Thesis to get the degree of master in computer science

Data schemas of noSQL stores for the storage of sensor data

Fotis Katsantonis

17/02/2014

University of Groningen
Faculty of Mathematics and Natural Sciences
Dean
prof. dr. J. (Jasper) Knoester

Supervisors
Elena Lazovik (TNO)
Bram van der Waaij (TNO)
prof. dr. Ir. Marco Aiello (RUG)
# Contents

Abstract 1

1. Introduction 3

2. Problem statement 7

3. Sensor data characteristics 9
   3.1. Sensor data dismantling ........................................... 9
   3.1.1. Measurement values ................................................ 9
   3.1.2. Metadata ............................................................ 10
   3.2. Sensor data properties ............................................... 12
   3.2.1. General dataset properties ...................................... 12
   3.2.2. Sensor data properties ............................................ 13
   3.2.3. Time series properties ............................................ 13
   3.2.4. Applicability of noSQL databases ................................ 15
   3.3. Sensor data usage .................................................... 15
   3.4. Business cases ....................................................... 17

4. State of the art 21
   4.1. Database theorems .................................................... 21
   4.1.1. CAP ................................................................. 21
   4.1.2. ACID & BASE ...................................................... 23
   4.2. noSQL taxonomy overview ......................................... 23
   4.2.1. General taxonomy ................................................ 23
   4.2.2. Specific taxonomies ............................................... 25

5. Overview of time series data schemas for noSQL stores 29
   5.1. Column-oriented noSQL databases .................................. 29
   5.1.1. Column-oriented databases data model ......................... 29
   5.1.2. General guidelines ............................................... 30
   5.1.3. Data schemas for column-oriented databases .................. 31
   5.2. Document noSQL databases ......................................... 44
   5.2.1. Document databases data model ................................ 44
   5.2.2. Data schemas for document databases .......................... 44
   5.3. Next steps ............................................................ 51
6. Test scenarios
  6.1. Pre-test phase ........................................ 53
  6.2. Different scenarios .................................. 54
  6.3. Metrics measured ..................................... 55
    6.3.1. Data schema and terminology ..................... 56
    6.3.2. Test parameters .................................. 59
      6.3.2.1. Measured metrics ............................. 60

7. Test setup .................................................. 63

8. Cassandra Hector tests .................................... 67
  8.1. First set of load tests ............................... 67
    8.1.1. Client program .................................. 67
    8.1.2. Test parameters ................................ 69
    8.1.3. First Hector load test ......................... 70
      8.1.3.1. Pre-tests .................................. 70
      8.1.3.2. Test results ............................... 73
  8.2. Second set of load tests ............................. 77
    8.2.1. Tuning Cassandra ................................. 78
    8.2.2. Configuration settings for Cassandra .......... 79
      8.2.2.1. Thrift interface ............................ 80
      8.2.2.2. Java parameters ............................. 82
    8.2.3. Test results .................................... 82
  8.3. Comparison of the two tests ........................ 87
    8.3.1. Summary ......................................... 89

9. CQL performance tests ..................................... 91
  9.1. Client differences .................................. 91
    9.1.1. Metric measurement ............................. 93
    9.1.2. Different scenarios ............................. 94
      9.1.2.1. Pre-tests .................................. 95
      9.1.2.2. Manual batching ............................. 95
      9.1.2.3. Typed values ................................ 96
      9.1.2.4. Metadata .................................... 96
      9.1.2.5. Timewidth ................................... 99
  9.2. Test structure ........................................ 99
    9.2.1. Metrics ......................................... 101
    9.2.2. Resulting diagrams ............................. 101
    9.2.3. Test parameters ................................. 102
  9.3. Test results .......................................... 103
    9.3.1. Number of column families pre-test ............. 103
    9.3.2. Typed value tests ............................... 106
      9.3.2.1. Different techniques for sensor metadata using typed values ............................. 106
Contents

9.3.2.2.  Timewidth for typed values .......................... 112
9.3.3.  Number of values per batch pre-test ....................... 116
9.3.4.  Manual batching ........................................ 120
  9.3.4.1.  Different techniques for sensor metadata using manual batching .......................... 121
  9.3.4.2.  Timewidth for manual batching ........................ 125
9.3.5.  Tests summary ........................................... 130
  9.3.5.1.  Suggestions for data schemas with regard to different use cases ....................... 131

10. Conclusion ..................................................... 135

Acknowledgments .................................................. 139

A. Database techniques and terminology .......................... 141
  A.1. Consistency guarantees ..................................... 141
  A.2. Data replication ........................................... 142
  A.3. Performance enhancing techniques .......................... 144
  A.4. Detailed tables for CQL 3 Driver tests ..................... 145

Bibliography ....................................................... 149
Abstract

In this thesis we are concerned with the storage of sensor data. Sensor data can also be characterized as time series data. The exponential increase of sensor data exposes the scalability issues of relational databases. noSQL databases are a new type of data stores, which are built with scalability and availability in mind. The capabilities provided by these databases, makes them a good candidate for the storage of time series data.

The purpose of this thesis is to compare different data schemas for noSQL databases and their applicability for the storage of sensor data. An overview of noSQL database taxonomies is presented. Also, various data schemas used by experts in production are researched and presented. This gives an insight on which data schemas are appropriate for the storage of time series.

Next, we perform load tests using Cassandra, which is a noSQL database suitable for the storage of sensor data. The tests are performed with regard to performance of different data schemas for the storage of sensor data. A test structure and scennarios are defined and different data schemas are tested on the basis of throughput and latency. The results are expected to provide suggestions for the suitable noSQL data schemas to use in different situations when storing time series data.
1. Introduction

Sensors are playing an increasingly critical role in society during the last years. Governments depend on sensors in order to foresee, plan and act against natural disasters, such as hurricanes, floods or earthquakes. Moreover, governments also use sensors for safety and security reasons, for example surveillance cameras ensure public safety in large events. Another case where we encounter sensors is in the industry where sensors are used for automating certain tasks, preventive maintenance and more.

Sensors have also started appearing in devices used in our everyday life. For example, a modern smart phone typically has a Global Positioning System (GPS) sensor, a proximity sensor and an accelerometer among other types of sensors. Another typical use of sensors is for researchers to use the sensor data for various analytical purposes, for example monitoring the air pollution levels or monitoring the health of a patient, so in different domains: energy, health care and more.

Another domain that makes heavy use of sensors is domotics. The sensors are used to enable the devices to sense certain aspects of the home environment and automate the device’s reactions to certain events. One common goal in many smart houses is to optimize energy consumption of the devices. For example, some energy companies distribute energy monitors to their customers. These monitors provide real time information on electricity consumption to the user. Healthy aging is another area where sensors are utilized, the sensors are used to monitor the activities performed by elderly people. For example, implantable wireless electrocardiography devices capture electrocardiogram data for diagnosing human cardiac arrhythmia. This data is transmitted to monitoring centers where irregular behavior is detected in time, before the problem fully manifests. This way the issue can be treated before it becomes life threatening.

The amount of data generated by all these sensors is really big and it requires efficient storage and processing. The traditional solution for storing sensor data is using a relational database, since this type of database is the most commonly encountered. However, due to the amount of data generated by sensors (petabytes/day are starting to become more and more common) this type of databases fails to meet the expected performance. Specifically, the traditional SQL databases utilized today were not designed to handle such large amount of data. Moreover, the database needs to be distributed over multiple computers, for processing and storage to be performed in reasonable time. But data sharding (c.f. sec.A.2) is a hard and error prone process to perform in a relational database. Furthermore, SQL databases reduce
the availability in a scenario of a network partition, because relational databases usually reject writes in favor of strong consistency.

Another problematic area of relational databases is the provided schema in the case of sensor data. Even though the relational model is strong and offers many capabilities for querying, in the process of adapting the data schema of a live database used in a production environment it seems to be too rigid. In order to make changes to an existing schema, at least some downtime of the database is to be expected. Therefore, a more flexible schema could be advantageous in the case of sensor data, since downtime cannot be tolerated due to the frequency of the data.

A possible solution to overcome the aforementioned issues is using a new type of databases, the noSQL stores. One of the goals of these databases is to allow horizontal scaling. Horizontal scaling means that instead of upgrading a single existing machine, new commodity machines are added to the database cluster which is preferable from a cost perspective. Further, noSQL databases typically support much better availability and fault tolerance by replicating (c.f. sec. A.2) the data. However, a drawback of noSQL stores is that they do not always guarantee strong consistency (c.f. Appendix A) of the data; for some applications it is tolerable (e.g. social media), while for other applications strong consistency is always needed (e.g. banking).

Furthermore, the relationships that are provided by relational databases can be used to model and store the relations between sensor metadata (c.f. sec. 3.1.2). But for raw measurements these relations are not required, on the contrary it could be said that additional underutilized overhead is created. A noSQL store that provides a more relaxed schema in favor of performance and availability, might be a better candidate for the storage of raw measurements of sensors.

On the whole the applicability of noSQL stores for the storage and retrieval of sensor data is explored. First, the domain of sensor data is narrowed down and briefly examined. Then, an overview of data schemas for noSQL is performed, which proves helpful in formulating different data schemas. Finally, the chosen data schemas are compared and evaluated with regard to performance. The test results suggest some general guidelines that can be followed when using noSQL databases to efficiently store sensor data.

The structure of the thesis is described next. In chapter 2 the problem that we are trying to tackle is described. Also the approach we take on the problem is outlined on the same chapter.

Next, in chapter 3 we provide an overview of the characteristics of time series/sensor data, which helps us understand the problem domain. Furthermore, some common usage patterns of sensor data and some business cases around sensor data are presented.

Afterwards, in chapter 4 a small overview of database theorems is given. In the same section, an overview of noSQL taxonomies is presented along with some databases for the different noSQL categories.
Following in chapter 5 is an overview of different data schemas suggested by experts with experience in the usage of noSQL databases for the storage of time series data.

Next we move to the practical part of this thesis, which is performing load tests with regard to different data schemas using Cassandra[1]. First, in chapter 6 we formulate the test cases upon which the different data schemas are compared. After, in chapter 7 the hardware and network setup of the Cassandra cluster is shown.

We proceed with the first set of tests using a synchronous client in sec. 8.1 and sec. 8.2. Then we perform the same test cases using an asynchronous client of Cassandra in chapter 9. The results of the tests are compared and discussed and the applicability of the different data schemas for different use cases are discussed.

Finally, some conclusions with regard to the outcome of this thesis are presented in chapter 10. Also possible directions for future work are.
2. Problem statement

The amount of data generated by machines increases exponentially, reaching even peta-bytes of data on a daily basis. Relational databases, that have been typically used for a plethora of use cases in the past, are not able to efficiently cope with this amount of data. Data sharding is a possible solution, however this is a complex and error prone process. Upgrading single machines (scaling-up) is cost inefficient, scaling-out (adding new commodity servers) is the preferred approach.

noSQL databases are a relatively new technology that use special means to cope with the issues described above, making them a good candidate for the storage of sensor data. They provide an easy way to add and remove nodes from a cluster, which leads to greater scalability, availability and flexibility. Some noSQL stores are known to be able to handle hundreds of thousands to millions of inserts per second (e.g. Cassandra, HBase). However, this performance, scalability and availability do not come without a trade-off, the schema is negatively affected. noSQL databases provide different possibilities schema-wise, which are not as rich as the relational schema. Therefore, an efficient way of modeling domains for noSQL databases is required.

First, we explore the domain of sensor/time series data. Next, we formulate an overview of noSQL database taxonomies and an overview of data schemas to be used with noSQL databases for the storage of time series data. After gaining some insight on how to proceed for the storage of time series data, we perform load tests on Cassandra[1].

First, a structure is defined for the tests. The load tests are performed with regard to the performance of different data schemas for the storage of time series data. Different data schemas are evaluated on the basis of some generic database metrics (throughput (operation/second) and latency).

The results of the tests are discussed with regard to the applicability of different data schemas to different use cases. Furthermore, some general suggestions on how to proceed with the storage of time series data using noSQL databases are provided, after gaining knowledge by performing the tests. Some general suggestions for the data schema to be used are also provided. Finally, the outcome of this thesis and future work are discussed.
3. Sensor data characteristics

As with any process that involves modeling a real world scenario in a computer system, the start is to have good understanding of the problem domain. For this thesis the problem domain is massive amount of sensor data. However, sensor data is a big domain and can be further divided in sub-domains. We try to keep the scope of sensor data to a high level, so the results of the thesis can be applied to more than one domains. Therefore, having a strong classification and understanding of the properties of sensor data is important.

Sensor data consist of measurement values and metadata. These categories of data have different properties, which acts as a hint that they should also be treated differently in order to be efficiently stored. Furthermore, the general purpose of sensor metadata is different than that of measurement values. Metadata is used mostly for management of the sensor data, while raw measurements reflect the state of a real world object/phenomenon. A description is given for each type of sensor data in sec. 3.1.1 and sec. 3.1.2. After, some properties of sensor data are presented in sec. 3.2, followed by some common access patterns for the sensor data in sec. 3.3. Finally, some examples of sensor data usage are presented in sec. 3.4.

3.1. Sensor data dismantling

In this section sensor data are further subdivided to two categories: measurement values and metadata. This distinction is made because these two categories present different characteristics. A description of each category along with their characteristics is further elaborated next.

3.1.1. Measurement values

This category of sensor data represents the measurement values that the sensor has recorded. Besides the measurement values, the timestamp when the data was recorded needs to be recorded. Otherwise, the meaning of the values looses context over time. Measurement values need a lot of physical storage, since each reading needs to be stored. This is in contrast to metadata that need to be stored only once and apply for all the measurement values. Measurement values do not have complex relationships defined among them usually, unless we are dealing with multivariate...
data (c.f. sec. 3.2.1). They only need to be related to the corresponding metadata. This usually achieved by using a naming convention for the raw measurements and the metadata. For example using the ID of the sensor in the metadata and raw measurements is a good way to correlate the metadata to raw measurements.

Sensor data is often called time series data, because the measurement values arrive in regular time intervals (e.g. every minute). Corsello in [35], describes time series as “A time series is defined as a fixed structure of data collected repeatedly over time at fixed intervals. This definition is very broad and as such allows for variability in several areas”. This kind of data is also called temporal data, since the time of recording for each measurement can be used as reference to the specific data. However, the time series properties are mostly evident in the measurement values. Metadata, does not exhibit these time series properties. They need to be stored once and do not get updated frequently.

3.1.2. Metadata

This category of sensor data provides additional information on the actual raw measurements. For example, metadata could possibly include the location of the sensor, the measurement unit used by the particular sensor(s) (e.g. degrees in Celsius or Fahrenheit for temperature sensors), physical grouping of sensors (e.g. with respect to location or object measured) or logical grouping (e.g. a water monitor sensor could be used for measuring the pollution levels in a project, and the water conditions with regard to fish reproduction in another. The same sensor would need to be grouped differently for the two projects).

Metadata about time is also important. For example, the timezone of the location of the sensor or the time during which a measurement is sent (in contrast to the time that the database stored it) are considered metadata. Metadata is a very important component of sensor data, because without it the context of measurement values is lost. Without this context a lot of information is lost, which stops analysts from reaching meaningful conclusions about the object being observed. Utilization of metadata leads to context awareness, which is a hard problem to solve in many occasions.

Metadata usually needs to be stored only once, so it can be characterized as static data (c.f. sec. 3.2.1). For example, the type of the sensor or the measurement unit used by the sensor is very unlikely to change. Even though there could be scenarios that some part of the metadata is changing hence, needs to be treated like raw measurement data. For example, if a sea sensor is floating in the sea it makes sense to also store the location of the sensor along with its readings, otherwise false conclusions might be drawn.

Furthermore, predictions about the future measurement values can also be considered metadata. These predictions are usually calculated based on an algorithm,
3.1 Sensor data dismantling

that takes into account different conditions and entities (e.g. past data, environmental conditions, etc.). These predictions can then be validated against the actual measurements and depending on the deviation, the data assimilation algorithm can be optimized. Therefore, this type of metadata could be characterized as derived metadata.

Sensor fusion

Li in [46] describes sensor fusion as a process that information and data from different types of sensors is combined to achieve more efficient and useful data. Data fusion can be classified in three different approaches, as mentioned in [46]:

- According to the information content before and after integration, the data fusion can be divided to the lossless fusion (no information lost) and the lossy fusion (some information is lost).
- According to the relationship between data fusion and data semantic of application layer, data fusion can be divided into: application-dependent data fusion, application-independent data fusion, and the combination of these two.
- According to fusion operation rank, data fusion is divided into: data-level fusion, feature-level fusion and decision-level fusion. Sensor fusion at the data level allows to overcome some of the inherent limitations of single elements of the ensemble. Fusion at the feature level involves the integration of feature sets corresponding to different sensors. Decision level fusion is generally based on a joint declaration of multiple single source results (or decisions) to achieve an improved classification or event detection\(^1\).

The authors in [46], mention five data fusion methods that are the most commonly used: Fusion method based on weight coefficient; Data fusion method based on parameter estimation; Fusion method based on Kalman filtering; Fusion method based on rough set theory; Fusion methods based on information entropy. The interested reader is prompted to [46] for more information.

One of the advantages of sensor fusion is that more information on the object/entity being monitored are available. For example, if we measure the temperature of an object having information on the local environmental humidity, might help explain certain temperature fluctuations (e.g. temperature went down, because humidity levels decreased, which is due to.. etc.). Furthermore, if the fusion (the pre-processing) is performed on a sensor level, the energy consumed by sensors can be minimized. This is the case because the major cause for loss of energy of sensors, is transmitting data. Consequently, less data transmissions result in better energy utilization.

\(^1\)http://www.capsil.org/capsilwiki/index.php/Sensor_Fusion
3.2. Sensor data properties

Sensor data could be considered a specific subset of data, which as mentioned is composed from the measurement values as time series and metadata. Figure Fig. 3.1 shows the composition of this data.

![Composition of sensor data](image)

**Figure 3.1.: Composition of sensor data**

Each dataset present in Fig. 3.1 has some properties. In order to find an efficient way to store data, first specific properties of the dataset must be understood. Therefore, the properties of each dataset category are discussed further. Some of these properties that are assigned below a specific area (time series or sensor data in this case), might be applicable to more domains. However, this is beyond the scope of this thesis since the thesis concentrates on sensor data.

### 3.2.1. General dataset properties

In this subsection, some general properties of data that also apply to sensor data are presented.
3.2 Sensor data properties

**Static/dynamic:** Static datasets, are datasets that have been completed. By completed datasets we mean for example, in a research experiment when the researcher has enough raw data to test his/her case, no further data is collected. Another example of a static dataset, is the use of a completed dataset for historical data analysis. Such datasets are termed as static. Dynamic datasets on the other hand are continuously updated. In plain words, dynamic datasets can be thought of as monitoring an object “real time”, continuously providing new data.

**Univariate/Multivariate:** Univariate data is the type of data that is only affected by one factor, the data is independent from other values. Multivariate data on the other hand is dependent on a number of different factors.

### 3.2.2. Sensor data properties

Sensor data exhibit some particular characteristics that are not present in all datasets. In this subsection these specific properties of sensor data are pointed out.

**Traceability/provenance:** The history of sensor data needs to be treated with care. This is not always the case, but especially in research oriented uses of sensor data tracing the exact origin of the data is very important. This is called provenance of data. For example, the historical data can be used to trace why a certain prediction has given erroneous results.

**Small size:** Measurement values usually have a small data size. It also depends on the specific type of sensor. For example a frame from a video camera is relatively big, when compared to a numerical reading that is transmitted by other types of sensors (e.g. temperature).

**Structured data:** Sensor measurement data is usually in the form of timestamp - measurement value. This is not a highly structured kind of data (such as trees), but a basic structure does exist that could possibly be exploited. Also the measurement values need to be linked to the respective sensor metadata, otherwise the context of the data is lost, which is very important.

**Write-massive operational dataset:** The majority of operations performed on the database are insertions of new records. Updates and deletes might be needed from time to time (e.g. to correct a human mistake, a wrong sensor reading etc), however the vast majority of operations consists of insertions. The rest of the operations are reads. The amount of reads varies between different use cases.

### 3.2.3. Time series properties

As shown in Fig.3.1, time series is a big part of sensor data. Therefore, in this section properties of time series data are elaborated upon.
Temporal: Time is one of the most important attributes of time series data. The order in which data is stored can have a big impact on the performance of the datastore used to store them. A typical way to query a time series dataset, is by requesting all the values between two different dates or times (e.g. the data of a specific month) or one date until the latest value (e.g. the data of the last hour). Furthermore, time (a timestamp usually) is used in order to distinguish single measurement values. So, time is a key aspect of measurements, which is why this property is present.

Regularity: Regular time series are those that provide a reading on precise set times (e.g. every minute/hour/day etc.). Irregular time series are those that do not have a set interval between subsequent measurements. For example, an actuator that monitors the state of a door does not need to send readings all the time, unless the state changes. This property is also related to the architecture of a sensor, event-based actuators vs monitoring-based sensors.

Insignificance of single measurement values: For sensor data, single measurement values do not give an objective view of the object being monitored. Typically, in order to get the context of the measurement, the previous measurement values are of importance. This is the case, because without a set of values no meaningful conclusions can be reached about the state of the monitored object. It also implies that if a single measurement value is lost, it does not have a huge impact on the final dataset. The lost value could be calculated (e.g. using interpolation).

High frequency: Sensor data usually have a high frequency of sending. This can range from milliseconds, to hours or even more, depending on the particular sensor.

High/low variation: High variation datasets can have high and random variations between subsequent measurement values. Low variation data do not differ that much from the previous measurement values. The change in low variation data is more linear, which is easier to predict. For example, a volcano is expected to fluctuate a lot when volcanic activity is taking place. On the contrary, room temperature is better characterized as low variation data. Of course it depends on the context of the object that is being monitored. For high variation data, it is harder to detect sensor errors (e.g. a sensor that malfunctions and sends wrong readings), while for low variation data it is easier to detect such anomalies.

Type of behaviour: The authors in [60] identify four types of behaviour of time series data.

- Discrete: a set of data values with unconnected data points.
- Continuous: any data with infinite values and connected data points.
3.3 Sensor data usage

- Step-wise constant: any data that changes in constant “steps” (e.g. value can only increase/decrease by 1 each time).
- Event: values for this kind of data may vary between them, and the values also are not evenly distributed.

Some of the properties mentioned do not necessarily fall into sensor data or time series properties. This is the case, because they are overlapping as shown in Fig. 3.1. For example, it is hard to determine on which of the two categories the high/low variation and the structured data properties belong. Therefore, some of the properties would best be characterized as the intersection of sensor and time series data.

3.2.4. Applicability of noSQL databases

Out of the properties just described, it can be concluded that noSQL stores are a good candidate for storing sensor data. Their ability to scale is a “solution” for the write-massive operational dataset property (c.f. sec. 3.2.2). Also, the property of structured data (c.f. sec. 3.2.2) makes noSQL stores a good candidate for this kind of data. This is the case due to the schema capabilities that each category of noSQL databases provides, which are briefly presented in sec. 4.2.1. Moreover, the flexible data schema provided by noSQL, can easily adjust to regular or irregular time series.

However, not all datasets exhibit this kind of properties. In this thesis we are mostly interested with write-massive time series data. This choice was made because of certain limitations that relational databases encounter, as described in chapter 1. noSQL databases should not be applied in all cases, relational databases are a good candidate for datasets that they can handle. noSQL databases are a good candidate for massive datasets, because of their excellent scalability and availability. This holds true for the raw measurements. As mentioned already in sec. 3.1 metadata exhibit different properties than raw measurements. In this thesis we deal with raw measurements. The storage of metadata and the combination of the two is interesting, however left as future work due to the limited time for this thesis. So noSQL databases are a good candidate for write-massive sensor data, which is the main focus of this thesis.

3.3. Sensor data usage

In this section some common usage patterns of sensor data are presented. Esling and Agon in [40] mention the most common techniques for querying and analyzing time-series.

Query by Content: One of the most common encountered time series uses. The basic idea is retrieving a set of solutions that are similar to a query provided by the end-user.
Clustering: The process of finding natural groups, called clusters, in a dataset. The objective of this type of usage is finding the most homogeneous clusters, that are as distinct as possible from other clusters. One of the hardest tasks in this type of analysis is defining the “correct” number of clusters, that will result in homogeneous clusters that are distinct from each other.

Classification: The goal is to attach labels to each series of a set. The main difference compared to clustering, is that the classes are known in advance and the algorithm is trained on a sample dataset. The key to this type of analysis is discovering distinctive features, that distinguish classes from each other.

Segmentation: Aims at creating an accurate approximation of time series, by reducing the dimensionality of the time series, while retaining the essential features.

Prediction: Time series usually have a low variance (c.f. sec.3.2.3). Prediction algorithms try to forecast the future values of a time series. This is easier for low variance time series and increasingly harder, depending on the variance of the time series.

Anomaly Detection: Seeks to find abnormal subsequences in a time series. This is easier to perform for data that exhibit a pattern in their values and harder for random data, that do not exhibit a pattern.

Motif Discovery: This technique tries to find patterns of values in large datasets. It could be termed somehow as the “opposite” of anomaly detection.

Esling and Agon in [40], give a thorough overview of different techniques within each category. The interested reader is prompted for further details to [40].

A different categorization of uses is provided by Corsello in [35]. The different use cases for time series data are categorized according to the access pattern used. The categories of access patterns are:

Random extraction: In this scenario a user requests data according to varying criteria that the user formulates at the access time (not planned or expected at data collection time).

Temporal extraction: In this scenario the user requests all of the data between two given dates.

Spatial extraction: In this case the user requests the data according to a logical or physical grouping of sensors. For example, fetching the results for a particular time from a specific geographic area (e.g. a region or a city, depending on the scenario).

Complete delivery: This is not an extraction, since the whole dataset is delivered. However this type of query might not be possible, depending on the amount of data.

Combinations of these categories of extraction patterns are also possible. For example, fetching a time range (e.g. last two days) of a particular geographic area (e.g.
a city or a smaller area) that is being monitored. These are general categories for categorizing the sensor data usage examples. Specific examples from different areas are presented hereafter.

### 3.4. Business cases

Some examples of business cases that rely on sensor data are presented. It further denotes the need for efficiently storing and processing sensor data.

Progressive\(^2\) is a device that is installed in the car and measures the time of day that trips are conducted, number of miles/kilometers driven and the number of “hard brakes”. This data is analyzed and is used to determine whether a driver is driving safely or not. This information is useful to car insurance companies that can adjust the fees, depending on the driver’s behavior on the road.

Another vehicle example utilizing sensors is Google’s self-driving car\(^3\). It utilizes a plethora of sensors (e.g. Lidar, GPS, radars, wheel encoders and more) in order to recognize its local environment and drive accordingly. The data generated by such a car can be utilized for a number of reasons, such as tracing who was responsible for an accident, re-route cars according to current traffic situation, automatically make way for emergency vehicles (e.g. ambulances, firetrucks, etc) and more.

NinjaBlocks\(^4\) is a device/platform that can be used to automate certain tasks within the household. It has a temperature sensor and a motion detection sensor, which can be configured to perform certain tasks. For example it can notify the user if motion is detected, while the user is not home or if a window/door was breached among other possible uses.

Sensors can also be used to monitor building integrity. SmartStructures\(^5\) is such an example, a box device with some sensors is planted within a structure when it is built. It provides information on the quality of the concrete during concrete curing, transport and installation. Its operation continues after the completion of the building, providing information on the integrity of the building. For example it can be used on a bridge to monitor if there are any big cracks and fix them in time, avoiding possible disasters.

Sight Machine\(^6\) is a quality assurance system, that can connect with any camera and it automatically checks the quality of the manufactured product. It analyzes everything machine vision tracks, including: presence, distance, color, shape, size

---

\(^2\)http://www.progressive.com/ [online accessed 26/04/2013]
\(^3\)http://www.google.com/about/jobs/lifeatgoogle/self-driving-car-test-steve-mahan.html [online accessed 26/04/2013]
\(^4\)http://ninjablocks.com/ [online accessed 26/04/2013]
\(^5\)http://www.smart-structures-inc.com/ [online accessed 26/04/2013]
\(^6\)http://sightmachine.com/ [online accessed 26/04/2013]
Chapter 3 Sensor data characteristics

and motion. Tests are flexible and can be created to suit a particular organizations needs.

