Athens University of Economics and Business
Department of Management Science and Technology

Combining Relational and Stream Data for Real-Time Analytics

Yannis Sotiropoulos

Thesis submitted for the degree of

Doctor of Philosophy

July 2013

Abstract

Nowadays, more and more organizations realize the importance of analyzing available data. While most of the times data is stored, over the last years there is a growing amount of stream data. Such data arrives on-line from multiple sources in a continuous, rapid and time-varying fashion. Some well-known stream applications are: sensor networks, RFID supply chain management, financial analysis, network and environmental monitoring. The importance of stream data management has been identified by many researchers and as a result stream data processing has gain the focus of intense research activity in the past few years. Moreover processing data streams has been identified as a crucial element for real-time enterprises (RTEs).

Currently, most stream management applications and systems exploit stream data with the objective to answer monitoring queries. However, the real potential of stream data lies in the possibility to capture new types of information in (near) real-time and support decisions. To support this kind of analysis we need analytics queries that can support multiple and correlated stream aggregates over stream data coming from multiple and heterogeneous stream sources. Moreover a wide range of analytics applications need to combine already available data (e.g. stored data) and stream data to empower business with (near) real-time insights that can be used for improved decision making. As relational databases are extremely widespread our research focuses on how relation data can support relational-stream analytics applications. Overall, this thesis provides query formulation methods and tools that combine relational and stream data to support (near) real-time data analysis.

In this thesis we introduce stream variables to support analytics over stream data. This kind of analytics queries can contain multiple stream aggregates, correlated stream aggregates and use data from multiple and heterogeneous stream sources. We provide SQL language extensions to support this kind of queries. These extensions are minimal, succinct and easily understandable by common SQL users. Moreover we provide a spreadsheet-like approach to perform stream analytics. The intuition is that stream queries can by defined in a column-by-column fashion. The columns can contain either relational data or stream aggregates. We argue for the easiness of this approach and that the developed tool can be useful in real world applications.

The thesis studies how to extend current Relational Database Management Systems (RDBMSs) to handle stream data for (near) real-time decision making. We present a relational-based integration framework that sits atop any RDBMS and mix RDBMS' data and stream aggregates managed by different stream systems. A SQL extension is provided to define relational-stream views and an API is developed to carry out the required communication between the relational and the stream systems. The proposed framework can serve as a standard for relational-stream interoperability.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview and Motivation

Today's complex world requires state-of-the-art data analysis over truly massive data sets. Until recently, data were stored and processed in database systems. Processing persistent data has been the main focus of the database research community for many years. However the technological advances in sensor technology along with the emergence of web and mobile services gave birth to a new generation of data applications. These applications must handle data items that arrive on-line from multiple sources in a continuous, rapid and time-varying fashion [21]. Example applications include financial applications (streams of transactions or stock ticks), network monitoring (stream of packets), telecommunications data management (stream of calls), web applications (click-streams), sensor networks (RFID data) and location-based services (GPS data). This new class of data stream applications has recently attracted a lot of attention from database research community and the current thesis belongs to this area of research.

The volume and the high speed of continuous data flows make extremely hard to store the data in a database system. Also, the "store-and-then-query" data processing paradigm is not suitable for stream data. In most cases users need to get results as fast as possible so the computation must be performed on-the-fly as the data enters the processing system. In stream applications data is processed with "continuous queries" [22], which provide results continually as new stream data arrives from stream sources. Continuous queries are used for monitoring and alerting operations i.e. when a condition is satisfied or a specific event takes place an alert mechanism is triggered. Additionally, some stream data may need to be stored for offline data analysis.

The database research community has responded to data stream application needs with an abundance of ideas, prototypes and architectures to address the new issues involved in this field. From a relational perspective the stream data is modeled not as persistent relations but rather as transient relations [12]. Apart from simple and efficient stream querying (e.g. simple filtering queries over streams), how to best model, express and evaluate analytics over data streams is a challenging problem. The purpose of analytics is to help analysts make informed decisions by

uncovering insights hidden in large volume of data. Analytics over data streams can be used for real-time decision making. For example the ability to make decisions on-line (i.e. as data stream arrives) is extremely important for critical tasks that have significant economic benefits for large companies (e.g. telecom fraud detection). Stream analytics requires data modeling, rich querying capabilities and novel evaluation processing techniques. Additionally the time plays a central role in stream analytics as metrics of interest (e.g. moving average) are computed over different time-scales (e.g. hours, minutes or seconds). Stream querying with these characteristics will be a crucial component of any future data management and decision support system [4][39]. One of the most challenging tasks in decision making using stream data is the transformation of raw stream observations to information that is understandable by analysts [151]. In most cases raw stream data lacks semantic meaning, making them inappropriate for business applications. Furthermore stream enabled infrastructures generate large volume of raw data and passing high volume raw stream data observations directly to applications and users is not a proper solution. To support stream data analysis and alleviate the high volume problem stream data must be summarized and analyzed continuously as the data arrives i.e. we need continuous analytics over stream data.

Data streaming technologies are an evolutionary concept in the field of databases. While a lot of work has been done in analytics over offline data [47][48] the support of analytics over streams is in ongoing research and the thesis studies this problem. The continuous flow of stream data makes such queries difficult to be defined and evaluated. Online computation is one aspect of applying analytics over stream data. On the other hand, analytics over data streams involves data synthesis of stream data with other types of data. As relational database systems are the most popular and widely used data systems for the storage of structured data, the capability to combine stream and relational data for analytics purposes and decision making is of great importance. A key observation is that in most cases stream values are bound to a relational value. For example a stock has a price; a sensor provides a stream of temperature measurements and a web user with a specific internet address generates a stream of clicks. The details of stocks, sensors and users are stored in a relational database and provide the semantic meaning that can be associated with stream data.

Applying continuous analytics over stream data is a crucial element for real-time enterprises (RTE). Until recently data is collected in centralized places allowing analysts to extract useful information by issuing decision support queries. In a typical scenario, an organization stores detailed records of its operations in a database, which are then analyzed to improve efficiency, detect sales opportunities, identify irregularities, verify hypotheses, segment customer base, and so on. Performing complex analysis on this data is an essential component of these organizations' businesses. These technologies are collectively known as Business Intelligence (BI). However, in stream applications the huge amounts of continuous data makes it impractical to store all the data

at a centralized site. Real-Time Business Intelligence (RTBI) is a new research area providing techniques that can be used for analysis of data and events as they occur. RTBI enables passive organizations to be transformed to active in order to respond immediately to business needs. As a result decision making is “tactical” rather than “strategic”. Stream data analysis is a common RTBI technique and has gain many supporters in recent years. In this thesis we provide methods and tools to support Real-Time Business Intelligence applications. A real-time data report that combines relational and stream data for analytics purposes is a useful tool to achieve this goal. The current thesis describes query languages, architectures and frameworks on how we can define and evaluate such (near) real-time data reports.

Except from combining relational and stream data at the logical level (relational tuples with stream data) the capability of current Relational Database Management Systems (RDBMSs) to handle stream data enables relational-stream system level integration. In this way users/analysts can use stream data inside their current database systems. As a result they can express queries that can use the already available relational data exists in their relational database systems. The challenge is that while the stream data is provided by a stream source or by a specialized stream processing engine, stream data must be transparent to RDBMS users. Being able to easily express and efficiently evaluate queries over heterogeneous stream data sources and combine stream data with relational data to create integrated analytics applications is a major challenge in data management field and is the focus of the current research.

As stream sources continue to increase a vast amount of data will become available. The analysis of stream data is becoming crucial for companies and organizations as there are many practical applications and business needs. Combining relational data with stream data for analytics purposes and decision making is a large part of these applications and motivates the research described in this thesis. Also providing to users the capability to use stream data in their current relational database systems makes stream processing available to a large number of current database users showing the practical aspect and the usefulness of the research conducted in the current thesis.

1.2 Research Challenges

The goal of this research work is to design query formulation methods and query processing algorithms for stream data. The focus is on queries that can use stream data coming from heterogeneous and possible distributed stream data sources. Such queries process stream data on the fly and provide results continuously. Moreover, these queries can use relational data either to define the semantic meaning of stream data (e.g. for each stock compute the running maximum and average price) or the relational data enhances query results with historic information (e.g. for each stock compute the running maximum and average price and compare with the maximum

prices of previous week). In both cases stock's details are available in relational data existing in a RDBMS while the running maximum and average prices are computed on-the-fly from stream data.

The first challenge is how the relational and stream data can be combined to form analytics queries in a semantic level. The focus of our research is on analytics queries similar to group-by aggregate queries. The group-by attribute is the relational part while the aggregates are produced from stream data. The second challenge is how to provide richer analytics than simple group-by aggregate queries. Complex queries can contain multiple group-by attributes and stream aggregates, correlated stream aggregates and stream aggregates from multiple and heterogeneous data stream sources. Such queries are useful for (near) real-time decision making. We provide two methods to achieve this: an SQL-like approach and a spreadsheet approach. In the first case the theoretical foundations and an extension of SQL is provided to support stream analytics queries. In the second case a spreadsheet-like query method is provided that enables the declaration of complex analytics queries in a spreadsheet fashion, column-by-column.

Moreover extending a relational Database Management System to combine already available relational data with stream data coming from various sources for (near) real-time data analysis and decision making is a major challenge. A framework that uses the database engine in a collaborative fashion with stream engines is provided.

To address these challenges several issues have to be taken into consideration, both from a theoretical and a practical perspective:

- **Analytics over stream data:** The main approach to process infinite data streams is continuous queries, which provide results continually as new data arrives from stream sources. In most cases such queries are simple monitoring queries. On the other hand, applying analytics over stream data is useful for (near) real-time decision support. Such queries use multiple and heterogeneous stream sources, combine multiple stream aggregates and can use offline relational data. So, the research questions are:
  - *How one can model and perform analytics over stream data?*
  - *What are the semantics of the query language to support this kind of queries?*

  A number of issues must be taken under consideration trying to answer these challenges. Firstly, the provided query formulation methods must be expressed through a simple language that it is easily understood. Also efficient evaluation and optimizations must be supported.
- **Relational Database Management Systems (RDBMSs) and stream system integration:** One of the main concerns is how to extend current RDBMSs to handle

queries that use stream data. While many systems have been developed to address the various challenges present in stream applications, few deal with simple SQL extensions to incorporate stream processing in present relational systems. Additionally a relational and sound theoretical framework must be provided. So the research questions are:

- *How database users can use stream data in their current relational database systems?*
- *What is a relationally sound framework for RDBMS-stream systems integration?*

Similarly a number of issues must be taken under consideration trying to answer these challenges. The most important requirement is that the usage of stream data in the RDBMS must be transparent to database users. Relational semantics must be applied in the RDBMS while stream engines must handle stream data using their native stream processing language. A data protocol between RDBMS and stream engines must be developed to allow the usage of stream data from database systems.

1.3 Contributions

The goal of the thesis is to provide methods and tools on how stream and relational data can be combined in order to be used for real-time analytics. Real-time analytics involves rich querying capabilities for (near) real-time decision support. We focus on analytics queries that aggregate stream data and combine them with relational data. Below we provide the contributions of the thesis:

- We provide a theoretical framework to support continuous queries than can contain multiple stream aggregates, correlated stream aggregates and can use data from multiple and heterogeneous stream sources. Also, we define SQL language extensions to support this kind of queries. The provided SQL extensions are minimal, succinct and easily understandable by common SQL users.
- We introduce a spreadsheet-like approach to perform stream analytics. The intuition is that continuous queries can be defined in a column-by-column fashion. This simple approach enables a number of optimizations for efficient stream query processing. A simple SQL-like language and a tool are provided for the definition and the computation of stream analytics.
- Stream analytics applications need to integrate and manage aggregates produced by a variety of stream engine tools, complete Data Stream Management Systems (DSMSs) or stand-alone stream-handling components. We present a relational-based integration framework that sits atop any relational DBMS and mix DBMS' data and stream

aggregates managed by different stream systems. A SQL extension is provided to define views that contain both relational and stream data. We developed an Application Protocol Interface (API) to carry out the required communication between the relational and the stream systems. The proposed framework can serve as a standard for relational-stream interoperability.

1.4 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 contains an overview on stream data, applications and systems. It also provides background information on stream query languages and stream processing. Related work on spreadsheets and data analysis is given. We outline existing work in analytics for relational and stream data.

Chapter 3 discusses stream variables, an approach for the formulation of continuous queries that can be used for analytics purposes. The theoretic framework is provided and SQL extensions to support multiple and correlated stream aggregates in queries are defined. Additionally a thorough number of examples are provided. Finally evaluation algorithms and performance results are given.

Chapter 4 presents a spreadsheet-like approach for stream queries that are useful for real-time decision making. A real case application is described and the challenges are identified. We present how these challenges are handled by our spreadsheet framework. We describe optimizations and performance results are provided. Finally we describe and preview the developed spreadsheet tool.

Chapter 5 describes a special type of view that can contain relational data and stream aggregates from multiple and heterogeneous stream engines. We present the theoretical foundations and a framework to enable relational-stream integration. An Application Protocol Interface (API) is defined to support the communication between RDBMSs and stream engines. An SQL-like language is presented for the definition of relational-stream integrated views. A relational evaluation operator is described for query evaluation. Finally a prototype system is described along with optimizations and performance results.

Chapter 6 summarizes the results of this work, discusses open issues and suggests areas for future work.

Chapter 2

Background and Related Work

2.1 Data Streams

2.1.1 Introduction

The proliferation of computer technology has brought big changes in data management. Data is produced in greater amounts and in most cases is inherently distributed. In many applications, the data is generated in real-time, in a continuous and transient manner a concept known as data streams [21]. To accommodate the enormous processing requirements of these applications, novel architectures have been proposed as conventional database management systems (DBMS) cannot handle real-time data processing [27]. For example Relational Database Management Systems (RDBMS) follow a "store-then-process" data processing model: data records are stored in disk and users can query the data using a query language (e.g. SQL). However, in case of data streams the database management systems have not the entire data set when a query is issued and common query processing techniques [74] cannot be applied. In addition continuous data must be processed on-the-fly and queries results must be provided in (near) real-time. These challenges [89] along with real world applications that require to process voluminous amounts of stream data have given an increased priority to the design and development of novel data stream engines.

Formally, a data stream can be described as a sequence of data items that produced continuously in real-time. The unique characteristics of data streams [71] are:

- Data streams are unbounded in size. Due to the large size it is not feasible to locally store a stream in its entirety.
- Applications that process stream data cannot control the order in which data items arrive.
- The data items are processed once or a small number of times due to: (a) the large volume (b) the processing time constraints and (c) the limited computation and storage capabilities of the processing system. In most cases the data stream items that are processed are discarded or archived making difficult to retrieve and process them multiple times.

- The processing of stream items are happen in-memory. The memory is small relative to the size of stream data and as a result there is an increased query processing cost for processing stream data items that are not in memory.

These unique characteristics of data stream create some challenging research questions such as:

- How data streams are modeled?
- How data streams can be queried?
- How an infinite data stream can be processed in bounded memory?

A partial list of research work trying to address these issues is given below:

Data stream modeling: Data streams can adhere to the relational model i.e. a data stream is a sequence of relational tuples [108]. We can classify such streams either as transactional data streams or measurement data streams. Transactional data streams contain tuples that log interactions between entities while measurement streams contain tuples coming from the monitoring of the evolution of a phenomenon. In most cases a phenomenon can have a number of entity states that may change during time. Examples of transactional streams are credit card transactions and web logs. In both cases a customer or a web user interacts with a system leaving transactional trails. Measurements streams contain tuples coming from the monitoring of a network (e.g. tuples containing IP packets fields) or sensor observations (e.g. temperature observations). More formally a data stream can be modeled as sequence of tuples with schema (v, t) where v is a single value or a vector of values and t is the timestamp that defines the order of the sequence [79]. The timestamp can be attached to the data stream item when it is first created (in stream source) or assigned from the stream system when it arrives. In [109] a data stream is modeled as a n dimension vector $\vec{a} = (a_1, ..., a_n)$ initialized to 0 and updates presented to it in a stream. In the *Cash Register Model* each update has the form (a_i, I), so that a_i is incremented by some positive integer I. In the *Turnstile Model* each update is in the form (a_i, I), so that a_i is incremented by some (possible negative) integer I.

Querying data streams: Queries over data streams differ from traditional database queries. Traditional database queries are one-time queries i.e. issued once and applied over static data exist in the disk. Such queries compose a query plan consists of operators. Query plan evaluation returns a complete result set. On the other hand, queries over data streams are "continuous": the answer to a continuous query is produced over time, reflecting the stream data seen so far [140]. As new data stream items arrive the continuous query updates the results on-the-fly. So the query output is not a

static and finite result but can be seen as a new data stream. The differences between one-time and continuous queries are depicted in Figure 2.1.

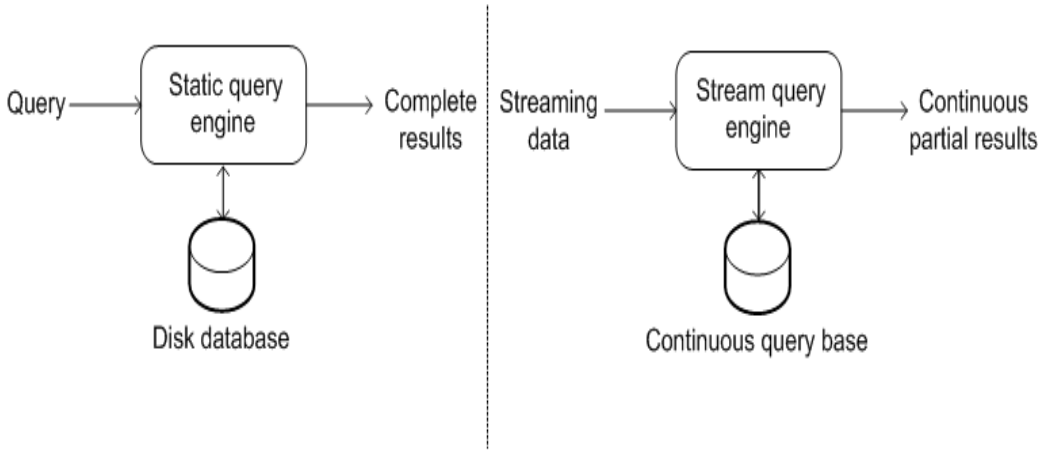


Figure 2.1: One-time queries vs. continuous queries

Consequently continuous queries can be added and deleted in a stream system at run-time. In case of query addition the queries can be either predefined or ad-hoc. Predefined continuous queries are registered to stream system before any relevant data stream item has arrived. On the other hand, ad-hoc queries are register when a data stream has begun.

