Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual

Jan Sipke van der Veen, Bram van der Waaij
TNO
Groningen, The Netherlands
{jan_sipke.vanderveen, bram.vanderwaaij}@tno.nl

Robert J. Meijer
University of Amsterdam
Amsterdam, The Netherlands
rmeijer@science.uva.nl

***Abstract*—Sensors are used to monitor certain aspects of the physical or virtual world and databases are typically used to store the data that these sensors provide. The use of sensors is increasing, which leads to an increasing demand on sensor data storage platforms. Some sensor monitoring applications need to automatically add new databases as the size of the sensor network increases. Cloud computing and virtualization are key technologies to enable these applications. A key issue therefore becomes the performance of virtualized databases and how this relates to physical ones. Traditional SQL databases have been used for a long time and have proven to be reliable tools for all kinds of applications. NoSQL databases have gained momentum in the last couple of years however, because of growing scalability and availability requirements. This paper compares three databases on their relative performance with regards to sensor data storage: one open source SQL database (PostgreSQL) and two open source NoSQL databases (Cassandra and MongoDB). A comparison is also made between running these databases on a physical server and running them on a virtual machine. A minimal sensor data structure is used and tested using four operations: a single write, a single read, multiple writes in one statement and multiple reads in one statement.**

***Keywords*-Sensor Data; Data Storage; Performance; SQL; NoSQL; Physical Server; Virtual Machine.**

I. Introduction

The last decade has seen a large increase in sensor usage. Sensors have become cheaper, smaller and easier to use, leading to a growing demand on storage and processing of sensor data. Typical applications include sensing of physical structures to detect anomalies [1], monitoring the energy grid to match the production and consumption of energy [2], the internet of things [3] and smart environments such as smart homes and buildings [4].

The aforementioned sensor systems have in common such features as storing and processing of large amounts of data. Scaling these systems becomes an increasingly difficult task as both the amount of sensors and the number of clients accessing these systems grows rapidly. The CAP theorem [5] states that you can obtain at most two out of three properties in such a shared data system: consistency (C), availability (A) and tolerance to network partitions (P). SQL and NoSQL databases often make two different choices out of these three properties.

SQL databases are often chosen to store and query large amounts of data. This is mainly due to developers being familiar with them and because of the stability of these databases. However, distributing SQL databases to a very large scale is difficult. Because these databases are built for the support of consistency and availability, there is less tolerance for network partitions, which makes it difficult to scale SQL databases horizontally. Adding more capacity to such a database thus boils down to adding more processing and storage power to a single machine.

NoSQL databases try to solve some of the problems traditional SQL databases pose. By relaxing either consistency or availability, NoSQL databases can perform quite well in circumstances where these constraints are not necessary [6]. For example, NoSQL databases often provide weak consistency guarantees, such as eventual consistency [7]. This relaxation of a certain constraint may be good enough for a whole array of applications. By doing so, these databases can become tolerant to network partitions, which makes it possible to add servers to the setup when the number of sensors or clients increases, instead of adding more capacity to a single server.

This paper discusses the possibilities to use NoSQL databases in large-scale sensor data systems and provides a performance comparison to the use of a traditional SQL database for such purposes. PostgreSQL is chosen as a representative of SQL databases, because it is a widely used and powerful open source database supporting the majority of the SQL 2008 standard. Cassandra and MongoDB are chosen as representatives of NoSQL databases, because they are both powerful open source databases with a growing community developing and supporting them. In addition, the same tests are performed on a physical server and on a virtual machine to assess the performance impact of virtualization.

Related work includes papers aimed at the design or use of benchmarking suites to assess the performance of a system under test. Some of this work is general in nature [8], while others are aimed specifically at database performance [9] [10] [11] or aimed specifically at the impact of virtualization on performance [12] [13]. This paper differs from these benchmarking approaches because of a focus on sensor data storage and a combination of comparisons between databases and virtualization.