Preventive maintenance is another area where sensors provide excellent support. For example Rolls Royce\(^7\) the engine manufacturer, provides an engine health monitoring unit that monitors the health of engines (airplane, ship, helicopter engines, etc). This data is stored in a data warehouse where engineers analyze the data, determine if there is a problem that needs to be fixed in the engine and send out a team to repair the engine if there is indeed a problem. This can bear great savings for the companies, but also for their customers (less unexpected waiting time for airplanes for example).

For natural disasters (flood & surge management and forest fire management), Sem-SorGrid4Env\(^8\) is a project that aims to deploy a service-oriented architecture and middleware, that allows application developers to create sensor network applications for environmental management. By monitoring these environmental conditions, natural disasters could be avoided/mitigated by taking preventive actions.

Last but not least, sensors are installed on aircrafts for various reasons. Examples of the uses of sensors on airplanes include surveillance, preventive maintenance, tracking of airplanes, border patrols and more. Sensors on aerial vehicles are particularly effective for military purposes. For example tiny helicopters are used by the military to spy the battlefield before advancing\(^9\). Traditionally this was done by soldiers, but with these tiny helicopters, the troops no longer need to risk their lives in recon missions. Besides military operations, such vehicles can be used by researchers in order to observe locations that are hard to access (e.g. Antarctica). This way further insight about the local environment and wildlife can be acquired, in otherwise inaccessible environments.

Internet of Things (IoT)

The Internet of Things (IoT)[13] is a relatively new concept that has drawn research attention. The concept is still vague, since it is not yet fully realized. It is about machine-to-machine communication (M2M), where all the devices will be interconnected and able to exchange information. The advantages that such an approach could bring are tremendous. For this sensors will play a crucial role. In 2010 a workshop on the IoT-Architecture[55] was held, where specialists from different industries were asked about how they envision the future IoT. Their input is briefly described here. This is differentiated from the previous business cases because they are still only a vision of the domain experts.

\(^7\)http://www.rolls-royce.com/ [online accessed 26/04/2013]

\(^8\)http://www.semsorgrid4env.eu/ [online accessed 26/04/2013]

For the healthcare domain, smart medical devices (e.g. tagged insulin pumps, pacemakers, artificial joints, etc) that can report changes in their status or the state of environmental conditions (e.g. temperature, humidity, etc) are a possibility. Another possible use for the healthcare domain is to provide a platform that allows monitoring of the location of drugs. If the drug appears in an area where it is not supposed to be, the pharmaceutical or the authorities could be notified, so they can act accordingly.

Service and technology integrators, are interested in the possibility of a network that will interconnect all devices and enable communication between them. New types of services will also be enabled by the IoT, such as smart metering, personal devices to car inter-connection as well as home devices inter-connection and home remote control.

For the logistics domain, it is already possible to track each single cargo at all times. What they would like to see, is more efficient energy consumption by the sensors tracking the parcels/boxes. As in the healthcare domain, they also would like to see the boxes carrying the cargo to be more environment aware (e.g. monitor temperature/pressure to ensure the quality of the cargo).

For the retail domain, they would like to see more agility in the possibilities for making payments. For example, enabling the customer to pay using his/her mobile phone in an easy and safe manner or automatic supply of raw materials using the Radio Frequency IDentification (RFID) technology are some of the possibilities described by this expert.

The IoT can also benefit the automotive domain. Applications that improve the mobility with the help of vehicle diagnostics is an option. Furthermore lifecycle services for the vehicles will become more common. The safety of the electric vehicles will be improved, due to the continuous sensing and preventive maintenance. Another interesting possibility is the integration of smart devices with the car.

In the telecom domain, they envision the IoT as a future where the telecom operators will be able to provide a “marketplace” for applications and services. Third parties will be able to utilize this marketplace in order to provide services and applications. However they are concerned with the security and privacy issues of IoT. Specifically, there should be a unique identification scheme for the IoT resources (devices) and their users.

With respect to the veterinary domain, traceability of the production of meat would be advantageous. This will provide assurance about the health of the consumers and also the quality of the meat. Furthermore, automation for the operational monitoring in animal waste management will provide cost reductions. Besides animal waste management, crop monitoring can also benefit from a similar approach.
4. State of the art

In this section an overview of noSQL databases is shown. As mentioned already in sec. 3.2.4, noSQL databases are a good candidate for massive sensor data, which is why we further explore them. In sec. 4.1 two database theorems are presented and in sec. 4.2 an overview of noSQL database taxonomies is provided.

4.1. Database theorems

Modern distributed databases are complex software that offer a plethora of functionalities. These functions are not always compatible with each other though, decisions about the tradeoffs have to be made. Two main database theories about these tradeoffs exist: CAP and ACID & BASE. These theorems are given in the following sections.

4.1.1. CAP

The Consistency, Availability, Partition tolerance (CAP) theorem demonstrates a proof that there is a fundamental tradeoff between these three properties. Gilbert & Lynch in [41] define the three properties as:

- **Consistency**: “The property that each server returns the right response to each request, that is, a response that is appropriate to the desired service specification. The exact meaning of consistency depends on the type of service”[41].
  The authors further elaborate on the different types of services and the consistency each type needs, but they are out of the focus of this thesis.

- **Availability**: “The property that each request eventually receives a response. A fast response is clearly preferable to a slow response, but in the context of the theorem, even requiring an eventual response is sufficient to create problems”[41].

- **Partition tolerance**: “Unlike the other two requirements, partition tolerance is really a statement about the underlying system rather than the service itself: communication among the servers is unreliable, and the servers can be partitioned into multiple groups that cannot communicate with one another. We model a partition-prone system as one that is subject to faulty communication: messages can be delayed and sometimes lost forever”[41].
The basis of the theorem states that at any point only two out of three properties can be adopted by a system. Fig. 4.1 shows this idea and the possible combinations. NoSQL databases are heavily influenced by this theorem, making different solutions try to aim for different combinations. No single combination is the solution to all problems. On the contrary, each specialized solution should be applied where it best fits to the needs of the particular use case. Different NoSQL databases provide different combinations of CAP. A lot of the NoSQL databases favor A and P over C, even though this is not always the case. For example, most graph databases support strong consistency (c.f. sec. A.1). Relational databases usually support C and either A or P, since the majority of relational databases provides strong consistency. Partition tolerance is usually the weak spot of relational databases.

However, the two out of three (properties) choice for the CAP theorem can be misleading sometimes. As Brewer argues in [30], since partitions are not that frequent there is no need to entirely forfeit C or A while the system is not partitioned. Furthermore, this decision between C and A can occur many times within a system. Different choices can be made for subsystems, depending on the needs of the particular subsystem. Partitions have nuances, including disagreement about whether a partition exists or not. Partitions do not occur often, therefore CAP should allow C and A most of the time, but when partitions occur a strategy is needed. This strategy should have three steps according to Brewer: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.
4.2. ACID & BASE

ACID (Atomicity, Consistency, Isolation, Durability) and BASE (Basically Available, Soft state, Eventually consistent) are two quite different approaches for database design. Pritchett in [57] defines ACID as:

**Atomicity:** All of the operations in the transaction will complete, or none will.

**Consistency:** The database will be in a consistent state when the transaction begins and ends.

**Isolation:** The transaction will behave as if it is the only operation being performed upon the database.

**Durability:** Upon completion of the transaction, the operation will not be reversed. These features are most of the times present in relational databases. The author in [57] describes BASE as: “BASE is diametrically opposed to ACID. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this sounds impossible to cope with, in reality it is quite manageable and leads to levels of scalability that cannot be obtained with ACID”.

Most noSQL stores adopt the BASE approach, which favors availability, scalability and high performance over consistency. Another advantage of noSQL stores is that they usually provide an easy way to scale horizontally. Horizontal scaling means adding new machines to a cluster instead of upgrading existing machines. This is a preferable upgrade from a cost perspective, since upgrading an existing machine is more costly than purchasing another commodity machine. Moreover, machines can also be removed in the case that total workload is reduced for example. The easy addition and removal of machines is called elasticity.

4.2. noSQL taxonomy overview

Due to the variability of features and possibilities provided by noSQL databases, no single thorough taxonomy has been widely recognized by the community. In this subsection existing taxonomies for noSQL databases are presented. The amount of taxonomies shows the variability that exists in noSQL databases. This overview will provide a context on what type of databases we believe the data schemas are applicable to. First, in sec.4.2.1 we present the taxonomy used in this thesis along with a brief description of each category. Next, in sec.4.2.2 a brief overview of noSQL taxonomies found in papers and Internet sources is given.

4.2.1. General taxonomy

The suggestions given later in this thesis, are with this noSQL taxonomy in mind. We use the four most commonly encountered categories for noSQL databases. Other
general taxonomies can also be “mapped” to this taxonomy, since similar concepts are used. We believe that the most commonly used taxonomy will be similar to this (because it is quite general). A brief description of each database category along with some representative databases for that category is given hereafter.

**Key value stores:** This category of databases provides the simplest schema possibilities. Usually a grouping (buckets, collections, etc.) is provided, which is similar to a database table. Within this “table” key-value pairs are stored. They are indexed for retrieval by keys. This type of database can be used both for structured and unstructured data. Some of the most used key-value stores are: Project Voldemort[23], Riak[25], Redis[24] and BerkleyDB[4].

**Document-oriented stores:** This type of databases offers rich data modeling capabilities. A database is provided (which is similar to the relational term database), within which collections of documents are organized together. In each collection documents are stored. Documents provide capabilities similar to that of objects in programming languages. Users can add any kind fields to a document, to create a custom structure that fits their needs. This makes it a very attractive option from a perspective of implementation, as this model feels more natural from an object oriented programming language. Some of the most prominent document stores are: MongoDB[17] and CouchDB[7].

**Column-oriented stores:** This group of noSQL databases features a more structured data model compared to the key value category. It provides a database and a “table”, and within each table wide rows are stored, that each contains multiple key-value pairs. So it provides another grouping within each table from a data schema perspective. We also perform some tests with a wide column database, the data schema possibilities are described in detail in sec. 6.3.1. Some representative databases from this category are: HBase[3], Cassandra[1] and Hypertable[12].

**Graph stores:** This type of noSQL databases store data as a graph which is a generic data structure, capable of representing different complex data. A graph consists of nodes which have properties. These nodes are organized/related by relationships, which in turn may also have properties. Relationships and nodes are used to represent and store information. This type of databases is good for datasets that have complex relationships connecting data items (e.g. social networks). It can be referred to as “connected data” database. Some popular databases from this group are: Neo4j[19], OrientDB[22], HypergraphDB[11] and Titan[26].

This categorization is also used in [45], even though it is not the purpose of the authors to categorize noSQL databases, they also go along with the herein above taxonomy.
4.2.2. Specific taxonomies

Even though alternative data models to relational data models have been used in the past (e.g., object and XML data models), existing classifications fail to encompass the new products arising in the database domain. This happens mainly due to the variability of features provided by different database solutions. To give the reader an overview of noSQL taxonomies a short overview of existing classifications is presented.

For each taxonomy, only the categories and subcategories are given. The original authors also present sample databases for each category to give the reader a better overview. However, this is omitted in this thesis in order to save space. The interested reader is prompted to the respective sources. The order in which these taxonomies are shown is random.

(I) Cattell in [31] conducts a survey among the most popular noSQL databases. Even though it is not the purpose of the author to categorize noSQL databases in a taxonomy, a taxonomy is mentioned. The categories for this taxonomy are the following:

- Graph databases
- Object oriented databases
- Distributed object oriented databases
- Key value stores
- Document stores
- Extensible record / wide column stores

The only difference between this taxonomy and the general taxonomy mentioned in sec. 4.2.1 are the object oriented and distributed object oriented databases. Object oriented databases have a schema similar to document databases, with some subtle differences. Objects used in programming languages can be directly stored in the database, which also includes inheritance relations. Also the notion of a class is present, many instances (objects) of a particular class can be instantiated. Further differences between document and object oriented databases is beyond the scope of this thesis. The author in [31] mentions Versant[27] as an example of object oriented database and GemFire[9] as an example of distributed object oriented database.

(II) Tudorica & Bucur in [64] perform a comparison between different noSQL products. Two different taxonomies are mentioned, whose origins can be traced on the Internet. The first taxonomy[39] divides noSQL stores in core and soft and is as follows:

- Core noSQL systems
  - Wide column stores / column families
Chapter 4  

State of the art

- Document stores
- Key value / tuple stores
- Multimodel databases
- Graph databases

• Soft noSQL systems
  - Object databases
  - Grid & cloud databases
  - XML databases
  - Multidimensional databases
  - Multivalue databases

The author in [39] differentiates between core and soft noSQL systems by their use. For core noSQL systems the author mentions that they were created as components for web 2.0 services. Soft noSQL systems on the other hand are not related to web 2.0 services, but share some common features with the rest of the noSQL databases. The core noSQL systems category is very similar to the general noSQL taxonomy mentioned in sec. 4.2.1. The difference is on multimodel databases, which is a hybrid of two other categories. For example OrientDB (c.f. ??) is a hybrid of document and graph databases, trying to bring the advantages from both categories. Last, the multivalue databases in soft noSQL systems is another category that was not encountered before. This type of databases is quite similar to traditional SQL databases. The main difference is that instead of allowing only single values for each field, lists of values can be assigned to fields.

(III) The authors in [64], give a second taxonomy that is cited to a wiki page[20] by an unknown author. It divides the databases in eight categories which are the following:

• Document stores
• Graph databases
• Key value stores
  - Eventually consistent key value stores
  - Hierarchical key value store
  - Hosted services
  - Key value cache in RAM
  - Key value stores on solid state or rotating disk
  - Ordered key value stores
4.2 noSQL taxonomy overview

- Multivalue databases
- Object databases
- RDF databases
- Tabular
- Tuple stores

This taxonomy is quite similar to taxonomy (I), the main difference is that it further elaborates on subcategories of key value stores. This is not surprising, since functionality provided between different key value stores can vary greatly, which is shown by the six different subcategories for the key value stores. The tabular category is a different name for the extensible record stores category used in (I), since Google’s BigTable and Apache HBase are mentioned in this category. In the RDF databases category only one database solution is presented, Meronomy SPARQL Database Server[15]. This could be a new category for noSQL databases, as this product is quite new it is planned to go live later in 2013.

(IV) Strauch in [63] presents an overview of noSQL databases. An overview of taxonomies that the author found on the web are given. They categorize noSQL solutions according to the data model that they provide.

First taxonomy in [63]:
- Key value cache
- Key value store
- Eventually consistent key value store
- Ordered key value store
- Data structures server
- Tuple store
- Object database
- Document store
- Wide columnar store

Second taxonomy in [63]:
- Distributed hash table, key value data stores
- Entity attribute value datastores
- Amazon platform
- Azure services platform
- RDF and semantic data stores
- Document stores, column stores
- SQL/XML databases
- In-Memory databases, cache

The first taxonomy shown in (IV) also further categorizes key value stores. Besides that, the data structures server category is mentioned which is not present in the rest of the taxonomies. This taxonomy was shown in a presentation[67] by Yen. It would be interesting to see how the data structures server differentiates to object databases, however since this was a presentation no argumentation was given behind this taxonomy.

The second taxonomy in (IV) can be traced in the Internet[54]. The author provides an overview of database categories with regard to their applicability for cloud
environments. That is why he also includes Amazon platform and Azure services platform as separate categories.

In conclusion for the taxonomy overview, it can be noticed that there is some overlapping between the different taxonomies. It is hard for the community to settle to a single taxonomy to be used since this is a new field and opinions are conflicting. This is especially true for the key value category, as the term key value is too general, many subtypes can be defined. One can easily come to this conclusion since this is the category that most of the presented taxonomies differ. Furthermore, as noted the features provided by each database vary between different releases of a product. Lastly, even new categories can arise, such as the RDF databases mentioned in (III) and (IV). More taxonomies exist on the web (blogs, wikis, etc) however, it is not the purpose of this thesis to fully classify noSQL solutions.
5. Overview of time series data schemas for noSQL stores

In this chapter we present data schemas used by users and experts in the field. The data schemas are optimized for the storage of time series data using noSQL databases. The information does not only come out of published papers, but also from resources in the internet. noSQL databases are an emerging technology, and not that many papers with regard to the data schema for noSQL exist at this point. A classification of data schemas for noSQL databases needs to be formed to give a general idea on how other organizations are storing their time series using noSQL. This classification will give us insight on how to proceed with the storage of sensor data for the tests we perform in this thesis.

The data schemas and suggestions presented here were taken from sources about specific databases (mostly Cassandra, HBase and MongoDB), however these concepts could also be applicable in noSQL databases in general. Differences might exist between the databases, but the general idea at least should be present for each category of databases (e.g. column oriented, document databases, etc.). In sec. 5.1 some data schemas for time series data using column-oriented databases are presented. Next, in sec. 5.2 data schemas for document databases are given. Finally, in sec. 5.3 the next steps for this thesis are discussed.

5.1. Column-oriented noSQL databases

In this section some general guidelines for data schema design on column oriented databases (e.g. Cassandra, HBase) are presented. The data schema features for each database might differ, but the concepts presented here are applicable to other column oriented databases. In sec. 5.1.1 a brief overview of the data schema possibilities of this type of databases is given. Next in sec. 5.1.2 and sec. 5.1.3 some general suggestions and examples on data schemas for time series are shown.

5.1.1. Column-oriented databases data model

Fig. 5.1 shows the usual elements found in different column oriented databases. Fig. 5.1 is only presented to give a brief overview, the data schema elements are elaborated in sec. 6.3.1 for our tests. We assume that similar capabilities schema-wise
are provided by other column oriented databases also. The data schemas described in this subsection are applicable to such a database.

As shown in Fig. 5.1, the outermost grouping is a database followed by column families, which are similar to relational tables. Within this column families multiple key/value pairs are stored in each row. Each record is identifiable from the combination of column family, row key and column key. As mentioned this is further elaborated in sec. 6.3.1.

### 5.1.2 General guidelines

M. Dennis in [38] gives a presentation on how to model time series in Cassandra, which should also be applicable to other column oriented databases. He proposes to use location-time combinations for the row key (e.g. Groningen001:01/02/2013), for the column name the precise timestamp at which the measurement was recorded and for the value a serialized version of the value (e.g. JSON, XML, etc). Furthermore, he advises to “bucket” data together by time. By this the author means to aggregate many measurements together in a single row. This way multiple disk seeks are avoided. Bucketing also reduces compaction overhead, since old rows do not have to be merged again (assuming that no updates are done to the data, which is a “property” of time series in general see sec. 3.2.3).

The size of the buckets depends on the average range of data queried (e.g. 1 hour, 1 days worth of data), the average measurement size, the frequency of measure-
ments and the Input/Output capacity of the nodes. The author also provides some guidelines for picking a correct bucket size, which are:

- Each bucket should not be bigger than a few gigabytes per row.
- The bucket size should be greater or equal to the average range query for the particular use case.
- The number of data points per row should not exceed 10 million.

If the latest data is requested often (e.g. dashboards), it should be considered reversing the order in which data is stored, by setting the comparator in data retrieval request to descending mode. For use cases where we only need the number of events in a given time interval, a different row or column family can be used that has the bucket name as its column name and a counter as its value. This type of column families are called counter buckets. The counter needs to be incremented with each new insert on the respective bucket though.

### 5.1.3. Data schemas for column-oriented databases

In this subsection we present some data schema references that we found mostly on the Internet, about how organizations/people are storing time series and related data.

**Simplest approach**

A very simplistic data schema as mentioned in [58], is storing the object monitored as the row key, a timestamp of when the measurement was received as the column key and the actual measurement as the respective value. This way we can easily query for single values at a specific timestamp or ranges of values within the same row. This works fine for time series with a low frequency (e.g. one measurement per day), however with high frequency time series data the size of the row would quickly get too large. This is problematic because if the size of a row gets too large it will be too large to fit into memory. The downside of this is that read requests will always have to fetch the data from disk, which will reduce performance.

An almost identical approach is also presented in [49], for time series with a low frequency. The author suggests grouping each source of data in its own row and then simply appending the data in its respective row. The data include the timestamp (date and time) as the column name and the respective value as the actual value for that row.

A solution to this problem is sharding/grouping (c.f. sec.A.2) the data in a way, starting a new row for each interval (in the grouping). The most straightforward solution is to group them on a per day basis or some other static time interval. To accomplish this, the starting time stamp is appended to the row key (e.g. sensor1-123456789). By appending timestamps in this way, we are able to determine in
which row(s) the particular query spans. To query values over multiple rows a multi-get can be used. The term multi-get query refers to a client method, which allows the user to fetch multiple rows with one call. Usually this function is more efficient than issuing multiple single read requests, by eliminating the overhead from multiple requests.

The author in [58] suggests that the size of each row should not get much larger than 10MB per row. However, these numbers can vary between applications. It is dependent on the type and requirements of each application and also on the queries that are required by the particular application. The author proposes a formula for calculating a good sharding interval, which is:

\[
\text{shardSizeInSeconds/updateFrequency*averageDataSizeInBytes} = \text{rowSizeInBytes}
\]

**Metadata utilization**

The author in [44] proposes two data schemas to be used with column oriented databases. The first is to use a column family with column values that point to other row keys, in a different column family that stores the actual raw measurements. The first column family that points to the actual measurements holds the metadata. This approach is similar to indexing. In order to query values using this strategy we first get the row keys that are relevant from the metadata column family and then perform a multi-get request on the respective rows. This approach is more “normalized”, it allows for easy updates of events, does not require repetition of data between multiple column families and allows us to add built-in secondary indexes.

On the other hand, the data fetching process is relatively slow. An additional read needs to be done for each read request, one to the metadata column family and one on the raw measurements column family. Due to this extra read, this approach does not scale very well (due to the extra read). If this pattern (additional read for each read request) can be avoided, it is highly advisable to avoid it. However, the number of raw measurements is huge, which makes it hard to create materialized views, as the author in [44] suggests in the following subsection

**Materialized view**

The second data schema proposed by the author in [44], is storing the complete set of data for each event in the entire row. This should be adapted to each use case, depending on the queries of the particular use case. This is like keeping a materialized view. This provides much more efficient reads, since retrieving a range requires reading only a contiguous portion of a row. With this approach some denormalization occurs more often (e.g. if an event is tracked in multiple rows or if we store the data multiple times for different purposes). The author mentions that this is the preferred approach except for the case that the size of each event is too large.
This is not very applicable to the storage of raw measurements, but it is a perfect candidate for particular uses within a use case. For example the data accessed by a dashboard, can be stored following the materialized view approach. Repetition of data is acceptable in terms of write throughput, since column oriented databases usually have a high write throughput. Furthermore, the consistency of data should not be a problem with time series data, since the updates of existing data are few to none.

**Reduced disk space**

The author in [65] proposes a way to reduce the disk space used by Cassandra. Even though disk space is a cheap commodity, if the time series stored are massive the size of the database can easily get out of hand. So to tackle this problem, the author proposes creating a second backup column family for the data. In this backup column family the data will be grouped together. So for example, instead of using the standard one timestamp-one value setup, one timestamp would group multiple values. For grouping multiple values together the author proposes using a byte vector. By a byte vector the author means converting all values to 64bit integers, concatenating them together and writing them as a single value. Instead of a byte vector, a byte array is also a good option for serializing the data. It provides a more standardized way to serialize and deserialize the data.

**Short/Medium-term analysis**

The author in [49] suggests using a Time To Live (TTL) for the data. This is only applicable in cases where historical data is not needed though, since when the TTL expires the data is lost. He does not mention a way to somehow backup the data. This could be a good option for cases where we are not interested in keeping the data for historical analysis, but rather focus on a more short to medium term analysis of the data. The TTL removes the burden of managing what data to delete (because they are outdated). However, caution is advised since in many cases if the database has been marked for deletion after a certain period, it cannot be undone which could lead to loss of data.

**Storing financial time series data at BlueMountain Capital**

The authors in [48] give a presentation on how they store financial time series data at BlueMountain Capital\(^1\) using Cassandra. They present the data schema that is used for the storage of the time series data. They have two main queries that they are serving. One is for a range of data. The user passes as parameters a start time, end time and the periodicity of the measurements. The second type of query is

---

\(^1\)https://www.bluemountaincapital.com/ [online accessed 02/09/2013]
getting all measurements that were stored on a particular point in time (e.g. all values stored at 11:00). Fig. 5.2 shows a diagram of the data schema used to store the data.

![Data Schema Diagram](image)

**Figure 5.2.:** Visual view of the data schema of BlueMountain Capital

They provide an API for storage as a service interface to their clients. For each query issued by the client, they request all the data points that are included in the start-end times that they receive and filter out the data that is only needed on the application level. At the time of giving the presentation, they considered switching from this paradigm (doing the filtering on application level), to push the filtering to Cassandra using pushdown filters. Pushdown filters include: downsampling on writes, where the coordinator will be able to do the filtering locally with its local cache, instead of pushing it to the service to perform it. This way a round-trip between the database and the service is avoided.

**Time series data at eBay**

J. Patel in[56] has given a presentation about the different use cases where they use Cassandra at eBay. They use it for a number of cases which include: social signals on eBay product and item pages and hunch taste diagram for eBay users and items. The use cases that are related to time series are: mobile notification logging and tracking, tracking for fraud detection, SOA request/response payload logging and RedLaser server logs and analytics. The time series related cases are described next.

Their general idea for the storage of time series data, is storing the data multiple times to serve commonly encountered requests with one request to the database. They are using one column family to store the raw events. The row key for this column family is a composite key of `event_type<date format>`, for a column key they use the `timeUUID` at which the event happened and as a value the payload (or whatever is being measured). The rest of the column families store aggregate data. So in this particular example the author presents 3 different column families: rollups-minute, rollups-hour and rollups-day. These 3 column families are counter column families that are incremented along with each raw measurement stored on the column family storing the actual data. Fig. 5.3 shows an overview of all the column families that eBay uses Cassandra for.
For each raw event received 3 additional write operations are issued to the database to update the aggregation tables. This might sound like extra overhead, however Cassandra is really good at writes and this results in additional benefits when it comes to reads. Dashboards that provide statistics have to query only the aggregation column families. Furthermore, by aggregating results like this all the information that is needed for a particular dashboard can be read in a single read operation, which is very efficient (for reads). For the storage of raw events similar data schemas can be found in other use cases also. Aggregating the data to a row according to a time span is a very common practice, since it then results in efficient read requests.

Patterns for data modeling from Datastax

The author in [50] gives a presentation on real world use cases for Cassandra. The presentation focuses on different data schemas that are being used in different organizations. The author is a Principal Solution Architect at DataStax and has advised many times organizations on the data schema used. So the use cases presented here are not a one time thing, but rather can be considered as “patterns”, since very similar data schemas are being used in many organizations as the author mentions.

The first data model is a model for an online shopping cart. The requirements for it are for each user to be able to have multiple shopping carts. For this the data is denormalized, which leads to one call to the database for each shopping cart. This also provides row isolation, which guarantees consistent data. Row isolation means that each write request, uses its own row (since each user has his own row) which leads to consistent data. Fig.5.4 shows a visual representation of the two tables that the author suggests for this particular use case.

So for this use case we create a column family for the users, which holds the data for that particular user. Additionally, it also holds a field of type <set> that will
hold the ids of the shopping carts for that user. The other column family holds the actual data for each shopping cart. For the partition key (row key) of this table the username and cart name are used. The item_id itself has a secondary index defined on it. Furthermore, he also suggests to further denormalize data. For example, he mentions that for each item that the user wants to purchase, a <map> (with related products for example) should be used. This way all of the data which correspond to one item can be read in 1 database operation, which results in very efficient reads.

