An Experimental Analysis of Different Key-Value Stores and Relational Databases

David Gembalczyk[1] Felix Martin Schuhknecht[2] Jens Dittrich[3]

Abstract: Nowadays, databases serve two main workloads: Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP). For decades, relational databases dominated both areas. With the hype on NoSQL databases, this has changed. Initially designed as inter-process hash tables, some key-value store vendors have started to tackle the area of OLAP.

In this performance study, we compare the relational databases PostgreSQL, MonetDB, and MonetDB with the key-value stores Redis and Aerospike in their write, read, and analytical capabilities. Based on the results, we investigate the reasons of the database's respective advantages and disadvantages and examine whether key-value stores can serve as the storage-layer within relational databases. Additionally, we evaluate storing denormalized values versus rebuilding them via join and show that in certain cases key-value stores have an advantage when processing OLAP queries.

Keywords: Relational Systems, Key-Value Stores, OLTP, OLAP, NoSQL, Experiments & Analysis

1 Introduction

Nowadays, databases are almost present everywhere. Obvious application areas are for instance Big Data Analytics or back-ends for e-commerce systems. They are also used in more hidden places such as session storage for webservers or in embedded systems like smartphones or activity trackers. Although databases are applied in so many places, they serve mainly two workloads: Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP). Relational databases have started their triumphant advance after the introduction of the relational model in the area of the databases in 1970 [Co70]. During the decades plenty relational database have been developed. Some of them focused only on OLAP or only OLTP deploying specific optimizations, respectively. Although a variety of other database types have emerged until today relational ones dominate the area [DB16].

Since around 2009, a different category of databases appeared and became hyped, the so-called NoSQL databases [SF12, pp. 9-12]. Within this category, one group of competitors for the domain of OLTP are key-value stores. Initially intended to serve just as inter-process hash tables, speaking on a high level, they provide today much more functionality than just storage and retrieval of key-value pairs. Some vendors, such as Aerospike [Ae16], even started to tackle the area of OLAP.

[1] Saarland University, Information Systems Group, Campus E1.1, 66123 SB, s9dagemb@stud.uni-saarland.de
[2] Saarland University, Information Systems Group, Campus E1.1, 66123 SB, felix.schuhknecht@infosys.uni-saarland.de
[3] Saarland University, Information Systems Group, Campus E1.1, 66123 SB, jens.dittrich@infosys.uni-saarland.de

Regarding the recent development, two questions emerge: first, what distinguishes these systems besides the way of storing their content and what are their advantages and disadvantages? Second, could relational databases use a key-value store as underlying storage-layer while preserving the advantages of both approaches: mainly the key-value store's flexibility and the relational database's optimization towards fast OLAP? To provide an answer to both questions, we compare PostgreSQL, MonetDB, and HyPer as relational databases and Redis and Aerospike as key-value stores. Additionally, we use two data types, Hstore and JSONB, within PostgreSQL to simulate the behavior of Redis and Aerospike in order to get a better understanding of how key-value stores differ from classical row-stores. While gaining the flexibility of key-value stores, both simulations still preserve full SQL functionality provided by PostgreSQL. We make a performance study to compare the aforementioned databases and simulations in three categories: write, read, and analytical queries. In the first category, we investigate how fast these databases can insert and delete content. The second category focuses on various methods to read content. Besides of simple selects, we examine the efficiency of secondary indexes. Furthermore, we look into the actual gain of retrieving denormalized values from key-value stores in contrast to rebuilding them from relational databases using join operations. The last category utilizes queries which are provided by TPC-H [Co14] and compares the databases' OLAP capabilities.

1.1 Relational Databases

The basic principles of relational databases reach back to 1970 when E. Codd introduced the relational model [Co70] in the area of databases. The ultimate goal pursued by this model was a real separation between *what* the database has to store and *how*. Tables represent the foundation of the relational model. They consist of columns on the horizontal dimension and rows on the vertical. Columns describe the schema according to which each value has to be stored in the table. On top of the relational model, "A Structured English Query Language" [CB74] (SEQUEL, later SQL) was introduced in 1974. Until today, it is the de facto standard interface to operate relational databases. In this study, we inspect the following three relational systems:

1. **PostgreSQL** [Po16]: PostgreSQL is a disk-based relational database and uses row-oriented tables. To serve multiple clients it relies on forked child processes. PostgreSQL uses multiversion concurrency control to manage concurrent access to its content. Furthermore, it has two data types which are of special interest for this work: Hstore and JSONB. Speaking on a high level, Hstore works similar to a hash table, which stores an arbitrary amount of key-value pairs, and still is one value in a column. JSONB allows to store any valid JSON document as a parsed, binary representation. The main advantage compared to Hstore is the possibility to store values of a more complex and nested structure instead of simple key-value pairs. However, both in common allow indexing of single attributes similar to indexing columns.

2. **MonetDB** [Mo16a]: From the very beginning, MonetDB has been developed with a clear focus on OLAP queries and large data sets. Therefore, it is designed as a disk-based database with a column-store as underlying storage-layer. For concurrent data access MonetDB uses optimistic concurrency control which "is particularly useful for read-heavy applications" [Mo16b]. Furthermore, threads are not only utilized to provide parallel

database connections but also for intra-query optimization. Whenever possible, MonetDB will split a query across multiple threads to reduce the total response time. To speed up OLAP queries, MonetDB does not rely on user defined secondary indexes. Instead, it decides based on the query whether it will create an index or not.

3. **HyPer** [KN10]: HyPer is designed as pure in-memory database and fully ACID-compliant [HR83]. By default HyPer uses column-oriented tables but it also supports row-oriented tables. One of its key features is query compilation. Instead of generating an operator tree at run-time to accomplish a request HyPer generates and compiles data-centric code to be executed. HyPer utilizes a different perspective on threading and combines advantages of single-threaded systems with those of multi-threaded ones. Therefore, the data need to be split into exclusive, smaller partitions and one shared partition. Transactions using only one exclusive partition or just reading from the shared partition are executed in parallel whereas transactions across partitions or updating the shared partition are executed sequentially. Concurrent data access is managed with multiversion concurrency control. The latest available version of HyPer [KN15] emulates the PostgreSQL wire protocol allowing to utilize it with common database tools and even with JDBC using the PostgreSQL drivers.

1.2 Key-Value Stores

Key-value stores work similar to large hash tables and provide the basic functionality of inserting, reading, and deleting values according to a unique, user defined key [SF12, pp. 81-88]. In the simplest approach the database client serializes values to a BLOB, for instance a simple byte array, and the database stores these serialized values. They do not make any assumptions on what explicitly is stored inside a value or how exactly it is structured. Furthermore, another feature of nearly all key-value stores is scalability across several nodes in a cluster. We will look at the following key-value stores in the following:

1. **Redis** [Re16]: Redis offers several data structures in which the client can store its values and functions to operate on them. These data structures are never transferred between the database and the client application. Instead, Redis can only be instructed on how to manipulate or query these structures. Strings are the only scalar value and lists, sets, sorted sets, or hashes are some of the more complex structures, which only allow string values as their child elements. Other characteristics of Redis are: it is a pure in-memory database and within an instance all OLTP requests are handled by a single thread. To perform more complex tasks on the database side Redis provides the ability to create user-defined functions (UDF) by using LUA scripts.

