Utilizing a NoSQL Data Store for Scalable Log Analysis

Khalid Mahmood
Dept. of Information Technology
Uppsala University,
Sweden
khalid.mahmood@it.uu.se

Tore Risch
Dept. of Information Technology
Uppsala University,
Sweden
tore.risch@it.uu.se

Minpeng Zhu
Dept. of Information Technology
Uppsala University,
Sweden
minpeng.zhu@it.uu.se

ABSTRACT
A potential problem for persisting large volume of data logs with a conventional relational database is that loading massive logs produced at high rates is not fast enough due to the strong consistency model and high cost of indexing. As a possible alternative, a modern NoSQL data store, which sacrifices transactional consistency to achieve higher performance and scalability, can be utilized. In this paper, we investigate to what degree a state-of-the-art NoSQL database can achieve high performance persisting and fundamental analyses of large-scale data logs from real world applications. For the evaluation, a state-of-the-art NoSQL database, MongoDB, is compared with a relational DBMS from a major commercial vendor and with a popular open source relational DBMS. MongoDB is chosen as it provides both primary and secondary indexing compared to other popular NoSQL systems. These indexing techniques are essential for scalable processing of queries over large scale data logs. To explore the impact of parallelism on query execution, sharding was investigated for MongoDB. Our results revealed that relaxing the consistency did not provide substantial performance enhancement in persisting large-scale data logs for any of the systems. However, for high-performance loading and analysis of data logs, MongoDB is shown to be a viable alternative compared to relational databases for queries where the choice of an optimal execution plan is not critical.

Categories and Subject Descriptors
H.2.m [**Database Management**]: Miscellaneous

General Terms
Measurement, Performance, Experimentation

Keywords
NoSQL data stores, large-scale log analysis, log archival, bulk loading, sharding

IDEAS '15, July 13 - 15, 2015, Yokohama, Japan
 ISBN 978-1-4503-3414-3/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2790755.2790772

1. INTRODUCTION
Relational databases can be used for large-scale analysis of data logs from industrial applications such as sensor readings [21] [25] [29] and stream logs [27] [28]. Persisting large volume of data logs produced at high rate requires high performance bulk loading of data into a database before analysis. However, the loading time for relational databases may be time consuming due to full transactional consistency [9] and high cost of indexing [23]. In contrast to relational DBMSs, NoSQL databases are designed to perform simple tasks with high scalability [5]. For providing high performance updates, NoSQL databases generally sacrifice strong consistency by providing so called eventual consistency compared with the ACID transactions of regular DBMSs. NoSQL databases can be utilized for typical historical analysis of log data or numerical log analytics where transactional consistency conforming ACID compliance is not required.

It has been argued in [23] that relational DBMSs can achieve the same performance as NoSQL database systems by specifying relaxed consistency to eliminate overhead. In [12] it is shown that this overhead is almost equally divided between four components for a typical relational DBMS: logging, locking, latching, and buffer management. However, we did not find any experimental benchmark that investigates how a weaker consistency model for relational DBMSs and NoSQL databases can be utilized to enhance performance for persisting and analysis of data logs. Although [10] compares the performance of SQL Server and MongoDB [15] for interactive data-as-a-service based on the YCSB benchmark [6], it does not investigate the performance of the systems for scalable log analysis. A more recent investigation [13] did not consider the state-of-the-art NoSQL database, MongoDB for performance evaluation. None of the papers consider the option for relaxing the consistency for both types of systems.

Unlike NoSQL data stores, relational databases provide advanced query languages and optimization technique for scalable analytics. Paper [19] demonstrates that indexing is a major factor for providing scalable performance, making relational databases having a performance advantage compared to a NoSQL data store without proper indexing to speed up analytical tasks. Like relational databases, MongoDB provides a query language as well as primary and secondary indexing, which should be well suited for analyzing persisted logs. Unlike relational databases and MongoDB, most other popular NoSQL data stores [22], Cassandra [2], Redis [20], HBase [1], Memcached [7], and CouchDB [3], do not provide full secondary indexing and query processing to transparently utilize indexes, which is essential for scalable performance of inequality queries. CouchDB has secondary indexes, but queries have to be written as map-reduce views [5], not transparently utilizing indexes.

In this paper we compare MongoDB with state-of-the-art relational DBMSs to investigate at what degree a state-of-the-art NoSQL database is suitable for persisting and analyzing large scale data logs compared with relational databases. The performance of MongoDB is compared with a commercial DBMS from a major relational vendor, called DB-C, and a popular open source relational DBMS, called DB-O.