As a second example the author provides an example of a user activity tracking example. The author advises to feel free to denormalize data and write the data multiple times. In this example he used one table that is suitable for very fast real time activity (e.g. making a suggestion to the buyer once he decides to buy something). For the second table (the one holding historical data), he uses as a partition row key the username along with the date that the event happened (on the previous table it was time of day, this is a higher level grouping). Fig. 5.5 shows a visual representation of the two tables that the author suggests for this particular use case.

On the user activity table (the table for the fast decisions), the author suggests using a reverse order (descending) so that it is really fast to know the latest purchase/move made by the consumer. For the analysis of the second table the idea is to use
### 5.1 Column-oriented noSQL databases

Users activity

<table>
<thead>
<tr>
<th>Row key: Username (guest123)</th>
<th>Interaction time</th>
<th>Activity code</th>
<th>Details</th>
<th>...</th>
<th>Interaction time</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0D14-D202-...</td>
<td>0008</td>
<td>Normal login</td>
<td>...</td>
<td>0E19-D236-...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key: Username (jack19)</th>
<th>Interaction time</th>
<th>Activity code</th>
<th>Details</th>
<th>...</th>
<th>Interaction time</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0D13-D605-...</td>
<td>0153</td>
<td>Failed login</td>
<td>...</td>
<td>0F16-D242-...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

User activity history

<table>
<thead>
<tr>
<th>Row key: Username</th>
<th>Interaction time</th>
<th>Activity code</th>
<th>Details</th>
<th>...</th>
<th>Interaction time</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest123</td>
<td>0D13-D605-...</td>
<td>0008</td>
<td>Normal login</td>
<td>...</td>
<td>0D13-D805-...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key: Username</th>
<th>Interaction time</th>
<th>Activity code</th>
<th>Details</th>
<th>...</th>
<th>Interaction time</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>jack19</td>
<td>0D13-D123-...</td>
<td>0008</td>
<td>Normal login</td>
<td>...</td>
<td>0D13-F605-...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 5.5: User activity data schema](image)

Hadoop\(^2\) to analyze the data. This is a good option, since the data in this table are not required real-time and Hadoop allows flexibility for the data analysis. Other analytical frameworks can also be used (e.g. Apache Pig\(^2\)). This is the difference between the two tables, one is for real time feedback to the client and the second is for long term analysis. For the short term table he also uses a TTL (of 30 days), since after that it is not really required anyway.

As a third example the author provides an example of storage of logs for a login mechanism. As the log data comes in, it is fed to Flume\(^3\), which stores the data in 3 different places depending on the message. He stores the raw logs, the latest success and the latest failure. In this particular use case, three column families are defined: log_lookup, login_success and login_failure. Fig. 5.6 shows a visual representation of the two tables that the author suggests for this particular use case.

For the log_lookup a composite row key of source, date_to_minute is used and the timestamp is part of the primary key (which means it cannot be empty). He also suggests to compress the actual raw data, since it takes very little time and it really saves a lot of bandwidth. For the login success column family, the partition key is the source and the date_to_minute column is part of the primary key. Furthermore, on this column family a counter is defined that counts the number of successful

---

\(^2\)http://pig.apache.org/ [online accessed 08-12-2013]

\(^3\)https://github.com/cloudera/flume [online accessed 02/09/2013]
logins. He also reverses the order of sorting (descending), so that it is more efficient to retrieve the number of successful logins for the last 10 minutes for example. The exact same schema is applied to the logins_failed column family, with the difference that failed logins are recorded in this column family.

As a final example the author provides an example of user forms versioning. For this example he proposes creating one column family that has the following attributes: username, form_id, version_number, locked_by and form_attributes (which is a <map>) and the primary key for this is PRIMARY KEY ((username, form_id), version_number). Fig. 5.7 shows a visual representation of the two tables that the author suggests for this particular use case.

So for this the first time a form is accessed/created the initial data is set up. When a user uses the form we update the locked_by attribute. So if another user tries to access the same form, we simply check if the locked_by attribute is empty or if a username is returned. Whenever the form is submitted, we just increment the
version number of the particular form and set the locked_by attribute to empty.

A three dimensional model for HBase

The authors in [43] present a three dimensional model that also takes time into account for storing time-series data in HBase[3]. HBase provides the possibility to store different versions of data for the same cell. The authors associate the version of each cell, with each subsequent data-element value in the time series. If the time series continues for a long time a maximum period is defined, which essentially is the maximum number of versions stored per cell.

The authors in [43], have tried three different data models for two different use cases: a cosmology dataset and the bixi dataset. The cosmology dataset is produced by an N-body simulation of the universe evolution, where the universe is represented by a set of particles. Particles are points in a 3D space and their evolution is simulated over discrete timestamps. Every few timestamps a snapshot is generated that holds the state of the simulated universe at that timestamp. This dataset consists of 321,065,547 particles from 9 snapshots with a total size of 14GB. The bixi dataset is a public dataset for a bicycle renting service in Montreal. Users can borrow a bike from a station and return it after in any station within the city. The data were collected for 70 days, it contains 96,842 data points for all the Montreal stations and is 12GB in size.

HBase offers a data model similar to Cassandra, however it also has a version attribute for each cell. The cosmology dataset has “snapshot id” (sid), “particle type” (type) and “particle index” (pid) to identify each particle. Furthermore, particles have other properties with corresponding values. Tab. 5.1 shows the three different data schemas used for the cosmology dataset.
Table 5.1.: Data models for the cosmology dataset

<table>
<thead>
<tr>
<th>Data model</th>
<th>Row</th>
<th>Column</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema 1</td>
<td>sid-type-pid</td>
<td>particle properties</td>
<td>no meaning</td>
</tr>
<tr>
<td>Schema 2</td>
<td>type-pid</td>
<td>particle properties</td>
<td>snapshot id</td>
</tr>
<tr>
<td>Schema 3</td>
<td>type-reversed pid</td>
<td>particle properties</td>
<td>snapshot id</td>
</tr>
</tbody>
</table>

The authors mention that for data schemas 1 and 2 of the cosmology dataset they encountered hotspot problems. These problems were resolved by using the 3rd data schema, which distributes the row keys more evenly across the cluster, therefore avoiding the hotspot problem. The authors executed 3 different queries on the cosmology dataset. The first query invokes a range scan in one snapshot, the second query compares the data across two snapshots and the third query retrieves data from multiple snapshots. For each of these three queries, different queries with increasing values are used (each subsequent query involves more data than the previous one).

For the first query, schema 1 outperforms schemas 2 and 3. This is the case because not a lot of computations are involved, since only data from 1 snapshot are retrieved. Schemas 2 and 3 have similar execution times for the first query.

For the second query, schema 1 is only able to complete the initial queries that do not involve a lot of data. After a certain point it is unable to fulfill the queries. Schema 2 and 3 have very similar performance again for most of the queries in this query category. Schema 2 and 3 interchangeably surpass each other. However, for the last two queries in this query schema 2 outperforms schema 3 by a large difference.

For the third query, schema 1 again is only able to perform the first queries that do not involve a lot of data points. Schema 3 outperforms schema 2 for all queries in this query. The difference becomes more evident when the number of data points increases a lot. In the end schema 3 outperforms schema 2 by 2-3 orders of magnitude.

The conclusions they reach is that the 1st schema is better suited for small queries that do not involve a lot of comparisons. However, when the number of computations increases schema 1 fails to execute the query, while the three dimensional models perform the query without any problems. Schema 2 and 3 have similar execution times for most queries. When the calculations involve multiple snapshots to be queried, schema 3 outperforms schema 2 in the third query. However, the opposite happens for the second query (schema 2 outperforms schema 3). It is hard to draw some concrete conclusions on one particular data schema, since schema 2 outperforms schema 3 on some other cases. The only concrete conclusion is with regard to the 2D schema versus the 3D schemas, the 3D schemas are better when many computations are included.

For the bixi dataset the authors perform only one query, which is retrieving the average bike usage for a set of given stations and time for the last 1, 2, 4, 8 and
16 days. The bixi dataset is smaller than the cosmology dataset, however it has a rich history of rentals. Tab. 5.2 shows the different data schemas used for the bixi dataset.

<table>
<thead>
<tr>
<th>Data model</th>
<th>Row</th>
<th>Column</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema 1</td>
<td>hour-sid</td>
<td>minutes</td>
<td>current time</td>
</tr>
<tr>
<td>Schema 2</td>
<td>hour-sid</td>
<td>monitoring properties</td>
<td>minutes [0, 59]</td>
</tr>
<tr>
<td>Schema 3</td>
<td>day-sid</td>
<td>monitoring properties</td>
<td>minutes [0, 1439]</td>
</tr>
</tbody>
</table>

Table 5.2.: Data models for the bixi dataset

The sid again stands for snapshot id, which helps distinguish the different snapshots stored in the database. For this dataset, schema 3 outperforms the other two. Schema 1 outperforms schema 2 for the first two queries (1, 2 and 4 days), however for the rest of the queries, schema 2 outperforms schema 1 (8 and 16 days).

The clear winner in this scenario is schema 3, but we can see that the 2D schema is better than schema 2 on the smaller queries. Therefore, again for smaller datasets the 2D schema proves to be performant. However, for larger datasets, which is the main target for a distributed database such as HBase, the 3D schemas perform better. This gives insight that a higher level grouping of the data (by using a 3rd dimension in this case), gives better performance for large datasets that involve many retrievals/calculations. For the detailed results of these tests performed in [43], the reader is prompted to read the respective publication, which includes many details that were omitted in this thesis since they are not in the focus for this thesis.

**Storing time series in OpenTSDB**

The author in [61] gives a presentation about Open Time Series Data Base (OpenTSDB) [21], about the decisions they made for storing time series. OpenTSDB was built on top of HBase and its aim is to provide an easy way to store time series data efficiently and performantly. OpenTSDB was designed and implemented at stumbleUpon as their main production monitoring system, which has been used for 2 years at the time of presentation. The author provides some suggestions and some pitfalls to avoid when designing a schema for OpenTSDB.

The first suggestion is to go for wide tables (large rows), instead of tall tables (many rows). Another technique that can improve the performance of OpenTSDB is making writes idempotent and independent. This way if writes are idempotent, failed writes can simply be retried, without any further consequences. Writes being independent is important for the scalability of the database. Another recommendation he gives is to store many raw measurements in each key/value pair. This reduces the disk space used, since HBase internally stores for each single key/value pair the

\[\text{http://www.stumbleupon.com/ [online accessed 26/09/2013]}\]
column family name and the row key. This becomes an important issue when data is expected to be collected for a long period of time, otherwise additional disk space will be required often.

Next, the author gives some suggestions on what to avoid when storing data in OpenTSDB. Using variable length fields for composite row keys is not a good idea, because this makes the performance of row scans slow in HBase. What he suggests instead of storing variable-length names for row keys, is to use a small mapping table that will map these variable-length names to small fixed-length IDs.

The author then goes through the different attempts they had when designing OpenTSDB. After some first attempts, they decided to use a mapping table for the variable-length ID’s with a fixed-length set of ID’s and storing multiple measurements in each row. Having less rows also means that HBase has to scan less rows, which results in more efficient retrievals. So they use a ID-timestamp key for their row key. An interesting scheme that they use for the column keys is storing the differentials of the time with regard to the time stored in the row key. This has as a result that less storage space is required, for example lets assume that the row key timestamp is 1234567890, for the first timestamp instead of storing 1234567891 they store +1.

Furthermore, due to the way HBase works they had some issues with the idempotency of writes. When a client contacts one region server and starts storing measurements, a new row is created where these measurements are stored. If for some reason, that particular node failed the client would contact a different node, however the different node would not have that particular row that the previous node created (because it did not have enough time to replicate the data). Therefore, a new duplicate row would be created leading to inconsistencies. The way they overcome this for OpenTSDB is by defining a predetermined limit for the rows. Initially it was set to 10 minutes and in a later version increased to 60 minutes. This way rows always start on a 10 minute / 1 hour basis. This way by just applying a modulo operation they are able to determine to which row the particular measurement should be stored.

A last optimization they do with regard to disk space and read request performance, is compacting “finished” rows. Once a row has been filled it does not get re-written. Therefore, once a row is full they go back to that row, read all of the measurements, store them in a single byte array and store the byte array as a single key/value pair (they do this both for column keys and values, so 2 byte arrays). This saves a lot of disk space, since the column family and row key do not have to be repeated many times and this also has some affect on the network usage, since less data have to be transmitted (due to the reduced “duplicates” of column family and row key). Another advantage of this compaction is that less objects are stored in the JVM. Normally each key/value pair is stored as an object on the heap memory. By using this compaction scheme, many measurements (1 hour worth of measurements in case of OpenTSDB) are represented as one object. This has a positive effect on the
5.1 Column-oriented noSQL databases

Performance of garbage collection.

**Storing metrics for time series**

The author in [62] gave a presentation on storing time series metrics. His main suggestion is denormalizing the data. For example, in their model they are storing the metrics by second, by minute, by hour and by day. This means that each incoming metric is being stored four times, once for each column family. This way they can fulfill all the views that are required by their particular application. The details are not that important, this concept can be applied in many different situations, depending on the needs of the particular application. However, most applications need some metrics as the example mentioned up to some extent.

Another general suggestion the author makes is that sometimes more rows is preferable over having very wide rows. For example, if a particular node holds data that is often requested, we might encounter hot spot problems for that node. If this occurs, checking the distribution of data or reducing the number of measurements stored per row might help with the hot spot problems. Reducing the number of measurements per row can possibly fix the hot spot problem, because the load will be shared by two or more nodes instead of one.

**Summary**

To summarize all this up, for the storage of time series data using column oriented noSQL databases, the data schema should definitely adhere to some basic rules. The most basic rule is with regard to the size of rows. Column oriented databases are good with wide rows, however if the size of the row becomes too large the requests will become slower. Therefore, the size of each row needs to be managed. Two possibilities to manage this is according to row size, number of measurements stored per row or a combination of the two. With regard to size, the size should not exceed 10 MB by a lot. With regard to measurements it depends on some other variables also, as the formulas presented in the simplest approach subsection in sec. 5.1.3.

Another aspect that is present in almost all data schemas is a higher level grouping of data into rows. In some examples a materialized view for a particular request is stored in one row and in other they use the row to group certain elements together (e.g. 1 days worth of log data). Both approaches are good and one does not exclude the other. As we have seen in many examples data repetition is encouraged and can lead to more efficient reads. Therefore, both of these strategies should be taken into account. An issue to watch out when doing this, is the partition key. For the row keys composite names can be used, while taking into consideration that the size should not be too big. As the author in [61] provides as an example, using a mapping table to map long variable ID’s to shorter fixed length ID’s can boost the performance. Techniques like this can ensure that the row key size is not too big.
Another very useful technique mentioned by the author in [61], is grouping multiple measurements into a single cell. This way less repetition of identifiers (row key and column key) are required and the only downside of this is that the measurements will always be fetched together. The number of measurements can be adjusted per use case to match their needs, but overall this is a very good suggestion.

5.2. Document noSQL databases

In this section some general techniques and guidelines for the storage of time series data in document noSQL datastores are presented. The sources are specific for MongoDB since its development community is very active and provides a lot of information. But the solutions could also be applicable to other document stores similar to MongoDB. In sec. 5.2.1 a brief description of the data schema possibilities provided by document databases is given and in sec. 5.2.2 an overview of data models for document stores is presented.

5.2.1. Document databases data model

The data model provided by document databases was briefly described in sec. 4.2.1. The data schemas provided in this section are specific to MongoDB, so the schema capabilities of MongoDB will be briefly discussed. Similar data schema capabilities should also be present in other document oriented stores, since MongoDB is a popular document database. Fig. 5.8 shows the two different approaches for data modeling in MongoDB.

The data schema capabilities are very similar to that of object oriented programming. Each document can have a custom structure with varying number of fields. Relationships between documents are enforced either by using references (top part of Fig. 5.8) or by embedding sub-documents to documents (bottom part of Fig. 5.8). The approach with references feels more normalized and closer to the traditional relational model. The embedded sub-documents approach is about denormalized data. This denormalization allows applications to retrieve and manipulate related data in a single database operation.

5.2.2. Data schemas for document databases

In this subsection some data modeling example found in literature are briefly discussed. A lot of details are skipped, but the interested reader is prompted to [51] and [34] respectively for more detailed information.
Mo and Wang in [51] present a flexible near real-time high performance and high availability solution with indexes for data, Asynchronous Index Strategy (AIS). This solution is suitable for time series big data stream storage with intensive inserts and quick random seeks.

The application scenario for which the authors tested their solution involves two billion documents per day (~23,000 documents per second) to be analyzed and stored on average. The frequency of data is variable, there is a peak throughput during the day and a trough during the night. The documents are structured including fields like strict ascendant timestamp, content, location, etc. The average size of a document is 100 Bytes.

MongoDB’s sharding architecture uses partition strategy to scale up. Even though
pre-splitting is an effective method in insert-intensive applications in theory, in practice it turns out not to be that effective in inserting a large amount of documents in an indexed collection. It is bounded by disk I/O. Mass inserts cause frequent index updates. With MongoDB’s memory mapped storage engine, frequent page fault saturates disk utilization. The workload is shared between multiple shared-nothing machines, however the cluster is still not efficient enough.

The authors mention what we have seen in other solutions, partitioning by time which is a “natural” solution for massive time series data. They use AIS to solve the index performance bottleneck that they encountered. MongoDB nodes are divided at least into three groups. Each is called a MongoDB unit whose nodes data is replicated synchronously. Within a MongoDB unit, while the data is shared, indexes are built asynchronously. Requests sent to a MongoDB unit is executed by the Read One Write All (ROWA) rule.

Document streams are partitioned by time, the documents are sliced in time slice collections, each assigned to a particular MongoDB unit. All documents that belong to the same time slice will be inserted into the same MongoDB unit without any user defined index. When the time slice ends, document stream is shifted to the next MongoDB unit. Meanwhile the current MongoDB unit creates indexes for that particular time slice. While creating the indexes, the particular MongoDB unit is “tagged” as read-only. Read requests are randomly distributed among nodes that are not creating an index. With regard to the application scenario mentioned, the data stream with a cyclic rate is partitioned into N collections and we assume that there are M MongoDB units. The sizes of the collections are not equal, therefore it is not trivial to distribute the N collections among the M MongoDB units. A simple way to determine N and M according to the authors in [51] is:

$$\text{Greatest common divisor}(N, M) = 1$$

The authors proceed by providing an architecture diagram for AIS as a middle ware software. However, these details are too specific for this general overview of different data schemas that is being performed in this thesis.

The authors perform some experiments to provide an indication of the index and insert performance. Tests are performed with a single node MongoDB instance, a MongoDB sharding cluster and AIS based on MongoDB. The test dataset is a randomly generated document stream. An integer field is chosen to be a shard key, uniformly distributed from 0 to 30. Index is created on a 12 byte length string field and an integer field, which are randomly generated. Total length of a document is 185 bytes on average.

A prerequisite of AIS is that an index should be created for each timestamp. There are two types of index creation: foreground and background. The difference is that foreground indexing locks the whole database, while the background indexing does not lock the whole database.
5.2 Document noSQL databases

For their test, they have used a single node MongoDB instance and they try four different types of indexing: Foreground Integer Index (FII), Foreground String Index (FSI), Background Integer Index (BII) and Background String Index (BSI). Fig. 5.9 shows the diagram the authors provide in [51].

![Index creation performance (single node)](image)

From these results we can immediately see that the performance of indexing on integers is better than the performance of indexing on strings. The background indexing strategy has the best performance out of the four different strategies. An interesting note is that BII surpasses the respective FII, however the opposite takes place for strings, FSI surpasses BSI. So for time series that require indexing using MongoDB, the background indexing method is preferable usually because as mentioned it does not lock the entire database. Furthermore, another suggestion is to use a table to map large string identifiers into integer identifiers, since as we can see the performance of indexing on integers steadily surpasses that of string indexes.

**MongoDB applied design patterns**

The author in [34], presents some common patterns to be used for storing data with MongoDB. A chapter is dedicated to the storage of time series data. The first advice the author gives is to store the data in a “pre-processed” way, which means that if for example we are storing the log information of an Apache web server, the data should not be stored exactly as outputted by the web server. Instead each property of the log data should be tagged. In the case of Apache for example, it could be that we tag the data by id, host, name, path, time and request. This is a way to also refine the data, skipping information that we know that will not be needed. This makes it easier to query items also. If we stored the raw data as they came in, performing a search based on a specific attribute would require using regular
expressions, which results in a full document scan. The preferred approach is to store each property on its respective field in the MongoDB document. Furthermore, using the correct BSON types can be very handy. For example, storing a date as a string makes it hard to actually compare two dates. Even more, using the correct type for each attribute can result in more efficient storage usage.

Another, rather obvious but still useful advice, provided by the author is to use secondary indexes for fields that we know will be queried for a particular value. For example, for the Apache web server a typical field to use a secondary index would be the path. This way we can issue queries that only return entries that have the particular path. This is a common requirement for many applications. However, with indexes one needs to take into account the space that the index takes. Indexes ideally should be kept in memory for fast access. Otherwise if the machine is out of memory and the index is only kept on disk, the added benefit of using an index is lost. The database will have to perform a disk seek/search, which is a costly operation that we want to minimize as much as possible.

Another field that typically needs an index in the case of time series is time. This is very important, since range queries in time series data represent the most common access pattern. Therefore, a secondary index on the time field of time series data is highly recommended. The author mentions that this is a right-aligned index (in case our queries focus on the recent history). Right-aligned refers to the access pattern of a regular index. A lot of the queries for time series request latest data (e.g. dashboards), so most of the index is never actually used. In this case, a small part of the index is ever held in RAM at a particular time, so index size is of much less concern. Overall, right-aligned indexes take up less space and are more performant for finding things like maximum, minimum, etc.

A nice feature of MongoDB is that it provides the `explain()` method, which provides detailed information on how many objects had to be scanned, response times and some other information. Furthermore, MongoDB provides the option to create compound indexes (on two fields or more). Depending on the query that we want to satisfy, compound indexes can be useful. It is important to note that the order in which compound indexes are defined, does have an impact in the performance of the query. Index scans are a lot faster than collection scans, but still one should not overdo it with the use of indexes mainly due to RAM limitations (as mentioned it is preferred if the indexes can be held in memory for fast access). The author provides some guidelines for efficient indexes, which are:

- Any fields that are queried for equality should occur first in the index definition.
- Fields used to sort results of a query should follow the fields for equality.
- Fields queried by range should occur last in the index definition.

Another really good feature that MongoDB provides is the aggregation framework available from version 2.1 onwards. Essentially it is a thin abstraction layer from
5.2 Document noSQL databases

the map/reduce paradigm (e.g. instead of Hadoop). The author in [34] provides a nice example of pipelining. The example can be seen in Fig. 5.10.

![Figure 5.10: Aggregation query on MongoDB][34]

The results are first filtered by only getting the results that match the time provided by our query, with the `$match` command. These results are then passed to the next filter. The `$project` term selects the path field and creates some custom fields (year, month and day of month) which are to be passed to the next filter. The `$group` filter creates a new document for each unique combination of path and date. Within the `$group` function, the `$sum` function of MongoDB is used in order to create the aggregation statistics/results that we need to present to the end user. This pattern that is being used by MongoDB is clearly a pipes and filters design pattern. For optimal results in aggregation queries, an index is advised on the field that is being filtered by the `$match` command.

Another issue that needs attention with time series data and MongoDB is the size of collections. MongoDB does not compress the stored data on disk. As a consequence,
Chapter 5  
Overview of time series data schemas for noSQL stores

after some point with continuous time series the size of the data will get out of hand. A solution to it is using capped collections. These collections have a predefined data size. Once the limit is reached old data start to get discarded. This could be a good solution for use cases that do not need to hold historical data for analytical purposes. Another use case that capped collections could cover is if we are collecting data during day time for example, but are less active during night (we could backup the data during night).

The author in [34] also provides some general guidelines for schema design in MongoDB, which include:

- Individual documents growing too much after creation forces MongoDB to move the documents on disk, which degrades the performance.

- Collection scans impair the performance of queries, therefore they should be avoided as much as possible (by using indexes).

- Documents with hundreds of keys can result in unpredictable query performance on those documents. This is related to the way internal document storage (BSON) stores documents.

Another tip that the author provides is to use with time series data that continuously grow throughout the day. As documents grow, MongoDB will need to relocate the document on disk, which as mentioned slows things down. The solution to this problem is pre-allocating documents with fields holding 0 values before the documents are actually used. If the documents are fully populated during creation, the documents will not grow beyond their initial size, therefore avoiding the costly moving of files on disk.

Another pointer provided by the author is when storing hours of the day for example. If we store each hour sequentially and we need the value for the last minute, MongoDB will have to scan through all of the previous hours to give us the result. This can be mitigated by using a hierarchy for hours. For example instead of storing them sequentially, we could split the hour field into 24 hourly fields. This way we can find the minute we are interested in a lot faster.

Pre-allocating the documents is a good technique to speed up results of short range queries. However, for queries that require historical data (from multiple documents), pre-allocation falls a little short. In such cases it is preferable to store daily aggregates in a higher-level document. The author further elaborates on techniques used to gather aggregated data for statistics/reports. This is done using map/reduce also, not only MongoDB’s aggregation framework. The interested reader is prompted to [34], which is worth to read with some very useful usage patterns of MongoDB for different scenarios.
5.3 Next steps

**Metrics for time series using MongoDB**

Cube\(^5\) is a system for collecting timestamped events and deriving and visualizing metrics out of those events. Cube is built on top of MongoDB. By storing the original events instead of only metrics, Cube allows the user to perform post-hoc data analysis on the stored events. Cube uses automatic caching of metrics to MongoDB capped collections, to enhance performance. Capped collections are fixed-size collections that support high-throughput operations. Once a collection fills its allocated space, it overwrites the oldest documents in the collection to free space. Cube also uses pyramidal aggregation, which is basically built-in auto-summarization of computed metrics by multiples of the requested computed metric interval. For example, if you ask for the number of events in a particular day, Cube can use previously-computed hourly sums without a full event scan.

**Summary**

The main conclusion from [51], is to prefer asynchronous indexing techniques if possible. Another conclusion from the same paper is to try to prefer integer based indexes/keys instead of string. Integers in all cases perform better with regard to the amount of writes. The next set of suggestions from [34] all are very useful. A large part of those suggestions overlaps with some of the suggestions for column-oriented databases. Finally, Cube is a good option for visualizing and storing time series data.

As we can see in the suggestions from [34], document databases are really good at performing ad-hoc queries. The aggregation framework provided by MongoDB is a nice abstraction from the map/reduce paradigm. It provides many commonly used functions to the programmer directly, while it is not so straight-forward to implement such methods using map/reduce. This is the case because map/reduce is a distributed algorithm, which has inherent complexity and hard problems to solve. Almost all of the suggestions discussed can be implemented together without one excluding the other.

**5.3. Next steps**

Having all the insight from the suggestions and data schemas presented in this section, the next step for this thesis is to perform some tests with regard to performance of different data schemas. A big portion of the data schemas provided in this section were specific for Cassandra and HBase and a bit less for MongoDB. This is related also to their purpose, column-oriented databases seem more suitable for massive time series, because they can handle more inserts compared to document databases.

\(^5\)https://github.com/square/cube/wiki [online accessed 06/11/2013]
A project undertaken by TNO is the reason TNO was interested in exploring data schema possibilities using noSQL databases. The experts at TNO had already settled for using Cassandra for the project, after performing research in the field. But as our finding in this section also show, Cassandra is widely used as a time series storage, which further verifies the choice of database by the TNO experts.

Therefore for the practical part of this thesis, we perform load tests using Cassandra. The tests are performed with regard to performance of different data schemas. This is why this overview is very relevant to this thesis.
6. Test scenarios

In this section the structure of the tests and some terminology used throughout the tests are presented. The tests are performed on Cassandra. We dedicated extra time exploring how Cassandra works and different possibilities for the data schemas. This prohibited us from performing tests using other noSQL databases, due to the limited time for this thesis. Different data schemas for Cassandra will be compared with regard to operation throughput and latency. Fig. 6.1 shows the structure of the tests.