2. **Aerospike** [Ae16]: In contrast to Redis, Aerospike is a highly multi-threaded key-value store. For each group of operations a different group of threads is responsible. Aerospike utilizes a hierarchical data model. At the top of its data model are namespaces. Within these namespaces records may be optionally grouped together in sets. Records are the key-value pairs in Aerospike and uniquely identified by the namespace, the optional set, and a key. Each record consists of one or more bins which represent the attributes of a value. This model allows to draw an analogy between the relational and Aerospike's terminology: namespaces are databases, sets are tables, records are rows and bins are columns. Aerospike offers also the ability to create user-defined functions with LUA to

fulfill complex tasks. Another feature is the support for secondary indexes on a per bin basis within a set or a namespace. To avoid problems with the schema-free philosophy within Aerospike secondary indexes are typed. If the bin does not exist within a record or is not of the same type as the index then the record will not be indexed.

1.3 Related Work

In the past years, there has been work on comparing systems of different types. In [AMH08] the authors try to answer the question how different column-stores and row-stores are. C-Store [St05] as column-store and a not further announced commercial System X as row-store are used as test subjects. They find that column-stores can be simulated by row-stores to a certain degree and thus speed-up the queries. However, compression and late materialization used in real column-stores have a beneficial influence which cannot be simulated. The authors of [Kl15] compare Riak, MongoDB [Mo16c], and Cassandra as representatives of the NoSQL-groups key-value store, document store, and column store respectively using the YCSB benchmark. One key point in this comparison is the influence of different consistency assumptions within a cluster of nine nodes. Findings are that for MongoDB switching from eventual consistency to strong consistency reduces the performance by up to 25%. A performance comparison between Microsoft SQL Server Express and MongoDB [Mo16c] is made in [PPV13] based on a custom benchmark. According to their findings MongoDB is faster with OLTP queries using the primary key whereas the relational database is better with aggregate queries and updates based on non-key attributes. In [Fl12] the authors draw a comparison between Microsoft's relational database Parallel Data Warehouse (PDW), the document store MongoDB [Mo16c], and the Hadoop based solution Hive [Hi16]. In the TPC-H benchmark PDW is for smaller data sets up to 35 times faster than Hive. For the largest set PDW is still 9 times faster. PDW and Hive are faster than both MongoDB versions (one with client-side sharding and one with server-side sharding) which "comes in contrast with the widely held belief that relational databases might be too heavyweight for this type of workload" [Fl12].

1.4 Comparing Apples and Oranges

Comparing two databases, by all means, is not an easy task and may result in comparing apples to oranges. The first and maybe most obvious point is comparing disk-based with in-memory systems. To minimize this discrepancy, we do the following: first, we store the database files on a RAM disk to improve at least the disk access times. Second, if possible we enlarge the caches and make sure the benchmark makes the same requests during each run. During the first run the database will load most of the data into the caches. Afterwards, we omit the first run from the results. Another difficulty are the different client interfaces used to communicate with the servers. SQL in connection with a programming language specific database binding is the de facto standard to use relational systems in client applications. In the case of Java it is JDBC and a database specific driver. In contrast, as a result of the variety of features, almost all key-value stores have their own client libraries, sometimes even multiple different libraries such as in the case of Redis. Finally, it remains the question how to compare single and multi-threaded databases. Well, there are multiple ways to enforce a single-threaded usage or to simulate multi-threading using multiple local instances and client-side sharding. Either way, it is unfair to one or another because they

are particularly designed with one of these two concepts in mind. We will discuss these comparison issues in the respective experiments in more detail.

2 Experimental Setup

All experiments are performed on a machine equipped with two Intel Xeon X5690 hexacore CPUs running at 3.47 GHz with 192 GB DDR3-1066 RAM. All BIOS settings are set to default. In total the system runs with 24 hardware threads. The installed operating system is Debian 8.3 with kernel version 3.16.0-4-amd64 and openjdk 1.7.0 64-bit is used as java runtime environment. The following versions of the databases and their bindings are used:

- PostgreSQL 9.5.2 with PostgreSQL JDBC driver 9.4.1207
- MonetDB 11.21.13(Jul2015-SP2) with MonetDB JDBC driver 2.19
- HyPer 0.5 demo with PostgreSQL JDBC driver 9.4.1207
- Redis 3.0.6 with java client jedis 2.8.0
- Aerospike 3.7.2 community edition with java client 3.1.8

The durability features of the systems are turned off, if possible. Additionally, PostgreSQL's configuration is adjusted with the recommendations by pgTune for type `OLTP` and 40 clients. Aerospike is configured to use all 24 threads provided by the processors and the maximum batch size is increased to 10,000. Both the benchmark client and the database server are running on the same machine.

2.1 A Custom Benchmark

The simplest way would be to use and extend YCSB (Yahoo! Cloud Serving Benchmark) [Co10]. Nevertheless, we are going see in the first experiment that YCSB has one major disadvantage: it has a rather poor performance. The reason for this lies in its design decisions, which are aimed at providing flexibility and extensibility. Therefore, it is necessary to create a custom benchmark tool to address these problems. A requirement besides the high throughput is scalability in terms of concurrent clients and batch sizes. While YCSB has also support for multiple clients it lacks the support for grouping multiple queries of the same type into one batch. Using batches reduces in total the amount of requests send to the database and in consequence the measured overhead by network IO.

As foundation for the benchmarks a slightly modified TPC-H schema is used. An additional attribute has been introduced into both the `partsupp` and `lineitem` tables which serve as artificial primary keys. This change is made rather for the relational databases than for the key-value stores. A concatenated key might result in multiple comparisons, for each part of it, whereas in key-value stores always a single value is used as primary key. On top of it the provided generator tool is used to generate the test data. Overall the new benchmark tool is split into three main parts: (1) The query generator is responsible for creating all query data in a generic way. (2) The query converter, as the name states, converts the generic query data as far as possible into actual database specific statements. Afterwards, these are stored in a shared queue. (3) The actual query threads get the statements out of the shared queue and perform the queries. Besides, in case of PostgreSQL and MonetDB prepared statements are used. The available HyPer demo lacks of support for prepared statements. Therefore, usual unprepared statements are used.

2.2 Database Schema

In general, seven database variants are subjects in the following experiments: the relational ones are PostgreSQL, HyPer, and MonetDB which are respectively denoted as **PG-row**, **HyPer**, and **MonetDB**. In addition, for PostgreSQL we test also the Hstore and JSONB data types denoted as **PG-hstore** and **PG-jsonb**. With these two data types we are able to simulate the behavior of Redis and Aerospike with PostgreSQL. The key-value stores Aerospike and Redis are denoted as **AS-simple** and **Redis**.

