suited for these requirements. Our results show that the database performance differs significantly depending on the access pattern used and that different software combinations lead to substantial performance differences. Depending on the access pattern, Cassandra, MongoDB and PostgreSQL achieved the best throughputs, which were mostly the highest when Apache Flink was used.

1 INTRODUCTION

The ongoing digitalization of all sectors of the economy as well as the rapid development of the Internet of Things are leading to an ever increasing number of Big Data applications. This includes the analysis of sensor data to control automated workflows as well as crowdsensing-based measurements and online interactions with millions of users. The goal of providing results in ever shorter time has led to the gradual disappearance of classic batch processing approaches such as Map Reduce and their replacement by real-time technologies like Stream Processing.

Despite the fact that the integration of databases into stream processing pipelines contradicts the idea of keeping the data flowing all the time, this cannot always be avoided in practical scenarios. In particular, when access to historical data is needed to calculate new results, the use of a database is necessary (Stonebraker et al., 2005). This is especially the case when data accesses are random or hard to predict and thus, historical and constantly arriving data sets become too big for buffers and volatile memory. Using persistent data storage will however usually lead to a significant performance decrease, as the underlying storage hardware has high access times in relation to the actual processing operations. For our use cases, earlier research has shown that the reading throughput doubles when a pure in-memory database is used (Weißbach, 2018). With respect to the overall performance of the system, it is therefore extremely important to select software components which offer the highest possible performance for the particular use case and which can be efficiently combined with each other.

One example of such an use case is the live analysis of crowdsensed traffic data, which we are investigating as part of our research. Both historical data sets and large amounts of permanently incoming sensor data are processed using stream processing. The individual data to be processed is only a few bytes in size, but belongs to long tracks and big datasets forming a whole. It is collected using GPS, gyroscopes, magnetometers and acceleration sensors. The amount of data is large and unbounded, so that our research interest is focused on the question which streaming architectures and databases are suitable for processing countless tiny records that are related to historical data. In this context, we consider the database

Weißbach, M., Hilbert, H. and Springer, T.
Performance Analysis of Continuous Binary Data Processing using Distributed Databases within Stream Processing Environments.
DOI: 10.5220/0009413301380149
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 138-149
ISBN: 978-989-758-424-4

as a pure storage system that should not perform any analysis tasks. Instead, analyses are performed in the stream processing applications using the manifold advantages of the SPEs for distributed data analysis. Therefore entire GPS tracks and other sensor data (binary data) should be stored and accessed quickly. So, CRUD operations, (especially read, insert, and update) are the only operations of importance to us.

The software development within our project has shown that database performance is the key to the overall performance of our processing. Unexpectedly, we discovered the effect that the performance characteristics of the databases change when they are accessed from streaming applications. Our resulting assumption is that stream processing changes the access patterns used to query the databases. This could result from the fact that the engines use windowing or micro-batching mechanisms, which lead to short interruptions between the individual processing steps. In addition to this presumed unusual access behavior, the direct stream handling confronts the databases with countless small queries, whose amount can constantly change and which usually would be bundled into larger transactions in a batch processing world. This results in a relatively uncommon and quite special access behavior for which the databases may not have been optimized.

To further analyze this behavior, we have performed extensive studies on the performance of distributed databases integrated in streaming applications. We assumed that a pure analysis of the databases, independent of the stream processing, would have possibly led to unreliable results for our use case, since the presumed access patterns, resulting from the stream processing, would not have been considered. Consequently, we have analyzed the interaction of common databases and SPEs on the basis of database queries typical for our use cases, in which we mainly work on binary data rather than typed data. Our study is focused on three research questions:

1) Which distributed databases are best suited for high-performance processing of binary data?

2) Is there a SPE that offers performance advantages regarding the integration of distributed databases?

3) Are there specific combinations of SPEs and databases that work more efficiently than others?

In this paper we present the results of this study, in which we have namely benchmarked the databases Cassandra, HBase, MariaDB, MongoDB and PostgreSQL across the SPEs Apex, Flink and Spark.

Within the scope of several measurement series, we have identified the weaknesses and strengths of the storage systems in distributed streaming environments when processing binary data in order to achieve a well tuned and balanced data processing with low latency and high throughput.

In the following, we will discuss the related work and introduce the examined software systems before our test setup is explained in detail. The results of these tests are presented and discussed afterwards. Finally, the results will be summarized and an outlook on our further research will be given.

2 RELATED WORK

The performance of SQL and NoSQL databases for Big Data processing has already been examined from several perspectives. The Yahoo! Cloud Serving Benchmark (YCSB) (Cooper et al., 2010) is widely used to test storage solutions based on a set of predefined workloads. It is further extensible with respect to workloads and connectors to storage solutions and can thus, serve as a base for comparative benchmarks.

In (Cooper et al., 2010) the YCSB was used to benchmark Cassandra, HBase, PNUTS and sharded MySQL as representatives of database systems with different architectural concepts. Hypothetical compromises derived from architecture decisions were confirmed in practice. For example, Cassandra and HBase showed higher read latencies for high-read workloads than PNUTS and MySQL, and lower update latencies for high-write workloads. While YCSB is designed to be extensible, the YCSB client directly accesses a database interface layer which does not support an easy integration in a benchmark for stream processing. Thus, we adopted several workloads for our benchmark but implemented it by ourselves.

(Abramova and Bernardino, 2013) analyzed MongoDB and Cassandra regarding the influence of data size on the query performance in non-cluster setups. They used a modified version of YCSB with six workloads. Their results showed that as data size increased, MongoDB's performance decreased, while Cassandra's performance increased. Cassandra performed better than MongoDB in most experiments.

In (Nelubin and Engber, 2013) the authors examined the performance of Aerospike, Cassandra, MongoDB and Couchbase in terms of differences between using SSDs as persistent storage and a purely in-memory data management. They also used the YCSB benchmark, with a cluster of 4 nodes. They found that Aerospike had the best write performance in distributed use with SSDs, while still offering ACID guarantees. However, the authors themselves state that this result is partly caused by the test conditions, which matched closely the conditions for which

Aerospike was optimized.