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.18

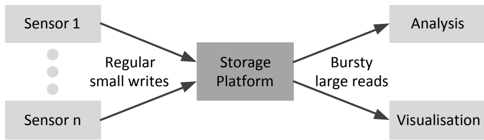

Figure 1. Regular small writes and bursty large reads

II. Sensor Data

Sensors typically output their measurements at regular intervals. For example, a temperature sensor might transmit its sensed value every 30 seconds. The data associated with that temperature measurement is typically a few bytes. If no or little buffering is used, the sensor data system therefore receives fairly small write requests at regular intervals.

Analysis and visualization components are typical consumers of data from the sensor data system. An analysis component might take the values of a 24 hour period and calculate the average, and a visualization component might visualize the same 24 hour period in a graph. These read requests are more ad-hoc and contain more than one value.

As we can see in Fig. 1, the read requests from these components are more bursty in nature and contain more than one value, while the sensors output single values at regular intervals.

Sensors take measurements of their surroundings at some interval. Each measured value is stored in the sensor data system along with an identifier of the sensor and a timestamp that signifies at what date and time the value was sensed. This is the minimum amount of information a general purpose sensor data system needs. Other information may be added as well, e.g. the location of the sensor at the time of measurement.

III. Test Subjects

A. *SQL versus NoSQL*

Traditional SQL databases and more recent NoSQL databases differ in some significant aspects. SQL database use fixed table structures and offer the ability to select data from these tables with a structured query language. Using the *join* operation, it is possible to select data that spans multiple tables. An SQL database typically scales well in a vertical manner, i.e. by upgrading the single server it runs on. However, it scales worse in a horizontal manner, i.e. by adding a server to a cluster.

NoSQL databases on the other hand typically use key-value stores, where data is stored and retrieved by key. There is no fixed data structure and a less powerful query language is available to retrieve the stored data. In return, the NoSQL database is often easier to scale horizontally, which makes it easy to add capacity at the moment it is needed.

NoSQL databases are also attributed higher performance, because they are easier to design and implement. In chapter IV we test this claim for the sensor data case, using three open source databases: one traditional SQL database and two NoSQL databases.

Cassandra [14] [15] is an open source NoSQL database. It exists since 2008 and is designed to cope with very large amounts of data spread out across many commodity servers. It provides a structured key-value store using the mechanism of eventual consistency [16]. Cassandra uses Thrift [17] for its external client-facing API. Although Thrift can be used as-is, it is recommended by the authors to use a higher-level client. In this paper, we use Hector [18] as the Java client library to connect to Cassandra.

MongoDB [19] [20] is an open source NoSQL database. It exists since 2007 and provides a key-value store that manages collections of BSON (binary JSON) documents [21]. It is designed to run on one or a few commodity servers and includes a powerful query language that allows for regular expressions and Javascript functions to be passed in as checks for matching keys and values.

PostgreSQL [22] [23] is a traditional open source SQL database. It exists since 1995 and can be seen as a heavily used representative of SQL databases. It is designed to support the majority of the SQL 2008 standard, is ACID-compliant (Atomicity, Consistency, Isolation, Durability) and is fully transactional.

B. *Physical versus Virtual*

Cloud computing and more specifically virtualization of resources is gaining in popularity. Virtualization can provide better utilization of resources because they are shared between several applications and users. This means that the cost of computing can be lower, it is easier to move computation from one physical location to another location, and designing for scalability becomes easier. Virtualization also has a negative side, however, because the virtualization layer itself imposes an impact on performance.

The raw performance impact has been measured in other works such as [24] and [25], but we focus here on the specific impact for databases that store sensor data. The databases present real-life workloads on the systems which may be quite different from (synthetic) workloads used by benchmark suites.

In our comparative overview we use a physical server and a virtual machine running on exactly the same physical server, but then with a virtual machine monitor (VMM) installed on it. In this test we use XenServer [26], one of the most widely used virtualization solutions. It supports both fully virtualized guest operating systems, unaware that they are running on a virtual machine, and para-virtualized guest operating systems with specific XenServer drivers.