The performance evaluation covers the bulk loading capacities of the systems w.r.t. indexing and relaxed consistency. We define three fundamental queries for accessing and analyzing persisted logs to investigate the efficiency of query processing and index utilization of the DBMSs. The properties of these queries are key selection, range search, and aggregation, which are fundamental to the analysis of persisted logs [25] [29]. We utilize data logs from a real world application [21] consisting of more than 1 billion sensor readings from industrial equipment.

Furthermore, the impact of MongoDB's auto-sharding (automatic partitioning) is investigated for persisted log analysis in order to explore whether its data partitioning can provide performance advantages for bulk loading and query execution.

In summary, the main contribution of the paper is a performance evaluation of persisting and analyzing data logs under different consistency configurations. The paper provides a comparison of the suitability of the two kinds of database systems for large-scale log analysis and reveals the trade-offs between bulk-loading and different levels of consistency. We discuss the cause of the performance differences influenced by how the systems choose different indexing strategies under relaxed consistency. The investigations provide insights in the issues that future systems should consider when utilizing weaker consistency of back-end storage for persisting and analyzing of data logs.

2. PERFORMANCE EVALUATIONS

In this section, we present bulk loading strategies and fundamental queries for persisted data logs. We first measure the performance in terms of execution time for different loading strategies by relaxing consistency overhead. Then we compare the performance of fundamental queries. For MongoDB, we also investigate the impact of auto-sharding on loading and querying. Based on inspecting the query execution plans, we discuss the causes of the performance differences.

2.1 Application Scenario

The Smart Vortex EU project [21] serves as a real world application context, which involves analyzing data logs from industrial equipment. In the scenario, a factory operates some machines and each machine has several sensors that measure various physical properties like pressure, power consumption, temperature, etc. For each machine, the sensors generate logs of measurements, where each log record has timestamp *ts*, machine identifier *m*, sensor identifier *s*, and a measured value *mv*. Each measured value *mv* on machine *m* is associated with a valid time interval *[bt, et)* indicating the begin time and end time for *mv*, computed from the log time stamp *ts*. The table measures *(m, s, bt, et, mv)* will contain a large volume of log data from many sensors on different machines. There is a composite key on *(m, s, bt)*.

In the performance measurements, the logs are bulk loaded into MongoDB and the two relational DBMSs. Since the incoming sensor streams can be very voluminous, it is important that the measurements are bulk-loaded fast. After data logs have been loaded into the *measures* table, the user can perform queries to detect anomalies of sensor readings by analyzing values of *mv*. The queries are used in the performance evaluation in order to understand the performance differences for both kinds of systems.

2.2 Data Set

The evaluation is made based on measurements from a real-world application in the Smart Vortex project [21]. A typical time series formed by a small piece of a larger numerical log from the application is plotted in Figure 1. In the performance evaluations more than 1 billion log measurements, which occupies 60GB of raw data from industrial sensors is used. It is important that data loading can keep up with increasing log volume. To investigate DBMS performance with growing data volume, increasing sections of the data logs were loaded into the databases.

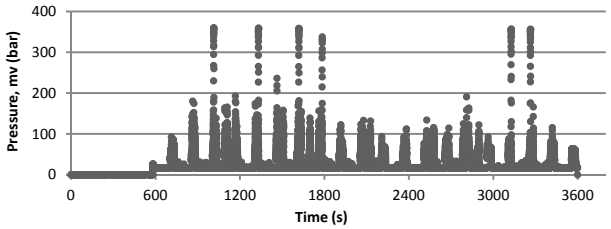

Figure 1. Pressure measurements of sensor for 1 hour

2.3 Consistency Configurations for Bulk Loading

In relational DBMSs, typical transactional overhead such as logging can be turned off and isolation level be relaxed to boost the performance. To investigate the impact of relaxed consistency levels, we configure the systems as in Table 1, which also defines the acronyms for the experiments. The lowest isolation level for the relational databases (dirty reads) corresponds to unacknowledged write concern in MongoDB, while serializable transactions correspond to *acknowledged write concern* [16]. For MongoDB, we also investigate the performance impact of auto-sharding, which can be combined with both consistency levels per shard. MongoDB does not have distributed transactions, therefore, synchronized updates of several shards is not supported.

Table 1. Consistency configurations for the experiments

Acronym	Name and Consistency Level	Properties
DB-C-S	DB-C & strong consistency	Logging, serializable isolation level
DB-C	DB-C, & weak consistency	Dirty reads, no logging
DB-O-S	DB-O & strong consistency	Logging, serializable isolation level
DB-O	DB-O & weak consistency	Dirty reads , no logging
Mongo-S	MongoDB & acknowledged write concern	Logging, serializable isolation level
Mongo	MongoDB & unacknowledged write concern	Dirty reads, no logging
Mongo-AS-S	MongoDB auto-sharding & acknowledged write concern	Logging, no distributed transactions, serializable isolation
Mongo-AS	MongoDB auto-sharding & unacknowledged write concern	No logging, no distributed transactions, dirty reads

We evaluated several alternatives of bulk loading by utilizing the different levels of consistency configurations. First, the impact of relaxed consistency is investigated and then the best consistency option for each system is used in all other experiments.