In traditional Relational Database Management Systems (RDBMSs) queries are converted in an ordered set of steps composed of relational operators (query plan). In such a query plan data enters at the leaves, tuples are processed from the intermediate operator nodes and the result is given at the root of the tree. The evaluation of a continuous query using this approach is not always possible because an operator may need the complete data set from a previous operator (e.g. group by). But a stream is infinite and has no end. Such blocking operators can be replaced with no-blocking operators when it is possible [71]. Another approach is the usage of special assertions (e.g. punctuations [138][139]) that provide information what data items can and cannot appear in the stream. Their semantics can define the partial results that can be output from the stream system. Also, for a large number of applications only a subset from the whole data stream might be important. For example a simple query is to find the average temperature in a room only for the last ten minutes. In this case we are only interest for recent data. Prior data are not important and can be discarded. Such constructs that can define the range of items over a data stream are named windows [67]. Windows can be classified according to the following criteria [71]:

- **Based on window endpoints:** A window contains stream items that are enclosed in a starting endpoint and ending endpoint. The direction of movement of these endpoints corresponds to the following types of windows:
  - **Landmark windows** have a variable size extending from a fixed point in the stream to the latest received tuple.
  - **Sliding windows** have fixed size and both ends of the window moving (slide) as new tuples appear in the stream.
  - **Fixed windows** have stable endpoints resulting in a data stream snapshot for the defining range between the two endpoints.
- **Count-based or time-based windows:** continuous queries can restrict the range of stream data to a window that contains the last N items or those items that have arrived in the last t time units. The former are called count-based or logical windows and the latter are called time-based or physical windows.
- **Window evaluation strategy:** aggregates or other computations over windows can be done in a batch mode or per each arrived data stream element. The appropriate evaluation strategy can be decided based on performance/accuracy requirements and the need for near or real-time results.

Queries for distributed stream processing studied in [1][152][51]. In such environments, remote stream nodes process stream data and push asynchronously results to the main stream engine or the stream nodes are part of a query execution plan. Several problems arise in these architectures as: operator placement, load sharing and resource aware query execution.

Some researchers study the usage of standard database engines for stream processing. In [140] authors study how DBMS features like transaction management and concurrency control can be used for stream processing. Similarly in [81] authors study stream processing performance by tuning a standard DBMS system using already available features (e.g. indexes, triggers). In [82] a DBMS engine is extended to provide stream querying functionality. Stream operators are developed as UDFs (User-Defined Functions) and queries process chunks of stream data. Authors in [93] extend a column oriented database system to support stream queries.

Memory bounded processing of streams: Due to the infinite size of data streams there are cases that the amount of storage required for an answer of a continuous query may be unbounded [21] (e.g. if a continuous query is a self-join). In [11] authors provide the theoretical foundations and an algorithm for determining if a conjunctive query with arithmetic comparisons can be evaluated using bounded memory. Several other techniques are proposed from various researchers to handle

the infinite size of streams: processing only a part of a stream (windows), approximate answers and query optimization.

Stream window constructs transform infinite data streams to finite sets allowing continuous queries to be evaluated in bounded memory. For example in most cases joining two infinite data streams requires unbounded memory while a window construct over the data streams enable memory bounded computation [72].

Another approach to alleviate the problem of unbounded memory is to use small space structures (e.g. synopsis) that provide a concise representation of stream seen so far at the expense of some accuracy (e.g. approximate answers) [70]. In [68] wavelet based approaches are presented to summarize aggregates over streams. Algorithms for the computation of approximate frequency counts of elements in a data stream are described in [107][55]. The proposed algorithms require a small main memory footprint. Such algorithms are useful in group-by queries that the user is interested in only those groups whose frequency exceeds a certain threshold. Authors in [63] study the problem of approximately answering possibly multi-join, aggregate SQL queries over continuous data streams with limited memory. They suggest randomizing techniques that compute small summaries of the streams that are able to provide approximate answers to aggregate queries with provable guarantees on the approximation error.

Also due to the long-standing nature of a continuous query the query plan might change dynamically to handle fluctuations in memory resources. In [15][103] an adaptive query processing operator called *eddy* is proposed that is able to reorder operators in a continuous query plan. Operators can be reorder on the fly based on runtime selectivity and query execution cost leading to better memory utilization. In [20] an operator scheduling strategy is proposed to minimize run-time memory usage for continuous queries involving selections, projections, joins with stored relations and sliding-windows joins.

2.1.2 Data Stream Management Systems

Traditional Database Management Systems (DBMSs) are designed to support applications that use static data stored in disk. Stream applications require on-the-fly processing of data (no disk storage) and (near) real-time results. A new type of systems named Data Stream Management Systems (DSMSs) is designed to support continuous queries over stream data. DSMSs have the following requirements [71]:

- Query plans must contain operators that can provide results even if the input is infinite. In other words they must support non-blocking operators. Also query plan must optimized continuously due to dynamic changes in system conditions (e.g. available resources) or changes in stream characteristics (e.g. bursts, unreliable data)

- Processing algorithms have limited or no access to data streams elements that have already processed by the system. In most cases old data are dropped and stream system can make only one pass over the data.
- Continuous queries must be supported and query results must be provided in (near) real-time. Results can be exact or approximate due to performance and storage constraints. Also query semantics must support order and time operators due to data stream sequential nature.

A generic DSMS architecture [71] is depicted in Figure 2.2.

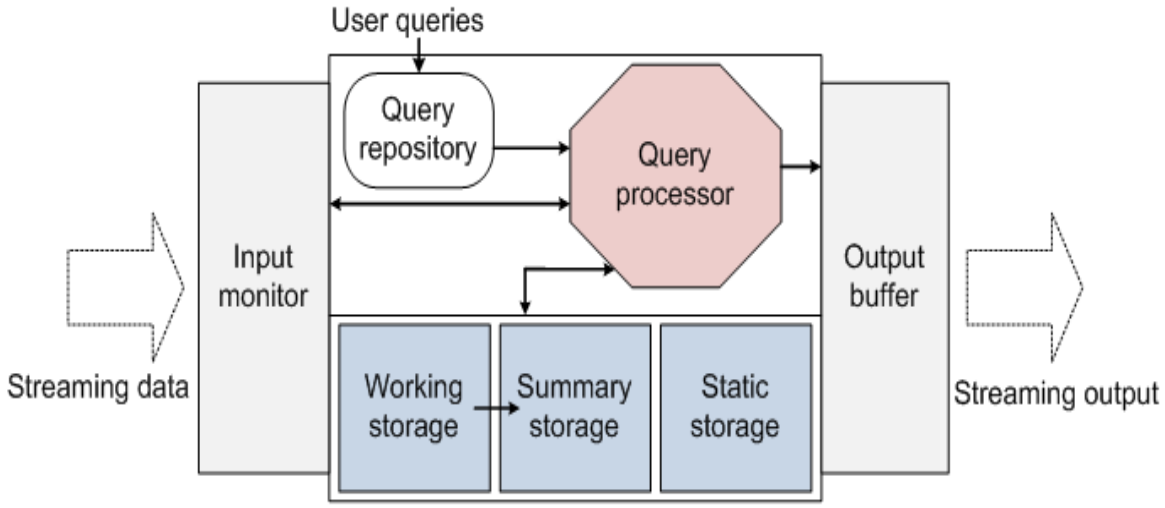


Figure 2.2: Generic architecture for a Data Stream Management System

Input monitor receives data streams and can apply load shedding techniques to regulate the input rate. Continuous queries declared by users are stored in the query repository. Queries that contain window structures keep the data in the working storage. Data synopses and other approximate-based structures are stored in summary storage. Static storage holds metadata for queries and data stream sources. The query processor processes streams and may re-optimize query plans base on stream rate, Quality of Service specifications and system resources. Query results are buffered or streamed out to the users.

Some early works on active databases present how Relational Database Management Systems (RDBMSs) can handle streams using triggers and Event-Condition-Act rules (ECA) [106][123]. However the large volume of stream data makes these approaches not suitable for stream applications as they cannot scale. For this reason a number of DSMSs have been developed either as research prototypes or as commercial ones. Most of these stream systems extend SQL [14] to support continuous queries or the queries are built as data flow graphs [83]. These graphs contain

stream sources management elements, stream processing operators and output components. Below we provide a partial list of DSMSs:

- **Tribeca:** In [131] authors describe an early version of a Data Stream Management System (DSMS) named Tribeca that is used for network traffic analysis. Tribeca provides a data flow language that can process stream data applying sequences of simple operators (aggregates, filtering).
- **CQ project:** The CQ project [96][95] proposes a distributed architecture that can monitor real-time updates in web pages and other sources (databases, files) using monitoring programs (CQ robots). When updates are detected in each distributed node the robots return results to the central system. Continuous queries in distributed sources are defined as a sequence of a SQL query, a trigger and a stop operation. When the trigger condition applies, the SQL query is executed. The stop condition terminates the monitoring operation.
- **NiagaraCQ:** The NiagaraCQ [49] system is similar to the CQ project but enables optimization for multiple monitoring continuous queries over distributed XML data sets. The proposed optimization approach it is based on the fact that many web monitoring queries share similar structures and conditions. In general NiagaraCQ enables shared computation and better memory utilization for multiple continuous queries applied over web sources.
- **Aurora:** The Aurora system [2][3] is a data flow system using a network of operators to process incoming streams. Aurora supports Quality of Service (QoS) specifications that control how resources are allocated for each query based on response times, on the percentage of tuples delivered and on the importance of values produced (some values are more important than others). Aurora can shed load data to satisfy QoS specifications by dynamically inserting and removing drop operators into query plans [133].
- **Gigascope**: is a special purpose stream engine used for network applications (traffic analysis, network monitoring, etc) [57][58]. It supports two kinds of queries: the low-level queries (LFTAs) which monitor network interfaces (i.e. the data stream sources) and higher-level queries (HFTAs) that act on LFTAs results. Gigascope is a stream database i.e. it consumes streams and produces streams.
- **PSoup**: supports queries that need data that arrived prior and after the query specification [32]. Data and queries are stored in special purpose data structures called State Modules (SteMs). There is one SteM for all continuous queries defined in the system and one SteM for each data stream. When a new stream data item is inserted in the data SteM it probes the query SteM to evaluate all the registered queries.

Symmetrically when a new query is inserted in the query SteM it applied to the data in the data SteM. These characteristics make PSoup appropriate for applications that periodically connect to the internet and not need to get stream results continually. For example mobile users may be offline for extended time duration but when they connected back to the network they want to be informed about query results. Also users might want to be informed periodically and not continually. This enables users to avoid information overload and the network can have better bandwidth utilization. PSoup achieves these functionalities supporting pre-computation and materialization of stream results.

- **STREAM:** is a general-purpose DSMS supporting continuous queries over multiple data streams and stored relations [12]. STREAM is based on relational semantics supporting windows, relation-to-stream and relational operators. For each continuous query an execution plan it is generated. A query plan is composed of: (a) operators, (b) operators' queues to keep stream data, (c) synopses containing operator state information and statistics. Plan sharing and approximation techniques are used for query optimization [108]. Moreover optimization algorithms for reducing operators' queue sizes are proposed to reduce query memory overhead.
- **TelegraphCQ:** extends PostgreSQL DBMS to support continuous queries over high volume and high variable data streams [31][103][86]. Streams can be created using Data Definition Language statements (e.g. CREATE STREAM) and continuous queries are SQL statements with an optional window clause. TelegraphCQ focuses on shared and adaptive processing of continuous queries.
- **Continuous Adaptive Query Processing Engine (CAPE):** [121] is stream engine designed to handle streams of varying rates providing an adaptive optimization framework with the following unique characteristics: (a) online query optimization with adaptive operator scheduling that can change operator scheduling algorithm dynamically based on system resources (b) plan distribution among multiple machines (c) punctuations on streams [139].
- **SPADE:** is a large-scale, distributed data stream processing system [66]. It provides a rapid application front-end and an intermediate language for composition of parallel and distributed data-flow graphs. It supports a large number of built-in stream-relational operators (e.g. windows). Also users can build their own user-defined operators which can integrate with the built-in operators. A data-flow graph is consists of processing elements containing connected operators. Finally a broad range of stream adapters is provided to enable connectivity from stream sources and publish data to external repositories.

Moreover, specialized stream handling components have been developed to support stream processing: MapReduce online [54] adds pipeline functionality between Map and Reduce operators [59][60] for stream data processing; [87] describes how a Map-Reduce operator can be used in stream queries that are defined as data flow graphs; [98] studies stream processing in a cloud architecture; [112] depicts a stream manager that supports disk-based incremental processing. Finally a large number of stream applications are built from scratch using general purpose programming languages. Stream programming libraries [125][136] can also be used for the development of custom stream applications.

2.1.3 Stream Query Languages

Continuous queries are defined in Data Stream Management Systems via a stream query definition language. A number of stream query languages have been proposed and developed from database researchers. The querying paradigms can be distinguished on relational-based languages, object-based languages and procedural languages [71]. Relational based languages model streams and windows as relations ordered by timestamp. Object-query languages assume that each stream element is an object that can be manipulated with object-oriented methods or processed inside a class hierarchy. Procedural languages define exactly how streams are processed either by a workflow of operators or by specialized commands applied over stream data elements in a sequential manner. A partial list of stream query languages is provided below:

- **Tapestry Query Language (TPL):** is a SQL-like language proposed in [134]. TPL supports continuous queries for append-only databases. In append-only databases only the new added records are of interest while the old ones are never deleted. Consequently, users issue continuous queries and notified whenever new incoming data matches the query. TPL enforces that deterministic results will be provided to users applying the following query semantics: *"the results of a continuous query is the set of data that would be returned if the query were executed at every instant in time"*. This introduce the notion of monotonic query: if $Q(t)$ is the set of records returned by query Q over a database at time t (one-time query) then the monotone query $Q_M(t)$ denotes the set of all records returned by executing Q up until time t. Monotonicity of a continuous query implies that any tuple that appears in the answer at any point continues to do so forever. Monotonic queries can be transformed to incremental periodic SQL queries implementing continuous query semantics. TPL transforms each user query into an incremental query that is run periodically. This is similar to execute the user query after every update of the database. Tapestry system use TPL for filtering mail and news messages.

- **Hancock**: is a domain-specific language that defines efficient signature programs [56]. A signature provides a compact view of the evolving behavior of an entity. For example in a telecommunications application a signature might contain a measurement showing the five most frequent telephone numbers placed from a specific number. Hancock query processing is based on event detection and event response over data stream items (e.g. when a new phone number is detected system re-initializes a counter as a response).
- **Continuous Query Language (CQL):** The STREAM system [12][108] implements the Continuous Query Language (CQL) [14] that derives from SQL:1999. CQL can support in the "from" clause streams, relations or both. A stream is considered as a multiset of relational tuples arriving at time T and consider append-only. Relations are an unordered set of tuples supporting time-stamped insertions, deletions and updates. Also derived streams (streams that are the result of sub-queries) can be handled efficiently. Sliding windows can be applied over streams and relations can be mapped to streams with specialized operators (Istream, Dstream). CQL processing emphasizes on memory usage optimization and operator scheduling.
- **ATLAS**: is a database language and system that allows users to develop data mining and data stream applications in SQL [141]. ATLAS SQL supports User Defined Aggregates (UDAs) that can contain an *initialize*, an *iterate* and a *terminate* computation statement. These statements are defined in a single procedure written in SQL. ATLAS SQL is Turing-complete and can easily support the definition of standard aggregates (e.g. avg, sum), online averages and stream window aggregates. For stream applications the terminate computation is replaced with the *revise* computation. The revise step is take place when the window is full. In that case the tuples contained in the window expire and the expired tuples participate in the computation of the wanted aggregate.
- **Gigascope query language (GSQL)**: is a restricted SQL-like language consuming streams and producing streams [57]. Its declarative nature allows query composition and query optimization that is similar to SQL. GSQL query model is based on the ordered attributes of the input stream. In most network applications there is a timestamp attribute or a sequence number per stream data element. GSQL uses the ordering characteristic (e.g. always increasing, no repeating, etc) of these fields to execute a query. For example a group-by query that groups on an ordered attribute can emit results when a tuple arrives with an ordered attributed that is larger than any current group. This simple evaluation scheme result to increased performance during

query execution. The GSQL supports selection, join between streams, aggregation and stream merge (union streams from multiple sources).

2.1.4 Data Stream Applications

A number of applications need to process stream data continuously and provide (near) real-time results. Some well-known application areas that validate the importance of stream processing are:

- **Network monitoring:** these applications process rapid and continuous data streams as packet traces and error signals. Typical scenarios that such applications are used are protocol performance analysis, network traffic analysis, detection of anomalies (link congestion), intrusion detection, billing, etc. For example a useful query is to monitor the load of a backbone link over 5-minute periods and inform a network operator if the load exceeds a threshold [22]. Also analytics and trend analysis over network data is important (e.g. find the total number of incoming calls for a region and compare with the total number of incoming calls for another region in the past three hours). Another application is to use continuous queries for load balancing and for the redirection of traffic to another router or server. While there are specialized tools for network monitoring in most cases are inflexible [58]. On the other hand modeling network flows as data streams leads to a unified database-oriented approach for processing network data. As a result, a database stream-oriented network management approach can provide a structured query environment for network data applications making complex network analysis an easy task.
- **Web site monitoring:** the always increasing growth of World Wide Web (WWW) creates a data overload problem for users as pages change constantly and dynamically. Users want to monitor changes in web pages while avoiding visit pages multiple times to find the information they want. Continuous queries over web pages and XML sources (e.g. RSS) can provide the mentioned functionality to users [96][95][49].
- **Road traffic management:** real-time traffic analysis is of great importance for efficient street and car usage utilization. Sensors can be embedded on highways and GPS-enabled cars can provide in real-time the position of cars. Stream engines process these data and provide useful information to drivers as car flow and volume statistics [103]. The benefits from such applications can be the reduction of traffic, the reduction of carbon dioxide emissions and the lower traveling times through hints for alternate driving routes. Other examples are variable tolling, speed estimates, real-time accident detection and notification.
- **Healthcare applications:** live patient data (e.g. blood pressure, pulse rate) is provided by smart sensors by monitoring patients either in their home or in a hospital room [23].

In this way doctors and healthcare professionals can react in real-time in critical incidents. The monitoring of patients can be achieved with a network of smart sensors and a network-enabled infrastructure. Reliability is a crucial issue for healthcare monitoring applications i.e. the failures must be handled efficiently. This can be done with backup sensors and alternative communications channels (e.g. cable, wireless, mobile). As a last resort in case of a sever failure there must be an alarm informing about this situation. The combination of live patient data with historical data (e.g. patient record data) can be used for information correlation enabling better medical diagnosis for patients.

- **Radio Frequency Identification (RFID):** is a key technology with a wide number of applications including supply chain (e.g. inventory monitoring) and asset monitoring. RFID tags enable unique product identification and can be used in any object such as pallets, cases or individual items. RFID readers scan tagged products and generate a stream of data consist of the unique product identification and the capture-time timestamp. The high volume of the scanned data poses several challenges requiring in most cases to process the generated data with specialized stream engines [145].
- **Environmental monitoring:** sensors can be deployed in wide geographic areas to monitor physical phenomena [7][104][61][102]. Example applications are temperature monitoring, monitor volcanic activity, water quality monitoring, animal monitoring etc. Also outlier detection over sensor measurements provided by distributed sensor nodes can help on detection of chemical spills or other disasters.
- **Energy management:** the utilization of energy lines and the reduction of consumption are of great importance for large factories in order to decrease costs per produced unit. Energy monitoring via smart energy meters can provide alerts for increased consumption in real-time resulting in economic benefits. Also real-time monitoring can detect power failures or power spike problems in large energy corporations enhancing the safety standards. Moreover, retail customers can analyze in real-time their consumption and compare it with the consumption of other customers or with older consumption rates statistics. Such analysis can help users to identify better energy consumption habits. In a more advanced scenario the energy management system can provide hints for energy conservation.
- **Financial applications:** financial transactions, stock ticks, currency exchange transactions can be seen as a stream of data. A large number of real-time decision queries over stocks are described in [45] and [91]. Such queries can provide information to financial analysts to sell or buy a stock or plan their trading actions. Algorithms for the identification of correlations between pairs of stocks, a useful

technique for stock trading, are described in [156]. Similarly, querying stream data, news and historical companies' data can be useful for trends identification and for the detection of sell/buy opportunities.

