UNIVERSITY OF TWENTE

MASTER THESIS

The Storage and Retrieval of Sensor Data and its Annotations

Bart Sikkens

Department of Computer Science

Groningen, May 31, 2010

Supervisors:
Dr. Andreas Wombacher
Dr. Ir. Maurice van Keulen

Abstract

The focus of this thesis is on the storage part of a sensor management system. Sensor data can already be enormous, but because every measurement stored can also contain multiple annotations the total size can really explode. Besides this, annotations are not necessarily bound to a single measurement, but can also be attached to multiple sensors and on time spans. This creates challenges in storing the data efficiently while also maintaining efficient retrieval. On the retrieval side there are all kinds of queries possible on the sensor data, annotations, time or combinations of those. To provide efficient storage for all these types of queries makes things even more challenging, because every storage method has its advantages and disadvantages.

This thesis looked into a number of methods which try to solve these problems. Three of these methods were chosen and tested on scalability after which the most scalable was implemented into an open-source software package. This implementation was tested again on scalability.

Contents

List of figures

List of tables

1. Introduction

The IJkdijk project [IJk09] is a cooperation of many companies and parties with as goals the improvement of dikes, real-time monitoring of their strength and conduct research to the use of sensor systems in early warning systems. These sensor systems consist of a multitude of sensors (video, audio, infrared, temperature, pressure etc.) which all produce *sensor data* at a certain frequency. Scientists involved in this project have the need to annotate this data using *annotations*, which say something about the measurements. While this might look like an easy task at first sight, there are complications making this a challenging task. We will start with taking a look at these complications.

1.1. Problem description

One of the complications mentioned earlier is how to store the huge amounts of sensor and annotation data in an efficient way. Sensor data can already be enormous, but because every measurement stored can also contain multiple annotations the total size can really explode. Besides this, annotations are not necessarily bound to a single measurement, but can also be attached to multiple sensors and on time spans. This creates challenges in storing the data efficiently while also maintaining efficient retrieval. On the retrieval side there are all kinds of queries possible on the sensor data, annotations, time or combinations of those. To provide efficient storage for all these types of queries makes things even more challenging.

1.2. Research goal and contribution

The goal of this master thesis is to take a look at the currently available methods for storing and retrieving sensor data and its annotations and test them on scalability. Due to time constraints we will have to make a selection from all available methods and we like to test at least 1 non-relational method. What we hope to contribute with this thesis is providing an overview of available methods and providing scalability numbers for a few of these methods.

Besides that we aim to create a working implementation of one of these methods. This implementation should be able to retrieve data and their annotations as well as supporting annotations in queries.

1.3. Research structure

The following structure is used in this master thesis. First in chapter 2 the actual problem will be defined and in chapter 3 a literature study will be done. Next we will present a detailed approach in chapter 4. In chapter 5 a case will be described to provide material to base further examples on. After that, possible storage models for sensor data and annotations will be described in chapter 6 and we will take a look at how to retrieve sensor data and annotations again in chapter 7. Then in chapter 8 a number of solutions for the problem will be validated using experiments with as criteria scalability. The best solution from the experiments will be implemented (chapter 9) and tested in terms of scalability (chapter 10). Finally we will draw our conclusions and discuss future work in chapter 11.

1.4. Research questions

The main research question and sub questions for this research are as following:

- How can we efficiently store sensor data and its annotations in such a way that it can be queried efficiently again with different types of queries?
    - What kinds of storage and storage models for sensor data and its annotations are available?
    - How can we efficiently retrieve data and annotations?
    - Which solution has the best scalability?

1.5. Research scope

The main focus of this research will be on the storage and retrieval of the sensor data and its annotations. We are specifically interested in the annotations part. Interfaces and possible other software that is required will just be made or used to reach the main goals, but will not be main focus. Considering the comparison part, scalability will be the most important criteria to measure. The amount of data will be scaled and tested in terms of storage and retrieval.

2. Detailed problem description

We will first take a look at what actually is the problem with storing sensor data and its annotations. The simplest approach would be to attach the annotations to the actual measurements. But measurements can have multiple annotations and a basic relational database does not support set-valued attributes. A solution is to use the first normal form (1NF) for databases, meaning we will give annotations a separate table and we create a table to connect measurements to annotations. This will result in a lot of joins when querying and redundant storage, which is inefficient. Another solution would be to use a database scheme in non-first normal form (N1NF) [JS82]. This form allows sets and therefore we can attach multiple annotations to one measurement. However relational databases do not support this form.

Next we will distinguish a number of types of annotations and queries. These will later on be used to compare all found methods in the related work section.

2.1. Annotation and query types

We can think of different types of annotations and query types. At first we will take a look at the annotation types. We will start with annotations on different granularities of a database. Annotations can be on an entire table/relation, so every row and column in the table is annotated with a certain annotation. Whole rows/tuples and columns can be annotated, but also specific cells or a random set of cells. Annotations can be across tables, meaning that data that is split between tables is annotated using the same annotation without duplication. Annotations can also be within relations. All above mentioned types could be combined with a time element. All so far mentioned types are bound to certain sensor data. But we can also think of situations where it is hard to find to which data an annotation belongs. For example a power cut happened and all sensors went off. No data is available at that moment and someone wants to annotate why there is no data at that time span. Because no data was created, there is no data to attach the annotation to. In this case the annotation would have to be bound to time itself.

Considering queries we can retrieve data, annotations or both. We can also add time constraints to our queries. Further we can think of annotation management like creating and deleting annotations or adding to and removing annotations from data. In the next chapter we will take a look at the related work.

3. Related work

In this chapter we will discuss related work. We will start by taking a look at relational storage and end with column databases. After discussing all literature, the summary will contain 2 tables summing up what annotations types and query types (as discussed in the previous section) are supported by which system/method.

3.1. Relational storage

This section is divided into several categories of papers/systems. We will start looking at the similarities and differences between relational storage and XML.

XML

Where relational storage doesn't allow the storage of a set of values in a single field, XML can store set-valued attributes. We will first have a look into literature that converts XML to a relational database. This process is called shredding.

[STH99] defines three special types of shredding using Document Type Definitions (DTDs). Normal shredding directly maps elements to relations and this creates a lot of fragmentation. The proposed solution is to use inlining, which puts as many descendants of an element as possible into a single relation. Because the traditional relational model cannot handle set-valued attributes, basic inlining uses the standard technique for storing sets in a relational database (1NF). The shared inlining method goes one step further and ensures that an element node is represented in exactly one relation. Finally the hybrid inlining method inlines even more elements. One of the conclusions this paper draws is that relational systems could more effectively handle XML query workloads with support for sets. It would reduce fragmentation, which is a big win because most of the fragmentation they observed in real DTDs was due to sets.

The SQL/XML standard as proposed in [EM02] and [EM04] describes the mapping of SQL data to XML data. The standard has a function called XMLTable [EM04] which can transform XML data into relational data. This function uses a form of shredding, but doesn't require a DTD. Instead it uses XQuery [W3C01], a query language for XML, and manually chosen column names.

We can learn from this literature ([STH99] [EM02] [EM04]) that even though there are methods to convert set-valued attributes in XML to relational, the efficiency is limited by the absence of set support in the relational model. The conversion is also a lot more complicated without it. The conclusion is that the conversion of SQL to XML and vice versa has no special trick or method that could give us an advantage in storing annotations in a relational database.

BDBMS

[EOA07] proposes a database system for managing biological data. Part of this project is about annotation management. [EAE09] talks specifically about this part and proposes a relational storage model for annotations at various granularities (table, tuple, column, cell levels). They introduce three types of annotations, namely snapshot (normal annotations),

view (automatic annotation of newly inserted data if it satisfies certain conditions) and join (attached to data across relations).
They also introduce the Mapped-Space storage scheme for efficient and compact representation of multi-granular annotations. This scheme numbers all rows and columns and therefore annotations can be represented by rectangles. To achieve the most efficient storage, the columns with the most annotations should be next to each other, creating fewer rectangles per annotation. For this reason they created an algorithm that, based on statistics, gives the ideal mapping of columns. This process does not influence the actual physical storage order of the columns, but only the numbering of columns. Using a columns mapping quality (CMQ) metric it is decided whether it is worth re-constructing all annotations according to the new mapping. This will depend on the gain of reducing the storage overhead and I/O cost of queries compared to the cost of re-constructing.
To realise all of this, they extended PostgreSQL with new constructs and added declarative mechanisms to support adding, storing, archiving and querying snapshot, view and join annotations. The Mapped-Space schema achieves more than an order-of-magnitude reduction in storage and up to 70% reduction in the queries execution time.
For *BDBMS* it is important to have as much information (data and time) as possible in 1 table to make sure the number of rectangles per annotation is as low as possible. Using 'join annotations' it is possible to annotate across relations and hence tables, but this is a more expensive operation. For each data table multiple annotation tables can be assigned, but not the other way around. It is always possible to annotate an entire table, whole rows, whole columns or a random set of cells. If information about time can be found inside the table, it should be possible to combine all options with time. It must be said that the authors didn't look at the time aspect, but because they extended the SQL select statement it should be possible to add time constraints in the 'where' clause. It is not possible to annotate on time itself. All types of queries are possible, although the time component is not discussed in the paper.
BDBMS is a very interesting and relevant system. It has the ability to store all kinds of annotations and the query language is also sufficient. The smart way of storing annotations reduces storage requirements. The only disadvantage is that the paper does not talk about support for time constraints.

MAAS

[KMP09] proposes a system for massive annotation and aggregation of sensor data. They developed a query language for efficient retrieval of data and its annotations, which can filter on both time and value ranges and supports Boolean logic operations. The language also supports adding, deleting and describing annotations. The storage is done using tables for sensor stations and tables for sensors which are linked to a station. Annotations are stored in a separate table and a relation between sensor data and an annotation is stored into another table. They use data aggregation on different levels on the time scale and annotation caching to achieve high efficiency. Their prototype features a web interface that is able to visualize data including its annotations.
MAAS stores sensor stations and sensors in separate tables. An annotation is assigned to a specific sensor belonging to a station (which translates to a row/tuple in the sensor table). Because of this it is not possible to annotate a whole table without duplication. It is possible to assign time constraints to an annotation, but only on rows, because of the nature of the storage model and query language. All types of queries are possible, although getting both sensor data and annotations is a bit uncertain. The language suggests it, but it is not mentioned nor used in their examples.

In relation to our work, this system is quite relevant. It might not be able to support all types of annotations, but for sensor data it provides enough functionality. It can store multiple annotations on a measurement, but not one annotation over multiple measurements without duplication. The query language describes the functionality as we would expect from such a system.

DBNotes

[CCV05] proposes a 'post-it note' system for relational data. In this system it is possible to attach zero or more notes to every value in a relation. The propagation of the notes is based on provenance, which is not the focus of our research. They developed an extension to a fragment of SQL called pSQL, which can query both data and its annotations and propagate annotations. Although the query part is relevant, the rest of the paper focuses on provenance. The original paper doesn't describe the actual storage methods used, but [Bog05] did and also developed an alternative storage scheme for DBNotes. The original storage model created an extra attribute for every attribute in a relation in order to store the annotations. In other words, for every column an annotation column was added. This leads to redundant storage and as a direct consequence increased response time. The proposed solution is a separate relation to define the annotations present. In other words, for every table an annotation table is created storing the annotations. Besides measuring absolute performance, 2 tests were run to compare pSQL to normal SQL. The observed results were that using the default propagation, pSQL was 1 to 2 times slower than normal SQL.
[BCT04] describes work done before [CCV05] and is also about pSQL, propagation of annotations and data provenance.
DBNotes supports annotations within relations/tables. It is possible to annotate a row/tuple within a relation or one or more attributes of it. Because it is relation-based, annotations cannot be across tables. Support for time is also not available. Considering querying, data, annotations and both can be retrieved. Time support and annotation management are not supported. DBNotes main focus is on providing data provenance, which is not our focus.

Mondrian

[GKM06] proposes the Mondrian system. It has an annotation mechanism to annotate both single value and the associations between multiple values. It is able to query not only for data but also for annotations. It annotates using blocks that are put on relations. Using Boolean values it is stored which parts of the relation are affected by the annotation. The actual annotation is called a colour. The relations and annotations are stored in a new table, hence not affecting the schema of the data. Experiments done show that each colour operator costs from three to five times as much as its relational counterpart. They call the overhead not prohibitive and say it is balanced by the added value of being able to represent and query complex annotations. Planned further optimizations will further reduce the cost. Provenance was one of the motivators to develop this system, along with integrating, annotating and cross-referencing scientific databases.
Mondrian also annotates within relations/tables and supports the same types of annotations as *DBNotes*. But it adds to this the ability to annotate across tables because the annotation table can contain IDs of different tables. It can also query data, annotations and both. It is a bit vague what is actually possible considering annotation management, but adding colour blocks is mentioned.

DBNotes has the disadvantage of not supporting time constraints on annotations. Further it is not clear what annotation management is actually supported. It does not have the annotation flexibility of BDBMS, but it still supports some annotation types.

Other work

[WHB08] investigated ways to store both sensor and annotation data. They looked at the advantages and disadvantages. Validation was done by querying with a fixed set of queries. Their conclusion was that the sensor data storage model as later seen in Table 4, storage per sensor station, is the best solution under their assumptions (sensor are not synchronized on time, most queries contain a time constraint and more, see [WHB08]). Considering the annotation storage, they chose the storage model that will be described later in this document in the section 'Annotation storage' on page 19.
This research gave a nice insight into the problems arising when storing sensor and annotation data. The annotation storage model chosen is an interesting one and we will elaborate on it later on.

3.2. Column databases

Besides XML and Relational database there is a third development in the area of storage called column databases. These databases store their data by column instead of row. This has advantages for computing aggregates and updating whole columns. We will start with taking a look at Google Bigtable, Google's database which shares characteristics of both row-oriented and column-oriented databases.

Google Bigtable

Literature study shows there is only a limited amount of papers available about Bigtable and its open-source alternatives. [Goo06] describes the basics of the Google Bigtable database. This database is the method of storage behind Google projects like Google Analytics, Google Earth and Google Personalized Search [Goo08]. It is built on top of Google File System (GFS) and some other Google programs like for example Chubby Lock Service. Each table in Bigtable is "a distributed multi-dimensional sorted sparse map", sharing characteristics of both row-oriented and column-oriented databases. The table consists of rows and columns, and each cell has a time version. There can be multiple copies of each cell with different times, so it is possible to keep track of changes over time. Columns are grouped into sets called column families. These families need to be defined before storing data, but inside them can be an unbounded number of columns, created on the fly.
Bigtable is designed to scale into the petabyte range across hundreds or thousands of machines. Also it is easy to add more machines to the system and automatically start taking advantage of those resources without any reconfiguration. One of the advantages is cheap storage of NULL values and only changes in time are stored, reducing storage overhead.
[KG08] compares the performance of Hypertable and HBase, two open source alternatives for Google Bigtable. It is unknown whether the measurements done are still relevant.
Unfortunately the development of the open-source alternatives is not very far and for example Hypertable lacks support for joins and can only store strings.

Cassandra

[LM09] proposes the Cassandra system, which is the database behind the Facebook website. This system is built with continuous failure of components in mind. It does not support a full relational model, but supports dynamic control over data lay-out and form. It can handle high write throughput while not sacrificing read efficiency.
A table (also called keyspace) in Cassandra is a distributed multi dimensional map indexed by a key. The row key in a table is a string with no size restrictions, although typically 16 to 36 bytes long. Every operation under a single row key is atomic per replica no matter how many columns are being read or written into. Columns are grouped together into sets called column families, very much similar to what happens in the Bigtable [Goo06] system. Cassandra exposes two kinds of columns families, Simple and Super column families. Super column families can be visualized as a column family within a column family. Columns within a super column or simple column family can be sorted on time or name. Cassandra can store binary data and timestamps.
Facebook currently stores 50+TB of data on a 150 node cluster. [LM09-2] shows a significant performance gain over a MySQL database. On 50 GB data it took MySQL ~300ms to write and ~350ms to read, while Cassandra did it in ~0.12ms and ~15ms.
Cassandra is a robust and scalable system. It focuses on high write throughput while still reading efficiently. Sensor data on the other side is written once and never updated, but read a lot. In this sense the focus is different. The easy adding of columns makes it a very flexible system. The system is actively developed and is an interesting alternative for Bigtable.

MonetDB

MonetDB [Bon02] is a database system that is a crossover between row and column based. It shows high performance especially in data mining, OLAP and GIS applications. The MonetDB core is usable with SQL and XQuery. For SQL there are application bindings for many programming languages and it runs on nearly any platform. MonetDB achieves its goal by innovations at all layers of a DBMS, e.g. a storage model based on vertical fragmentation, a modern CPU-tuned query execution architecture, automatic and self-tuning indexes, run-time query optimization, and a modular software architecture [CWI10].

3.3. Summary

Table 1 shows the types of annotations and what relational systems/methods are able to handle which types. There are no systems able to handle annotations on time itself. BDBMS is a very flexible system regarding annotations. MAAS is limited in this by its storage model, which only allows elements of the sensor tables to be annotated. DBNotes annotates within relations/tables and therefore support per cell and row. Mondrian is a bit more flexible and can also annotate across tables, because it can create an annotation table using IDs from multiple tables. WHB08 can only annotate rows/tuples and therefore also whole tables and within a relation. Time is supported if it is stored for each row. Because there were no systems or methods found for column databases they are not mentioned in this and the following table.