2.4 Fundamental Queries

The queries used in the performance evaluation are very fundamental to analytics over numerical logs and provide basic building blocks of analytics of persisted logs [21] [25] [29]. We made the experimental queries simplistic in nature, which is one of the four major criteria of domain specific benchmarks [11], to demonstrate the credibility of the performance trade-offs for the systems. The queries essentially explore the performance and scalability of primary and secondary index utilization for growing data logs. The first query, *key look-up query (Q1)*, gets a sensor reading for a given timestamp. The second query, *range query (Q2)* detects deviations of sensor readings from expected values. The third query, *aggregation query (Q3)*, performs aggregation of measurement deviations from persisted logs.

2.4.1 Key Lookup Query, Q1

The task involves finding measured values mv for a given machine *m*, sensor *s*, and begin time, *bt*. The query expressed in SQL and MongoDB's query language [16], respectively, is specified as follows:

```
-- SQL
SELECT m, s, mv FROM measures
 WHERE m =? AND s =? AND bt =?

//MongoDB
db.measures.find(
 { m:?, s:?, bt:? },{ m: 1, s: 1 mv: 1})
```

In order to provide scalable performance of the query, we need an index on the composite key. In all systems we index by defining a composite B-tree primary key index on *(m, s, bt)*. This query demonstrates the performance of primary key index utilization of the three systems.

2.4.2 Range Query, Q2

This query involves finding unexpected sensor readings by observing measured values *mv* that deviate from an expected value. Here, the sensor readings with the measured value *mv* higher than the unexpected value are retrieved. Such a query can be expressed in SQL and MongoDB as follows:

```
-- SQL
SELECT * FROM measures WHERE mv > ?

//MongoDB
db.measures.find({ mv: {$gt: ?}})
```

In order to improve the performance of this query, we need a secondary ordered index on value, *mv*. Query Q2 shows the performance of secondary B-tree indexing and how well the query optimizer can utilize the index. Since the efficiency of a secondary index is highly dependent on the selectivity, the query was executed with different query selectivities by providing the appropriate ranges of *mv*. The correspondence between choice of *mv* and query selectivities for Q2 for the data set is plotted in Figure 2.

Based on the data illustrated by Figure 2, we execute Q2 for value of *mv* resulting in the selectivities 0.002%, 0.02%, 0.2%, 0.25 %, and 1%, resulting around 0.02, .2, 2, 2.5,5 and 10 millions of log records.

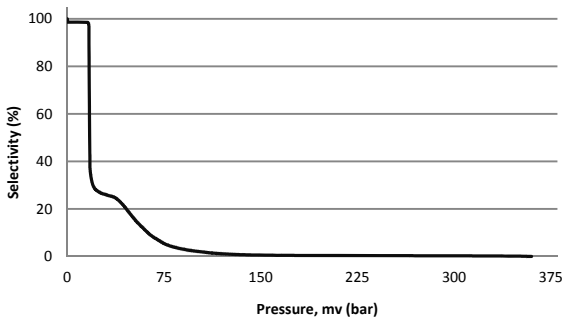

Figure 2. Measured value to selectivity mapping

Query Q2 is an example of a very fundamental analytics query that involves inequality comparisons. Complex analytics queries usually involve such inequalities and can often be rewritten into inequality queries like Q2, as automated in [25].

2.4.3 Aggregate Query, Q3

This query counts the total number of sensor readings having a measurement anomaly, using the same inequality as in Q2. Such a query is expressed in SQL and MongoDB as follows:

```
-- SQL
SELECT COUNT(*) FROM measures WHERE mv > ?

//MongoDB
db.measures.count( { mv: {$gt: ?}})
```

Similar to Q2, this query was executed for different selectivities utilizing a secondary index on *mv*. In difference to Q2, which returns a large volume of abnormal sensor readings, Q3 returns a single aggregated value. The query has insignificant network communication overhead compared to Q2.

2.5 Indexing Strategy

To speed up lookups of sensor readings for a given timestamp, we define a composite index on machine id *m*, sensor id *s* and begin time *bt*. For query Q2 and Q3, we define an extra secondary index on *mv* in addition to the composite key index. The data is then bulk loaded and the three fundamental queries were executed.