(Klein et al., 2015) examined the performance of distributed NoSQL databases, namely Cassandra, MongoDB and Riak. The focus was on a setup of 9 database servers, which were optimized for productive use to process medical data with a high number of reads and updates to individual health records. Using YCSB, different workloads were tested to collect results for both strong and eventual consistency. For Cassandra and Riak, they were able to verify that they achieve a slightly lower throughput when using strong consistency (for MongoDB not all experiments could be performed). Cassandra delivered the best overall performance in terms of throughput in all experiments, but had the highest average access latencies.

In (Fiannaca, 2015) it was investigated which database system achieves the best throughput when querying events from a robot execution log. The authors examined SQLite, MongoDB and PostgreSQL and finally recommended MongoDB because it provides good throughput and usability for robot setups with a small number of nodes or a single node only.

In (Ahamed, 2016) Cassandra, HBase and MongoDB were investigated with different cluster sizes for different workloads. Cassandra always delivered the lowest access latency and the highest throughput, followed by HBase and MongoDB.

In (Niyizamwiyitira and Lundberg, 2017) the performance of processing queries on trajectory data of mobile users with three data sets from a telecom company was investigated. The study included Cassandra, CouchDB, MongoDB, PostgreSQL and RethinkDB and was performed on a cluster of four nodes with four location-related queries and three data sets of different sizes. During testing, Cassandra achieved the highest write throughput when multiple nodes were used, while PostgreSQL achieved the lowest latency and the highest throughput in single node setup. MongoDB had the lowest read latency for all query types, but did not achieve such a high throughput as Cassandra. In addition, they found that reading throughput decreased with increasing record sizes, especially for random accesses.

While all studies examined the performance of databases in specific scenarios and domains, none of them addressed the questions of how efficient binary data can be accessed and how well databases perform in conjunction with SPEs. To the best of our knowledge, currently no studies are available that focus on databases used as persistent storage in stream processing. Thus, our study is conducted to fill this gap.

3 SOFTWARE

In the following, the considered SPEs and database systems are introduced.

3.1 Stream Processing

SPEs are software frameworks designed to process and analyze incoming unbounded data streams instantly. In this context we focus on Apache Apex, Apache Flink and Apache Spark Streaming, which we consider to be the most appropriate based on our requirements and which we have been examining in our research for a long time now. All three systems are widely used and have a large community.

3.1.1 Apache Apex

Apache Apex is a YARN-native platform for both stream and batch processing, developed under the Apache License 2.0. Apex consists of two main parts, the Apex Core, which is a platform for building distributed Hadoop applications, and Apex Malhar, which is a library of logic functions and connectors for third party software including databases like Cassandra, MongoDB, Redis and HBase. Apex is intended to enable the rapid development of high-performance, fault-tolerant applications that are typically built using Maven. Algorithms are modeled in Apex as directed acyclic graphs, whose nodes are called operators that represent the different data processing steps. The software provides end-to-end exactly-once processing based on checkpointing and an incremental recovery process.

DataTorrent, the company that played a major role in developing Apex, shut down in May 2018. Although the software continues to exist as an Apache project, there has been no new release since then. Despite this, our previous research has shown that Apex delivers good results in terms of latency and throughput for the use cases we are investigating, which is why we continue to examine the engine.

3.1.2 Apache Flink

Apache Flink is a framework provided under the Apache license 2.0 that supports batch and stream processing in a hybrid fashion. As a native streaming platform, Flink is a direct competitor to Apex and provides similar functionalities. A main difference to Apex is that Flink does not rely on YARN, although it can be used with it. While Flink doesn't depend on any Hadoop feature, it integrates well with many of the Hadoop components including HDFS and HBase. Flink can also be used on top of the Apache Mesos

cluster manager. The data processing workflows in Flink are also modeled in operators on the basis of an directed acyclic graph and are generally comparable with those of Apex. By default, Flink provides at-least-once processing. As Apex, Flink offers exactly-once processing on the basis of checkpointing.

3.1.3 Apache Spark Streaming

Apache Spark Streaming, released under the Apache License 2.0, allows streaming analysis based on a micro-batching approach. Thus the data processing model of Spark Streaming differs fundamentally from the native streaming solutions outlined before. Incoming data is not processed immediately, but collected in small "micro batches", which are then processed together. The basic idea of Spark is that data is stored and processed in so-called "resilient distributed datasets" (RDDs). A RDD is a read-only multiset of data items, which is distributed over a cluster of machines and thereby maintained in a fault-tolerant way. The Spark cluster consists of driver nodes that control the processing and tell the worker nodes what transformations they should perform on the data. Unlike Apex and Flink, Spark doesn't have operators that process the incoming stream continuously and store states. Instead, data processing takes place on the execution of various transformations on the RDDs. In comparison to native streaming solutions, micro batching approaches are usually associated with higher throughput during processing phase of the system, but also with higher latency. Like Flink, Spark can operate without Hadoop or using Hadoop YARN and provides exactly-once processing.

3.2 Databases

Our focus is on persistent data storage systems that are suitable for use in distributed systems. In our investigations we've examined Cassandra, HBase, MariaDB, MongoDB and PostgeSQL, as these are the systems that, after analysis of the related work, appear to be most suitable for our use cases while also having a large support and distribution in the community.

3.2.1 Cassandra

Apache Cassandra is a NoSQL wide column store, released under the Apache 2.0 license. Cassandra is designed for high scalability and reliability. It processes data as key-value pairs and distributes them evenly across the nodes by hashing the keys. The data can be managed using the Cassandra Query Language (CQL). Fault tolerance is provided through automatic data replication. In terms of the CAP theorem, Cassandra can be seen as an AP system, considering availability and partition tolerance as more important as consistency. To prevent the existence of a single point of failure, each Cassandra node has the same tasks and abilities. The nodes form a peer-to-peer network in which each node can be queried for data. If the data is not stored locally, the queried node routes the query to the responsible node.

3.2.2 HBase

HBase is a non-relational distributed database, modeled after Google's Bigtable (Chang et al., 2008) and released under the Apache 2.0 license. It is part of the Hadoop infrastructure, runs on top of the Hadoop Distributed File System (HDFS) and depends on Zookeeper. It can be seen as an abstraction layer on top of HDFS that provides several performance advantages for certain access patterns. HDFS itself operates on larger block sizes and is not well suited for managing lots of small files. HBase, on the other hand, is optimized to quickly manage small datasets within very large amounts of data and to quickly update frequently changed data. A HBase cluster consists of master and region servers. The master servers coordinate the data and job distribution in the cluster with the help of Zookeeper. The region servers store the actual data. Therefore tables are divided into sequences of rows, by key range, called "regions". These regions are then assigned to the region servers, which are spread across the cluster to increase the read and write capacities. To access data, clients communicate with region servers directly. With regard to the CAP theorem, HBase is a CP type system.