Annotations	BDBMS	MAAS	DBNotes	Mondrian	WHB08
On a table	√	-	-	-	√
Per cell	√	-	√	√	-
Per row/tuple	√	Sensor only**	√	√	√
Per column	√	-	-	-	-
Per random set of cells	√	-	-	-	-
Within a relation	√	Sensor only**	√	√	√
Across tables	√	-	-	√	-
Any above in combination with time	√*	Per row, sensor only**	-	-	√***
On time itself	-	-	-	-	-

Table 1: Supported types of annotations on database level by proposed systems/methods in literature
* = depending on storage model, not explicitly mentioned in paper
** = only the tables storing sensor data
*** = if time is stored for each row/tuple

Table 2 shows for the proposed systems/methods in the literature what types of queries are supported. *BDBMS* can handle all types, but time constraints are not explicitly mentioned in the paper and neither are they tested on performance. *MAAS* can retrieve data and annotations, but it is not explicitly mentioned that the combination of those two is also possible. Both *DBNotes* and *Mondrian* support retrieving data, annotations and both. Mondrian also has some annotation management. Both cannot handle time constraints. WHB08 can retrieve data, annotations, both and use time constraints if the time is stored for each row.

Type of query	BDBMS	MAAS	DBNotes	Mondrian	WHB08
Get data	√	√	√	√	√
Get annotations	√	√	√	√	√
Get both	√	?**	√	√	√
Time constraints	√*	√	-	-	√****
Annotation management	√	√	-	?***	-

Table 2: Supported query types by proposed systems/methods in literature
* = depending on storage model, not explicitly mentioned in paper
** = the language suggests it is possible, but it is not explicitly mentioned nor used in an example
*** = it is mentioned somewhere that colour blocks can be added, but not how and what else is possible
**** = if time is stored for each row/tuple

One of the goals of this research was to provide an overview of available systems and methods. This goal has been achieved and the goals remaining are providing scalability numbers for a few of the found systems/methods and creating an implementation with one of them. The next chapter will present a detailed approach for reaching these two goals.

4. Detailed approach

This section will present the detailed approach for this research. Figure 1 shows an overview of the involved parts for this research. On the left side there are sensor stations (which contain sensors) or 'simple' sensors. The data will have to be collected, annotated and stored somehow. The main focus for this research will be on the storage and retrieval parts. We assume there is a data set available and therefore we don't need to worry about data collection. Because our storage model will most likely be different from the original one, the data will have to be manipulated into the new format. The original data will not contain annotations and therefore they will have to be added. After the retrieval some form of display is necessary to show the results.

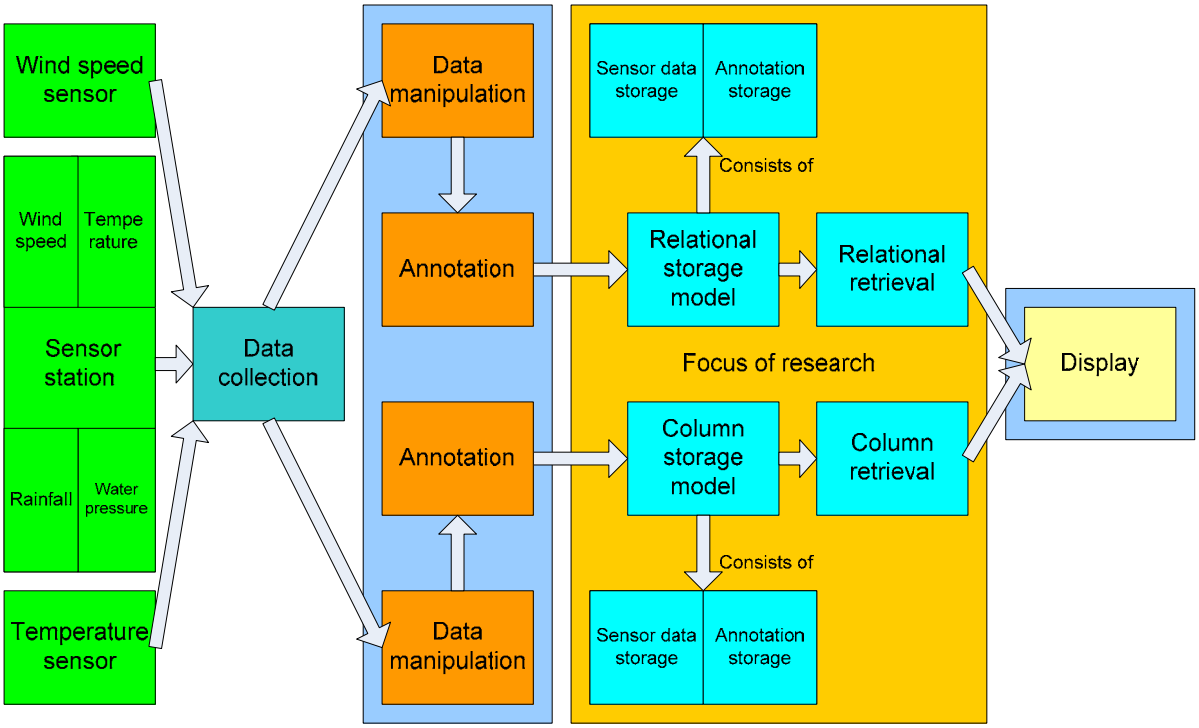


Figure 1: Overview of involved parts

For the storage part the plan is to use both a relational and a column database. For both database types a storage model will have to be developed for the storage of sensor data and annotations. They will also both need a retrieval component. This retrieval component should have support for querying on annotations, sensor data and both in combination with time constraints.

The validation of the designed solutions will be done using experiments. The main criteria for these experiments will be scalability. A few methods for storing and retrieving sensor data and annotations will be chosen. For these methods 3 data sets will be created, each 10 times bigger than the previous one. An independent query set is required in order to perform the experiments and this query set should also be used for testing the final implementation. The results will consist of measurements for each query on each data set size.

Figure 2 shows the detailed action plan for this research. At first the storage models for both relational and column will have to be developed and also the query languages that will be used to retrieve data again. Besides that a subset of the available data will have to be chosen. Once these steps are done the chosen data can be manipulated to fit in both storage models. Because there are no annotations yet these will have to be added to the data. Also an experiment design will have to be made describing what will exactly be done during the experiments. This should give as a result a set of queries to use in the experiments. Once all these steps are done the actual experiments can be done. If the results are not as hoped, it is possible to develop another storage model and go through the procedure again if time allows.

Finally the best solution found during the experiments will be implemented and tested again on scalability. The implementation should be usable by users and have some form of display.

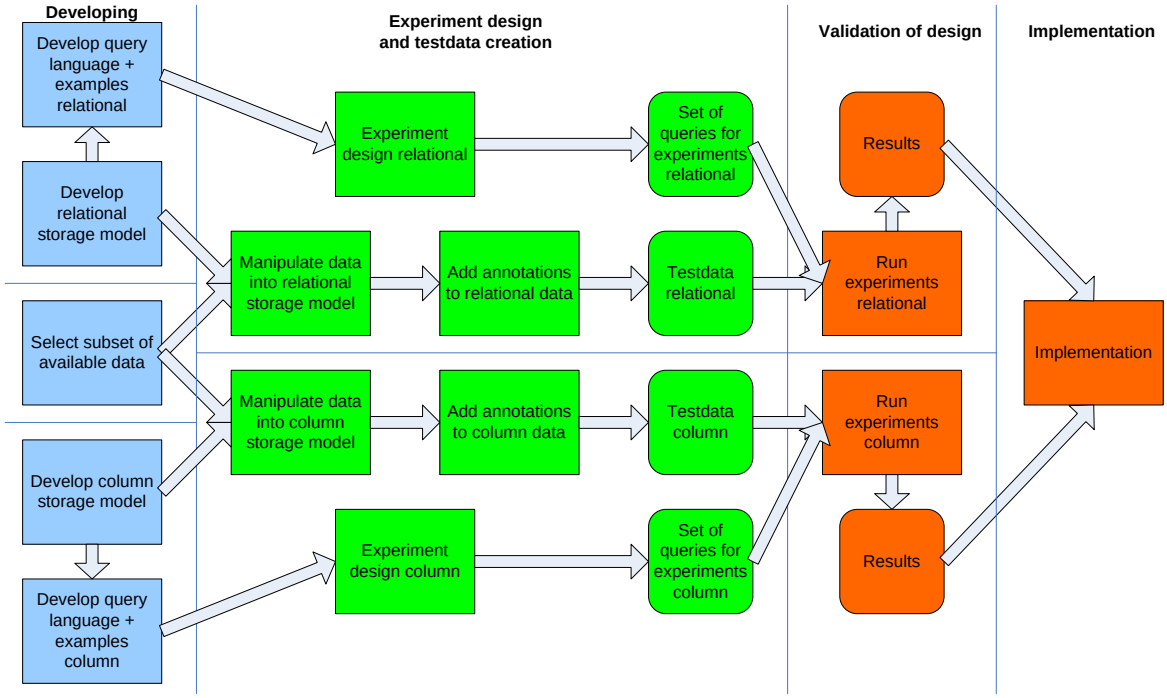


Figure 2: Detailed action plan

5. Case

This section will describe a case to show the complexity of sensor data and its annotations and to provide material to base further examples on. In the IJkdijk project [IJk09] one of the sensor stations used is a weather station which contains multiple sensors. In this case we will use this weather station as an example.

Imagine we have 3 of these weather stations with each a number of sensors. We will assume all 3 stations have the same sensors, namely a temperature sensor, a rainfall sensor, a wind speed sensor and a video camera. As we have seen in the section 'Annotation and query types' on page 8, annotations can be a lot more than simply be attached to a single measurement. Figure 3 shows a time span of data measurements for 6 sensors from the 3 weather stations and the following types of annotations (on sensor level) on this sensor data:

- Annotations over multiple sensor types (e.g. Warm rain)
- Annotations over multiple sensor stations (e.g. Hot)
- Annotations over multiple sensor types and -stations (e.g. Storm, Nice weather, Warm rain)
- Annotations over 1 sensor (e.g. Warm, Lightning)
- Annotations overlapping in time (e.g. Warm, Storm and Lightning)
- Annotations on time itself (e.g. Power failure)

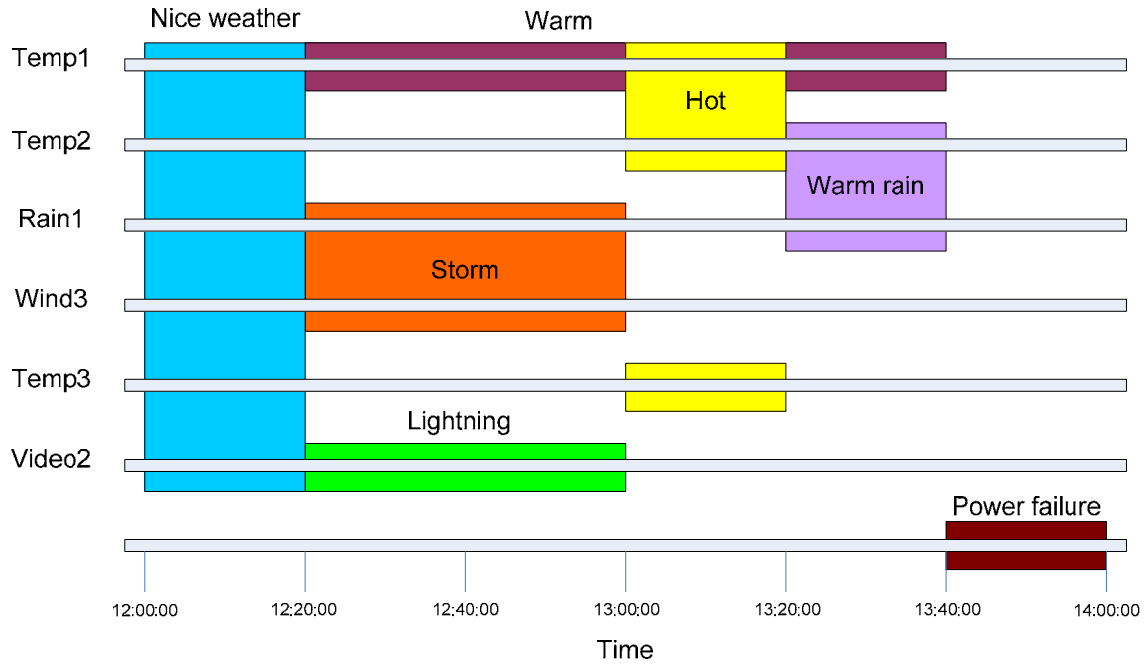


Figure 3: Examples of annotations over sensor data

The difficulty that comes with all these types of annotation constructs is that they can all be stored efficiently in their own way and that some storage methods won't be able to support them all. The challenge therefore is to be able to store all these types efficiently while maintaining efficient retrieval. Of course no storage method is perfect and there are always trade-offs to make. Next we will look into a number of ways to store sensor and annotation data and what their advantages and disadvantages are.

6. Storage of sensor data and annotations

In related work (chapter 3) we have found a number of systems and methods to store sensor data and its annotations. In this section we will discuss these methods more in detail. We will start by taking a look at the relational part and then at the column database part. For relational we will first look at a number of methods for storing sensor data and annotations. After that we will take a look at the systems found and discuss their methods used. For all methods we will discuss their advantages and disadvantages.

6.1. Relational storage

In chapter 2 we already mentioned the simple approach of attaching annotations to measurements and the theoretic solution of using N1NF schemas. The first was inefficient due to duplication of data when multiple annotations are attached to a single measurement. The second option is an interesting theoretic solution, but there are no databases supporting this form. Using the 1NF, a lot of redundant data is created and when querying a lot of joins have to be done, making this solution both inefficient on storage and retrieval terms.
Next we will take a closer look at the storage solutions in [WHB08]. In their opinion sensor data and annotation data are different and therefore require different storage models. Sensor data most likely consists of a timestamp, value and some sensorID. A sensor station can contain multiple sensors with different measurement types (e.g. wind speed, temperature, rainfall or video). Annotation data consists of an annotation and possibly a comment on it and it has to be related to a measurement or multiple measurements. We will first take a look at sensor data storage.

6.1.1. Sensor data storage

A few options [WHB08-2] mentioned are storing measurements per measurement type (see Table 3), per sensorID (see Table 4) or all data in one big table (see Table 5). All have their advantages and disadvantages which we will discuss next.

Table 3 provides the advantage of efficient search for a specific measurement type like temperature. It also prevents empty cells coming from different measuring frequencies of different measurement types. Disadvantages are redundant storage of timestamps, inefficient retrieval of all measurement types or data at a certain timestamp or time space from 1 sensor and sensors must be synchronized.

Temperature			
Time	**Temp1**	**Temp2**	**Temp3**
12:00:00	19.0	19.9	19.2
12:20:00	20.5	21.2	20.3
12:40:00	22.7	23.4	22.9

Table 3: Example of sensor data storage per measurement type

Table 4 provides efficient search for all data of 1 sensor or for a {time, sensor} tuple. Disadvantages are inefficient search for time intervals over multiple sensors and querying over a measurement type. Also timestamps are stored redundant and empty cells will exist in the table if different measurement types have different measuring frequencies.

Weatherstation1			
Time	**Wind speed**	**Temperature**	**Rainfall**
12:00:00	2.0	19.0	0.0
12:20:00	8.0	20.5	0.4
12:40:00	10.0	22.7	0.5

Table 4: Example of sensor data storage per sensor

Table 5 has the advantage of no redundant storage of timestamps and provides efficient search for all sensor data on a certain timestamp or time space. Disadvantages are that sensors must be synchronized and different frequencies create a lot of empty cells. Also scalability is low because new sensors will always introduce new columns.

Data									
Time	**Wind1**	**Wind2**	**Wind3**	**Temp1**	**Temp2**	**Temp3**	**Rain1**	**Rain2**	**Rain3**
12:00:00	2.0	3.0	1.0	19.0	19.9	19.2	0.0	0.0	0.0
12:20:00	8.0	9.0	7.0	20.5	21.2	20.3	0.4	0.5	0.4
12:40:00	10.0	7.0	9.0	22.7	23.4	22.9	0.5	0.4	0.4

Table 5: Example of sensor data storage with all data in 1 table

6.1.2. Annotation storage

There are many possible ways to store annotations. [WHB08] describes 3 types of storage that use the annotation name as table. The first uses a {time,sensor_id} tuple, the second a {time, sensor_1, sensor_2, sensor_3} tuple and the third a {start_time, end_time, sensor_id} tuple. All have their advantages and disadvantages in terms of space and search efficiency.

A fourth option mentioned uses a table per sensor station and {query, sensor_id, annotation_id} tuples in it (see Table 6). Annotation_id refers to an ID in an Annotation table (Table 7) which contains all annotations for that ID and possibly a comment on it which is stored in the table Comments (see Table 8). The query returns all measurements to which the annotation applies. When searching for all measurements with a certain annotation set, the DBMS can union the queries found in that set. This type of storage also allows annotations to change name and the use of multiple annotations on a certain time interval on a sensor. If a specific time interval is requested, this can be appended to the queries as another constraint, removing the need for post-processing. Disadvantages are that storage requirements will be high if annotations are spread over small time intervals and overlapping annotations create a whole new problem for both storage and retrieval.

