Data Models and Query Languages for Data Streams

Master's Thesis

Jes Søndergaard

Department of Computer Science
Aalborg University
Denmark

June, 2005

Faculty of Science and Engineering
University of Aalborg

Department of Computer Science

Title:

Data Models and Query Languages for Data Streams

Semester:
Dat 6

Project period:
1 February 2005
– 15 June 2005

Author:
Jes Søndergaard

Supervisor:
Janne Skyt

Number of copies: 5

Total number of pages: 54

Abstract:

This report is examining theoretical issues of data streams. Data streams are a way of describing data. In contrast to relational database systems, stream data are infinite and arriving continuously. Some researchers have proclaimed streams as a new paradigm of data even though streams by other may be seen as modified relations. By examining the data models and query languages of existing stream systems, it is evaluated how streams are different from relations. It is concluded that certain properties of streams make them special. However, the differences of streams and relations are not greater than, that streams may be implemented on existing database systems.

Preface

This report is the product of the second and final part of my master's thesis. The first part [5] was written in collaboration with three other students. The first project was also about data streams, but with a broader focus including the application of streams in a specific context. This project shall not be considered as an extension of the previous project but rather as a project inspired by some of the topics from the first project. In particular it is inspired by the stream query language that was designed for a running prototype. Even though the language was functional there was some theoretical issues that was only superficially considered. This project has been motivated by the desire to investigate some of these issue more in dept.

Before reading the entire report it is possible to get a brief overview of the content of the report by reading the summary in Chapter 6.

Contents

Chapter 1

Introduction

This chapter introduces the problem of this project and how the report is organized. The project topics are data models and query languages for data streams. By examining these topics it will be evaluated how data streams differentiate from traditional ways of storing data. Section 1.1 describes different data paradigms and how data streams match these paradigms. Section 1.2 gives an overview of the research in data streams and how it has developed over time. Section 1.3 describes what the goals of the project are and what methods that are used.

1.1 Data Streams

During the last decades a few major paradigms have dominated the area of data storage. The relational data model was first introduced in 1970 [8]. Later this model has gained so much popularity that it has become the dominant model for market-leading DBMSs like, e.g., Oracle, IBM DB2 and Microsoft SQL Server. Originated in the success of the object-oriented programming languages, the object-oriented and object-relational data models have likewise become popular models that are available in many DBMSs. Some of the strengths of the relational and object databases are the abilities to store and update data with complex internal relations and to execute complex queries on these data.

Within the last few year new data extensive applications have appeared that do not fit well with the existing database systems. These applications

are for example monitoring and logging services and are characterized by the fact that they produce a (partly or completely) continuous stream of data. Traditional DBMSs are not to prefer for such applications because of two reasons that regards storage and querying. Because the data rate of a stream may be high a DBMS is typical not able to store data in the same speed with which data arrives. Users of monitoring and logging services require fast response when certain conditions are observed, and this is difficult to obtain with the one-time queries of DBMSs. When running queries on streams, the queries must therefore be long-running and continuously produce results. Because query updates are returned continuously it is not necessary that the entire streams are persistently stored in a database.

Database systems that are designed for managing streams are called *datastream management systems* (DSMSs). A typical DSMS consist of a storage, a query processor, and a repository for holding long-running queries. The input to a DSMS is streams of elements, and the output of the long-running queries also streams.

1.2 Research of Stream Systems

By the paper of 1992 describing the stream system *Tapestry* [12], it is argued that one-time queries in relational DBMSs are insufficient when using continuous data streams. For the first time continuous queries are proposed as a technique for executing queries on streams. Since the introduction of continuous queries in Tapestry, most stream systems have adopted that type of queries. Tapestry, however, was still build on traditional DBMSs and Tapestry queries were translated into SQL queries before they were executed on a database. A present stream system like Telegraph [7] is also implemented on a (modified) DBMS, PostgreSQL.

Later on stream systems like STREAM [1] and Aurora [6] have not only adopted the continuous query model but also designed specific models that moves away from the relational models in favour of stream specific models. The designers of these systems believe that the relational data model, which most major DBMSs build on, are unsuitable for handling streams. However, there is still much disagreements on what exactly a data stream is and how the data model is to be designed. As an example some systems are designed with streams as the only data type while other allow to combine both streams

and updateable relations.

While Tapestry was designed to use the build-in SQL language of DSMSs, stream systems have later introduced new query languages. The languages are motivated by the need of new or extended operators because of the fact that streams are different from relations and because the are other demands to continuous queries than to one-time queries. For example because a stream may be infinite it is often desirable to query in only a subset of the stream. Like the data models there is also no agreement on an ideal query language to be used. Query languages of systems like Tribeca [11] and Aurora [6] are total different from the ones known from traditional databases, while a language like CQL from the STREAM system [1] is an extended SQL language.

While stream systems like Telegraph and Tapestry build upon existing DBMSs some stream systems reject DBMSs and builds new DSMSs from scratch. The objectives of DSMS are efficient processing of data and fast responses to long-running queries rather than persistent storage and handling of complex queries.

To summarize, the differences between traditional database systems and stream database systems may be considered at mainly three levels: data model, query language and management system. From a database point of view the major databases rely on the relational model while streams may demand more stream specific models. Likewise the design of query languages requires special needs for querying in streams. From a software point of view traditional data are often stored in DBMSs while DSMSs are developed specific for handling streams.

1.3 Project Goals

As described in the previous section some believe that the nature of data streams makes existing DBMSs inappropriate for handling streams with respect to storage and querying. Nevertheless the stream research has a great subset in the traditional database community and commercial database software systems. From Section 1.2 it also appears, however, that the distinction between these "two world" are indeterminated and varies from system to system. Many research papers on streams give statements that claim, that data streams are completely in opposite to the traditional databases and that

these two world are incompatible. As an example the following is written in a paper about Aurora [6]: "*For these reasons, monitoring applications are difficult to implement using traditional DBMS technology. To do better, all the basic mechanisms in current DBMSs must be rethought.*"

This project will examine the distinction between traditional databases and stream databases further and seek to answer whether the distinction is legitimate or if streams just are special cases of traditional relational data that can be handled by existing DBMSs.

The way it will be answered how streams are different from relations is first to examine the data models used for storing streams to see if and how they vary from the relational model. Important properties that existing systems emphasize as important will be examined further. Next the query languages that are used for extracting data stored using the models just described will be examined. The most important property of stream queries is the fact that they are continuous but many languages do also contain extra or extended operators compared to relational operators. It will be evaluated what these extra functionalities are and how they are motivated.

The basis of the work just described are, beyond from database literature, a few selected stream systems described in Chapter 2. For further readings about streams in general the survey paper by Golab and Özsu [9] covers many issues of data stream management, including data models and query languages of data streams. Another survey paper by Babcock et al. [4] is written by researchers of the comprehensive STREAM project. The paper covers likewise data stream management with special effort on data models and query languages based on own experiences.