As mentioned previously a modified TPC-H schema is used for the benchmarks. It is obvious that this schema is replicated inside the relational databases. Nevertheless, the schema needs to be mapped to both key-value stores, PG-hstore, and PG-jsonb. For Redis all records are stored in one table. The keys consist of the table name and the primary key value, for example `nation3`, `customer146`, or `lineitem5890412`. Hashes are used as data structures for the values with the column names as the corresponding field names. For Aerospike each table goes into a corresponding set and the primary keys of the rows serve as the keys for the records within the sets. The columns of each table are also mapped to corresponding bins in each record. For PG-hstore and PG-jsonb a similar schema with all eight tables is used but all tables have only two columns. One is the primary key and the other is the value, which takes all the attributes. All three PostgreSQL variants are stored within separate databases.

In addition to these seven variants, in two experiments we use aggregated variants of AS-simple and PG-jsonb, which are denoted as **AS-complex** and **PG-complex**. The only difference within these variants is that the `customer`, `orders`, and `lineitem` tables are aggregated into one large, denormalized value. Each customer contains all its orders and these again contain all their related line items.

3 Experimental Analysis

Each experiment contains one or more benchmark types. For each database variant the benchmark performs four consecutive runs, one warm-up and three measuring runs. During each run, 50,000 operations are executed. In case of the read experiments, each run performs exactly the same set of operations. Before each experiment is conducted all databases are initially loaded with content from the generator tool used for TPC-H with SF 1. For MonetDB an extra warm-up run is necessary because of its caching behavior. It decides based on the query whether to create caches or not. In return, not all warm-up runs enforce the creation of caches. In this additional run 1,000,000 rows are selected using the primary key. The benchmarks are executed either for an increasing amount of concurrent clients using one single operation per request or for an increasing batch size using only one client. In the batch benchmarks the amount of required requests is reduced according to the batch size while the amount of operations stays the same. In general, all test results are presented in terms of performance, represented by operations per second, which means higher is better. Also each database variant is represented with consistent colors throughout the remainder of this paper and each database has consistent line markers.

3.1 Setting the Baseline

Benchmarks are usually done by measuring the time how long it takes until a request is processed and the result is returned by the database. This leads to the problem that measured time also includes overhead of round trips, data transmission, and the used database client library. Depending on the design, the benchmark tool itself can be also part of the overhead. These are all factors which may hide the real performance of a database especially in cases where the overhead of a request takes more time than processing it by the database.

```
//echo operation for PostgreSQL, MonetDB, and HyPer
jdbc.executeQuery("SELECT 1");
//echo operation for Redis
redis.echo("1");
//existence check for Aerospike
aerospike.keyExists(nonExistingKey);
```

List. 1: Examples of echo queries for JDBC and Redis and the existence-check for Aerospike.

One very simple method to get a good estimate of the overhead is to measure the time of simple echo operations, shown in Listing 1. Thus, a full request to the database is done without touching any data or performing any real operation. Additionally, to prove whether the measured performance is bounded by the benchmark at all, the same experiment is made without any database, denoted as **NO-DB**. Instead of querying the database just a counter is increased while the rest remains *exactly* the same. This shows us the maximum amount of operations which can be processed in one second giving us a clear indication if the results are bounded. With this experiment we pursue three goals: (1) Getting an impression of how many requests the databases are able to handle. (2) Proving that our benchmarks are actually measuring the database and generate valid results. (3) Deriving an estimate for the overhead to adjust following results.

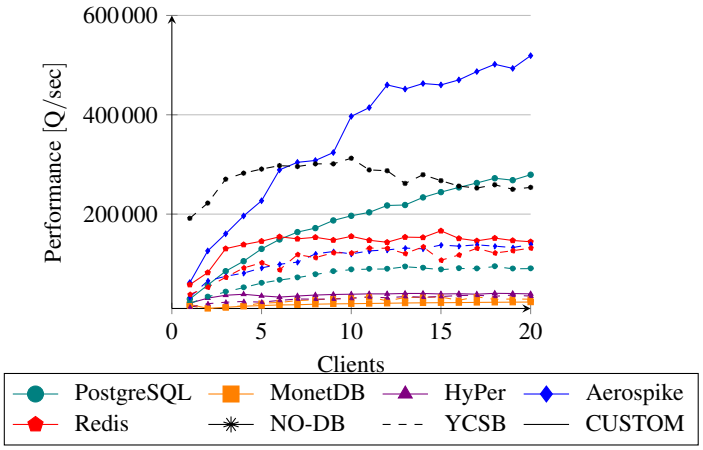


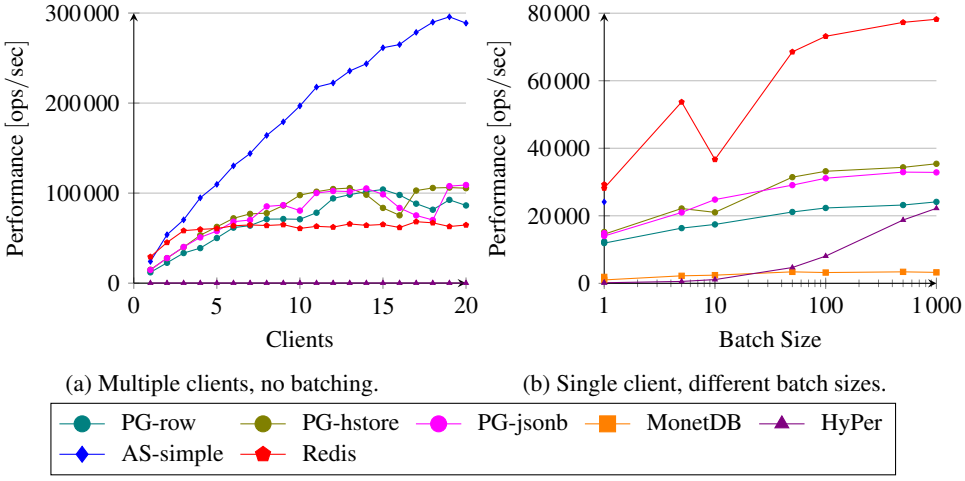
Abb. 1: **Echo Performance**. 50,000 echo requests are performed using multiple clients. **YCSB** are the results from the YCSB tool and **CUSTOM** are the results from our custom benchmark. The results of CUSTOM for NO-DB are omitted because they are beyond 4 million operations per second.