For each data schema 3 different scenarios will be tested, namely: write, read and mixed scenario. These scenarios should be as generic as possible. This will help in finding applications for the results in different scenarios that involve time series data. First, the process followed before the tests is presented in sec. 6.1. Then, a brief description of the different scenarios for each test is given in sec. 6.2. Finally, a description of metrics that are recorded for each test is given in sec. 6.3.

6.1. Pre-test phase

In this subsection the process performed before the actual tests is described. This is required because as already mentioned, noSQL databases do not provide a uniform
way of access, nor do they all share the same set of features and configuration settings. Therefore, before performing the tests mentioned above, each database will need to be configured and optimized for the particular test. If no optimization of configuration settings is performed we will not get the most out of each database, therefore the results of the tests will not be very useful. The intent is for the results to be applicable to real world production environments, therefore the database is optimized. The specific configuration settings for Cassandra are presented in sec. 8.2.2. This optimization phase will also involve some tests, to find out the optimal settings for Cassandra. The results of these tests will be briefly presented, to keep the scope of this thesis small.

6.2. Different scenarios

Each test that is performed per data schema, involves these three different scenarios. The scenarios are briefly described next.

**Write scenario:** In this scenario the database will be tested for its ability to store massive amounts of data. The term massive here does not refer to the total size of the data to be inserted, rather on the frequency at which data arrive. This will provide us with information on what is the maximum number of requests that our cluster can handle. This is an important characteristic, because of a project undertaken by TNO where the number of incoming measurements per second is massive.

**Read scenario:** In this scenario the database will be tested for its ability to perform read requests. Since we are dealing with time series data accessing single measurements at particular timestamps is not very meaningful. The most common pattern used against time series data, is range queries (requesting all values between a start date and an end date). This will give us an estimation of how different data schemas affect performance of read requests. However analytical queries will not be explored in this test. In this test we are only interested in the performance of the database when fetching a range of raw measurements.

**Mixed scenario:** In this scenario a mixture of read and write operations will be performed. This should give us a view of how each data schema performs in a production environment. This is the case because usually companies use one database to store their time series data, however a varying number of applications are querying the database in order to retrieve information. The ratio of writes to reads varies between different companies, however since we are dealing with massive amounts of sensor data (e.g. thousands per second), it is assumed that the ratio of writes will be higher than that of reads. So a ratio of 1:3 (reads to writes) is used. There probably are exceptions, but we believe it is a common case for massive sensor data. Furthermore, at TNO a
6.3 Metrics measured

In this subsection the metrics measured per test are presented. Our interest for the outcome of these tests is how performance is affected with regard to the data model being used, so some performance metrics must be measured. The authors in [29] provide a taxonomy for characterizing performance. The attributes that are of most importance for our tests will be described here as they are presented in [29].

**Latency:** The time it takes to respond to a specific event. This is the time interval during which the response to an event must be executed. The time interval defines a response window given by a starting time (minimum latency) and an ending time (maximum latency).

**Throughput:** The amount of operations that the database can process over a given interval of time. For throughput, the number of operations on a specific time interval must be specified. This leaves some freedom to the system. For example, if a datastore has a throughput of 60 operations per minute, it does not necessarily mean that one operation needs to be executed every second. Maybe in the first thirty seconds 10 operations are executed and the remaining 50 at the remaining 30 seconds.

**Capacity:** The demand that can be placed on the system while continuing to meet latency and throughput requirements.

Unfortunately not all of our tests are identical, some differences between tests existed. The three different scenarios are the same throughout the tests, but these differences affected the resulting diagrams. We encountered some issues with the client we used and we decided to switch to a different client. This is described in more detail in chapter 9. For the first client we run each test multiple times with an increasing number of client instances each time. We did this because a single client instance did not fully utilize the client resources. The main reason for this is that the client we used was synchronous. The client switch we made had a big impact, in the sense that the new client was asynchronous. This allowed us to reach a higher capacity of the cluster. This is further elaborated in the following chapters (c.f. sec.8.2, chapter 9). The reason this is mentioned here, is because it had an impact on the resulting diagrams. The resulting diagrams from the tests are briefly described next.

- **Throughput:** The throughput achieved for each scenario is presented. In the case of the synchronous client, this is presented in a diagram of throughput over the number of clients. For the asynchronous tests a single client instance was able to max out the resources of the client machine, which eliminated the
need to run multiple client instances. For the asynchronous tests a bar chart that provides detailed information on throughput is shown.

- **Latency**: The other metric presented for each scenario is latency. In the case of the synchronous client, again the latency over the number of client instances is shown. When using the asynchronous client, a bar chart with the detailed latency metrics is given.

- **Latency / Throughput**: This diagram will give us a view of how each data schema performs. As the throughput increases, the latency is also expected to rise. Once the latency increases beyond a threshold, it is assumed that the database is not really usable. Therefore, this diagram should give us a concrete view of how many operations can be executed before the cluster reaches its “full” capacity. Comparing these diagrams between different data schemas should give us a view of which data schema performs best under certain circumstances.

The number of client instances is used for the synchronous tests, since this is what varies between different tests for the same data schema. Therefore, it will give us information on what the capacity of our cluster is. At some point, the cluster is expected to reach its limits and have a relatively steady performance. The latency over throughput diagram will give us information on how the cluster performs with regard to the particular schema, since different tests will be run on a per different data model basis.

### 6.3.1. Data schema and terminology

It is important to elaborate on the data schema possibilities provided by Cassandra. This makes it easier for the reader to keep track of the different parameters tweaked between the tests. Fig.6.2 shows a generic Cassandra data schema that helps us explain what each term used for the tests means.

**Keyspace**: Each keyspace is unique on a per database cluster basis and each database cluster can have multiple keyspaces. Each database cluster is distinguished by its particular cluster name. For our tests we are always dealing with one database cluster and one keyspace, therefore it is not elaborated further. A keyspace can be thought of as a database in relational databases.

**Column family**: Each keyspace can have multiple column families, given each column family has a unique name. Column families can be thought of as tables in relational databases.

**Row key**: Within column families data are first grouped by their row key. Each row can have a variable length of column key-value pairs, we call this the timewindow of each row. Each column family can store multiple rows and each row key must be unique on a per column family basis. The number of rows is
6.3 Metrics measured

### Figure 6.2.: Visual overview of Cassandra data schema elements

Dynamic and does not need to be known in advance. Each row in Cassandra can store a variable number of column key-value pairs.

**Column key:** This is used as a "label" for recognizing the respective value that is stored. However, Cassandra does not limit its usage to labeling. It can also be used to store values that are not in the form of key-value pairs (e.g., followers of a user). The respective value can be left empty (this is different from null, no actual storage space is used). Each column key must be unique on a per row key basis.

**Value:** This field is used to store the actual value in key-value pairs. It does not need to be unique in any way, since it is meant to store values.

In case of time series data, rows can be thought of as timelines. Rows contain a range of raw measurements. The term timelines also refers to the nature of the measurements. For example, one timeline could be the actual raw measurements as sent by a sensor, a different timeline a prediction we have made about a specific time range and a different timeline storing the raw measurements after some processing has been performed on them (e.g., noise removal). Each timeline has multiple rows, for different time intervals. This is a logical distinction and does not really affect the storage of raw measurements, however it is important to describe the concept.

For read tests on Cassandra some extra terms are being used that require explanation. Fig. 6.3 shows a diagram to help the reader understand what each term means.
Figure 6.3.: Explanation of terms used in read tests

**Timerange:** The timerange refers to a range read request and is defined as the number of values requested by the particular range read request. Essentially this is the difference between the end timestamp and start timestamp. So a timerange of 50,000-55,000 refers to a range read request that requests 50,000 to 55,000 measurements. The 50,000-55,000 means that a random number between these two ranges is requested.

**Query count:** This is configured by the client and is a parameter to the client API. It is similar to what the `LIMIT` clause does for SQL queries. It refers to how many results will be returned in each response. This is a rather low level term, however it is important to mention since it does drastically affect performance of read requests.

For example, if a range read request with a timerange of 20,000 is issued but the query count is set to 10,000, only the first 10,000 measurements will be returned. The rest of the measurements are “lost” for that particular read request. So big requests can be split up into smaller ranges to match the query count. The examples in Fig. 6.3 are dummy data with small numbers to help the reader understand the terms and their purpose. These terms will become relevant in the read scenarios.
Cassandra write path

Another thing relevant to the tests for Cassandra is the process followed for persisting a write request. This will help understand concepts later (for the configuration settings and the actual tests). In order for Cassandra to be performant for the write requests it tries to avoid as much disk operations as possible. All this information comes from the Cassandra documentation [6]. Cassandra has a commitlog which is used to ensure that writes are persisted, however this is a temporary storage to ensure that no data is lost. A write is considered to be durable once it has been written on the commitlog and in memory. For the storage of data in memory they use a data structure called memtable. Writes are batched in memory and periodically written to disk to a persistent table structure called an SSTable (Sorted String Table). SSTables are flushed into disk sequentially, which leads to no disk seeks when a flush is performed.

Furthermore Cassandra on the background periodically merges SSTables together into larger SSTables. This process is called compaction. Compaction merges row fragments together, removes deleted columns and rebuilds primary and secondary indexes. Once a new SSTable is complete, the smaller SSTables that were used to make up the larger SSTable are marked for delete, which is handled eventually by the JVM garbage collection. Compactions result in high disk space usage and increased disk I/O operations.

6.3.2. Test parameters

For Cassandra various different data schemas will be used. Each data schema is denoted by a different subsection. The relevant configuration settings for Cassandra are expected to be the same for all data schemas tested, since we are working with the same database. The configuration settings are:

Timewidth: What the term means is explained in sec. 8.1. This is not expected to affect the performance of write requests, but is expected to have a big impact on the performance of read requests. Therefore, a pre-test to find a good value for the timewidth will be performed. The results are presented in sec. 8.1.3.1.

Replication: Cassandra offers configurable data replication. The first setting is the replication factor (N), which is on how many replicas each write operation is persisted. The higher this factor is, the more reliable the data is in case of failure of a node. Furthermore, for each write request the write request consistency (W) can be set. This indicates the number of replicas that need to acknowledge a write request, for the write request to be successful. Obviously, this needs to be equal or below the replication factor (N). Last, there is the read request consistency (R) setting which can be configured. This is the number of replicas that need to serve a specific read request. The higher this
number is, the higher the chance that the results returned are consistent. In the documentation of Cassandra it is mentioned that if \( W + R > N \) the results will be consistent. The lower \( W + R \) are, the chances for inconsistent results is increased.

Another configuration setting that could be used is using a different data partitioner. A data partitioner is responsible for distributing the different rows among different nodes (each row is always stored in one machine). Choosing a wrong data partitioner can lead to hotspots, that can slow down a particular machine. Our choices are between Murmur3, ByteOrdered and Random partitioners. The Random partitioner is an old version of the Murmur3 partitioner. In the documentation it is mentioned that Murmur3 is more efficient than Random partitioner, therefore Murmur3 is prefered over Random partitioner. The ByteOrdered partitioner seems like a good fit since it orders rows lexically by key bytes. However, it involves extra administration effort and its results can be achieved by manually creating indexes\(^1\). Therefore, we have decided to use Murmur3 partitioner and manually create indexes for the records.

### 6.3.2.1. Measured metrics

For measuring the performance of Cassandra, OpsCenter\(^2\) provided by Cassandra Datastax edition is used. OpsCenter is a web application that is used to monitor the performance and status (running, stopped, etc.) of Cassandra. It provides a nice GUI for monitoring the cluster and a plethora of metrics are measured and are available to display. Moreover, the measured metrics are provided on a cluster wide basis or on a per node basis, which is quite useful. For the test results we are interested in the cluster wide performance, however we can use the per node metrics, to ensure that all the nodes are running properly. Furthermore, by monitoring each node separately once we hit a bottleneck in the system, these metrics should provide us with some useful information in detecting where exactly our bottleneck is. A brief description of these metrics is presented now, as it is described in Cassandra Datastax documentation\(^3\):

**CPU usage:** Shows average percentages for CPU utilization metrics, which is the percentage of time the CPU was idle subtracted from 100 percent. CPU metrics can be useful for determining the origin of CPU performance reduction.

**Load usage:** The amount of work that a computer system performs. An idle computer has a load number of 0 and each process using or waiting for CPU

---

\(^1\)http://www.datastax.com/docs/1.2/cluster_architecture/partitioners [online accessed 07/09/2013]

\(^2\)http://www.datastax.com/what-we-offer/products-services/datastax-opscenter [online accessed 07/09/2013]

\(^3\)http://www.datastax.com/what-we-offer/products-services/datastax-opscenter [online accessed 07/09/2013]
6.3 Metrics measured

- **Time increments the load number by 1. Any value above one indicates that the machine was temporarily overloaded and some processes were required to wait.**

  **Disk queue size:** The average number of requests queued due to disk latency issues. This should denote if disk becomes the bottleneck.

  **Disk latency:** Measures the average time consumed by disk seeks in milliseconds. Consistently high disk latency may be a signal to investigate causes, such as I/O contention from compactions or read/write loads that call for expanded capacity.

  **Network traffic:** The speed at which data is received and sent across the network, measured in kilobytes per second. This should denote if network becomes a bottleneck in our case.

However, these metrics will only be monitored to ensure correct behavior of the Cassandra cluster. These metrics will not be detailed in this thesis, however they should provide a good indicator if something is going wrong with the cluster. Besides the metrics measured by OpsCenter, the frequency of requests and latency will also be measured on the client side.
7. Test setup

For this thesis some tests are performed on Cassandra to test its performance with regard to different data schemas. The setup of the tests will be the same for all tests and it is presented here. This is important to specify, in the case that somebody is interested in reproducing the results of the tests. Otherwise with different hardware and a different setup, different results might occur. Figure Fig. 7.1 shows an overview of the network topology.

![Network topology for the tests](image)

**Figure 7.1.:** Network topology for the tests

The two X’s in Fig. 7.1, represent some hardware failures that we encountered on the client side. Each failure happened on a different set of tests. So for the first set of tests (c.f. sec. 8.1) we had 7 client machines, for the next set of tests (c.f. sec. 8.2)
6 machines and for the final tests (c.f. chapter 9) 5 machines. This was a little bit problematic, however it was not possible to replace these machines in time for this thesis. This is an unfortunate occurrence and we could not somehow avoid this.

The hardware specifications of the machines involved in the tests are presented next. First, the database cluster hardware specifications are presented, followed by the client machines hardware specifications. Finally, the hardware specifications of the switches are shown.

**Database cluster nodes**

As depicted in the diagram, the database cluster consists of 6 nodes. Each node has the same hardware characteristics which are:

- **Operating system:** Ubuntu 12.04
- **CPU:** AMD Opteron 4226 @ 2700 MHz
- **Number of cores:** 6
- **RAM:** 48 GB
- **HDD:** 2 TB
- **Network interface:** Intel 82574L Gigabit Network Connection

**Client machines**

The client machines consist of three laptop and three server machines. They are used to run the client that is used for each test. For the tests using the synchronous client (c.f. sec. 6.3), each client machine will be running more than one instances of the client program. This number of client instances is specified on a per test case basis. For the asynchronous tests one client instance is enough to max out the resources of the client machine (c.f. chapter 9). The Hard-Disk Drive (HDD) is not listed for the laptop clients as it is not very relevant, while for the database cluster it is more important. The only different hardware characteristic between the laptops is CPU, therefore only that will be distinguished. The rest of the hardware characteristics are same among the three laptops. The hardware characteristics of the laptop machines are:

- **Operating system:** Windows XP Professional
- **CPU:** Intel Core Duo T2300 @ 1.66 GHz, Intel Core Duo T2400 @ 1.83 GHz, Intel Core Duo T5500 @ 1.66 GHz
- **Number of cores:** 2
- **RAM:** 3326 MB
- **Network interface:** Broadcom NetXtreme 57xx Gigabit Controller
Test setup

The hardware characteristics of the server machines used as clients are described next. To avoid any misunderstanding, each of these machines has two CPU’s.

**Operating system:** Ubuntu Server 12.04.2

**CPU:** 2 Intel Xeon CPU’s @ 3.20GHz

**Number of cores:** 1

**RAM:** 4 GB

**Network interface:** Intel 82541GI Gigabit Ethernet Controller

Network switches

Finally the specifications for the two switches that are connecting the clients with the servers need to be described. The normal setup at TNO is going through local cloud infrastructure that the company has. However, for our tests we used this setup with 2 switches to bypass the TNO cloud. This was done because it is a production environment and the additional network traffic generated by our tests, might interfere with their day-to-day operations. Following are the specifications of the two switches, which are identical.

**Manufacturer:** Hewlett Packard

**Product model:** HP 2910-24G-PoE+ al Switch

**Number of network ports:** 20 auto-sensing 10/100/1000 ports(IEEE 802.3 Type 10BASE-T, IEEE 802.3u Type 100BASE-TX, IEEE 802.3ab Type 1000BASE-T)

**Ethernet technology:** Gigabit Ethernet

**Network technology:** 10/100/1000Base-T
8. Cassandra Hector tests

In this section the tests we performed using Hector\(^1\). Hector is a synchronous client for Cassandra\(^1\). In total we performed two sets of tests because the results were not what we expected for the first load tests that we performed. For both load tests Cassandra version 1.2 from Datastax was used. First, in sec. 8.1 we give some information on the client program used and we then present the results of the first load tests. Next, in sec.8.2 we describe the configuration changes we made to Cassandra before proceeding with the second set of load tests. Then, in the same section the results of the second set of load tests are presented. Finally, in sec.8.3 we compare and discuss the results of the two sets of tests that we performed.

8.1. First set of load tests

In this section the first set of load tests we performed using Hector are presented. First, some information on the structure of the client program used is given in sec.8.1.1. Then, the first set of tests before making the configuration changes are presented in sec.8.1.3.2.

8.1.1. Client program

In this section some general information about the client program used for the tests will be presented. Fig. 8.1 shows a visual overview of the available ways to interact with Cassandra.

The lowest level API’s provided by Cassandra are Thrift and a binary protocol specifically designed for Cassandra Query Language (CQL). Hector provides a higher level API than Thrift that Cassandra ships with. We are going to use Hector for the tests, since Hector along with Astyanax\(^2\) are the two most popular Cassandra clients. There was no particular reason for choosing Hector over Astyanax, they are both considered stable and well tested clients. From Cassandra version 1.2 onward a native binary protocol was added, which supports CQL 3. Before version 1.2 CQL was built on top of the Thrift interface, however this was not optimal, which is why they implemented it using the native binary protocol. The Hector client program we are using is minimal. The most important classes are briefly described next.

---

\(^1\)http://hector-client.github.io/hector/build/html/index.html [online accessed 03/08/2013]

\(^2\)https://github.com/Netflix/astyanax [online accessed 26/09/2013]
Access2Cassandra: This class sets up a connection to the database and sets up the keyspace if it does not exist.

DataAccessInterface: This interface defines the methods to be implemented by the actual class responsible for the Hector interaction. The two main functions that are used are write() and read().

MeasurementSeries: This class is responsible for the actual interaction with Hector. It implements the DataAccessInterface. Within the write() and read() methods the main functionality for storing and retrieving records in the database is implemented. The actual class is a little more complicated and has some additional classes that it interacts with, however these details are not essential for this report. These additions are implemented for internal purposes of TNO, however these additional classes are not expected to interfere with the performance of our client, since they are not really utilized.

LoadGenerator: This keeps track of how many events have been invoked, printing messages about the status of the client and finishing the test once all of the operations have been invoked.

Sender: This is an abstract class that defines what each event in the LoadGenerator will perform. Two different subclasses of this are used for each different test scenario, ReadSender and WriteSender.

AssembleMeasurement: This class is responsible for creating the data to be inserted or read for each scenario.

The client does not somehow limit the frequency at which events are generated. Each client instance generates events constantly. However, it is a synchronous client, therefore when it waits for the response from the database it is blocking (not performing other requests while waiting). Because of this, one client instance does not fully utilize the resources of the client machine. This is why for each scenario we run an increasing number of client instances on each machine (e.g. 1, 2, 3, 4, .. instance(s) per client machine). Each client instance is thread-safe, so it is not problematic to run multiple client instances on the same machine. We stop adding new
client instances when the additional throughput gained from the additional client instance is minimal.

The client keeps track of the average frequency at which events were generated. Once the test is finished the average frequency for the entire test duration is reported. We take a note of the average frequency for each test. This is done because the frequency at which clients generate messages is directly related with the performance Cassandra achieves.

Each row key is a timelineID_timestamp combination that works as an index for our client. Each measurement has a timestamp for the column key and a random generated string for the value. The timestamp is simulated (they are not actual timestamps), we start from one number (e.g. 1,000,000) and increment it by 1 for each new measurement. This is not a very realistic scenario, however if this was not used we would be limited to generating 1 timestamp / millisecond which would be limiting.

**Issues with the client**

For the generation of events from the client we faced a limitation without an obvious reason (e.g. CPU, RAM or networking usage). With the help of a TNO employee, Frens Jan Rumph, we found out that this limitation is most likely accounted to the fact that the length of a Thrift frame is sent as a separate TCP packet from the TCP packet containing the Thrift frame itself. The result of this is additional protocol overhead, which might be an explanation for the limit we face in the generation of events.

The impact of this limitation is that for our tests, the cluster is not stressed to its limits. This way we cannot know the exact difference in performance between different data schemas for Cassandra. The results of our tests with this issue are consistent between them (they result in similar numbers), therefore the stability of the client is not a problem. However, the optimal case would be for each test to stress the cluster to its limits and then compare the performance of different data schemas.

**8.1.2. Test parameters**

In this subsection, we present a table with the configuration parameters for all of the tests performed in this section. In Tab. 8.1 we see the configuration parameters.

In Tab. 8.1 T1 stands for test 1, so the tests for this case are shown in sec. 8.1.3. Similarly T2 stands for test 2, so the tests for this case are presented in sec. 8.2.3. As we can see the test parameters for the scenarios are identical between the two tests. Only the number of events consistently varies between read and write scenarios. We adjusted the number of operations per scenario so that it would last approximately 7
Chapter 8

Cassandra Hector tests

<table>
<thead>
<tr>
<th>Test and scenario</th>
<th>Number of events</th>
<th>Timewidth</th>
<th>Replication factor</th>
<th>Write consistency</th>
<th>Read consistency</th>
<th>Timerange (min - max)</th>
<th>Query count</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1-write</td>
<td>300,000</td>
<td>100,000</td>
<td>3</td>
<td>Quorum (2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T1-read</td>
<td>50,000</td>
<td>100,000</td>
<td>3</td>
<td>-</td>
<td>One</td>
<td>9,000-10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>T2-write</td>
<td>300,000</td>
<td>100,000</td>
<td>3</td>
<td>Quorum (2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T2-read</td>
<td>50,000</td>
<td>100,000</td>
<td>3</td>
<td>-</td>
<td>One</td>
<td>9,000-10,000</td>
<td>10,000</td>
</tr>
</tbody>
</table>

Table 8.1.: Parameters for test scenarios

to 10 minutes. This time period was chosen for the tests, because of the limited time we have to perform the load tests for this thesis. That is why we have a different number of events for read and write scenarios. The throughput of reads is much lower compared to writes as we see on the test results in sec. 8.1.3.2 and sec. 8.2.3.

8.1.3. First Hector load test

First, we perform two pre-tests to determine the optimal configuration settings for our tests as mentioned in sec. 6.1. We perform a pre-test for timewidth and one for replication. For the replication factor, the different settings will provide us with exact numbers on what the tradeoff is between availability and performance. A high replication factor ensures higher availability, while a low replication factor is expected to provide more performance. The availability and performance requirements vary from project to project, therefore the results of this could be interesting. For the timewidth, different ranges should be tried for the same row width.

8.1.3.1. Pre-tests

These pre-tests are run for Cassandra to determine good values for certain configurations for each database. For Cassandra we run two different pre-tests. One for determining what is the relation between the width of each row (timewidth) with regard to the range requested by read queries and one for testing the performance difference between different replication factors. The results of each of these tests are described here. All pre-tests where done using the first data schema, shown in Fig. 8.4.
8.1 First set of load tests

Different time range queries for different timewidths

In this pre-test we are interested to see if there is any major performance difference between different timewidths with regard to the range of the values requested by the query. In order to test this we tried different ranges for the read query, on different timewidths. So the initial approach was to try ranges of 3,600, 10,000, 50,000 and 100,000 against a timewidth of 10,000 and 100,000 respectively. The diagrams in Fig. 8.2 show the results of these tests.

![Diagrams showing range reads pre-test](image)

**Figure 8.2:** Range reads pre-test

The different series in the diagrams in Fig. 8.2 show the results when performed on a timewidth of 10,000 and 100,000. In Fig. 8.2 (a) we can see the average number of read operations per second, with regard to the query range used. For this diagram higher values on the number of operations is better. In Fig. 8.2 (b) we see the average latency per operation, with regard to the query range used again. In this case, a lower number for latency is better. Last, in Fig. 8.2 (c) we see a combination of the two previous diagrams, the average latency per operation, with regard to the average number of read operations per second.
From the diagrams in Fig. 8.2 it is clear that the timewidth of 100,000 performs better than the timewidth of 10,000. The tests were supposed to also invoke a query range of 100,000, however this was not really feasible. Our client got timed out exceptions after some time and was not able to fulfill the requests. This is because these queries invoked too many raw measurements which took too much time, therefore the request was timed out. For the timewidth of 10,000, this was also the case for the query range of 50,000. For the timewidth of 100,000, the 50,000 query was feasible, however performance degraded a lot as can be seen on the diagrams in Fig.8.2.

Besides this, as mentioned the timewidth of 100,000 performs a lot better. This is with regard to both the number of operations and latency. This can clearly be noticed in Fig.8.2 (c). The best performance is achieved on the lower-right part of the diagram (many operations, with low latency). Therefore, the timewidth of 100,000 will be used for the actual tests also.

**Replication pre-test**

In this pre-test we tested the performance difference between using a replication factor for values of 1, 2 and 3. Furthermore, we also tested what impact the different number of acknowledge per request has on the different replication factors. For these tests we did not add a single instance each time like we do for the normal load tests. We run it with 10 instances / client (60 instances) and 17 instances / client (102 instances). We found out that after 102 instances the added frequency is too small to try with more instances. For all of the pre-tests for the replication factor we invoked a total of 300,000 write operations per client, with a timewidth of 100,000. The timewidth should not affect the performance of writes, however we mention it for test setup purposes. Fig.8.3 shows the results of these tests.

In the legend of Fig.8.3, RF stands for replication factor and ack for write request consistency level. So ack 1 means that only 1 replica has to reply and ack 2 means that two replicas must reply to a request. As we can see in Fig.8.3, there is small variation in the replication factor of 3 and 2. The test scenario that we run with acknowledge of 1, steadily has slightly higher performance than the case of acknowledge 2. This makes sense, since each request has to await for less replicas in order to fulfill the request. The biggest performance gain can be noticed when using a replication factor of 1. Even though data are not safe that way (e.g. if the HDD of a node fails, all of that data is lost), however there could be use cases that such performance is needed. For example, in a scenario where we have massive incoming data during the daytime and a decrease in data during the evening, the data could be backed-up during the night. Still losing a days worth of data could be too much to tolerate, however this depends on each different use case. In any case it is good to know that if you forfeit availability, you can get a significant benefit in raw performance. For our tests we are going to use a replication factor of 3, since loss of data cannot be tolerated.
8.1 First set of load tests

8.1.3.2. Test results

This is the first attempt at the load tests that we performed. The tests themselves went fine, however the performance of the database did not meet our expectations. Therefore, we re-run the tests after changing some configuration settings of Cassandra to improve the performance. The results of the first attempt are presented in this section.

The first data schema that we try for the storage of sensor data can be seen in Fig. 8.4. The first term appearing in each box is the term used by Cassandra for that particular attribute (e.g. column family, row key, column key, etc). The term described after the “|” sign denotes the actual value that is stored there. Each set of brackets (< >) denotes one term, so for example in the case of the row key the value is a compound string of the timeline and starting timestamp for that particular row.