Chapter 2

Related Stream Systems and Query Languages

The basis of the analysis of data streams in this report is a few stream systems briefly described in this chapter. All systems are developed as academic projects with running prototypes.

2.1 Tapestry and TQL

Tapestry [12] is a system for managing streams of electronic documents, and is the system that introduced continuous queries. It runs upon any append-only RDBMS with SQL as query language. Even though the system is several years old, the system is described because some fundamental issues of Tapestry are still valid and referred in recent papers about continuous queries.

Queries in Tapestry are written in a language called TQL (Tapestry Query Language). In [12] it is considered how continuous queries could be executed simply by running an ordinary query multiple times. The major drawback of this approach is, that the query results are non-deterministic: if the same query is issued at two different clock-times, two different result sets are returned. As a consequence, the query semantics must be time independent, and therefore the query must be executed at every time instant, where a time instant is the smallest unit of time in the system.

It may seem that executing a query at every time instant is very inefficient

and maybe practical impossible. Tapestry solves this problem by converting non-monotonic queries to monotonic ones. Thereby only newly arrived data needs to be evaluated. Before a monotonic query is executed it is optimized and finally translated to SQL.

2.2 Aurora

Aurora [2, 6] is a research project and protype system with people from Brandeis University, Brown University, and MIT. The system is focusing on the group of monitoring applications by using streams.

Aurora is a stream-only system supporting continuous queries (both predefined and ad-hoc) and views. Queries in Aurora are specified very untraditional as workflow diagrams in a graphical user interface by using boxes and arcs. A box is one of the 8 primitive operators which accepts one or more input streams and returns a single output stream. The arcs specifies the dataflow between the boxes. The operators are very distinct compared to the widely-used relational algebra language. The operators includes windowing techniques.

2.3 Tribeca

Tribeca [11] is a system for analyzing network traffic. It operates only on streams, which can be live network feeds, disks, and tapes. A query specifies the input stream, a number of intermediate operators, and one or more result streams. Like relational algebra, the data flow between each operator is explicitly expressed in the query. The result of one operator can be input to more than one operator. This means that different queries can be combined into a single query. Remark the limitation in only a single stream as input to a query.

There is three simple operators in Tribeca which are very alike to certain relational algebra operators: qualification, projection, and aggregate. The qualification operator applies a filter to a stream. For each stream element a condition needs to evaluate true if the element will pass the filter. The projection operator preserves only the specified attributes from each stream element. The aggregate operator calculates a single value from the entire stream.

The following example reads weather statistics from `file1` and writes the number of days with temperature over 30 degrees to `file2`:

```
source_stream s1 is {file file1 Weather}
result_stream r1 is {file file2}
stream_proj {{s1.Temperature}} p1
stream_qual {{p1.Temperature.gte 30}} p2
stream_agg {p2.Temperature.count} r1
```

The demultiplexing and multiplexing operators are used to partition a stream into multiple substreams, and after some intermediate operator(s), recombine them into one stream. If the intermediate operator is the aggregate operator, the same behavior as the group by operator from SQL is obtained. However, the intermediate operator can be of any type, which makes demultiplexing more powerful than SQL. The multiplexing operator can also be used to combine two or more separate streams of the same data type.

2.4 TelegraphCQ

TelegraphCQ [10, 7] is a system for processing continuous queries over data streams. It is developed as part of the Telegraph research project at UC Berkeley.

The TelegraphCQ system supports both unbounded data streams and tables. The syntax is based on SQL with an extension for specifying windows. In [7] a low-level language for continuous queries is given with particular focus on windowing mechanisms. It is argued that traditional windows, such as landmark and sliding windows, are insufficient for expressing many types of queries. This includes e.g. additional filters than just the two endpoints and reversed traversal.

If a continuous query contains a reference to one or more stream windows, a conditional loop is added to the query. Every loop traverse corresponds to a relatively increase or decrease in time and for each window the two endpoints are updated.

As an example, the following continuous query returns for every week between day 100 and day 200, the average temperature of the preceding

week (the second and third parameter of the `WindowIs` function is the left end and right end marks of the stream):

```
SELECT AVG(Temperature)
FROM Weather
for (t = 100; t <= 200; t++) {
    WindowIs(Weather, t - 7, t);
}
```

Every time the loop is traversed the window (or windows) is updated with the new end point(s) and the query is executed on the tuples in that window. The result of an execution of a loop traversal is a set of tuples associated with the time `t`. The result of the entire continuous query is therefore a sequence of timestamped sets. Remark that a timestamp can both be logical and physical.

In [10] a more concrete language is described when using the prototype system. The language is a subset of SQL with addition of the *windows* clause, which is used to specify time-based sliding windows.

2.5 STREAM and CQL

STREAM [1] (STanford StREam DatA Manager) is a project at Stanford University that includes the design and implementation of a prototype DSMS. The query language of STREAM is CQL [3] (Continuous Query Language) which is a declarative and all-purpose continuous query language. CQL is build on an abstract semantics for continuous queries, where CQL is an example of a concrete language build on that semantics.

CQL supports two data types: streams and updateable relations. Both data types are defined in such a way that either type may easily be converted to the other type. Furthermore manipulation of data is only possible on relations (because relations are the only finite data type). All operators belong to one of the following three classes: relation-to-stream, stream-to-relation, and relation-to-relation. The first two classes of operators are used for convertion between the two data types. Special care is needed when converting from streams to relations, because streams are infinite while relations are finite. This is solved by applying windows to the streams. The last class of

operators is the data manipulation operators and is equal to the operators from relational algebra. The syntax of CQL queries is based on SQL.

Chapter 3

Stream Data Models

Examining data models is the first step in understanding the theoretic foundation of stream systems. Data streams can be seen in different ways depending on the background of the people seeing it and the applications streams are intended for. A data model contributes with a formal understanding of streams which is the foundation for the semantic description of continuous queries.

Many reseach projects have contributed to define data models for stream systems, e.g. [6][1]. Even though the different models have much in common there are still many general as detailed issues that divides the contributions. A reason may be that some models are attached very specific applications. Also the fact that the modern stream systems (e.g. [6][1][7][11]) first appeared in the 1990-ies and therefore are relatively young is an important reason.

This chapter will serve to describe the most fundamental issues when defining stream data models and discuss how certain stream systems are related to these models. The dominant data model of present DSMSs—the relation-based model—is described in Section 3.1. The model is described by taking a starting point in the relational data model, which is used by major DBMSs, and considering the special needs for models in a stream environment. By using the strengths of the relational model and by considering the requirements to a stream model, a general stream model is designed step-by-step. The developed model shall not be considered as the ideal model but rather as a general and abstract model that covers the most fundamental requirements of most stream systems.

One important requirement to most streams is that a stream must be ordered. A stream is usally infinite and it is therefore important to distinguish newer elements from older ones. In Section 3.2 the motivation for ordering is described and a data model that is ordered is developed. It is also discussed how the systems in Chapter 2 support ordering.

An extended class of the relation-based models (Section 3.1) is the relation supporting models. While still designed for stream applications these models also support updateable relations. This class of models is described in Section 3.3.