2.6 Benchmark Configuration

The non-sharding experiments are performed on a computer running Intel® CoreTM i7, 3.0GHz CPU with Windows Server 2013 operating system. The computer has 16GB of physical memory.

2.6.1 MongoDB Configuration

MongoDB version 2.4.8 is used for the performance evaluation. Relational tables are represented as collections of binary-JSON (BSON) objects [14] in MongoDB. Since the attribute names are stored inside each BSON object, short attribute names are used to make the database compact.

The sharding experiments were run on a cluster of five nodes connected by a one Gbit Ethernet switch. Each node had the same hardware configuration as the non-sharding experimental setup. We ran one *mongod* process managing each shard, one *config_db* process managing meta-data, and one *mongos* process for the MongoDB coordinator. The client, *mongos*, and *config_db* were run on the same node separate from the shards *mongod*.

2.6.2 Relational DBMS Configurations

The query result cache was turned off for both relational DBMSs. Also MongoDB does not utilize any query result cache when executing queries.

2.6.3 Benchmark Execution

For each system we measured the bulk-load time for 167, 333, 667, and 1000 million sensor readings consisting of approximately 10GB, 20GB, 40GB, and 60GB of data, respectively. The raw data files were stored in CSV format where each individual row represents a sensor reading for machine *m*, sensor *s*, begin time *bt*, end time *et*, and the measured value *mv*.

The bulk loading into the relational DBMSs and MongoDB were performed utilizing their batch commands for bulk loading CSV files.

The scalabilities of all fundamental queries were evaluated with the largest data set of 1 billion sensor records (60 GB) for all the systems.

To enable incremental bulk loading of new data into existing tables, the indexes should always be predefined in all experiments, rather than building them after the bulk loading. Although one might consider the option of bulk loading first and then building the index, this will contradict the notion in our application scenario, where bulk loading and analyzing streaming logs is a continuous process that demands incremental loading of the data into pre-existing tables. Nevertheless, we also made performance evaluations of building the indexes after bulk loading, which is faster compared to incremental bulk loading (Figure 4).

To provide stable results for bulk loading, we made all the experiment starting with empty databases. For each query, we measured the average time of three executions. The standard deviations of the measurements were less than 1%.

2.7 Experimental Results

For investigating the performance of bulk loading and queries for different consistency configurations, the following experiments were conducted.

2.7.1 Performance of Bulk Loading

In Figure 3, we observe that all systems offer scalable loading performance, except DB-O (*DB-O* and *DB-O-S*) and sharded MongoDB (*Mongo-AS* and *Mongo-AS-S)*. DB-O scales significantly worse than DB-C and MongoDB for bulk-loading, whereas Mongo-AS is faster than DB-O. Both Mongo-AS and Mongo-AS-S are much slower compare to DB-C and non-sharded MongoDB. We speculate that this performance degradation is due to MongoDB's internal re-balancing of data among the shards during bulk-loading.

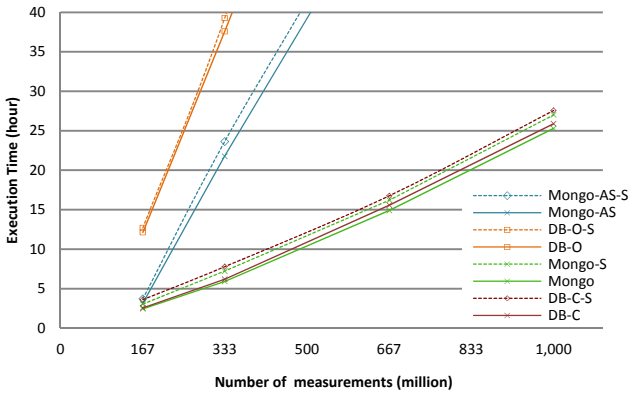

Figure 3. The performance of bulk loading with different consistency configurations

For bulk loading of large databases, the improvement of weak consistency is around 24.8% for DB-C (*DB-C* vs. *DB-C-S*), while it is around 26% for MongoDB (*Mongo* vs. *Mongo-S*). MongoDB with weak consistency performs best compared to other systems. In summary, relaxing transactional overhead did not provide substantial performance improvement for any system. From now on we always use the faster weak consistency levels in the experiments.

Figure 4 illustrates the performance degradation for bulk loading one billion records incrementally with predefined indexes (*Indexed-Before*) compared with building the indexes after all data are loaded (*Indexed-After)*. For *Indexed-After,* we show the total time of bulk loading and building the indexes. Here for the *Indexed-After* experiment, DB-C performs best. Although all systems demonstrate better performance with *Indexed-After*, this option prevents incremental bulk loading and is not suitable for incrementally persisting logs.