- **Ambient devices:** can change their characteristics based on real-time data in order to provide access to information at a glance. For example these devices translate data information into color, motion or sound representations (output) which are easily captured by human sensory modalities. In most cases ambient devices use simple conditions to monitor changes. However complex condition can be supported. For example in a weather application a monitor query can calculate the average temperature per day (drill up per day) and change the color of the ambient device if today's temperature is greater than a previous day. Also, monitoring of composite measures can be achieved using multiple different data sources (i.e. in the weather application we can change the device color based on the temperature and the speed of the wind).
- **Click-stream analysis:** web site personalization and advertisements campaigns need to process users click-streams in real-time to identify and predict users' preferences. For example a useful stream query can be: "Which products advertised in a web page are currently the most popular?" [19]. Having such insights in real-time an analyst can change the advertisement strategy dynamically for products with few sales. Another example is the prediction of the next web page request for a user using previous click-streams and the time spend on each page [69]. As a result the web site can provide a next web page recommendation list while the user is surfing the page.
- **Sensor networks:** consists of small sensors and actuators that can "sense" the real world and provide real-time measurements. Some of the aforementioned applications (e.g. environmental monitoring, healthcare applications etc) use sensors. Sensors characterized by limited communication bandwidth, limited energy supply, limited computation power and uncertainly in sensor readings due to the environmental noise. Most research in sensor networks tries to alleviate these limitations. In [149] a database oriented approach is used: a sensor network is viewed as a distributed database where sensor nodes hold part of data. In this way users can declare declarative queries over a sensor network while the processing of data can be done inside the network resulting to minimized communication costs. Authors of [100][101] describe a complete system architecture for sensor data management. Their focus is on managing multiple queries over many sensors by limiting sensor resource demands while maintaining high query throughput. Except from small sensors with limited processing power there exist powerful sensing devices (e.g. webcams, microphones). Such devices create a

distributed network providing voluminous streams and can be used for a number of useful applications [62]. For example a parking can be monitored by a webcam and continuous queries over the provided stream can be used for a parking space finder service.

2.2 Data Analytics

Data analytics refers to combination of methods and techniques for the analysis of large amount of data with the purpose of gain better insights and facilitate decision making across a wide range of applications domains [94]. Until recently analytics methods and techniques applied over offline data [107]. However the need to shorten the time between data acquisition and decision making give birth to stream data analytics [48]. For offline data sets, data collection and extraction technologies (e.g. data warehouses, Extract-Transform-Load tools) are support large scale data analytics. The main goal of data analytics is to apply data analysis over these collections. Data analysis is performed via aggregation queries, analytical queries (OLAP) and reporting tools that can visualize the important data characteristics. In most cases these tools provide next-day analytics i.e. data are collected incrementally and analysis is performed over historical data. Another aspect of data analysis is data mining used for pattern discovery and predictions (predictive analytics). In stream analytics the data analyzed online as they arrive and there is no requirement to load and store the data on large data warehouses.

2.2.1 Offline Data Analytics

The need of complex data analysis involving aggregation of data became apparent since the conception of Database Management Systems (DBMSs). While the *group by* clause was sufficient at the beginning, the dawn of new applications in the last ten years, such as web analysis, social networks and others, necessitated advanced grouping constructs (cubes, grouping variables, windows) and novel programming paradigms such as MapReduce [59].

Grouping was the first approach in database theory to support data analysis: the relation is partitioned based on one or more attributes and column-based aggregates are computed over each partition. Modeled as a relational operator (e.g. [64]), with multiple implementation algorithms (e.g. [74]), incorporated in query optimization (e.g. [46],[150]) and with a simple SQL syntax, it became an essential part of any DBMS. For complex analytics, users have to rely on multiple view definitions or nested queries. Usually, most commercial systems' performance break in queries representing trends, correlations or hierarchical aggregation.

With the rise of data warehousing and OLAP [47] came the need of multi-dimensional analysis, i.e. aggregations over multiple combinations of group by attributes. While traditional group by could be used to express and evaluate multidimensional analysis, there were significant linguistic and implementation benefits in introducing a new grouping construct, called cube [75], which computes an aggregate over all possible subsets of an attribute set. The cube by clause, an SQL syntactic sugaring extension, made it easier for users and allowed the optimizer to use efficient evaluation algorithms [6], [118] to compute the cube – mainly by overlapping computation. While a major breakthrough, it lacked the aspect of separating the base values definition and the subset formation process as two distinct phases. For example, one may want to provide an ad-hoc set of group by attribute combinations and not the entire powerset, or compute multiple aggregations constrained by different conditions for the same group by attributes (e.g. [119])

A *set variable* is a variable containing rows of a table, i.e. denotes a subset of the table. It is usually the result of a selection operation. It is frequent in data analysis to define a set variable for each distinct value of a column (or combination of columns) and then compute some aggregated value. For example assuming a relation named `Stocks` that contains the opening and closing prices per day for each stock:

```
Stocks(stockID,descriptio,openingPrice,closingPrice,date)
```

A useful query is: "for each stock, get the opening prices of that stock in January". For each distinct value *s* of column `stockID` we should define a set variable as the rows of table `Stocks` having *stockID = s and month(date) = 1*. A grouping variable, introduced in [42] and described in [34][35][85][36], depicts this idea. A grouping variable is attached to a group by clause and for each distinct value of the grouping attributes a new set variable is instantiated. The definition of the grouping variable is given with the newly introduced clause "such that". The previous example could be expressed as:

```
group by stockID; X
such that X.stockID=stockID and month(date)=1
```

The syntactic extensions are:

- **Group By clause.** The group by clause is the same as in standard SQL, with the following addition: after specifying the grouping columns, it may contain a list of grouping variables. For example, we may write:

group by stockID; X_1, X_2, …, X_n

- **Such that clause.** This newly introduced clause contains one defining condition for each grouping variable, separated by commas. Each defining condition is similar to a where clause. For example, we may write:

  such that C_1, C_2, …, C_n

  Each C_i is a (potentially complex) condition used to define X_i grouping variable, $i = 1, 2, …, n$. It may involve (i) attributes of X_i, (ii) constants, (iii) grouping columns, (iv) aggregates of the group and (v) aggregates of the $X_1, …, X_{i-1}$ grouping variables. Part (v) implies that aggregates of grouping variables appearing earlier in the list can be used to define grouping variables later in the list.
- **Select clause.** The select clause is the same as in standard SQL, with the following addition: attributes and aggregates of the grouping variables can also appear in the select clause.
- **Having clause.** The having clause is extended to contain aggregates of the grouping variables.

With these syntactic extensions, the `group by` clause acts as an implicit iterator over the values of the grouping attributes. The group itself can be considered as another grouping variable, denoted as X_0. Aggregates of the group are considered as aggregates of the X_0 grouping variable. The standard SQL formulation is cumbersome to express in SQL, requiring repeated joins, group-bys and views. The following example describes a representative ad hoc data analysis/decision support query on a `Stocks` table (a pivoting example).

Example: Assume that we want to find for each stock of 2005 the average opening price in January and February (in two columns, one next to the other), but only if the latter is greater than the former. The having clause can be used to select the appropriate groups.

```
select stockID, avg(X.openingPrice), avg(Y.openingPrice)
from Stocks
where year = 2005
group by stockID; X,Y
```

```
such that X.stockID = stockID and month(X.date) = 1,
          Y.stockID = stockID and month(Y.date) = 2
having avg(X.openingPrice) < avg(Y.openingPrice)
```

In this example, for each stock `stockID`, `X` grouping variable contains the rows of table `Stocks` that agree on `stockID` and have month equals to 1 (i.e. the prices of stock `stockID` in January) and `Y` grouping variable contains the sales of stock `stockID` in February. For each stock, we just want to print out the average of the opening price of `X` and `Y` subsets, if the latter is greater than the former.

A grouping variable X expresses the following idea: for each distinct value v in grouping attribute(s) C of relation R, define a subset X_v using a condition θ and compute one or more aggregated values over X_v. Then attach these aggregated values next to v to formulate the output row of the resulting table. This is expressed in relational algebra via the MD-Join operator [43][8][9], which generalizes the conventional notion of group-by: it distinguishes between the definition of the "base values" used to aggregate-by and the actual computation of aggregates of these. Grouping variables represent the later. Formally the MD-Join has the following definition:

MD-Join: Let B and R be relations, θ a condition involving attributes of B and R and l a list of aggregate functions $(f_1, f_2, \ldots, f_n)$ over attributes $c_1, c_2, \ldots, c_n$ of R. A new relational operator between B and R, called the MD-join is denoted as:

$$MD(B, R, l, \theta)$$

with the following semantics:

- table B is augmented with as many columns as the number of aggregate functions in l. Each column is named as $f_i\_R\_c_i$, $i = 1, \ldots, n$ (e.g. *avg_Sales_sale*). If a duplicate name is generated, the table R must be renamed.
- for each row r of table B we find the set S of tuples in R that satisfy θ with respect to r, i.e. when B's attributes in θ are replaced by the corresponding r's values. Then, the value of column $f_i\_R\_c_i$ of row r is the $f_i(c_i)$ computed over tuples of S, $i = 1, \ldots, n$. B is called the base-values relation (or table) and R is called the detail relation (or table).

B represents the group-by structure of an EMF (Extended Multi-Feature) SQL query [34]; condition θ corresponds to the defining condition of the grouping variable in the `such that`

clause; the list of aggregate functions *l* corresponds to the grouping variable's aggregates mentioned in the `select`, `having` or `such that` clauses. The definition of the MD-join operator allows the user a tremendous amount of flexibility in defining an aggregation query, as *B* and *R* can be arbitrary relational expressions and θ can be an arbitrary join predicate. The row count of the result of the MD-join is the same as the row count of *B* (i.e., the MD-join performs an outer join) and as a result this semantics captures more accurately the user's intentions than the standard aggregation does. In addition, this property is valuable for efficient implementation and optimization. Note also that the MD-join operator can be considered as a shortcut for a somewhat more complex expression. However, the expression that the MD-join represents occurs very often in OLAP queries and the properties of the operator enable to easily obtain many query transformations leading to efficient execution plans.

The associated set (ASSET) query concept is described in [44] and [38]. It is applicable in both continuous and traditional data settings. The idea of ASSET queries is: "Given a set of values *B*, an associated set over *B* is just a collection of annotated data multisets, one for each value of *B*". The goal is to efficiently compute aggregates over these data sets. An ASSET query consists of repeated definitions of associated sets and aggregates of these, possibly correlated. The ability to loop over the values of a domain and perform a task for each value is the main construct in programming languages and its presence leads to very strong theoretical results. Formally an associated set is simply a set of potential subsets of a data source *S*, one for each value *b* of a domain *B*, i.e. $\{S_b : b \in B\}$. An associated set instance (just called associated set) is a set of actual subsets of *S*. *B* is usually a relation (the base relation), the data source *S* can be anything with a relational interface and an iterator defined over it, and is a defining condition that constraints (creates) the associated set instances. This simple approach: (a) generalizes most grouping analytics in existence today, (b) separates the relational concept from the analysis (grouping) concept, (c) can lead to rich optimization frameworks, and (d) provides a formal (and semi-declarative) base for MapReduce [59]. For example, given a relation *B* of all 2009's sales, the associated set (instance) $\{S_b=\{s \text{ in Sales, such that Sales.date} <= b \text{ and Sales.year} = 2009\}, b \in B\}$ could be used to compute the daily cumulative sales of 2009. An associated set instance is just a collection of multisets. Although aggregation is a separate process, significant optimization can take place for the built-in aggregate functions. An ASSET query consists of the computation of one or more associated sets, recursively defined: starting from a base table B_0, associated set $(i+1)$ uses as its base table the base table of associated set *i* extended by its aggregates. This approach has as result that a significant class of data analysis queries can be easily represented and efficiently evaluated through this formalism. ASSET queries can be useful in:

- **Incorporating heterogeneous data sources:** the data source of an associated set can be anything with a relational iterator defined over it – different database vendors, flat files, even the output of the query defined so far.
- **Distributed OLAP computation:** if the data source of an associated set is distributed to more than one nodes, the subset formation process can be easily distributed to these nodes – that does not mean that the associated set is materialized.
- **Performance:** as no traditional relational optimization can be applied over the data sources, he optimization process (indexing, decorrelation, specialized join algorithms, distributed computation) is shifted/replicated to the data structure representing the ASSET query answer – which can always be made memory resident.

SQL/OLAP Amendment introduced certain new features in SQL language to support on-line analytical processing. One of the significant extensions is the ability to define windows over rows. A window enables users to determine the set of rows over which calculations can be performed with respect to the current selected row. In detail, in a window clause declaration we can define: the attribute list used for partitioning, the ordering of rows within partitions and an aggregation group. The aggregation group specifies which rows of each partition, with respect to the current row under examination, should participate in the evaluation of declared aggregate functions. This construct enable advanced data analysis over data.

Spreadsheet is a well know paradigm to analyze data and spreadsheet applications are accessible and used by millions of users. It is the de-facto application for day to day computations and has been used in many different domains. A spreadsheet consists of a grid made from columns and rows. The intersection of columns and rows represents cells. In most spreadsheet applications rows are represented by a number and columns by a letter. In this way each cell can be referenced by its column letter and its row number. For example C7 denotes the cell in the third column and in seventh line. The start cell of a spreadsheet is the A1. Additionally we can refer to more than one cell (e.g. C7, C8 using comma as a separator) and in a range of cells listing the first and last cell in the range (e.g. E1:E10 using colon as a separator). Each cell can contain data values (i.e. numbers, text) or formulas. Formulas express calculations using other cells, formulas and constants. For example we can express arithmetic operations between cells (e.g. =E1+E2 where the equal sign defines a formula) and use specialized formulas (called functions) to perform calculations over data (e.g. min(E1:E10) to find the minimum value in a range). Formulas can depend on other formulas and cells. These dependencies create a dependency graph guiding the ordering of computations. Also spreadsheet applications support many spreadsheets (called worksheets) inside one workbook to break our data analysis task in small and concrete subparts. During the years spreadsheets grew from simple applications to complex analytics tools offering

advance database and OLAP capabilities. In [143] and [144] authors propose spreadsheet-like computations in Relational Database Management Systems through extensions to SQL. By this approach relations can be viewed as n-dimensional arrays and formulas can be defined over their cells. The SQL extensions can be used for array based calculations for complex data analysis. In [97] a spreadsheet-like algebra that is consists of a set of operators that can express simple SQL queries and can be intuitively implement visually in a spreadsheet is proposed. Query definition is a sequence of progressive steps. In each step the intermediate results are provided helping user reformulating and refining the query. Such direct manipulation interface where queries can be defined with clicks and drags is more intuitive for non-technical database users and as result they can perform data analysis tasks in a simple manner.

Also large-scale data analytics platforms have recently been introduced and becoming widely applicable to the real world. MapReduce is a well-known programming paradigm to perform large-scale data analysis [59][60]. It is consisting of two phases, modeled as functions: the mapping phase, where a set of values is derived, each associated with a list of values, and the reduce phase, where each list is reduced by some ad hoc aggregation method. It can express and evaluate in a natively parallel and fault tolerant way simple analytics (similar to group-bys in SQL-based systems). While this approach offers significant procedural flexibility over declarative approaches and employs a simple computational model, it lacks the optimizability and ease of use of modern database systems [114]. In [111][137][5] authors propose the addition of declarative interfaces on top of MapReduce implementations. A comparison of parallel DBMS and MapReduce-based platforms for data analytics is given in [127].

2.2.2 Stream Data Analytics

Real-time decision support can be achieved by continuous analytics queries [65]. In [124] authors provide an architecture called decision-centric information monitoring (DCIM) that enable users to monitor information that can change a decision. Relevant information for a decision is identified via sensitivity analysis of decision models on distributed and heterogeneous databases.

The window construct defined in the SQL/OLAP Amendment is applicable in data stream processing [21]. Due to the infinite nature of data streams in most cases analysts are interested only in recent data while older data are less significant [52]. Windows can limit the unbounded size of a data stream by defining time or count-based conditions. Stream data flow in and out of the defined window and aggregates computed continually over window transient data. Moreover, correlating aggregates over windows existing in multiple streams is important for real-time data analysis [155][77].

Stream aggregates can be defined with User-Defined Aggregates (UDAs) as described in [89] and [99]. Correlated aggregates define dependencies over aggregates (depended/independent aggregates) requiring multiple pass over data for their evaluation [42][34]. As this is not feasible for data streams efficient approximate computation of correlated aggregates over data streams is studied in [67]. Authors provide one-pass algorithms for computation of correlated aggregates over landmark and sliding windows. Online aggregation [80] is another approach to support continuous analytics: traditional data sets are consider as infinite and early query results are provided as a running aggregate with associated error bounds. Temporal aggregates [153] over data streams maintain aggregates at multiple levels of temporal granularities i.e. recent data is aggregated with finer detail while older aggregated at a coarser time granularity. Aggregating at different granularities resembles the roll-up operation in traditional analytics but in case of streams the roll-up task is happen automatically and on-the-fly [154]. In [157] authors study the problem of finding abnormal aggregates over windows with different time intervals which can be used for outlier detection analysis. A query evaluation framework for hierarchical aggregates is proposed in [37]. Such aggregates can be used in applications where stream sessions can be organized in hierarchical fashion e.g. sessions may contain sub-sessions. A session is modeled as an object allowing rich querying capabilities over streams for this kind of applications. Finally, Complex Event Processing engines [147] allow event pattern detection over streams of data.

In [151] authors identify the problem that OLAP-like queries that provide real-time multidimensional and summarized views of stream data are not well supported from current stream solutions. They introduce a multi-dimensional stream query language that can turn low-level data streams into high-level aggregates. Real-time OLAP-like analysis of streams is achieved with a cube algebra supporting continuous operators that can convert continuous streams into conventional cubes and the opposite. Moreover operators that resemble roll-up and drill-down operations exist on traditional data cubes are provided for stream data. The question how online analytical tasks can be applied over data streams is studied in [50]. The stream cube [78] is an architecture used for on-line, multidimensional and multilevel analysis of stream data. A tilted time frame is proposed where more recent data are registered at finer resolution, where older data at coarser resolution. Such an approach is useful for stream analytics as in most stream applications the most recent data are more important. A traditional cube contains cuboids describing a subset of its dimensions. In stream cube only a small number of dimensions are materialized based on two types of layer: (a) *observation layer* which is the layer that an analyst would like to check for data analysis and decision making tasks (b) *the minimal interesting layer* which is the minimal layer that an analyst would like to examine. Computing only aggregates corresponding to the two layers leads to performance benefits which is necessary due to the large volume and high speed of data streams.