3.2.3 MariaDB

MariaDB is a relational database system that originated as a fork of MySQL and was published under the GPL. Many commonly used Linux distributions (f.e. Debian, Ubuntu, Arch, Fedora, CentOS, openSUSE and Red Hat) have replaced MySQL as their default database system with MariaDB, which is why MariaDB is nowadays considered more important than MySQL in the open source community. For a distributed use of MariaDB, the extension "Galera" has to be used, which replicates all databases to all servers of the cluster. Hereby a synchronous multi-master server setup is established in which each node can be contacted by clients for both read and write queries. MariaDB guarantees fail-safe operation by majority decisions between the servers. As long as more than the half of the servers of a cluster can interact with each other, the cluster is functional. If more servers fail or split off, the cluster stops operating un-

til enough servers are online again to achieve a quorum. In order to avoid so-called "split brain" states, it is therefore important that the total number of servers in a cluster is always odd. MariaDB is to be classified as a CA system with regard to the CAP theorem.

3.2.4 MongoDB

MongoDB is a document-oriented NoSQL database, that uses JSON-like documents with schema. It's licensed under the Server Side Public License (SSPL). The JSON-like data storage allows the creation of complex data hierarchies while maintaining the possibility of indexing and quering the data. MongoDB provides replication and sharding functionalities to ensure high reliability and availability. Data is stored in collections and distributed to the data nodes called "shards". For this purpose the data distribution can be freely configured using a config server (based on hash functions). Clients do not directly send their queries to the data nodes but to a router ("mongos"), which forwards the query to a responsible node according to its knowledge about data distribution. MongoDB is a CP system according to the CAP theorem.

3.2.5 PostgreSQL

PostgreSQL is a relational database published under the PostgreSQL license (similar to the MIT or BSD license). PostgreSQL supports transactions according to the ACID properties and is designed to be extensible. Thus, there are various extensions for the database, such as PostGIS, a software variant that allows the management of geographical objects. For distributed use, PostgreSQL can be used with multiple nodes configured as a master-slave setup. This means that write requests can only be sent to the master server, while read queries can be placed to all nodes. A multi-master replication is not natively supported, but there are third-party (open- and closed-sourced) tools for this purpose, which we have not investigated. As MariaDB, PostgreSQL can be classified as a CA system according to the CAP theorem.

4 BENCHMARKING DATABASES IN STREAMING PLATFORMS

Our research is initially motivated by a real-world scenario in which traffic data is to be processed live with the lowest possible latency. Since the related use cases are computer-intensive and the amount of data to be processed is big and unbounded, the use of stream processing and a distributed database is appropriate. However, realisation has shown that data access quickly becomes the biggest bottleneck in the streaming pipeline due to the necessary disk I/O. Due to this issue, we decided to investigate the performance of databases embedded in streaming architectures especially with regard to their processing capacities for small binary data sets. Our benchmark therefore addresses the specific problems of our use cases and uses data and file sizes as they are typical for them. Since the algorithms of the use cases should not influence the performance analysis, we have replaced them with simple mathematical operations, which do not require any significant CPU time for processing.

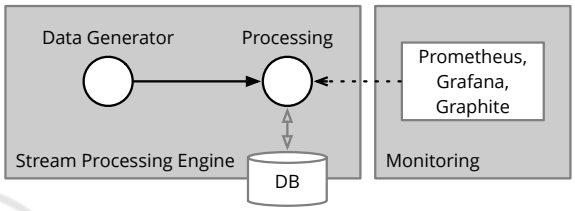

Figure 1: Benchmark Setup.

4.1 Benchmark Setup

The setup is depicted in Figure 1 and consists of the following components:

- The **stream processing engine** under review (Apex, Flink or Spark) runs the streaming application from which the database is accessed. The application logic is the same for all engines.
- The **data generator** generates pseudo-acceleration-sensor-signals, as they are typical for our use cases. Thereby, a single data record consists of a twelve character string, which is used as identifier, three integer values and three double values. For the individual test runs the number of records that the generator emits per second can be specified. The generator is an independently developed Java library that is integrated in the stream processing application during compilation. It is embedded in a different operator (task) than the data processing in order to obtain an access pattern typical for stream processing when querying the database. The use of a message queue such as Apache Kafka is purposely omitted as this could influence the measurements.
- The **processing operator** is also part of the stream processing application. It calculates results using data it receives from the generator, which are then stored in the database. If there already exists an older database entry for the given index, this

entry is loaded and taken into account for the result calculation. In addition, a binary large object (BLOB) must be stored with the data set, which can vary in size depending on the use case under consideration. Since we focus on the performance measurement of databases, we do not use the actual calculations from our use cases. Instead, the calculation of new values to be stored is limited to simple additions[1] of the new values and the possibly already existing previous values given in the respective field. We examine BLOBs of different sizes (1 Byte, 10,000 Byte, 100,000 Byte), as we are particularly interested in the performance of the databases with regard to the processing of binary data. These BLOB sizes well reflect various use cases in which we sometimes just want to store individual measurement values and sometimes entire measurement series in a data set. To keep the overhead low, we use prepared objects that are not recalculated for every write operation.

- The distributed **database** is tested in terms of achievable throughput for different access patterns and for read latencies when querying existing data records. We consider four access types for each of the previously specified BLOB sizes: Querying (reading) existing entries, inserting new entries, updating existing entries and a mixed-access pattern in which 50% of the entries are new inserts and 50% are updates of existing values.
- The **monitoring** is responsible for watching the application and logging the measurements. The necessary timestamps are collected in the streaming application, to capture the database access times. Counters are used to record the throughput, which are read and logged every second by the monitoring system.