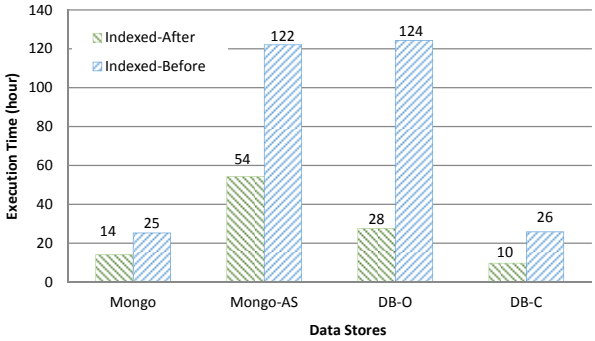

Figure 4. The performance of building indexes before and after bulk loading

2.7.2 Performance of Key Lookup Query (Q1)

Figure 5 shows the performance of key lookup query Q1 to retrieve a particular sensor record. As expected, indexing the key provides scalability of Q1 in all systems, with DB-C being fastest.

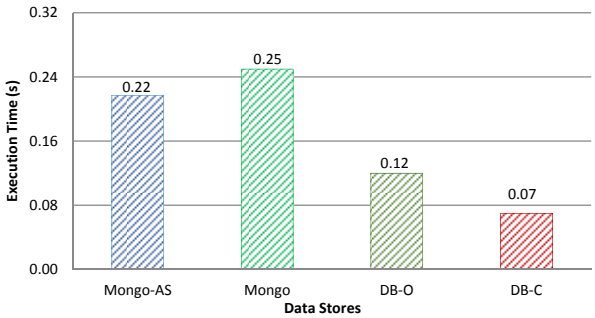

Figure 5. Performance of key lookup query (Q1)

2.7.3 Performance of Range Query (Q2)

Figure 6 shows the performance of query Q2 with both primary and secondary indexes defined. The selectivities are varied from 0.002% up to 1.0% for 1 billion sensor records. Clearly there is a problem with secondary indexes for inequality queries in DB-O. Both sharded and non-sharded MongoDB and DB-C scale substantially better and is therefore investigated further in Figure 7.

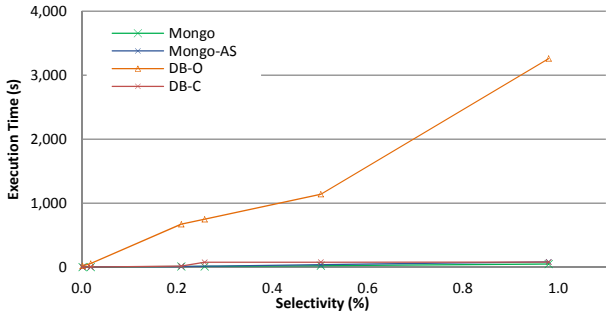

Figure 6. The performance of range query (Q2)

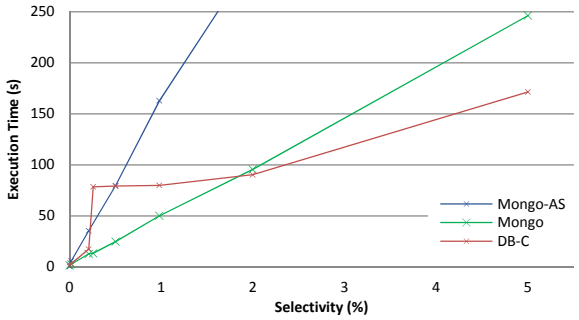

Figure 7. The performance of Q2 for DB-C and MongoDB

Figure 7 compares Q2 for DB-C and both sharded and non-sharded MongoDB while varying the selectivities from 0.002% up to 5.0%. The figure shows that DB-C switches from scanning the secondary index to full table scan when around 0.25% of the rows are selected. This makes DB-C faster than all configurations of MongoDB for non-selective queries (selecting more than 2.0%), because MongoDB does not switch the execution strategy and continues with an index scan for growing selectivities. Mongo-AS is clearly slowest for non-selective queries, while the query optimizer of DB-C makes it the system with the most stable performance. In the Figure, no performance differences can be observed for selectivities less than 0.2% and therefore Table 2 details the performance differences for selectivities up to 0.2%.

Table 2. The performance of very selective Q2 for DB-C and MongoDB

Selectivity	Records	Performance of Q2 (second)		
%	*n*	Mongo-AS	Mongo	DB-C
0.002	237,360	0.56	1.6	1.8
0.02	2,256,240	5.89	2.7	3.3
0.20	22,261,280	35.5	12.7	17.4

Table 2 shows that for very selective queries (selectivity from 0.002% to 0.2%) Mongo-AS is the fastest, since a relatively small number of records have to be transferred, which results in less network communication overhead. Sharded MongoDB is fastest only for highly selective queries.