A number for analytics solution for RFID data have been proposed from database researchers. FlowCube [73] is a method used to construct a warehouse of RFID trails for analytics purposes. RFID-enable devices (e.g. pallets, items) generate a large flow of data containing the full history of the locations that this item passed (e.g. from a production line in a factory to the store). Analysts would like to track this movement in the entire supply chain. FlowCube is an OLAP cube aggregate item flows at a given abstraction level. The difference is that a FlowCube does not contain scalar aggregates but flow-graphs representing the movement trends and deviations of the items aggregated in each cell. Also the item's flow paths can been seen at different level of abstraction enabling complex analysis over RFID data. SQL/OLAP functionality for handling anomalies in RFID reads is proposed in [117]. RFID readers can provide duplicate and missed reads to stream applications. To handle this problem, authors propose a declarative based language that uses SQL/OLAP functionality to support data cleansing methods for anomalies detection and removal. A number of architectures operate over traditional data warehouses to enable (near) real-time analytics. In [122] a data store is proposed that monitor workflow business operating over a data warehouse to enable real-time decision making. MeshJoin [115] is introduced in to support on-line warehouse refreshment for applications requiring up-to-date information. MeshJoin is used for joining a fast stream of source updates with a disk-based relation in a data warehouse under the constraint of limited power. Such operation transforms passive data warehouses to active in order to support real-time analytics.

The usage of spreadsheets for managing and processing sensor stream data is depicted in [146]. Authors provide an Excel based interface for sensor data management and programming. What is making spreadsheets widely acceptable is the simple interface that is well understood by the end users. In [40][41], we consider how spreadsheets can be used to model and express complex spreadsheet-like continuous queries over RFID data. We argue that the advantage of a spreadsheet-like query interface is the concise and intuitive representation of queries. This research is described on Chapter 4 of the current thesis.

Data integration has multiple applications and a wealth of techniques have been developed from database researchers. The main goal of data integration is to combine data residing at different sources and provide users with a unified view of the data. Our work in Chapter 5 is toward this goal and we focus specifically on relational and stream data. The challenges for DBMS and stream systems integration are presented in [132]. Authors emphasize that this is a new and challenging research area as current trend mostly focus on static data integration. Also the importance of integration of stream and stored data is analyzed in [128]. These authors discuss integration of stream and relational data from a stream system perspective i.e. how a stream system can use offline data. In [26] authors propose a descriptive model to analyze the execution behavior of heterogeneous stream processing engines. MaxStream [24] is a data integration system that can

use multiple stream engines and databases for real-time business intelligence applications. MaxStream queries are translated into the native language of each stream system and the architecture is similar to federation database systems [25] i.e. it consists from a middleware and a set of data wrappers interacting with each stream engine. A framework for situation aware applications that use stream and stored data is described in [28][29]. A data flow interface is proposed for building situational aware applications.

Chapter 3

SQL Extensions for Real-Time Analytics

3.1 Introduction

The ability to query data streams is of increasing importance and has been identified as a crucial element for modern organizations and agencies. In this chapter a class of useful and practical analytical continuous queries is examined. Analytical continuous queries are used for decision making in (near) real-time. We demonstrate that such queries can be concisely modeled by a simple relational approach coupled with a simple SQL extension. We introduce the notion of a stream variable, a conceptual entity representing an ordered subset of a data stream, aggregated and attached next to a standard relational schema as a new column. A logical expression that involves the relation's attributes and the entity's methods determines whether a stream data should be added to the stream variable. The ability to define in the same query multiple, consecutive, possibly correlated stream variables allows for great flexibility in expressing complex analytical continuous queries. Moreover, such an approach presents several opportunities for efficient optimizations.

3.2 Rationale and Motivation

The technological explosion in the web, mobile communications, sensor/wireless technology, as well as the need for security, personalization, fraud detection, real-time billing, dynamic pricing, and others emphasize the necessity of real-time analysis and "stream" systems. We are moving toward real-time enterprises (RTE) and a stream world. Examples of stream applications include financial systems, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, environmental monitoring, ambient intelligent systems and others.

Queries over data streams are quite different than traditional ones. In data streams we usually have continuous queries [134][22]. The answer to a continuous query is produced over time, reflecting the stream data seen so far. The database research community has responded with an abundance of ideas, prototypes and architectures to address the new issues involved in data stream

processing [152][135][100][57]. However, expressing complex data analysis queries on top of data streams is a major challenge.

While many systems have been developed to address the various challenges present in stream applications [27], few deal with simple SQL extensions that can be used for analytic tasks over data streams. For this purpose we formally define a stream variable: a collection of data structures (representing queues), each functionally dependent on a subset of a relation's *R* attributes, continuously reporting one or more aggregate values. These values are "attached" as separate columns to *R*. A simple SQL extension is used to express stream variables corresponding to a straightforward execution plan for query evaluation. By defining a series of stream variables one can express complex analytics queries over one or more data streams.

3.2.1 Motivating Examples

We use a financial application as a motivating example. There are two relations, `Stocks` and `Categories`, storing the opening and closing prices per day for each stock and category, and two data streams, `Prices`, `Volumes` reporting several times within the unit of time the current price of a stock and the volume of the stock executed from the previous reported value. Schemas of relations and data streams are presented below:

Relations:

```
Stocks(stockID, description, categoryID, openingPrice,
closingPrice, date)
Category(categoryID, description, openingPrice, closingPrice,
date)
```

Data streams:

```
Prices(stockID, price, timestamp)
Volumes(stockID, volume, timestamp)
```

There are several interesting continuous queries one can register on top of `Prices` and `Volumes` streams to monitor stock activity for analytics purposes. Below we provide some examples:

Q1. Assume that we want to monitor for each stock the minimum, maximum and average price that has been seen so far. With this query we can detect severe fluctuations of a stock's performance at real time.

Q2. Sometimes it is useful to monitor the minimum, maximum and average price of a stock's performance not from the beginning of the day but only within a specified moving window, e.g. for the last 100 reported prices.

Q3. Being able to express continuous values at different granularities and compare these is an important aspect of financial applications. For example, we may want to find for each stock the percentage variation between the running average reported price and yesterday's closing price and compare it with the percentage variation between the running average reported value and yesterday's closing price of the stock's category. With such a query, we may find buy or sell opportunities on a category basis.

Q4. It is also crucial to integrate in a simple and succinct way values from different stream sources in a single query. For example, we may want to combine in a continuous report aggregated values from both `Prices` and `Volumes` streams: find for each stock the total volume of the last 10 reported volumes, the maximum price of the last 10 reported prices and contrast these with the total volume and maximum price from the beginning of the day.

Q5. In many occasions it is useful to express correlated aggregation [42][34][67] in the context of data streams, i.e. use a continuously aggregated value to constraint a subset of stream data. For example, we may be interested in monitoring the running total volume of each stock, but summation should take place only when the average price of the last 10 reported prices is greater than the running average price of the stock. Then, we want to contrast this with the (regular) running total volume. This query can show periods of time of increased volume traffic.

Q6. Assume that we want for each stock to continuously know when its average price of the last 10 reported prices is greater than its running average price. In that case, a "True" value should appear next to the stock id, otherwise a "False" is displayed. This query can be used to alert analysts for "hot" periods of a stock.

Q7. Finally, in many cases we want to monitor a stream of data and treat the generated values as a new data stream (composability). For example, assume that we monitor for each stock the average price of the last 10 reported prices and we want to also monitor the maximum of these averages. In this case we can identify when (within a window of 10 values) the maximum average price occurred.

Figure 3.1 (a) to (g) shows instances of the results of queries Q1 to Q7 respectively. Let us first consider query Q1. In this example, we want to keep for each stock s the reported prices for s (since the registration of the query with the system) and compute the min, max and average price of this data set (the running min, max and average prices). In other words, for each `stockID` we want to define a data set, modeled as a queue, $Q_{stockID}$, and compute several aggregates over this

queue. Each aggregate can be "appended" next to the `stockID` column of the answer (Figure 3.1(a)) to form a "table" that has two (vertical) sections: a relational part represented by column `stockID`, coming from a relational expression and a stream part represented by columns `min_price`, `max_price` and `avg_price` coming from aggregation over $Q_{stockID}$. The bold line in Figure 3.1(a) shows this division. Note that each $Q_{stockID}$'s aggregate is functionally dependent on the `stockID` attribute.

stockID	min_price	max_price	avg_price
MSFT	29.12	29.31	29.15
ORCL	19.12	19.19	19.17
BAC	54.48	54.81	54.67
…	…	…	…
GM	35.35	35.87	35.54

(a)

stockID	min_price	max_price	avg_price
MSFT	29.14	29.28	29.16
ORCL	19.17	19.19	19.18
BAC	54.55	54.78	54.65
…	…	…	…
GM	35.43	35.87	35.59

(b)

stock ID	stock_closing_price	category ID	category_closing_price	stock_variation	category_variation
MSFT	29.15	AppSft	45.34	0.997	1.024
ORCL	19.16	AppSft	45.34	0.998	1.024
BAC	54.58	Bank	86.49	1.012	1.006
…	…	…	…	…	…
GM	35.46	Auto	56.74	1.016	0.985

(c)

stockID	sum_vol_10	max_price_10	sum_vol	max_price
MSFT	230.873	29.25	43.120.345	29.31
ORCL	145.899	19.18	12.178.981	19.19
BAC	82.630	54.76	8.230.778	54.81
…	…	…	…	…
GM	34.982	35.75	4.195.946	35.87

(d)

stockID	sum_vol_prc	sum_vol
MSFT	10.185.445	43.120.345
ORCL	8.128.559	12.178.981
BAC	1.263.983	8.230.778
…	…	…
GM	2.078.878	4.195.946

(e)

stockID	alert_flag
MSFT	True
ORCL	False
BAC	False
…	…
GM	True

(f)

stockID	avg_price_10	max_avg_price
MSFT	29.17	29.28
ORCL	19.14	19.17
BAC	54.68	54.68
…	…	…
GM	35.42	35.76

(g)

Figure 3.1: Instances of results for queries Q1 to Q7

Now consider query Q3. In this case we want for each `stockID` to compute the running average price, divide it with the stock's previous day's closing price, do the same for the stock's category and have these two values attached next to the `stockID`. In other words, for each row of the relation shown at the left of the bold line of Figure 3.1(c) – which contains all the necessary information for the computation – we want to define two data sets, $Q_{stockID}$ and $Q_{categoryID}$, aggregate over these and attach the aggregates next to the relation (shown at the right of the bold line). Figure 3.2 depicts graphically the idea.

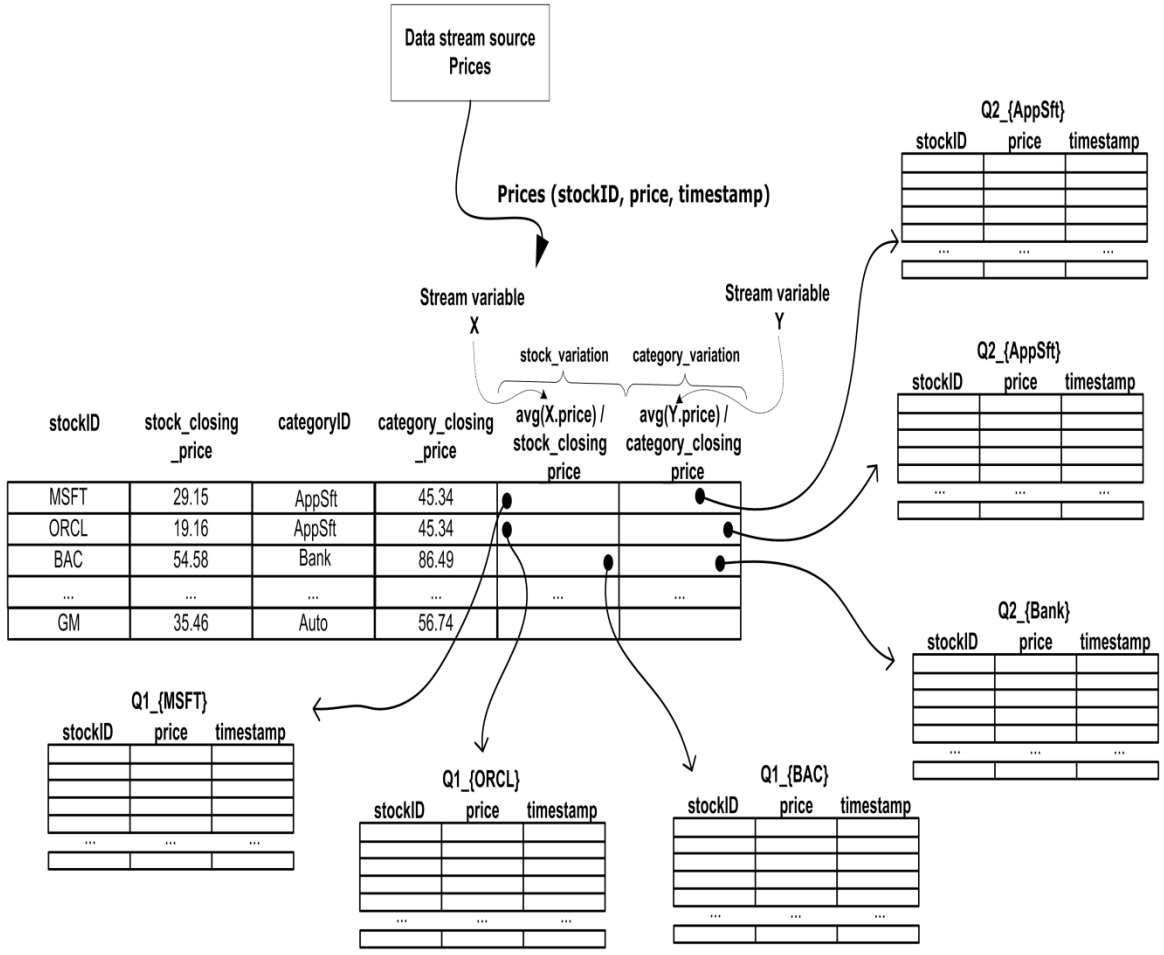


Figure 3.2: Representation of query Q3 using queues

Note that in this case we can keep a data set for each category value and not for each row, which is a form of decorrelation. This is not always the case. For example in query Q5, one can see that the definition of the queue that keeps the volume values depends on aggregates of other stream data sets.

Figure 3.2 shows the idea we model: a relational expression "extended" by one or more columns, where each column represents a stream aggregate of a queue (in principle it can be any abstract data type). We may have more than one queues contributing columns in one such query and there may be interdependencies on the definition of the queues, i.e. a stream aggregate of one queue can be used to restrict the definition of another queue. This approach allows succinct and concise representations of many practical and real-life analytical continuous queries both at user, algebraic and evaluation levels.

3.3 Stream Variables

In this section we define the concept of stream variables to formalize the idea of attaching stream aggregates to relational rows. We use a dynamic queue to store the part of the data stream that is relevant to a relation's row, using a condition θ to determine both containment and queue's size and properties.

3.3.1 Theoretical Framework

Definition 3.1: (Stream Source) A stream tuple s is an ordered list of n values $(s_1, s_2, ..., s_n)$ (an n-tuple), where each value is either an element of a domain A_i or a *NULL* value. A stream source S is any medium able to generate a sequence of stream tuples s (an ordered list of n values $(s_1, s_2, ..., s_n)$) in the unit of time. The schema of S is denoted as $S(S_1, S_2, ..., S_n)$. Each value $s_i \in dom(S_i)$, $i=1, 2, ..., n$ □

Definition 3.2: (Stream Variable) Assume a relation $R(A_1, A_2, ..., A_n)$. A stream variable X over R is defined as a quadruple (A, S, Q, θ), where:

- A is a subset of the attributes of R, i.e. $A = A_{i_1}, A_{i_2,...,}A_{i_k}$ where $\{i_1, i_2, ..., i_k\} \subseteq \{1, 2, ..., n\}$.
- S is a data stream source with schema $S(S_1, S_2, ..., S_m)$.
- Q is a collection of parameterized dynamic queues holding S's stream tuples, $Q = \{Q_t, t \in \pi_A(R)\}$. Implementation-wise one can think Q as an object-oriented class implementing a queue. Instances of this class correspond to Q_t.
- θ is a condition determining what stream data get pushed and popped to which queue. Formally, θ is a (potentially complex) logical expression where each atomic boolean expression has either the form (a) $X.S_i$ <relop> v, $i \in \{1, 2, ..., m\}$, or (b) $X.f(p)$ <relop> v, where:
  - $f(p)$ a function defined over Q having a set of parameters p
  - <relop> is a relational operator such as $=, >, <$ etc.
  - v is either a constant or an attribute of A (or an attribute of R functionally dependent on A).

  We denote this condition as $X.\theta$ □

Intuitively, a stream variable represents a collection of queues, one for each row of the relation R. The logical expression θ determines, given a stream tuple s, to which queues of Q this tuple is

pushed: if θ evaluates to *true* with respect to Q_t and s (Definition 3.3), then s is pushed to Q_t. Note that a stream tuple may be pushed to several queues of Q.

Definition 3.3: (Containment Test) Given a relation R, a tuple r of R, a stream variable over R $X = (A, Q, S, \theta)$ and a stream tuple s of S, we say that $X.\theta$ evaluates to true with respect to r and s, iff the logical expression θ', constructed by the method below evaluates to true:

- Let Q_t the specific queue of Q corresponding to r, $(t = \pi_A(r))$,
- Each term of θ defined as $X.S_i <relop> v$ is replaced by $s.S_i <relop> v$,
- Each term of θ defined as $X.f(p) < relop > v$ is replaced by $Q_t.f(p) <relop> v$. □

We want to be able to "attach" aggregated value(s) of each queue Q_t next to the corresponding row t. We define below the notion of a reporting function over a stream variable.

Definition 3.4: (Reporting Functions) Assume a stream variable $X = (A, S, Q, \theta)$. Any function $f_p : Q \rightarrow D$, where p a set of parameters and D a domain of atomic values, is called a reporting function with respect to X □

Examples include the well-known aggregate functions min, max, sum, count and average as well as UDAFs (user-defined aggregate functions).

Definition 3.5: (Widened Relations) Assume a relation $R(A_1, A_2, ..., A_n)$, a stream variable $X = (A, S, Q, \theta)$ over R and a set of reporting functions $f = (f_1(p_1), f_2(p_2), ..., f_k(p_k))$. We define the *widened* relation with respect to R, X and f, denoted as:

$$WD(R, X, f)$$

as a new relation with schema $(A_1, A_2, ..., A_n, f_1\_p_1, f_2\_p_2, ..., f_k\_p_k)$ and instance: for each tuple t of R, there is a new tuple t', formed by t's values followed by k additional values, the results of $(f_1(p_1), f_2(p_2), ..., f_k(p_k))$ applied on $Q_{\pi_A(t)}$. □

Definition 3.6: (Mapping to Stream Variables) Assume a query named *QR* that contains stream variables. The *QR* produces a relation using the relational attributes and stream variables, called the *widened relation with respect to QR* and is given by the following method:

Algorithm 3.1: Stream variable query output relation schema

```
1:  A list of stream variables {Xi, i=1,2,…,n}
2:  R a relation with schema (A1,A2,…,Ak) computed by the
    select..from..where clause
3:  f1→, f2→, …, fn→ are the reporting functions of X1,X2,…,Xn
    respectively
4:  X1 = (R,S1,Q1,θ1)
5:  V = WD(R,X1,f1→)
6:  for i = 2,…,n do
7:    Xi = (V,Si,Qi,θi)
8:    V' = WD(V,Xi,fi→)
9:    V = V'
10: end for
```

Note that a stream variable is defined over all the attributes of the previously defined widened relation, which is inefficient in some cases since we create a queue for every row of the resulting relation. For instance, in Q3 this is not necessary. By syntactically analyzing the defining condition θ, we can reduce this number, which is a form of decorrelation.