Weatherstation 1		
Query	**Sensor_ID**	**Annotation_ID**
Temp > 20.0	Temp	1
Temp > 25.0	Temp	2
Temp > 20.0 and rain > 0.3 and wind > 6.0 and video.event = lightning	Temp	3
Temp > 20.0 and rain > 0.3 and wind > 6.0 and video.event = lightning	Rain	3
Temp > 15.0 and rain = 0.0 and wind < 4.0	Temp	4
Temp > 15.0 and rain = 0.0 and wind < 4.0	Rain	4
Temp > 20.0 and rain > 0.0	Rain	5

Table 6: Table for weather station 1 containing annotations

Annotations	
Annotation_ID	**Value**
1	{warm}
2	{hot)
3	{storm, lightning, warm}
4	{nice weather}
5	{warm rain}

Table 7: Table for storing annotation sets

Comments	
Annotation_ID	**Comment**
1	Warm outside
3	Lightning visible
4	No clouds in the sky
5	Warm rain manually confirmed

Table 8: Table for storing additional information about annotations

6.1.3. Storage systems

The previous two sections showed ways of storing sensor and annotation data. This section will take a look at some complete systems which were also discussed in chapter 3.1. Here we will focus on how they actually store the sensor and annotation data.

MAAS

The MAAS [KMP09] system stores both sensor stations and sensors in their own table. The sensor table is linked to a station and will contain all data produced by that sensor. The 'SensorAnnotation' table links sensor data to an annotation. Text can be added to describe a specific annotation instance. Time ranges are supported by using aggregation of data. This method of storage produces a lot of joins in the queries. Possible existing data will have to be converted into the used format in order to use this system.

DBNotes

DBnotes [CCV05] produces extra tables in order to store the annotations. Further it uses an extension of SQL called pSQL. It does not affect the original sensor data. Experiments showed that pSQL query execution time was a factor 1 to 2 slower than normal SQL query execution time. The gain is the ability to propagate annotations.

Mondrian

Mondrian [GMK06] also produces extra tables to store annotations. Using Booleans for each attribute in the new table, it can be decided which attributes are relevant to the annotation. This solution also doesn't affect the original data. Experiments showed that their annotation query language was around 3 to 5 times slower than SQL. This loss of performance is compensated by the ability to represent and query complex annotations.

BDBMS

BDBMS [EAE09] does not affect the original sensor data. Every data table can have multiple annotation tables. These tables store the annotated data using rectangles. The rectangles are built by numbering both rows and columns. In order to be truly effective, columns with a high correlation need to be next to each other. This is done using an algorithm based on statistics about the annotations present in a table. Their solution achieves more than an order-of-magnitude reduction in storage and up to 70% reduction in the queries execution time.

6.2. Column database storage

There were no systems or methods found that are able to store and retrieve sensor data and its annotations using a column database. Therefore we will have to think of a storage method ourselves and we have chosen to do this using Google Bigtable because of its built-in time support. Since it will not be possible to use the actual Google Bigtable system, open-source alternatives will have to be used. Unfortunately these open-source alternatives are still young and have limited functionality. For example they only support the String type and do not support joins of tables.

Next we will describe a solution using a ‘Bigtable’ system. Both the sensor data and the annotations will be stored in the same column family (called ‘data’) and ‘Bigtable’ provides the version management. The data will be stored under the column qualifier ‘sensor’ and the annotations under the column qualifier ‘annotation’. When multiple annotations have to be stored at a specific timestamp this works as long as they are not exactly the same.

Measurements				
Row identifier	**Timestamp**	**Data**		
		Sensor	Annotation	
Weatherstation1.temp	12:00:00	19.0	Nice weather	
Weatherstation1.temp	12:20:00	20.5	Warm	
Weatherstation1.temp	12:40:00	22.7	Warm	
Weatherstation1.temp	13:00:00	25.1	Warm	Hot
Weatherstation1.temp	13:20:00	23.6	Warm	
Weatherstation1.temp	13:40:00	NULL	Power failure	
Weatherstation1.temp	14:00:00	21.9	NULL	
Weatherstation1.rain	12:00:00	0.0	Nice weather	
Weatherstation1.rain	12:20:00	0.4	Storm	
Weatherstation1.rain	12:40:00	0.5	Storm	
Weatherstation1.rain	13:00:00	0.2	NULL	
Weatherstation1.rain	13:20:00	0.1	Warm rain	
Weatherstation1.rain	13:40:00	NULL	Power failure	
Weatherstation1.rain	14:00:00	0.1	NULL	
Weatherstation1.wind	12:00:00	2.0	NULL	
Weatherstation1.wind	12:20:00	8.0	NULL	
Weatherstation1.wind	12:40:00	10.0	NULL	
Weatherstation1.wind	13:00:00	6.0	NULL	
Weatherstation1.wind	13:20:00	5.0	NULL	
Weatherstation1.wind	13:40:00	NULL	Power failure	
Weatherstation1.wind	14:00:00	4.0	NULL	

Table 9: Example of sensor data and annotation storage in a 'Bigtable' system over time

Table 9 shows an example of what this storage will look like. Multiple sensor stations should be put into the same table in order to overcome the lack of joins. Table 10 shows what the table will look like at a specific time with all 3 weather stations in it, in this case at 12:00:00.

Measurements			
Row identifier	**Data**		
	Sensor	Annotation	
Weatherstation1.temp	19.0	Nice weather	
Weatherstation1.rain	0.0	Nice weather	
Weatherstation1.wind	2.0	NULL	
Weatherstation2.temp	19.9	Nice weather	
Weatherstation2.rain	0.0	NULL	
Weatherstation2.wind	3.0	NULL	
Weatherstation3.temp	19.2	Nice weather	
Weatherstation3.rain	0.0	NULL	
Weatherstation3.wind	1.0	Nice weather	

Table 10: Example of sensor data and annotations in a 'Bigtable' system at one specific time

This storage model is able to store annotations on a row and supports time. Further it should be able to support all types of queries, although some extra logic will be necessary.

6.3. Storage models to be used in scalability experiments

Because time is limited, we will have to make a selection of storage models to be used in our experiments. One of our goals was to use at least 1 non-relational method. Further we will choose 2 relational methods from the found ones. These models will be tested using experiments which will be discussed later on.

6.3.1. Relational storage

Considering relational storage we have chosen for the WHB08 [WHB08] and the BDBMS [EAE09] storage models. Looking at Table 1 and Table 2 we can see BDBMS has the most support for storing and querying annotations. WHB08 uses a different approach to storing annotations and we are curious what its scalability is. With WHB08 we will store sensor data per sensor station and use their 4th annotation storage option as described in the section ‘Annotation storage’ on page 19. For BDBMS extra logic will be needed to work with the annotation rectangles. We will leave out the algorithm to arrange the columns as efficient as possible and do this manually.

6.3.2. Column database storage

As mentioned earlier there were no methods available for column databases and that’s why we have developed our own storage method using Hypertable (an open-source alternative to Bigtable [Goo06]) and its built-in time support. Therefore we will use this method for the experiments.

7. Retrieval of sensor data and annotations

This section will describe the retrieval part of the system. We will start with looking at what kinds of retrieval are desired. Next we will give a metasyntax description of our proposed query language and we will take a look at how to convert our language into SQL and HQL.

7.1. Types of queries

At first we will have to take a look at what types of queries should be supported in the retrieval. In chapter 2.1 we mentioned a few types. Of these types we like to have support for:

- Querying for sensor data, annotations and time (in SELECT)
- Creating constraints on sensor data, annotations and time (in WHERE)
- Combinations of above mentioned types
- Sensor data can be 'simple' (1 type, 1 station), multiple types, multiple stations and both
- Annotation management, create, add, delete and remove annotations

7.2. Proposed query language

Next we will define the query language that will be used to retrieve the sensor data and annotations from the storage. It is inspired by chapter 4.1 from [KMP09]. The metasyntax description of the language is the following:

query = GET TIME? ANNOTATION? "<sensor_data>" FROM "<station>"
(, "<station>")* where
| ADD ANNOTATION "<annotation>" TO "<sensor_data>" AT "<station>"
where
| REMOVE ANNOTATION "<value>" FROM "<sensor_data>" AT
"<station>"
| CREATE ANNOTATION "<value>"
| DELETE ANNOTATION "<value>" ;

where = WHERE constraint ((AND | OR) constraint)* ;

constraint = ("<sensor_data>" | ANNOTATION | TIME) NOT? BETWEEN "<value>"
AND "<value>"
| ("<sensor_data>" | ANNOTATION | TIME) NOT? (> | < | >= | <= | =, <>)
"<value>" ;

Which should support all types of queries as mentioned in the above section. Examples can be found in the section 'Query set for validation' on page 27.

7.3. Translation rules

This section will describe the translation rules to get from the proposed query language to both SQL and HQL. Table 11 shows how we can translate our language to SQL and Table 12 does this for HQL. The left column shows our syntax and the middle column the syntax used by the respective languages. Where necessary comments are placed in the right column.

Query language	SQL	Comments
GET	SELECT	-
TIME (get)	Timestamp*	* What the time field in the database is named like
ANNOTATION (get)	Depending on storage model*	* Depending on how annotations are stored
FROM	FROM	-
WHERE	WHERE	-
AND	AND	Logical AND
OR	OR	Logical OR
ANNOTATION (where)	Depending on storage model*	* Depending on how annotations are stored
TIME (where)	Timestamp*	* What the time field in the database is named like
NOT	NOT	-
BETWEEN	BETWEEN	-
>	>	Greater than
<	<	Less than
>=	>=	Greater than or equal to
<=	<=	Less than or equal to
=	=	Equal to
<>	<>	Not equal to

Table 11: Translation rules for SQL
* See comment

Query language	HQL	Comments
GET	SELECT	-
TIME (get)	DISPLAY_TIMESTAMPS	Returns timestamps
ANNOTATION (get)	Depending on storage model*	* Depending on how annotations are stored
FROM	FROM	-
WHERE	WHERE	
AND	AND	Logical AND
OR	OR	Logical OR. Not available in time predicates?
ANNOTATION (where)	Depending on storage model*	* Depending on how annotations are stored
TIME (where)	TIMESTAMP	-
NOT		Not available in HQL
BETWEEN	< operator <	-
>	>	Greater than
<	<	Less than
>=	>=	Greater than or equal to
<=	<=	Less than or equal to
=	=	Equal to
<>		Not available in HQL

Table 12: Translation rules for HQL
* See comment

8. Scalability testing of storage models

This section will describe the design of an experiment to validate possible storage methods for sensor data and its annotations. For this experiment we will use a subset of the data as produced in IJkdijk [IJk09] experiments and provided by TNO-ICT (part of 'Stichting IJkdijk') to be used in this research. We will start with discussing what parts of the dataset will be used and how we will convert this data into its new storage models. Next we will present a query set which will be used to validate and compare storage models. We will describe what exactly will be tested during the experiments and finally we will present and discuss the results.

8.1. Data set for scalability testing of storage models

TNO-ICT has provided a dataset to be used in this research. This relational dataset contains a part of the sensor measurements from the 2009 IJkdijk experiments. It has a total of 48.524.227 rows. The schema of the table containing this data is as seen in Table 13. The timestamps are the number of milliseconds passed since January 1st 1970.

Column	Description
PK	Primary key
timed	Time added by the University of Twente
ANYSENSETIMESTAMP	Timestamp as given by the Anysense platform
OWNERID	Who owns the sensor producing this data
OWNERTIMESTAMP	Timestamp produced by the sensor
SENSORID	The sensor that produced this data
PARTID	Part of sensor that produced this data
PARTDOUBLE	Value as double
PARTLONG	Value as long
PARTSTRING	Value as string

Table 13: Schema of table containing IJkdijk data

We will select a part of this data to work with. We want to use different sensor stations with multiple sensors/parts. The weatherstation sensor contains 13 parts and a total of 81.046 rows. The luisterbuis sensor (an audio sensor that also measures pressure and temperature) has 4 parts and 1.719.426 rows. We will use these 2 together as our dataset for the experiments. This makes a total of 1.800.472 rows.

8.1.1. Data manipulation

'Appendix A: Data manipulation details' will give more details about the manipulation process including the actual queries used. We will start storing our selected subset into a new table with the exact same schema. Next we create tables for the weather station and the luisterbuis sensor.

After taking a look at the data it seems there is erroneous data. The weather station measurements normally consist of 13 different parts, but in a big amount of rows all these 13 parts have partid 25 and the same value. We deleted these rows, a total of 31.863, leaving us with 49.183 rows. 19 measurements contained more than 13 parts and 115 less. We will leave these in, because data will never be perfect.

The luisterbuis data also contains some erroneous data. 168 measurements contain more than 4 parts and once again this is duplicate data at the same timestamp. 529 measurements contain less than 4 parts. We will again leave this data in. This makes a total of 1.768.609 rows.

The actual manipulation from the subset table to the sensor station specific tables will be done using PHP. A script will query all data for a sensor station and extract the values for all partid's of a single measurement. This measurement will then be inserted again into the new table. Missing parts will be inserted as nulls. In the case of Hypertable the luisterbuis and weatherstation tables were used and each column was inserted as a row with the ownertimestamp as timestamp.

For relational data the manipulation resulted in 3.831 rows in the weatherstation table and 429.986 rows in the luisterbuis table. This makes a total of 433.817 rows. For Hypertable it was 49.803 rows for the weatherstation and 279.276 for the luisterbuis making a total of 329.079 rows.

8.1.2. Adding annotations

Next we will have to add annotations to the data. We will start creating the tables necessary for storing them. 'Appendix B: Annotation adding details' describes the actual queries used. For the [WHB08] method, both the weather station and luisterbuis get a table for their annotations. Further we create the actually annotation table and a comments table. In the case of BDBMS each sensor station also gets an annotation table, but no additional tables are needed.

We will add all annotations from the case, except for lightning and power failure, to both storage models. Further we added 3 annotations for the luisterbuis data. For [WHB08] this means adding the annotations and attaching them to data using a query. For BDBMS this means creating rectangles, which will be done automated. The manually chosen column mapping for weatherstation is as follows: PK=1, OWNERTIMESTAMP=2, WINDSPEEDAVG=3, TEMPAVG=4 and RAINFALLTOTAL=5. For luisterbuis this is: PRESSUREA=1, PRESSUREB=2, TEMPA=3, TEMPB=4. The other columns are not used in our annotations.

The results were that WHB08 got 530, 15.825 and 158.385 annotated for respectively the weatherstation, luisterbuissubset and luisterbuis. For BDBMS the number were 125, 7.649 and 76.356 rows containing rectangles.

For Hypertable we can only attach annotations to a specific sensor. For the weatherstation we will add the annotations 'hot', 'warm' and 'cold' to the 'tempavg' sensor. Further we will add 'heavy rain' (rain > 0.3) and 'storm' (wind > 6) to the 'rainfalltotal' and the 'windspeedavg' sensors. Together they form the original 'storm' annotation from our case. The luisterbuis will receive the same annotations as the relational luisterbuis data.

The results were 931 rows of annotations for the weatherstation, 54.885 for the luisterbuissubset and 276.494 for the whole luisterbuis table.

8.2. Query set for validation

In order to test and validate the storage model, we will need a query set. This set of queries should represent the majority of possible queries over our data. Section 7.1 presented an overview of what types of queries should be supported in the query language. That list is the basis of the query set.

For our experiments we will manually translate the queries from our language to SQL for both WHB08 and BDBMS and to HQL for Hypertable. Some translations will need extra logic to produce the desired result.

Query 1, create an annotation (annotation management):

```
CREATE ANNOTATION 'cold';
```

WHB08:

```
INSERT INTO annotations(TIME, VALUE) VALUES(unix_timestamp(now()), 'cold');
```

BDBMS and Hypertable:

There are no separate tables for annotations in BDBMS and Hypertable.

Query 2, delete an annotation (annotation management):

```
DELETE ANNOTATION 'cold';
```

WHB08:

```
DELETE FROM annotations WHERE STRCMP(VALUE, 'cold');
```

BDBMS and Hypertable:

There are no separate tables for annotations in BDBMS and Hypertable.

Query 3, add an annotation to data (annotation management):

```
ADD ANNOTATION 'cold' TO 'weatherstation' AT 'tempavg' WHERE tempavg < 5;
```

WHB08:

```
SELECT * FROM annotations WHERE STRCMP(value, 'cold') = 0;
```

If not existing yet:

```
INSERT INTO annotations(TIME, VALUE) VALUES(unix_timestamp(now()), 'cold');
```

Always execute:

```
INSERT INTO weatherstationannot(QUERY, TIME, SENSORID, ANNOTATIONID)
values('tempavg < 5', unix_timestamp(now()), 'tempavg', '6');
```

Annotationid comes from either the select or the insert.

BDBMS:

BDBMS will have to select all rows with tempavg < 5, create rectangles of them and then:

```
INSERT INTO weatherstationannot2(annotationid, time, value, rectangle)
values('6', unix_timestamp(now()), 'cold', ' ((4,3749),(4,3777))');
```

For each rectangle created.

