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, the picture has changed. Initially designed as inter-process hash tables handling OLTP requested, some key-value store vendors have started to tackle the area of OLAP as well. Therefore, in this performance study, we compare the relational databases PostgreSQL, MonetDB, and HyPer 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.

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, back-ends for e-commerce systems, session storage for webservers, or 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]. Some of them focused only on OLAP or only OLTP deploying specific optimizations, respectively. 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, they provide today much more functionality than just storage and retrieval of key-value pairs. Some vendors, such as Aerospike, even started to tackle the area of OLAP.

Regarding the recent development, the following question emerges: what distinguishes these systems besides the way of storing their content and what are their advantages and disadvantages? To provide an answer to this question, 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 a storage similar to Redis and Aerospike in order to get a better understanding of how key-value stores differ from classical relational systems. We make a performance study to compare the aforementioned databases and simulations in three categories: write queries, read queries,

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

and more complex analytical queries. In the first category, we investigate how fast these databases can insert and delete content. The second category focuses on two access methods to read content: besides of simple selects, we examine the efficiency of secondary indexes. Finally, the last category utilizes queries which are provided by TPC-H [Co14] and compares the databases' OLAP capabilities.

1.1 Relational Databases and Key-Value Stores

Table 1 and Table 2 show the relational systems respectively the key-value store we discuss in this work. Alongside, we present the configurations that have been applied in this evaluation. Note that the used configurations are only subsets of all possible configurations.

Configuration	**PostgreSQL**	**MonetDB**	**HyPer** (Demo) [KN10]
Layouts	Row-store, Hstore, JSONB	Column-store	Column-store
Concurrency	MVCC	Optimistic CC	Chunking & MVCC
Secondary Indexes	yes (manual)	yes (on-demand)	yes (manual)
Prepared Statements	yes	yes	no

Tab. 1: **Relational Systems** alongside with their properties.

Configuration	**Redis**	**Aerospike**
Layouts	Predefined Datastructures (Hashes, Ordered Sets)	JSON
Concurrency	Serialized	Multi-threaded
Secondary Indexes	simulated with Ordered Sets	yes (manual)
UDFs	yes (LUA)	yes (LUA)

Tab. 2: **Key-Value Stores** alongside with their used properties.

1.2 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. They find that column-stores can be simulated by row-stores to a certain degree and thus speed-up the queries. The authors of [Kl15] compare Riak, MongoDB, 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. A performance comparison between Microsoft SQL Server Express and MongoDB 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, and the Hadoop based solution Hive. 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].

1.3 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

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, we 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 the amount of requests send to the database and in consequence 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 we use prepared statements. The available HyPer demo lacks of support for prepared statements. Therefore, usual 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 also test 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.

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.

3.1 Setting the Baseline

Before we can start with the full-fledged experiments that measure end-to-end runtimes, let us begin with a simple experiment, that focuses purely on the overhead of the communication with the system. Most importantly, we want to identify wether the benchmark tool itself can be part of the overhead. For the relational systems, we simply fire a `SELECT 1` query in SQL. For Redis, we can perform an echo operation. Unfortunately, this is not possible in Aerospike, where we fire the `keyExists()` function to test for a non existing key in an empty database. Additionally, to prove whether the measured performance is bounded by the benchmark at all, the same experiment is made without communicating with any database, denoted as **NO-DB**, where simply a counter is increased while the rest remains *exactly* the same.