3.3.2 Query Definitions

In this section we use the theoretical definitions of previous section (3.3.1) to describe the examples given in section 3.2.

Example 3.1: Consider query Q1. We first define a relation R as $R = \pi_{stockID}(Stocks)$. This way we get in a column all the distinct *stockIDs*. Then, we define a stream variable X over R as:

$$X = (\{stockID\}, Prices, Q, (X.stockID = stockID \text{ and } X.size() = 0))$$

The first condition of the θ expression of X ensures that a stream tuple s will only be added to the queue that agrees on s' value of `stockID`. The second condition is a built-in function over Q that

allows a queue to have "infinite" size. The fact that a queue is declared with infinite size does not necessarily imply an analogous implementation. For example for distributive [75] reporting functions we can have a queue of size 1. The result of query Q1 is given as the widened relation:

$$WD(R, X, min(price), max(price), avg(price)).$$

Example 3.2: For query Q2 we first define a relation R as $R = \pi_{stockID}(Stocks)$ to get in a column all the distinct *stockIDs*. Then, we define the stream variable X over R as:

$$X = (\{stockID\}, Prices, Q, (X.stockID = stockID \text{ and } X.size() = 100))$$

The first part of θ expression of X ensures that a stream tuple s will only be added to the queue that agrees on s' value of `stockID`. $X.size() = 100$ condition is a built-in function over Q that determines that queue size is equal to 100. The result of query Q2 is given as the widened relation:

$$WD(R, X, min(price), max(price), avg(price)).$$

Example 3.3: Consider query Q3. In this case we need a relation:

```
R(stockID,stock_closing_price,categoryID,category_closing_price)
```

where each row contains information on the stock, its closing price of the previous day, its category and the category's closing price of the previous day. This can be expressed in relational algebra as a join between `Stocks` and `Category` on `categoryID`, a subsequent selection on the `closingPrice` (equal to *date()-1*) –for both `Stocks` and `Category` tables- and a projection to keep only the necessary attributes with proper renaming (`stock_closing_price` ← `Stocks.closingPrice`, `category_closing_price` ← `Category.closingPrice`). Then, we define two stream variables over R, X and Y, where:

$$X = \{stockID\}, Prices, Q_1, (X.stockID = stockID \text{ and } X.size() = 0))$$
$$Y = (\{categoryID\}, Prices, Q_2, (Y.categoryID = categoryID \text{ and } Y.size() = 0))$$

We now get the answer to query Q3 as:

$R1(R, stock\_variation) \leftarrow WD(R, X, avg(price)/stock\_closing\_price),$

$R2(R1, category\_variation) \leftarrow WD(R1, Y, avg(price)/category\_closing\_price)$

Note that we use the standard renaming relational operator (←) [64] to rename the produced widened relation reporting function names (Definition 3.5).

Example 3.4: For query Q4 we project from `Stocks` table the `stockIDs`, $R = \pi_{stockID}(Stocks)$ and we define four stream variables over R, X, Y, Z and W, where

$X = (\{stockID\}, Volume, Q_1, (X.stockID = stockID \text{ and } X.size() = 10))$

$Y = (\{stockID\}, Prices, Q_2, (Y.stockID = stockID \text{ and } Y.size() = 10))$

$Z = (\{stockID\}, Volume, Q_3, (Z.stockID = stockID))$

$W = (\{stockID\}, Prices, Q_4, (W.stockID = stockID))$

For X, Y we define a window with size 10, while for Z,W we define an infinite window. We now get the answer to query Q4 using a sequence of widened relation transformations presented below:

$R1(R, sum\_volume\_10) \leftarrow WD(R, X, sum(volume))$

$R2(R1, max\_price\_10) \leftarrow WD(R1, Y, max(price))$

$R3(R2, sum\_volume) \leftarrow WD(R2, Z, sum(volume))$

$R4(R3, max\_price) \leftarrow WD(R3, W, max(price))$

Example 3.5: In query Q5 we want to use a queue for each `stockID` in order to keep the volume values for the stock. But we should keep the volume value only if the average price of the last 10 reported values of this stock is greater than the running average price. Once again, we first define a relation R as $R = \pi_{stockID}(Stocks)$ and then define two stream variables X and Y as:

$X = (\{stockID\}, Prices, Q_1, (X.stockID = stockID \text{ and } X.size() = 0))$

$Y = (\{stockID\}, Prices, Q_2, (Y.stockID = stockID \text{ and } Y.size() = 10))$

to continuously monitor the running average price and the last 10 reported values of each stock. Then, we integrate *R*, *X* and *Y* in a widened relation *R1* defined as:

$$R1(R, avgPrice, avgPrice10) \leftarrow WD(WD(R, X, avg(price)), Y, avg(price))$$

Widened relation is a standard relation and we can apply rename operation on its schema using relational algebra renaming operator. What we need now to complete the answer to query Q5 is one stream variable *Z* over *R1* to sum volume values if `avgPrice10` is greater than `avgPrice` and one stream variable *W* over *R1* for the running volume total to use for the comparison:

$$Z = (\{stockID\}, Volumes, Q_3, (Z.stockID = stockID \text{ and } Z.size() = 0 \text{ and } avgPrice10 > avgPrice))$$
$$W = (\{stockID\}, Volumes, Q_4, (W.stockID = stockID \text{ and } W.size() = 0))$$

We can use `avgPrice` and `avgPrice10` for the θ condition of *Z* because they are functionally dependent on `stockID`. Otherwise we should have them as attributes of the *A* set (Definition 3.2) of *Z*. This is not always the case. The answer that contains all the required information to select from is given by the widened relation:

$$R2(R1, sum\_vol\_prc) \leftarrow WD(R1, Z, sum(volume))$$
$$R3(R2, sum\_vol) \leftarrow WD(R2, W, sum(volume))$$

Finally we use $\pi_{stockID,\ sum\_vol\_prc,\ sum\_vol}(R3)$ to get the final result.

Example 3.6: For query Q6 we first define a relation *R* as $R = \pi_{stockID}(Stocks)$ to get in a column all the distinct *stockIDs*. Then, we define two stream variables *X, Y* over *R* as:

$$X = (\{stockID\}, Prices, Q_1, (X.stockID = stockID \text{ and } X.size() = 0))$$
$$Y = (\{stockID\}, Prices, Q_2, (Y.stockID = stockID \text{ and } Y.size() = 10))$$

to continuously monitor the running average price and the last 10 reported values of each stock.

The first condition of the θ expression of X and Y ensures that a stream tuple s will only be added to the queue that agrees on s' value of `stockID`. The second condition is a built-in function over Q_1 and Q_2 that allows a queue to have "infinite" size and size of ten respectively. We integrate R, X and Y in a widened relation $R2$ applying the following transformations:

$$R1(R, avgPrice) \leftarrow WD(R, X, avg(price))$$

$$R2(R1, avgPrice10) \leftarrow WD(R1, Y, avg(price))$$

Finally we use $\pi_{stockID,\ avgPrice10>avgPrice}(R2)$ to get the final result. The $avgPrice10>avgPrice$ boolean condition returns either "True" or "False"

Example 3.7: Finally we consider query Q7. This will require treating a stream variable as a new data stream source. We define a relation R as $R = \pi_{stockID}(Stocks)$ and then define a stream variable X and a widened relation $R1$ as:

$$X = (\{stockID\}, Prices, Q_1, (X.stockID = stockID \text{ and } X.size() = 10))$$

$$R1(R, avgPrice) \leftarrow WD(R, X, avg(price))$$

We now define the stream variable Y over $R1$ as:

$$Y = (\{stockID\}, X(avg(price)), Q_2, (Y.stockID = stockID \text{ and } Y.size() = 0))$$

Note that the stream source for Y is X with schema `avg_price` (the reporting function `avg(price)` will be renamed to `avg_price` (Definition 3.5)). We can now use Y to define a widened relation to get Q7's answer:

$$WD(R1, Y, max(avg\_price))$$

3.4 Query Language

In this section we propose a syntactic extension of SQL to handle the incorporation of stream variables and we provide several examples using the new constructs.

3.4.1 Syntactic Constructs

The general syntax is:

```
monitor A1,A2,…,Am,Xi1.f1(p1),Xi2.f2(p2),…,Xik.fk(pk)
from R1,R2,…,Rl:X1(S1),X2(S2),…,Xn(Sn)
where θ
attach when θ1,θ2,…,θn
```

where $A_1, A_2, ..., A_m$ are attributes of relations $R_1, R_2, ..., R_l$, $\{i_1, i_2, ..., i_k\} \in \{1, 2, ..., n\}$, and $f_1(p_1)$, $f_2(p_2)$, ..., $f_k(p_k)$ are reporting functions (Definition 3.5). In particular the added syntactic constructs are described below:

- **monitor:** This newly introduced clause is identical to `select` when applied on relational columns. When applied on reporting functions of stream variables, it rather follows a link to the appropriate queue and shows the current value. We introduce the `monitor` clause in order not to change the semantics of the projection operator. One can still use `select`, but reporting functions of stream variables evaluate to a special constant value (similar to `NULL` and `ALL`).
- **from:** This clause is extended to contain relation names and stream variables, separated by colon. After specifying the participating relations, one may declare one or more stream variables, separated by commas, in the form of: *<stream_variable_name> (<stream_source>)*. The *<stream source>* is an alias name for a data stream that has been declared in a configuration metadata catalog.
- **attach when:** This newly introduced clause contains the defining expressions of the stream variables, separated by commas. The format of each θ_i, $i = 1, 2, ... ,n$ is as described in Definition 3.2

3.4.2 Example Queries

We provide the definition of queries [Q1-Q7] given in section 3.2.1 using the proposed syntactic extensions given in the previous section.

Query example 3.1: Using the proposed SQL extensions, query Q1 can be expressed as:

```
monitor stockID,
        X.min(price) as min_price,
        X.max(price) as max_price,
        X.avg(price) as avg_price
from Stocks : X(Prices)
attach when X.stockID = stockID
```

Query Q1 uses the `Stocks` relation and `Prices` data stream. The detailed information how to access relations and data streams is stored on stream variable system metadata catalog. We define the `X` stream variable over the `Stocks` relation that uses stream data from `Prices` stream. The condition "`X.stockID = stockID`" defines that for each `stockID` we hold stock prices values in `X`'s queue. The `X.min(price)`, `X.max(price)`, `X.avg(price)` are reporting functions computing the running minimum, maximum and average stock price respectively over `X`'s queue.

Query example 3.2: Query Q2 can be expressed as:

```
monitor stockID,
        X.min(price) as min_price,
        X.max(price) as max_price,
        X.avg(price) as avg_price
from Stocks : X(Prices)
attach when X.stockID = stockID and X.size() = 100
```

For query Q2 we define the `X` stream variable over the `Stocks` relation that uses stream data from `Prices` stream. The "`X.stockID=stockID and X.size()=100`" condition defines

that for each `stockID` the query keeps the stock price values in a sliding window of size 100. The query calculates the minimum, maximum and average price over this queue using the reporting functions: `X.min(price)`, `X.max(price)`, `X.avg(price)` respectively. The `size()` function is a custom function defining a sliding window holding the last 100 reported prices. In other words it defines a queue of size 100. The output of reporting functions becomes the columns `min_price`, max_price and `avg_price` next to the `stockID` column.

Query example 3.3: Using the proposed SQL extensions, query Q3 can be expressed as:

```
monitor stockID, stock_closing_price, categoryID,
      category_closing_price,
      X.avg(price)/stock_closing_price as stock_variation,
      Y.avg(price)/category_closing_price as category_variation
from Stocks as S, Category as C: X(Prices),Y(Prices)
where S.categoryID = C.categoryID and S.date = date()-1
attach when X.stockID = stockID and X.size() = 0,
           Y.categoryID = categoryID and Y.size() = 0
```

Query Q3 defines a join between `Stocks` and `Category`. The where clause restricts the query result to contain the stock and category closing prices of the previous date (`date = date()-1`). There exist two stream variables `X` and `Y` using the `Prices` data stream. The condition "`X.stockID = stockID and X.size() = 0`" defines that for each `stockID` we hold the running average price in `X`'s queue. Y's defining condition has the same operation for stock categories. The `X.avg(price)` computes the running average reported price per stock and the `Y.avg(price)` the running average reported price per stock category. Using these values in the select clause, the query computes the percentage variation of both stock and category prices using yesterday's prices.

Query example 3.4: The query Q4 can be expressed as:

```
monitor stockID, X.sum(volume) as sum_vol_10,
        Y.max(price) as max_price_10,
        Z.sum(volume) as sum_vol,
        W.max(price) as max_price
from Stocks: X(Prices), Y(Volumes), Z(Volume), W(Prices)
attach when X.stockID = stockID and X.size() = 10,
            Y.stockID = stockID and Y.size() = 10,
            Z.stockID = stockID and Z.size() = 0,
            W.stockID = stockID and W.size() = 0
```

For query Q4 we define four stream variables: `X, Y, Z` and `W`. `X` and `Y` receive data from the `Prices` stream and `Z` and `W` from the Volumes stream. Both X and Y define a window of size 10, while `Z` and `W` an infinite window. The reporting functions in the select clause compute the aggregates specified in the query definition.

Query example 3.5: Using the proposed SQL extensions, query Q5 can be expressed as:

```
monitor stockID,
        Z.sum(volume) as sum_vol_prc,
        W.sum(volume) as sum_vol
from Stocks: X(Prices), Y(Prices), Z(Volume), W(Volume)
attach when X.stockID = stockID and X.size() = 0,
            Y.stockID = stockID and Y.size() = 10,
            Z.stockID = stockID and Z.size() = 0 and
            Y.avg(price)>X.avg(volume),
            W.stockID = stockID and W.size() = 0
```

For query Q5 we define four stream variables: `X, Y, Z` and `W`. `X` and `Y` receive data from the `Prices` stream and `Z` and `W` from the Volumes stream. The `X` stream variable computes the running average price per stock and `Y` computes the average stock price for the last 10 reported

stock prices. The `Z`'s condition "`Y.avg(price)>X.avg(volume)`" specifies that the reporting function `Z.sum(volume)` is computed only when the running average price is greater than the average price of the 10 last prices. This is a type of correlated aggregation [42][67]. The `W` stream variable calculates the running total volume for each stock.

Query example 3.6: Using the proposed SQL extensions, query Q6 can be expressed as:

```
monitor stockID, (Y.avg(price)>X.avg(price)) as alert_flag
from Stocks: X(Prices), Y(Prices)
attach when X.stockID = stockID and X.size() = 0,
            Y.stockID = stockID and Y.size() = 10
```

The query Q6 contains two stream variables `X` and `Y` that use the `Prices` stream. `X` defines an infinite queue and `Y` defines a queue of size 10. The monitor clause identifies in real time if the running average price is greater than the average price of the last 10 reported values. This predicate is a boolean condition resulting to either a "True" or a "False" value.

Query example 3.7: Using the proposed SQL extensions, query Q7 is:

```
monitor stockID,
        X.avg(price) as avg_price_10
        Y.max(price) as max_avg_price
from Stocks: X(Prices), Y(avg_price_10)
attach when X.stockID = stockID and X.size() = 10,
            Y.stockID = stockID and Y.size() = 0
```

The query Q7 contains two stream variables `X` and `Y`. `X` uses the `Prices` stream and defines a queue with size 10. The `Y` stream variable uses as a source the aggregates of `X` stream variable (`avg_price_10`). Over these aggregates an infinite queue is defined and the condition "`Y.stockID = stockID and Y.size() = 0`" computes the maximum of these aggregates.

3.5 Evaluation and Optimizations

In this section we provide a straightforward evaluation algorithm for stream variable queries and several optimizations.

3.5.1 Evaluation Algorithm for Stream Variable Queries

The computation of stream variable queries is based on a straightforward but highly optimizable evaluation algorithm presented below:

Algorithm 3.2: Evaluation algorithm of a stream variable query (W)

```
1:  variables
2:  A list of stream variables {Xi, i=1,2,…,n}
3:  A list of queues for each stream variable {Qt, t=1,2,…,m}
4:  A data source S producing tuples s(a1,a2,…,ak)
5:  end variables
6:  when a tuple s from S becomes available do
7:    for each stream variable Xr of W query such that the
      data source for Xr is S do
8:      for each row r of the widened relation with respect to W
        such that Xr.θ evaluates to true with respect to r and s
          do
9:          push s into Qt
10:         calculate Xr reporting functions over queue Qt
11:       end do
12:     end for
13:   end for
14: end when
```

The algorithm operates as follows: for a given stream tuple s of stream source S and for each stream variable X_r that mentions S as its source, we check all rows of the widened relation w.r.t W to see whether they satisfy X_r's condition.

3.5.2 Optimizations

This simple algorithm can be very expensive if the widened relation w.r.t W is large. There are several optimizations that can be applied to reduce this cost, mentioned briefly below:

- **Indexing:** It is important to analyze the θ condition of a stream variable in order to deduct (if possible) which rows of the widened relation w.r.t W will be updated and

avoid a full scan. In most of the cases, it is only a few. By using cleverly created indices, cost can be reduced further.

- **Decorrelation:** By default, according to mapping of Definition 3.6, stream variables are defined over the relation of the previous widened relation. However, sometimes this is not necessary, as in Q3. By performing some syntactic analysis of the θ condition of a stream variable, several rows of the widened relation w.r.t W may use the same queue.
- **Parallelism:** One can horizontally partition the widened relation w.r.t W to several processing nodes and distribute the stream tuples to all of these. The result is the union of all the sub-results.

3.6 Implementation and Experiments

3.6.1 Stream Variables System

We have developed a prototype system which incorporates many of the concepts described in the previous sections. Our system has been implemented in C/C++ and follows the general DSMS architecture described in [21]. The main purpose was to build a prototype system to be used as a proof of concept.

Our prototype system follows a component-based architecture. Each component has a well specified API which increases code reusability and makes future improvements easier. Figure 3.4 shows the main components of our system.

Initially the user formulates a query following the syntactic extensions described in subsection 3.4.1. *Query parser* validates query's syntax and *Query optimizer* analyzes the query for possible optimizations (as described in Section 3.5.2). Once the base relation has been computed and loaded and the window structures initialized and linked to the base relation, the *Scheduler* starts probing input queues (one for each data source) in a round robin fashion for incoming stream tuples. These are forwarded to the *Executor* for processing. *Executor* implements Algorithm 3.2. Base relation and window structures are stored in memory. Metadata catalog provides information for data source names and types, window schemas, etc.