Hypertable:

Select all data from the sensor we want to annotate:

```
SELECT * FROM weatherstationd WHERE
cell='weatherstation1.tempavg','timeline:base';
```

If it meets the annotation condition, insert the annotation into the column ‘timeline:annotation’ at the timestamp of the data.

Query 4, remove annotation (annotation management):

```
REMOVE ANNOTATION 'cold' FROM 'tempavg' AT 'weatherstation';
```

WHB08:

```
DELETE FROM weatherstationannot WHERE STRCMP(SENSORID, 'tempavg') = 0 AND
ANNOTATIONID IN (SELECT ANNOTATIONID FROM annotations WHERE STRCMP(VALUE,
'cold') = 0);
```

BDBMS:

```
DELETE FROM weatherstationannot2 WHERE STRCMP(VALUE, 'cold') = 0;
```

Advanced logic is necessary to delete the annotation only from the 'tempavg' column.

Hypertable:

Because in Hypertable a delete will not only delete the data at the given timestamp, but also all earlier timestamps of that cell, only 1 delete query will delete all annotations. This is already very impractical, but after a delete it is also not possible to immediately insert the same data again, making it even harder to test this aspect of Hypertable. Our solution is to add a unique annotation each time with query 3, then delete that one again with this query and start over with a new unique annotation. The delete will be done the following way:

```
SELECT * FROM weatherstationd WHERE
cell='weatherstation1.tempavg','timeline:annotation';
```

Then check if the value = ‘cold’ and using that timestamp execute:

```
DELETE timeline:annotation FROM weatherstationd WHERE
row='weatherstation1.tempavg' TIMESTAMP timestamp;
```

Query 5, get 'simple' data with sensor data constraint (select 'simple' + data constraint):

```
GET tempavg FROM weatherstation WHERE tempavg > 10;
```

WHB08 and BDBMS:

```
SELECT tempavg FROM weatherstation WHERE tempavg > 10;
```

Hypertable:

```
SELECT * FROM weatherstationd WHERE cell='weatherstation1.tempavg',
timeline.base';
```

Then filter all resulting cells with value > 10;

Query 6, get sensor data with time and annotations with sensor data constraint (select 'simple' and annotations + data constraint):

```
GET TIME ANNOTATION tempavg FROM weatherstation WHERE tempavg > 25;
```

WHB08:

At first all queries attached to tempavg will be selected and looped through:

```
SELECT query, annotationid FROM weatherstationannot WHERE STRCMP('tempavg',
sensorid) = 0;
```

Next we will execute the following query with the query column of the result of the first query attached:

```
SELECT ownertimestamp, tempavg FROM weatherstation WHERE tempavg > 25 AND
{query};
```

If this query produces at least 1 result, we will get the annotation using:

```
SELECT value FROM annotations WHERE annotationid = {annotationid};
```

Where {annotationid} is the id of the annotation as returned by the first query.

BDBMS:

```
SELECT ownertimestamp, tempavg FROM weatherstation WHERE tempavg > 25;
```

For these rows it must be checked whether they are inside a rectangle and attach the annotation(s) to the result.

Hypertable:

```
SELECT * FROM weatherstationd WHERE
cell='weatherstation1.tempavg','timeline:base';
```

Then filter all resulting cells with value > 25 and also retrieve annotations belonging to those.

Query 7, get time and sensor data consisting of multiple parts with annotation constraint (select multiple sensors + annotation constraint):

```
GET TIME tempavg, tempmax, tempmin FROM weatherstation WHERE ANNOTATION =
'hot';
```

WHB08:

```
SELECT a.value, wa.query FROM weatherstationannot as wa, annotations as a
WHERE STRCMP(wa.query, 'tempavg > 25') = 0 AND wa.annotationid =
a.annotationid;

SELECT ownertimestamp, tempavg, tempmax, tempmin FROM weatherstation as w,
weatherstationannot = wa WHERE w.{result->wa.query};
```

BDBMS:

```
SELECT rectangle FROM weatherstationannot2 WHERE STRCMP(value, 'hot') = 0;
```

The rectangle(s) have to be converted to rows and colums and then we query:

```
SELECT ownertimestamp, tempavg, tempmax, tempmin FROM weatherstation WHERE
pk = row(s);
```

Hypertable:

```
SELECT * FROM weatherstationd WHERE
cell='weatherstation1.tempavg','timeline:annotation' and
cell='weatherstation1.tempmax','timeline:annotation' and
cell='weatherstation1.tempmin','timeline:annotation';
```

Compare with ‘hot’ and if it matches, get data at that timestamp.

Query 8, get time and sensor data with time constraint (select ‘simple’ + time constraint):

```
GET TIME tempavg FROM weatherstation WHERE TIME BETWEEN '1253358800000' AND
'1253369970000';
```

WHB08 and BDBMS:

```
SELECT ownertimestamp, tempavg FROM weatherstation WHERE ownertimestamp
BETWEEN '1253358800000' AND '1253369970000';
```

Hypertable:

```
SELECT * FROM weatherstationd WHERE
cell='weatherstation1.tempavg','timeline:base' and '2009-09-19 13:13:20' <
TIMESTAMP < '2009-09-19 16:19:30';
```

Query 9, get annotations, time and sensor data in a certain time range (select 'simple' and annotations + time constraint):

```
GET TIME ANNOTATION tempavg FROM weatherstation WHERE TIME BETWEEN
'1253358800000' AND '1253369970000';
```

WHB08:

First we select the queries and values of all annotations and loop through them:

```
SELECT wa.query as q, a.value as v FROM weatherstationannot as wa,
annotations as a WHERE wa.annotationid = a.annotationid;
```

Next we execute the following query where {query} comes from the result of the first query:

```
SELECT pk, ownertimestamp FROM weatherstation WHERE {query} AND
ownertimestamp BETWEEN '1253358800000' AND '1253369970000';
```

And then we can display all data with that annotation within the given time range.

BDBMS:

For BDBMS the rows from weatherstation will have to be selected and then check if the row numbers are inside a rectangle.

Hypertable:

```
SELECT * FROM weatherstationd WHERE
cell='weatherstation1.tempavg','timeline:base' AND '2009-09-19 13:13:20' <
TIMESTAMP < '2009-09-19 16:19:30';
```

Loop through data and query annotations at the same timestamp as the data.

Query 10, get time and sensor data from multiple stations (select multiple stations):

```
GET TIME tempavg, tempa FROM weatherstation, luisterbuis WHERE tempavg =
tempa;
```

WHB08 and BDBMS:

```
SELECT w.ownertimestamp, tempavg, l.ownertimestamp, tempa FROM
weatherstation as w, luisterbuis as l WHERE w.tempavg = l.tempa;
```

Hypertable:

```
SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:base';
SELECT * FROM weatherstationd WHERE
cell='weatherstation1.tempavg','timeline:base';
```

Manually join the results and find timestamps where tempavg = tempa.

8.3. Experiment design

We have selected a dataset and created a query set. Now we will define what exactly will be done. Our goal is to test the systems in terms of scalability to find out whether a data set that is 10 times larger produces query execution times that are 10 times bigger.

Most queries of [WHB08] can directly be translated to SQL, but for BDBMS this is not always possible. We will have to create some extra logic to be able to translate these queries and this will be done using PHP. We will use MySQL for the relational data. Two files were created, one for each relational storage model. Each query is put into a function and using a while loop we can execute the queries we want as many times as we want. Where necessary the MySQL query cache was emptied. The resulting average execution time per query is then outputted to the screen. Time will be measured in 'wall clock time', which means the total time used by the CPU, I/O and communication channels (which is equivalent to the real time passed). Further we will calculate the standard deviation for the produced results. In the case of BDBMS extra functions were created to deal with the annotation rectangles. In PHP time is measured using the 'microtime' function, which produces the time in microseconds.

For Hypertable we will use C++ as a programming language to directly communicate with the database. Using a program called Thrift it is possible to use other languages like PHP and Java, but this will severely impact the performance and scalability. In C++ time is measured using the 'gettimeofday' function, which also produces the time in microseconds.

For all systems we will try to run the 10 queries 1000 times on data set size 1, 100 times on data set size 2 and 10 times on data set size 3. This will be done because some queries will start taking a lot of time at larger data sets. More on these data set sizes can be found in the next section.

8.3.1. Data set sizes

In order to measure scalability we will have to run our tests on different scales. We can scale two things, data and annotations. In these experiments we will only scale the data, but as the data grows the annotations will grow along. The preferred data set sizes are 10.000, 100.000 and 1.000.000, but currently our dataset only consists of 433.000 tuples. For now we will use this set, but for the real scalability tests we will either receive more data from TNO or expand the data ourselves to 1 million. The three data set sizes will all get their own table. The smallest data set size will be the weatherstation data, consisting of 3.831 rows. The middle data set size will consist of a subset of the luisterbuis data (42.999 rows) and the largest data set size will be the whole luisterbuis table containing 429.986 rows. This distribution of scales means we will have to use different queries for the largest 2 data sets compared to the smallest. These queries can be found in 'Appendix C: Queries for larger data sets using luisterbuis'. Query 10 is only possible on the larger data sets since it involves both the weatherstation and luisterbuis data.

For Hypertable the weatherstation has 49.803 cells, the luisterbuis subset 171.996 and the luisterbuis 776.871. The numbers are larger because each sensor is a row instead of a column. The largest data set is also larger because we duplicated data in order to get a better scale.

Table 14 shows an overview of the 3 data set sizes for both relational models and Hypertable.

	Data set size 1	Data set size 2	Data set size 3
Rows of relational data	3.831	42.999	429.986
Rows of WHB08 annotations	530	15.825	158.385
Rows of BDBMS annotations	125	7.649	76.356
Rows of Hypertable data	49.803	171.996	776.871
Rows of Hypertable annotations	931	54.885	276.494

Table 14: Overview of data set sizes

8.4. Results

This section will talk about the results of the experiments for all three storage models. At first we will give the 10 queries more meaningful names than just 'Query x'. The annotation management queries get short versions of their full names. For the selection queries this will not work and therefore we created abbreviations with the following format: 'Query number. Select(S): Data(D)/Annotation(A)/Time(T)/Constraint on(C): Data(D)/Annotation(A)/Time(T)'. These abbreviations show what a query selects and which constraints it has. Table 15 and Table 16 show the mapping of queries to these abbreviations.

	Create_annot	Delete_annot	Add_annot	Remove_annot
Query number	1	2	3	4

Table 15: Mapping of annotation management queries to abbreviations

CONSTRAINT ON / SELECT	Data	Annotations	Time
Data	5, 6, 10	7	8, 9
Annotations	6	x	9
Time	6, 10	7	8, 9

Table 16: Mapping of selection queries to abbreviations

Next we will take a look at what exactly each query does. Table 17 shows this for the relational data and for all 10 queries on the 3 scales. The standard deviations for all measurements of all models can be found in 'Appendix E: Measured standard deviations for experiments'. We will start with the results by taking a look at the results of WHB08.

Query	Data set size 1	Data set size 2	Data set size 3
1. Create_annot	Insert 1 row	Insert 1 row	Insert 1 row
2. Delete_annot	Delete 1 row	Delete 1 row	Delete 1 row
3. Add_annot	Insert 1 row	Insert 1 row	Insert 1 row
4. Remove_annot	Delete 1 row	Delete 1 row	Delete 1 row
5. S: D/C: D	Select 3076 rows	Select 29.767 rows	Select 298.116 rows
6. S: DAT/C: D	Select 588 rows, 3 tables involved	Select 5342 rows, 3 tables involved	Select 52.793 rows, 3 table involved
7. S: DT/C: A	Select few annotation rows + 20 rows	Select few annotation rows + 61 rows	Select few annotation rows + 617 rows
8. S: DT/C: T	Select 304 rows	Select 3169 rows	Select 31687 rows
9: S: DAT/C: T	Select 174 rows	Select 1269 rows	Select 12690 rows
10: S: DT/C: D	*	Select 588 rows, 2 tables involved	Select 5782 rows, 2 tables involved

Table 17: What each query does on every scale
* Not applicable

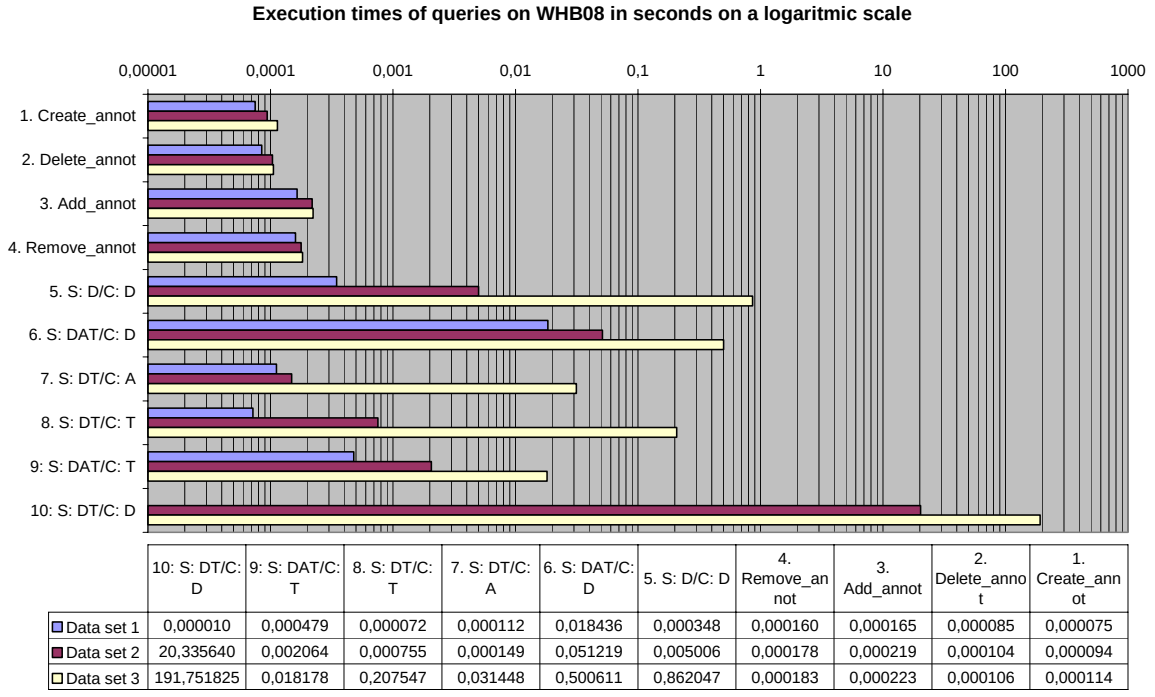
8.4.1. WHB08

Table 18 and Figure 4 show the results of the experiments with WHB08 data and annotations. Query 10 was not run on data set 1 because it involves both the weatherstation and luisterbuis data and therefore is at least data set 2. The column "1<>2" shows the difference between the execution times of data sets 1 and 2. The column "2<>3" does the same for data sets 2 and 3.

Creating, inserting, deleting and removing are very scalable processes. Queries 6, 9 and 10 have a good scalability and 5, 7 and 8 a poor scalability. In general the standard deviation was low, but on scale 1 it was high, making the "1<>2" column less trustworthy. Also some measurements have such low execution times that the standard deviation becomes less relevant due to the accuracy of the measurements.