Based on different access patterns, our research provides information about the suitability of several technology combinations for different use cases, as they are typical for our work, but also for other application areas. The **read-only** access pattern can be used to evaluate the performance of applications that access a database (almost) only for read operations, while it is not possible to predict what data will be requested next (random access). This is a common problem when dealing with IoT- or crowdsensing-data. The **insert-only** access pattern reflects applications that need to store data quickly without querying it again. For example, this can be the case if logging processes are to be implemented in a stream-processing pipeline. In the **update-only** pattern, data is read and then updated. This is typical for periodically running algorithms, such as logging or monitoring processes, in which new values for certain events have to be updated cyclically. The **mixed access** pattern combines reading stored information with inserting new values and updating existing values. Such an access pattern is typical for almost all software systems that interact with customers or employees.

The benchmark was performed on a cluster of six physical servers connected via 10GbE, each equipped with a 12 core Intel Xeon Gold 6136 processor (24 threads, 3.0 GHz normal clock speed, 3.7 GHz turbo clock speed), 360 GB RAM (DDR4) and 360 GB NVME-SSD memory. The deployment is illustrated in Figure 2. Ubuntu 18.04.2 LTS was used to operate the machines. Containerization based on Docker 18.09.7 was used to distribute the software within containers, orchestrated by Docker Swarm. As Apache Apex depends on Hadoop, we decided to use Flink and Spark on top of Hadoop as well, for reasons of comparability. Swarm was configured to use 24 threads per node, giving a total of 144 available threads. On one server, Zookeeper and the required monitoring tools (Grafana, Graphite Exporter, Prometheus) were installed. The monitoring was limited to use a maximum of two threads. An upper limit of 17 threads per available server was set for stream processing environments. The unused resources were available to the database application under review. The software systems were adjusted to the hardware setup and configured as recommended by the manufacturer tutorials. Further performance tweaks were not made to allow a fair comparison of the systems.

4.2 Benchmark Results

In the following we present the results of our measurements. Each experiment was performed with a runtime of 10 minutes. All system components were reset to their initial state between test runs. The results were calculated by averaging the measured values.

4.2.1 Preliminary Remarks

To avoid wrong conclusions being drawn from the results, two aspects should be noted before introducing them:
As pointed out before, we focused on analyzing the performance of databases concerning the management of binary data. Therefore all experiments were performed for different sizes of binary data attachments, even for the smallest possible ones with only a

[1] Since we assume that the specific access pattern is primarily caused by the processing techniques of stream processing (windowing, micro-batching), the actual operator logic does not matter, as long as the data changes, which forces the database to rewrite it.

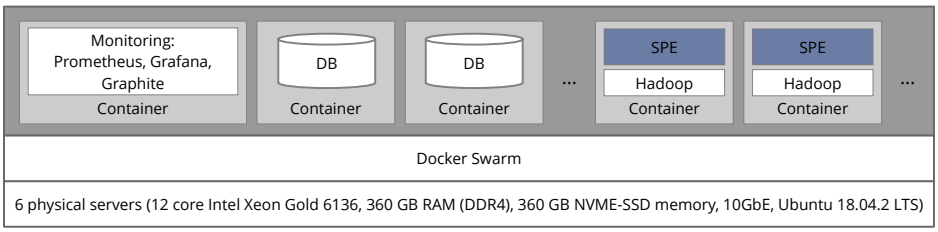

Figure 2: Benchmark Deployment.

single byte in size. It could be assumed that processing a database entry containing such a small amount of binary data hardly differs in performance from processing an entry containing no binary data at all, but this assumption is not always correct. Some database architectures generally manage and store binary data independently of typed data. Pointers to the binary parts are then stored with the corresponding database record. Consequently, querying data demands additional disk I/O, which increases the access latency even if only one byte has to be read. Thus, the results cannot provide reliable information on the performance of databases in managing exclusively typed data.

It should also be noted once again that we have exclusively examined CRUD operations. Neither joins nor complex queries or analysis functions have been investigated, so the following results don't say anything about how well the databases can handle them.

4.2.2 Read Performance

Figure 3 shows the throughputs achieved for reading previously stored data sets with different BLOB sizes (1 Byte, 10,000 Byte, 100,000 Byte). The first thing that stands out in comparing the three diagrams is that the difference in performance among the individual databases is more significant than that caused by using different SPEs. This finding is consistent for the majority of our experiments, although the following results show that there are also measurable differences in performance resulting from the choice of the SPE.

When reading small records with 1 byte or 10,000 byte BLOBs, Cassandra delivered the highest throughput with all the SPEs, followed by HBase and PostgreSQL. As expected, throughputs of all systems decrease with increasing BLOB size. HBase's read performance is by far the best for larger BLOBs of 100,000 bytes. A look at the corresponding access latencies in figure 7 shows that MongoDB, MariaDB and Cassandra can access small data sets (1 byte BLOB) very quickly and that the access times diverge more with increasing BLOB size, whereby Cassandra delivers the best results for 10,000 and 100,000 Bytes sized BLOBs.

4.2.3 Insert Performance

Figure 4 shows the results of the throughput measurements when inserting new database entries. It was ensured that the keys of the data records to be inserted did not previously exist in the database.

Cassandra achieved a significantly higher throughput for smaller BLOBs (1 byte, 10,000 bytes) than the other databases. The insert throughput for 10,000-byte BLOBs was noticeably higher when using Cassandra with the Apex compared to Flink and Spark. When inserting slightly larger data sets (100,000 byte BLOBs) PostgreSQL clearly performed best. Furthermore a performance advantage of the native streaming engines over Spark was evident for this database system. A comparison with figure 3 shows an inversed performance behavior of HBase and PostgreSQL. HBase achieves a high read throughput for larger data sets but a lower when inserting data, PostgreSQL reads data with a low throughput but inserts it with a high one. This underlines the need to make the choice of database system dependent on which access patterns are most relevant for the particular use case.

4.2.4 Update Performance

Figure 5 shows the results of the throughput measurements when updating existing data records, figure 8 shows the corresponding read latencies. Only records with keys for which older entries already existed in the database were used in the experiment.The generator used a fixed set of 1,000,000 records, which also caused repeated updates of the same records, offering advantages for databases with proper caching.

Again, Cassandra showed by far the best throughput for small data sets (1 byte and 10,000 byte BLOBs), followed by PostgreSQL and MongoDB. When it came to processing the larger 100,000-byte BLOB, Cassandra was also ahead with Apex and Flink, while MongoDB performed better than Cassandra on Spark. For records with a 1 byte BLOB, PostgreSQL and Cassandra provided the lowest latencies very close to each other. Interestingly, this was also the case for BLOB sizes of 100,000 bytes, but

for entries with 10,000 bytes, Cassandra was faster than PostgreSQL on all engines.