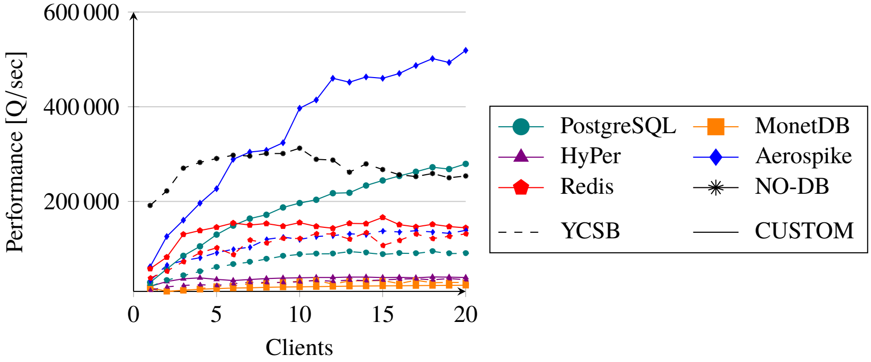


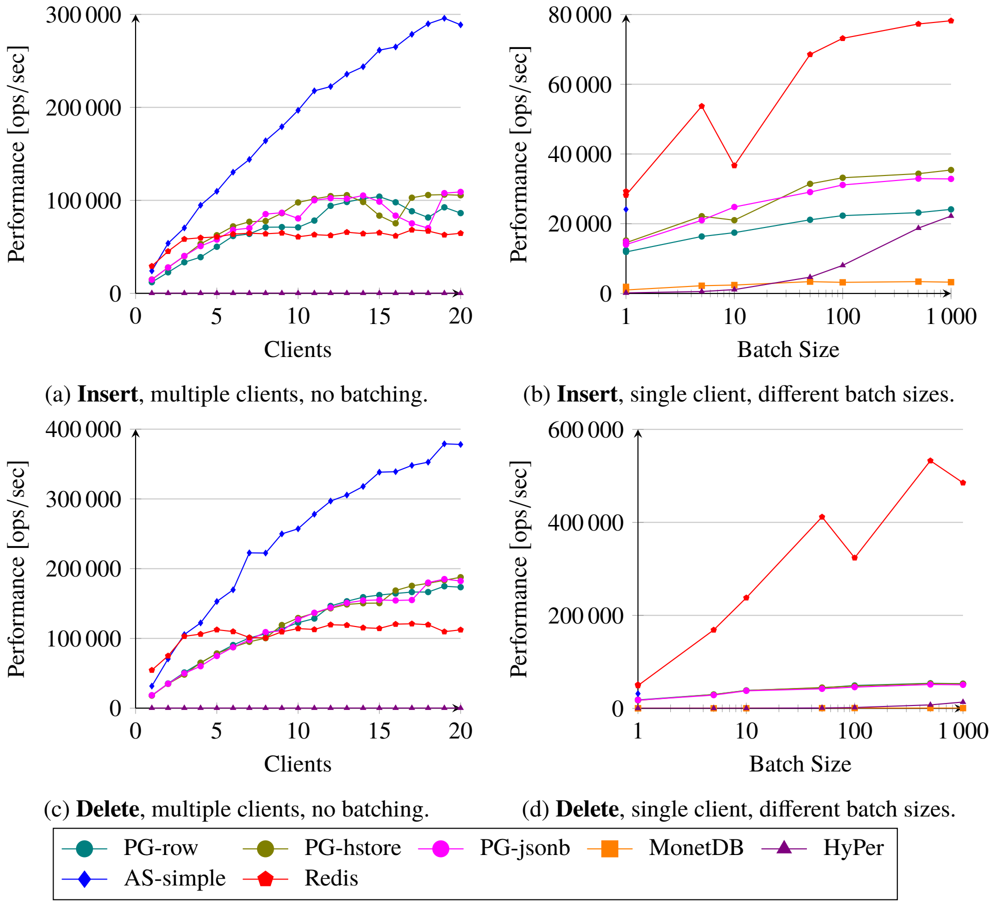
Fig. 1: **Echo Performance**. 50,000 echo requests are performed using multiple clients. **YCSB** are the results from the YCSB tool and **CUSTOM** are the results from our custom benchmark.

First, let us have a look on the results for YCSB in Figure 1. NO-DB shows the maximal throughput which YCSB is able to achieve. The results of both benchmark tools for PostgreSQL, Redis, and Aerospike show that their performance in YCSB is bounded by the benchmark itself. MonetDB and HyPer perform in both benchmarks equally. Thus, their throughput is obviously bounded by the database. As a consequence, in following experiments we are not going to take YCSB into account and consider our CUSTOM benchmark as a more meaningful alternative.

3.2 Write Experiments

In this category we start by investigating how well insertion and deletion from `orders` and `lineitem` is performed. 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: 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, with HyPer as the only exception. The results in Figure 2(a) and 2(c) 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



(a) **Insert**, multiple clients, no batching.