Query	Data set 1 [s]	1<>2	Data set 2 [s]	2<>3	Data set 3 [s]
1. Create_annot	0.000075	*1.25x*	0.000094	*1.21x*	0.000114
2. Delete_annot	0.000085	*1.22x*	0.000104	*1.02x*	0.000106
3. Add_annot	0.000165	*1.33x*	0.000219	*1.02x*	0.000223
4. Remove_annot	0.000160	*1.11x*	0.000178	*1.03x*	0.000183
5. S: D/C: D	0.000348	*14.39x*	0.005006	*172.20x*	0.862047
6. S: DAT/C: D	0.018436	*2.78x*	0.051219	*9.78x*	0.500611
7. S: DT/C: A	0.000112	*1.33x*	0.000149	*211.06x*	0.031448
8. S: DT/C: T	0.000072	*10.49x*	0.000755	*274.90x*	0.207547
9: S: DAT/C: T	0.000479	*4.31x*	0.002064	*15.82x*	0.018178
10: S: DT/C: D	*	*	20.335640	*8.81x*	191.751825
PHP max memory	190.376 bytes	*1.00x*	190.376 bytes	*1.00x*	190.376 bytes

Table 18: Execution times and peak memory usage of queries on WHB08 in seconds and bytes
* Not applicable



	10: S: DT/C: D	9: S: DAT/C: T	8. S: DT/C: T	7. S: DT/C: A	6. S: DAT/C: D	5. S: D/C: D	4. Remove_annot	3. Add_annot	2. Delete_annot	1. Create_annot
Data set 1	0,000010	0,000479	0,000072	0,000112	0,018436	0,000348	0,000160	0,000165	0,000085	0,000075
Data set 2	20,335640	0,002064	0,000755	0,000149	0,051219	0,005006	0,000178	0,000219	0,000104	0,000094
Data set 3	191,751825	0,018178	0,207547	0,031448	0,500611	0,862047	0,000183	0,000223	0,000106	0,000114

Figure 4: Graphical view of WHB08 measurements

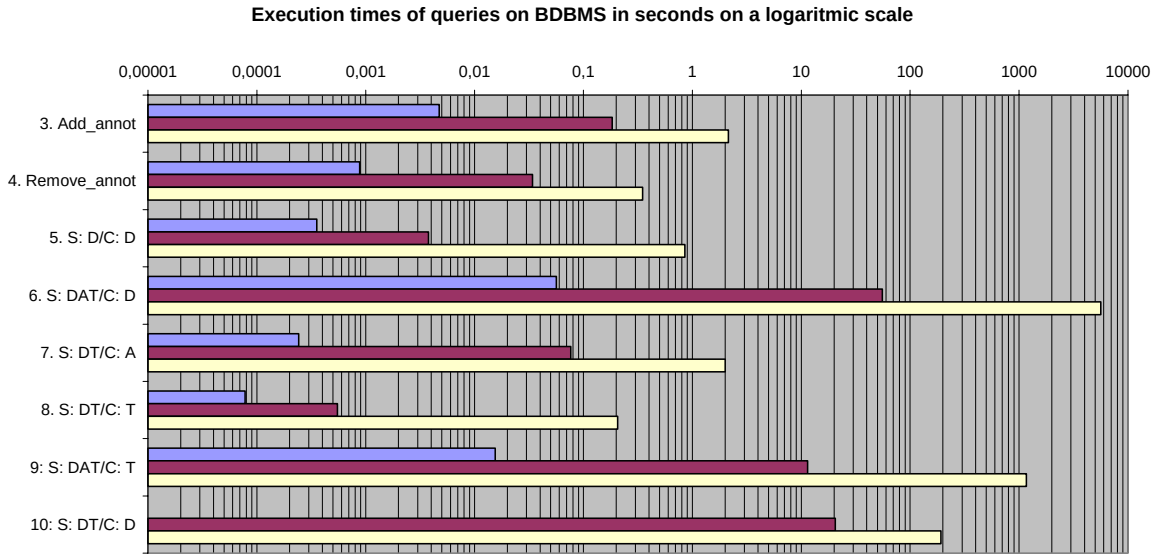
8.4.2. BDBMS

Table 19 and Figure 5 show the results of the experiments with BDBMS data and annotations. Query 1 and 2 were not run because BDBMS does not have the annotations separately stored, the annotations are simply linked to the data directly. Query 10 was not run on data set 1 because it involves both the weatherstation and luisterbuis data and therefore is at least data set 2. The column "1<>2" shows the difference between the execution times of data sets 1 and 2. The column "2<>3" does the same for data sets 2 and 3.

The insertion and removal of annotations for BDBMS is less scalable than WHB08, but it is still acceptable. Queries 7 and 10 are the only other queries with a good scalability. Queries 5 and 8 have a reasonable scalability and 6 and 9 a bad scalability. The absolute numbers of query 6 and 9 get very high for data set 3. The standard deviation was overall low.

Query	Data set 1 [s]	1<>2	Data set 2 [s]	2<>3	Data set 3 [s]
1. Create_annot	*	*	*	*	*
2. Delete_annot	*	*	*	*	*
3. Add_annot	0.004724	*38.85x*	0.183509	*11.66x*	2.140202
4. Remove_annot	0.000881	*38.63x*	0.034033	*10.27x*	0.349678
5. S: D/C: D	0.000356	*10.58x*	0.003765	*225.90x*	0.850495
6. S: DAT/C: D	0.056388	*983.18x*	55.439357	*101.60x*	5632.72
7. S: DT/C: A	0.000242	*315.45x*	0.076339	*26.22x*	2.001602
8. S: DT/C: T	0.000078	*7.05x*	0.000550	*374.74x*	0.206108
9: S: DAT/C: T	0.015473	*738.58x*	11.428085	*102.39x*	1170.066720
10: S: DT/C: D	*	*	20.504747	*9.34x*	191.569171
PHP max memory	1.702.120 bytes	*8.00x*	13.617.968 bytes	*9.67x*	131.615.152 bytes

Table 19: Execution times and peak memory storage of queries on BDBMS in seconds and bytes
* Not applicable



	10: S: DT/C: D	9: S: DAT/C: T	8. S: DT/C: T	7. S: DT/C: A	6. S: DAT/C: D	5. S: D/C: D	4. Remove_annot	3. Add_annot
Data set 1	0,000010	0,015473	0,000078	0,000242	0,056388	0,000356	0,000881	0,004724
Data set 2	20,504747	11,428085	0,000550	0,076339	55,439357	0,003765	0,034033	0,183509
Data set 3	191,569171	1170,066720	0,206108	2,001602	5632,720000	0,850495	0,349678	2,140202

Figure 5: Graphical view of BDBMS measurements

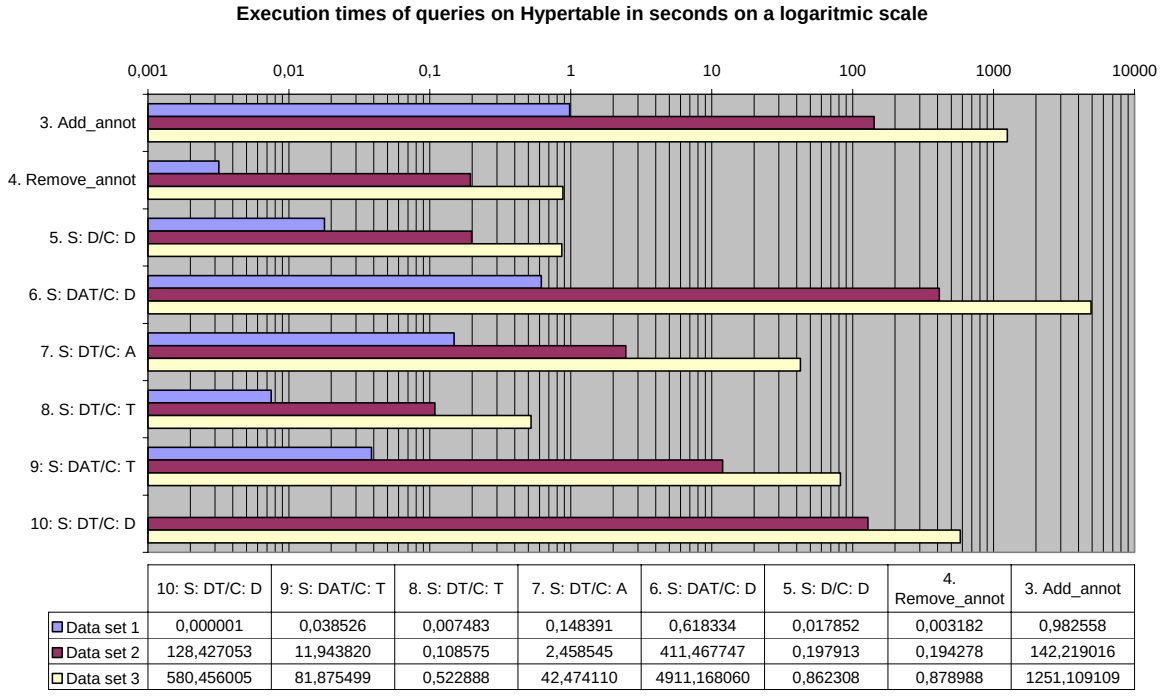
8.4.3. Hypertable

Table 20 and Figure 6 show the results of the experiments with Hypertable data and annotations. Query 1 and 2 were not run because Hypertable does not have the annotations separately stored, the annotations are simply linked to the data directly. Query 10 was not run on data set 1 because it involves both the weatherstation and luisterbuis data and therefore is at least data set 2. The column "1<>2" shows the difference between the execution times of data sets 1 and 2. The column "2<>3" does the same for data sets 2 and 3.

Most queries have a good scalability. Query 6 is an exception because again on data set 3 the absolute numbers become huge. Query 10 has a good scalability, but the absolute numbers are also pretty high. All standard deviations were low. The insertion and removal of annotations is a less scalable process than it is for the relational systems. Especially inserting gives big absolute numbers. Deleting seems to be pretty scalable.

Query	Data set 1 [s]	1<>2	Data set 2 [s]	2<>3	Data set 3 [s]
1. Create_annot	*	*	*	*	*
2. Delete_annot	*	*	*	*	*
3. Add_annot	0.982558	*144.74x*	142.219016	*8.80x*	1251.109109
4. Remove_annot	0.003182	*61.06x*	0.194278	*4.52x*	0.878988
5. S: D/C: D	0.017852	*11.09x*	0.197913	*4.36x*	0.862308
6. S: DAT/C: D	0.618334	*665.45x*	411.467747	*11.94x*	4911.168060
7. S: DT/C: A	0.148391	*16.57x*	2.458545	*17.28x*	42.474110
8. S: DT/C: T	0.007483	*14.51x*	0.108575	*4.82x*	0.522888
9: S: DAT/C: T	0.038526	*310.02x*	11.943820	*6.86x*	81.875499
10: S: DT/C: D	*	*	128.427053	*4.52x*	580.456005

Table 20: Execution times of queries on Hypertable in seconds
* Not applicable



	10: S: DT/C: D	9: S: DAT/C: T	8. S: DT/C: T	7. S: DT/C: A	6. S: DAT/C: D	5. S: D/C: D	4. Remove_annot	3. Add_annot
Data set 1	0,000001	0,038526	0,007483	0,148391	0,618334	0,017852	0,003182	0,982558
Data set 2	128,427053	11,943820	0,108575	2,458545	411,467747	0,197913	0,194278	142,219016
Data set 3	580,456005	81,875499	0,522888	42,474110	4911,168060	0,862308	0,878988	1251,109109

Figure 6: Graphical view of Hypertable measurements

8.5. Conclusions

In general the annotation management for the relational systems is a scalable process, but for BDBMS the absolute numbers are a lot higher than for WHB08. For Hypertable deleting gave no problems, but inserting the annotations was very slow. Query 5 scales bad on WHB08 and BDBMS, but very well on Hypertable. Query 6 scales bad on BDBMS and performed bad on Hypertable because the absolute numbers become huge. Query 7 scaled bad on WHB08, but with low absolute numbers. It scored well on the other systems. Query 8 had a poor scalability on the relational systems, but no high absolute numbers yet on data set 3. Query 9 scaled well on WHB08 and Hypertable, but very bad for BDBMS. The absolute numbers became huge for BDBMS. Query 10 scaled reasonable, but the absolute numbers were high for all systems. For the relational systems it was clearly limited by the database itself. WHB08 barely uses any memory in PHP because not much extra logic is needed. BDBMS does, but the amount of memory needed scales well. Unfortunately we were unable to measure the memory usage for Hypertable, but because extra logic is needed here a well we expect a reasonable amount of memory was used.

Figure 7 and Figure 8 show a direct comparison between both relational models. Both figures contain all queries for both systems on the vertical axis and the horizontal axis shows the time as a percentage of the total time (execution time of sets 1+2+3=100%). Figure 7 does this on a normal scale, Figure 8 on a logarithmic scale. The reference bar shows what the graph looks like for the execution times of a query where data set 2 takes 10 times the time of data set 1 and data set 3 10 times the time of data set 2 (so a linear scalability). For Figure 7 goes, the more even the 3 coloured bars are distributed, the better the scalability is. In Figure 8 data set 1 should be large and data set 3 small for maximal scalability. Next we will discuss our conclusions for each storage model and after that we draw our final conclusions.

WHB08 scored very well overall. This can be explained by the minimum amount of extra logic needed to produce the results. Queries 5, 7 and 8 had a poor scalability on the large scales though, but the absolute numbers stayed low.

BDBMS scored poorly overall. The reduction of storage requirements means the scalability becomes very bad for certain queries due to the extra logic needed for handling rectangles. It does have the most advanced annotation capabilities of all three systems though.

Hypertable scored well in terms of scalability, but the absolute numbers became way too high for some queries. The gain of a faster database is diminished by the extra logic needed to produce the same results. This clearly became a bottleneck at the higher scales.
A few other disadvantages of the current implementation of Hypertable were the lack of regular expressions and data types (only strings can be stored). Our storage model also cannot delete single annotations at a certain timestamp, because such a delete causes all other entries in the annotation column to be deleted together with all older ‘versions’ of that column.

So while the database itself is very fast, the simplicity of it requires a lot of post-processing, making it only slower in the end. The expressiveness and flexibility of annotations is also a lot lower than in both relational systems making it an impractical solution in its current implementation.

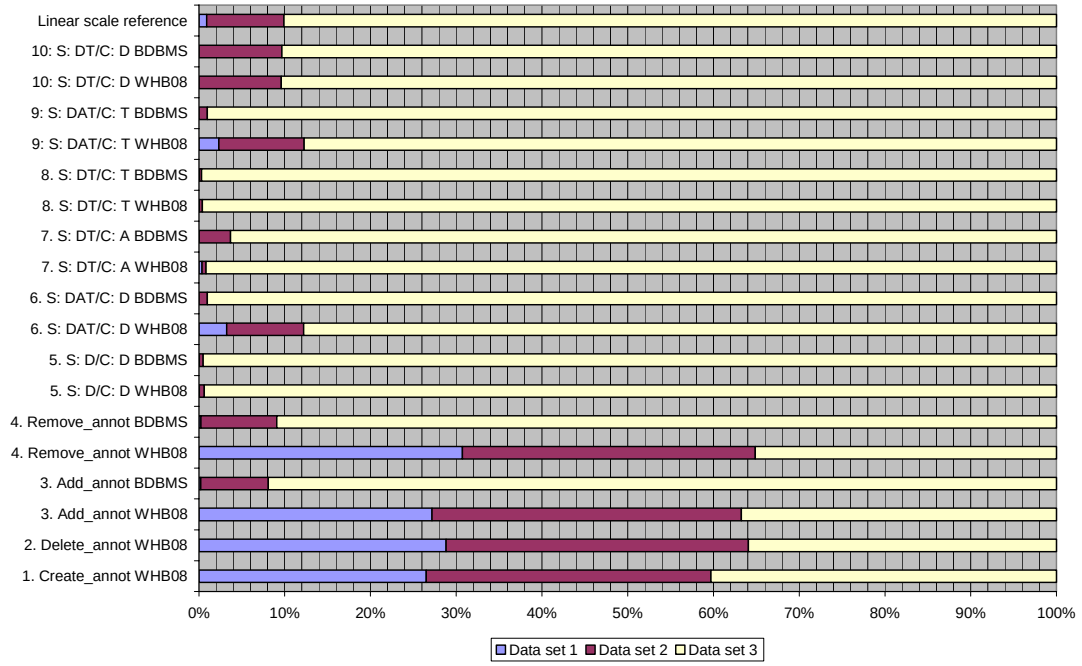


Figure 7: Comparison between WHB08 and BDBMS on a normal scale

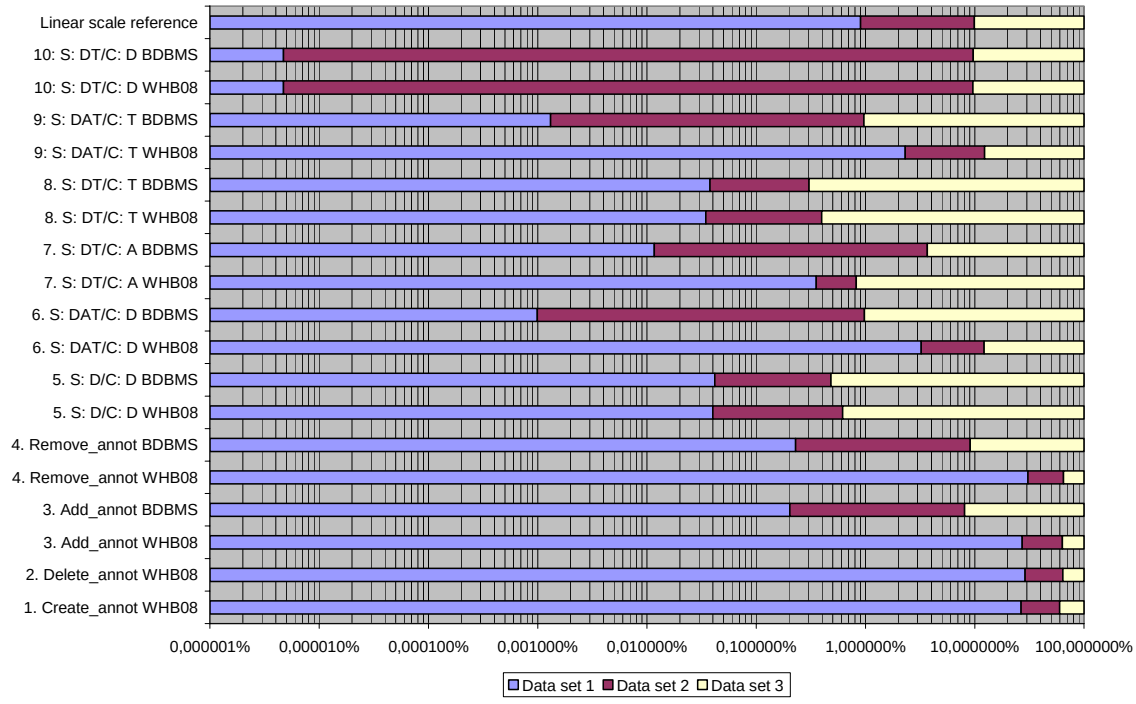


Figure 8: Comparison between WHB08 and BDBMS on a logaritmic scale

8.5.1. Final conclusions of scalability testing

Although BDBMS provides more flexibility regarding annotations, it is also less scalable and needs a lot of extra logic to work with the rectangles. Hypertable does not only miss this flexibility, but also produces high absolute numbers on the higher scales and needs a lot of extra logic. For these reasons we like to further investigate the WHB08 storage model, which scored very well overall, while still providing some interesting annotation capabilities.