**Keyspace:** Standard keyspace, does not affect schema in this case.

**Column family:** Use one column family per sensor. The column family name is the sensor ID. Example: waterSensor-86

**Row key:** A compound field containing the type of sensor measurements (e.g. core, prediction, processed) and the time range contained in the particular row (e.g. start timestamp). The separator is “_”. Example: core_12569537329

**Column key:** The timestamp of the time that the particular measurement was received. This timestamp should be in the range described by the corresponding

![Figure 8.3.: Replication pre-test results](image)

Figure 8.3.: Replication pre-test results
Figure 8.4: Visual representation of first data schema

row key. Example: 12569537331

Value: The actual measurement value recorded by the sensor. This can be string or double, depending on the type of sensor. Example double: 145.9863, string: “sample measurement”

Write scenario

The test parameters for this scenario can be found in sec. 8.1.2 on the T1-write row of Tab. 8.1. The metrics for these tests were recorded using OpsCenter and the throughput was also measured in the client side. Fig. 8.5 shows how the database performed for the write scenario.

Fig. 8.5 shows the results of this test scenario. It can be noticed from Fig. 8.5 (a) and (c) that the additional frequency from each instance after a certain point drastically
8.1 First set of load tests

Figure 8.5: Write scenario result diagrams

diminishes. For example, from 28 to 35 instances 149 additional requests per second were generated. This is also the reason we decided to run up until 5 instances per client and not more. Fig. 8.5 (a) shows the average frequency reported by the client and the average frequency reported by OpsCenter. These numbers largely overlap with each other.

From the results of the diagrams we can see that the database is starting to get stressed, from the fact that the latency gradually goes up as the number of client instances increases as shown in Fig. 8.5 (b). However, 1.2 ms / operation is not a prohibitive response time. This is why as mentioned in the beginning of this section we rerun the tests after doing some configuration changes. Furthermore, Cassandra ships with a small program to stress test the database. When we run that program on each node (each node run the program and each program accessed all nodes), it was able to achieve about 55,000 write operations per second (peak), with an average latency of 5–5.5 ms per operation. This was a hint for us that better performance could be achieved.
Read scenario

The test parameters for this scenario are described in sec. 8.1.2 on the T1-read row of Tab. 8.1. The total number of requests is lower than the write scenario, because the frequency at which read requests are sent is lower compared to writes. The following diagrams show how the database performed for this test.

![Read scenario result diagrams](image)

**Figure 8.6.:** Read scenario result diagrams

Fig. 8.6 shows the results of the diagrams. Fig. 8.6 (a) shows the number of operations per second that OpsCenter reported and the average client frequency reported by the clients in the end. The server reported metrics are almost double from what the client reported. This can be accounted to the fact that for each read we first do a read on the indexing row that we are keeping to determine the key of the row that holds the requested data. After that the actual data are requested. Furthermore, some of the requests might span two rows since the starting and ending timestamp are random, only the range is standard. So when a request spans two rows our client reports that as one request, while OpsCenter as two requests. This could also explain why there is such a large difference in the reported metrics.
As mentioned the timestamps that we store are artificial. We are assuming that each increment in a timestamp, corresponds to the next second for calculating the time range that each request asks for. So 9,000-10,000 in our case corresponds to 150-166 minutes. This is not always the typical range requested, however bigger ranges can be broken down to multiple smaller ones to get the desired size. We chose this since it is more flexible (e.g. you can estimate how many queries you need to cover a larger one, however it is harder to estimate how a smaller query would compare to a large one), the larger time ranges and query count values drastically decrease the performance as noticed in the pre-test phase (c.f. sec.8.1.3.1).

As expected the performance of Cassandra for read requests is much lower than the performance for write requests. This can be largely attributed to the fact that Cassandra is optimized for writes. It performs reasonably for read requests, however this is not were it excels. For time series data as already mentioned, massive write requests is the main problem with regard to storage. There are ways to work around this relatively poor performance of reads.

For example, in a machine with a lot of memory row caches can be enabled. This enables caching rows in memory for faster access. This can be set on a column family basis. Row caches should be used for column families that are getting a lot of requests. An example of such a column family, could be one that is storing metrics for the actual raw measurements stored. If the metrics are accessed by many applications in order to show dashboards for example, enabling the row cache for those could have a big impact on read performance. In our test scenario we are using a single column family that stores raw measurements and we request ranges of those measurements. However the size of the dataset used for the read test is not that large (7 million measurements). This can easily be kept in memory in our case, however this would provide biased results. The way we run the tests most of the reads hit the disk, which is more representative of a real world scenario.

Again we can see that the database is stressed a little. Especially for read requests the latency can increase drastically as we have seen in the pre-test, depending on the query count and query range parameters. As mentioned already because of the performance of Cassandra, especially for the write scenario, was not what we were expecting we decided to tweak the configuration of Cassandra in order to achieve higher performance. The configuration changes and the new set of tests are presented next in sec.8.2.

8.2. Second set of load tests

In this section we present the results of the second series of tests, after making configuration changes to Cassandra to enhance performance. First, we describe the summary of a presentation for configuration changes of Cassandra in sec.8.2.1. Next, in sec.8.2.2 the configuration changes that we made for our tests are given. Then in sec.8.2.3 the results of the tests after the configuration changes are presented.
8.2.1. Tuning Cassandra

The authors in [48] discuss some configuration changes that they made to Cassandra to get the most out of it (performance wise). Scaling this kind of system is not very straightforward, and configuration settings on Cassandra have a big performance impact. Even though the default settings are sensible enough, each Cassandra cluster should be configured to match the custom specifications of the machines and the expected workload. They are using SSD disks for the data that need very fast access. On machines with SSD disks it is important to get as much processing power as possible, since the bottleneck stops being the I/O operations, but that transfers to the CPU which then becomes the bottleneck.

Below are some configuration settings of Cassandra that they tweaked in order to boost performance. This does not mean that these configuration settings will work for every single application, but at least it is worth trying to experiment with these parameters and see how performance is affected.

**JVM heap size:** This was increased to 12 GB in their particular setup. The default value for this is \( \frac{1}{4} \) of total memory, or a maximum of 8 GB in case that 25% of memory is above 8 GB. For their particular setup this is as high as they can push it.

**Space for young generation (Xmn):** This setting controls how much space the young generation is allowed to consume on the heap. In their setup they set this to 1,600 MB. Young generation\(^3\) is the place in memory of the JVM, where objects are allocated in a pool dedicated to young objects, most of which die there. This is related to Java’s garbage collection.

**Survivor ratio:** Ratio of eden/survivor space size. Eden is the heap space where new objects are created. In their particular setup they have set this to 16.

**Use compressed oops:** This enables the use of compressed pointers (object references represented as 32 bit offsets instead of 64-bit pointers). They enabled this setting.

**Use TLAB:** This setting enables use of thread-local object allocation. In their own setup they have enabled this. It had a direct influence on performance of read requests, boosting it by 15%.

Besides these JVM parameters, they have also tweaked some Cassandra settings. They set the hinted hand-off to a single thread, versus the default of two threads. They also modified the size of the memtable to 64. In their cluster they have 16 cores in total, they have assigned 4 cores for the compactions and the remaining 12 for reads and writes. They also turned off the inter-node compression, since it was causing too much garbage collection. The reason for this in their particular setup is

\(^3\)http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html [online accessed 26/09/2013]
that they have a 10 GB line connecting the nodes, so compressing the data was not very meaningful.

For the rpc_server_type parameter in Cassandra configuration, they are using half-synchronous half-asynchronous instead of the default which is synchronous. A last configuration that they changed is the default compression (snappy), they switched to using LZ4 JNI which is more efficient.

### 8.2.2. Configuration settings for Cassandra

In this section we are going to mention the configuration settings that we changed for our Cassandra cluster. As mentioned initially we believed the client was the bottleneck. Influenced by the suggestions described in sec. 8.2.1, we decided to proceed with configurations changes in Cassandra. We read about the different configuration settings available in Cassandra, then we changed the ones that were expected to affect performance. In our case we were mostly interested in the performance of writes due to a project that TNO will take, that has to deal with massive amounts of raw measurements. Each different configuration setting is presented and some words about what values are suitable for it are then presented. Some general Cassandra configuration changes we made are given next.

**Commitlog directory:** Cassandra supports durable writes by storing writes in a commitlog before storing them on disk. The commitlog has the really nice property that it is append only, which means that no time is spent on disk seeks. Each disk seek on rotational media take 5-10 milliseconds, which is a lot for a database. So what the developers of Cassandra suggest is using a separate disk for the data and the commitlog. Unfortunately the servers at TNO were only equipped with one Hard Disk Drive (HDD), so we were not able to apply this recommendation. In order to simulate this we increased the time that the commitlog flushes to disk from 10,000 to 1,000,000 milliseconds. It did not have a huge impact on the number of write operations, however the performance was steadier than when the commitlog was flushing to disk regularly. This was tested on relatively short term tests (300,000 write operations, 6-8 minutes), however especially for production environments that the data is expected to come in non-stop, having the commitlog on a separate HDD is expected to have a big impact on the performance of write requests.

**Caching mechanisms:** If read performance is of importance and the hardware that Cassandra is run on has enough memory, the caching mechanisms provided by Cassandra should be used. By default only column key caching is enabled. If a column family is expected to have a lot of read requests, it is a good idea to enable row caching for that table. By using row caching the disk is not reached at all, which can have a huge performance difference on read requests. However, not both column key and row caching should be enabled at the same time, since this is duplicate work that does not
need to be performed. Another option to tune with regard to caching is the row_cache_provider. This can be either SerializingCacheProvider or ConcurrentLinkedHashCacheProvider, the difference lies in how they deal with updates. The SerializingCacheProvider is 5-10 times more memory efficient. However, the SerializingCacheProvider is not efficient for update heavy workloads, since it invalidates the particular cached row on update instead of updating it in memory, like the ConcurrentLinkedHashCacheProvider does.

8.2.2.1. Thrift interface

The Thrift interface is used by the original clients for Cassandra. For new projects they propose to use the Cassandra Query Language (CQL) client for interaction with the database, which uses the native transport protocol for communication. However, our initial client was built using Hector, which utilizes the Thrift interface. These are some configuration settings that affected the write performance of requests.

**Thrift server type:** This configuration option is related to the Thrift server that Cassandra uses to handle requests. In version 1.2.4 of Cassandra according to grepcode\(^4\) there are three options provided by Cassandra by default. A synchronous, an asynchronous and a Half-Synchronous Half-Asynchronous (HSHA) server. Each of these options corresponds to a custom Thrift server. Mutsuzaki in [52] performs a performance test between the different servers provided by Thrift. The conclusions provided by the author in the conclusions for the tests, suggest using TThreadedSelectorServer for most use cases. Depending on the hardware setup of each project, if enough resources are available for multiple concurrent threads, the author suggests also considering TThreadPoolServer. In the Cassandra source code in grepcode the CustomTThreadPoolServer is used for the synchronous case, CustomTNonBlockingServer is used for the asynchronous case and CustomTHsHaServer for the HSHA case. We assume that the servers are custom modified for Cassandra, since the servers used do not agree with the results presented in [52]. The default value for this parameter is the synchronous case.

We found the Thrift server type to be a limiting factor in our setup. While using the synchronous server our clients were not able to generate more than 21,000-24,000 requests per second, which was disappointing. The first performance improvement on this was noticed when we increased the size of the JVM heap. By testing we set it to 20 GB and this resulted in a client frequency of about 30,000 operations per second using the synchronous Thrift server.

Furthermore, when we changed the server type from synchronous to HSHA, our clients were able to achieve around 34,000 requests per second. The difference of HSHA to synchronous Thrift server is not that much when using a high value for the JVM heap size. But as mentioned already, having a JVM heap larger than 8 GB results in slow garbage collection, which affects performance. During our tests we encountered this issue sometimes, when the cluster almost stopped performing operations for a short period and resumed. The advantage of HSHA is that it performs the same with a small JVM heap size (8 GB). Moreover, the server is half-asynchronous which leads to higher throughput, since more requests are handled in parallel and resources are more efficiently utilized.

In the synchronous case one connection per thread in the RPC pool is used. For the HSHA case the RPC pool is used again, however the threads are multiplexed across different clients. The Thrift clients are handled asynchronously using a small number of threads that does not vary with the number of clients. This leads to good scalability for many clients which was the case for our tests. For the HSHA server its important to note that the RPC requests are synchronous (one thread per active request).

**Compaction settings:** For compaction, Cassandra provides two different compaction strategies: size-tiered and leveled compaction. The size-tiered is appropriate for append heavy datasets, such as time series. The leveled compaction better suits workloads that involve many updates of existing columns. This can be set on a per column family basis. Another parameter that can be used to manage the behavior of compactions is the size limit of the rows being compacted in memory. However, larger rows spill to disk and use a slower two-pass compaction process, which degrades performance. This was briefly tested, however the results of the tests had no significant difference. Another parameter that can be tweaked with regard to compaction is the compaction throughput. This throttles compaction to the specified total throughput (measured in MB). If the workload is write intensive, we need to compact faster to keep the number of SSTables low.

**Concurrent reads and writes:** Two other parameters that are available for tuning performance in Cassandra are the number of concurrent reads and writes. If the dataset being stored does not fit in memory the bottleneck for the database is reads, because fetching data from disk is a slow process. This parameter should be set with regard to the number of drives that the particular Cassandra node has. This allows operations to queue low enough in the stack so that the operating system and drives can reorder them. On the other hand the value for concurrent writes depends on the number of CPUs present in a node, since writes in Cassandra are rarely I/O bound. So the more CPUs a node has the higher this value should be set. In our setup we used 16 for concurrent writes, since we only have 1 HDD per node and 64 for concurrent writes, since each node has 8 CPU cores.
8.2.2.2. Java parameters

Besides the parameters mentioned that are exposed by the Cassandra configuration file (cassandra.yaml), Cassandra is a Java project, therefore the parameters passed to the JVM also have an impact on performance. Actually these along with the Thrift server type configuration settings are the ones impacted performance the most. The parameters that we changed are presented here along with a short discussion on why the performance is affected and some possible values for these parameters.

**JVM heap size:** The default heap size in Cassandra is \( \frac{1}{4} \) of the entire RAM. For 32-bit JVM’s this is actually limited to 2 GB, since the system cannot recognize more than 4 GB of RAM. For 64-bit systems this restriction does not hold. However, in the documentation of Cassandra when they mention what to take into account for setting the heap size of the JVM, is that for heap sizes above 8 GB garbage collection can slow down things a lot. So they suggest that even for machines with a lot of RAM (e.g. 48 GB as is the case for our cluster), setting it to 8 GB is still the suggested value. When using a Thrift client to interact with Cassandra however, if the synchronous Thrift server is used increasing the memory above 8 GB can improve performance (number of requests per second). For the HSHA server type, this is not needed as the memory usage is constant.

**Heap dump when out of memory:** This is passed to the JVM by passing the argument `-XX:+HeapDumpOnOutOfMemoryError`. Essentially this tells the JVM to dump the heap to a file when a `java.lang.OutOfMemoryError` is thrown. This is used to ensure that the heap size does not reach and stay full. But when the heap dump occurs, performance is expected to degrade.

**Thread Local Allocation Buffer (TLAB):** This option is enabled by passing `-XX:+UseTLAB` to the JVM. This enables thread-local object allocation per machine. TLAB is a region of Eden that is used for allocation by a single thread. Eden is the place in memory where the JVM stores new objects. It enables a thread to do object allocation using thread local top and limit pointers, which is faster than doing an atomic operation on a top pointer that is shared across threads.

8.2.3. Test results

The scenarios for this set of tests is the same as the ones described in sec. 8.1.3.2. The data schema used is the same as in Fig. 8.4. Actually the client was barely changed, some minor structural changes in the way that the tests are performed were made and some additional code was added to measure the latency of requests on the client side also. So for these tests some additional information is presented namely: average, minimum and maximum latency per test, recorded on the client side. This was added to help us gather more information for deciphering the results of the tests.
So with the client and data schema essentially remaining the same as in the previous test, it would be interesting to see the performance difference that the configuration changes brought. So first the results of the second test are presented and discussed on their own and after in sec. 8.3 we compare the numbers between the two different set of tests.

**Write scenario**

The test parameters for this scenario are shown in sec. 8.1.2 on the T2-write row of Tab. 8.1. The diagrams in Fig. 8.7 show how the database performed for the write scenario. The names next to the axis should help the viewer understand what is presented in these diagrams. Number of client instances refers to the total number of client instances run on all client machines. Many client instances per client machine were used for our previous tests also, as described in sec. 8.1.1. For all the tests we start from 1 instance and sequentially increase the instances until 17 for write and 10 for read scenarios. Another clarification to be made is about the latency reported in the diagrams. The server measured latency is what OpsCenter reported for the duration of the test. OpsCenter measures the average latency over 1 minute intervals and it is measured in milliseconds. The client measured latency, is the latency that we measured on the client side. Each point in the diagrams represents a different run of the same test, with more client instances. For this we averaged the metrics mentioned by the clients and OpsCenter and present that as a single point. This was done to summarize the results, otherwise there would be too many diagrams, which would make it harder to comprehend the results.

As can be seen, the performance overall is better with regard to the number of write operations. Also what is clearly visible is that by using the ISHA server for Cassandra we are able to support more clients. In Fig. 8.7 (a), we can see that what OpsCenter reports largely overlaps with what the client reports in the end of the test. For the points that the client reported frequency is higher than what OpsCenter reported, can be accounted to unavailable exceptions that we were getting. Especially for the test with 84, 96 and 102 instances we had 2,731, 6,878 and 1,955 unavailable exceptions thrown in total from all clients. Unavailable exceptions occur in Cassandra when a node is down or a node is too busy to handle the request. Therefore when this occurs in an actual application the normal thing to do would be to retry the operation. However in our client we only count the number of unavailable exceptions, since we do not really insert actual data. Especially for the case of 96 instances and 6,878 unavailable exceptions, it was noticed that heavy garbage collection was taking place (we noticed this on OpsCenter). This explains the high number of unavailable exceptions also. The GC caused the node to not have enough available resources to handle the request. After some point the added operations diminishes too much, which is why we stopped adding instances at 17 instances per client.

Fig. 8.7 (b) is quite similar to the results of the previous test. The latency reported
by OpsCenter is quite low, it peaks at 1.5 ms. However, what is interesting is that the latency reported by the client is almost always double than what OpsCenter reports. Part of this is definitely because the client and OpsCenter record different paths of the request. OpsCenter starts measuring a request when it arrives, until Cassandra sends the response. The client on the other hand starts measuring the time from the point it sends the request, until it receives the acknowledge. Therefore, the client also measures the time it takes to travel back and forth twice (once for the request and once for the response). This partly explains this difference, however we are running the tests on a LAN, so the latency difference because of the network should not be so high.

Fig. 8.7 (c) shows the minimum and maximum latency for each run of this scenario. The minimum latency is constantly around 0 milliseconds. Besides this, no other particular patterns can be observed. The maximum can go pretty high and this occurs more often as more client instances are involved. However, this is not always the case, for example when near to 80 client instances the maximum is pretty low, compared to the test runs near 100 client instances. The longest a request took was 10 seconds, which is the case because this is the configuration setting we are using for time out of requests. If this configuration setting was higher, the maximum latency.

**Figure 8.7.:** Write scenario result diagrams
8.2 Second set of load tests

would be higher, but 10 seconds is already a long period and is not acceptable for a production environment.

In Fig. 8.7 (d) we see a combination of the top two diagrams, the average latency per request over the number of operations. It does not add a lot of insight compared to the other two diagrams, but it gives us a clear picture of what is the maximum operations and the respective latency for our tests. So the maximum number of operations per second achieved in this scenario is around 32,000 requests per second with an average latency per operation of 1.5 milliseconds. The latency differs between the client and server metrics for the reasons explained before, when discussing Fig. 8.7 (b).

Read scenario

The test configuration parameters for this scenario can be found in sec. 8.1.2 on the T2-read row of Tab. 8.1. The diagrams in Fig. 8.8 show how the database performed for this scenario. Again the names of the axis should make it self-explanatory what is presented in each diagram. The terms are the same used as in the write scenario, where they were briefly explained. For this scenario the maximum number of client instances was 42, since the degrade in performance became obvious at that point. We tried also 60 and 90 instances, to ensure that no significant amount of requests is added. The results for 60 are presented in the diagrams, which as can be seen only adds about 200-300 requests per second. This is an insignificant amount taking into account that 18 extra clients were added. For 90 client instances, the majority of the instances got a timed out exception and could not complete the test. The performance for as long as the test was able to run was around 5,000 operations per second, therefore not that different from the throughput with 30-40 clients.

Fig. 8.8 (a) shows the number of read operations with regard to the number of client instances. We can see there is a difference here between the server and client reported metrics. This can be accounted to the fact that some of the requests, request values from 2 timelines. The client measures this as one request, while Cassandra sees it as 2 requests. Furthermore, we do the indexing “manually” in our client. This means that when a request asks for a range, we check to see in which row key this request falls. This is an additional read request, since the indexing row is kept in the same column family as the raw measurements. These reasons explain why there is a big difference in the reported metrics from OpsCenter and our client. Another note on this diagram is that as the number of client instances increases the number of timed out exceptions received from Cassandra also increases. As mentioned earlier, when we tried to run the test with 90 clients, most of the clients got timed out and were not able to execute the entire test. This is the case because the nodes were overloaded with requests from the clients and most of them had to perform disk seeks to serve those requests.

In Fig. 8.8 (b) we see the average latency per operation with regard to the number
of client instances. Again there is a large difference between the metrics reported by the client and OpsCenter. The reasons described for the previous diagram also apply here, since 1 request for the client is interpreted as 2 requests for Cassandra, therefore it makes sense that the latency is double or even more at times. Another factor that affects the client metrics is as mentioned for the write scenario, the client records a longer period than the server for the request. In addition to this, the responses of the server are quite large since 9,000-10,000 requests were returned for each operation, which takes more time than a single acknowledge (which was the case in the write scenario).

In Fig. 8.8 (c) we see the maximum and minimum latencies reported by clients for each test run of this scenario. Again the minimum is constantly close to 0 milliseconds. The maximum definitely increases as the number of client instances increases. However, no particular pattern can be noticed about the maximum besides that it increases as the number of client instances increases. This again makes sense, since the nodes are stressed more with more client instances, which results in increasing the maximum response time.

In Fig. 8.8 (d) the combination of the top two diagrams is shown, average latency per request over the number of operations per second. It is even clearer in this diagram that the maximum operations for this given range is between 4,000 and 5,000. The average latency does not vary that much either (the one reported from the server

![Figure 8.8: Read scenario result diagrams](image)
at least). The latency reported by the clients keeps getting higher for the reasons we outlined when we discussed Fig. 8.8 (b). The mixed scenario was not performed since we skipped it for the first test (pre-configuration), due to the results that we got. We decided to run the full scenarios with the new client for Cassandra, CQL.

8.3. Comparison of the two tests

In this section we are going to compare the two sets of load tests we run, before the configuration changes and after. The metrics used are the ones from the server, since as mentioned we started measuring the latency on the client in the second set of tests. The different series in the diagrams have a varying number of points, because as mentioned already for the first set of tests the total amount of client instances supported was limited. This is why the post-config series has more points per diagram.

Write comparison

Fig. 8.9 shows the comparison of the write scenarios between the two tests.

As can be seen from the diagrams in Fig. 8.9, the post-config test performed better in terms of operations per second. While on the first set of tests the maximum throughput that we could get was around 20,000 requests per second, after the configuration changes this number increased to around 32,000 operations per second. The latency was also slightly higher for the post-configuration tests, but this makes sense since a lot more client instances were involved. Even though if we compare the additional latency (around 1.55 maximum), compared to the additional clients (73 additional client instances) the added latency is small.

This is mainly accounted to the fact that more client instances could be supported. The main reason for this change, was the configuration of the Thrift server type used by Cassandra from synchronous to HSHA. As mentioned already in sec. 8.2.2.1, the HSHA server provided the best results for massive time series data.

Furthermore, the concurrent writes configuration setting that we set to 64 also contributed to this difference in performance. This makes sense, since this is solely the purpose of this parameter and our machines on the cluster have 6 cores each.

Read comparison

Fig. 8.10 shows the diagrams comparing the read scenarios between the two tests. Both tests had exactly the same parameters, with the only difference of the server configuration changes. However, as mentioned already the configuration changes
mostly focused on the performance of writes, because this is what is most interesting to TNO for a project they have.

In this case the pre-config test seems to be performing slightly better. For each point, the post-config test seems to do slightly lower than the pre-config test. However, the maximum throughput reached by both of them is roughly the same (4,800 for the pre-config and 4,900 for the post-config). Besides throughput, the pre-config test did slightly better in the read scenario with regard to latency. It is consistently slightly lower than what it is with the post-config test. This could be accounted to the different Thrift server used. It makes sense that the synchronous server replies faster to requests than the HSHA server.

The performance for reads can be said that it has slightly degraded, since roughly around the same throughput is achieved, however with higher latency. This is the case, because our configuration changes were more focused on achieving more write operations than read. This is the case because of a project that TNO is researching about, were they have to deal with really massive rates of incoming records. They expect very few read queries, when compared to the number of writes. Therefore, for the case of reads it can be said that the synchronous Thrift server is better suited.

Figure 8.9.: Write scenario comparison
8.3 Comparison of the two tests

![Graphs showing comparison of read operations and average latency](image)

Figure 8.10.: Read scenario comparison

However, in order to support a large number of clients, the JVM heap memory size should be increased. This can lead to increased GC times (which we encountered for some tests when we were using synchronous Thrift server with 20 GB of memory).

8.3.1. Summary

The results of these tests were better compared to the first set of tests. The configuration changes of Cassandra had an impact on performance. At the time of writing this thesis a new version for the CQL client of Cassandra was introduced. This client utilized the binary protocol for communicating with Cassandra, as described in sec. 8.1.1. From impressions and suggestions on the Internet we decided to perform the tests with this new client, because it is easy to use and provides capabilities to issue requests asynchronously. With an asynchronous client we would not need to run multiple instances of the client, since one instance can better utilize the resources of a client machine. This was also a suggestion made in [61].

So these two first set of tests were more of an introduction to load testing for us and experimenting with Cassandra. This was required since we did not have any prior experience with load tests or Cassandra, which took some time to get familiar with. However, our tests should provide useful information to the reader with regard
on how to proceed with load tests. Furthermore, the test results and the pre-tests also provide information on what to look out for with regard to the data schemas. The actual comparison of data schemas is performed in chapter 9, using Cassandra’s CQL 3 driver.
9. CQL performance tests

When we performed the tests using Hector (c.f. sec. 8.1 and sec. 8.2) it was not easy to stress the database to its limits. We had to run multiple instances of the client to maximize the number of requests from each client machine, which by itself (running multiple instances) creates overhead. The results of the tests could be limited due to the way in which Hector works. Therefore, we decided to perform the tests using Cassandra’s CQL interface. The three main advantages of CQL are:

- It utilizes the binary native transport protocol of Cassandra which is more efficient than the Thrift interface.
- Provides possibility for asynchronous calls. Hector did not provide this functionality out of the box. It is possible to make asynchronous calls, however this functionality is not there it needs to be implemented. We avoided this due to the limited time for this thesis.
- The syntax of CQL is easy to learn for developers familiar with SQL. Most developers are familiar with SQL, because it has been widely used for many years. Due to this similarity it is easy for developers to adopt CQL.

The structure of this chapter is as follows: in sec. 9.1 we present the client differences from Hector to CQL in detail, in sec. 9.2 we show the structure for these tests and finally in sec. 9.3 we give the results of the tests that we performed.