(b) **Insert**, single client, different batch sizes.

(c) **Delete**, multiple clients, no batching.

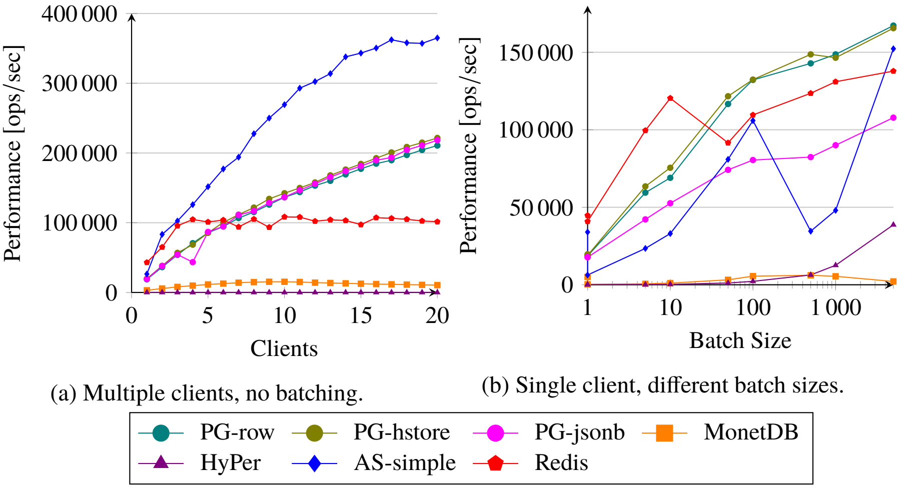
(d) **Delete**, single client, different batch sizes.

Fig. 2: **Write Performance**. In total 50,000 rows are inserted/deleted from `orders` and `lineitem`.

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. As we never introduced a vertical partitioning to the schema, 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.

3.3 Read Experiments

To evaluate the read performance, let us now first look at simple selects on the primary key column. With this set of experiments we conclude the basic OLTP request types. In this section, we use all tables of our schema, that contain at least as many rows as the batch size. Furthermore, no key occurs twice within one batched request. Let us first look at Figure 3(a), where we vary the number of clients and avoid batching. We can see that Aerospike scales the best with the number of clients as one would expect from a multi-threaded key-value store under a concurrent OLTP workload. In contrast to that, Redis serializes the requests from all clients and thus saturates quickly. PostgreSQL scales less effectively than Aerospike but still improves till 20 clients. On the other end of the performance lie MonetDB and HyPer, which suffer from their focus on OLAP and expensive query compilations. In Figure 3(b), we vary the batch size while limiting the evaluation on a single client. Interestingly, all systems monotonically gain performance with an increase of the batch size except of Aerospike and Redis. Especially Aerospike suffer under the batch sizes 500 and 1000, probably due to a suboptimal distribution to the executing threads. Furthermore, we can also observe that PG-jsonb is slower than PG-row and PG-hstore with larger batches: PG-jsonb has to send significantly more meta-data.



(a) Multiple clients, no batching.

(b) Single client, different batch sizes.

Fig. 3: **Select Performance**. In total 50,000 rows are selected from all tables.