IV. Test Setup

Fig. 2 shows the setup that is used to test the performance of the different databases. A single client machine running a Java program connects to a single server machine running

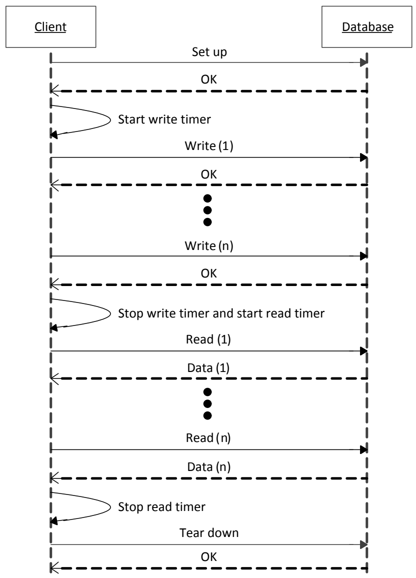

Figure 2. Test setup

Table I
DATA STRUCTURE

Name	Type
sensor_identifier	uuid
sensed_timestamp	long
sensed_value	double

one of three databases. The Java program uses threads to simulate multiple clients. The databases each run on a single machine, no duplication and/or distribution is used.

Each test in this paper measures the time it takes to write data to the database or read data from it. This time is measured from the client that performs the test. Any setup work that needs to be performed, e.g. creating a table, is not measured. Each test starts with an empty database, write tests are performed of a certain size and then read tests of the same size are performed.

A. Data structure

The basic data structure for storing the sensor data can be seen in table I. This is the minimal set of columns for a general purpose sensor system. It contains an identifier of the sensor, which is necessary if there is more than one sensor. It also contains the time the measurement was performed, and the actual value of the measurement.

B. Operations

Four types of operations are performed on the databases:

- A single write in one statement. In this case a single measurement is inserted into the database, with one sensor identifier, one timestamp and one value.
- A single read in one statement. In this case a single measurement is read from the database, specifying the sensor identifier and the timestamp of the measurement.

Table II
HARDWARE

	Server	Client
CPU type	AMD Opteron	Intel Xeon
CPU cores	24	2
CPU speed	2.2 GHz	3.0 GHz
Memory size	64 GB	4 GB
Disk size	2 x 2.0 TB	250 GB
Disk rotations	7200 RPM	7200 RPM
Disk cache	64 MB	8 MB
Network speed	1 Gb/s	1 Gb/s

Table III
SOFTWARE

	Server	Client
OS	CentOS 6.2 x86_64	CentOS 6.2 x86_64
JDK	Oracle 1.6.0_30	Oracle 1.6.0_30
Cassandra	1.0.7	Hector core 1.0-3
MongoDB	2.0.2	Java driver 2.7.3
PostgreSQL	9.1.2	JDBC4 9.1-901

- Multiple writes in one statement. In this case 1,000 measurements are inserted at once into the database, with one sensor identifier and multiple timestamp + value combinations.
- Multiple reads in one statement. In this case 1,000 measurements are read from the database, specifying the sensor identifier and a begin and end timestamp.

When multiple operations are performed in one statement, the given possibilities of each database are used. In case of PostgreSQL, a concatenated SQL query is constructed. In case of MongoDB the batch option of the client library and for Cassandra the batch option of the Hector client is used.

This paper focuses on sensor data and therefore the results of the single write and multiple read operations are most interesting, but the other operations are included as well to be of more general use.

C. Hardware and software

Table II lists the hardware and table III lists the software that was used for the tests. In addition, XenServer 6.0.0 software was used for hosting a para-virtualized virtual machine. This virtual machine contains drivers (guest tools) with speed and management advantages compared to normal drivers that are provided by the operating system.

D. Configuration

The server machine contains two hard disks. Each database is configured to make use of one hard disk for its data and the other hard disk for logging and caching. This allows the databases to write log files and perform caching without a large impact on data write and read performance.

The server and client machines are connected through a dedicated network connection between them, i.e. a UTP cable directly connecting them. This ensures that there is no other network traffic that might disturb the tests, but does account for the fact that database servers and clients are typically connected through a network.