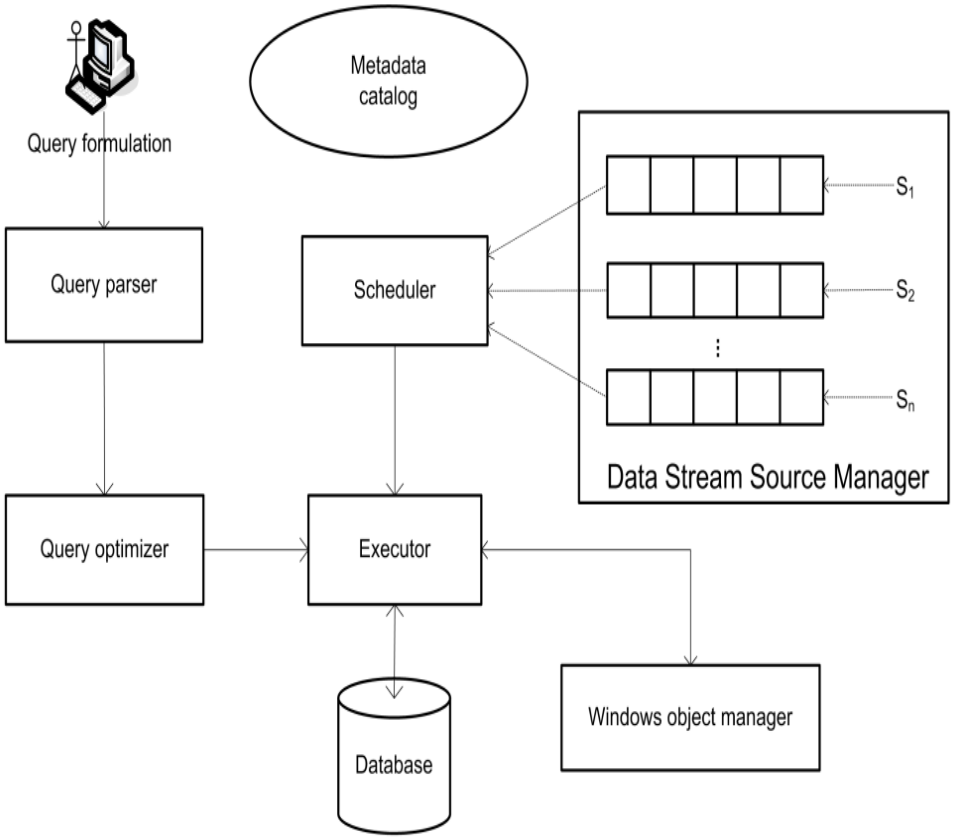


Figure 3.3: Stream Variables system

3.6.2 Experiments

We conducted some experiments to measure the efficiency and scalability of Stream Variable system. Our tests were performed on an Intel Core 2 CPU @ 2.0Ghz with 2GB main memory, running Windows XP

Figure 3.5 shows performance results for query Q2 with different optimization parameters. In this test we use a flat file as a data stream source (`Prices` data stream). The base relation contains 100 values (read from a flat size) and we varied the size of the incoming stream (read from a flat file) from 20000 to 100000 tuples with a step of 20000. We plot the query completion time. The completion time contains: (a) reading from disk the base relation values, (b) building windows, (c) reading from disk the incoming stream tuples and (d) query evaluation (Algorithm 3.2). For both line plots the *optimizer* identifies the equality predicate in `such that` clause and the *executor* built a memory hash index (C++ hash map) on `stockIDs`. For the top line plot, we force optimizer not to identify that the declared aggregate functions (min, max, avg) can be evaluated using a window of size 1. In this case *executor* builds a sliding window (in-memory array) of size 100 and re-evaluates the min, max and avg functions over 100 values when a new

value inserted in the window. For the bottom line plot, executor uses a window of size 1 resulting in better performance results. This simple example indicates that our system behaves satisfactory in processing high-rate streams coming from disk.

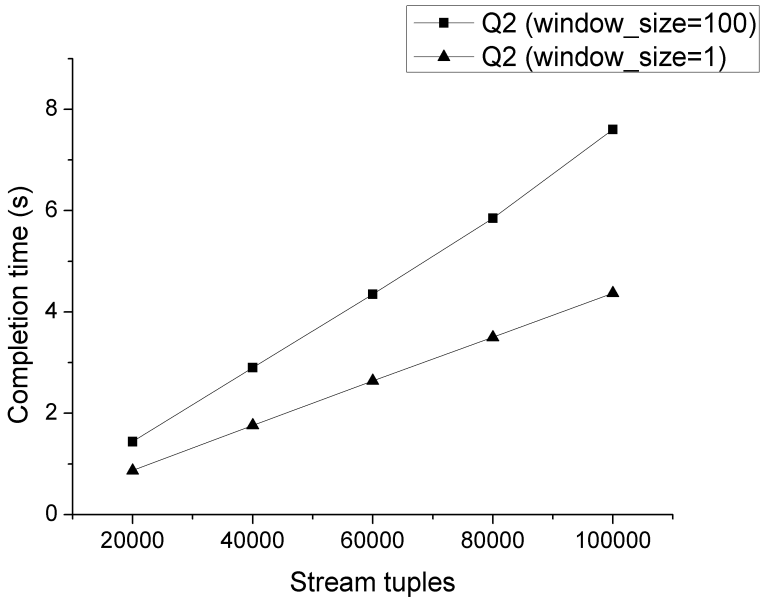


Figure 3.4: Query Q2 completion time

Similarly, Figure 3.6 shows performance results for query Q2. In this case stream tuples (`Prices`) are kept in memory to avoid disk reading. Also we do not count the time the system needs to load the base relation values from disk, the time for building the appropriate window structures and the time needs to build a hash-index for the base relation values. We focus only on the performance of evaluating the query Q2 using Algorithm 3.2. The windows used for the computation of min, max, avg are of size 1 and are common C++ memory arrays. We varied the size of the incoming memory stream from 100000 to 500000 tuples with a step of 100000 and plotted the query Q2 evaluation time when the relation size of the base relation ranges from 100 to 500 tuples with a step of 100. This simple example indicates that our evaluation algorithm performs well in processing high-rate streams (existing in-memory), while it scales well as the base relation size increases. Comparing with the plot in Figure 3.5 we can see that there is a lot of overhead in the *Scheduler* (reading from disk) and the *Windows object manager* (creating window structures).

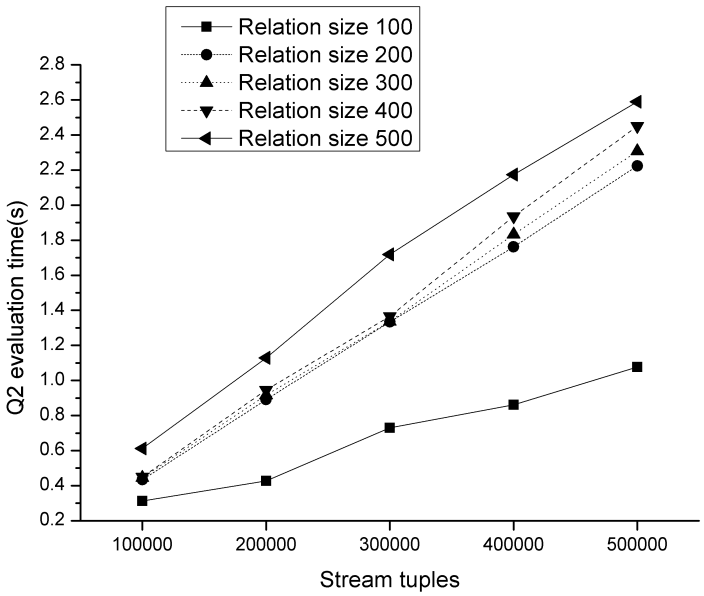


Figure 3.5: Query Q2 evaluation time for different base relation sizes

3.7 Summary and Conclusions

Data stream management systems have been the focus of intense research activity in the past few years. Real-time analytics and continuous queries become increasingly important topics both in the research and the industry worlds. The goal of this work is not to build a complete data stream system, handling issues such as load shedding, approximate answers and others, but to model and handle useful class of continuous queries for real-time analytics.

A simple SQL extension is introduced in order to facilitate the succinct expression of analytics over stream data. These analytics queries consist of aggregates of repeated, multiple, possibly correlated and at different granularities stream selections. We presented a motivating application from the financial world along with several query examples. We define the concept of widened relations. The key idea is that aggregates over stream data can be repeatedly added to a base relation.

The proposed language can provide a generic framework for declaring analytics queries over stream data coming from multiple stream sources.

Chapter 4

Spreadsheet-like Stream Processing

4.1 Introduction

Decision support systems (DSS) are based on data stored statically and persistently in a database, typically in a data warehouse. The queries applied over these data enable analysts to take proper and efficient decisions. In many applications however, it may not be possible to process queries within a Database Management System (DBMS). These applications involve data items that arrive on-line from multiple sources in a continuous fashion [27]. In Data Streams Management Systems (DSMSs) we usually have "continuous" queries [21][71] rather than "one-time". Computing real-time analytics (potentially complex) on top of data streams is an essential component of the real-time enterprise and an essential requirement of DSMSs.

In this chapter we present the theoretic foundations and a system called COSTES (Continuous Spreadsheet-Like Computations) that allows users to formulate easily continuous queries for analytics and decision purposes. These queries mingle traditional and stream data in a single, correlated view. This class of queries – which resemble spreadsheet documents, where the definition of a column usually depends on previously defined columns and some initial "basic" columns - is particular suitable for stream data management and are used mainly for decision support. As a result, the purpose of COSTES is not to serve as a complete and generic Data Stream Management System (DSMS), but rather to form a useful and practical tool for stream queries used in (near) real-time decision making.

4.2 Challenges

Stream queries used for decision support tasks are important for companies and organizations to gain insight about their operations in (near) real-time. The goal is to provide a theory and a platform for real-time analytics beyond common offline and ad-hoc analytics which are quite common. In order to support decision support stream queries a theoretical framework and a proper language must be defined.

To achieve this goal a number of requirements should fulfilled:

- **Continuous queries (R1):** The nature of the queries should be continuous, i.e. the result is updated as new data arrives.
- **Tabular format (R2):** The output of such a query should be in tabular format, appropriate for (subsequent) traditional database processing or input to visual tools.
- **Multiple data sources (R3):** The language should allow consolidation/aggregation of multiple data streams with different schemata into one query.
- **Balancing declarative/procedural tradeoffs (R4):** We should specify declaratively most of the query - ripping the benefits from traditional database access methods, while allowing procedural flexibility in the aggregation part.
- **Correlated computations (R5):** Many queries have a common pattern: the leftmost column(s) consists of one or more fixed values (e.g. stock names, vehicle IDs, sensor IDs etc) and possibly related information, while the remaining columns are defined successively based on previously defined columns.
- **Optimizations (R6):** The proposed approach must enable a set of optimizations for performance and efficiency reasons. This is important as (near) real-time decision support systems must handle high throughput stream data and compute results in a fast manner.

4.3 Radio Frequency Identification (RFID) Technology and Applications

We use a motivating application from the supply chain management field. Radio Frequency Identification (RFID) technology is used for real-time product monitoring and supply chain automation. In this chapter we provide background details about these areas.

In the recent years, the development of automatic identification technologies, such as Radio Frequency Identification (RFID) paired with sensor–based technologies and ubiquitous computing have caused an explosion in data capturing and real-time information processing requirements, presenting new challenges and opportunities for the development of data stream management applications.

RFID technology has been extensively used for a diversity of applications ranging from access control systems to airport baggage handling, livestock management systems, automated toll collection systems, theft-prevention systems, electronic payment systems, and automated production systems. Nevertheless, what has made this technology popular nowadays is the application of RFID for the identification of consumer products and the management of supply-chain processes. Tagging individual product instances at item level and tracking them across the supply chain generates immense streams of data at various stages. The challenge is to efficiently

filter these data to support real-time decisions in the context of various business applications, from upstream warehouse and distribution management down to retail-outlet operations, including shelf management, promotions management and innovative consumer services.

As a result the emergence of new automatic identification technology (RFID) is expected to revolutionize many of the supply chain operations by reducing costs, improving service levels and offering new possibilities for identifying unique product instances. The advanced data capture capabilities of RFID technology coupled with unique product identification and real-time information integrated from different data sources define a new and rich information environment that opens up new horizons for efficient management of supply chain processes and decision support.

Currently, most applications of RFID in supply chain management exploit the automation capabilities of the technology with the objective to speed-up processes and reduce costs, such as the automatic identification of incoming and outgoing goods in warehouse operations or asset tracking in closed-loop applications. However, the real potential of RFID lies in the possibility to capture new types of information in real-time and support decisions. We are towards that RFID tags are not simply used to replace barcodes and automate processes, but to provide real-time information in order to create new business opportunities and experiences for the customers.

Real-time information capturing and decision support present complex technical challenges, related to managing huge streams of data coming from multiple data sources and converting them into meaningful information in a way to support decisions. Until recently, decision support systems (DSS) were based on data that were stored statically and persistently in a database, typically in a data warehouse. Complex queries and analysis were carried out upon this data to produce useful results for managers. In many RFID applications however, it may not be possible to process queries within a database management system. RFID applications involve data items that arrive on-line from multiple sources in a continuous fashion. This data may or may not be stored in a database. We must have “continuous” queries rather than “one-time”. The answer to a continuous query is produced over time, reflecting the stream data seen so far. Computing real-time analytics (potentially complex) on top of RFID data streams is an essential component of the real-time supply chain enterprises and an essential requirement for decision making.

Supply chain applications range from upstream warehouse and distribution management down to retail-outlet operations, including shelf management, promotions management and innovative consumer services [116]. Most of these applications are characterized as “open-loop”, meaning that they require the involvement of different supply chain partners in an open environment in order to be implemented, as, for example, the involvement of the supplier/manufacturer to attach an RFID tag to the product and the involvement of the distributor or retailer to monitor product movement in warehouses or stores by installing RFID readers at various locations.

This fact is probably what poses the greatest challenge for the application of RFID technology in supply chain management today, as the involved partners cannot equally share the associated costs and benefits. For suppliers, RFID, as a tag that has to be placed on their products, is often considered to be an unfortunate strategic necessity [18] they have to comply with in order to satisfy the plans of their big customers for increased internal efficiency. For suppliers to benefit from RFID they need to share RFID information with their partners and exploit this information in order to streamline supply chain processes and gain new market knowledge [130]. At the same time, for both retailers and suppliers, investments in RFID technology cannot be justified by operational gains alone and more strategic benefits need be materialized through advanced information acquisition and decision support.

Overall, and in line with [90], the gradual contribution of RFID in supply chain management, as an automatic product identification technology, across the following axes:

- the automation of existing processes, leading to time/cost savings and more efficient operations;
- the enablement of new or transformed business processes and innovative consumer services, such as providing consumer self check-out or product-information services;
- the availability of richer and more accurate information in real-time, offering the potential for advanced decision support and market knowledge acquisition.

In order to move from the level of automation and operational benefits to the level of advanced decision support we need to efficiently and effectively transform RFID data into meaningful reports, both internally within a company and in a collaborative supply chain environment where information is shared among supply chain partners.

4.4 RFID Motivating Application

4.4.1 Application Scenarios

An RFID system consists of RFID readers with antennas, host computers and transponders or RF tags. The EPC (Electronic Product Code) standard specifies unique product IDs in the supply chain environment. RFID applications generate large volume of streaming data, which have to be automatically filtered, processed, grouped and transformed into meaningful information to be used in business applications.

This need is better illustrated by the following two application scenarios. These scenarios employ product item-level RFID tags, while RFID readers and antennas are placed on store shelves, so that the product movement on and off the shelf is monitored.

The first scenario monitors shelf availability. This scenario refers to the possibility of monitoring the existence or not of products on supermarket shelves in order to replenish them on time. Given the negative impact that "out-of-shelf" (OOS) has on consumer attitude, sales and loyalty, retailers and suppliers have raised this issue to a top priority for their industry today and confront RFID as a possible solution to this problem. The requirements for a real-time report monitoring shelf availability would be as following:

- The report presents the remaining quantity of each product on the shelf, where a product is identified by its description at the product type level.
- The report is updated in real-time, depicting product sales off the shelf as well as shelf replenishment activities.
- A user, e.g. store employee or supplier, can view the report or get OOS notification alerts in order to make shelf replenishment decisions.
- When the last item of a product on the shelf has been sold, an OOS is reported.
- The duration of the OOS is tracked until the shelf is replenished back.

The second scenario relates to products' promotion management. A particularly important marketing activity for fast moving consumer goods is sales promotions, which represent the majority of manufacturers' marketing budgets. Despite the importance of sales promotions and the amount of revenues devoted to them, suppliers often fail to evaluate sales promotion effectiveness and when they do so, this is usually several weeks after a promotion has ended. Being able to monitor the effectiveness of in-store sales promotions in real-time gives a supplier the possibility to act proactively and ensure the success of a sales promotion. In order to do so, a supplier should have access to real-time information about product sales in the store, including information about:

- The sales off-take from a specific promotional stand versus the shelf or other locations in a store. If a location is underperforming, then the supplier should probably request the store to change the location of a promotion.
- The availability of products on both the shelf and the promotion stands in order to drive replenishment decisions.
- The products that a consumer has already put in her basket when selecting a product from a promotional stand or those that she replaces as a response to a sales promotion.

The above application scenarios and respective reports are indicative examples that demonstrate the need for real-time decisions in supply chain management, exploiting the possibilities offered for automatic and unique product identification through the use of RFID technology. In the next sections of the current chapter we propose a decision support tool, utilizing

a data stream management system to aggregate RFID data, addressing the aforementioned requirements for real-time decisions in the supply chain context.

For both applications scenarios we assume that we have installed RFID readers in a super market monitoring the presence of products on shelves and five promotional stands. The application-level-event (ALE) middleware services [10] retrieve filtered RFID data from RFID readers. Particularly, the asynchronous subscribe mode of ALE service, where a client registers a subscription and the ALE service periodically sends aggregated events back to the client. Our system manages data streams over ALE. The setting is described in Figure 4.1:

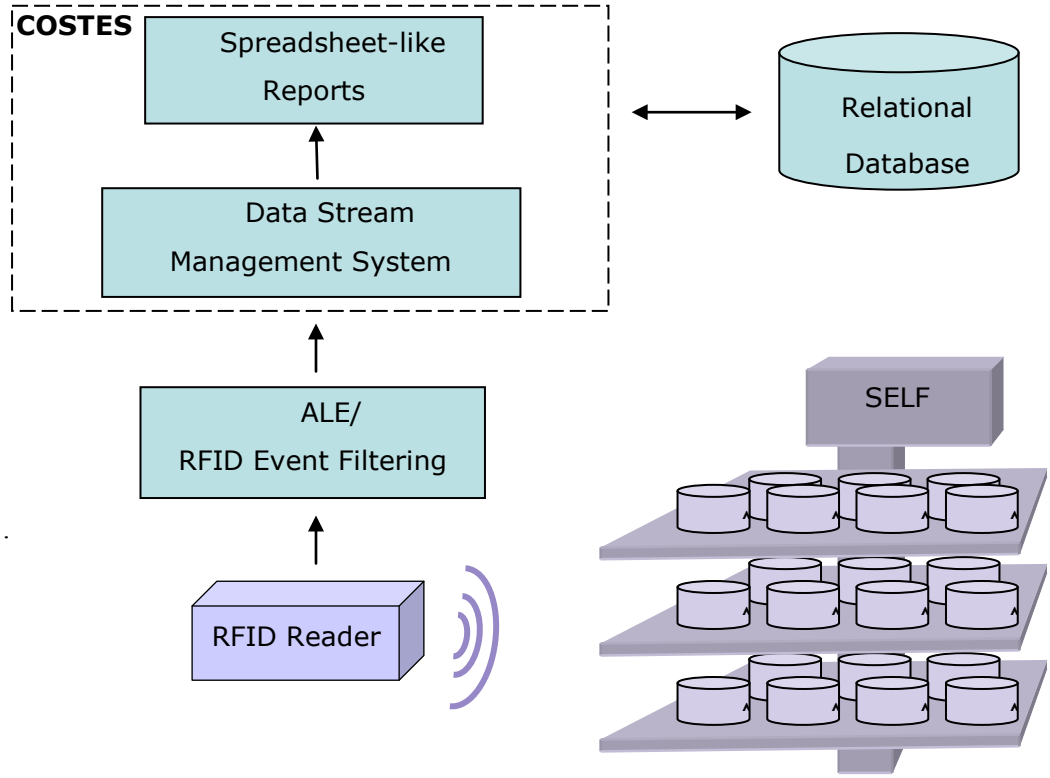


Figure 4.1: Application scenarios setup

Let's assume that a product may be displayed at more than one location in the store, e.g. the regular shelf position and a promotional stand. The schemata of the RFID data streams presented to our system, after filtered and aggregated by the ALE middleware, are:

```
Readings(EPCProdCode, quantity, timestamp)
Stand1(EPCProdCode, quantity, timestamp)
Stand2(EPCProdCode, quantity, timestamp)
```

```
Stand3(EPCProdCode, quantity, timestamp)
Stand4(EPCProdCode, quantity, timestamp)
Stand5(EPCProdCode, quantity, timestamp)
```

Each stream tuple reports the EPC product-type code (`EPCProdCode`), the measured quantity of the product (`quantity`) and the timestamp of the measurement (`timestamp`). Streams are aggregated by the ALE service at the product-type level, i.e. the GTIN number occupying the first 13 digits of an EPC (usually referred to as barcode). `Readings` report products' status placed on regular shelves and `Stand1` to `Stand5` report products' status placed on stands one to five respectively. We assume that a product exists only in one location at the store's shelves and possibly at one promotional stand. While customers often misplace products, ALE middleware only reports product quantities found at their designated places (regular shelf and promotional stands). In other words, `Readings` and `Stand1` to `Stand5` streams report clean data.