First, let us have a look on the results for YCSB in Figure 1: NO-DB shows a really bad performance given the fact that absolutely no database is involved. Thus, this is the maximal throughput which YCSB is able to achieve. The results of both benchmark tools for PostgreSQL, Redis, and Aerospike prove that their performance in YCSB is bounded by the benchmark itself: (1) Regarding YCSB, the performance of these three databases is lower than for NO-DB. (2) In the custom benchmark they show a far better throughput. The difference for Redis is much lower but this only due to the fact that it is bounded by a single thread. (3) MonetDB and HyPer perform in both benchmarks equally. Thus, their throughput is obviously bounded by the database. Therefore, in following experiments we are not going to take YCSB into account anymore. The results of the custom benchmark in Figure 1 show that Aerospike and PostgreSQL are the best scaling databases in each group. Although Aerospike checks whether a key exists or not it is nearly twice as fast as PostgreSQL. However, this is not completely related to JDBC being an inefficient database binding. If one would add the lost 25% to the results from PostgreSQL there would still be a gap of at least 100,000 operations per second. This may be explained by the fact that the existence check is compiled into Aerospike and thus optimized by the compiler. In PostgreSQL an operator tree, which is created once for the prepared statement at run-time, processes the echo request. Furthermore, the results also reveal the downside of being only single-threaded. Redis reaches really fast its maximum throughput while PostgreSQL and Aerospike keep scaling along with the amount of clients. The overall throughput of MonetDB is rather bad and it handles multiple clients not very well. With the second client the performance decreases by almost 45%. This may be explained by the fact that MonetDB is optimized for OLAP and not for high frequent, short OLTP. HyPer's poor performance derives from the overhead by the query compilation. In this case the compilation takes up to 99% of the time to process one echo. The lack of support for prepared statements makes this even worse.

3.2 Write Experiments

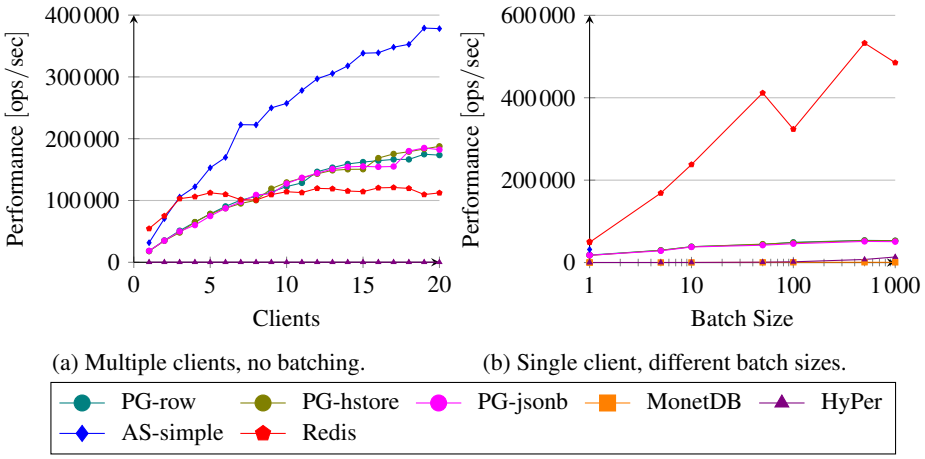
This category focuses on various methods to manage the database's content. Particularly, we investigate on inserting and deleting from `orders` and `lineitem`. New rows are created using the generator tool from TPC-H and adjusted to fit into our modified schema. Aerospike and MonetDB underlie some restrictions which prevent them to be part in all experiments in this section. Aerospike is not capable of batched insert or delete operations and is used only in experiments with multiple clients. Since, MonetDB uses optimistic concurrency control (OCC) for transaction management it is used only in the batched experiments because OCC prevents concurrent modifying transactions.

A side note on all batch experiments: the results in Figure 2(b) show two measured values for the batch size of one. For this particular batch size two experiments are made: an unbatched variant of a single operation and a batched variant containing a single operation. This is necessary because batching operations together into one transaction comes at a cost, which is the difference between both measured points. The higher value is mostly the unbatched variant. The only exception is HyPer because the SQL statements in these cases are either identical or semantically equivalent such that they generate the same executable code. Again, this applies to all batch experiments. The results in Figure 2(a) and 3(a)



(a) Multiple clients, no batching. (b) Single client, different batch sizes.

Abb. 2: **Insert Performance**. In total 50,000 new rows are inserted into `orders` and `lineitem`.



(a) Multiple clients, no batching. (b) Single client, different batch sizes.

Abb. 3: **Delete Performance**. In total 50,000 rows are deleted from `orders` and `lineitem`.

show a similar outcome as the echo experiment in the previous section. Aerospike and PostgreSQL scale along with the amount of clients. In contrast to Aerospike, PostgreSQL does not scale as good as in the echo experiment. The most likely reason is the additional locking mechanism to provide concurrent transactions. These costs are hidden for Aerospike because they are already accounted when checking whether a record exists and thus the echo performance is reduced. Although multiple clients are used, HyPer has a bad performance. The reasons are its view on concurrent transactions in addition to the overhead induced by query compilation. According to Section 2.2 we never introduced a vertical partitioning to the schema. In return, all transactions are serialized. A look at the CPU usage during the benchmark hardens this point as only one core is used. However, the batched experiments give us a notion of how fast compiled queries are. Among all relational databases HyPer shows the best performance improvement along increased batch sizes. From the results in 2(b), we derive two very interesting points. First, PG-hstore and PG-jsonb are faster than

PG-row. While both key-value variants have to send and store additional information about the structure they need just two integrity checks: are primary keys integers and are the values of type Hstore or JSONB. Second, the single-threaded Redis instance competes with up to six processes forked by PostgreSQL. Furthermore, for a single client it is also faster than Aerospike. This may have two reasons: a faster storage-layer due to its focus on non-nested data structures or it utilizes its single thread better than Aerospike or PostgreSQL because it does not have to lock the data against other threads. Furthermore, Redis has a downward spike in both batched experiments. This may be the result of the transaction mechanism which is used for batching the requests. Similar to transactions in relational databases, Redis stores all operations until an EXEC command is received. Upon this command, all previously stored commands are executed. Delete operations have a much smaller size and the spike apears later. Thus, this downward spike may indicate that the buffer which stores the commands is enlarged.

3.2.1 Discussion

In multi-client experiments, Aerospike shows the best overall performance. Although all tables within PostgreSQL are UNLOGGED, Aerospike is still twice as fast as PostgreSQL. This is the result of three points: (1) Key-value stores have to store much more information but in return they omit the integrity checks. (2) JDBC as database binding has a higher overhead. (3) Aerospike's underlying architecture is completely different from PostgreSQL'. The results in Section 3.1 already show that Aerospike handles requests much faster than PostgreSQL. Redis has the best performance per thread, the single instance performs up to 120,000 operations per second in multi-clients experiments. Aerospike gets in no point close to this throughput. Obviously, the absence of a locking mechanism in Redis is one reason for its good performance. It is still vague if the focus on a limited set of data structures is also beneficial for Redis. Comparing the performance of PG-hstore and PG-jsonb, only the results in Figure 2(b) indicate that this may be possible because PG-hstore is slightly faster. Both, HyPer and MonetDB are the slowest databases, but this is not simply related to being column-oriented. Although their storage files are on a RAM disk, which reduces logging latency, they still log changes. However, this does not justify the huge gap between these two and PostgreSQL. MonetDB is explicitly developed and optimized for analytical queries and "its transaction management scheme is tuned for reading large chunks of data, rather than writing small chunks of data at high speed concurrently" [Mo16b]. This is not only shown by the low throughput in this category, but also by using OCC as the transaction management scheme, which prevents parallel data modifications. With HyPer we have the problem of measuring mostly the compilation performance since we cannot use prepared statements. Only, the results for large batch sizes in Figure 2(b) give a notion of what the database is capable of. It reaches almost PostgreSQL's throughput.