2.7.4 Performance of Aggregate Query (Q3)

Figure 8 shows the performance of the aggregate query Q3 where a single value is returned. We use the same selectivities of the condition inside the aggregate as for Q2. Here it turns out that Mongo-AS performs much better compared to the corresponding performance of Q2 in Figure 7, since each shard performs a parallel scan and then sends a single result object to the coordinator.

The performance of DB-O for Q3 is much better than for Q2 in Figure 6, but it still scales worse than the other systems. For aggregates over non-selective conditions (5%), Mongo-AS scales best, being 1.4 times faster than non-sharded MongoDB and DB-C, respectively. However, five shards provides only 40% speed-up in our settings.

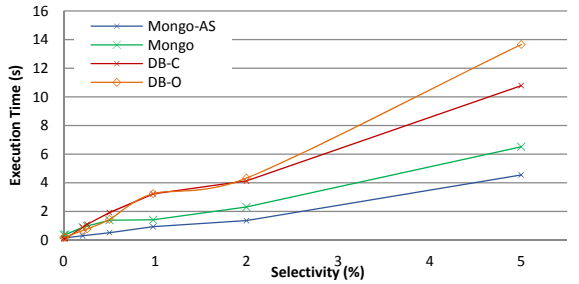

Figure 8. The performance of the aggregate query (Q3)

Figure 9 further highlights the performance of Q3 with highly selective conditions, where DB-C is fastest for selective queries, while the performance of Mongo-AS is slightly slower due to overhead of coordination among shards.

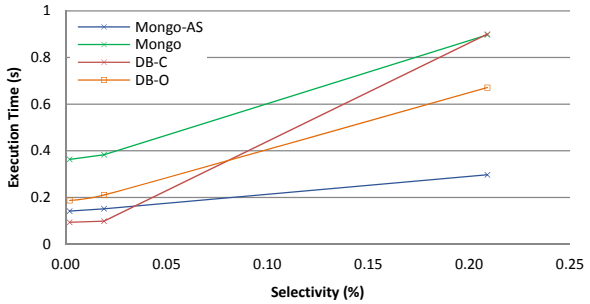

Figure 9. The performance of Q3 for smaller selectivities

The overall results of the performance evaluation are summarized in Table 3, where MongoDB is shown to have comparable performance as the state-of-the-art relational database from a major commercial vendor (DB-C).

Table 3. Qualitative summary of the experimental results

Task\ System		DB-O	DB-C	Mongo	Mongo-AS
Bulk Loading (Figure 3)		very bad	good	very good	bad
Key lookup, Q1 (Figure 5)		good	very good	good	good
Range query, Q2	Selective (Table. 2, Fig. 7)	very bad	good	good	good
	Non-selective (Figure 6, 7)	very bad	very good	good	bad
Aggregate query, Q3	Selective (Figure 9)	good	good	good	good
	Non-selective (Figure 8)	bad	bad	good	very good

3. RELATED WORK

Typical TPC benchmarks [24] such as TPC-C, TPC-DS, and TPC-H are targeted towards either OLTP or decision support, not for large scale log analysis, which often requires scalable persisting and querying over persisted logs, the focus of this paper.

Floratou et al. [10] compared the performance of SQL Server and MongoDB for interactive data-as-a-service queries based on the YCSB benchmark [6], showing that SQL Server has significant performance advantages over MongoDB. However, the work neither explored the options of relaxing consistency overheads nor investigated indexing and query optimization issues for scalable execution of persisted data logs. Dede et al. [8] evaluated the performance of MongoDB and Hadoop for scientific data analysis, but not for scalable log analysis and there was no comparison with relational DBMSs.

Barahmand et al. [4] compared the performance of an SQL solution with MongoDB for interactive social networking actions and sessions, which does not fit into the context of persisting and analyzing logs.

Wei et al. [26] utilized MongoDB for storing and analyzing network logs. Although they provide queries that analyze network logs, they did not compare the performance with other systems.

Finally, the performance of online incremental bulk loading with a main-memory DBMS was investigated in [17] [18]. By contrast, our focus is on comparing disk-based NoSQL and relational databases for persisting large-scale data logs.

To our best knowledge we did not find any performance evaluation that compares MongoDB with relational DBMS in the context of persisting and analyzing of numerical logs.