Because streams are append-only they are often considered as non updateable. However, it is described in Section 3.4 how a data model also may support updateable streams.

3.1 Relation-Based Stream Models

The relational data model was introduced in 1970 [8] and has later been the dominant model in major commercial DBMSs. One of reasons why it has gained so much popularity is because of the simplicity of the model. Many stream systems that have developed own data models have used models that are, at some level, based on the relational model (e.g. Aurora [6], STREAM [1], and Tribeca [11]). Because of the simplicity of the relational model it is easier to adopt and modify it to special requirements. When combining streams with relations it is also easier if both models are close to each other. Another important reason for using a relation-based model is with respect to query languages. The SQL language is the primary query language of most DBMSs and known by almost any database user. If it is possible to implement a SQL-like language in a stream system, the system is more likely to gain popularity by users because the language is well-known. Of course it is less problematic to implement a SQL-like language if the underlying data model is close to the relational model.

In order to clarfy how a data model may be applicable for streams and still be close to the relational model, a *relation-based stream data model* will be developed in this section. The model will be related to existing data models for streams and compared if it is far or close to these models. In order to evaluate whether a model specific for streams is necessary or superfluous (according to the project goal) the stream model will be designed as close to

the relational model as possible.

In the first attempt to define a data model for streams, the definition of a relation is considered:

Definition 3.1 (Relation). A relation R is a subset of the Cartesian product of the domains $D_1, D_2, \cdots, D_n$.

At first glance this definition expresses a fundamental way of describing data that may also be suitable for streams: A stream source, e.g. a network or traffic monitor, records and streams data elements that usally are described over same same domains. There is, however, still some requirements to data streams that makes a relation insufficient for describing streams. The following definition is based on a relation but with additional expressive power which will be described afterwards (partly inspired by [3]):

Definition 3.2 (Relation-based stream). A stream S is a finite or infinite multiset of k-tuples (stream elements), where exactly one of the k-tuple elements is a tupel belonging to the same schema.

Comparing a relation from Definition 3.1 and a stream from Definition 3.2 shows that any relation can be expressed as a stream while only some streams are expressive as relations.

A stream is defined as finite or infinite in contrast to finite relations. In most applications, however, streams are infinite. When querying data on infinite streams (described in Chapter 4), the fact that a stream may be infinite is important to deal with.

While a relation is defined as a set, a stream is a multiset. This is a consequence of the property of infinity. Even though the domains may be finite a set will have a maximum size and therefore not infinite. Multisets are also more efficient to implement and are consequently also used by most relational systems.

According to the definition stream elements consist of at least the schema data and optionally also additional data. The schema data shall be considered as user or application data while the additional data is considered as meta-data that describes the schema tuples. The most obvious example of such meta-data is a timestamp for ordering the stream. Of course a timestamp, and other meta-data, can also be expressed direct as part of the tuple. However, because such data only are used by the stream operators it is most suitable that this data is separated from the schema data.

Some stream models allows streams to be *sequences* of elements, because a sequence is an ordered set and order is an important property of streams. Sequences are obmitted because sets are just as powerfull as sequences using this model. This is because order can be easily made with an extra variable in the k-tuple of a set. The importance of order is disuccesed in Section 3.2.

Finally it shall be noticed, that the just described model only supports streams (even though a stream may contain relational data it is still defined as a stream data). Some systems consider relations just as important as streams in a stream data model. This issue is discussed further in Section 3.3.

3.1.1 Related Models

In order to evaluate how representative the stream model from Definition 3.2 is, it will be compared to data models of the systems described in Chapter 2. The relation-based stream from Definition 3.2 will hereafter be denoted the *RB stream*.

Aurora and STREAM

The data models of Aurora and STREAM are formally and detailed defined. Both models are relation-based and therefore relevant for comparison with the RB stream. The definitions of streams in Aurora and STREAM are:

Definition 3.3 (Aurora stream [6]). "A stream is a potentially infinite set of tuples ordered by index values (such as timestamps or integer positions). More formally, a stream, S, is a set of (index value, tuple) pairs (stream elements):

$$S = \{(i_1, t_1), (i_2, t_2), \ldots, (i_n, t_n), \ldots\}$$

such that index values, i_j, belong to an index type (below), and all tuples, t_j, belong to the same schema." □

Definition 3.4 (STREAM stream [3]). "A stream S is a (possibly infinite) bag (multiset) of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in \mathcal{T}$ is the timestamp of the element."

By comparing the STREAM and Aurora streams with the RB stream it is clear that STREAM and Aurora streams are special cases of the RB

System	Container	Stream-only	Boundary	Meta-data
RB stream	Multiset	Yes	(In)finite	Optional
Aurora	Set	Yes	(In)finite	Index
STREAM	Multiset	Also relations	(In)finite	Timestamp

Table 3.1: Relation-based stream data models

stream. A STREAM and Aurora stream is characterized by mandatory ordering of elements and allows no possibility for extra element meta-data like a RB stream makes. The mandatory order is required because the data models are used in a system where queries depends on ordered elements (e.g. queries with window specifications). The STREAM order is more strict than the Aurora order because the STREAM order domain must be timestamps. Finally remark that Aurora streams are sets while STREAM and RB streams are multisets. This does not conflict with that fact that a RB stream is the most general one because a set is a special case of a multiset.

Table 3.1 summarizes some of the important properties of the models just described.

Other Systems

The data models of TelegraphCQ and Tapestry are not formally defined and it is therefore difficult to compare them to the RB stream. However, both systems are build upon RDBMSs (TelegraphCQ is implemented on PostgreSQL) and it can therefore be concluded that the models are at least relation-based.

3.2 Ordering

Even though a RB stream is not defined as ordered, streams are often ordered in practice because it is required by some types of queries. Each system may has its own reasons for demanding streams to be ordered and in Section 3.2.1 two very common reasons are described. In Section 3.2.2 an order semantics is proposed which builds on the relation-based models. Some order implementation issues described in Section 3.2.3 does not direct influence the data models but are still important to consider when discussing stream order. In Section 3.2.4 it is described how different stream systems deal with order.

3.2.1 Motivation

The requirement of streams beeing ordered is often primary motivated by window specifications (as the case with e.g. CQL [3] and Aurora [6]). In the next section this issue is described in more details. At a more abstract level the issue also becomes clear when comparing streams to relations. In a traditional relational database, data is updated and replaced by new data when the old data get outdated. The result of a query on such databases reflects always the updated data. Stream elements, however, do never get replaced by newer ones. New data is just added to the stream. The result of a continuous query over streams therefore reflects all data seen, valid as outdated. Because of this difference in result sets it may be important to maintain an order of stream elements so newer elements can be easily selected.

Another important reason for streams to be ordered becomes clear when using models with both streams and relations. In short when a stream element is joined with a relation, the element shall not be joined with the most recent relation but with the relation that existed at the time the element was streamed. This requires a mapping between streams and relations. The issue is discussed in details in Section 3.3.