1) Cassandra: The Cassandra configuration file *cassandra.yaml* is changed from the default to set directories for data, log and cache files, and specify network parameters:

```
data_file_directories:
   - /data1/cassandra/data
commitlog_directory:
   /data2/cassandra/commitlog
saved_caches_directory:
   /data2/cassandra/saved_caches
listen_address: 10.0.0.1
rpc_address: 10.0.0.1
seeds: "10.0.0.1"
```

2) MongoDB: The MongoDB configuration file *mongod.conf* is changed from the default to set directories for data and log files:

```
dbpath=/data1/mongo/data
logpath=/data2/mongo/log/mongod.log
```

3) PostgreSQL: The PostgreSQL initialization script *postgresql-9.1* is changed from the default to set directories for data and log files:

```
PGDATA=/data1/postgres/data
PGLOG=/data2/postgres/log/pgstartup.log
```

The configuration file postgresql.conf is changed from the default to specify network parameters:

```
listen_addresses='10.0.0.1'
```

The configuration file pg_hba.conf is changed from the default to allow non-local clients to connect:

```
host all all 10.0.0.0/24 md5
```

V. Results

The results are split up into five different categories:

- The presence of *indexes* (section V-A).
- A *single* client with a *single* read or write operation in one statement (section V-B).
- A *single* client with *multiple* read or write operations in one statement (section V-C).
- *Multiple* clients with a *single* read or write operation in one statement (section V-D).
- *Multiple* clients with *multiple* read or write operations in one statement (section V-E).

For each of these five categories the results are presented for the physical server case and the virtual machine case.

A. Use Of Indexes

MongoDB and PostgreSQL can be used with or without indexes. In these databases, the user is responsible for defining one or more keys or unique constraints that can be used by the database to create indexes and thereby speed up locating data. Cassandra always uses the type of the key the user defines, e.g. integer or long, to create its own indexes.

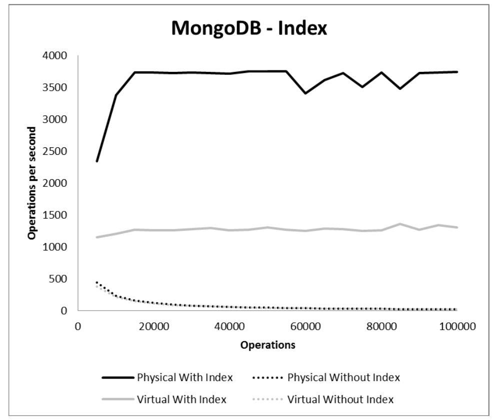

Figure 3. Use of indexes in MongoDB

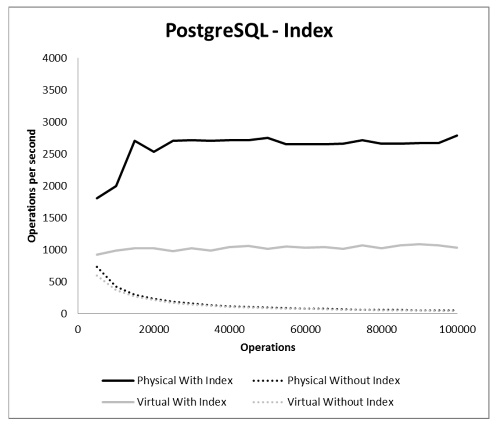

Figure 4. Use of indexes in PostgreSQL

See Fig. 3 and Fig. 4 for a comparison between reads with and without an index for MongoDB and PostgreSQL. The speed difference between writes with and without indexes is very low, so they are not shown in the figures.

The presence of an index has a tremendous effect on performance both for MongoDB and PostgreSQL. If no index is present, they need to perform scans to reach the data the client asks for. This is a very slow operation, especially for large data structures. Because using indexes has such a tremendous positive effect on reads and such a low overhead on writes, we use indexes in the remainder of this paper.

The difference between a physical server and a virtual machine running the databases is quite high in the case with indexes and low in the case without indexes.