4. CONCLUSIONS AND DISCUSSIONS

The conclusions from the evaluation can be divided into three different factors influencing performance: (i) relaxing consistency, (ii) indexing and query processing, and (iii) sharding.

First, we discovered that relaxing the consistency does not provide any substantial performance enhancement in querying large scale data logs for neither SQL nor NoSQL databases. Although it is shown in [12] that removing transactional overhead can improve performance up to 20 times for updates, we discovered that both commercial and open source relational databases provide less than 25% performance improvement for bulk-loading with relaxed transaction consistency. In contrast to the aggressive modification of the database kernel in [12], a common user will not be able to modify the DBMS kernel but has to rely on the options provided by the system. For MongoDB, weak consistency configuration of bulk loading provides around 26% improvement.

For bulk loading in general, both MongoDB and DB-C scale substantially better than DB-O. For the largest data size, bulk loading with non-distributed MongoDB and DB-C are more than five times faster than DB-O. Distributing MongoDB by auto-sharding is about 4 times slower than non-distributed MongoDB and DB-C.

All systems perform well for looking up records matching the key (query Q1) by utilizing a primary key index. For the analytical task of range comparisons between a non-key attribute and a constant (query Q2), both MongoDB and DB-C scale substantially better than DB-O. A more careful comparison of DB-C and MongoDB revealed that DB-C scales better for non-selective queries, while MongoDB is faster for selective ones. The reason is that, unlike MongoDB and DB-O, DB-C switches from a non-clustered index scan to a full table scan when the selectivity is sufficiently low, while MongoDB (and DB-O) continues to use an index scan even for non-selective queries.

The aggregation query (query Q3) scales for all systems by utilizing the secondary index when computing an aggregated value. Here, sharded MongoDB scales best being around 1.4, 2.4, and 9.5 times faster than non-sharded MongoDB, DB-C, and DB-O, respectively. The reason is that a parallel scan without sending lots of results among distributed shards speeds up query execution. Therefore, we conclude that, only when an analytics task is inherently parallel with insignificant communication/data-transfer among parallel nodes, distributed MongoDB (or similar NoSQL data store) is an alternative to vertical scaling to speed up the analytics.

To conclude, non-sharded MongoDB performs significantly better compared to DB-O and has comparable performance with DB-C, making it suitable for large scale persisting and analyzing logs. However, DB-C demonstrates that relational databases can have performance advantages compared to both distributed and non-distributed NoSQL databases by having a sophisticated query optimizer. NoSQL databases can also be equipped with more sophisticated query optimizers as in state-of-the-art relational DBMSs, which will improve query performance. However, some NoSQL databases such as MongoDB provide a flexible schema-less paradigm which makes query optimization challenging due to the absence of rigid schema and proper data statistics.

Finally, although we have discussed the issues of persisting and analysis of data logs, our results can be utilized also in other large-scale and data intensive application scenarios where bulk loading with relaxed consistency and scalable query execution are required.

For high performance loading and analysis of large-scale data logs, MongoDB is shown to be a viable alternative compared to relational databases.

5. ACKNOWLEDGMENTS

This work is supported by EU FP7 project Smart Vortex, the Swedish Foundation for Strategic Research under contract RIT08-0041, and eSSENCE.

6. REFERENCES

[1] Apache Software Foundation. 2015. Apache HBase. (June 2015). Retrieved June 18, 2015 from http://hbase.apache.org/

[2] Apache Software Foundation. 2015. Cassandra. (June 2015). Retrieved June 18, 2015 from http://cassandra.apache.org/

[3] Apache Software Foundation. 2015. CouchDB. (June 2015). Retrieved June 18, 2015 from http://couchdb.apache.org/

[4] Barahmand, S., Ghandeharizadeh, S. and Yap, J. 2013. A comparison of two physical data designs for interactive social networking actions. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management* (CIKM '13). ACM, New York, NY, USA, 949-958. DOI=http://doi.acm.org/10.1145/2505515.2505761

[5] Cattell, R. 2011. Scalable SQL and NoSQL data stores. *ACM SIGMOD Rec.* 39, 4 (May 2011), 12-27. DOI=http://doi.acm.org/10.1145/1978915.1978919

[6] 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). ACM, New York, NY, USA, 143-154. DOI=http://doi.acm.org/10.1145/1807128.1807152

[7] Danga Interactive. 2015. Memcached. (June 2015). Retrieved June 18, 2015 from http://www.memcached.org/

[8] Dede, E., Govindaraju, M., Gunter, D., Canon, R.S. and Ramakrishnan, L. 2013. Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM workshop on Scientific cloud computing* (Science Cloud '13). ACM, New York, NY, USA, 13-20. DOI=http://doi.acm.org/10.1145/2465848.2465849