9.1. Client differences

The client that we used for this set of tests is the DataStax Java Driver for Apache Cassandra[36], to which we refer as CQL 3 driver. There are some differences between this driver and Hector. The most notable difference is that it uses CQL to interact with Cassandra, which is a different style from Hector. With Hector we had to create the records and send the records themselves, while with the CQL 3 driver one simply executes CQL statements against the database. This adds a layer of abstraction, which makes it easier from a programmers perspective to understand the flow of statements, since it resembles statements from SQL which appear more natural to a programmer that is used to SQL databases. Another difference is related to the way in which requests are performed. Using Hector only synchronous calls are supported (without any modification of the client itself). The CQL 3 driver supports asynchronous calls out of the box.
Chapter 9  
CQL performance tests

The previous tests were performed using Cassandra v1.2. The respective version of the CQL 3 driver for this version of Cassandra was 1.0.2. When using this version of the CQL 3 driver we were not able implement some of the functionality required for the tests. Namely we had some issues with range read requests. Therefore, we decided to upgrade Cassandra to version 2.0 and use the respective CQL 3 driver (version 2.0.1). Besides the problems we encountered for the functionality, the new version of the client is easier to use. It provides convenient statements for CQL queries, such as the \texttt{IF NOT EXISTS} statement. This statement is used when creating a column family, to automatically check if the column family exists and if it does not exist it is created. With the previous version of the client this had to be checked by the developer.

The upgrade from Cassandra v1.2 to v2.0, included some big changes on the Thrift interface. The Thrift server used internally by Cassandra was changed, which as we saw in sec.8.2 can have a big performance impact. However, the binary protocol that the CQL 3 driver utilizes did not change much. This is a fortunate thing, because this means that the binary native transport protocol of Cassandra did not change that much between version 1.2 and 2.0. So the differences in the performance of the CQL 3 driver should be similar between the two versions. The main problem with the upgrade of Cassandra from version 1.2 to 2.0, was that OpsCenter (c.f. sec.6.3.2.1) at the point of performing the tests was not compatible with Cassandra 2.0. This was a major issue, that we tackled by measuring more metrics on the client side. Therefore, only the results on the client side are presented on these tests, since OpsCenter was not available. The way in which metrics were recorded is explained later in this subsection and in sec.9.2.1.

The client from version 1.0.2 to 2.0.1 had some differences. These differences were mainly related to the available functionality and ease of use of the client. We decided to use the latest version of the client because as mentioned it is a lot easier to use. Another reason for upgrading to the latest client was that some of the functionality required for the tests (namely some of the read scenarios), was not possible using the previous version of the client. Therefore, we decided to use the latest version of the CQL 3 driver. This also required an upgrade of Cassandra to version 2.0 as mentioned.

Another really nice feature of the CQL 3 driver is that it allows you to easily make asynchronous calls to the database. By making asynchronous calls, the client is not limited to the amount of requests it generates by the rate at which the database deals with requests. It continues with requests normally, while waiting asynchronously (without blocking next requests) for the responses of the previous requests. This allowed us to utilize fully the resources of the client machines with a single instance of the program. Namely for writes the CPU usage was almost constantly at 100%. When we tried running multiple instances on the same client the total number of requests remained the same, just divided over the two instances. Therefore, we decided to only use 1 instance for each client machine instead of multiple client instances, which was the case for the tests with Hector.
The CQL 3 driver provided some options regarding how many threads to use and the minimum & maximum number of connections per client instance. These settings were the same for all the tests presented. For the threads we always used 2 threads, since the client machines had 2 cores each and increasing the number beyond that caused some unexpected behavior, which is why we decided to use 2 threads per client instance. For the number of connections per client, we run some short tests to compare the performance and the outcome was that using 2 for minimum and 6 for maximum, is the best option for the connections per client instance parameter. This is specific for our setup, if tests are performed on a different setup this should be tested.

Another unfortunate occurrence was the fact that one of the servers that we used as client stopped functioning. There was no possibility to replace the server anytime soon, therefore we run the tests using 5 client machines instead of 6 which was the case for Hector. This happened after finishing the tests for the typed values (which is explained in the next section), therefore for the last set of tests for the manual batching we run the tests using 5 client machines. Unfortunately this is another small difference between the tests.

9.1.1. Metric measurement

In this subsection we explain in detail what was the difference for the measurement of metrics between Hector and the CQL 3 driver. We encountered some issues with this change, however we successfully overcame them. The main difference is in the way we record metrics. For the previous tests we kept the approach for measuring throughput and latency as simple as possible. This way, the additional overhead from the measurement of metrics is minimized. However, because this was an asynchronous client, some major changes had to be implemented for the way we measure the metrics on the client side.

**Algorithm 9.1 Client measured metrics using Hector**

```java
long start = System.nanoTime();
mc.write(columnFamily, timeline, timestamp, value);
long end = System.nanoTime() - start;
double difference = (double) end / 1000000.0;
```

For measuring metrics on the client side on the Hector version we used the code in Algorithm 9.1. As mentioned the implementation is very simple because we wanted to avoid interfering (performance wise) with the tests. In the code in Algorithm 9.1 the `mc.write()` method performs a single write. The method itself is not presented as it is not that relevant for this. Also we were measuring the metrics using OpsCenter also, so it was not the only source of metrics. While on the case of the CQL
3 driver because of the asynchronous calls we had to change the way the metrics were measured.

**Algorithm 9.2** Client measured metrics using CQL 3 driver

```java
ctx.start();
ResultSetFuture rsf = session.executeAsync(boundStatement);
Futures.addCallback(rsf, new FutureCallback<ResultSet>() {
  public void onSuccess(final ResultSet result) {
    ctx.done();
  }
  public void onFailure(final Throwable t) {
    log.info('Error during request: ' + t);
    ctx.done();
  }
}, executorService);
```

The code in Algorithm 9.2 shows the approach we took for measuring the metrics using the CQL 3 driver. It is important to note that the code in Algorithm 9.2 is within the `write()` method that we showed in Algorithm 9.1. The `ctx` variable is part of the way we measure metrics. It starts measuring the time when the `start()` method is called, until the `done()` method is invoked. This is part of the Metrics library[16], which we utilized for the measurement of metrics.

After starting the “timer” we execute the query asynchronously with the `executeAsync()` method call. The callback that is added to the `Futures` class is part of the Guava library[42] which is utilized by the CQL 3 driver. This is needed because it is an asynchronous call, the result set along with an `executorService`¹ are passed as parameters on the callback. When the response from the server is received the `onSuccess()` or `onFailure()` method is executed which stops the “timer”. This was the main change done in the client. A few more changes were done on the client mainly related to class inter-dependencies and some code refactoring. However, these changes are not that interesting to show.

### 9.1.2. Different scenarios

In this subsection the different data schemas will be explained from a source code perspective, specifically what was different between the scenarios for the data schemas.

¹[http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html)  [online accessed 04/01/2014]
9.1.2.1. Pre-tests

In this subsection the client changes for the different pre-tests are elaborated upon. These changes were only used for the pre-tests, in order to determine good values for the parameters for the actual tests. The first parameter is the number of column families to use for the tests. The other parameter is the size of each batch for the manual batching tests (c.f. sec.9.3.4). The types of tests are presented and explained in sec.9.2.

Column family pre-test: For this pre-test we passed as a parameter the number of column families to be used. Each client instance on start up created the column families (according to the argument passed). Then within each client instance we iterated over the available column families and performed one insert on each column family on a sequential order (e.g. starting from CF1, to CF2, ….). Besides this difference, the rest of the client parts remained the same as for the normal tests.

Varying batch size pre-test: For this pre-test we passed as a parameter the number of values to be stored in each manual batch. Besides this it was really simple to have a variable number of values per batch, we simply used the number of values as a check on a loop that populated each byte array with that many records.

9.1.2.2. Manual batching

For this, we first concatenated all of the values in a String, using commas to separate the different values. Then we assigned the String to a byte array and then we wrap the byte array with a ByteBuffer, since this is what Cassandra expects for blob values. One byte[] with values is created at the beginning of the test and it is re-used for each operation. This means that the values of each batch are not random, however if we created a new batch for each operation the performance of the client degraded.

For the timestamps we did not use the same approach. Since the batch size was known beforehand, we stored the timestamp of the first measurement in the batch. Because we are not using actual timestamps but simulated timestamps (e.g. just increase by 1 for the next millisecond), we can determine where each requested timestamp is stored. We do this by calculating the difference of the column keys (timestamps) and mapping the requested timestamps to the intervals of the differentials. For example, if we started storing timestamps from 100 and store 50 measurements per key-value pair, the timestamps on the column keys would be 100, 150, 200, etc. If the client requested the values between timestamp 130 and 280, we can determine that column keys 100-250 are required for this query.

The process for reading the results is exactly the opposite than the one just described. The other scenarios (metadata and timewidth) are essentially the same as
the ones for typed values (c.f. sec.9.1.2.4 and sec.9.1.2.5). The only difference was that the values being inserted are `byte[]`, however this did not have an impact on the code for the other tests.

### 9.1.2.3. Typed values

This is the most straightforward test that we performed. Straightforward in the sense that the structure and the logic was exactly the same as the tests we performed using Hector (c.f. sec.8.1.1).

### 9.1.2.4. Metadata

In this subsection the changes on the client for the metadata tests are described. The changes for metadata and timewidth (c.f. sec.6.3.2) were almost identical for typed values and manual batching. Metadata refers to the technique used to store and retrieve metadata, it is described in sec.9.2.

**No metadata**: The idea behind this scenario is that each client needs to determine the relevant row keys (as in the other scenarios with metadata) and perform a range query like all the other scenarios. A range query requests all the measurements between two timestamps (c.f. sec.6.2). So the main difference is that instead of querying a row as is the usual case, all of the row keys have to be scanned. This is achieved using this query: "SELECT DISTINCT * FROM <columnFamily>". Another important notice on this is that the query scanning the rows is synchronous. This is the case because in order to perform the actual range query, the relevant row keys have to be determined first. Therefore, we synchronously query the available rows and then asynchronously request the actual range of values. This is not efficient because we still have to determine the row keys on our client and the database also performs a row scan, which is a costly operation. However, this data schema serves as one extreme on the metadata scale (thus the no metadata name). Fig.9.1 shows a visual representation of the metadata setting.

**Metadata on the same column family**: This is the usual approach that we used for the tests with Hector also. On the same column family we have a row that is named metadata and we store the row keys of this column family in that row. So before performing the range scan we synchronously request the relevant row keys from this row. This is a more efficient approach compared to the no metadata technique. It only has to query a specific row within the same column family, which is much more efficient than scanning all rows. Fig.9.2 depicts a visual representation of the flow for this metadata setting.
9.1 Client differences

**Figure 9.1.** No metadata data schema

**Figure 9.2.** Metadata in same column family

**Metadata on a different column family:** This is very similar to the previous approach. The only difference is that instead of writing the row keys on a row in the same column family, we write the row keys for each sensor on a separate column family that is dedicated to the storage of the row keys for other column families. The only modification that we had to make is that on each row of the metadata column family, we also store the sensor name (column family name) to be able to determine to which column family the particular row belongs. The advantage of this is that even with huge amounts of data the size of the metadata column family will be small, which makes it easy to enable row caching. Row caching is configured on a per column family basis and it enables storing the rows of the particular column family in memory. This makes access to that data very fast. Fig. 9.3 shows a visual representation of this.
Secondary indexing: In this approach we just create a secondary index on the column family that we read data from. Then in order to perform the range query we enable the "ALLOW FILTERING" directive, without specifying a row key constraint. This forces the database to perform a scan of its own secondary indexes to find the appropriate values. The advantage of this scenario is that the logic implemented in the client is minimized. The client simply sends the column family and a starting and ending timestamp without specifying any row keys. The database handles this, while in the other scenarios we specify the row keys to prevent the database from scanning them (which is slow). In Fig. 9.4 we see a visual representation of this simple approach to make it clear.
9.1.2.5. Timewidth

For the timewidth pre-test we figured out how much space X records take in the database. We checked this using Cassandra’s `nodetool cfstats` command, which displays information about all of the column families in the database. We kept track of how many records are inserted and once we found out the value for 1 MB we just multiplied this value by the number of MBs that we wanted to store in each test. We verified that this calculation brings a result close enough to the specified size.

9.2. Test structure

The test structure is similar to the test structure that was presented in sec. 6.2. We are again using the write, read and mixed scenario for the different tests. The variations of this set of tests is presented in this section. Fig. 9.5 shows the structure of the tests.

![CQL test structure](image)

**Figure 9.5.:** CQL test structure

The top box shown in Fig. 9.5 represents the amount of measurements stored in each key-value pair. For these tests we used two different settings for this: typed values and manual batching. Typed values is the regular 1 timestamp-value pair for each key-value pair. We call it typed values because the measurements are stored as the variable types they are (String in our case). By manual batching we mean that for each timestamp multiple values are grouped together in the same key-value pair. For this we store the starting timestamp as the column key and multiple, concatenated values as the value part. The changes from a source code perspective were described in sec. 9.1.2.2. The type of values for the manual batching case is BLOB (Binary
Large OBject). For determining how many values should be stored in each batch we performed a pre-test which is presented in sec. 9.3.3. So the different values for this setting of the tests is typed values and batching. For each of the values (typed values and batching) the different data schemas vary in two attributes: metadata technique and timewidth.

The timewidth has been used throughout this thesis and refers to how many measurements are stored in each row. We experimented with different values for the timewidth to see how performance is affected. For this test instead of measuring the amount of data stored per row by the number of measurements, we measure it by the disk space used to store the measurements (e.g. 5 MB, 7 MB, etc). The changes from a source code perspective were described in sec. 9.1.2.5.

For each of these different parameters we run the write, read and mixed scenario that we also used for the previous tests. So these are the variations of the data schema that we tried and the performance of each data schema is evaluated on the basis of write, read and mixed scenarios. The reason for this change in the test structure is to be able to group the results in a nice and meaningful way. The different data schemas are expected to best suit certain use cases and not appropriate for some others. By this grouping we can categorize the results according to the type of values (typed or binary), the metadata used and the timewidth of each row.

For the mixed scenario for the typed values we were running the tests using 6 client machines. Therefore, to achieve a ratio of 75-25% we run 8 client instances in total. On two of the machines, we run in parallel 1 instance of the write scenario and 1 instance of the read scenario. This was the case because we did not have abundant client machines available in order to run 1 read scenario in a separate client machine. However, the ratio at which we run the instances was the same (75% writes, 25% reads).

For the manual batching tests, as mentioned already in sec.9.1 we had 1 less client machine, leaving us with 5 client machines for this test. For the mixed scenarios we run a total of 7 client instances. So again we run 1 write and 1 read instance in parallel on 2 of the client machines. This results in a slightly different ratio, roughly around 72% writes 28% reads. This is not a significant change for the results, the best option would be to run exactly the same setup as the previous, however as mentioned one of the client machines stopped working which we could not overcome in time for this thesis.

A final thing to note for the way we run the tests, is the number of times each test was run. As mentioned each scenario is only run once, since adding extra client instances does not necessarily add extra throughput. Therefore, in the beginning we decided to run each test twice with the exactly same parameters and number of client instances to ensure that the results are consistent with each other. We only did this for the column family pre-test presented in sec. 9.3.1. The results are similar with small differences that are not significant. Therefore, we decided for the rest of the tests to run each test only once, due to the limited time for completing this
thesis.

9.2.1. Metrics

As mentioned in sec. 9.1, unfortunately OpsCenter was not compatible with Cassandra 2.0 that we used for these tests. Thus, we only recorded the metrics on the client side. However, as mentioned in sec. 9.1, major changes had to be done in the way that we measure metrics due to the nature of asynchronous operations. For measuring the metrics we used Metrics[33], which is a library for measuring metrics in many different types of applications. For measuring the metrics we have mimicked the way in which Metrics is used to measure metrics in the small stress application\(^2\) that is shipped along with the CQL 3 driver. This was elaborated further in sec. 9.1.

So the metrics that we measure on the client side are:

- **Throughput**: The number of operations per second.
- **Mean latency**: The average of all latencies for the duration of the test.
- **Median latency**: This is the latency of the middle operation if we have all of the operations sorted in ascending order.
- **Standard deviation for latency**: Refers to how much above or below the mean latency the latency varied on average.
- **75, 95 and 99 percentile for latency**: These are similar to the median latency. They refer to the 75th, 95th and 99th operation if we have all of the operations sorted in ascending order.

9.2.2. Resulting diagrams

For the diagrams besides the usual operations over latency diagrams that we present, we are going to also show some bar charts for latency and for operations. The bar charts are showing the results for each type of test. By type we mean for example the test for typed values, the metadata tests for all different metadata settings. Another example is again for typed values, the timewidth tests for all the different timewidths that we tried. In the previous tests we were presenting the usual operations over latency diagrams on a per scenario setting. This was the case because each scenario was run multiple times, with increasing number of client instances. As mentioned, due to using the CQL 3 driver which is asynchronous it eliminated the need to run multiple instances of the same scenario, which results in having only 1 point for each scenario. Therefore, grouping the different data schemas together seemed to be more appropriate, in order to convey the most and detailed amount of information for the test results.

\(^2\)https://github.com/datastax/java-driver/tree/2.0/driver-examples/stress [online accessed 09/11/2013]
The bar charts are split between writes (write and mixed (write) scenarios) and reads (read and mixed (read) scenarios). This is the case because the numbers between the writes and the reads varies too much, which made the read results barely readable. So both for latency and operations diagrams the reads are separate from the writes.

Another note to take for the latency bar charts is that the standard deviation is included there. The scale at which metrics are measured in this diagram is in milliseconds. The standard deviation represents the deviation of the mean latency, which is not in milliseconds, but rather related to the actual measurement of the mean latency (which is in milliseconds). Therefore, the standard deviation should not be interpreted as milliseconds, but as the respective number with regard to the mean latency for the same diagram.

Furthermore, even after splitting the diagrams like this, sometimes the values varied too much which again led to the low values not being visible. This is why in some cases we have used a logarithmic scale for the values of the diagrams. The low values on the logarithmic scale can be distinguished easily, however high values are harder to determine. This is why in some occasions we also depict the actual values on the bars, to improve the readability and give the reader an accurate view of the results. The logarithmic scale and actual values are used on a diagram-to-diagram basis. On some of them it is easy to read the diagram without the logarithmic scale or the values, while on others it is harder. This is not standard, so it is done on a per diagram basis.

### 9.2.3. Test parameters

In this section the parameters used for the different tests are presented. Besides the attribute that the particular scenario tests, the rest of the parameters remain the same. Tab. 9.1 shows the parameters for the tests.

<table>
<thead>
<tr>
<th>Scenario type</th>
<th>Number of events</th>
<th>Timewidth</th>
<th>Replication factor</th>
<th>Write consistency</th>
<th>Read consistency</th>
<th>Timerange (min - max)</th>
<th>Query count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>3,000,000</td>
<td>100,000</td>
<td>3</td>
<td>Quorum (2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read</td>
<td>10,000</td>
<td>100,000</td>
<td>3</td>
<td>-</td>
<td>One</td>
<td>9,000-10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>Mixed (writes)</td>
<td>2,000,000</td>
<td>100,000</td>
<td>3</td>
<td>Quorum (2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mixed (reads)</td>
<td>10,000</td>
<td>100,000</td>
<td>3</td>
<td>-</td>
<td>One</td>
<td>9,000-10,000</td>
<td>10,000</td>
</tr>
</tbody>
</table>

**Table 9.1.:** Parameters for test scenarios

So these parameters are used for the pre-tests and the metadata scenario. The only scenario that slightly differs from this table is the varying timewidths scenario.
The column that changes, as expected is the timewidth column. For the timewidth tests, the timewidth is the different setting among the tests, so it is varying. The size used for the timewidth tests is shown at the tests. For the number of key-value pairs for the different timewidths, the process to figure out how many measurements correspond to the size that is tested was outlined in sec.9.1.2.5.

The number of operations for this set of tests is higher because the clients generated requests a lot faster, due to the clients doing asynchronous calls. When we tried to use the settings used for the tests performed with Hector, the test was too short. Also the number of operations varies for the mixed (writes) scenario, so that both the read and write client instances finish in a similar time. This is more related to the way we run the tests, to make it more convenient.

9.3. Test results

The results of the tests we performed using the CQL 3 driver are presented in this subsection. For these tests we used the CQL 3 driver version 2. This also required an upgrade of Cassandra to version 2.0 (from 1.2). The binary protocol interface, which we use for these tests was not changed that much from version 1.2 to 2.0. This was mainly done due to the additional functionality provided by the client with Cassandra 2.0.

In addition to the tests that we described in sec.9.2, some additional pretests are run to determine good values for some parameters. So in sec.9.3.1 the results of a pre-test for the number of column families to be used was run. Then, in sec.9.3.2 the results of the tests using the typed values are presented. Next, another pre-test for the number of values on each batch is shown in sec.9.3.3, followed by the tests for the manual batches in sec.9.3.4. Finally, a summary of the tests is presented in sec.9.3.5.

9.3.1. Number of column families pre-test

Before proceeding with the actual tests we decided to run a pre-test for the number of column families to be used in each test. The number of column families is important because in our data schemas each column family represents a sensor. If many column families degrade performance, then we would need to group multiple sensors within one column family, which would change the structure of our data schema. Another motivation for this pre-test were the findings of Jeroen Broekhuijsen, a TNO employee. His findings suggested that as the number of column families increases, the throughput of operations drops drastically. For this pre-test we only run the write scenario, since Jeroen’s tests also suggested that the number of write operations drops drastically. We were interested in the performance of writes with
regard to the number of column families. Therefore, the read and mixed scenarios would not provide us with very useful information.

![Graph showing write operations and average latency for varying number of column families.](image)

**Figure 9.6:** Write operations / average latency for varying number of column families

Fig. 9.6 shows the number of write operations per second, with regard to the average latency. The different series in Fig. 9.6, represent different number of column families (CF). Each series has 2 points because this test was run 2 times to ensure that we are getting consistent results. As we can see the difference between the two tests is marginal. The values for the same tests only vary a lot between 1 CF and 100 CF, where the difference is around 5000 operations per second. Even though this is a significant difference, we believe that this was simply coincidence. A possibility is that more compactions from the previous tests occurred on these two tests and that is why there is a bigger difference. However, for most of the tests the difference between the two tests is minimal, therefore we decided to only run each test once for the rest of the tests.

As we can see the throughput does not vary that much between the different tests. The maximum difference is around 7000 operations (from 5 CF to 1 and 100 CF), however these two were “unlucky” runs. A heavy background process also occurred, which resulted in this difference. Overall the results between the different tests vary by 2000-3000 operations per second, which is not a significant amount. As the number of column families increases, the average latency also increases (for 50 and 100 CF’s). However, the difference is not that big the latencies are still comparable. Next, Fig. 9.7 shows the metrics for latency for the first set of tests and after Fig. 9.8 the metrics for latency for the second set of tests.
9.3 Test results

The results in Fig.9.7 confirm what the results in Fig.9.6 show; The latency tends to go up as the number of column families increases. It is easily noticeable that the 50 and 100 CF’s results steadily are higher than the results of 1, 5 and 10 CF’s. The best indicator for this is the standard deviation, which is about 3 times higher than the case of fewer column families.

The latency metrics are verified by the results in Fig.9.8. As the number of column
families increases, the 50 and 100 CF’s result in an overall higher latency than the fewer CF’s. The standard deviation is again the best indicator for this to be noticed. For this test all clients had been set to use the same set of column families, depending on the number of column families for the test. The process was elaborated upon in sec. 9.1.2.1.

So for the outcome of this pre-test, it can be said that using fewer column families has a better performance than using a lot of column families, since latency tends to go up as a lot of column families are involved, even though the throughput difference is not that significant. For our tests we have decided to use one column family for each client instance for the writes (so 5-6 column families), which has the best results for the pre-test that we performed. Furthermore, using this schematic (1 client instance - 1 column family) was the simplest approach from a source code perspective, which is another reason for using this setup.

9.3.2. Typed value tests

After determining the number of column families to be used with the tests, we proceed with performing the actual tests. We start by presenting the results for the typed values test. By typed values we mean that each key-value pair holds one timestamp-value pair. The value is encoded according to its type, which for our tests was String. We first present the results for the metadata tests in sec. 9.3.2.1. These tests explore which is the most efficient technique to store metadata for our sensor data (c.f. sec. 9.1.2.4). Then we show the results for the timewidth tests in sec. 9.3.2.2. As mentioned already these tests were performed using six client machines.

9.3.2.1. Different techniques for sensor metadata using typed values

In this subsection the results of the tests for metadata are given. As mentioned already the different settings for the metadata are: No metadata, metadata in the same column family, metadata in a different column family and secondary indexing. First, the bar charts for the operations are presented, followed by the bar charts for latency and finally the usual throughput over latency diagrams are presented. The settings that we used for these tests can be found in sec. 9.2.3.

Fig. 9.9 shows the bar chart for the write and mixed (write) operations for the different setups for metadata. In general we did not expect to have big difference in performance for writes the type of metadata used. This is the case because the advantage of metadata comes into play for reads not for writes. We can see that the metadata in the same or a different column family have almost identical performance. It is interesting to see that with no metadata a little higher throughput is achieved. This makes sense, since the metadata means that some additional operations need to be performed for each request.
The secondary index performed a lot slower than the other 3 techniques. This is the case because Cassandra internally stores an index for each column. This means that a lot more indexes are maintained than in the case of manual indexing. On the manual indexing case we delegated almost all of the indexing part to the client, meaning that determining the relevant row keys was done on the client. When we created a secondary index on the timestamp, Cassandra created an index for each entry, which is quite inefficient. This is why the performance for secondary indexes dropped so much. As the developers from Datastax suggest, the more unique values that exist in a particular column the more the overhead you will have on average, to query and maintain the index. However, it was still interesting to examine its performance as the other “extreme” (compared to no metadata).

Moreover, when using the secondary index it was not really feasible to perform a test for the other scenarios. For the read and mixed (read) scenario, we performed the request specifying the start and end timestamps without specifying any row keys. Algorithm 9.3 shows the query we used when using the secondary indexing technique.

Algorithm 9.3 Secondary indexing query

```sql
SELECT * FROM <sensorID>
WHERE timeline IN ? AND timestamp >= ? AND timestamp <= ?
ORDER BY timestamp ASC LIMIT <limit> ALLOW FILTERING;
```
The values in angle brackets represent values that we define during runtime. We used bound statements for executing all of the queries, because the queries are the same, only the values change. Bound statements are types of queries that are “prepared”, only the values vary between different queries. These values are represented by the question marks in Algorithm 9.3. Bound statements have better performance than regular queries, which is why we decided to use them. The ALLOW FILTERING statement of CQL 3, forces Cassandra to perform a scan on its secondary indexes to find the values. Yet, because of the number of different unique indexes, this was not an efficient way to perform the request. Therefore, the majority of the requests got timed out, which led us to skipping the read and mixed scenario for secondary indexing, since it was not really feasible.

![Figure 9.10: Read operations for read and mixed (read) scenario](image)

Fig. 9.10 shows the other bar chart for the same set of tests, but for read and mixed (read) scenario this time. We can see that the metadata in the same or a different column family do not differ a lot with regard to the number of operations per second. The reason for this could be that our data set was not that large (it is in the order of MBs). In a larger working data set in the order of GB and higher, the metadata in a different column family could have a bigger impact with row caching on the metadata column family. As expected the no metadata method is a lot slower. This is the case because many indexes have to be internally scanned by Cassandra, while our manual indexing approach was more optimized for the particular data schema.

The no metadata approach performs best for writes, which makes it applicable to cases with lots of writes and few reads. The metadata in the same column family
perform best for reads. Also, the difference in throughput with no metadata is not significant, which makes this technique applicable to more cases. The metadata in a different column family performed very similar to the metadata in the same column family, their performance is very close. The secondary indexing technique is not appropriate for time series data because of the uniqueness of values. Secondary indexing is more performant in cases where some repetition of data exists.

Fig. 9.11 shows the latency metrics for the write and mixed (write) scenarios. As we can see the mean latency for the mixed scenario is higher than the latency for the write scenario. This makes sense since reads are also happening at the same time, which stress the database more because disk seeks have to be made for reads. This trend is not visible when using no metadata, however this is the case because of the very slow performance of reads in the no metadata scenario. The secondary indexes have by far the highest latency and are not really performant for this kind of data set as already mentioned. This is also verified by the fact that the read and mixed scenario were not really feasible for the secondary indexing.