B. Single Client, Single Operation

Fig. 5 shows the performance of a single client issuing single write requests to the database. There is a huge difference between the three databases in this case, as

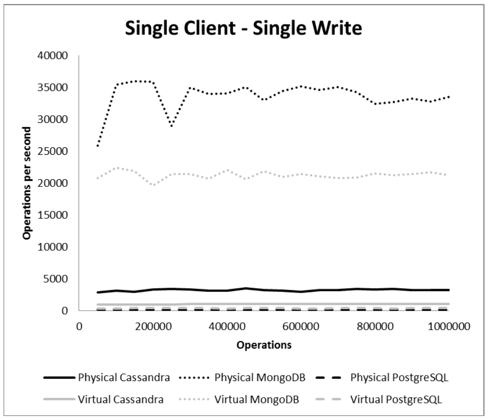

Figure 5. Single client, single write

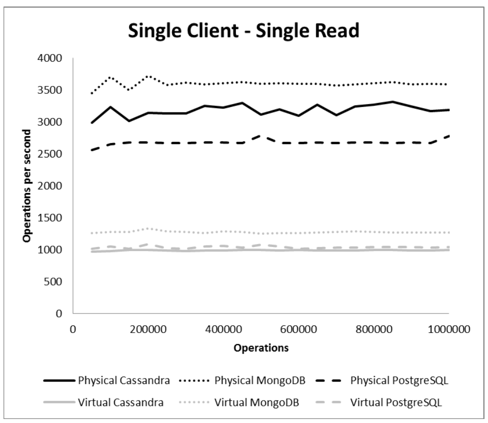

Figure 6. Single client, single read

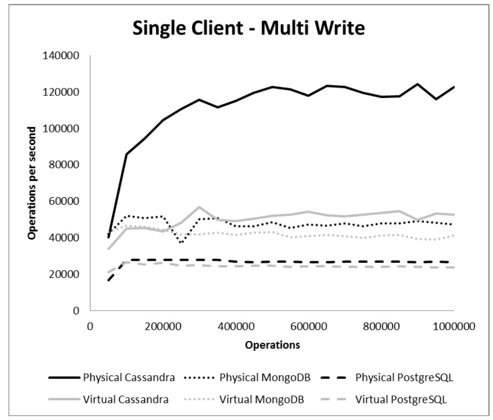

Figure 7. Single client, multiple writes

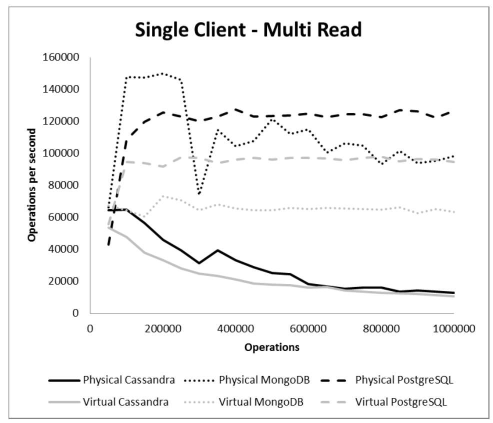

Figure 8. Single client, multiple reads

MongoDB performs very well, Cassandra scores moderately and PostgreSQL performs very poorly.

Fig. 6 shows the performance of a single client issuing single read requests to the database. The order is the same as with writes, but the difference between the three databases is much less when reading.

The difference between a physical server and a virtual machine running these databases is remarkable. The read performance is affected about equally between the databases, but the write performance is quite different. The physical server wins in the case of Cassandra and MongoDB, but in the case of PostgreSQL the virtual machine is a lot faster than the physical server.

C. Single Client, Multiple Operations

Fig. 7 shows the performance of a single client issuing multiple write requests to the database. Cassandra is the clear winner in this case, MongoDB is second and PostgreSQL is third again.

Fig. 8 shows the performance of a single client issuing multiple read requests to the database. PostgreSQL takes the lead here and MongoDB is a close second. Cassandra performs poorly in this test as the number of operations per second drops significantly when the number of operations is increased.

The difference between a physical server and a virtual machine running these databases is not very high. Cassandra seems to be a bit more affected by virtualization in the write case than MongoDB and PostgreSQL.