Also we assume the presence of two tables in our system:

```
Products(prodCode, threshold,…)
Promotions(promCode, standNumber, prodCode, profit, …)
```

`Products` stores information about products, such as product code, minimum threshold values, location, etc. and `Promotions` stores information about past promotions.

4.4.2 Example Queries

There are several useful continuous queries one can register on top of these streams to monitor shelf replenishment and compare sales of the same product placed at different locations:

Q1. Shelf replenishment has been identified as one of the main benefits of RFID technology. It is important to know when a product's quantity on the shelf has reached a critical threshold and notify the store's manager to replenish it.

Q2. Similarly, it is useful to know how long it takes to replenish the shelf (for each product), starting counting from the first occurrence of "below-the-threshold" event, in order to monitor product availability on the shelf and duration of out-of-stocks.

Q3. During promotional periods of a compilation of products placed on a stand we are interested in measuring the effectiveness of the promotion. We can do so by comparing the sales rate of the same product from the shelf and the promotional stand.

Q4. A more demanding report would be computing the time a product takes to get from its average quantity on the self to its threshold. This would allow store managers to better understand and schedule shelf replenishments and make shelf-space allocation decisions.

Instances of the output of query Q1 to Q4 are is shown in Figure 4.2 (a) to (d):

EPCProdCode	threshold	max_quantity	alert
1	10	47	FALSE
2	12	7	TRUE
3	8	22	FALSE
4	15	11	FALSE
…	…		…

(a)

EPCProdCode	threshold	time_to_repl
1	10	NULL
2	12	3200
3	8	NULL
4	15	1800
…	…	…

(b)

EPCProdCode	self_variance	stand_variance
1	0.61	0.82
2	0.43	0.35
3	0.81	0.84
4	0.35	0.61
…	…	…

(c)

EPCProdCode	threshold	avg_quantity	time_to_threshold
1	10	48	32400
2	12	53	267988
3	8	26	169366
4	15	39	92102
…	…	…	…

(d)

Figure 4.2: Instances of results for queries Q1 to Q4

All queries have a common pattern: the leftmost part of the resulting table corresponds to a traditional relational expression, while the rest of the columns represent aggregates over sets of values formed over stream data. Let us consider Query Q2. The idea we want to express is the following:

Algorithm 4.1: Query Q2 evaluation algorithm

```
1:  compute table R(EPCProdCode, Threshold);
2:  add column time_to_repl to R;
3:  for each row r of R do
4:   define an ordered set S^r of real type values
5:   empty S^r;
6:   r.time_to_repl = null;
7:  end-for
8:  for each stream tuple (p, q, t) from Readings do
9:   for each r in R do
10:   if (r.EPCProdCode==p && (r.threshold > q || empty())) do
11:    push t to S^r;
12:    r.time_to_repl = S^r.last_elem - S^r.first_elem;
13:   end-if
14:  end-for
15: end-for
16:
17: function bool empty()
18:  begin-function
19:   if(r.Time_to_Repl is null)
20:    return false;
21:   end-if
22:   else
23:    set r.Time_to_Repl to (last_elem - first_elem of S^r);
24:    store r;
25:    clear S^r;
26:    r.time_to_repl = null;
27:   return false;
28:  end-else
29: end-function
```

Table R is computed by a traditional relational expression. Then, for each row of R we define an ordered data set that contains values (`timestamps`) from the `Readings` stream. The timestamp of a stream data must be appended to a row's data set if a condition is satisfied – in our case if the reported `EPCProdCode` corresponds to the row's `EPCProdCode` and the reported quantity is below the row's threshold. However, when the condition does not hold, the data set must be emptied - `empty()` is a function returning always false, having as a side effect to empty the ordered set of the row (S^r). It also calculates and stores the out-of-stock duration and clears the out-of-stock variable (line 26) at the first occurrence of "above the threshold" event. Note that we have short-circuited evaluation. One could examine only the row that agrees on the `EPCProdCode`, but this is an optimization specific on the condition. In other examples, a stream value may be inserted to more than one data sets.

This is the idea we model: start from a relational expression (the base table), use each row as a parameter to define one or more, possibly correlated, parameterized subsets of stream sources (associated sets) and extend the schema with aggregates over these. Membership to these sets should be defined declaratively, while the aggregate computations can be any user-defined aggregate function. We argue that this approach allows succinct and concise representations of many practical monitoring queries. The challenge is to have a simple query language to express and an efficient, optimizable, algorithm to evaluate such queries. Note that each ordered data set S^r is "attached" to a persistent relational value, i.e. S^r can be thought as labeled by a "stable" value. This can be used to develop a relational operator to express this class of queries.

4.5 Continuous Spreadsheet-like Computations

4.5.1 Theoretical Framework

Below we provide the definitions of our framework to support queries as those given in the previous section and taking into consideration the requirements mentioned in section 4.2.

Definition 4.1: (Associated Set) Given a relational schema B, a relational schema of a stream source S and a condition θ involving attributes of B and S and constants, then we define the associated set of B, S and θ, denoted as $Assoc(B, S, \theta)$, as a collection of parameterized multi-sets able of storing S's tuples, where the columns of B serve as the parameters. Each parameterized multi-set of $Assoc(B, S, \theta)$ is denoted as $Assoc^r(B,S,\theta)$, where r is a row of B. B is called the base-values table, S the source and θ the defining condition of the associated set. $Assoc^r(B, S, \theta)$ is called the instance of the associated set with respect to r. □

Implementation and selection of these data structures (e.g. multisets) are left to the optimizer. Note that associated sets can also be defined over traditional data sources, such as flat files and relations, as long as they present a relational interface and there is a tuple iterator over the data source.

The class of queries that we want to support (COntinuous SpreadsheeT-likE computations - COSTES query-) is based on repeated, consecutive definitions of associated sets and their schema can be described by the following algorithm:

Algorithm 4.2: COSTES queries schema evaluation algorithm

```
1:  assume S1,S2,...,Sn stream sources;
2:  B is the initial schema of the base-values table;
3:  for (i=1 to n) do
4:   let Θi be a condition involving attributes of B and S and
     constants;
5:   Ai = Assoc(Bi, Si, θi);
6:   let f1(si1),f2(si2),…,fk(sik) a set of aggregate functions on
     attributes of Si;
7:   extend B's schema with k columns, Ai_fj(sij), j=1,2,…k
     and name it B1;
8:   attach a null value to A^r_i at row r and column Ai_fj(sij),
     j=1,…,k;
9:   B = B1;
10: end-for
```

Algorithm 4.1 is just the schema of a COSTES query. How such a query is evaluated is explained in section 4.7.

Definition 4.2: (Associated Set Membership Test) given a stream tuple s of S_j *we* say that s satisfies the membership test for associated set A^r_i, *if* $i=j$ and θ_i evaluates to true with respect to r and s. In all other cases the return value is NULL.

4.6 Query Language

The goal is to express a large class of practical continuous queries as those shown in the mentioned application (Section 4.4.2) using some intuitive extension of SQL.

4.6.1 SQL Extensions

The syntactic constructs of the proposed language is as follows:

```
select A1, A2, …, Am, {Ci.fj(S1)}i =1,…,d
from R1, R2, …, Rk
<where θ>
<group by  A1, A2, …, Am>
extended by C1(S1), C2(S2), …, Cn(Sn)
such that θ1, θ2, …, θn
```

where $A_1, A_2, \ldots, A_m$ attributes of $R_1, R_2, \ldots, R_k$ tables, i in $\{1, 2, \ldots, n\}$, f_j an available aggregate function and S_l an attribute of S_i's schema. The newly introduced `extended by` and `such that` clauses define the additional columns that will be "attached" to the relation defined by the `select…from…where…group` by clause (the initial base-values table). In particular the added syntactic constructs are described below:

- **select:** The `select` clause may contain one or more aggregate functions of the associated sets defined by the `extended by` clause. Since more than one aggregate functions per associated set is possible, we may have more than m+n columns in the output table.
- **extended by:** This clause is used to declare the names and the data sources of the associated sets, in a comma-separated list.
- **such that:** This is a comma-separated list of the defining conditions of the associated sets. Condition θ_i involves attributes of the initial base-values table, constants and aggregates of associated sets $C_1 \ldots C_{i-1}$.

Given the nature of the `such that` clause, all continuously updated associated sets are functionally dependent on some column(s) of the initial base-values table (directly or transitively).

4.6.2 Example Queries

In this section we show how queries Q1 to Q4 described in section 4.4.2 can be expressed with the proposed SQL extensions.

Query example 4.1: Using the proposed SQL extensions, query Q1 can be expressed as:

```
select prodCode as EPCProdCode, threshold,
       X.max(quantity) as max_quantity,
      (X.max(quantity)<threshold) as alert
from Products
extended by X(Readings)
such that X.EPCProdCode=prodCode and X.size()=1
```

Query Q1 requires an associated set of size 1 to be defined for each `prodCode` tuple of `Products` – to keep only the last quantity reported. We use the max aggregate function to retrieve this single value. `size()` is a method that enforces `X` to have size 1 (implementation wise, these are methods of the data structures that will implement associated sets.) `Threshold` is also required in the base table because it is used in an expression in the `select` clause.

Query example 4.2: Using the proposed SQL extensions, query Q2 can be expressed as:

```
select prodCode as EPCProdCode, threshold,
       X.diff(timestamp) as time_to_repl
from Products
extended by X(Readings)
such that X.EPCProdCode=prodCode and
          (X.quantity < threshold cor empty())
```

This query requires an associated set `X` for each `prodCode`, which is populated by the stream's `timestamp` when the reported quantity drops below the threshold. However, this set has to empty whenever this condition does not hold. We use a system function called `empty()` which always returns false and as a side effect empties the corresponding set. `cor` is the short-

circuited disjunction operator. The aggregate function `diff` computes the difference between the last and the first element of `X` – forcing the optimizer to use an ordered data set for `X`.

Query example 4.3: Using the proposed SQL extensions, query Q3 can be expressed as:

```
select ProdCode as EPCProdCode,
       X.var(quantity) as shelf_variance,
       Y.var(quantity) as stand_variance
from Promotions
where promCode=172 and standNumber=1
extended by X(Readings), Y(Stand1)
such that X.EPCProdCode=prodCode and X.size()=200,
          Y.EPCProdCode=prodCode and Y.size()=200
```

Assume that, for those products participating in promotion 172, we want to compare their sales rate from the self and the first stand. For each such product code, we define two associated sets of maximum size 200, named `X` and `Y`, where `X` and `Y` contain the reported quantity from the standard stream (`Readings`) and first stand's (`Stand1`) streams. We want to monitor the variance of those sets.

Query example 4.4: Using the proposed SQL extensions, query Q4 can be expressed as:

```
select prodCode as EPCProdCode, threshold,
       X.avg(quantity) as avg_quantity,
       Y.diff(timestamp) as time_to_threshold
from Products
extended by X(Readings), Y(Readings)
such that X.EPCProdCode=prodCode,
          Y.EPCProdCode=prodCode and ((Y.quantity > threshold
          and Y.quantity < X.avg(quantity) cor empty())
```

Finally, Query Q4 is similar to Q2, but an extra comparison between the reported quantity and the shelf's running average is required. Consequently, we need to define for each `prodCode` a data set `X` keeping the reported quantities and use `X`'s average to constrain membership to `Y`.

4.6.3 Requirements

By construction, the output of our queries is continuous and tabular (Requirements [R1] and [R2].) By allowing associated sets to be defined over different data sources we address requirement [R3]. Membership to associated sets is defined declaratively through the `such that` clause, which allows selection of the access methods that can be used. Aggregate computation over the defined associated sets can be anything, which adds to procedural flexibility. However, even in this case some optimization is possible, such as appropriate selection of data representation (e.g. stacks, queues, min-max heaps, etc.) of the associated sets (Requirement [R4].) Finally, the fact that aggregates of an associated set may constrain the definition of subsequent associated sets, as in Query Q4, addresses Requirement [R5].

4.7 Query Evaluation and Implementation

In this section we describe the query evaluation algorithm and suggest possible optimizations for COSTES queries. We present our implementation and provide query performance results.

4.7.1 Query Evaluation

A COSTES query can be continuously updated in a very simple manner. The evaluation algorithm operates as follows:

Algorithm 4.3: COSTES (naïve) evaluation algorithm

```
1:  for a tuple s of data source S do
2:   for each associated set X having source S do
3:    for each row r of B do
4:     if(θx is true with respect to r and s) do
5:      append s into associated set instance X^r;
6:      evaluate X's aggregate functions mentioned in select
        clause over associated set instance X^r;
7:     end-if
8:    end-for
9:   end-for
10: end-for
```

4.7.2 Optimizations

Although the evaluation algorithm is rather simple, several optimizations are possible:

- **Optimization 4.1 - Projection:** Line 5 dictates that a stream tuple s must be appended to the data structure representing associated set instance X^r. A naive approach for associated sets' implementation is to store the complete data stream tuples. However, one can parse the select and such that clauses and identify the necessary attributes to keep. So, one can append to X^r only those attributes of s needed to the computation of aggregates of X. For example, in query Q3, only `quantity` will be appended to associated sets X and Y.
- **Optimization 4.2 - Filtering:** Parts of the defining condition θ_x may be relevant just to the stream tuples, i.e. the defining condition may be rewritten as $\theta_x = \theta'_x \wedge \theta_c$ where θ_c is an expression involving only attributes of S and constants. Some systems allow pushing simple filtering conditions to the stream source, saving iterations of the main loop at Line 3. Push simple selections to the source of the stream, is similar to Gigascope's approach of low- and high- levels of query processing [57][58] or low-level filtering in sensor networks [86], to avoid extra processing power or battery consumption respectively. For example, in query Q3 we may want to consider `Readings` tuples for X associated set only if the `quantity` is greater than a constant value (e.g. threshold). The defining condition of X would be in this case: *X.EPCProdCode=ProdCode and X.quantinty>threshold and X.size()=200*. This could be broken down to two generic windows, X_1 and X_2, X_1 having as defining condition *X.quantinty>threshold and X.size()=200* and source the `Readings` stream and X_2 having as defining condition *X.EPCProdCode=ProdCode and X.size()=200* and source the X_1 stream. X_1 can then be pushed to the source of `Readings` stream, if possible.
- **Optimization 4.3 - Sources-sets mapping:** In some cases, we may have queries with a large number of associated sets, or multiple queries with a significant number of associated sets defined in each. Given a tuple s of a data source S, it is important to quickly locate the associated sets this tuple affects (Line 2). Besides simple data source-associated sets mapping techniques, we can also build a query index scheme [113] based on the defining conditions and data stream sources, to continuously determine which associated sets must be evaluated.
- **Optimization 4.4 - Base-table indexing:** One can analyze the defining condition of an associated set and build appropriate indexes in order to quickly locate matching rows

of the base table, avoiding thus the full scan of Line 3. For example, all example queries (Section 4.4.2) could benefit from the existence of a hash index on `ProdCode` on the associated sets, to quickly identify the matching instances to the incoming `EPCProdCode`.

- **Optimization 4.5 - Data structure selection:** An associated set is a collection of multisets. The representation of associated set instances is a major issue and contributes significantly to the performance of query evaluation. So another optimization is to use data structures to most appropriately represent associated sets, based to declared aggregate functions and methods. For example, if we want to compute the running max quantity of each product, an infinite multiset is required for each instance. Of course, such queries are never implemented as such, since a single value for each product suffices. However, this should be an optimization issue and not left to the semantics of the aggregate function. Another example is the computation of a min (or max) value of a sliding window (i.e. size( )<>0). The most appropriate representation of the associated set instances is a circular queue with a min (or max) tracking algorithm implemented (keeps the minimum of the queue and checks at dequeuing time whether the deleted element is the minimum). A rule-based approach seems appropriate for such data structure selection.
- **Optimization 4.6 - Scheduling:** One can think associated sets as containers (or object instances) that are sent to different data sources in a distributed stream environment. The computation takes place locally at the stream source. However, there are many open issues that should be investigated (rate of updating results at the coordinator, distributed architecture, information to be sent, synchronization, etc.)
- **Optimization 4.7 - Parallelism:** Each associated set can be assigned to a different thread, run on a separate processor of the same node, or even on different nodes. In that case, parallelism can be achieved in query evaluation. Additionally horizontal portioning is possible to parallelize the computation of associated sets: the base relation R can be horizontally partitioned and each partition processed separately. Formally if $R = R_1 \cup R_2 \cup ... \cup R_k$ then $Assoc(R, S, \theta) = \cup_{i=1,...,k} Assoc(R_i, S, \theta)$.
- **Optimization 4.8 - Eager Evaluation:** We assume that a stream source S is able to indicate the end of stream (EOS) event and the engine can detect it - this could be implemented via time-outs. In that case, associated columns corresponding to associated sets of source S can be evaluated and become traditional relational columns. In some cases this can be done earlier and for selected rows, if we can deduct that steam tuples further in the stream will not affect references of these rows. This is usually the case with expressions involving temporal conditions [16][142]. For

example, assume that you have a set of package IDs stored as a relation and a sequence of RFID readings *(package_id, product_id, timestamp)* meaning that product *product_id* has been placed to package *package_id*. We want to monitor the number of products per package. Since packaging takes place in consecutive packages, we know that if a package's window instance has not been updated for a time period exceeding a threshold to, then this package can be considered "completed". In this case the system can replace the associated column value of that row with the actual value, although the `Readings` stream has not reported an EOS. One can periodically issue insert/delete SQL statements to store or delete these "completed" rows (e.g. *delete from ProdsInPacks where X_count_all >= 0* - the *where* clause evaluates to FALSE only for associated-valued columns).