Another thing to note is that the latency overall is a lot higher than the tests we performed using Hector. This is the case mainly because the requests are asyn-
chronous. Asynchronous requests by their nature have a higher latency, since the order is not guaranteed. But if the high latency is a problem for a particular use case, synchronous requests can also be issued easily, yet the throughput will be lower than the results in Fig. 9.9.

![Latency stats for read and mixed (read) scenario](image)

**Figure 9.12.** Latency stats for read and mixed (read) scenario

Fig. 9.12 shows the latency metrics for the read and mixed (read) scenarios. As we can see the metadata in same and a different column family have very similar latencies. Only the 95th and 99th percentiles of metadata in a different column family are higher, however these are the extremes, the mean latencies are quite similar. The standard deviation for all of them is quite high, which means that latency tends to vary quite a bit between the requests in each test. The no metadata scenario has the highest latency of all and this makes sense since for each request a full row scan had to be performed from the database. As already pointed, this is inefficient compared to only querying one row for the metadata.

Finally, Fig. 9.13 shows the usual diagrams that we showed for the Hector tests also. It shows the number of operations with regard to the average latency for the three different scenarios, namely: write, read and mixed scenarios. The data is the same as the data shown in the previous diagrams in this subsection. The extra value
9.3 Test results

Fig. 9.13 adds is that it shows the throughput with regard to the latency in a single diagram, while in the other diagrams latency and throughput are separate.

In Fig. 9.13 (a) we can see that all three different settings had similar performance, but this was anticipated since the metadata is expected to mostly affect the performance of reads. In Fig. 9.13 (b) we can see that the tests with the manual indexing performed similarly, the metadata in a different column family performed slightly worse. However, the difference in performance is negligible. In Fig. 9.13 (c) we see the results for the mixed scenario (both writes and reads). The throughput for the reads is not really visible, but these numbers are clear in Fig. 9.10. For this we see that the no metadata performed the best with regard to write operations out of the three. However, it also performed the worse with regard to reads. The data schemas with the manual indexing performed very similarly for this one also.

The no metadata approach could be applicable to a use case that there are few to no read requests and the latency can be tolerated. In such a case using a metadata technique is not worth it, since the advantages of metadata are applicable for reads.
For other use cases with a more balanced load between writes and reads, using the metadata within the same column family performs best. By balanced load we mean still leaning towards writes (e.g. 75% writes, 25% reads), as mentioned Cassandra excels at writes, not at reads. This technique is also good for applications that do real-time processing, due to the low latency. The metadata in a different column family performs very similarly, just slightly lower. The metadata in the same column family performed best overall, both in terms of throughput and in terms of latency. This is why we chose to use the data schema with the metadata in the same column family for the timewidth tests that are presented next.

9.3.2.2. Timewidth for typed values

In this subsection we present and discuss the results of the tests for typed values, for varying timewidths. As mentioned the data schema that we decided to use for this test parameter is the one with the metadata in the same column family, because it performed slightly better for the metadata set of tests. The parameters for this test are shown in sec.9.2.3. The timewidth is not 100,000, but the number of measurements required to reach the indicated size for each row, as explained in sec.9.2.3. This is the variable between the tests and we measure it according to the space it takes on disk in MB. The method used to measure the size was described in sec.9.1.2. The different timewidths that we used were: 5, 7, 10, 13, 15 and 20 MB per row.

![Figure 9.14. Write operations for write and mixed (write) scenario](image)

Figure 9.14.: Write operations for write and mixed (write) scenario

Fig. 9.14 shows the operations per second for the write and mixed (write) scenarios.
As we can see the performance for 5 MB is slightly higher than the respective operations when using a timewidth 100,000 (c.f. Fig. 9.9). However, the performance for the write scenario steadily drops by about 2,000 operations on each timewidth increase. This trend stops at 13 MB and increases again slightly for 20 MB. This is not a big difference (comparing 20 to 13 MB), again a heavy background process was taking place on the server side (e.g. compaction), which resulted in the slightly worse performance for 13 and 15 MB. For the mixed (writes) scenario the performance of all different timewidths is quite similar. The 5 and 10 MB have the highest performance, but the difference is negligible.

Fig. 9.15 depicts the performance of reads for the read and mixed (read) scenarios. In this diagram the exact opposite trend from Fig. 9.14 is noticed; As the timewidth increases the operations per second increase. The highest increase can be noticed up until 13 MB, afterward the additional throughput gained is not that much compared to the previous increases. For the mixed (read) scenario the same upward trend is noticed until 10 MB, which achieves also the highest throughput. The performance for 13 MB slightly drops, then increases to the level of 10 MB for 15 MB and finally drops gradually for 20 MB at the mixed scenario. The size of the timewidth seems to have a direct impact on read performance, the bigger the timewidth, the better the performance in terms of throughput. The peak point seems to be 15 MB since for 20 MB, the performance of reads on the mixed scenario drops tremendously.

With regard to throughput, the results are quite obvious. For a write-heavy case with few reads using a small timewidth provides the highest write throughput. If on
the other hand, we are dealing with a case where the number of reads per second is relatively big. Using larger timewidths performs best. The 5 MB timewidth performs best for writes, the 15 MB timewidth best for reads and the 10 MB timewidth performs best in the mixed scenario. In conclusion, small timewidths favor write throughput, while large timewidths read throughput.

**Figure 9.16:** Latency stats for write and mixed (write) scenario

Fig. 9.16 shows the latency stats for write and mixed (write) scenario. As we can see, the latency is rather evenly distributed among the different timewidths. The standard deviation is quite high with regard to the mean latency, but it is comparable for all different timewidths. This is also noticeable by the 95th and 99th percentiles which are both pretty high with regard to the mean latency. Overall, the latency for writes is quite stable, which means that the different timewidths do not affect the latency that much.

Fig. 9.17 depicts the latency stats for read and mixed (read) scenarios. The results are similar to what the results for operations showed in Fig. 9.15, as the timewidth increases, the latency decreases (lower is better for latency in contrast to throughput). However, the standard deviation for all timewidths is quite high. Another thing to note is that the performance for the mixed (read) scenario is extremely high compared to the rest. This is the case because writes are also taking place, which place additional load to the database. Another observation is that a heavy background process (e.g., compaction) was taking place at the time of the mixed scenario. This is clear both from the throughput of reads (cf. Fig. 9.15) and by the latency of reads for the mixed scenario as shown in Fig. 9.17.
9.3 Test results

Figure 9.17.: Latency stats for read and mixed (read) scenario

Figure 9.18.: Operations / average latency for varying timewidth
Fig. 9.18 shows again the combination of operations per second over latency. On Fig. 9.18 (a) we can see the write operations with regard to the average latency. The 5 MB timewidth got the highest throughput, however also the highest latency. The 7 MB resulted in slightly less operations, yet the latency is also the lowest. The difference in latency is minimal though, it does not differ that much between the different timewidths.

In Fig. 9.18 (b) we see the read operations with regard to the average latency. As on the above diagrams the higher the timewidth, the better the throughput and the latency. So for reads, large timewidths perform better both with regard to operations and latency. The highest throughput is achieved by 15 and 20 MBs per row. The latency also is the smallest for those two timewidths.

Finally, Fig. 9.18 (c) shows the results for the mixed scenario. Unfortunately because too many settings were used for this test, the diagram is not that readable. But it is clearly visible that with regard to write operations all of the different timewidths performed almost the same. For reads only the latency is visible, because the scale is too large. The highest latency is for 5, 7 and 20 MB. However, for the 20 MB test as mentioned a heavy background process was taking place in the background.

Even though the highest throughput for reads is achieved by the timewidth of 20 MB, the fact that the performance greatly degraded for the mixed (read) scenario indicates that such a big size is unstable. The next highest throughput for reads is achieved by 15 MB, which also performed consistently for the other tests also. The performance of 15 MB is good both with regard to latency and throughput. Nonetheless, as we can see the big timewidths are more suitable for read intensive scenarios. With regard to write throughput, on the write scenario lower timewidths perform better (5 and 7 MB). Therefore, it can be concluded that for write intensive use cases a lower timewidth would be a better candidate and for read intensive use cases a larger timewidth would be most beneficial.

One note on this conclusion, is that the difference in writes from 5 to 15 MB (c.f. Fig. 9.14), is not that big compared to the increase in reads. It is about 62,000 for 5 MB and around 52,000 for 15 MB. While if we compare the gain of the reads from 5 to 15 MB (c.f. Fig. 9.15), the difference has more impact. It is close to 1,000 for 5 MB and approximately 2,300 for 15 MB. This does not alter the conclusion mentioned, but it is mentioned to signify that the gain in reads from timewidth has a higher impact, compared to the number of write operations “lost”.

9.3.3. Number of values per batch pre-test

Before running the tests using manual batching we decided that a pre-test is needed, in order to determine the most appropriate size for each batch. The outcome of this pre-test will be used for the tests performed next, using manual batching. So we tried 5 different parameters for the batch size: 50, 100, 150, 200 and 300 measurements per batch. We believe that these numbers are wide enough to provide a good view
9.3 Test results

of the best batch size. In practice people use a number close to these parameters also. For this pre-test we decided to run the write and read scenario to see how performance is affected according to the varying number of measurements. The mixed scenario was skipped, since this is not a normal test and the mixed scenario would not affect our decision on which number of measurements per batch will be used. For the purposes of this pre-test, the write and read scenario provide a pretty clear picture of the performance. The settings that we used for these tests are shown in sec.9.2.3. For this pre-test we used the metadata within the same column family setup for metadata. We chose to use this since it was our initial data schema and all of the other pre-tests were run using this data schema.

![Figure 9.19: Write operations / average latency for varying number of measurements per batch](image)

Figure 9.19: Write operations / average latency for varying number of measurements per batch

Fig.9.19 shows the number of operations with regard to latency. Because it is batching though, this is not the actual number of measurements stored. Each batch operation is represented as 1 operation in these results, which means that the number indicated in Fig. 9.19 needs to be multiplied by the measurements per batch for that particular test. The clear indication from Fig. 9.19 is that as the batch size increases the operations per second, steadily drops. This makes sense since larger batches are stored, which takes more time and slows down the database. The number of operations for 300 measurements per batch already has more than halved, with regard to the number of operations for typed values. Another interesting note for this pre-test is that as the batch size increases, the number of timed out exceptions for requests also increases. This also makes sense, since each operation has to transfer a larger amount of measurements, which makes the storage process slower.
Next, Fig. 9.20 shows the amount of actual measurements stored in each test run. As mentioned earlier, this number is simply the operations per second multiplied by the number of measurements stored in that particular case. This number is steadily increasing, however the additional gain between each different setting slowly diminishes. Especially after 150 measurements per batch we can see that the additional measurements stored decreases a lot. The additional gain diminishes after some point because each request to the database holds more data. This causes the transferring and processing of requests to become slower, which results in diminished gains.

Fig. 9.21 shows the number of read operations for the varying number of measurements per batch. As we can see the number of operations for reads drastically increases compared to the number of operations of reads for the typed values (c.f. Fig. 9.13 (b)). We can see that the best performance is achieved by 150 measurements per batch.

However, the difference in the number of operations between the varying number of measurements per batch is not that high. All the different parameters increase the throughput for reads, because in terms of how Cassandra works less key-value pairs of larger size are requested. The size of the value influences throughput and latency for writes (c.f. Fig. 9.20). The impact for reads is smaller though, because storing multiple values in one key value pair already eliminates most of the overhead.
9.3 Test results

Figure 9.21: Read operations / average latency for varying number of measurements per batch

Figure 9.22: Latency stats for writes for varying number of measurements per batch

Fig. 9.22 shows the latency for the write scenario of this pre-test. We can see a clear trend of the latency going up as the number of measurements per batch increases. This makes sense since each operation stores a record of larger size in each increase in the measurements per batch, which causes the operation to be slower. Especially after 150 measurements the latency drastically increases. Also the standard devia-
tion of the latency is very high for all batch sizes. The 95th and 99th percentiles for 200 and 300 measurements are very high, which also explains the reason for the increasing timed out exceptions that we observed during those tests.

![Latency stats for reads for varying number of measurements per batch](image)

**Figure 9.23.** Latency stats for reads for varying number of measurements per batch

Fig. 9.23 shows the latency metrics for the read scenario of this pre-test. As we can see the latency among the different batch sizes remains rather low. This is the case because we are still requesting 9,000-10,000 measurements, however each record stores more than 1 measurements. This translates to Cassandra having to fetch less records, but larger in size. This eliminates some of the overhead which results in this improved performance for reads (compared to typed values).

The results of this batch size pre-test, prompt to the use 150 measurements in each batch. This size achieved the best performance for reads and with regard to the actual measurements stored in the write scenario, after 150 measurements per batch, the additional measurements stored on each test decreases. The latency for the write operations is quite high for this size, however it is not that much of a problem when taking into account the advantages that were just mentioned.

### 9.3.4. Manual batching

In this section we present the results of the tests using the manual batching technique. As described in sec. 9.1.2 for manual batching we create byte arrays that contain concatenated measurements, instead of 1 measurement per key-value pair. After
running the pre-test for varying number of measurements per batch (c.f. sec. 9.3.3), we determined that the optimal size for batches is 150 measurements per batch.

9.3.4.1. Different techniques for sensor metadata using manual batching

In this subsection we present the results of using the different settings for the storage of metadata. The different settings are the same as they are for the metadata tests for typed values. In brief the different settings are: no metadata, metadata in same column family, metadata in different column family and Cassandra’s secondary indexing. The settings that we used for these tests can be found in sec. 9.2.3.

![Figure 9.24.: Write operations for write and mixed (write) scenario](image)

Fig. 9.24 shows the number of operations per second for the write and mixed (write) scenarios. The operations are in terms of Cassandra operations, which means that the number of actual measurements stored is what is displayed in Fig. 9.24, multiplied by 150. We chose not to display the actual number of measurements because the only difference is the scale, the diagram itself is exactly the same, which makes sense since the number of actual measurements stored is dependent on the number of operations (in Cassandra terms). The different settings for metadata did not have any impact on the performance of writes, however this makes sense since the metadata mostly affects the read performance.

The secondary indexing method again had very similar results as for the typed values. Even though the performance in terms of actual measurements stored is better than the case of typed values, the performance is about half of what it is for
the other metadata settings. We hoped that the fact that less columns are created (because more measurements are stored in each key-value pair), would make the secondary indexing technique more performant. Unfortunately this is not the case as the results in Fig. 9.24 show. Also the secondary indexing technique again was not able to finish the read and mixed scenarios, which is why it is not present in the respective diagrams. The reasons are the same as for the typed values. Even though, less indexes had to be maintained (due to batching multiple values together), the performance is not good because of the number of different unique indexes.

![Figure 9.25: Read operations for read and mixed (read) scenario](image)

Fig. 9.25 presents the results for different types of metadata for manual batching, for the read and mixed (read) scenarios. The highest performance is achieved when storing the metadata in a different column family. This can be accounted to the fact that for each column family is stored permanently in one SSTable\(^3\). The size of the SSTable is quite large because of the batching technique which increases the size of each row. This causes the seeks for the metadata row within the column family slower. While on the case that the metadata is stored in a different column family, the size of the SSTable is smaller, therefore it results in more efficient reads. So in this scenario (batching) storing the metadata in a different column family performs better. When using no metadata performance drastically drops, as was the case for the previous tests. This again verifies that an indexing scheme is very useful for time series data.

\(^3\)http://wiki.apache.org/cassandra/ArchitectureSSTable [online accessed 17/11/2013]
9.3 Test results

With regard to write throughput, metadata in a different or the same column family and no metadata performed very similarly. Therefore, the read throughput is what will determine the best metadata technique with regard to different metadata techniques. For the reads the clear winner is metadata in a different column family. In both read and mixed (read) scenarios it performs better. So for the majority of use cases, from a throughput perspective the metadata in a different column family is clearly the best option.

Fig. 9.26 shows the latency metrics for the write and mixed (write) scenarios for different metadata using manual batching. The mean latencies for all the scenarios are quite comparable. However, all of them are pretty high. It makes sense to have higher latency when doing manual batching because each record is a lot bigger in size, compared to the typed value tests. Also these are the results for write and mixed (write) scenarios, which were not expected to affect performance on different metadata settings heavily anyway. Again, the secondary indexing has the worst performance from these different metadata settings. The reason for this is the same as when discussing the results of the write throughput scenarios (c.f. Fig. 9.24). Another thing to note is that the standard deviation for the mean latencies is very high, which gives a good indication of how slow the write operations are.

Fig. 9.27 presents the latency metrics for the read and mixed (read) scenarios for different metadata using manual batching. Again the metadata in the same or in a different column family have the best performance (compared to using no metadata which in all cases performed worse than when we use our indexing method). The metadata in a different column family performs better than the metadata within the
Figure 9.27: Latency stats for read and mixed (read) scenario

same column family. This can be accounted to the SSTable sizes as explained when discussing Fig. 9.25. Again the worst performance is on the no metadata setting, which is consistent with the previous tests.

The results for latency have the same trend as the diagrams for throughput. For the write and mixed (write) scenarios all of the metadata techniques, except for secondary indexing, perform similarly. So again, the difference is mainly in reads. For reads, again the metadata in a different column family technique performs the best. So the same advice applies from a latency perspective; Metadata in a different column family performs best.

Fig. 9.28 shows the usual operations over latency for all three scenarios. In Fig. 9.28 (a) the performance of write scenario is shown. We can clearly see that all of the different settings for metadata (except for secondary indexing), perform very similarly both in terms of operations and latency. Again this makes sense, since the metadata setting is expected to affect the performance of reads and not writes that much.

In Fig. 9.28 (b) we can see the performance of the different metadata settings for the read scenario. It is clear that the metadata in a different column family has the best performance, both in terms of operations and latency. The metadata in the same column family performs worse for the manual batching case. The reason for this was presented in the previous diagrams.

The write performance for the mixed scenario, which is shown in Fig. 9.28 (c), is very similar among the different metadata settings. For this, it was already explained
9.3 Test results

that the metadata does not have a high impact on the performance of writes. The performance of reads for the mixed scenario is not very visible in this diagram because the scale is too large (compared to the number of writes per second). However, as we saw in Fig. 9.25 the metadata in a different column family performed better than the other two settings for metadata.

So for the manual batching technique the clear winner is the metadata in a different column family. This was both in terms of throughput and in terms of latency. Therefore, for the timewidth tests of the manual batching we are going to use a different column family to store the metadata.

9.3.4.2. Timewidth for manual batching

In this subsection we present the results for manual batching when using a varying timewidth. Varying timewidth, means varying number of measurements stored in
each row. The overview of the code used for this test is described in sec. 9.1.2. The settings that we used for these tests are shown in sec. 9.2.3.

![Figure 9.29: Write operations for write and mixed (write) scenario](image)

Fig. 9.29 shows the performance with regard to operations per second for the write and mixed (write) scenarios, for varying timewidths using the manual batching technique. The lowest performance for writes was observed when using a timewidth of 7 MB. The difference from other timewidths is about 1,000 operations. As a number this is small, however it should be taken into account that each operation stores 150 measurements. So it has a little more impact than what 1,000 would mean for typed values (1,000 is negligible for typed values). Marginally the best performance was achieved by the timewidth of 13 MB per row. However, the difference is very small compared to the other timewidths. Overall all of the timewidths performed almost equally for the write scenario.

Interestingly, for the mixed (write) scenario there is a higher fluctuation. In this case 5 and 20 MB have the highest performance by a small difference (around 1,000 operations more). Another interesting thing is that 13 MB performed the worst for the mixed (write) scenario, while it had the best performance for the write scenario. The difference for 13 MB is slightly more compared to the 5 and 20 MB (around 2,000 operations less). However, seeing all the rest timewidths performing slightly better, this could be accounted to a Cassandra compaction taking place at the time of the mixed scenario for 13 MB.

Fig. 9.30 shows the operations for the read and mixed (read) scenarios, for varying timewidths using the manual batching technique. As we can see the throughput steadily increases until 13 MB and after that it suddenly drops. The difference between the minimum and the maximum is not that high (around 600 operations per second), however this difference is still quite high if we take into account that...
9.3 Test results

the overall throughput for reads is not that much. So the best performance for reads is achieved by 13 MB, followed by the previous settings.

For the mixed (read) scenario, the performance of the different timewidths does not vary that much. All of the timewidths performed about the same. The timewidth of 13 MB again performed slightly better, but the difference is marginal. So for a read-intensive use case, using the timewidth of 13 MB appears to be the best option with regard to throughput. On the other hand, for a write intensive scenario interestingly, 5 and 20 MB seems to be the most suitable candidates. 5 MB performs slightly better in all scenarios, except write. Therefore, the 5 MB is the best suggestion for a write heavy use case, with regard to throughput.

Figure 9.30.: Read operations for read and mixed (read) scenario

Figure 9.31.: Latency stats for write and mixed (write) scenario
Fig. 9.31 shows the latency metrics for varying timewidths. As we can see the latency among all the varying timewidths, does not differ that much. Very slight differences exist, which does not allow us to reach some conclusion for the latency of writes for varying timewidths. Again as in sec. 9.3.4.1 the latency overall and the standard deviation are very high. This is for the same reason as described in sec. 9.3.4.1, more values per batch, means that each operation is slower.

An interesting thing to note is that the performance of write and mixed (write) scenarios are almost identical, while in the previous cases the latency for the mixed (write) scenario was almost always higher than that of the write scenario. This could be attributed to the fact that reads seem to perform better with the batching method. Since the reads perform better, it also means that the database can devote more resources to the write operations. Also the operations overall took longer because of each operations larger size, which helped in evening out the difference in latency between the write and mixed (write) scenarios.

Figure 9.32.: Latency stats for read and mixed (read) scenario

Fig. 9.32 shows the latency metrics for read and mixed (read) scenarios. The latency is steadily higher for the mixed (read) scenario, however this was the case for almost all of the previous read and mixed (read) tests also. It makes sense since the database is busy also with writes, which results in less operations and higher latency per operation. The performance of different timewidths for the read scenario is very close between different timewidths. The timewidth of 13 MB has slightly lower latency than the others, but the difference is really marginal. For the mixed (read) scenario again the timewidth of 13 MB has the best performance. The difference from other timewidths is also more obvious in the mixed scenario.

Fig. 9.33 shows the usual throughput over latency diagrams. Fig. 9.33 (a) shows that 13 and 20 MB have the best performance. However, the difference is marginal (around 200-300 operations per second). The latency between the different timewidths
9.3 Test results

Figure 9.33.: Operations / average latency for varying timewidth

does not vary that much. Therefore, all of the timewidths performed similarly with regard to write operations per second, except for the 7 MB timewidth which has a bit larger difference of about 1,000 operations. But still this is not a huge difference.

In Fig. 9.33 (b) we can see that the timewidth of 13 MB has the best performance both in terms of throughput and latency. Hence, the 13 MB timewidth is marginally the best choice for reads. The timewidths of 5, 7 and 10 MB also performed similarly, however with slightly higher latency.

Finally, in Fig. 9.33 (c) we see the performance for the mixed scenario. Unfortunately the results for throughput and latency for mixed (read) are not clearly visible, but they were presented in detail in Fig. 9.30 and Fig. 9.32. We can see that in terms of operations per second, all of the different timewidths have similar performance. The latency varies a little bit for the writes and more for the reads. The best latency is achieved by 13 and 15 MB in the case of reads. The latency for writes between different timewidths does not vary significantly.

So to summarize the results of the tests for the manual batching, the 13 MB
timewidth seems to perform best with regard to read operations. This is both
in case of mixed (read) and read scenarios. The timewidth of 13 MB also performed
reasonably well for the mixed scenario, it had the highest performance for the mixed
(read), but also the lowest for mixed (write). However, the difference in terms of
operations was around 2,000 lower than the maximum for the mixed (write), which
is not that much if we take into consideration its performance for reads. With
regard to metadata usage, storing the metadata in a different column family and
then enabling row caching for that column family performs the best. Overall the
timewidth of 13 MB has the best performance. With the exception of the mixed
(write) scenario, it performed the best in all other cases.

9.3.5. Tests summary

In the previous sections we have run different test scenarios for a set of carefully
chosen data schemas. We have noticed different behavior from a performance per-
spective between the different data schemas. The results and tendencies of different
data schemas are discussed in this section. The first features to discuss lay in the dif-
ference of behavior in a case of serialized and unserialized data. The manual batching
technique, moves the serialization process to the client side, since the client has to
serialize the batches and on retrieval the client also has to deserialize them to turn
them to readable data. While in the case of typed values, the data are transferred
carrying their value type information with them.

These two different data schemas could be applied to different use cases. For example
in a use case where a lot of different parties are storing data and performing a lot
of reads on the meantime, it is a better option to leave the database handle the
serialization, therefore using typed values. However, the latency in the case of
manual batching for reads is a lot better. As already explained this is mainly due
to the fact that less columns of larger size have to be retrieved, which results in
reduced latency. Yet, the client also has to serialize and deserialize the data. If
this is acceptable for a use case, then the manual batching method performs better.
Nonetheless, if for example, a company delivers the storage of time series/log data
as a service to clients, it could be the case that the manual batching approach is not
suitable for the clients. This is simply a matter of the particular use case. However,
if possible it is advised to prefer the manual batching technique, from a throughput
perspective.

Another tendency in performance to take into consideration lays in the differences
between manual batching and typed values. The mean latency for manual batching
for write and mixed (write) scenarios is 2-3 times the latency of these scenarios
for typed values. So another distinction for a different use case is the latency of
writes. For use cases that do not immediately need the data, the manual batching
approach is more performant. Yet, if the latest data need to be accessed as soon as
possible, the manual batching approach is probably not the best approach. Besides
9.3 Test results

the latency, due to the large size of each record, a lot more timed out exceptions are also thrown, simply because it takes too much time to store the data at the rates that we were querying the database. A timed out exception does not necessarily mean that data is lost. This just needs to be handled on the client side, to re-send the timed out request. But still for a use case that the data have to be accessed immediately (e.g. real time analysis of the data), using typed values performs better (latency wise).

In a use case where there is a massive amount of data to be stored and not that many concurrent reads (in comparison to writes) are required, the manual batching is a better approach. The fact that we store multiple measurements within each batch, drastically increases the amount of measurements stored as shown in sec. 9.3.3 and sec. 9.3.4. This approach also performs well for reads, but as already mentioned the client has to serialize and deserialize the data, which is an additional concern on the client side. Optimally, a user would expect the database to take care of such issues. So the data schema to pick highly depends on the use case, which was quite evident from the beginning. In almost all cases of articles for data modeling using noSQL databases, it is always stressed to consider the requirements of each particular use case and optimize the schema for that particular use case. Optimization still needs to be performed for the particular use case, but with the results of this thesis, a user has a better idea on how to proceed with the storage of time series data, what to look out for and where to focus the optimization process on.

Another suggestion is that in order to get the most out of performance, an asynchronous client is important. If there is not such a client even investing the time to develop one could be worth the effort as we also saw from OpenTSDB[21], where one of the developers mentions exactly this, that an asynchronous client helps a lot with the load testing. But besides load testing, in a production environment if a synchronous client is used, multiple client machines will need to perform the requests. For some use cases this might not be a problem (e.g. a server for each area gathers the measurements from nearby sensors and stores them), but this is not always the case. Furthermore, even though the defaults for each database usually are reasonable enough, the configuration should be tuned to meet the requirements of the particular use case. We saw this in practice with the tests for Cassandra (when using Hector). Another important point is that in order to increase performance, many times the internal mechanisms of the database need to be understood, to effectively change the settings/data schema to increase performance.

9.3.5.1. Suggestions for data schemas with regard to different use cases

For summarizing the results of the tests that we performed using the CQL 3 driver, the different combinations that make up each data schema are grouped according to different use cases that they are most suitable for. The use cases need to be as generic as possible, therefore we have chosen to form different use cases according
to their requirements with regard to the database. Each use case has 3 different attributes that characterizes it: number of writes, number of reads and latency.