D. Multiple Clients, Single Operation

Fig. 9 shows the performance of multiple clients issuing single write requests to the database. MongoDB is very fast when a single client is used, but decreases slowly when clients are added. Cassandra initially benefits from multiple clients, as the number of operations per second increases from 1 client to 12 clients, but decreases again when the number of clients is increased further. Although PostgreSQL is quite slow here, it benefits even more from multiple clients, as the number of operations per second increases steadily from 1 client to 16 clients.

Fig. 10 shows the performance of multiple clients issuing single read requests to the database. All databases benefit

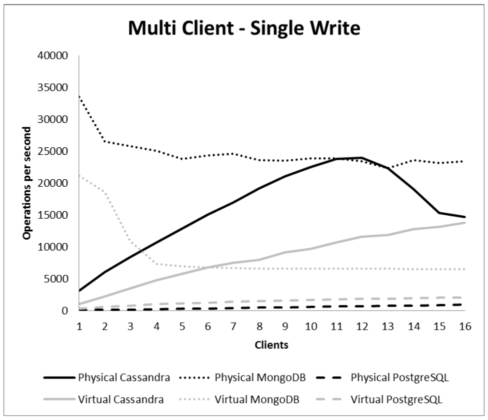

Figure 9. Multiple clients, single write

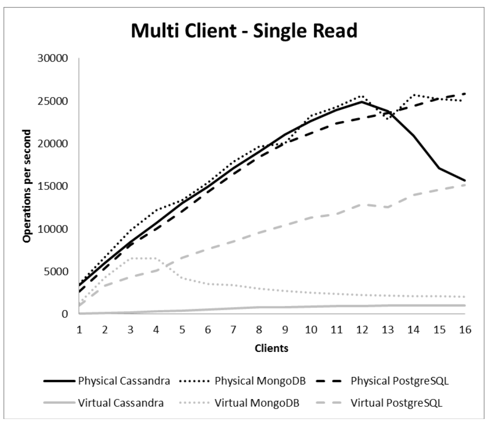

Figure 10. Multiple clients, single read

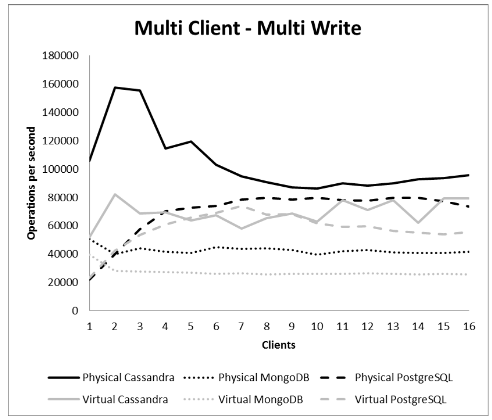

Figure 11. Multiple clients, multiple writes

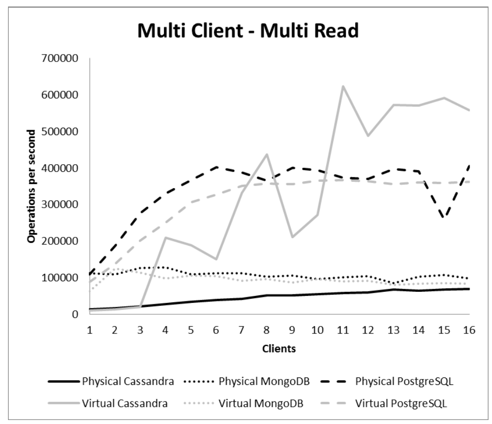

Figure 12. Multiple clients, multiple reads

from multiple clients when reading data. The number of operations per second rises steadily for MongoDB and PostgreSQL from 1 client to 16 clients. Cassandra shows the same behavior for reads as for writes, as the number of operations per second increases from 1 client to 12 clients and decreases again when more clients are used.

The performance difference between a physical server and a virtual machine is quite remarkable for both write and read operations. The physical server wins in the case of writing to MongoDB and Cassandra, but the virtual machine is faster when writing to PostgreSQL. The impact on read performance is very high for MongoDB and Cassandra, the virtual machine is more than ten times slower than the physical server in this case.