9. Implementation

The next step in our research is to create an implementation of WHB08 (which was chosen as the best candidate in the previous chapter) and run the scalability tests again. While during the scalability experiments only a fixed (static) set of queries could be executed, the implementation should be able to execute all possible queries (dynamic). It should also contain some form of display for interaction with the user. Because time is limited and implementing a whole system is time consuming, we will integrate the WHB08 annotation storage model into an existing system.

This existing system will be the sensordataweb, as described in [Lan09], which is a distributed infrastructure to handle and process sensor data including their provenance data. It consists of a query manager, query language (PASN-QL) and a provenance server (Tupelo2). It is currently not possible to query on/with annotations and together with the fact that it is open-source this is an ideal candidate to integrate the WHB08 annotation storage into.

Figure 9 [Lan09] gives an overview of the sensordataweb. The query manager consists of processing elements (PE) which produce output called views (V). Each processing element is a node in the network created by a command (data, project, select, union or join). A node can have a parent, which means it uses the data as produced by that node. This way it is possible to trace back the path data travelled (which is provenance information). The views of these nodes can be queried (using the show command) and handle the database management by calling methods of DBManager classes, which handle the actual communication with the database. This way the system is database-independent. Further there are data sinks for retrieval and formatting of requested data. Next we will take a look at the impact of annotation support on both the retrieval and storage side of the sensordataweb.

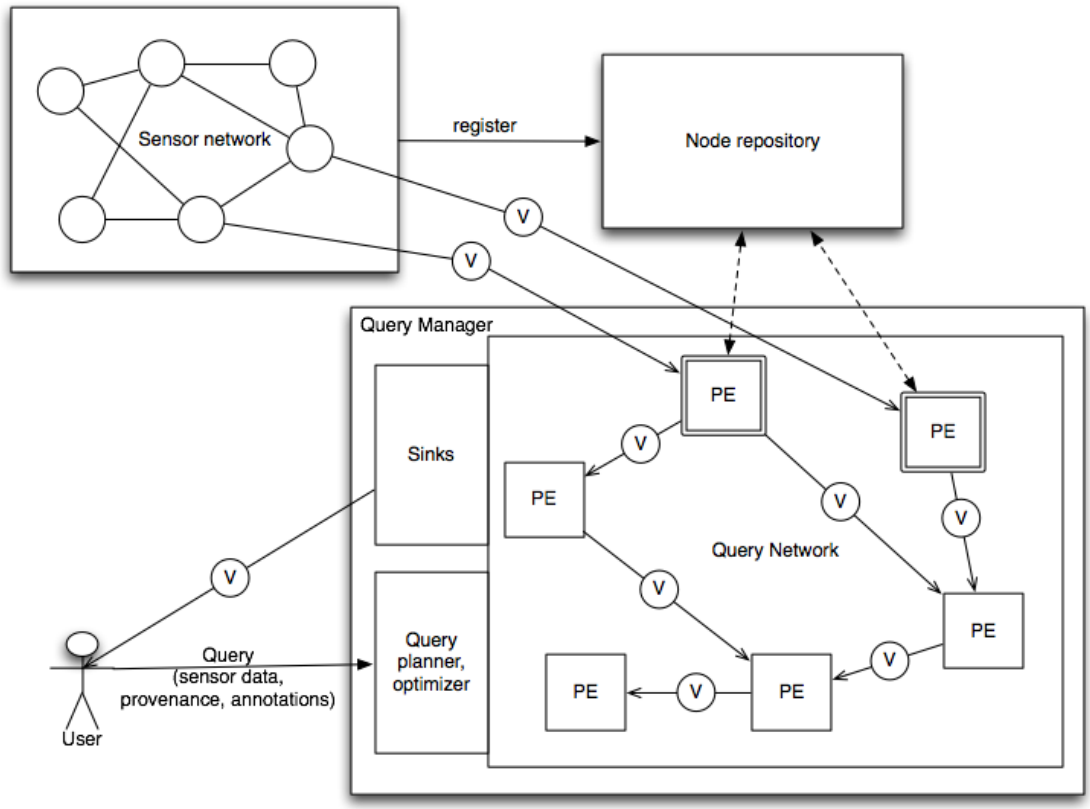


Figure 9: Overview of elements within the sensordataweb [Lan09]

9.1. Implementation requirements

This section will describe the requirements for the implementation of annotation support in the sensordataweb. We will start looking at the requirements for the query language.

9.1.1. Retrieval requirements

In order to reach annotation support using WHB08, the sensordataweb will have to support annotations in its query language (PASN-QL, as defined in [Lan09] on page 55/56). The following support is desired:

- Adding of annotations to data using a query (and if necessary, creation of an annotation)
- Removing data from annotations (removing the query)
- Selecting not only the data but also the annotations belonging to it
- Constraints on annotations in selection queries

In order to show annotations our annotation enabled view will need to not only give data to the sinks but also the annotations belonging to it. For selecting on annotations we have to change the 'IntervalSelector' class to not only have alphanumeric support but also annotation support. We assume the actual annotations will never be deleted using the query language. The first 2 are completely new operations and therefore require their own assign-methods:

```
add_annot ( String PE , String Annotation , String Query )
remove_annot ( String PE , String Annotation )
```

This functionality will have to be supported in the query language, parser and the actual commands will have to be implemented together with an annotation enabled view.

9.1.2. Storage requirements

In order to store the annotations the sensordataweb database will have to be altered. The [WHB08] annotation storage model assumes sensor data is stored per sensor station. For example all measurements coming from a weather station (temperature, rainfall, wind speed etc.) are stored in a table with the name of the sensor station and the sensors as columns.

Therefore all these sensor station tables will need their own annotation table. For example the 'weatherstation' table will need a 'weatherstation_annot' table. In the sensordataweb sources and PE's (to be exactly their views) will need annotation tables. In order to store the actual annotations and comments on them the tables 'annotations' and 'comments' will have to be created. More details about all tables can be found in 'Appendix A: Data manipulation details' and 'Appendix B: Annotation adding details'.

9.2. Implementation details

As mentioned earlier on the query manager consists of processing elements and views. The first does something with the data and the second can be queried and manages the storage of the data. In order to support annotations these views will have to support them. This means we will have to create a special type of view which is used whenever it is specified by the user that annotations are involved. This will be the case when annotations are added by the user. Further we will need to create classes to validate and execute the 'add annotation' and 'remove annotation' commands. Figure 10 gives an overview of the classes involved for annotation support in the sensordataweb.

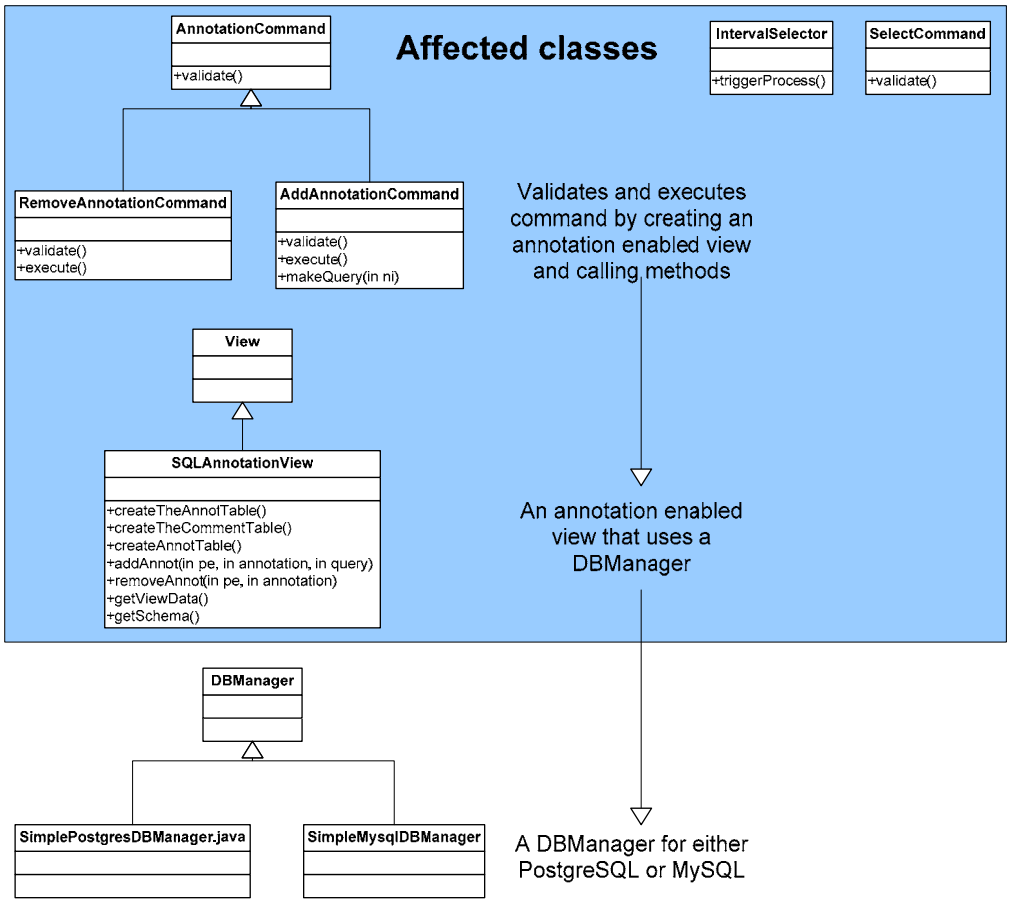


Figure 10: Overview of classes involved for annotation support

When a user sends a 'remove_annot' or 'add_annot command' to the sensordataweb, the system parses this and calls the respective commands, first to validate the command and if this succeeds to execute it. The validation of 'add_annot' checks if there is already an annotation enabled view present and if not, creates it (SQLAnnotationView). For 'remove_annot' it is checked whether the view of the PE is annotation enabled. If this is not the case annotations have not been added for sure and it is not possible to remove an annotation. On execution the 'remove_annot' or 'add_annot' method of the annotation enabled view is called.

In order to make the annotation support both database and sink independent, the decision was made to add all annotation support to the view and not to the DBManager classes. This way it doesn't matter which database or sink is used. The annotation enabled view can add and remove annotations using the methods as provided by the DBManager classes. When a sink asks for its data the view will also return the annotations present. In order to reach sink-independency it was necessary for the method 'getSchema' in the SQLAnnotationView class to behave differently when called by a sink class. This method returns the database schema used and when annotations are involved it is necessary to also return an annotation column in the schema so the sink knows it also has the show the annotations.

In order to get a selection with an annotation constraint it was necessary to change the class which decides if a tuple is selected (IntervalSelector) and to allow annotation conditions in the SelectCommand class. The IntervalSelector class had to be able to recognize there was an annotation constraint and then check if the new tuple fulfilled the constraint. It supports multiple attachments of 1 annotation to the data (i.e. 2 definitions of 1 annotation or a split definition). For example an annotation on a certain node can be defined as 'value > 0.0 and value < 0.2' but also as 'value > 0.8 and value < 1.0'. If this annotation is put as a selection constraint the IntervalSelector knows the value has to be in 1 of those 2 ranges.

The required tables in the database are the global 'annotation' and 'comments' tables and the PE-specific tables. The first are created on creation of a new SQLAnnotationView and using a static variable it is assured this only happens once. When the tables already existed nothing happens. The latter are also created on startup of a new SQLAnnotationView, but will always be created whenever a PE creates an annotation enabled view.

9.3. Implementation limitations

Due to time constraints the implementation has a limitation:

- Annotations on views of parent are only copied once to the selector PE (therefore new added annotations to the parent are not reflected in the selection process and display of the selector)

10. Scalability testing of implementation

This section will describe the scalability test for the implementation. For consistency and comparability we will use the same query set as used for our experiments. Except for queries 1 and 2 all queries can be executed on the sensordataweb. The creation of annotations (query 1) is done when adding an annotation (query 3) that doesn't exist yet. The deletion of annotations (query 2) is not supported in the system. Table 21 shows the execution plan for all queries that are possible. Next we will take a look at the data sets and the experiment design.

Query	Execution plan
3. Add_annot	Create annotation if not existing yet and then add to data
4. Remove_annot	Remove the annotation query from the PE-specific table
5. S: D/C: D	Make a selection node with value conditions and show its data
6. S: DAT/C: D	Make a selection node with value conditions, add annotations to it and show its data
7. S: DT/C: A	Make a selection node with annotation conditions and show its data
8. S: DT/C: T	Make a selection node with timed conditions and show its data
9: S: DAT/C: T	Make a selection node with timed conditions, add annotations to it and show its data
10: S: DT/C: D	Make a join node and show its data

Table 21: Overview of queries and their execution plan for the sensordataweb

10.1. Data set and scalability testing design

We have chosen to use 3 data sets with the sizes of 10K, 100K and 1000K entries. The first and second data set were created by running the system with a source node that stopped producing data after 10K or 100K. When this source was set to generate data faster than 1 second duplicates and other problems occurred. For these reasons it was not practically possible to generate the 1000K in the same way. Therefore this set was created manually by inserting data into the table of the source and then insert a selection of data from this table into the other nodes tables. The annotations 'hot', 'hotter' and 'hottest' were added to the source node and the other nodes requiring annotations. The ratios between the total data and resulting data of the queries were made the same as with the experiments.

On startup of the system all required nodes for the queries are created and the nodes that require annotations get them added as well. The queries were run 30 times on each data set and the actual time passed was measured with a precision of milliseconds using Java's System. currentTimeMillis() method. The standard deviations of the measurements were also calculated. The annotation management queries (3 and 4) were measured in microseconds using Java's System.nanoTime()/1000 because of the low execution times of these queries.

For more details and actual queries see 'Appendix D: Implementation testing details'.

10.2. Results

Table 22 and Figure 11 show the results of the experiments with our implementation. The standard deviations for the measurements can be found in 'Appendix F: Measured standard deviations for implementation'. Query 1 and 2 were not run because our implementation doesn't support deleting annotations and the creation of them is done while adding them to the data (query 3). The column "1<>2" shows the difference between the execution times of data sets 1 and 2. The column "2<>3" does the same for data sets 2 and 3.

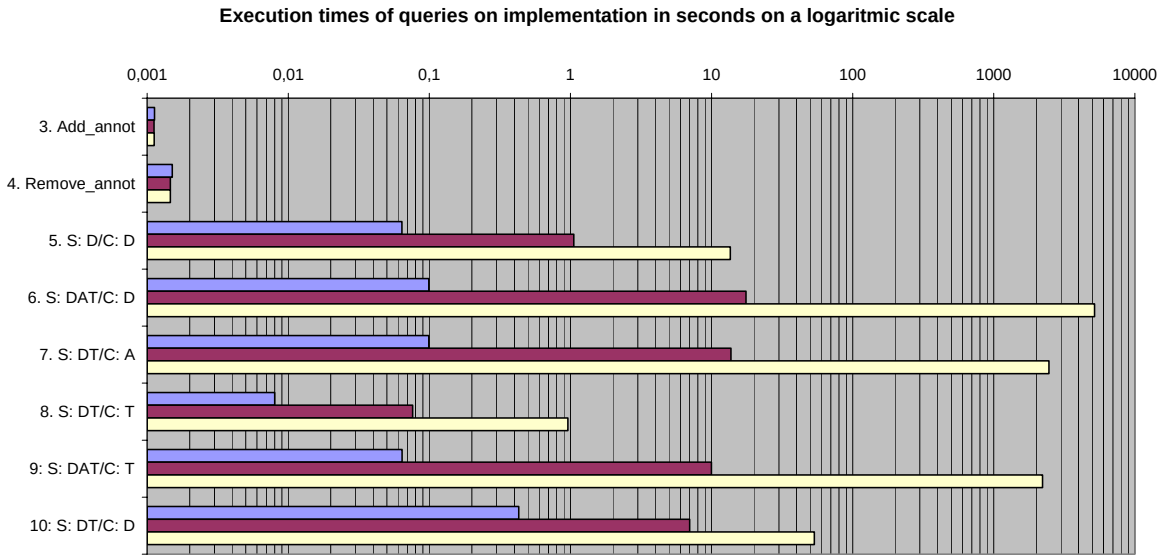
At first we must note that the data sets are bigger than the ones used in the WHB08 experiments and therefore the absolute numbers cannot be compared. Also the data set used for query 10 on size 3 was smaller than intended due to a lack of memory. It turned out this query was quite memory intensive and the maximum achievable was 7 times larger than the previous set. Another thing to note is that the first query executed by the sensordataweb was in general relatively slow. If it turned out to be the only big outlier, we left it out of the average and standard deviation calculations.

The annotation management queries are perfectly scalable just as it was during the experiments and so is query 10. Queries 5 and 8 scaled well, but this wasn't the case in the experiments. Queries 6, 7 and 9 scaled very bad, while at the experiments 6 and 9 had a good scalability. In general we can conclude that everything involving annotations is scaling bad.