3.2.2 Order Semantics

This section extends the unordered RB stream model to provide ordered streams. There is at least two approaches on how to order the stream elements: Either redefining the stream as a sequence or applying an order attribute to the stream elements.

The first choice excludes elements to be e.g. time ordered which makes it little expressive. The last choice allows any domain as the order attribute (however there are certain demands to the domain which will be discussed later in this section). The following stream S_o is an example of a stream, obeying Definition 3.2, that is ordered on an order attribute. Formally S_o is defined as the set of pairs of a data tuple s_i and an order attribute o_i:

$$S_o = \{(s_1, o_1), (s_2, o_2), \ldots, (s_n, o_n)\}$$

If window operators are applied to the ordered streams, there are certain demands to the ordering set, o (from [6]). A typical window is a chained

subset of a stream and o must therefore be *totally ordered*. To understand that think of a window containing the most recent i tuples from stream S_o. In this case the $\geq$ relation is used to ensure that any of the recent i tuples are greater than or equal to the remaining tuples that are not among the recent i tuples. Finally window models like time-based require distances to be calculated over the order elements. A minimum unit, e.g. seconds, will be the base of such calulations. To understand that think of a window where the distance between the two endpoints must be exactly 1 hour or a window where one endpoint is the current time while the other is current time minus 1 hour. A minimum unit, in these cases for example seconds, must be used to calulate when 1 hour is reached. As an example of two sets that are valid as ordering attribute is the natural numbers and time (with e.g. seconds as the minimum distance unit).

3.2.3 Order Implementation

A stream ordering can be categorized in many ways. Some of them listed here do not influence the data model but are included for the sake of compleness.

- Implicit or explicit ordering. When explicit ordered, the order appears as part of the stream data. Implicit order means order handled by the system implementation.

- Valid or transaction time (if timed). If data is ordered by time there are two different types of times. A *valid time* is when the associated event occured in reality. A *transaction time* is when the data was entering the database system.

- Ordered or unordered arrival of elements. If data are ordered before entering the database system it may be ordered (e.g. with valid time) or unordered on arrival.

- Tuple or element stamped. With respect to Definition 3.2 an explicit order attribute can be part of the schema tuple or the encapsulated element tuple.

3.2.4 Related Models

Because the data models of a Aurora and STREAM are formally defined they will be described here with respect to ordering. Refer to Section 3.1.1 for the definitions of the two models. Both Aurora and STREAM uses ordered streams and both models uses order attributes. The order attribute in STREAM is limited to timestamps. The order attribute of Aurora is more general and may be any type of domain as long it is totally ordered and it can be used to calulate distances.

3.3 Relation Supporting Models

Some stream models allow only streams to be input and output of the supported operators. An example of a stream-only model is the one in Definition 3.2. Other models, often relation-based models, support both streams and updateable relations. When reviewing existing systems it seems that there are very different opinions on which model to include. Of course for certain applications there is no need for relations at all. As an example Tribeca supports only relations because it is developed for network traffic monitoring where only the raw network data is of interest for analysis. Maybe the disagreement is, however, also due to the fact that the data models of streams and relations are not that far from each other (recall that according to the definitions in Section 3.1 any relation can be expressed as a stream).

There may be various arguments and motivations for combining streams with relations. Here is some of them:

- Some data in a stream system may need to be stored persistent (e.g. meta-data). Streams are virtual of nature and possible infinite and therefore not suited for persistent storage. In contrast relations are very suited for persistent storage.

- While streams are append-only (only insert), relations are updateable (insert, update, and delete). Even though relations are not more expressive of this reasons, the difference may still be important for certain applications.

- Because stream elements may be unlimited in numbers and arrive at high rates, approximate query answers (e.g. windows) are often re-

turned. In contrast the query answers to relations are always preceise which may sometimes be preferable.

- By supporting relations some of the operators and languages from relational databases may be applied with only minor modifications. As an example CQL allows only data manipulation of relations.

Extending the relation-based model in Section 3.1 with relations requires a definition of a relation. The naive approach is to use relations as they are implemented in DBMSs. That is, a relation is updateable and at any time exactly one instance of a relation is accessible. When defining a continuous query semantics (in Chapter 4) this approach, however, will be insufficient. Consider the following example:

Example 3.1. A stock market monitoring system holds a stream, S, for changes in share prices and a relation, R, for meta-data of each share. The continuous query CQ returns a continuous stream of changes in any share price on the market. Each element in CQ is a join of a share price from S and the corresponding meta-data from R. At time t_i share x is delisted from the market and the meta-data of x is therefore deleted from R. At time t_{i+1} the query semantics will face a problem: there is no related meta-data in R to join with the share prices of x before it got delisted.

The example clarifies the problem of the naive approach: When updating a relation, there may still be old stream elements refering to the replaced tuple(s). One way of solving this problem is to allow only monotonic relations—that is, relations where tuples cannot be deleted or updated. For most applications, however, it is unsuitable not being able to delete or update tuples, and therefore this solution is not preferable.

As proposed in [3] a relation must be considered as a set of *relation instances*. In other words a (stream) relation is a set of (traditional) relations. The domain of a relation must be the same domain used to order the streams. Assuming that the order set is time, any stream relation has a mapping from every time instant to a relation instance. The relation instance of relation R at time t is the instance that was valid at time t.

When combining a stream with a relation in a query, e.g. in a join, two data sets are used: at time t the stream data used are all tuples no greater than t and the relation data are the relation instance at time t.

3.4 Updateable Streams

While most stream systems consider streams as append-only (in contrast to e.g. database tables), updateable streams are proposed in [13]. In this model, a stream element consists of an identifier, a body, and a type. The identifier uniquely identifies each element. The body contains the element data. The type can be positive, neutral, or negative. A positive element adds the element to the stream. A neutral element updates the body of that element with the same identifier as the neutral one. A negative element removes that element with the same identifier as the negative one.

In many applications, for example monitoring services, it is natural that streams are append-only—when an event is observed and streamed, that fact will never change. On the other hand stream only systems (with no updateable relations) may be more powerfull with updateable streams because meta-data can be stream and later updated if necessary.

3.5 Conclusion

This chapter have contributed to describe what a stream data model is, how it is related to the relational data model, and how it is related to existing stream models. The method for describing these issues has been to develop a stream model, *RB stream*, which includes some of the main properties of streams.

The question is if such a specific model for streams are required at all or if the existing relational model is sufficient for holding stream data. There is at least two major issues that limit relations for stream data. The first is the fact that many streams are infinite which contradicts by the finite nature of relations. Secondly core relations are only capable of holding schema data and not additional meta-data that describes the schema data. There is at least one important case where additional meta-data may be required to describe stream data: when streams are required to be ordered there must be an order attribute.

It is argued that standard relations are too limited for holding streams. However, when comparing the definitions of relations and streams from Definitions 3.1 and 3.2, one may argue that a stream is in fact nothing more than a simple extended relation. While the difference in data models there-

fore may seem small, the difference of streams and relations should instead be focused more on the nature of data. For example stream data is typically more extensive in amounts and stream data is often approximate while relations are precise. The consequences of such facts regard the data processing rather than the data model.