Let us now focus on secondary indexes. These are very important to answer point-queries or in join operations. Thus, we will now inspect how the systems behave when using their respective secondary index structures. 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 three different query types, shown in Listing 1, that either retrieve all selected rows (**SELECT**), count all selected rows (**COUNT**), or aggregate the maximum over the selected rows of an unindexed attribute (**MAX**). In the experiments of Figure 4, we vary the size of the selected range, deactivate batching, and use eight concurrent client threads. Redis does not have secondary indexes but its developers provide an official workaround to simulate them with build-in functions. The SQL statements (see Listing 1) can be used instantly for all relational databases whereas for Aerospike and Redis these statements need to be translated. For instance, to count and to aggregate the elements, Aerospike needs to apply a stream UDF.

```
// 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. 1: All three queries to measure index performance in this experiment.

As we can see in Figure 4, for all range sizes the performance of MonetDB is almost stable, as no index is used by the system to answer the queries. Only at a range size of one, MonetDB performs better as it can test for equality. A similarly stable performance can be observed for HyPer. We can also see that PostgreSQL obviously has the best support for secondary indexes, with PG-hstore and PG-jsonb having a better performance than PG-row. This is caused by PostgreSQL building an additional bitmap index based on the already existing indexes. For Aerospike we are able to make two conclusions: 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. Second, Aerospike seems 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 of the simple selects. The simulated index for Redis shows a moderate performance.

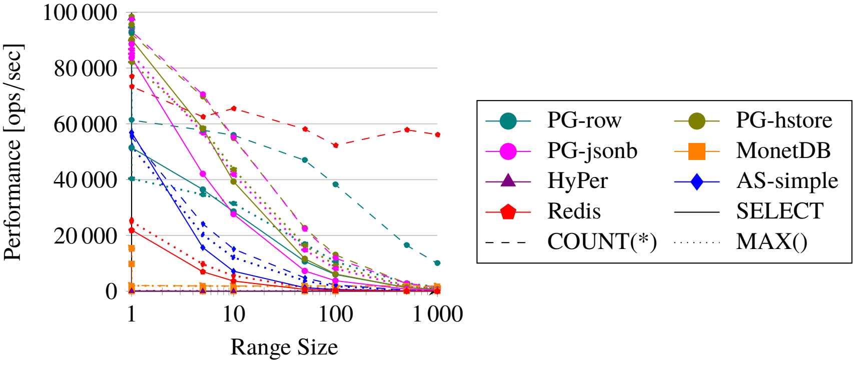


Fig. 4: **Index Performance**. In total 50,000 queries are performed using an indexed column as filter. The selectivity is represented as range of valid values.

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. Due to the lack of a join operation in Aerospike, we focus on the OLAP queries which use only one table (Q01 and Q06). In the experiments, we perform five runs with one OLAP query per run and take the average. Furthermore, we switch from the custom benchmark tool to the interface applications provided with the databases to simplify the execution, as no batching is needed anymore. For PG-hstore and PG-jsonb we map 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 UDFs.

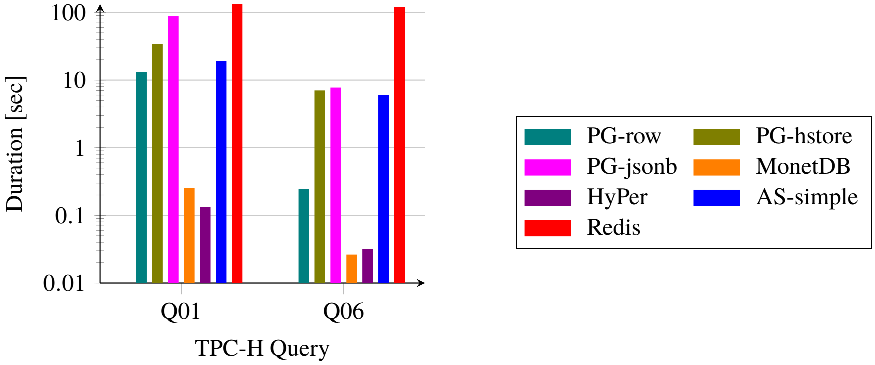


Fig. 5: **Duration of Singletable TPC-H queries.**

Let us inspect the results in Figure 5. 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. In all cases, the whole table is scanned with corresponding filters. Unlike in Section 3.3, this time PostgreSQL decides to perform a scan on the table in any case. Redis shows the longest run-time due to slow UDF's, similar to Section 3.3. 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 where Aerospike is much slower than PG-hstore particularly for larger ranges. Thus, Aerospike is faster at scanning a set than querying an index. 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.

4 Conclusion

This paper constitutes a performance study using various benchmarks[4]. 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

[4] Due to the page limitations, we reduced our evaluation to the presented content. The interested reader can additionally find the evaluation of deletes, joins, and multi-table TPC-H queries on our website.

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, the threads are 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. 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 and can omit integrity checks.

Summing it up, key-value stores are particularly recommended in situations with high frequent OLTP and are not yet ready to perform well under OLAP workloads. 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.

References

[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, pp. 967–980, 2008.

[Co70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM, 13(6):377–387, June 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, pp. 143–154, 2010.

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

[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.

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

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

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

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