Query	Data set 1 [s]	1<>2	Data set 2 [s]	2<>3	Data set 3 [s]
1. Create_annot	*	*	*	*	*
2. Delete_annot	*	*	*	*	*
3. Add_annot	0.001128	*0.99x*	0.001119	*1.00x*	0.001123
4. Remove_annot	0.001506	*0.97x*	0.001461	*1.00x*	0.001461
5. S: D/C: D	0.063933	*16.51x*	1.055633	*12.87x*	13.588267
6. S: DAT/C: D	0.099367	*176.21x*	17.509400	*295.88x*	5180.64553
7. S: DT/C: A	0.099100	*138.34x*	13.709333	179.66x	2462.99343
8. S: DT/C: T	0.008000	*9.53x*	0.076267	*12.57x*	0.958533
9: S: DAT/C: T	0.064233	*155.24x*	9.971467	*222.44x*	2218.05617
10: S: DT/C: D	0.430067	*16.27x*	6.997467	*7.63x* **	53.39975 **

Table 22: Execution times of queries on implementation in seconds
* Not applicable
** Data set was only 7 times bigger than previous

The differences with the other experiments can be explained by the approach that the sensordataweb uses. On insertion of a new data tuple all nodes with as parent the inserting node check if the new data fulfils their constraints. When the user queries for the data the constraints are already applied and all data in the nodes table can simply be returned. Basically every query without annotations boils down to a simple select of all data of a node and for that reason queries without annotations have a good scalability.



	10: S: DT/C: D	9: S: DAT/C: T	8. S: DT/C: T	7. S: DT/C: A	6. S: DAT/C: D	5. S: D/C: D	4. Remove_annot	3. Add_annot
Data set 1	0,430067	0,064233	0,008000	0,099100	0,099367	0,063933	0,001506	0,001128
Data set 2	6,997467	9,971467	0,076267	13,709333	17,509400	1,055633	0,001461	0,001119
Data set 3	53,399750	2218,056170	0,958533	2462,993430	5180,645530	13,588267	0,001461	0,001123

Figure 11: Graphical view of implementation measurements

But when annotations are involved, the system always has to find out which annotations apply to which measurement. Even when constraining on annotations (which is also done during insertion), the annotations that apply to the data need to be found for display. For that reason queries with constraints on annotations and queries selecting annotations are equal when we are not looking at the insertion part. In order to find out why queries with annotations involved score so bad we have to take a look into the code of the implementation and compare it to that of the earlier experiments.

When we compare the code from the experiments to the implementation, we see 2 for/while loops extra in the code of the implementation. First we need to add an annotation column to all data to make sure all data is displayed on retrieval (and not only data that has an annotation) and we can actually display the annotations after retrieval. This is necessary due to the way the systems handles the display of retrieved data. Second for each element found we need to check if it is already in the final list (the list of elements returned and finally displayed). It is possible that an element has 2 annotations on it and when the second one is found we don't want the element to be added a second time with that annotation, but we want to add the annotation to the existing element in our final list. These extra loops in the code could explain why the scalability of the implementation is lower than that of the experiments whenever annotations are involved.

11. Conclusions

This chapter will show the conclusions drawn from this research and the answers to the research questions. It will also discuss future work. We'll start by answering our research questions. Our main question was:

How can we efficiently store sensor data and its annotations in such a way that it can be queried efficiently again with different types of queries?

But in order to answer that we will first have to take a look at the answers to the subquestions.

What kinds of storage and storage models for sensor data and its annotations are available?

We have looked in the literature for systems and methods for storing sensor data and its annotations. This was done in the fields of XML, relational and column databases. Only for relational a number of systems and methods were found, but column had 3 interesting types of databases to consider as well. For 1 of those we developed our own storage model.

One of the conclusions drawn here is that there were no systems or methods found that can annotate on time itself. Annotations were always bound to sensor data, both in storage and retrieval terms.

How can we efficiently retrieve data and annotations?

We did scalability experiments with 3 of the found methods (2 relational, 1 column). The Hypertable method showed promising scalability but also very high absolute numbers and lacked functionality. The BDBMS method didn't scale well. Therefore the most scalable relational method (WHB08) was chosen to create an implementation with.

Which solution has the best scalability?

While the Hypertable solution showed the best scalability, it also had very high absolute numbers and lacked functionality like data types support and joins. Therefore the WHB08 solution was chosen as the winner. After implementing this method into an existing open-source system and testing it again it turned out that every query involving annotations had a bad scalability and that the implementation of WHB08 could not be considered a success. This could be explained by the way the existing system functioned.

How can we efficiently store sensor data and its annotations in such a way that it can be queried efficiently again with different types of queries?

There is not 1 system or method able to accomplish this. The main division between methods were the ones good at selecting annotations and the ones good at putting constraints on annotations. If it was good at the one, it was bad at the other. For this reason it is almost impossible to build a system capable of doing both scalable. A solution could be creating a hybrid storage system, but this creates a lot of storage overhead and other complications.

Next we will take a look at what other conclusions were drawn during this research.

Other conclusions:

As said earlier on, Bigtable (in practice: Hypertable) showed promising scalability, but lacked functionality and the absolute numbers were very high. One of the big limitations was the fact that Hypertable could only store strings and no other data types like integers. When searching for a temperature greater than 25, all rows had to be retrieved and the string with the temperature first had to be converted to an integer for each row before it could be compared. If a million rows are retrieved, this has to be checked for a million rows, slowing down the system significantly. Also the versioning turned out to be ineffective when retrieving both data and annotations and the ability to join tables was not present.

We think Hypertable would be a better alternative to relational systems if it would support other data types. This functionality is planned for an unknown future release. Also a new version came out recently (after 6 months of silence) and it would be worth testing again with the newest version to see if any performance improvements were made. The Bigtable type database is a promising one, but the current open-source alternatives do not suffice yet to replace Google's original.

11.1. Future work

There are a couple of interesting database technologies left out of this research due to time constraints. For example Cassandra [LM09], the database behind Facebook and Twitter, is a fast growing system that promises high read and write speeds compared to relational. It became open-source in 2008 and recently graduated to an Apache top level project. MonetDB [Bon02] [CWI10] is a relational system that is a crossover between row and column based. It shows high performance especially in data mining, OLAP and GIS applications.

Both these systems look very promising and it would be good to run experiments with them. Also the newest version of Hypertable and the other alternative to Bigtable, HBase, could be tested for scalability.

Other interesting future work is the creation of a 'hybrid' method using elements of methods good at selecting annotations and methods good at constraining annotations. It would be interesting to see how much storage overhead this creates, how difficult the complications that will occur can be solved and how such a hybrid method will scale and perform.

Acknowledgements

I would like to thank my supervisor Andreas Wombacher for all his support and feedback during my graduation. Also I like to thank Erik Langius and Bram van der Waaij for giving me the opportunity to be at TNO-ICT during my graduation and for all the ideas, suggestions and feedback they gave me. Finally I like to thank my girlfriend for all her support during my graduation and my parents for all their support during my entire study.

References

[BCT04] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan,Gaurav Vijayvargiya, An Annotation Management System for Relational Databases, *Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004*

[BJR07] Neerja Bhatnagar, Benjoe A. Juliano, Renee S. Renner, Data Annotation Models and Annotation Query Language, *World Academy of Science, Engineering and Technology 33 2007*

[Bog05] Bogdan Alexe, An Alternative Storage Scheme for the DBNotes Annotation Management System for Relational Databases, *University of California, Santa Cruz, 2005*

[Bon02] P. A. Boncz. Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications. *Ph.d. thesis, Universiteit van Amsterdam, May 2002.*

[CCV05] Laura Chiticariu, Wang-Chiew Tan, Gaurav Vijayvargiya, DBNotes: A Post-It System for Relational Databases based on Provenance, *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*

[CWI10] CWI, MonetDB, *http://monetdb.cwi.nl/, 2010*

[EAE09] M.Y.Eltabakh, W.G. Aref, A.K. Elmagarmid, M. Ouzzani, Y.N. Silva, Supporting annotations on relations, *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, 2009*

[EM02] Andrew Eisenberg, Jim Melton, SQL/XML is making good progress, *ACM Special Interest Group on Management of Data: SIGMOD Record, June 2002*

[EM04] Andrew Eisenberg, Jim Melton, Advancements in SQL/XML, *ACM SIGMOD Record, v.33 n.3, September 2004*

[EOA07] M.Y.Eltabakh, M. Ouzzani, W.G., Aref, BDBMS - A Database Management System for Biological Data, *3rd Biennial Conference on Innovative Data Systems Research (CIDR) January 7-10, 2007*

[GKM06] Floris Geerts, Anastasios Kementsietsidis, Diego Milano, MONDRIAN: Annotating and querying databases through colors and blocks, *22nd International Conference on Data Engineering (ICDE'06), 2006*

[Goo06] Google, Inc., Bigtable: A distributed storage system for structured data, *OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006.*

[Goo08] Google, Inc., Data Management Projects at Google, *SIGMOD Record, March 2008 (Vol. 37, No. 1)*

[IJk09] Stichting IJkdijk, *http://www.ijkdijk.nl/, 2009*

[JS82] G. Jaeschke, H. J. Schek, Remarks on the algebra of non first normal form relations, *Symposium on Principles of Database Systems, 1982*

[KG08] Ankur Khetrapal, Vinay Ganesh, HBase and Hypertable for large scale distributed storage systems, *Dept. of Computer Science, Purdue University*

[KMP09] J. Kuperus, A.H. Mols, J.S. Posthuma, T.J. Visser, MAAS Massive Annotation & Aggregation System, *University of Twente, 2009*

[Lan09] Reinier de Lange, Provenance Aware Sensor Networks for Real-time Data Analysis, *University of Twente, 2009*

[LLC08] Qinglan Li, Alexandros Labrinidis, and Panos K. Chrysanthis, User-Centric Annotation Management for Biological Data, *the Second International Provenance and Annotation Workshop, June 2008*

[LM09] Avinash Lakshman, Prashant Malik, Cassandra - A Decentralized Structured Storage System, *2009*

[LM09-2] Avinash Lakshman, Prashant Malik, Cassandra, Structured Storage System over a P2P Network, *http://www.slideshare.net/Eweaver/cassandra-presentation-at-nosql, 2009*

[STH99] Jayavel Shanmugasundaram, Kristin Chun Zhang, David DeWitt, and Jeffrey Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proceedings of the Very Large Databases Conference (VLDB), 1999*

[W3C01] W3C, XQuery 1.0: An XML Query Language, *W3C Working Draft 07 June 2001*

[WHB08] Matthias Walgers, Laurens Hendriksen, Faiza Allah Buhksh, Annotation of Streaming Data, *University of Twente, 2008*

[WHB08-2] Matthias Walgers, Laurens Hendriksen, Faiza Allah Buhksh, Comparison of storage models for sensor data and annotation data, *University of Twente, 2008*

Appendix A: Data manipulation details

This appendix will describe the details of the data manipulation, including the used queries.

Create statement of subset table:

```
CREATE TABLE `genericijkdijksubset` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `timed` bigint(20) NOT NULL,
  `ANYSENSETIMESTAMP` bigint(20) DEFAULT NULL,
  `OWNERID` varchar(20) DEFAULT NULL,
  `OWNERTIMESTAMP` bigint(20) DEFAULT NULL,
  `SENSORID` varchar(20) DEFAULT NULL,
  `PARTID` varchar(20) DEFAULT NULL,
  `PARTDOUBLE` double DEFAULT NULL,
  `PARTLONG` bigint(20) DEFAULT NULL,
  `PARTSTRING` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`PK`),
  UNIQUE KEY `genericijkdijk_INDEX` (`timed`),
  KEY `ownertimestamp` (`OWNERTIMESTAMP`),
  KEY `sensorid` (`SENSORID`),
  KEY `partid` (`PARTID`),
  KEY `PARTDOUBLE` (`PARTDOUBLE`)
);
```

Query to insert desired data into subset table:

```
INSERT INTO genericijkdijksubset(pk, timed, anysensetimestamp, ownerid, ownertimestamp,
sensorid, partid, partdouble, partlong, partstring) SELECT * FROM genericijkdijk WHERE sensorid =
7 OR sensorid = 136;
```

Query to find erroneous data of the weather station:

```
SELECT * FROM genericijkdijksubset WHERE sensorid = 7 GROUP BY ownertimestamp HAVING
count(pk) = 13; (or < or >)
```

Query to delete erroneous data of the weather station:

```
DELETE FROM genericijkdijksubset WHERE partdouble = 123 AND sensorid = 7 AND partid = 25;
```

Query to detect erroneous data of the luisterbuis:

```
SELECT * FROM genericijkdijksubset WHERE sensorid = 136 GROUP BY ownertimestamp
HAVING count(pk) > 4;
```

Create statement of weather station table:

```
CREATE TABLE `weatherstation` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `timed` bigint(20) NOT NULL,
  `ANYSENSETIMESTAMP` bigint(20) DEFAULT NULL,
  `OWNERID` varchar(20) DEFAULT NULL,
  `OWNERTIMESTAMP` bigint(20) DEFAULT NULL,
  `SENSORID` varchar(20) DEFAULT NULL,
  `RADAVG` double DEFAULT NULL,
  `RADMAX` double DEFAULT NULL,
  `RADMIN` double DEFAULT NULL,
  `WINDSPEEDAVG` double DEFAULT NULL,
  `WINDDIR` double DEFAULT NULL,
  `WINDGUSTMAX` double DEFAULT NULL,
  `TEMPAVG` double DEFAULT NULL,
  `TEMPMAX` double DEFAULT NULL,
  `TEMPMIN` double DEFAULT NULL,
  `HUMAVG` double DEFAULT NULL,
  `HUMMAX` double DEFAULT NULL,
  `HUMMIN` double DEFAULT NULL,
  `RAINFALLTOTAL` double DEFAULT NULL,
  PRIMARY KEY (`PK`),
  UNIQUE KEY `weatherstation_INDEX` (`timed`),
  KEY `ownertimestamp` (`OWNERTIMESTAMP`),
  KEY `sensorid` (`SENSORID`)
);
```

Create statement of luisterbuis table:

```
CREATE TABLE `luisterbuis` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `timed` bigint(20) NOT NULL,
  `ANYSENSETIMESTAMP` bigint(20) DEFAULT NULL,
  `OWNERID` varchar(20) DEFAULT NULL,
  `OWNERTIMESTAMP` bigint(20) DEFAULT NULL,
  `SENSORID` varchar(20) DEFAULT NULL,
  `PRESSUREA` double DEFAULT NULL,
  `PRESSUREB` double DEFAULT NULL,
  `TEMPA` double DEFAULT NULL,
  `TEMPB` double DEFAULT NULL,
  PRIMARY KEY (`PK`),
  UNIQUE KEY `luisterbuis_INDEX` (`timed`),
  KEY `ownertimestamp` (`OWNERTIMESTAMP`),
  KEY `sensorid` (`SENSORID`)
);
```

The ‘luisterbuissubset’ table uses the same schema as the ‘luisterbuis’ table.

Insert data from luisterbuis table into luisterbuissubset table:

```
INSERT INTO luisterbuissubset(timed, anysensetimestamp, ownerid, ownertimestamp, sensorid,
pressurea, pressureb, tempa, tempb) SELECT timed, anysensetimestamp, ownerid, ownertimestamp,
sensorid, pressurea, pressureb, tempa, tempb FROM luisterbuis WHERE MOD(pk,10)=1;
```

Appendix B: Annotation adding details

This appendix describes queries used for adding the annotation tables and data.