Chapter 4

Stream Query Languages and Semantics

In the preceding chapter it is examined what a stream data model is and how it is distinct from the relational model. The conclusion is, that, a stream has some fundamental requirements that a standard relation cannot meet with. However, a it is shown that a stream model can be developed that in its basic structure is very similar to a relation.

In this chapter it will be examined if the application of streams requires notable differences between traditional relational query languages and query languages for streams. In other words—it the data models of streams and relations are closely defined, will query languages like relational algebra and SQL consequently also be working on streams? If this is not the case it will be examined *why* this is not true and *how* a stream query language may be designed.

The most important distinction between traditional queries and stream queries is, that stream queries are continuous. In Section 4.1 it is explained what continuous queries are and what they are motivated by. Without going into detail of the query operators an overall semantics of continuous queries is given in Section 4.2. Different aspects of query languages for streams are described in Section 4.3 and in Section 4.4 details and characteristics of some of the most important operators are given.

4.1 Continuous Queries

The obvious contrast to continuous queries is *one-time queries* which are the standard way of querying in traditional database systems. A one-time query is executed once, the result is passed back to the querying user, and finally the query is destroyed. The query result set hereafter remains static even though the database data may change. The fact that the querying user only gets a single static result raises some problems when querying streams. These issues is discussed next.

The first issue regards the stream data model. Recall from Chapter 3 that streams are defined as potentially infinite (sometimes denoted unlimited). From a theoretical point of view this entails that it is impossible to get a final result set of an query over an infinite stream. If a traditional one-time query is executed on such a stream the query may block (no tuples will be returned at all) or at least run in infinity. To overcome this problem the query must continuously return an approximate result of the data so far seen.

Another issue regards the fact that streams are virtual and not totally stored in database system. When data are disposed only short time after arrival it is useless to run a one-time query because the database will contain no or only a small subset of a stream. Therefore a query must be submitted and stand in the database so it can respond as soon new data arrives. In other words data waits for one-time queries while continuous queries wait for data.

The last issue regards the responsibility of database systems and users of these systems. As described in [6] a traditional DBMS operates with the *Human-Active, DBMS-Passive* (HADP) model. This means data are stored in a database and the users are the active part querying data. The model does not fit very well to stream application, especially monitoring applications. Here users are waiting passive (on a continuous query) for changes in the monitored domain while the DBMS streams changes of interest, hence denoted the *DBMS-Active, Human-Passive* (DAHP) model.

Because streams most often are non-persistent the elements are only available to queries on the arrival and a short time after. When queries are referring back in time (which most windows do) recently submitted queries will have trouble in obtaining data for the result. Such queries are named

ad hoc queries because they may be submitted at any time. Some systems, however, allow only queries to be submitted *before* the involved streams are initiated, that is, before the first element is appended to the stream. These queries are named *predefined queries*. The design of the operators for continuous queries depends on which of these two query types that are allowed.

4.2 Continuous Query Seamantics

Later in this chapter detailed semantics is discussed that describes on lowest level the result of continuous queries. This section discusses abstract semantics of continuous queries. It can be considered as black box semantics because it gives some abstract requirements without dictating concrete design.

One naive approach to evaluate continuous queries is to execute the queries periodically and return the query results to the users after each execution. This strategy may be both simple and efficient. It is simple because it can easily be implemented on a traditional DBMS. If the execution intervals are relatively long and the sizes of the result sets are relatively small, the strategy is also efficient because the DSMS query processor will be idle for most time.

From the following example, inspired from [12], periodic evaluation, however, may lead to nondeterministic result sets:

Example 4.1 (Nondeterministic query). Let $Message$ be a stream of messages where a message consists of an id and a reply reference to a message id if the message is a reply to another message. The schema is:
$Message = (id, reply)$
At time 0 the stream is empty. A new message is appended to the stream at time 5 and a reply to that message is appended at time 15. The stream contains the following elements at time 5 and 15:
$Message(5) = \{(100, null)\}$
$Message(15) = \{(100, null), (.., 100)\}$
The following query, Q, is executed by two users, a and b, every 20th time instant and the final results, Q_a and Q_b, is the union of each execution of Q:
`SELECT id FROM Message WHERE reply = null`
For user a the query is executed at time instants $0, 20, 40, ...$ and user b at

time instants $10, 30, 50, ...$
The two users final result sets after the last element append are:
$Q_a = \{\}$
$Q_b = \{(100)\}$

To summarize, the query is asking for messages with no replies, and the only time where this was true was between time 5 and 15. Because b was the only user running the query between these two times (more specific at time 10), user a and b experience two different results even though they have run the same query on the same stream.

To overcome this problem the semantics of continuous queries must be time-independent. So instead of running a query with time intervals the query must be executed at *every time instant*. A time instant is the smallest discrete time unit a stream systems registers.

Formally the result of the continuous query Q_c run at time t is the union of the results when running the query Q at any time instant from 0 to t [9]:

$$Q_c(t) = \bigcup_{s=0}^{t} Q(s) \tag{4.1}$$

Note that the result of running query Q at time t means the set of tuples (or whatever data representation used) that is returned by the database at the state the database was in at time t.

Running query Q from Example 4.1 continuously as in Equation 4.1 yields the following result set (where $t \geq 15$):

$$Q_c(t) = Q(0) \cup ... \cup Q(5) \cup ... \cup Q(15) \cup ... \cup Q(t) = \{(100)\}$$

As seen no matter who is running the query the same result set is obtained assumed it is possible to calculate the result of query Q at any time up to time instant 15.

Equation 4.1 can be further optimized if a certain property of Q is valid, namely that of monotonicy which is defined as:

Definition 4.1 (Monotonic query). Let $Q(t)$ be the result of running query Q at time t. Then Q is monotonic if and only if $Q(t_1) \subseteq Q(t_2)$ whenever $t_1 \leq t_2$.

If a continuous query is monotonic it is sufficient to evaluate over the new tuples arrived since last time instant. If the query is nonmonotonic all tuples

should be evaluated at every time instant as in Equation 4.1. Formally the result of the monotonic continuous query Q_c runned at time t is [9]:

$$Q_c(t) = \bigcup_{s=1}^{t} \Big(Q(s) - Q(s-1) \Big) \cup Q(0) \tag{4.2}$$

Continuous queries which update or delete tuples that have previously been in the result of the query is non-monotonic. Queries executed on append-only databases is intuitive monotonic. However in certain cases such queries can also be non-monotonic. This includes queries with call to time functions and "not exists" constructions. In [12] it is described how to convert nonmonotonic queries to monotonic.

4.3 Language Strategies

This section describes some of the important issues that impacts how a stream language is designed. In Section 4.3.1 it described how the requirement of continuity influences the result of a query. While most DBMSs use SQL as primary query language it is very different from various stream systems. In Section 4.3.2 it is described which language syntax stream languages use. In Section 4.3.3 it is described how different characteristics of streams may influence the operator design.