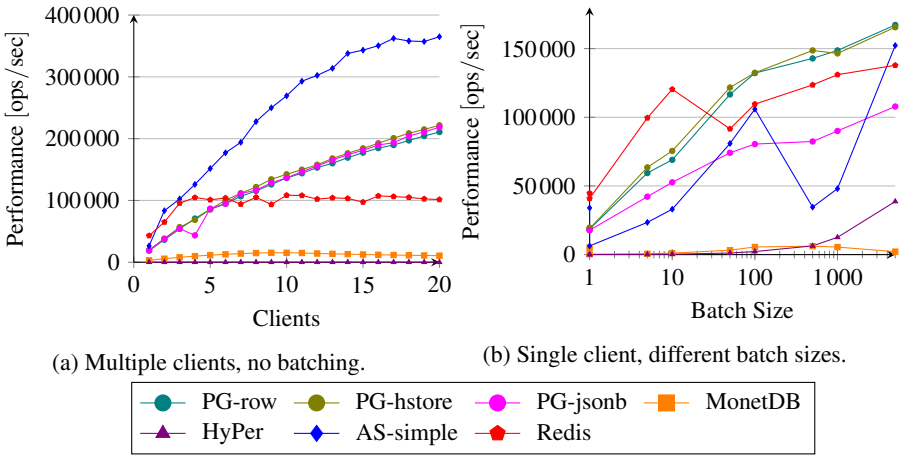
3.3 Read Experiments

In this category we investigate on the performance of various methods to read content from the database.

3.3.1 Select

With this set of experiments we conclude the basic OLTP request types. In this section, we do not focus only on `orders` or `lineitem`. Instead, all tables are used, if appropriate.

Tables with less rows than the batch size are not considered in such experiments. Furthermore, no key occurs twice within one batched request. This is mandatory because relational databases optimize this statement and return the row only once whereas key-value stores return the value twice. In settings with multiple clients Aerospike again performs the best but obviously it has some problems with batched requests as Figure 4(b) shows. Aerospike splits these across multiple threads, which concurrently work off all requested keys. Micro-benchmarks, which are provided along with Aerospike reveal that with larger batches these threads are overwhelmed. Probably because of some buffer or queue limit is hit or distribution on sub-threads becomes expensive. In consequence, the overall performance drops. Something very similar happens within MonetDB. Optimized for OLAP queries, it applies the optimization to select statements whether they are beneficial or not. The batched query is also distributed over multiple threads, which is the reason for the decreased performance with small batch sizes. MonetDB's results draw a similar graph in



(a) Multiple clients, no batching.

(b) Single client, different batch sizes.

Abb. 4: **Select Performance**. In total 50,000 rows are selected from all tables.

multi-client experiments in Figure 4(a). This may be again an effect of MonetDB's focus on OLAP requests. Threads are not only used for concurrent inter-query management but also for intra-query optimization. Another reason could be the shared CPU between MonetDB and the benchmark client. However, Aerospike and PostgreSQL show a different behavior in Figure 4(a). Thus, the reduced performance is probably not the consequence of shared resources. Due to its single thread Redis settles down around 100,000 operations per second in experiments with multiple clients. In contrast to the previous write experiments, this time the benefit of batching requests together is much smaller than in the previous write experiments. Furthermore, for larger batch sizes Redis is slower than PG-simple and the downward spike appears again earlier. There is a more general fact which holds for all key-value stores and relational databases. Whether batched or not, key-value stores need to send always additional meta information, such as the attribute names, for each requested key because the values may be totally different. Relational databases send the schema information only once and afterwards all rows. According to results for PG-simple and PG-hstore in Figure 4(a) the schema information are the heavier ones, since PG-hstore is faster than PG-simple. Within Hstore values all attributes are strings and thus avoid any domain

definitions. This is very similar to Redis. We need to be careful here with PG-jsonb because the values are returned as JSON strings and the deserialization is skipped on client-side. PG-simple and PG-hstore are parsed and their single attributes are directly accessible. The costs of sending meta information can be seen between PG-row and PG-jsonb in Figure 4(b). Although PG-jsonb skips the deserialization, it is slower than PG-simple. In this context, the similar performance of PG-hstore and PG-row may be the result of Hstore being much simpler to deserialize because all attributes are stored as strings. Similar to previous experiments, the query compilation hides most of HyPer's performance. Only for larger batch sizes we are able to overcome these costs. Furthermore, in the multiple clients experiment the requests are executed sequentially as no vertical partitioning is defined.

3.3.2 Secondary Index Look-ups

Secondary indexes are very important in search-like scenarios or for join operations which are considered in the next section. They provide fast access to records without using the primary key by avoiding to scan through the whole table and jump directly to the desired records instead. With this experiments, we obviously leave the common use cases for key-value stores and hit those of relational databases.

The larger the table the more impact an index has on the query; therefore, we employ only the three largest tables `lineitem`, `orders`, and `partsupp`. `l_orders`, `o_custkey`, and `ps_partkey` are used as indexed columns. We use not only one type of query against an indexed column but three different types, shown in Listing 2. Retrieving all records for an index query denoted as **SELECT**, counting the considered rows denoted as **COUNT(*)**, and aggregating a column which is neither the indexed column nor the primary key denoted as **MAX()**. Since, the interesting question is how the databases behave for range queries with varying selectivities we neither scale the batch size nor the amount of clients. Instead, we scale the requested range for the indexed column. The experiments are performed with eight concurrent client threads and no batching. Redis does not have secondary indexes but its developers provide an official workaround to simulate them with build-in functions. The basic idea for numeric attributes is to use sorted sets as index structure and query these for valid primary keys. The SQL statements (see Listing 2) can be used instantly for all relational databases. For Aerospike and Redis these statements need to be translated into UDF's if no build-in functions are provided.

```
// Retrieve/Count/Aggregate records
SELECT [* | COUNT(*) | MAX([ps_supplycost|o_totalprice|l_discount]) ] FROM
    [partsupp|orders|lineitem]
WHERE [ps_partkey|o_custkey|l_orderkey] = someValue
// Example how different selectivities are realized using a range query
SELECT * FROM
    [partsupp|orders|lineitem]
WHERE start <= [ps_partkey|o_custkey|l_orderkey]
  AND [ps_partkey|o_custkey|l_orderkey] <= end
```

List. 2: All three queries to measure index performance in this experiment.