Create statement of weather station annotation table [WHB08]:

```
CREATE TABLE `weatherstationannot` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `QUERY` text NOT NULL,
  `TIME` bigint(20) NOT NULL,
  `SENSORID` tinytext NOT NULL,
  `ANNOTATIONID` int NOT NULL,
  PRIMARY KEY (`PK`)
);
```

Create statement of luisterbuis annotation table [WHB08]:

```
CREATE TABLE `luisterbuisannot` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `QUERY` text NOT NULL,
  `TIME` bigint(20) NOT NULL,
  `SENSORID` tinytext NOT NULL,
  `ANNOTATIONID` int NOT NULL,
  PRIMARY KEY (`PK`)
);
```

Create statement of annotation table:

```
CREATE TABLE `annotations` (
  `ANNOTATIONID` bigint(20) NOT NULL AUTO_INCREMENT,
  `TIME` bigint(20) NOT NULL,
  `VALUE` text NOT NULL,
  PRIMARY KEY (`ANNOTATIONID`)
);
```

Create statement of comments table:

```
CREATE TABLE `comments` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `TIME` bigint(20) NOT NULL,
  `ANNOTATIONID` int NOT NULL,
  `COMMENT` text NOT NULL,
  PRIMARY KEY (`PK`)
);
```

Create statement of weather station annotation table BDBMS:

```
CREATE TABLE `weatherstationannot2` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `ANNOTATIONID` int NOT NULL,
  `TIME` bigint(20) NOT NULL,
  `VALUE` text NOT NULL,
  `RECTANGLE` tinytext NOT NULL,
  PRIMARY KEY (`PK`),
  KEY `annotationid` (`ANNOTATIONID`)
);
```

Create statement of luisterbuissubset annotation table BDBMS:

```
CREATE TABLE `luisterbuissubsetannot2` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `ANNOTATIONID` int NOT NULL,
  `TIME` bigint(20) NOT NULL,
  `VALUE` text NOT NULL,
  `RECTANGLE` tinytext NOT NULL,
  PRIMARY KEY (`PK`),
  KEY `annotationid` (`ANNOTATIONID`)
);
```

Create statement of luisterbuis annotation table BDBMS:

```
CREATE TABLE `luisterbuisannot2` (
  `PK` bigint(20) NOT NULL AUTO_INCREMENT,
  `ANNOTATIONID` int NOT NULL,
  `TIME` bigint(20) NOT NULL,
  `VALUE` text NOT NULL,
  `RECTANGLE` tinytext NOT NULL,
  PRIMARY KEY (`PK`),
  KEY `annotationid` (`ANNOTATIONID`)
);
```

Create statement of luisterbuis and weatherstation tables for Hypertable:

```
CREATE TABLE weatherstationd (
  sensor,
  timeline,
  ACCESS GROUP default (sensor, timeline)
);

CREATE TABLE luisterbuissubsetd (
  sensor,
  timeline,
  ACCESS GROUP default (sensor, timeline)
);

CREATE TABLE luisterbuisd (
  sensor,
  timeline,
  ACCESS GROUP default (sensor, timeline)
);
```

Inserting annotations and add them to data for [WHB08]:

```
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'hot');
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'warm');
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'warm rain');
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'nice weather');
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'storm');
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'cold');
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'high pressure');
INSERT INTO annotations (time, value) values(unix_timestamp(now()), 'low pressure');

INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('tempavg > 25',
unix_timestamp(now()), 'tempavg', '1');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('tempavg > 20',
unix_timestamp(now()), 'tempavg', '2');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('tempavg > 15 and
rainfalltotal > 0', unix_timestamp(now()), 'tempavg', '3');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('tempavg > 15 and
rainfalltotal > 0', unix_timestamp(now()), 'rainfalltotal', '3');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('tempavg > 15 and
rainfalltotal = 0 and windspeedavg < 4', unix_timestamp(now()), 'tempavg', '4');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('tempavg > 15 and
rainfalltotal = 0 and windspeedavg < 4', unix_timestamp(now()), 'rainfalltotal', '4');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('tempavg > 15 and
rainfalltotal = 0 and windspeedavg < 4', unix_timestamp(now()), 'windspeedavg', '4');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('rainfalltotal > 0.3 and
windspeedavg > 6', unix_timestamp(now()), 'rainfalltotal', '5');
INSERT INTO weatherstationannot(query, time, sensorid, annotationid) values('rainfalltotal > 0.3 and
windspeedavg > 6', unix_timestamp(now()), 'windspeedavg', '5');
INSERT INTO luisterbuisannot(query, time, sensorid, annotationid) values('tempa < 5 and tempb < 5',
unix_timestamp(now()), 'tempa', '6');
INSERT INTO luisterbuisannot(query, time, sensorid, annotationid) values('tempa < 5 and tempb < 5',
unix_timestamp(now()), 'tempb', '6');
INSERT INTO luisterbuisannot(query, time, sensorid, annotationid) values('pressurea > 238000 and
pressureb > 149000', unix_timestamp(now()), 'pressurea', '7');
INSERT INTO luisterbuisannot(query, time, sensorid, annotationid) values('pressurea > 238000 and
pressureb > 149000', unix_timestamp(now()), 'pressureb', '7');
INSERT INTO luisterbuisannot(query, time, sensorid, annotationid) values('pressurea < 237000 and
pressureb < 148500', unix_timestamp(now()), 'pressurea', '8');
INSERT INTO luisterbuisannot(query, time, sensorid, annotationid) values('pressurea < 237000 and
pressureb < 148500', unix_timestamp(now()), 'pressureb', '8');
```

Inserting annotations and add them to data for BDBMS:

The annotations for BDBMS were added automatically using a function. This function has as input a query that selects data, an annotation value, an array of column names and the table to put the annotations on. The query is executed and the results plus the column names are transformed into rectangles. These are then inserted into the annotation table of the table given. This process is done for all annotations on the weatherstation and luisterbuis data.

Inserting annotations and add them to data for Hypertable:

First all rows belonging to the respective sensor are selected. Next we iterate over this data and if it meets the annotation condition the annotation name is inserted into the column ‘timeline:annotation’.

Appendix C: Queries for larger data sets using luisterbuis data

The annotation management queries can stay the same since they don't use the actual data but the annotation tables. The only exception on this is query 3 for BDBMS and Hypertable. For those and the other queries these are the queries for the larger data sets:

3:

BDBMS:

SELECT pk FROM luisterbuissubset WHERE tempa < 15;

SELECT pk FROM luisterbuis WHERE tempa < 15;

Hypertable:

SELECT * FROM luisterbuissubsetd WHERE cell='luisterbuis.tempa','timeline:base';

SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:base';

5:

WHB08 and BDBMS:

SELECT tempa FROM luisterbuissubset WHERE tempa > 15;

SELECT tempa FROM luisterbuis WHERE tempa > 15;

Hypertable:

SELECT * FROM luisterbuissubsetd WHERE cell='luisterbuis.tempa','timeline:base';

SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:base';

6:

WHB08:

SELECT query, annotationid FROM luisterbuisannot WHERE STRCMP('tempa', sensorid) = 0;
SELECT ownertimestamp, tempa FROM luisterbuissubset WHERE tempa > 15.3 AND tempb > 15.3;

SELECT query, annotationid FROM luisterbuisannot WHERE STRCMP('tempa', sensorid) = 0;
SELECT ownertimestamp, tempa FROM luisterbuis WHERE tempa > 15.3 AND tempb > 15.3;

BDBMS:

SELECT pk, ownertimestamp, tempa FROM luisterbuissubset WHERE tempa > 15;

SELECT pk, ownertimestamp, tempa FROM luisterbuis WHERE tempa > 15;

Hypertable:

SELECT * FROM luisterbuissubsetd WHERE cell='luisterbuis.tempa','timeline:base';

SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:base';

7:

WHB08:

SELECT a.value, la.query FROM luisterbuisannot as la, annotations as a WHERE STRCMP(la.query, 'tempa < 5 and tempb < 5') = 0 AND la.annotationid = a.annotationid;
SELECT ownertimestamp, tempa, tempb FROM luisterbuissubset as l, weatherstationannot as la WHERE l.{la.query};

First query is the same.
SELECT ownertimestamp, tempa, tempb FROM luisterbuis as l, weatherstationannot as la WHERE l.{la.query};

BDBMS:

SELECT * FROM luisterbuisannot2 WHERE STRCMP(value, 'cold') = 0;
SELECT ownertimestamp, tempa, tempb FROM luisterbuissubset WHERE pk IN ({$list});

SELECT * FROM luisterbuisannot2 WHERE STRCMP(value, 'cold') = 0;
SELECT ownertimestamp, tempa, tempb FROM luisterbuis WHERE pk IN ({$list});

Hypertable:

SELECT * FROM luisterbuissubsetd WHERE cell='luisterbuis.tempa','timeline:annotation' and cell='luisterbuis.tempb','timeline:annotation';

SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:annotation' and cell='luisterbuis.tempb','timeline:annotation';

8:

WHB08 and BDBMS:

SELECT ownertimestamp, tempa FROM luisterbuissubset WHERE ownertimestamp BETWEEN '1253897800000' AND '1253900970000';

SELECT ownertimestamp, tempa FROM luisterbuis WHERE ownertimestamp BETWEEN '1253897800000' AND '1253900970000';

Hypertable:

SELECT * FROM luisterbuissubsetd WHERE cell='luisterbuis.tempa','timeline:base' and '2009-09-25 18:56:40' < TIMESTAMP < '2009-09-25 19:49:30';

SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:base' and '2009-09-25 18:56:40' < TIMESTAMP < '2009-09-25 19:49:30';

9:

WHB08:

For the first query ‘weatherstation’ is replaced by ‘luisterbuissubset’ and ‘luisterbuis’. The second query is as following:

SELECT pk, ownertimestamp FROM luisterbuissubset WHERE {query} AND ownertimestamp BETWEEN '1253898800000' AND '1253900970000';

SELECT pk, ownertimestamp FROM luisterbuis WHERE {query} AND ownertimestamp BETWEEN '1253898800000' AND '1253900970000';

BDBMS:

SELECT pk, ownertimestamp FROM luisterbuissubset WHERE ownertimestamp BETWEEN '1253898800000' AND '1253900970000';

SELECT pk, ownertimestamp FROM luisterbuis WHERE ownertimestamp BETWEEN '1253898800000' AND '1253900970000';

Hypertable:

SELECT * FROM luisterbuissubsetd WHERE cell='luisterbuis.tempa','timeline:base' AND '2009-09-26 00:00:00' < TIMESTAMP < '2009-09-26 10:00:00';

SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:base' AND '2009-09-26 00:00:00' < TIMESTAMP < '2009-09-26 10:00:00';

10:

WHB08 and BDBMS:

SELECT w.ownertimestamp, tempavg, l.ownertimestamp, tempa FROM weatherstation as w, luisterbuissubset as l WHERE w.tempavg = l.tempa;

SELECT w.ownertimestamp, tempavg, l.ownertimestamp, tempa FROM weatherstation as w, luisterbuis as l WHERE w.tempavg = l.tempa;

Hypertable:

SELECT * FROM luisterbuissubsetd WHERE cell='luisterbuis.tempa','timeline:base';

SELECT * FROM luisterbuisd WHERE cell='luisterbuis.tempa','timeline:base';

Appendix D: Implementation testing details

Queries used to create the 1000K data set for implementation testing and adding the annotations to the data.

Creation of the 1000K data set:

First, using PHP, a million rows with random values are inserted to source1 and source2. Next the following queries copied the desired amount of rows to the queryx_view tables:

```
INSERT INTO query5_view(channel, value, timed, timed2, fixed_int1, fixed_int2) SELECT channel, value, timed, timed2, fixed_int1, fixed_int2 FROM source1_view WHERE value >= 0.25 and value <= 1.0;
INSERT INTO query6_view(channel, value, timed, timed2, fixed_int1, fixed_int2) SELECT channel, value, timed, timed2, fixed_int1, fixed_int2 FROM source1_view WHERE value >= 0.87 and value <= 1.0;
INSERT INTO query7_view(channel, value, timed, timed2, fixed_int1, fixed_int2) SELECT channel, value, timed, timed2, fixed_int1, fixed_int2 FROM source1_view WHERE value >= 0.9 and value <= 0.99;
INSERT INTO query8_view(channel, value, timed, timed2, fixed_int1, fixed_int2) SELECT channel, value, timed, timed2, fixed_int1, fixed_int2 FROM source1_view ORDER BY RAND() LIMIT 80000;
INSERT INTO query9_view(channel, value, timed, timed2, fixed_int1, fixed_int2) SELECT channel, value, timed, timed2, fixed_int1, fixed_int2 FROM source1_view ORDER BY RAND() LIMIT 100000;
INSERT INTO query10_view(channel, value, timed, timed2, fixed_int1, fixed_int2) SELECT s.channel, s.value, s.timed, s.timed2, s.fixed_int1, s.fixed_int2 FROM source1_view as s, source2_view as t WHERE s.value = t.value;
```

Queries for inserting the annotations:

```
INSERT INTO annotations(value) VALUES('hot');
INSERT INTO annotations(value) VALUES('hotter');
INSERT INTO annotations(value) VALUES('hottest');

INSERT INTO source1_view_annot(query, sensorid, annotationid) VALUES('value >= 0.55 and value <= 0.8', 'value', '1');
INSERT INTO source1_view_annot(query, sensorid, annotationid) VALUES('value > 0.75 and value <= 0.9', 'value', '2');
INSERT INTO source1_view_annot(query, sensorid, annotationid) VALUES('value >= 0.9 and value <= 0.99', 'value', '3');
```

Appendix E: Measured standard deviations for experiments

The standard deviations (in seconds) as belonging to the measurements (in seconds) for all three systems.

Query	Data set 1 [s]		Data set 2 [s]		Data set 3 [s]	
	Measure.	Std. dev.	Measure.	Std. dev.	Measure.	Std. dev.
1. Create_annot	0.000075	0.000210	0.000094	0.000024	0.000114	0.000060
2. Delete_annot	0.000085	0.000038	0.000104	0.000007	0.000106	0.000004
3. Add_annot	0.000165	0.000357	0.000219	0.000053	0.000223	0.000035
4. Remove_annot	0.000160	0.000633	0.000178	0.000009	0.000183	0.000014
5. S: D/C: D	0.000348	0.000021	0.005006	0.000083	0.862047	0.001284
6. S: DAT/C: D	0.018436	0.000064	0.051219	0.000217	0.500611	0.001768
7. S: DT/C: A	0.000112	0.000156	0.000149	0.003270	0.031448	0.001935
8. S: DT/C: T	0.000072	0.000058	0.000755	0.004288	0.207547	0.000667
9: S: DAT/C: T	0.000479	0.000084	0.002064	0.000109	0.018178	0.000206
10: S: DT/C: D	*	*	20.335640	0.001142	191.751825	0.225297

Table 23: Standard deviations for WHB08 measurements
* Not applicable

Query	Data set 1 [s]		Data set 2 [s]		Data set 3 [s]	
	Measure.	Std. dev.	Measure.	Std. dev.	Measure.	Std. dev.
1. Create_annot	*	*	*	*	*	*
2. Delete_annot	*	*	*	*	*	*
3. Add_annot	0.004724	0.000796	0.183509	0.021117	2.140202	0.209431
4. Remove_annot	0.000881	0.000023	0.034033	0.000681	0.349678	0.060483
5. S: D/C: D	0.000356	0.000334	0.003765	0.008087	0.850495	0.005030
6. S: DAT/C: D	0.056388	0.000969	55.439357	0.251761	5632.72	63.229114
7. S: DT/C: A	0.000242	0.000022	0.076339	0.012888	2.001602	0.008585
8. S: DT/C: T	0.000078	0.000069	0.000550	0.002088	0.206108	0.000500
9: S: DAT/C: T	0.015473	0.000266	11.428085	0.087008	1170.066720	14.436027
10: S: DT/C: D	*	*	20.504747	0.004334	191.569171	0.027870

Table 24: Standard deviations for BDBMS measurements
* Not applicable

Query	Data set 1 [s]		Data set 2 [s]		Data set 3 [s]	
	Measure.	Std. dev.	Measure.	Std. dev.	Measure.	Std. dev.
1. Create_annot	*	*	*	*	*	*
2. Delete_annot	*	*	*	*	*	*
3. Add_annot	0.982558	0.216796	142.21902	2.067266	1251.10911	19.698642
4. Remove_annot	0.003182	0.000846	0.194278	0.105577	0.878988	0.441667
5. S: D/C: D	0.017852	0.000241	0.197913	0.001684	0.862308	0.004269
6. S: DAT/C: D	0.618334	0.003614	411.46775	0.430449	4911.16806	4.059857
7. S: DT/C: A	0.148391	0.000385	2.458545	0.005623	42.474110	0.057565
8. S: DT/C: T	0.007483	0.000064	0.108575	0.000520	0.522888	0.001457
9: S: DAT/C: T	0.038526	0.000198	11.943820	0.041720	81.875499	0.113853
10: S: DT/C: D	*	*	128.427053	0.650961	580.456005	3.023181

Table 25: Standard deviations for Hypertable measurements
* Not applicable

Appendix F: Measured standard deviations for implementation

The standard deviations (in seconds) as belonging to the measurements (in seconds) for the implementation.

Query	Data set 1 [s]		Data set 2 [s]		Data set 3 [s]	
	Measure.	Std. dev.	Measure.	Std. dev.	Measure.	Std. dev.
1. Create_annot	*	*	*	*	*	*
2. Delete_annot	*	*	*	*	*	*
3. Add_annot	0.001128	0.000049	0.001119	0.000167	0.001123	0.000067
4. Remove_annot	0.001506	0.000066	0.001461	0.000080	0.001461	0.000066
5. S: D/C: D	0.063933	0.017414	1.055633	0.173428	13.588267	1.615791
6. S: DAT/C: D	0.099367	0.012656	17.509400	0.706195	5180.64553	130.73351
7. S: DT/C: A	0.099100	0.009101	13.709333	0.492059	2462.99343	59.218374
8. S: DT/C: T	0.008000	0.000931	0.076267	0.011673	0.958533	0.128742
9: S: DAT/C: T	0.064233	0.002109	9.971467	0.235054	2218.05617	51.187941
10: S: DT/C: D	0.430067	0.057902	6.997467	0.822043	53.39975**	0.452108

Table 26: Standard deviations for implementation measurements

* Not applicable

** Data set was only 7 times bigger than previous

Appendix G: Software used

This appendix will give an overview of the software used in this project.

Silo1 server (relational experiments):

openSUSE 10.3 (X86-64)
PHP 5.2.4
MySQL Community Edition 5.1.23

Silo2 server (deployment implementation):

openSUSE 10.3 (X86-64)
Tomcat 5.5

Patriot server (Hypertable experiments):

Ubuntu 8.04.2
C++ gcc 4.2.4
Hypertable 0.9.2.4

Sensorlab PC (data manipulation Hypertable + compilation & packaging implementation):

Windows XP SP3
Apache 2.2.11
MySQL 5.1.33 (Community Server)
PHP 5.2.9
Java 6u18
Apache Maven 2.2.1