4.3.1 Continuity

The fact that streams are infinite require queries to be continuous in order to continuously return updates in the input elements seen so far. The result of a continuous query can be presented in different ways. Because the result is continuously modified it can be considered as a regular stream which, e.g., may be input to other continuous queries. Even though the output of stream queries are often streams, they can also be presented in other ways. In Aurora a query result can be returned as a view. In contrast to streams which is directed to a specific application, a view has no receiver and may be accessed be any application. CQL operates with both streams and relations, and both types can be specified in the query to be the result type.

As described in Section 4.2, if a query is monotonic, it is sufficient to execute the query only over new arrived data. Whether to use non-monotonic

query semantics (Equation 4.1) or monotonic query semantics (Equation 4.2) has great impact on performance and leads to two different result set for the same query over the same stream. CQL takes this idea a step further by providing three types of output streams: *insert, delete,* and *relation streams.* When applied as the outermost operator the insert stream operator provides the semantics of Equation 4.2 and the relation stream operator provides the semantics of Equation 4.1. The delete stream operator acts opposite to the insert stream operator, and returns elements that is deleted from the query result. Which CQL stream operator to use is user specified in each query.

4.3.2 Language Syntax

Comparing different stream systems shows that there is no agreement in using a specific syntax for stream query languages. Some languages are declarative while other are procedural, and some are very close to SQL while other have no SQL like syntax at all.

The advantage of using a procedural language is clear: the dataflow between operators in a query can be controlled entirely by a user. Stream applications like sensor and network traffic monitoring operate with heavy amounts of data and the dataflow between operators therefore has a great impact on the performance. Moreover some stream operators require more arguments than relational operators and a nonprocedural syntax would therefore be quite complex.

The Tribeca query language is procedural and a query is a list of operators. The output of an operator is named with an identifier which may be referred as input to any other operator.

An Aurora query is also procedural but is expressed in a graphical user interface. An operator is represented as a graphical box and the dataflow may be directed from one operator to another using a graphical line between two boxes.

As described in Chapter 3 many stream data models are defined with a subset in the relational data model. Consequently many streams systems use SQL in a more or less adapted version as the query language. An advantage of using SQL is that the language is known by almost every user familiar with querying in databases. Whether people like the declarative syntax of SQL or not, it is a fact that it is has much in common of expressing data in

human language. Another advantage of using SQL is that the language has been implemented on various database systems and much experience may be reused that way.

Stream systems like Tapestry and Telegraph are implemented on DBMSs and the query language is therefore SQL, though Telegraph has minor additions, e.g. windows.

All operators in CQL, except these that convert between streams and relations, must be operated on relations. This clearly makes it easy to use SQL as querying language even though the semantics does not make any assumptions on which syntax to use.

4.3.3 Operator Requirements

In Section 4.4 some basic operators are described in details. One of the reasons why these operators distinct from relational operators is because of some general characteristics of streams. In the following sections two important characteristics are described.

Blocking

When executing certain operators on sets where the entire set may not be fully available once (e.g. when a stream is infinite) it is possible to end up in a state where an operator is blocked. This means that no tuple will be produced before the entire set has been seen. Examples of blocking operators are certain aggregate and join operators. In contrast, for example projection and filtering operators are usually non-blocking. Of course no operator semantics must be blocking so that is why e.g. windows specifications are applied in many situations. In Sections 4.4.2 and 4.4.3 examples on operators that are blocking are given.

Order

Some systems have operators that require streams to be ordered, or at least an order within some bounds. The problem of unordered streams occurs when there is requirements (according to the query) to the order of the input data to an operator. By order means the order stated by the order attribute of the data elements (e.g. timestamps) and not the order of which the elements are appended to the stream. As an example of a problem

with unordered streams, consider the operator that returns, every hour, the average value of the last hour of elements on an unordered stream. The operator will face a problem: if just a single element arrives late, for some reasons, the average over the window may first be computed when the last element has arrived (and this may take hours or days) or it may be skiped (which is not desirable).

One way of solving this problem is to use transaction time when ordering tuples (as described in Section 3.2.3 in Chapter 3). If valid times are used the DSMS or operator semantics must try to deal with the late arriving tuples.

In Aurora elements may arrive unordered and to catch late arriving tuples, all order required operators have an additional order specification: `Order (On A, Slack n, GroupBy B1, ... , Bm)`. Such a specification tells an operator that elements may arrive at least n elements out of order with respect to the order attribute A grouped on the attributes B1 to Bm. With a finite buffer of a size depending on n, this specification allows operators to process almost sorted data.

4.4 Query Operators

Four different query operators will be examined further in this section because they all, in some way, are examples on operators that add expressive power to stream queries or solve problems that always appear when querying on streams.

The partitioning operator (Section 4.4.1) is an example of an extra operator that is only useful for streams because append-only streams alwalys contain history data. The functionality of the oprator can also be obtained by relational algebra but it will lead to much more complex queries then.

The aggregation (Section 4.4.2) and join (Section 4.4.3) operators are two examples on operators from relational algebra that may not be applied to streams with their original semantics. In the case of these operators they will make a query blocking and it is discussed how to avoid that.

Window specifications (Section 4.4.4) are an example of operators that are so expressive that it has become almost standard for query languages. Windows both solve problems like blocking operators and add additional expressive power because users may specify to query in only a subset of streams.

4.4.1 Partitioning

Issuing continuous queries over data streams require special needs for partitioning data. When a stream element gets outdated it is not replaced by a newer one as in DBMSs. Instead new elements are continuously appended to the stream while the old elements remain. A stream therefore contains several versions of single identities which it may be desirable to query advanced operations on. Consider as an example a stock market. Of course a DBMS may also register the history of share prices but it is very obvious to maintain only one tuple for each stock with its most recent price. For streams the old share prices will always remain on the stream. It may therefore be desirable to make advanced calculations on each share identity.

Even though the aggregation operator, $\mathcal{G}$, from relational algebra and the GROUP BY clause from SQL allows some kind of grouping functionality, they are too limited for certain stream applications. The aggregation operator only allows aggregate functions to be executed on each group of tuples.

Tribeca introduces *demultiplexing* and *remultiplexing* of streams which is used for partitioning streams. The demultipexing operator (demux) partitions a stream into multiple substreams. The demultiplexed substreams are named with an identifier and they are referred by that identifier until they are remultiplexed. A demultiplexed stream can be given as input to operators just like ordinary streams. The multiplexing operator (mux) combines a set of partitioned streams to a single stream that contains all tuples from all substreams.

Returning to the example of the stock market system it will be easy to separate the share price history of each share from each other by demultiplexing on the share identity. This allows advanced operations, also other than aggregate functions, to be applied on each share's price history and finally combining all results with the remultiplexing operator.