For the amount of writes, the possible values are high or low writes. High and low are not very specific and leave room for different interpretations. However, it must be taken into account that noSQL databases, and in particular Cassandra that we performed the tests, is very efficient for the storage of data, however not that much for the retrieval of data. So for writes, the term high refers to truly massive amounts of data. Putting a specific number on this does not make sense, since Cassandra is highly scalable (adding extra commodity hardware will increase the total throughput of the cluster). Low amount of write operations on the other hand refers to slightly lower requirements for writes. The only definite thing is that for the amount of writes, traditional (relational) databases must be falling behind the required performance (both with regard to high and low amount of writes). Otherwise, for that particular case traditional databases could be used, which are more standard and proven.

For the amount of reads again the same concept as writes is applied: high and low amount of reads. The high and low amount refers to the requirements for reads of the particular use case. But Cassandra is not as performant for reads as it is for writes. It performed reasonably for the range requests that we performed, however with tuning it is possible that relational databases can achieve the same performance if not more. The comparison of relational and noSQL databases could be itself a different type of tests, yet this is beyond the purpose of this thesis.

Finally, each use case is described by its latency requirements. For the latency requirements we are going to use low and high settings. If a particular use case needs to access the latest data as soon as possible, then latency needs to be as low as possible. This is the most demanding case with regard to latency requirements and represents the high latency requirement. On the other hand, if an application mostly deals with historical data analysis, high latencies can be tolerated. This case resembles the low latency requirements.

Fig. 9.34 shows the six different types of use cases that we have identified, along with the suggested data schema settings for the particular use case. The byte arrays in Fig. 9.34 denotes the manual batching technique. More use cases could be created (by using different combinations of the different settings), however they do not really make sense for Cassandra. By they do not make sense we mean that they do not define the most distinctive features of noSQL databases with regard to SQL databases. Relational databases have some issues that have been pinpointed throughout this thesis, however they have been used for many years, they are widely adopted, have a uniform way of access and provide better support for ad-hoc queries. Therefore, before adopting a noSQL database, ensure that there is a good reason for adopting it. Each use case depicted in Fig. 9.34 is further elaborated next. For a quick overview of which schema best suits which use case, we created 2 tables that compare the different data schemas on the scenarios. These tables can be found in
sec. A.4, the interested reader should also have a look there. The data essentially is the same as the ones presented in this section, but the tables make it easier to compare the different data schemas on a one to one basis. The use cases are described further in more details.

**High writes, high reads, high latency:** This is the most demanding use case. For this case we assume that the latest data need to be available as soon as possible, which makes the typed values data schemas a better candidate. This is the case because the latency for writes in the manual batching is very high. For the metadata used with this schema, the suggestions that we followed for our tests are also applicable. So for typed values storing the metadata within the same column family is advisable. For typed values the 15 MB timewidth seems to have the best performance in total. It has lower writes than the smaller timewidths, but it achieves more reads than the smaller timewidths.

**High writes, high reads, low latency:** This is pretty much exactly the same as the previous use case, however we assume that it is not that crucial to have the latest data available as soon as possible. So some delay before being able to display the latest data can be tolerated. So in this case the manual batching approach is a better candidate, since the higher latency on writes can be tolerated. The metadata schema should be the same as in the case of our tests, which means storing the metadata in a different column family. If the hardware
allows it, row caching should also be enabled for maximum performance on reads. For manual batching, the timewidth of 10 MB has the best overall performance as can be noticed in sec. A.4.

**High writes, low reads, high latency:** For this use case, the number of writes surpasses the number of reads by a lot. The high latency signals that the latest data need to be available as soon as possible. Therefore, again in this case the typed values data schemas are a better candidate since overall the latency for typed values is lower than for manual batching. The suggested metadata setting is storing the metadata within the same column family. With regard to timewidth, the timewidth of 5 MB has the best overall performance for writes. Therefore, using typed values, with the metadata in the same column family and a timewidth of 5 MB is suggested for this use case.

**High writes, low reads, low latency:** This is almost identical to the previous use case, with the difference that the latest data are not requested immediately, which leaves space for some extra latency tolerance. Accordingly, for this use case using the manual batching approach is suggested. For the manual batching the metadata stored in a different column family, consistently performed better than storing the metadata in the same column family. Thus, using the metadata in a different column family is suggested. With regard to timewidth, the 20 MB timewidth has the best performance with regard to writes in this case. Therefore, using manual batching, with metadata in a different column family and using a timewidth of 20 MB is suggested for this use case.

**Low writes, high reads, high latency:** For this use case, the amount of reads is higher than the writes, compared to the other use cases. Again the high latency denotes that the latest data should be available as soon as possible, which prompts us to use typed values, due to their overall lower latency. The metadata should be stored in the same column family, since we are dealing with typed values. The timewidth of 15 MB has the best performance overall, both with regard to number of reads and mean latency. So for this use case, using typed values with the metadata in the same column family and a timewidth of 15 MB is suggested.

**Low writes, high reads, low latency:** Similarly to the previous cases, this use case is identical to the previous use case, but the latest data do not need to be available as soon as possible. This prompts us to using manual batching, which stores more sensor measurements than typed values. The metadata should be stored in a different column family as already explained in sec. 9.3.4.1. With regard to timewidth, 13 MB timewidth has the best performance with regard to reads. Therefore, using manual batching, with the metadata in a different column family and a timewidth of 13 MB is suggested for this use case.
10. Conclusion

In this thesis we have presented sensor data as a type of time series data, an overview of noSQL database taxonomies and an overview of data schemas for the storage of sensor/time series data for different noSQL databases. This is the theoretical part of the thesis, which was the basis for the practical part. The theoretical part of this thesis has added value by itself. The suggestions for data schemas can be found on the Internet and other sources, however a review of data schemas for sensor data did not exist in the literature. In the practical part, load tests were performed with regard to performance of different data schemas, using Cassandra. We believe that the theoretical part along with the results of the load tests, provide guidance to interested parties on how to proceed with the storage of sensor data using noSQL databases. For instance, suggestions are proposed regarding the optimal data schema parameters for different types of generic use cases.

Possible use cases that these data schemas could be adopted for were presented in sec.9.3.5. A short summary of what was suggested in this thesis is the following:

- A higher level grouping is strongly suggested as it was seen on all the different data schemas. For example, in our schemas we grouped “chunks” of raw measurements in each row. This can be noticed in other data schemas also presented in chapter 5.

- For Cassandra, maintaining a manual index of what kind of data is stored in which row, improves the efficiency of read operations. The built-in mechanisms of the database did not perform well, due to the number of distinct values of time series.

- The performance results of different data schemas presented, demonstrates how different data schema parameters affect performance. However, each data schema should be adapted to best fit the particular use case. To adapt the data schema for a use case, it is important to understand the underlying mechanisms of the database.

- An asynchronous client utilizes all of the client resources more efficiently than the synchronous case. Therefore, asynchronous clients should be preferred.

- The database should be configured prior to deployment in a production environment. Furthermore, deployment tests are always needed before going live in a production environment.

- The practices adopted by OpenTSDB[21] appear very promising, and even though we did not test that particular database, it seems to handle time series
data very well. So for column-oriented databases, even if single values are stored on each key-value pair, grouping them together when a row is full improves performance.

The results of these tests did not have a huge impact on performance for our cluster (in terms of numbers, close to 10,000 difference was the highest difference). However, our cluster was a 6-node cluster with powerful hardware. In a cluster with more nodes the different data schemas are expected to have a higher impact. The percentages should be roughly similar, yet seeing the numbers could be of interest.

Before adopting a noSQL database, ensure that there is a good reason to do so. If the needs of an application can be covered by what relational databases provide, they should be preferred. This is due to the fact that SQL databases are widely adopted providing strong consistency and better support for ad-hoc queries. The main reason to adopt a noSQL database is due to the limited scalability of relation databases. This also includes foreseeable future needs for scalability and availability (e.g. it might not be needed now, but could be probable in the future).

There are more tests that could be run in the future, with regard to noSQL data schemas for sensor data. Some possibilities for future tests include the following:

- Testing similar data schemas in other noSQL databases to see the performance difference. This should provide further suggestions on what type of database best suits which scenario. For instance, it can be expected for example that document oriented databases would provide better support for ad-hoc queries (compared to column-oriented that we tested).

- Another set of tests would be to explore the storage of sensor metadata and the combination of raw measurements and metadata to test the database’s capability to perform ad-hoc queries. This would be of use to organizations and interested parties, since it will provide further guidance on how to proceed with the storage of time series data. This is an important issue because the amount of machine generated data increases exponentially. Also, the storage of this vast amount of data (without loss/aggregation) enables organizations to further utilize these figures by analyzing them and getting further insight.

- Performing longer duration tests (e.g. hours or even days), to ensure that performance remains consistent over time. Time series data are expected to be coming in a continuous stream, consequently longer duration tests will be more realistic than the tests which are usually carried out (e.g. our experiments only lasted 10 minutes). In longer duration tests the performance should not fluctuate too much, but this is something to examine.

- Another important test for a production environment would be testing the fail-over capabilities of Cassandra. In some of the tests, one node got overloaded with requests and rejected incoming requests (the node was still functional, which we checked using the `nodetool status` command). However, besides the node being up (but overloaded), we got exceptions that not enough replicas
were available to fulfill a request. This usually occurred with write scenarios, where the consistency was quorum, which was 2 for our 6-node cluster (using a replication factor of 3). Even if one node failed, the writes should not fail, since 2 replicas are up and running. Hence, the availability should be tested before deploying to a production environment. In our case when this happened, we simply re-run the test.

To conclude, the suggestions proposed in this thesis provide guidance to interested parties, in determining a data schema for the storage of sensor/time series data using noSQL databases. Furthermore, the overview of data schemas (c.f. chapter 5) gives the reader an idea of what is used by other experts for the storage of time series data. Finally, the future suggestions should provide more detailed results on the performance of different data schemas.
Acknowledgments

I am grateful first of all to Elena Lazovik, my TNO supervisor for the support and feedback she provided during the writing of this thesis and the support for the tests. Furthermore I would like to thank the rest of the supervisors for their useful suggestions and feedback. The professor from the university, Alexander Lazovik, for his expert opinion in databases.

Then I would like to thank all of the people in TNO for their useful input and help provided throughout this thesis. Also I would like to thank TNO for providing me the possibility to perform my thesis along with my internship, which proved to be a great learning experience.
A. Database techniques and terminology

In this section some of the new and well known techniques and concepts deployed by NoSQL databases are presented. This should provide some insight to readers that are not familiar with the concepts and techniques mentioned in this thesis.

A.1. Consistency guarantees

Consistency is one of the ACID properties (c.f. sec. 4.1.2) that ensures that any changes to values in an instance are consistent with changes to other values in the same instance. Databases can provide many different levels of consistency. To give the reader a brief overview three categories of consistency will be described in this section. The notion of consistency is somewhat different for distributed databases and single-server, non-replicated databases. In a distributed database that uses replication, consistency refers to the replicas reflecting the latest values held by the node they are replicating. In a single-server database, consistency refers to a notion of referential integrity. Referential integrity is a property of data which, when satisfied requires every value of one attribute of a table to exist as a value of another attribute in a different (or the same) table. In the definitions below, consistency in a distributed replicated database environment is assumed. It is also assumed that there is a master node that holds the original data and a slave node that replicates the data that is stored on the master.

**Strong consistency:** This consistency model implies that as soon a record is inserted or updated, any subsequent reads for that specific record will return the latest value. The read can be served either from the master, which is bound to hold the latest value since that is where the update/insert was performed, or it can also come from the slave, which under this consistency model should return the latest value for the particular record.

**Weak consistency:** Weak consistency means that after an update to the master, the slave may still have the old value for that record for an amount of time. This amount of time is called the inconsistency window. This model is not applicable in some domains (e.g. banking), but for some less critical applications (e.g. social media) it is acceptable. This tradeoff of consistency gives the application more scalability and availability.
Eventual consistency: This is a specific form of weak consistency. This model guarantees that if no new updates are made to an object the changes will be updated to the slave nodes as well. The main difference between eventual and weak consistency is about the time it takes for the changes to be propagated to the slave nodes. A well known protocol used to enforce eventual consistency is the gossip protocol (c.f. sec. A.2). A lot of noSQL stores adopt this consistency model, because it enables them to achieve greater availability and performance.

A.2. Data replication

In this section some concepts relating to data replication are briefly explained. In the context of data replication, master node will refer to the node holding the original data. Slave node or replica will refer to the node(s) that copies the data held by the master node.

Data replication: Data replication in the context of distributed databases, refers to keeping copies of data stored by a master node, in one or more slave nodes. The advantage provided by replication is that if the original node fails for some reason, the replica that has the same data can take over and continue serving requests. Furthermore, slave nodes can be used to serve read-requests in order to reduce the load of the master node. There are many techniques for replication such as synchronous or asynchronous replication. Each replication technique comes with its advantages and disadvantages. For example, if asynchronous replication is used, where the replicas lazily update the data to match the data to that of the master node, eventual consistency (c.f. sec. A.1) is guaranteed.

Consistent hashing: A hash function is any algorithm or subroutine that maps large datasets of variable length to smaller datasets of fixed length. In the context of NoSQL databases it is used to spread the data among the nodes of the cluster storing the dataset. If the data is evenly distributed among the nodes, hot spots can be avoided. Hot spots occur when a lot of incoming requests reach a particular node, which results in degradation of performance or even a crash of that particular node. Hashing solves the hot spot problem, however, another problem arises whenever a node joins or leaves the cluster. The data need to be redistributed in this case. This is what consistent hashing tries to overcome as it is described in [53]. Suppose that \( n \) is the number of nodes in the cluster, we use the function \( \text{hash}(\text{key}) \mod n \) to uniformly distribute each key around the cluster, where \( \text{hash}(\cdot) \) is a function used to hash the particular key. Using the regular hashing technique, if a new node is added to the cluster (which means computing \( \text{hash}(\text{key}) \mod (n+1) \) instead of \( \text{hash}(\text{key}) \mod n \)), results in reallocating \( n/(n+1) \) keys, almost all keys in the cluster. With consistent hashing the amount of data that need to be moved is reduced. If a machine is added to the cluster, only the data that will be stored
Data replication

in the new machine needs to be moved, which results in reallocating $1/(n+1)$. This greatly reduces the amount of data that have to be transferred if a node joins or leaves, which is very important for the performance and availability of the cluster.

**Data sharding:** Data sharding is a technique used for horizontal partitioning of data stored in a database. Data sharding goes beyond the basic notion of separating datasets that are requested by users from different geographic locations, for example. In simple terms it is splitting a large database into many smaller ones, which results in a more manageable database. This way hot spots are minimized, since the databases share nothing and can be spread among multiple servers. The same concept can also be applied in terms of data schemas, to refer to a logical grouping of the data. For example grouping various measurements in the same “bucket”/document/row or whatever is provided by the database by a time interval (e.g. hour, day, etc.).

**Read-repair:** This is a technique used in stores that support eventual consistency, to make data consistent. For example, it could be assumed that a certain part of the data is replicated in five nodes and for a read-request to be served a majority consensus is required. Therefore three different replicas will be queried for the same data. In this process, if the values between the replicas are not the same, the value with the latest timestamp is also passed to the outdated replica(s) in order to make the value consistent.

**Gossip protocol:** Bakhshi et al. in [28] describe gossip protocols as follows: “In a gossiping (also called epidemic) protocol, nodes exchange data similar to the way a contagious disease spreads. That is, a participating peer can select, according to some probability distribution, other peers to exchange information with. Gossiping protocols were originally applied in database replication, but more recently also for failure detection, and resource monitoring”. Gossip protocols are often employed by noSQL databases, due to the performance and availability enhancement they provide. It is important to make explicit that a gossip protocol results in eventual consistency (c.f. sec. A.1).

**Virtual nodes[66]:** This is a technique used by databases in order to determine which node is responsible for which range of data. When a cluster starts up data is distributed across the cluster using some technique (e.g. consistent hashing). The data that each node is responsible for is determined at that time. A simple version of this technique is to solve this problem is to create a token for each node in the cluster. Each token determines the node’s position in the “ring” and the range of data for which it is responsible. Virtual nodes shift the paradigm, from one token per node, to many tokens per node. Each token again is responsible for a certain data range. The tokens can be distributed randomly across the cluster, or a machine that has more powerful hardware can be assigned to take more tokens, for example. An advantage of the virtual nodes paradigm, is that smaller ranges are distributed across more nodes. This
enables faster “reconstruction” in case that a node goes offline and another node has to replace it.

**Hinted handoff:** Hinted handoff is a technique for dealing with node failure in the Riak cluster in which neighboring nodes temporarily take over storage operations for the failed node. When the failed node returns to the cluster, the updates received by the neighboring nodes are handed off to it. Hinted handoff allows Riak to ensure database availability. When a node fails, Riak can continue to handle requests as if the node were still there.

### A.3. Performance enhancing techniques

**Indexing:** This is a technique used by databases to fasten the search time of unique records. Such unique records are usually known as keys. Indexing essentially stores the location of keys on the disk in a key-value pair for fast access. This dramatically increases performance, compared to a disk search. However, at the same time extra storage for storing the indexes is required. Furthermore secondary indexes can also be used. Secondary indexes differentiate to key indexes, in that they do not necessarily need to be unique (as a primary key in relational databases, for example). It enables the database to be queried with criteria specified on such secondary indexes (which is not possible without an index in many NoSQL databases).

**Map/Reduce[37]:** MapReduce is a programming model for processing and generating large datasets. It arised as a programming model, due to the complexity of computing simple operations on large datasets. These operations need to be distributed among multiple computers, in order to finish in reasonable times. The complexity that MapReduce tackles, is that of distributing simple tasks among multiple computers. A map operation (defined by the programmer) is applied to each “logical” record in the input, which results in a set of intermediate key/value pairs. Then a reduce operation (also defined by the programmer) is applied to all values that shared the same key, in order to combine the derived data appropriately. Furthermore, fault tolerance is achieved by re-executing failed processes. Examples of operations that MapReduce applies include: computing various representations of the graph structure of web documents, determining the set of most frequent queries in a given day and more.

**Distributed File System (DFS):** Tzong-Jye et al. in [47] describe a DFS as “A distributed file system integrates many file servers on the network and becomes an efficiency file system with large storage space. The usage of a distributed file system is transparent for end users. Users can access files on a distributed file system just like using a local file system. They do not know where the files are stored in which network storage”. Examples of DFS include Hadoop DFS (HDFS)[10], Microsoft’s DFS[8] and Sun’s Network File System (NFS)[59].
In-memory caching: This is a technique used in many programs, ranging from web servers to databases and more. Serving files from a computer’s RAM is a lot faster compared to fetching that same file from disk\[5\]. Databases employ this technique in order to increase the performance throughput. Databases can either cache the location of keys on the disk, for fast access, or they can store entire records that are frequently accessed, which results in faster response times and less load to the database. However caching also brings the problem of stale reads, since the cache may hold an old value for a record. Specialized in-memory caching databases have been developed such as Memcached\[14\].

Multi-Version Concurrency Control (MVCC): This is a technique employed by databases in order to allow concurrent writes. The traditional way to tackle this problem is to use locks. If a record is being written/updated by a process, a lock is applied to ensure that the database is in a consistent state. However, this approach quickly becomes a performance bottleneck in databases that are under a heavy load. With the MVCC mechanism, each new update is stored as a new version for that particular record. This way multiple writes can update a record at the same time, while avoiding the locking mechanism. Reads can choose which version to read, which is in the discretion of the application/programmer. This results in more disk space being used by the database, however, disk storage is quite cheap. Therefore a lot of databases take this approach, in order to enhance performance and availability.

A.4. Detailed tables for CQL 3 Driver tests

Even though all of the diagrams presented in chapter 9 provide a lot of information, it is hard to compare the results between typed values and manual batching, since they are not compared on a one to one basis. That is why we created the tables presented in Fig.A.1 and Fig.A.2, to help the reader get a good overview of the comparison of the different tests. Only the most important metrics are presented, to keep the size of the table relatively small. These metrics are: throughput (operations per second), mean latency and standard deviation. For the manual batching write scenarios, the number of actual measurements stored is also presented. For the metadata tests presented in Fig.A.1, the performance of secondary indexing and no metadata were omitted, since they had the worse performance in all cases compared to using metadata.
<table>
<thead>
<tr>
<th>Write scenarios</th>
<th>Mean rate (ops/sec)</th>
<th>Actual measurements</th>
<th>Mean latency (ms)</th>
<th>Standard latency deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>METADATA SAME CF</td>
<td>3438.2</td>
<td>5167315</td>
<td>63.48</td>
<td>181.56</td>
</tr>
<tr>
<td>METADATA DIFFERENT CF</td>
<td>2012.1</td>
<td>-</td>
<td>20.134</td>
<td>26.66</td>
</tr>
<tr>
<td>Read scenarios</td>
<td>Mean rate (ops/sec)</td>
<td>-</td>
<td>Mean latency (ms)</td>
<td>Standard latency deviation</td>
</tr>
<tr>
<td>METADATA SAME CF</td>
<td>1378.6</td>
<td>-</td>
<td>20.134</td>
<td>26.66</td>
</tr>
<tr>
<td>METADATA DIFFERENT CF</td>
<td>2012.1</td>
<td>-</td>
<td>7.5072</td>
<td>8.37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mixed scenarios (READ)</th>
<th>Mean rate (ops/sec)</th>
<th>Actual measurements</th>
<th>Mean latency (ms)</th>
<th>Standard latency deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>METADATA SAME CF</td>
<td>318</td>
<td>53202</td>
<td>20.411</td>
<td>61.54</td>
</tr>
<tr>
<td>METADATA DIFFERENT CF</td>
<td>2012.1</td>
<td>-</td>
<td>20.411</td>
<td>61.54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mixed scenarios (WRITE)</th>
<th>Mean rate (ops/sec)</th>
<th>Actual measurements</th>
<th>Mean latency (ms)</th>
<th>Standard latency deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>METADATA SAME CF</td>
<td>31777</td>
<td>4785880</td>
<td>89.20</td>
<td>200.04</td>
</tr>
<tr>
<td>METADATA DIFFERENT CF</td>
<td>52596.6</td>
<td>4889340</td>
<td>55.47</td>
<td>184.44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Typed Values</th>
<th>Mean rate (ops/sec)</th>
<th>Mean latency (ms)</th>
<th>Standard latency deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>METADATA SAME CF</td>
<td>67681.7</td>
<td>8.57</td>
<td>16.63</td>
</tr>
<tr>
<td>METADATA DIFFERENT CF</td>
<td>2012.1</td>
<td>-</td>
<td>26.66</td>
</tr>
</tbody>
</table>

**Figure A.1.** Comparison of metadata scenarios for the CQL tests
### A.4 Detailed tables for CQL 3 Driver tests

#### Figure A.2.
Comparison of timewidth scenarios for the CQL tests

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Mean rate (ops/sec)</th>
<th>Actual measurements</th>
<th>Mean latency (ms)</th>
<th>Standard latency deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Write Scenarios</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 MEGABYTES</td>
<td>33345</td>
<td>489676</td>
<td>74.12</td>
<td>99.73</td>
</tr>
<tr>
<td>7 MEGABYTES</td>
<td>33284.4</td>
<td>4834860</td>
<td>76.85</td>
<td>137.24</td>
</tr>
<tr>
<td>10 MEGABYTES</td>
<td>33526</td>
<td>720365</td>
<td>76.85</td>
<td>137.24</td>
</tr>
<tr>
<td>12 MEGABYTES</td>
<td>33530.5</td>
<td>5025075</td>
<td>69.47</td>
<td>156.49</td>
</tr>
<tr>
<td>15 MEGABYTES</td>
<td>33193.8</td>
<td>4910715</td>
<td>66.22</td>
<td>172.32</td>
</tr>
<tr>
<td>20 MEGABYTES</td>
<td>33442.1</td>
<td>5016390</td>
<td>69.28</td>
<td>183.60</td>
</tr>
<tr>
<td><strong>Read Scenarios</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 MEGABYTES</td>
<td>270.7</td>
<td>949575</td>
<td>9.17</td>
<td>9.24</td>
</tr>
<tr>
<td>7 MEGABYTES</td>
<td>2240.1</td>
<td>-</td>
<td>8.41</td>
<td>8.41</td>
</tr>
<tr>
<td>10 MEGABYTES</td>
<td>2364.9</td>
<td>-</td>
<td>7.32</td>
<td>7.77</td>
</tr>
<tr>
<td>12 MEGABYTES</td>
<td>2628.2</td>
<td>-</td>
<td>6.35</td>
<td>6.76</td>
</tr>
<tr>
<td>15 MEGABYTES</td>
<td>2003.9</td>
<td>-</td>
<td>9.71</td>
<td>11.53</td>
</tr>
<tr>
<td>20 MEGABYTES</td>
<td>2008.9</td>
<td>-</td>
<td>9.41</td>
<td>10.13</td>
</tr>
<tr>
<td><strong>Mixed Scenarios (READ)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 MEGABYTES</td>
<td>452.1</td>
<td>-</td>
<td>28.96</td>
<td>117.64</td>
</tr>
<tr>
<td>7 MEGABYTES</td>
<td>374.3</td>
<td>-</td>
<td>35.29</td>
<td>133.17</td>
</tr>
<tr>
<td>10 MEGABYTES</td>
<td>418.5</td>
<td>-</td>
<td>18.98</td>
<td>77.24</td>
</tr>
<tr>
<td>12 MEGABYTES</td>
<td>455.4</td>
<td>-</td>
<td>22.88</td>
<td>85.93</td>
</tr>
<tr>
<td>15 MEGABYTES</td>
<td>335.6</td>
<td>-</td>
<td>27.73</td>
<td>90.40</td>
</tr>
<tr>
<td>20 MEGABYTES</td>
<td>416.6</td>
<td>-</td>
<td>27.73</td>
<td>90.40</td>
</tr>
<tr>
<td><strong>Mixed Scenarios (WRITE)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 MEGABYTES</td>
<td>117.7</td>
<td>4990110</td>
<td>42.91</td>
<td>51.65</td>
</tr>
<tr>
<td>7 MEGABYTES</td>
<td>31571.9</td>
<td>4795770</td>
<td>62.59</td>
<td>178.63</td>
</tr>
<tr>
<td>10 MEGABYTES</td>
<td>31584.8</td>
<td>431726</td>
<td>13.13</td>
<td>127.50</td>
</tr>
<tr>
<td>12 MEGABYTES</td>
<td>31062.8</td>
<td>4569420</td>
<td>65.53</td>
<td>214.97</td>
</tr>
<tr>
<td>15 MEGABYTES</td>
<td>31534.6</td>
<td>4782860</td>
<td>69.71</td>
<td>241.00</td>
</tr>
<tr>
<td>20 MEGABYTES</td>
<td>32062</td>
<td>4592360</td>
<td>65.53</td>
<td>149.76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Mean rate (ops/sec)</th>
<th>Actual measurements</th>
<th>Mean latency (ms)</th>
<th>Standard latency deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Typed Values</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 MEGABYTES</td>
<td>9620.9</td>
<td>-</td>
<td>85.76</td>
<td>20.93</td>
</tr>
<tr>
<td>7 MEGABYTES</td>
<td>59363.6</td>
<td>11.08</td>
<td>24.42</td>
<td></td>
</tr>
<tr>
<td>10 MEGABYTES</td>
<td>54584.9</td>
<td>12.96</td>
<td>24.70</td>
<td></td>
</tr>
<tr>
<td>12 MEGABYTES</td>
<td>59437.5</td>
<td>11.25</td>
<td>41.21</td>
<td></td>
</tr>
<tr>
<td>15 MEGABYTES</td>
<td>52863.0</td>
<td>12.67</td>
<td>30.47</td>
<td></td>
</tr>
<tr>
<td>20 MEGABYTES</td>
<td>57784.5</td>
<td>13.75</td>
<td>37.86</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