[9] Doppelhammer, J., Höppler, T., Kemper, A. and Kossmann, D. 1997. Database performance in the real world: TPC-D and SAP R/3. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data* (SIGMOD '97), Joan M. Peckman, Sudha Ram, and Michael Franklin (Eds.). ACM, New York, NY, USA, 123-134. DOI=http://doi.acm.org/10.1145/253260.253280

[10] Floratou, A., Teletia, N., DeWitt, D.J., Patel, J.M. and Zhang, D. 2012. Can the elephants handle the NoSQL onslaught?. *Proc. VLDB Endow.* 5, 12 (August 2012), 1712-1723. DOI=http://dx.doi.org/10.14778/2367502.2367511

[11] Gray, J. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Harizopoulos, S., Abadi, D.J., Madden, S. and Stonebraker, M. 2008. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (SIGMOD '08). ACM, New York, NY, USA, 981-992. DOI=http://doi.acm.org/10.1145/1376616.1376713

[13] Kuhlenkamp, J., Klems, M. and Röss, O. 2014. Benchmarking scalability and elasticity of distributed database systems. *Proc. VLDB Endow.* 7, 12 (August 2014), 1219-1230. DOI=http://dx.doi.org/10.14778/2732977.2732995

[14] MongoDB Inc. 2015. BSON Types. (June 2015). Retrieved June 18, 2015 from http://docs.mongodb.org/manual/reference/bson-types/

[15] MongoDB Inc. 2015. MongoDB. (June 2015). Retrieved June 18, 2015 from http://www.mongodb.org/

[16] MongoDB Inc. 2015. The MongoDB 2.4 Manual. (June 2015). Retrieved June 18, 2015 from http://docs.mongodb.org/v2.4/

[17] Neumann, T. and Weikum, G. 2010. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endow.* 3, 1-2 (September 2010), 256-263. DOI=http://dx.doi.org/10.14778/1920841.1920877

[18] Neumann, T. and Weikum, G. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (February 2010), 91-113. DOI=http://dx.doi.org/10.1007/s00778-009-0165-y

[19] Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., Dewitt, D.J., Madden, S. and Stonebraker, M. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (SIGMOD '09), Carsten Binnig and Benoit Dageville (Eds.). ACM, New York, NY, USA, 165-178. DOI=http://doi.acm.org/10.1145/1559845.1559865

[20] Salvatore Sanfilippo. 2015. Redis. (June 2015). Retrieved June 18, 2015 from http://redis.io/

[21] Smart Vortex. 2015. Retrieved June 18, 2015 from http://www.smartvortex.eu/

[22] solid IT. DB-Engines Ranking. (June 2015). Retrieved June 18, 2015 from http://db-engines.com/en/ranking

[23] Stonebraker, M. 2010. SQL databases v. NoSQL databases. *Commun. ACM* 53, 4 (April 2010), 10-11. DOI=http://doi.acm.org/10.1145/1721654.1721659

[24] Transaction Processing Performance Council (TPC). 2015. Active TPC Benchmarks. (June 2015). Retrieved June 18, 2015 from http://www.tpc.org/information/benchmarks.asp

[25] Truong, T. and Risch, T. 2014. Scalable Numerical Queries by Algebraic Inequality Transformations. In *Proceedings of the 19th International Conference on Database Systems for Advanced Applications* (DASFAA '14) (Bali, Indonesia, 2014). Springer International Publishing, 95–109. DOI= http://dx.doi.org/10.1007/978-3-319-05810-8_7

[26] Wei, J., Zhao, Y., Jiang, K., Xie, R. and Jin, Y. Analysis farm: A cloud-based scalable aggregation and query platform for network log analysis. In *Proceedings of the 2011 International Conference on Cloud and Service Computing* (CSC '11). IEEE Computer Society, Washington, DC, USA, 354-359. DOI=http://dx.doi.org/10.1109/CSC.2011.6138547

[27] Zeitler, E. and Risch, T. 2011. Massive Scale-out of Expensive Continuous Queries. *Proc. VLDB Endow*. 4, 11 (2011), 1181–1188.

[28] Zeitler, E. and Risch, T. 2010. Scalable splitting of massive data streams. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications*, (DASFAA '10) (Tsukuba, Japan, Apr. 2010). Springer International Publishing, 184–198. DOI=http://dx.doi.org/10.1007/978-3-642-12098-5_15

[29] Zhu, M., Stefanova, S., Truong, T. and Risch, T. 2014. Scalable Numerical SPARQL Queries over Relational Databases. In *Proceedings of 4th international workshop on linked web data management* (Athens, Greece, 2014).