According to its results in Figure 5(b) for all range sizes the performance of MonetDB is almost the same. Especially, the counting query proves that absolutely no index is used. MonetDB scans through the whole table, at least through the required columns, and the generated plan exposes nothing contradicting. The only exceptions are the benchmarks

for range size of one using equal operator instead of the range operator and SELECT for range sizes 500 and 1,000. The former is again explained by MonetDB splitting sub-selects over multiple threads. The trace statistics which were collected for the range operator show that it is treated as multiple sub-selects because the range query is distributed over multiple threads. Still, it remains unclear why the performance increases for large ranges. MonetDB's plan is the same for all range sizes. Either MonetDB decides to build an index, to build additional in-memory caches, or the intra-query works better for larger ranges.

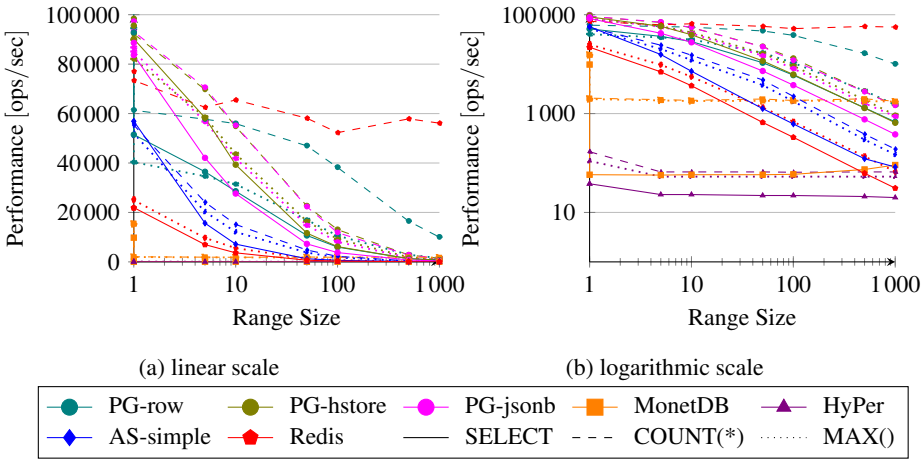


Abb. 5: **Index Performance**. In total 50,000 queries are performed using an indexed column as filter. The selectivity is represented as range of valid values.

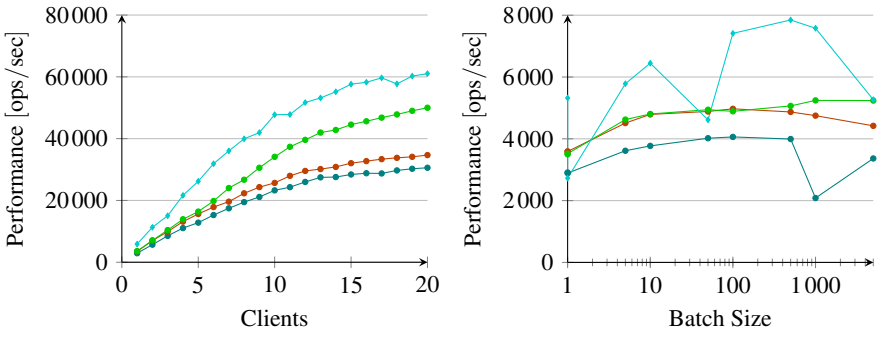
HyPer shows almost the same overall run-times for all range sizes larger than one. Both versions with a range size of one show similar run-times. Thus, we conclude that HyPer optimizes the range version and generates the same low-level code as for the equals operator. Thus, the difference in run-time for larger range sizes derives from the additional comparison. Furthermore, the varying run-times for SELECT, MAX(), and COUNT(*) are the result of faster query compilations and not of faster executions. Actually, the execution-time remains almost the same for all three statements.

The results in Figure 5(a) reveal that PostgreSQL has best support for secondary indexes. Furthermore, PG-hstore and PG-jsonb have a better performance than PG-row. Although, all indexes are B-Trees. The difference is how these indexes are used. PG-row simply scans through the index. PG-hstore and PG-jsonb first create a bitmap index based on the already existing indexes. Afterwards, the newly created index is used to filter the rows. This may be an optimization due to how these two data types are stored. The generated bitmap index could be optimized to point directly to these values. We assume that within the row just a pointer to the actual Hstore or JSONB value is stored to avoid varying row lengths. For Aerospike we are able to make two conclusions from the results in Figure 5(a): first, stream UDF's have slow start times, because SELECT is faster than MAX() and COUNT(*) for the range size of one. Although, these two return much less data. In all three cases, the index is queried at first and afterwards the returned records are handed off to the next step, either sending to the client or to the stream UDF. Second, Aerospikeseems to have performance

issues with range queries. With larger ranges, Aerospike becomes much slower for all three queries than PG-row with SELECT, which contradicts the results in Section 3.3.1. The fact that all three queries show a similar behavior strengthens this finding even more. Secondary indexes contain only a mapping between the indexed value and primary keys to records. After all valid keys are gathered, the corresponding records are retrieved in parallel similar to a batched select in Section 3.3.1. Thus, the bottleneck could be querying the index or again the distributed retrieval similar to Section 3.3.1 for batched requests. The simulated index for Redis has a moderate performance. COUNT(*) shows the best run-time in this experiments because it is a build-in operation. Thus, scanning the index is very fast but the bottleneck becomes retrieving the valid values. All values are retrieved using a single batched request but in the end each requested value uses a point query.

3.3.3 Joining Tables vs. Complex Records

Storing and retrieving denormalized values is an advantage of key-value stores. In contrast, to store such a value in relational databases it has to be normalized and stored in multiple tables. Afterwards, one would have to rebuild it by either joining multiple tables and having redundant data in the result set or use multiple queries and risk higher latency. In this experiment, we investigate how join operations perform compared to retrieve complex values. PG-row is used to join `customer`, `orders`, and `lineitem` whereas PG-complex and AS-complex are used to retrieve the complex customer representation. As already stated, join operations generate redundant data because `customer` and `orders` are joined with each corresponding `lineitem`. To get a better notion of the influence of the join operation itself compared to the overhead of sending the redundant data, we benchmark also an intermediate step.We first create a materialized view that contains all valid joins of `customer`, `orders`, and `lineitem`. Obviously, it is not possible to declare `c_custkey` as primary key for this view. Instead, an index is created upon this column. We measure the performance of retrieving complete customer information of this view, denoted as **PG-mview**, to distinguish exactly between the costs for sending redundant data and the costs for the join operation. Looking at the graphs for PG-row and PG-mview in Figure 6, we see clearly the costs induced by joining the three tables. Both variants return exactly the same data. The only difference are the join operations. Thus, the reduced performance by PG-row is the result of joining the tables. Comparing PG-mview and PG-complex the largest loss in performance seems to be caused by transferring redundant data, at least in the multiple client context in Figure 6(a). In the batched setting the redundancy can be seen only for large batch sizes in Figure 6(b). The main difference between both experiment settings is that in the batched version PG-mview sends the schema only once for multiple customer rows. In contrast, PG-complex needs to send it for each customer. Additionally, within one customer, the meta information for each order and lineitem are also redundant. In this setting, PG-mview and PG-complex send either duplicated attributes or duplicated meta information. Since PG-mview's performance drops for larger batches, we conclude that it sent in total more data than PG-complex. In experiments with multiple clients each response by PG-mview contains the schema and the customer data including the redundant attributes. Therefore, the gap to PG-complex spreads much faster in Figure 6(a). The decrease of performance is much lower than expected, because a complete customer represented in PG-complex takes up around 5 KB whereas in PG-mview 27 KB are needed. These sizes are



(a) Multiple clients, no batching. (b) Single client, different batch sizes.