4.4.2 Aggregation

Aggregation operators are problematic to perform on unlimited streams because they need to see the entire stream before they can return a concrete result. One simple approach to overcome this problem is to continuously run the aggregate function over all tuples from the stream up to the execution

time. In that way temporary results will continuously be returned and based on the data seen so far. For the five common SQL aggregate functions only finite space is required for this approach while other functions may require infinite space (e.g. the function that returns the middle value because it must know the value of all so far seen tuples). For predefined queries an aggregate over the entire stream may be desirable for certain application. If ad hoc queries, however, are allowed, the result of a query is nondeterministic because it is different how much each query has reached to see of the beginning of the stream.

Consider a stream where data has been appended for days, maybe months. How useful is it to get an aggregate over the entire stream if you can only say that the aggregate covers data from "when the stream was initiated" to present time? For most situations it is more desirable to have bounds on the data to the aggregate function, so queries like "the average of the last week" can be issued.

In Tribeca the aggregation operator returns the aggregate value of a stream when all elements have been processed. To avoid ending up in a blocking state the stream must be finite or the operator must be used together with windows.

Aggregation in CQL is made just like in SQL. To make an aggregation over an infinite stream a relation in terms of a finite window must be constructed from the stream. For every time instant such a relation is created and the aggregate value on that relation is return to the query issuer.

Sliding windows are defined as part of the aggregation operator in Aurora. For an aggregation the size and how to advance must be specified for the related window. As an example a window specification can be tuples from the last three hours, sliding every hour. The aggrigate function is one of the SQL functions or a user defined function. Aggregation can, like in relational algebra, be grouped on several attributes. Furthermore tuples are not required to be delivered in order. Therefore it is possible to specify a number of tuples to look forward before a final aggregate on a window is computed.

To conclude aggregation on streams is often made by requiring a window specification on the data that enters the aggrigation operator. In Aurora window specifications are part of the aggrigation operator while Tribeca and CQL require windows applied explicitly.

4.4.3 Join

Joining two or more infinite streams are problematic because a complete result set cannot be returned before the complete streams are known. Consider the join of stream s_1 and s_2, where s_2 is infinite. Initially the first tuple of s_1 is joined with every tuple of s_2. However, because s_2 is infinite, this process will never terminate, and the other tuples of s_1 will therefore never be joined. As with many other stream operators, a join is at least required to be executed on a finite subset of a stream, obtained e.g. with a window specification.

All joins in Aurora are performed on two streams. An order attribute is specified for both streams, and for two tuples to be joined, the distance between the two order attributes must to be beyond a specified value. Furthermore, because tuples do not need to arrive in order, the operator can look forward a specified number of tuples in the stream to see, if other tuples are beyond the difference. A basic join in Aurora is the Cartesian product of two streams. A user specified predicate allows advanced types of joins.

Tribeca does not support traditional joins because of performance reasons. However, a sort of join is possible by using window filters that combines a stream with a sliding window. Each stream element is compared, by a list of user specified predicates, to all elements in the present window.

Because join operators in CQL only are allowed on relations, which are finite, it is without problems to query a join on two relations.

4.4.4 Windows

Windows are an important part of many stream system data models. The main responsibility of windows is to give approximate answers to continuous queries. This is motivate by both a need to overcome technical limitations when executing certain operators and to provide users with more flexible querying techniques. Remember that data streams are usually characterized by heavy amounts of data relatively compared to traditional relational databases.

Consider how the projection and selection operators can be implemented to support unlimited data streams: when one stream element (or tuple) enters one of the operators, it is possible to give a result based on that single element. A aggregation operator, however, is first able to give a result when

all elements have been processed. The same is true for join operators. Due to limitations in memory, it is difficult to implement such blocking operators on unlimited data streams. Instead, windows can be used to bound a limited part of the streams, to which the operators are to be executed on.

In some situations it is desirable for users to query in only a subset of data streams. If a data stream contains data for a long period of time, users may want to query in only the most recent data. Another situation is data streams where new data are recorded very often. Here users may want only to query in, for example only every tenth element. Such requirements can easily be specified in continuous queries when using windows.

Endpoints and Measurement Units

Various window models have been proposed in different stream systems. A common way of defining windows is in terms of direction of movement and measurement unit of the two window endpoints. A fixed endpoint is a point on a element that is never changed, while a sliding endpoint is a point that moves in one the two directions (moving to either newer or older elements). A fixed window has two fixed endpoints, a sliding window has two sliding endpoints, and a landmark has both a fixed and a sliding endpoint. The unit used to defined endpoints is usually physical (time-based) or logical (tuple- or count-based).

Example 4.2 (Physical fixed window). Window w_f, which contains all daytime elements, is a physical fixed window. The left endpoint is fixed at timestamp 12.00 and the right endpoint is fixed at timestamp 18.00.

Example 4.3 (Logical sliding window). Window w_s, which contains the last 100 elements, is a logical right moving sliding window. The left endpoint is the total number of tuples minus 100 and therefore moving right when new elements arrives. The right endpoint is the newest element and therefore also moving right.

Example 4.4 (Physical landmark window). Window w_l, which contains all elements newer than timestamp 8.00, is a physical landmark window. The left endpoint is fixed at timestamp 8.00 and the right endpoint is moving right at the newest element.

Note that even though an operator is executed on a data stream window rather than the entire stream, the result will always be deterministic. This means that two identically window queries will always return the exact same result if they are executed at the same time.

Related Window Models

CQL Window specifications are supported by CQL. The abstract semantics allows various window types while three concrete types are part of CQL. Windows are produced by the stream-to-relation operators because this class of operators converts the infinite (or finite) streams to finite relations. The two other classes of operators work on finite sets and therefore they do not need to use windows.

The three window specifications of CQL are chained right sliding windows and they work on single streams. The right endpoints are locked at the newest element in the corresponding stream (with respect to the order domain). The left endpoints are user specified as physical or logical relative to the right endpoints.

Time-based windows are all elements from a stream with timestamp no older than a user specified timestamp. Tuple-based windows are the N elements from a stream with most recent timestamps, where N is user specified. Partitioned windows are similar to tuple-based windows except that N is with respect to one or more attributes instead of the entire stream (this is similar to the GROUP BY operator in SQL).

Tapestry Tapestry does not support window specifications. The system is implemented on a RDBMS with finite relations and some of the problems that motivate windows are therefore not present.

TelegraphCQ The window specifications in TelegraphCQ are very expressive. A window is defined by a stream and a two endpoints. For every time instant the window is updated according to its two endpoints. The window definition is contained in a for-loop where every loop traverse corresponds to an increment of time instant. The endpoints may be fixed time instants or defined relatively to the loop attribute. Refer to Chapter 1 for an example of a TelegraphCQ window specification.

By allowing users to control the values of both endpoints for every time instant make TelegraphCQ windows very expressive. For example it is possible define windows that slide backward and windows that jumps, that is, only include every x'th element from a stream.