4.2.5 Mixed Access Performance

In the last experiment (see figure 6 for the throughputs and figure 9 for the latencies) a mixed access was simulated as it is typical for a lot of our actual use cases. Thereby data records were submitted, of which 50% had to be newly inserted into the databases, since no entry existed for the respective key and 50% were updates of existing values.

For the smallest BLOBs, Cassandra achieved a slightly higher throughput than PostgreSQL, for which performance drops significantly with the increasing BLOB size, evident when looking at the 10,000 byte BLOB processing. MongoDB takes second place behind Cassandra here, PostgreSQL comes third, but with a substantially lower throughput. When it comes to the largest BLOBs, MongoDB achieves by far the highest throughput. Considering the fact that Cassandra ranked second place in the insert-only test and first place in the update-only test, it is remarkable that it achieves such significantly lower throughput than MongoDB here, while ranking third behind PostgreSQL. On the other hand, Cassandra achieved the best reading latencies for all BLOB sizes, while the throughput winner MongoDB had very high latencies. This can be partly explained by the architecture of MongoDB, in which requests have to be forwarded from the routers ("mongos") to the data nodes first, resulting in additional network latencies.

4.3 Stream Processing Engine Performance

Although the results show that the selection of the database has greater influence on the performance, there are also significant performance differences depending on the SPE used.

Each SPE has been tested 60 times regarding throughput, covering five databases, each with four access patterns and three BLOB sizes. In order to quantify the real differences in the performance resulting from the selection of the SPE, we have introduced a scoring scheme in which, for each of these 60 test variants, one point was given to the SPE that achieved the best result. Table 1 shows the scoring.
In terms of throughput, 64 points were given as there were four experiments with two equal winners. Flink scored best with 46 points, followed by Apex (14 points) and Spark (4 points). Flink also got the most points for each individual database system, so that a recommendation can be made for this engine with regard to its interoperability with different databases.

Table 1: Number of Experiments in Which a Stream Processing Engine Achieved the Highest Throughput.

	Cassandra	MongoDB	PostgreSQL	MariaDB	HBase	Total
Apex	3	3	3	4	1	**14**
Flink	9	10	8	8	11	**46**
Spark	0	0	2	1	1	**4**

The scoring scheme was also applied to the latencies, which were investigated in 45 experiments per engine. A corresponding number of points was awarded, as can be seen in Table 2.

Table 2: Number of Experiments in Which a Stream Processing Engine Achieved the Lowest Latency.

	Cassandra	MongoDB	PostgreSQL	MariaDB	HBase	Total
Apex	3	1	2	4	6	**16**
Flink	5	3	4	1	3	**16**
Spark	1	5	3	4	0	**13**

For read latencies, there is an almost equal distribution of the points given to the SPEs, albeit Spark scored marginally lower. Obviously the latencies depend mainly on the database system used. Consequently, we do not recommend a particular SPE here.

4.4 Database Performance

Table 3 shows the best databases in terms of throughput achieved for each of the twelve experiments, together with the SPE used in the particular experiment. In some of the experiments, the measurement results of the first-placed technologies were very close to each other, hence we also show the second bests in the table.

Table 3: Software Combinations That Achieved the Highest Throughput for the Specific Access Patterns.

Workload	Best Throughput	Second Best throughput
Read 1B	Cassandra / Flink: 97,300	Cassandra / Apex: 94,500
Read 10,000B	Cassandra / Apex: 49,300	Cassandra / Flink: 48,500
Read 100,000B	HBase / Flink: 19,900	HBase / Apex: 17,500
Insert 1B	Cassandra / Flink: 227,100	Cassandra / Apex: 222,900
Insert 10,000B	Cassandra / Apex: 38,500	Cassandra / Spark: 31,300
Insert 100,000B	PostgreSQL / Apex: 4,700	PostgreSQL / Flink: 4,600
Update 1B	Cassandra / Flink: 86,200	Cassandra / Apex: 85,400
Update 10,000B	Cassandra / Flink: 30,400	Cassandra / Apex: 30,200
Update 100,000B	Cassandra / Flink: 3,000	Cassandra / Apex: 2,900
Mixed 1B	Cassandra / Flink: 77,200	Cassandra / Apex: 76,900
Mixed 10,000B	Cassandra / Apex: 27,400	Cassandra / Spark: 27,200
Mixed 100,000B	MongoDB / Flink: 3,900	MongoDB / Flink: 3,500

There are some observations resulting from the throughput analysis:

- Inserts are processed faster than updates, but a mixed access pattern that includes inserts and updates is even slower than update only access.
- As was to be expected, the achievable throughput is indirectly proportional to the data set size.

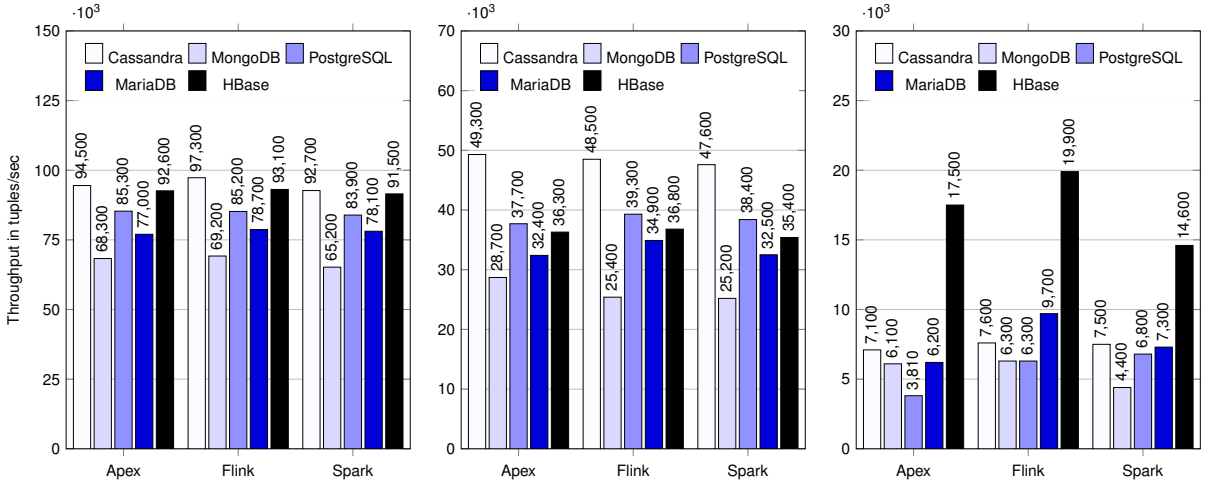

Figure 3: Throughputs Achieved during the Read-Only Test Runs (BLOB Sizes: 1 Byte, 10.000 Byte, 100.000 Byte).