Abb. 6: **Join Performance**. In total 50,000 `customer` with their `orders` and `lineitems` are selected.

the quotient of the table size which is given by PostgreSQL and the number of customers within the table. AS-complex is still faster than PG-complex but at the expense of a lower throughput. Comparing these results with their corresponding variants AS-simple and PG-jsonb in Section 3.3.1, we recognize that Aerospike's performance dropped by around 73% and PostgreSQL' dropped by 76%. In the end, for larger complex values, AS-complex is only 20% faster than PG-complex.

3.3.4 Discussion

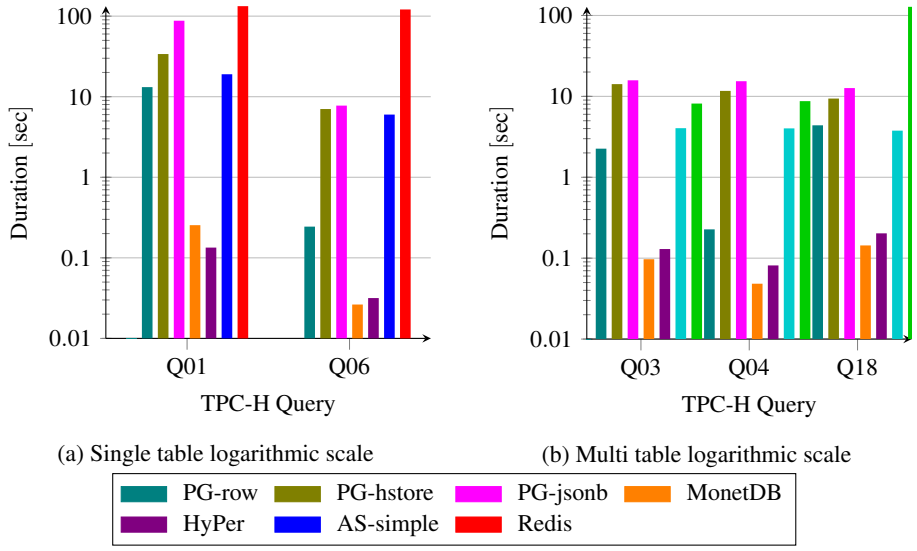
Key-value stores have proven to be the best with unbatched selects. Batching is always advisable, however, Redis and Aerospike have some issues handling large batches. This may be problems of the underlying architecture or the result of optimization towards small, high frequent requests. For simple range queries MonetDB does not seem to use any secondary index. PostgreSQL shows the best utilization of indexes. Although Aerospike supports innately secondary indexes it has some issues compared to its performance regarding simple selects. Using a simulated index with Redis may be helpful in some situations but in the end querying an index is nothing more than using batched selects. Storing and retrieving denormalized, complex values is indeed faster than using joins to rebuild them from normalized rows. However, both approaches come along with their own additional costs.

3.4 Analytic Query Experiments

In this final category we focus on OLAP and use the queries provided by TPC-H to compare the databases in this area. We distinguish between OLAP queries which use only one table and queries which use multiple. Furthermore, for these last experiments, we change the experimental process. As this type of queries is potentially long running, 50,000 requests might take awhile. Thus, instead of having three measuring runs with 50,000 operations we switch to five runs with one OLAP query per run. Furthermore, we switch from the custom

benchmark tool to the interface applications provided with the databases. The results are presented as the average run-time of all measuring runs, which means lower is better. For this experiment we utilize the queries Q01 and Q06 provided by TPC-H as these are the only ones using a single table. Both queries scan `lineitem` according to a filter and perform aggregation operations. In addition Q01 groups the result by the attributes `l_returnflag` and `l_linestatus`. The modifications for PG-hstore and PG-jsonb are straight forward because the changes are limited to switching from the columnar values to the attributes stored in the Hstore and JSONB values. For Redis and Aerospike, the declarative SQL statements are translated into procedural user-defined functions. Additionally, a list of all keys belonging to `lineitem` is generated and stored within Redis. Otherwise, it would be necessary for Redis in a first step to filter these keys. Within all other databases all values are already separated into several tables or sets.

First of all, the results in Figure 7(a) show a lower run-time for Q06 than for Q01. This is just Q06 being a simpler query: only aggregating and filtering. In contrast, Q01 also uses grouping and far more attributes. Thus, there is nothing special related to the databases. The varying run-times for all PostgreSQL variants in Q01 present again evidence that these have very different access times to a single attribute. The most expensive access accounts for JSONB values. According to the plans provided by PostgreSQL, in all cases the whole table is scanned with corresponding filters. They only differ in how they access the attributes. Unlike, in Section 3.3.2 where PostgreSQL optimizes PG-hstore and PG-jsonb this time it does not. Due to the lack of an index PostgreSQL has to scan the table in any case. Redis



(a) Single table logarithmic scale (b) Multi table logarithmic scale

Abb. 7: **Duration of Single- and Multitable TPC-H queries.**

shows the longest run-time which is again related to slow UDF's, similar to Section 3.3.2. This time it is even worse because not only a subset of keys is used but all `lineitems` are scanned. However, this time UDF's are called once during each run which proves that further access to the database is expensive. Specifically, regarding the relatively small

variation between Q01 and Q06 which results from the additional work in Q01. In contrast, Aerospike shows a much better run-time which is even comparable to PG-hstore. This outcome is contrary to the findings in Section 3.3.2 where Aerospike is much slower than PG-hstore particularly for larger ranges. Either PG-hstore is really bad with arithmetic operations or Aerospike is faster at scanning a set than querying an index. We state that the latter holds because in Section 3.3.2 MAX() is faster for PostgreSQL. The best results yield HyPer and MonetDB. Although they use different approaches, both perform almost equally. Still, compiling the query takes some time but in the return a much faster execution is achieved because of data-centric code. Especially, MonetDB proves that its optimization towards OLAP and its custom assembly language are as efficient as query compilation.