4.5 Conclusion

This chapter has taken a step further from the preceding chapter and examined how it is possible to query data stored using a stream data model. Even though the relational and stream data models may be defined very closely it is somewhat more difficult to answer how stream query languages distinct from relational query languages. This is because a language may be constructed in many different ways depending on the application of streams. As examples, some users prefer precise answers while other prefer approximate answers, some users require real-time continuous results while other are satisfied with periodical results. It is therefore difficult to design an ideal language syntax and semantics that most stream researchers would agree on. Rather it is possible to say something about properties of streams that demands special care when designing a stream language. Even though stream languages may seem very distinct from each other, there is constructions which are adopted by most languages and some of these and described in the following.

The most conspicuous property of stream languages in general is that of continuity which is commonly not available in relational languages. The reason of why only queries on streams are continuous is because of the application of streams. The question is then: shall relational languages consequently be rejected as query languages of streams? The question may be answered by considering Equation 4.1 which says that a continuous query is in fact just a traditional query executed many times. Of course a traditional DBMS is not geared for running the same query every smallest time unit. A DBMS must therefore be adjusted to be able to handle continuous queries.

With respect to the language syntax and operator semantics of query languages, it is, with the stream model from Chapter 3, possible to get a long way by using relational languages like SQL and relational algebra as stream query language. In fact systems like STREAM and Telegraph have proved that SQL may be used as a functional stream language with some

modifications. Also if basing the underlying operators on the relational algebra operators, it requires no or only some modifications in order to operate them on streams. As an example on a required modification is that of non-blocking operators. It is a requirement for certain operators that compute its result from more than one tuple that it does not block because it is waiting for future data. This is most often solved with windows which are included by most stream languages.

To summarize there may be very different requirements to stream queries which leads to different languages, each seeking to provide the most simple and intuitive way of expressing data results. It is therefore not reasonable to state that any stream language that do not build on SQL or relational algebra do not has a legitimacy. However, it is not fair to reject relational languages as query languages for streams because the demands to stream languages are not that different from relational languages.

Chapter 5

Conclusion

As described in Chapter 1 data streams are a way of conceiving data where data is seen as a continuous stream rather than single discrete data elements. Over more than a decade much research has been made to understand how we may consider such continuous streams and how systems may be designed to handle them. There is, however, very different opinions on how to consider streams and how closely they are related to the dominating relational model. This project has focused on examining data models and query languages for stream systems and evaluating how streams are differentiated from relations. It is important to know this difference when deciding if streams may be managed by modifying existing relational database systems or if new systems need to be build from scratch to handle that data type.

In Chapter 3 a stream data model was designed with the most important properties of a stream model and it was related to existing models. The conclusion was that the standard relational model is insufficient for storing streams but by extending the model with e.g. infinity and meta-data, it is possible for such a model to support both streams and relations.

So if the distinctions do not seem big at the level of data models, what about the languages that query on streams? This was examined in Chapter 4 by describing what requirements query languages for streams have. A major difference of relational languages and stream languages is the fact that queries on streams must be continuous because the stream itself also is continuous. There is different approaches on how to make queries continuous but in an overall perspective a continuous query is a repeated execution of ordinary queries. This means that, even though continuous queries are far

from equal to one-time queries, it is possible for a traditional database system to support continuous queries by some modifications.

It was also examined how the syntax and operator semantics of relational languages may be applied to a stream query language. In fact SQL and relational algebra do not require much adjustments when applied to streams. Some language constructions, such as windows, are however necessary to incorporate in a stream language because there are fundamental differences between streams and relations.

5.1 Levels of Differences

To summarize I have examined some of the theoretical foundation of data streams, namely data models and query languages. When considering a database system, e.g. a DBMS or DSMS, these two topics may be seen as the basis of the system. The data model describes the data and because data is what database are about, a data model is the first level of a database. Because data, most often, are modifiable we use data manipulation languages to retrieve and change data. Because streams are append-only and the insertions are simpler compared to relational insertions, querying is the most interesting part of these languages. Query languages may therefore be considered as the second level of a database. At the next level, which this project not has reached, we find the data processing. As described in Section 1 the input and output of a DSMS are streams. This level manages to process the incoming data and generating data as result of queries.

As described in Chapter 1 certain people believe that streams are totally different from relations and consequently that DBMS are incapable of handling streams at all. In this project I have examined this claim with respect to the first two levels of database systems just described. My conclusion is that at the first level, data models, there is little differences between the relational data model and the stream data models. At the second level, query languages, I have found some differences. However, the differences are not greater than that the fundamental requirements to a stream query language and a relational query language are not far from each other.

As described, this project has not examined the third level, data processing. However, I believe, that if notable differences between streams and relations and their corresponding database systems exist, they must be valid

at this level, rather than the two first levels. As described in Chapter 1 it is a fact that stream applications are characterized by high data rates and requirements to fast query responses. These issues make great requirements to the data processing of database systems.

Chapter 6

Summary

The topic of this project is data models and query languages for data streams. In short a data stream is a continuous and possible infinite sequence of data elements. A stream may be compared to the dominating model of DBMSs, the relational model, where a relation is finite and data is changed by discrete insertions, updates or deletions rather than streaming the updates as new elements. Because streams are continuous and infinite queries on streams are also continuous and long-running.

Some research project have developed prototype database systems that handle streams. Where some systems are developed as an extension to existing DBMSs some researchers claim that streams are so different from relational data that new data models and database systems must be developed for streams. This project has examined closer if this claim is legitimate by reviewing existing stream systems. There is two conclusions which have been obtained by examining data models and query languages.

Most stream models are inspired by the relational model and they are therefore denoted relation-based stream models. It is shown how a stream model may be designed that is based on relation but has some additional power. This includes e.g. infinity and element meta-data. Another group of models is also described that combines both streams and relations—denoted relation supporting models. Because streams are infinite it is useful when the elements are ordered and it is described how this may be build into a model. It is concluded that it is possible to design a stream model that is very close to a relation.

Stream queries are continuous and it is described what this means and

how it impacts the query languages. A concrete semantics is given that describes what the result of a continuous query is. Different language strategies is described with respect to language syntax and operator semantics. It is concluded that the relational languages like relational algebra and SQL may also be applied on streams but it requires some adjustments. For example it is very common to use window specifications in stream queries that makes a bound on which part of the stream to query on. Even though it is not required some stream languages have extended or additional operators because it makes queries simpler and more intuitive, and some of these operators are described.

Bibliography

[1] Stanford stream data manager. `http://www-db.stanford.edu/stream/`.

[2] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM Press, 2002.

[5] K. R. Bille, A. S. Jensen, M. Maach, and J. Søndergaard. AWHERE – Handling of Data Streams in a Mobile Context. Master's thesis, Aalborg University, Denmark, 2005. `www.cs.aau.dk/library/files/rapbibfiles1/1105501345.pdf`.

[6] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman,

F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[9] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[10] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah. Telegraphcq: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

[11] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annual Technical Conference (NO 98)*, pages 13–24, 6. 1998.

[12] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330. ACM Press, 1992.

[13] E. Vossough and J. R. Getta. Processing of continuous queries over unlimited data streams. In *13th International Conference on Database and Expert Systems Applications (DEXA)*, 2002.