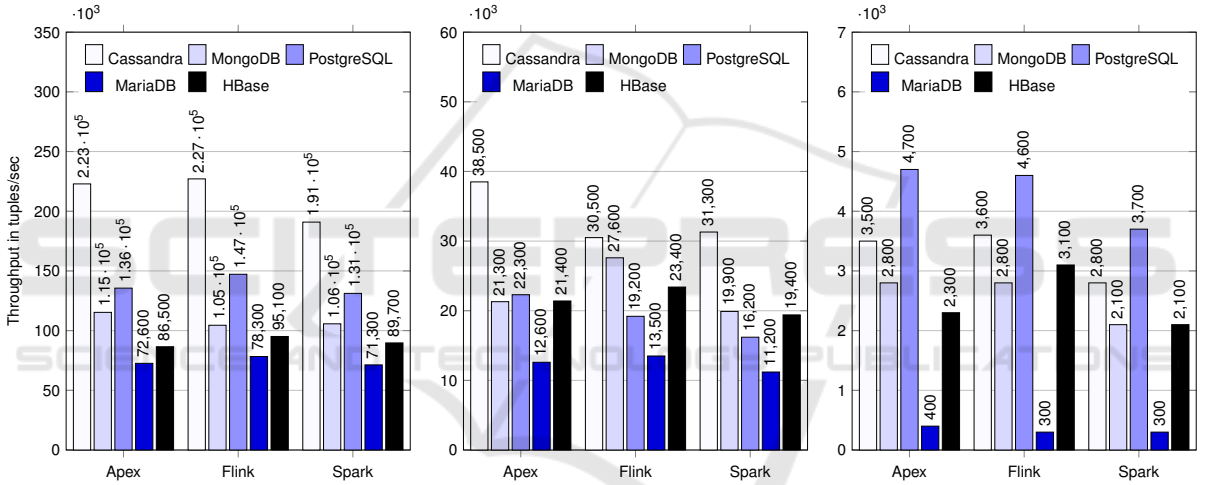

Figure 4: Throughputs Achieved during the Insert-Only Test Runs (BLOB Sizes: 1 Byte, 10.000 Byte, 100.000 Byte).

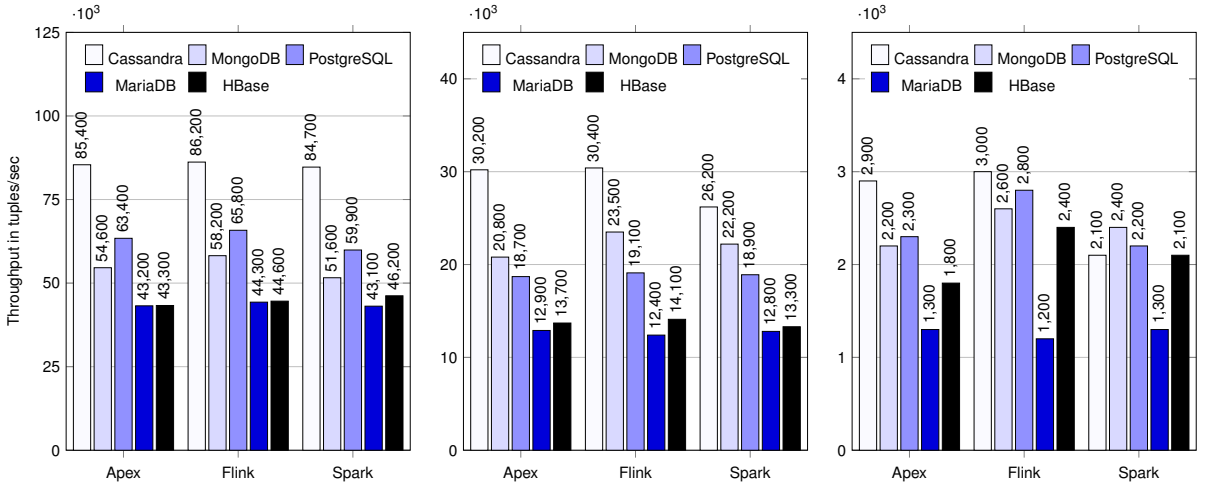

Figure 5: Throughputs Achieved during the Update-Only Test Runs (BLOB Sizes: 1 Byte, 10.000 Byte, 100.000 Byte).

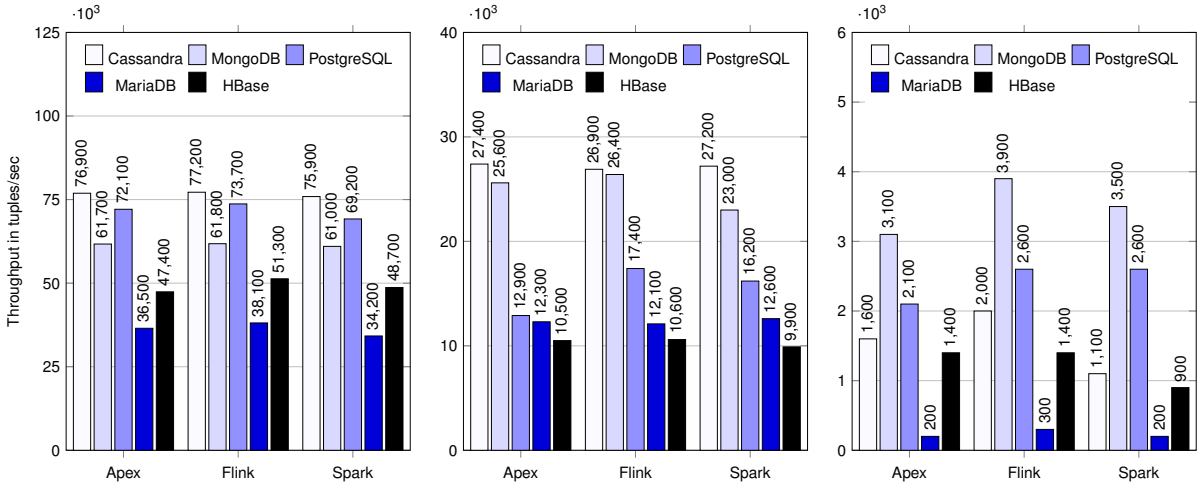

Figure 6: Throughputs Achieved during the Mixed-Access Test Runs (50% Inserts / 50% Updates, BLOB Sizes: 1 Byte, 10.000 Byte, 100.000 Byte).