E. Multiple Clients, Multiple Operations

Fig. 11 shows the performance of multiple clients issuing multiple write requests to the database. MongoDB and Cassandra both do not benefit from multiple clients. The number of operations per second stays roughly the same between 1 and 16 clients. PostgreSQL does benefit from multiple clients as the number of operations per second rises steadily from 1 client to 16 clients.

Fig. 12 shows the performance of multiple clients issuing multiple read requests to the database. The number of operations per second for MongoDB stays the same from 1 client to 16 clients. Both Cassandra and PostgreSQL benefit strongly from multiple clients in the read case.

The impact on write performance of virtualization is quite low for all three databases. The impact on read performance is about the same for PostgreSQL and MongoDB, but Cassandra is quite different as the performance on the virtual machine is much higher than on the physical server.

F. Discussion

Table IV summarizes the results of the previous sections for single write and read operations. Table V does the same for multiple write and read operations.

Cassandra performs well overall, but is heavily influenced by virtualization, both positively and negatively. Its multi client single read performance drops by a factor of 16 when a virtual machine is used instead of a physical server.

Table IV
NUMBER OF SINGLE OPERATIONS PER SECOND (GREY IS HIGHEST)

	Single Write		Single Read	
	Single Client	Multi Client	Single Client	Multi Client
Cassandra				
- physical	3,200	15,000	3,200	16,000
- virtual	1,000	14,000	1,000	1,000
MongoDB				
- physical	34,000	23,000	3,600	25,000
- virtual	21,000	6,500	1,300	2,000
PostgreSQL				
- physical	120	930	2,800	26,000
- virtual	400	2,100	1,000	15,000

Table V
NUMBER OF MULTIPLE OPERATIONS PER SECOND (GREY IS HIGHEST)

	Multi Write		Multi Read	
	Single Client	Multi Client	Single Client	Multi Client
Cassandra				
- physical	120,000	95,000	13,000	69,000
- virtual	53,000	79,000	11,000	560,000
MongoDB				
- physical	47,000	41,000	99,000	98,000
- virtual	41,000	26,000	64,000	83,000
PostgreSQL				
- physical	27,000	73,000	130,000	410,000
- virtual	24,000	55,000	95,000	360,000

Multiple reads in one statement on the other hand benefit from virtualization by a factor of 8.

MongoDB performs very well when a single client is used. Especially in the single write case it outperforms the other two databases significantly. Virtualization has a smaller impact on MongoDB than on Cassandra, but it is still visible. In the multi client single read case the physical server outperforms the virtual machine by a factor of 12.

PostgreSQL performs well when multiple clients are used. The impact of virtualization is lower for PostgreSQL than the other two databases. It even has a positive effect on single writes, the virtual machine outperforms the physical server with a factor of 3 here.

We ran each of the three databases on the same single machine and used another single machine as a client. This may have hurt Cassandra more than the other databases, because Cassandra was built from the ground up to run on multiple machines for robustness and scalability. We still chose this setup because it allowed the best comparison between the three databases.

In section II we concluded that sensors write small pieces of data to the storage system, and analysis and visualization components read large pieces of data from it. The ideal database would therefore be one that performs well in these areas. However, from tables IV and V we can see that there is no clear winner on both fronts.

VI. CONCLUSIONS AND FUTURE WORK

A database that performs well on single writes and multiple reads would be the ideal candidate for storing sensor data. Of the three databases we tested, there is no single database that performs best in both cases as MongoDB wins at single writes and PostgreSQL wins at multiple reads. It therefore depends on the requirements of the sensor application which database is better at the task.

Virtualization has an impact on the performance of the databases we tested, but not always as one might expect. A lot of tests showed that performance is lower when using a virtual machine instead of a physical server. However, there are also several tests which showed that performance may be positively affected by virtualization. We assume this is due to caching by the virtual machine monitor (VMM).

Comparing the uses of the three databases, we can conclude the following:

- Cassandra is the best choice for large critical sensor applications as it was built from the ground up to scale horizontally. Its read performance is heavily affected by virtualization, both positively and negatively.
- MongoDB is the best choice for a small or medium-sized non-critical sensor application, especially when write performance is important. There is a moderate performance impact when using virtualization.
- PostgreSQL is the best choice when very flexible query capabilities are needed or read performance is important. Small writes are slow, but are positively affected by virtualization.