Due to its bad performance in the single table experiment we omit Redis for multi-table queries. Further changes are: AS-simple is also omitted because Aerospike does not support join operations or anything similar. Instead, we use AS-complex and are able to utilize three different tables. The list of candidates is completed by adding PG-complex. By reason of the limitation introduced by AS-complex, we have to focus on TPC-H queries which only use `customer`, `orders`, and `lineitem` such as Q03, Q04, and Q18. In Figure 7(b), we see a very similar outcome as before. Again, Aerospike shows a moderate performance compared to PostgreSQL. The most unexpected result is presented by PG-complex for Q18. After a look into the generated plan it is not so unexpected anymore as the plan contains multiple scans through the arrays within the JSONB value. Thus, it seems that at the moment PostgreSQL has some issues with optimizing queries which contain denormalized values. This may be a result due to the limited expressiveness of the SQL extensions provided by PostgreSQL. Currently, there is nothing similar to a wildcard operator which would allow to check if an array element fulfills some search criteria. Instead, arrays need to be treated as sub-tables, which is the source of the scans. For Q18, AS-complex is even faster than PG-row while for both other queries PG-row is faster. The main difference between these queries is the amount of joins used in the query plans. Q04 uses only one join, Q03 uses two joins, and Q18 requires three joins, which is reflected by the duration for all queries. PG-row needs to perform three join operations for Q18 whereas AS-complex is able to scan through all records in the `customer` set. However, both in common need to group, filter, and aggregate intermediate data. Like in Section 3.3.3 using complex values spares the need for joins. To summarize we start with MonetDB and HyPer. Both have proven to be the best with OLAP requests. Of course, this is not only the advantage of column-oriented databases which use nothing but the relevant columns. Nevertheless, both approaches realized within the databases provide further advantages. In such settings HyPer's actual execution-time starts to dominate the overall run-time. Furthermore, MonetDB has proven that its custom assembly language and optimization towards OLAP are as fast as the compiled query in HyPer. In few cases Aerospike can be faster than PostgreSQL for queries which require multiple joins. However, this performance gain comes at the expense of storing denormalized values, which slows down OLTP requests. Particularly, in the context of all PostgreSQL variants we can see how good relational databases are at optimizing queries, if they have sufficient information or an appropriate storage-layer.

4 Conclusion

This paper constitutes a performance study using various benchmarks. Test subjects were Aerospike and Redis as key-value stores and PostgreSQL, MonetDB, and HyPer as relational databases. Additionally, PostgreSQL is used to simulate the behavior of Aerospike and Redis using the data types Hstore and JSONB.

Both, HyPer and MonetDB are very efficient with OLAP queries and obtain run-times in the range of milliseconds for TPC-H queries. In return, both are not able to handle OLTP requests to a satisfying extent. For HyPer, this is a side effect of relying on query compilation. Without support for prepared statements its performance is bounded by the compilation for short requests. In contrast, the low OLTP performance by MonetDB is the result of architectural decisions. In favor of OLAP performance, the OLTP throughput is neglected. The highest OLTP throughput achieves Aerospike by utilizing all threads which are provided by the system. At the same time, each thread is used inefficiently. The reason is the locking mechanism, which is necessary to manage concurrent access to the records. Furthermore, Aerospike performed almost as good as PostgreSQL when processing OLAP requests. Due to its single-threaded design Redis has the best performance per thread. However, the downside is its inability to scale automatically along with multiple clients. Instead, the user needs to decide whether multiple Redis instances are needed or not. PostgreSQL has proven to be an all-round database throughout all tested candidates. It has the best OLAP performance behind both column-stores and the best OLTP performance behind Aerospike. Additionally, due to the Hstore and JSONB data types it can be used as key-value store. It even allows to combine both approaches. For instance, within a table mandatory or OLAP relevant attributes are defined as columns while optional attributes are stored in an additional Hstore value. With the help of PostgreSQL we show that key-value stores have a small advantage towards OLTP requests. As schema-free databases key-value stores do not make any assumptions on the content of a value. Therefore, they are able to omit integrity checks which are needed by relational databases. Thus, a key-value store itself does not provide any benefits as storage-layer in relational databases. Furthermore, we prove that joining tables is indeed slower than retrieving a complex value. Still, both approaches have their costs in terms of duplicated content or meta information. One has to be also aware of the pros and cons for different use cases.

Summing it up, key-value stores are particularly recommended in situations with high frequent OLTP and almost none OLAP. Depending on the configuration, Redis and Aerospike do not guarantee that changes are written to disk after a request finished. For workloads with almost only OLAP, specialized databases are suggested. HyPer and MonetDB are just two possible candidates for such workloads. PostgreSQL provides a good trade off for mixed workloads.

Literatur

[Ae16] Aerospike, https://www.aerospike.com/, 2016.

[AMH08] Abadi, D. J.; Madden, S. R.; Hachem, N.: Column-stores vs. Row-stores: How Different Are They Really? SIGMOD '08, ACM, New York, NY, USA, S. 967–980, 2008.

[CB74] Chamberlin, D. D.; Boyce, R. F.: SEQUEL: A Structured English Query Language. SIGFIDET '74, ACM, New York, NY, USA, S. 249–264, 1974.

[Co70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM, 13(6):377–387, Juni 1970.

[Co10] Cooper, B. F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R.: Benchmarking Cloud Serving Systems with YCSB. SoCC '10, ACM, New York, NY, USA, S. 143–154, 2010.

[Co14] TPC Benchmark H (Decision Support) Standard Specification Revision 2.17.1.

[DB16] DB-Engines Ranking of Key-value Stores, http://db-engines.com/de/ranking/, 2016.

[Fl12] Floratou, A.; Teletia, N.; DeWitt, D. J.; Patel, J. M.; Zhang, D.: Can the Elephants Handle the NoSQL Onslaught? Proc. VLDB Endow., 5(12):1712–1723, August 2012.

[Hi16] Hive, https://hive.apache.org/index.html, 2016.

[HR83] Haerder, T.; Reuter, A.: Principles of Transaction-oriented Database Recovery. ACM Comput. Surv., 15(4):287–317, Dezember 1983.

[Kl15] Klein, J.; Gorton, I.; Ernst, N. et al.: Performance Evaluation of NoSQL Databases: A Case Study. PABS '15, ACM, New York, NY, USA, S. 5–10, 2015.

[KN10] Kemper, A.; Neumann, T.: , HyPer - Hybrid OLTP&OLAP High Performance Database System, 2010.

[KN15] Kemper, A.; Neumann, T.: , HyPer, http://databasearchitects.blogspot.de/2015/04/using-hyper-with-postgresql-drivers.html, 2015.

[Mo16a] MonetDB, https://www.monetdb.org/, 2016.

[Mo16b] MonetDB SQL transaction management scheme, https://www.monetdb.org/blog/monetdb-sql-transaction-management-scheme, 2016.

[Mo16c] MongoDB, https://www.mongodb.com/, 2016.

[Po16] PostgreSQL, https://www.postgresql.org/, 2016.

[PPV13] Parker, Z.; Poe, S.; Vrbsky, S. V.: Comparing NoSQL MongoDB to an SQL DB. ACMSE '13, ACM, New York, NY, USA, S. 5:1–5:6, 2013.

[Re16] Redis, https://redis.io/, 2016.

[SF12] Sadalage, P. J.; Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 1st. Auflage, 2012.

[St05] Stonebraker, M.; Abadi, D. J.; Batkin, A. et al.: C-store: A Column-oriented DBMS. VLDB '05. VLDB Endowment, S. 553–564, 2005.