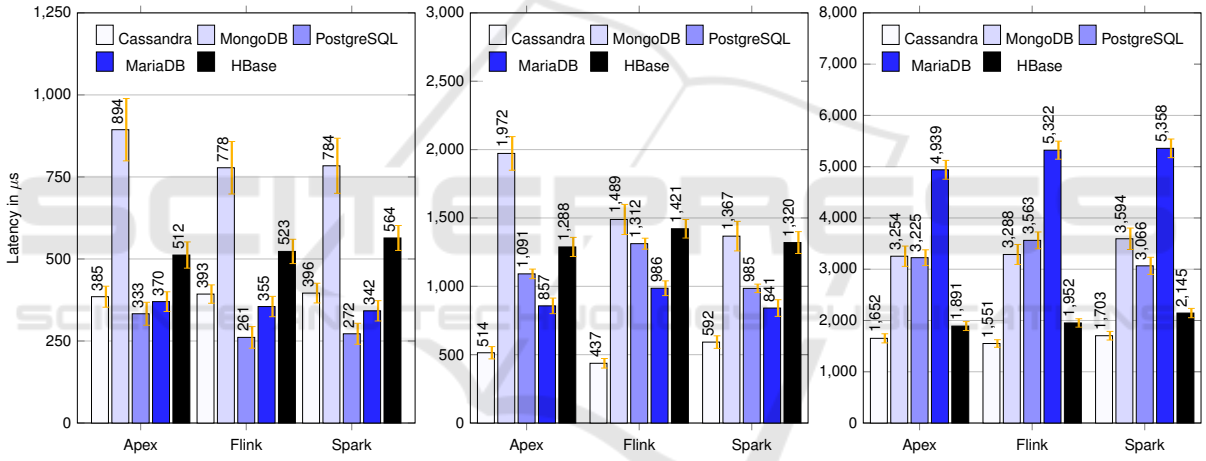

Figure 7: Read Latencies during the Read-Only Test Runs (BLOB Sizes: 1 Byte, 10.000 Byte, 100.000 Byte).

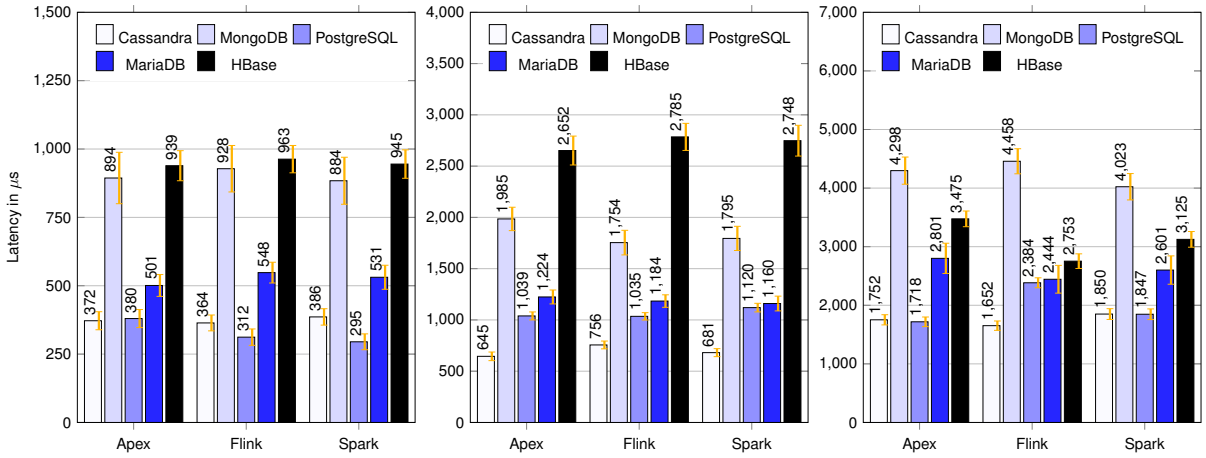

Figure 8: Read Latencies during the Update-Only Test Runs (BLOB Sizes: 1 Byte, 10.000 Byte, 100.000 Byte).

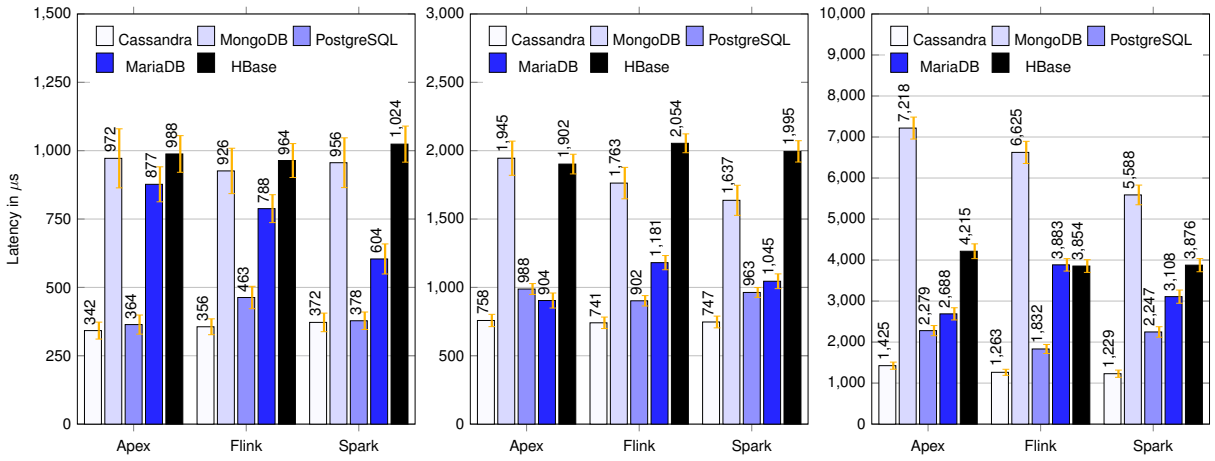

Figure 9: Read Latencies during the Mixed-Access Test Runs (50% Inserts / 50% Updates, BLOB Sizes: 1 Byte, 10.000 Byte, 100.000 Byte).