This paper uses only one machine to run the databases on. As future work we plan to distribute the databases across multiple machines. We also intend to add other types of databases to the test, such as those offered by cloud computing providers. Another important future work will be going into more detail on the positive performance impact of virtualization.

VII. ACKNOWLEDGMENTS

This publication is supported by the EC FP7 project UrbanFlood, grant N 248767, and the Dutch national program COMMIT.

REFERENCES

[1] V. Krzhizhanovskaya, G. Shirshov, N. Melnikova, R. Belleman, F. Rusadi, B. Broekhuijsen, B. Gouldby, J. Lhomme, B. Balis, M. Bubak, A. Pyayt, I. Mokhov, A. Ozhigin, B. Lang, and R. Meijer, "Flood early warning system design, implementation and computational modules," *International Conference on Computational Science*, 2011.

[2] S. Rusitschka, K. Eger, and C. Gerdes, "Smart grid data cloud: A model for utilizing cloud computing in the smart grid domain," *IEEE International Conference on Smart Grid Communications*, 2010.

[3] N. Bui, “Internet of things architecture,” 2012, http://www.iot-a.eu/public/public-documents.

[4] W. Leibbrandt, “Smart living in smart homes and buildings,” *IEEE Technology Time Machine Symposium on Technologies Beyond 2020*, 2011.

[5] E. A. Brewer, “Towards robust distributed systems,” *Symposium on Principles of Distributed Computing*, 2000.

[6] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. OReilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum, “The claremont report on database research,” 2008, http://db.cs.berkeley.edu/claremont.

[7] W. Vogels, “Eventually consistent,” *Communications of the ACM*, 2009.

[8] R. Eigenmann, G. Gaertner, W. Jones, H. Saito, and B. Whitney, “Spec hpc2002: The next high-performance computer benchmark extended abstract,” *Lecture Notes In Computer Science*, 2006.

[9] W. Hsu, A. Smith, and H. Young, “Characteristics of production database workloads and the tpc benchmarks,” *IBM Systems Journal*, 2001.

[10] S. Ray, B. Simion, and A. D. Brown, “Jackpine: A benchmark to evaluate spatial database performance,” *IEEE International Conference On Data Engineering*, 2011.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” *ACM Symposium on Cloud Computing*, 2010.

[12] M. A. El-Refaey and M. A. Rizkaa, “Cloudgauge: A dynamic cloud and virtualization benchmarking suite,” *Enabling Technologies: Infrastructures for Collaborative Enterprises*, 2010.

[13] N. Yigitbasi, A. Iosup, and D. Epema, “C-meter: A framework for performance analysis of computing clouds,” *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2009.

[14] Cassandra, “Cassandra,” 2012, http://cassandra.apache.org.

[15] E. Hewitt, *Cassandra: The Definitive Guide*. O’Reilly Media, 2010.

[16] A. Lakshman and P. Malik, “Cassandra - a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, 2010.

[17] Thrift, “Thrift,” 2012, http://thrift.apache.org.

[18] Hector, “Hector,” 2012, https://github.com/rantav/hector.

[19] MongoDB, “Mongodb,” 2012, http://www.mongodb.org/.

[20] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*. O’Reilly Media, 2010.

[21] BSON, “Bson,” 2012, http://bsonspec.org.

[22] PostgreSQL, “Postgresql,” 2012, http://www.postgresql.org.

[23] J. D. Drake and J. C. Worsley, *Practical PostgreSQL*. O’Reilly Media, 2002.

[24] N. Huber, M. von Quast, M. Hauck, and S. Kounev, “Evaluating and modeling virtualization performance overhead for cloud environments,” *International Conference on Cloud Computing and Services Science*, 2011.

[25] A. J. Younge, R. Henschel, J. T. Brown, G. von Laszewski, J. Qiu, and G. C. Fox, “Analysis of virtualization technologies for high performance computing environments,” *International Conference on Cloud Computing*, 2011.

[26] Citrix, “Xenserver,” 2012, http://www.citrix.com/xenserver.