In some cases, it can even be more efficient to split up large BLOBs into smaller chunks and to manage them with more queries (and a higher query overhead). For example, Cassandra processed 4,000,000 bytes more per second in the update-only workload with 10,000-byte BLOBs than when processing the 100,000-byte ones.

- Cassandra is a good choice for managing records with small binary data. If larger BLOBs are to be handled, the choice of the appropriate technology depends more on the access pattern used.
- Native SPEs (Apex and Flink) are better suited for the problems considered in our experiments than Spark's microbatching approach, as they achieved higher througputs in almost all experiments.
- MariaDB only achieves a comparatively low throughput for all write access patterns, but is in the mid-range for read accesses. The cause of this could be the fact that the Galera-based multi-master system was primarily designed for data integrity (replication) and not for the fastest possible accesses. While no other nodes need to be queried to perform read operations, a successful write operation does.

For the latencies, the best databases are shown in table 4 without mentioning the SPE used in the experiments, since it has been shown that the latencies hardly differ with regard to them.
The analysis of the standard deviations that can be seen in figures 7-9 shows that the latencies are quite stable for almost all databases. MongoDB has more variance in the read-only workload than the other databases, whereby it also increases with increasing BLOB size for HBase, PostgreSQL and MariaDB. In the update-only workload, PostgreSQL and HBase have the highest variances, but they are slightly less significant with increasing BLOB size for HBase, while they increase for MariaDB. In mixed access, PostgreSQL, HBase and MariaDB have the highest variations again, but the effect decreases significantly with increasing BLOB size, at least for MariaDB.

Table 4: Databases That Achieved the Lowest Read Latency for the Specific Access Patterns.

Workload	Best Latency	Second Best Latency
Read 1B	PostgreSQL: 261 μs	MariaDB: 342 μs
Read 10,000B	Cassandra: 437 μs	MariaDB: 841 μs
Read 100,000B	Cassandra: 1551 μs	HBase: 1891 μs
Update 1B	PostgreSQL: 295 μs	Cassandra: 364 μs
Update 10,000B	Cassandra: 645 μs	PostgreSQL: 1035 μs
Update 100,000B	Cassandra: 1652 μs	PostgreSQL: 1718 μs
Mixed 1B	Cassandra: 342 μs	PostgreSQL: 364 μs
Mixed 10,000B	Cassandra: 741 μs	PostgreSQL: 902 μs
Mixed 100,000B	Cassandra: 1229 μs	PostgreSQL: 1832 μs

Cassandra delivered the best read latencies in most of the experiments, remaining very stable across all workloads, which supports its recommendation.

5 CONCLUSIONS

We investigated the interaction of three SPEs with five databases in twelve different experiments each, and thus performed a total of 180 different experiments. As expected, the selected database system has a greater influence on the achievable throughput than the SPE. However, there were measurable differences resulting from the choice of the SPE, which can be clearly seen from the fact that Apache Flink performed slightly better in almost all experiments than Apache Apex and Apache Spark. This confirms our assumption that the SPE affects the access pattern to

the database.

The analysis of the reading latencies showed a different result, which is that the choice of the SPE has no significant influence on them. Both results make sense, as the overall throughput highly depends on the interaction between the SPE and the database system, in which the access pattern is influenced by the streaming-typical (window-/micro-batching-based) data processing, while the latency of the individual database queries is not directly affected by these effects. For most use cases, especially those using data sets with small BLOBs up to 10,000 bytes in size, the combination of Flink and Cassandra is recommendable, although this finding, like all others, only refers to CRUD operations, since we have not conducted any further data analysis with the databases.

When managing larger binary data entries (100,000 bytes), the type of access is more relevant for the choice of database system. For reading intensive applications the use of HBase (with Flink) is recommendable here, which however requires HDFS (and therefore Hadoop) when used distributed.[2] An Hadoop-free alternative is to use MariaDB with Galera and Flink for this. In use cases where a lot of data is to be inserted but does not need to be accessed frequently, PostgreSQL used together with Apex or Flink achieves high throughputs. If data is to be updated frequently, Cassandra also scores with the larger BLOBs and achieves the best performance on Flink and Apex. In case of a mixed access from insert and update operations (which include the prior reading of the data), the use of MongoDB and Flink can be recommended.

6 FUTURE WORK

We plan to expand our investigations in this area. It is considered to further analyze the influence of stream processing on the query patterns and to derive optimization recommendations from these analyses. In addition, we intend to investigate the performance of more complex queries (analyses of geodata, typical for our use cases) and thereby consider in-memory grids in addition to the existing databases.

[2]HBase also provides a standalone mode that doesn't rely on HDFS, but cannot be used distributed.

ACKNOWLEDGEMENTS

This work is financed by the German Federal Ministry of Transport and Digital Infrastructure (BMVI) within the research initiative *mFUND* (FKZ: 19F2011A).

REFERENCES

Abramova, V. and Bernardino, J. (2013). Nosql databases: Mongodb vs cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 14–22, New York, NY, USA. ACM.

Ahamed, A. (2016). Benchmarking top nosql databases. Master's thesis, Institute of Computer Science, TU Clausthal.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA. Association for Computing Machinery.

Fiannaca, A. J. (2015). Benchmarking of relational and nosql databases to determine constraints for querying robot execution logs [final report].

Klein, J., Gorton, I., Ernst, N., Donohoe, P., Pham, K., and Matser, C. (2015). Performance evaluation of nosql databases: A case study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, PABS '15, pages 5–10, New York, NY, USA. ACM.

Nelubin, D. and Engber, B. (2013). Ultra-high performance nosql benchmarking: Analyzing durability and performance tradeoffs. *White Paper*.

Niyizamwiyitira, C. and Lundberg, L. (2017). Performance evaluation of sql and nosql database management systems in a cluster. *International Journal of Database Management Systems*, 9:01–24.

Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47.

Weißbach, M. (2018). Live traffic data analysis using stream processing. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 65–70.