4.7.3 COSTES System

COSTES is a C/C++ system tool able to execute continuous queries using the proposed SQL syntax described in 4.6.1. Figure 4.1 shows the main COSTES components:

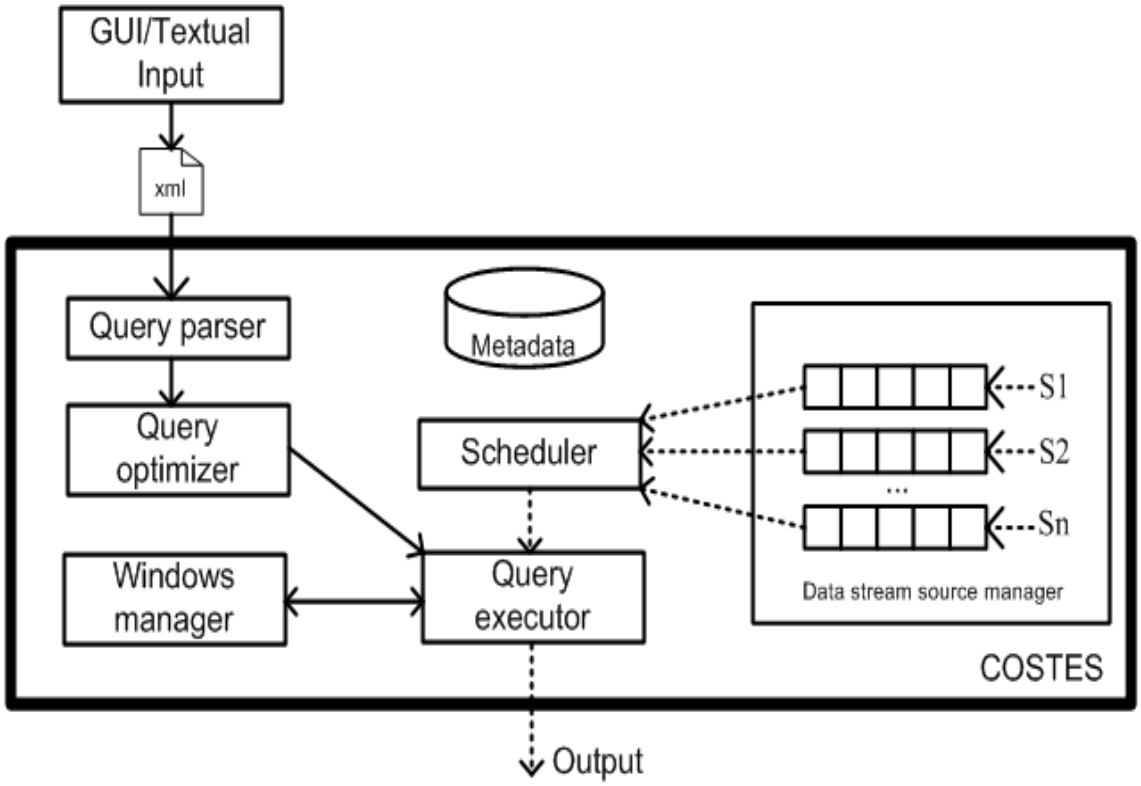


Figure 4.3: COSTES system

Below we give a brief description of each module:

- **Query input:** provides an interface where users can declare and manage COSTES queries. Users can use a textual interface to write the query or formulate it using graphical components. Additionally in this module users declare data source parameters (e.g. schema, type etc). This module generates an XML file, which contains an intermediate representation of the query.
- **Query parser:** validates the query and store information in the metadata catalog. For example such information is base table schema, data source names and schemas, associated sets characteristics and aggregate functions. The metadata catalog is used by other modules during query execution.
- **Query optimizer:** syntactically analyzes the query and consult metadata catalog in order to identify possible optimizations, as described in subsection 4.7.2. Optimization details are stored as metadata and used by the query executor.
- **Data stream source manager (DSSM):** provides functionality for handling data stream sources. Currently our system supports as data sources: flat files, databases (ODBC), web sources (HTTP protocol) and XML sources. DSSM is a multithreaded module allowing concurrent data retrieval from many data sources. To allow synchronized data retrieval, DSSM creates a FIFO queue for each data source and provides push and pop functions for each queue. DSSM obtain information for data sources from the metadata catalog.
- **Scheduler:** retrieves stream tuples from DSSM queues in a round robin fashion and forwards them to query executor. Moreover, user can prioritize stream tuple forwarding between data sources.
- **Associated sets manager:** build and handle suitable data structures for associated sets. Currently our system supports logical and time windows through appropriate method declaration in the `such that` clause.
- **Query executor:** implements the evaluation algorithm (Algorithm 4.3), taking into consideration optimizations that have been made by the query optimizer module. Query Executor allocates the initial base table, builds evaluation trees for the defining conditions and creates index structures according to optimization parameters. It interacts with the scheduler to receive stream tuples and with the associated sets manager to access (insert/delete) the implemented data structures.

They have implemented two versions one console based and a Graphical User Interface (GUI) version. Figure 4.2 shows the definition of associated set X of query Q1 using COSTES graphical interface and Figure 4.3 shows the query results.

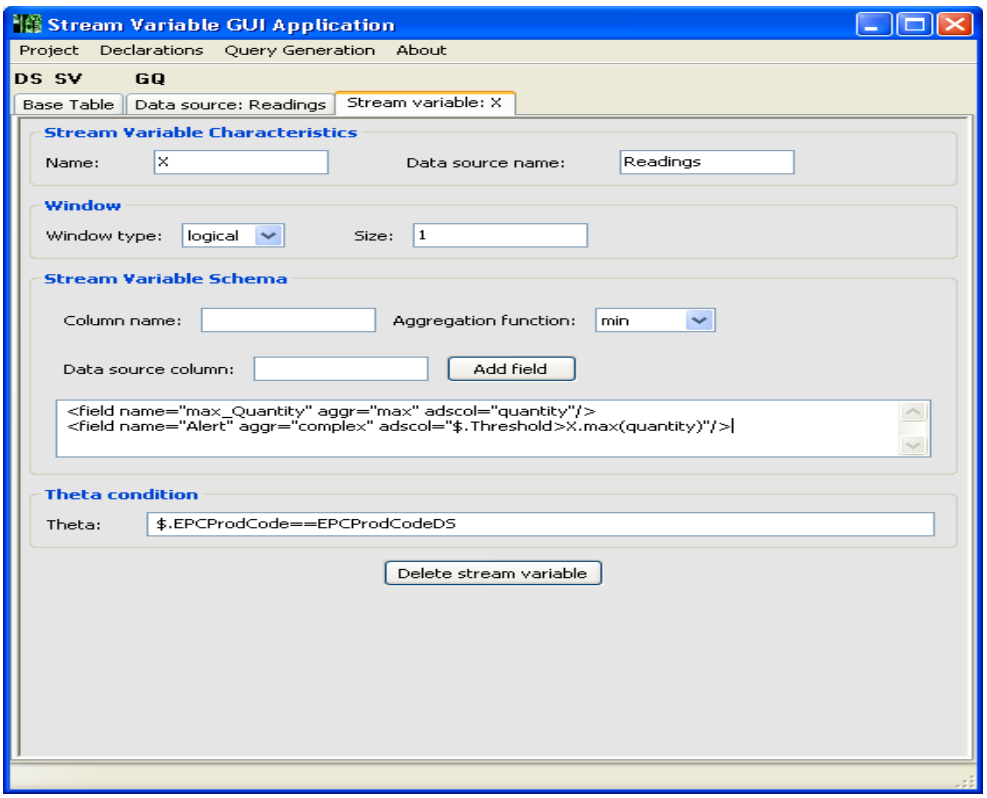
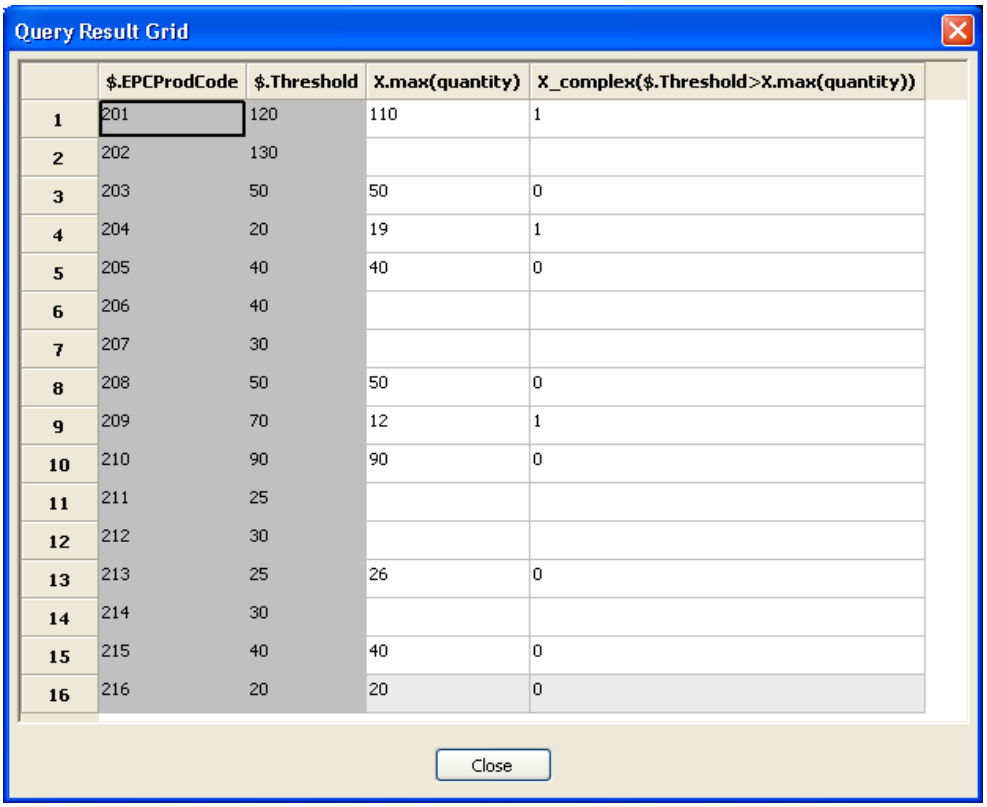


Figure 4.4: Query Q1 definition



	$.EPCProdCode	$.Threshold	X.max(quantity)	X_complex($.Threshold>X.max(quantity))
1	201	120	110	1
2	202	130		
3	203	50	50	0
4	204	20	19	1
5	205	40	40	0
6	206	40		
7	207	30		
8	208	50	50	0
9	209	70	12	1
10	210	90	90	0
11	211	25		
12	212	30		
13	213	25	26	0
14	214	30		
15	215	40	40	0
16	216	20	20	0

Figure 4.5: Query Q1 results

4.7.4 Experiments

We conducted several experiments to measure the efficiency and scalability of our system. Our tests were performed on a Pentium M 1.6GHz with 1GB main memory running Windows XP. In all tests we used flat files as data stream sources, since we were primarily interested in measuring the stream rate that our system could handle – the stream rate of the actual configuration was quite low. In all the experiments the base table has 5000 distinct values (i.e. 5000 distinct product codes) and the incoming stream contains pseudo-random tuples, based on actual measurements. We run three experiments, varying the size of the input stream, the size of the associated sets and the number of the data sources. The experiments are described below:

Experiment 4.1. In this experiment we measured the completion time of query Q1, varying the input stream size from 20,000 to 100,000 tuples having a step of 20,000, with and without Optimization 5.4. Results are shown in Figure 4.6

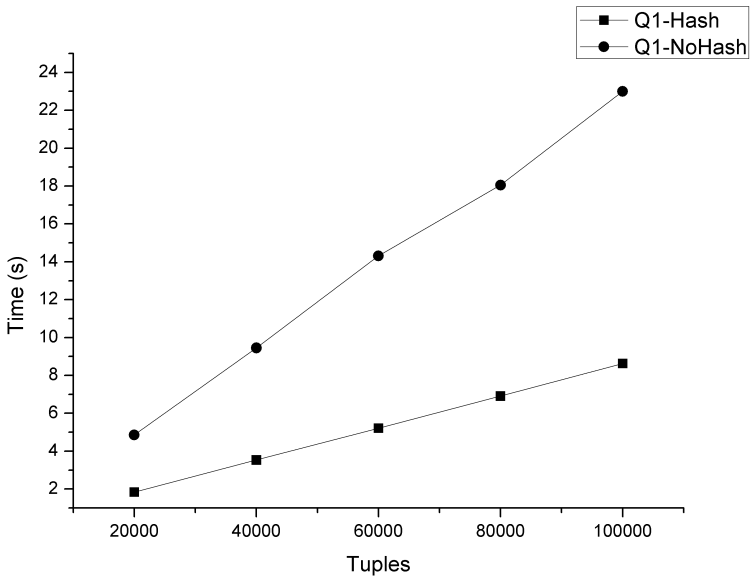


Figure 4.6: Query Q1 execution time varying the number of tuples

Q1-Hash line shows the performance of our system with a hash index built on the base table's attribute `ProdCode`, since the optimizer has identified the (`X.EPCProdCode = ProdCode`) term in associated set *X*'s defining condition. To measure the non-indexed performance (Q1-NoHash line), we forced executor not to build the hash index and proceed with a naive evaluation, i.e. a full scan of the base table for each incoming stream tuple. As expected, the former evaluation plan performs much better than the later (a factor of 2.5). Currently, COSTES system only supports single hash indexing, identified by equality predicates in conjunctive conditions. Other

indexes can be support for other type of queries (e.g. range queries), multi-query optimization techniques and query indexes [113]. Another observation is that COSTES system can consume about 5000 stream tuples per second, applying unoptimized executions on similarly sized base tables. While this is a relatively good number, given that COSTES queries are specified and executed within a fully runtime environment (i.e. they are not compiled), it also shows that there is significant overhead within our system's components. Although our system performs linearly proving that it scales well as the number of stream tuples increases.

Experiment 4.2. In this experiment, we used a variant of query Q1, where a "true" value is reported if the product's quantity on the shelf has reached a critical threshold within a sliding window of size $n > 1$, i.e. associated set X has $size()$ greater than 1 (the minimum value of this set is computed). We measured the performance (execution time) ranging X's size from 100 to 500 with a step of 100 and an input stream of 40.000 tuples, with and without applying Optimization 4.5. In both cases, there is a hash index built on the base table's attribute `ProdCode` (Optimization 4.4). Results are shown in Figure 4.7

. Q1-NoDSS line shows the performance of COSTES system with no special data structure selected for the implementation of the associated set X (a plain queue is used). There is an increase in performance compared to Q1-Hash of Experiment 5.1 due to the management overhead of associated set X (is not a single value any long) and the linear search within the queue to locate the minimum element. Again our system performs linearly but the stream rate is significant lower compared Q1 due to X's management overhead. This gap widens as the size of the queue increases from 100 to 500. Q1-DSS line shows the performance of COSTES system with a circular queue equipped with a min tracking algorithm for the representation of the associated set X. While there is an increase in performance compared to Q1-Hash of Experiment 5.1 due to the management overhead of associated set X (as before), the amortized cost to find the minimum element within the queue is constant. As a result, the completion time remains the same as the queue size increases.

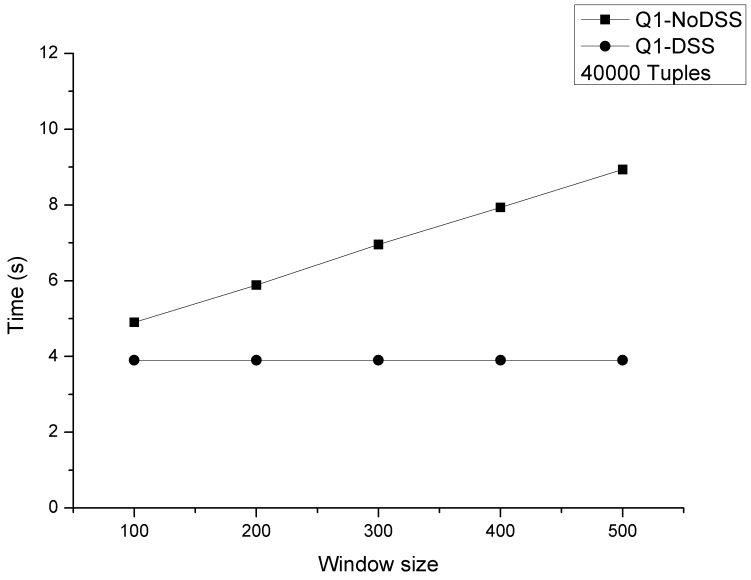


Figure 4.7: Q1 execution time varying the window size

Experiment 4.3. In this experiment we measured the performance of query Q3 varying the number of data sources from 1 to 100 with a step of 20. We can have a large number of stream sources in the presence of multiple stores and/or promotions. The input stream size was 40,000 tuples. Results are shown in Figure 4.8. The incoming tuples were evenly distributed among associated sets. We assume equal arrival rate for all stream sources and one associated set defined for each data source. The optimizer builds a hash index on base table's attribute `ProdCode` (Optimization 4.4). There is a small increase in completion time due to the additional queues that the DSSM module maintains for each data source. However, as the number of queues increases, the size of each queue decreases and the data is consumed quickly by the scheduler. The construction/destruction of queue objects is the main reason for the additional performance overhead. In general, the figure shows that multiple data streams can be handled efficiently by COSTES system.

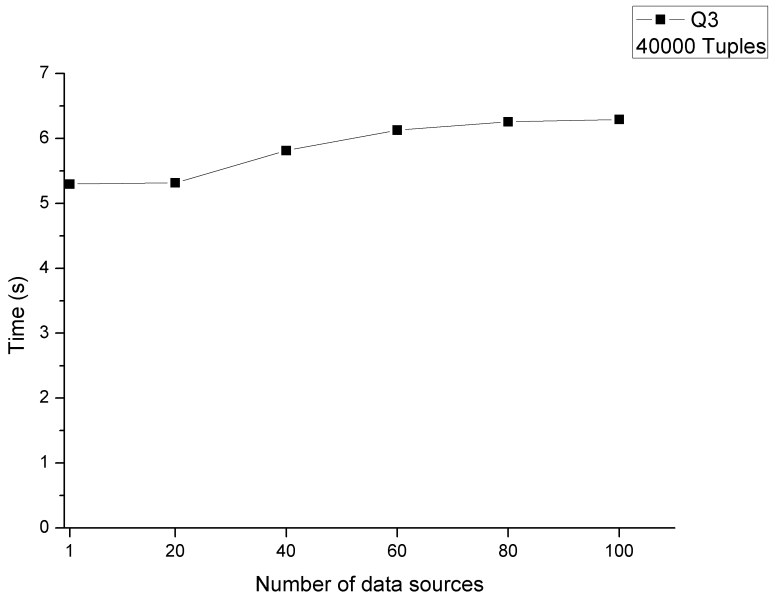


Figure 4.8: Query Q3 execution time varying the number of data sources

4.8 Summary and Conclusions

The RFID technology has been widely praised for its ability to streamline supply chain processes. This is achieved from its unique data capturing characteristics to support real-time decision making. Being able to efficiently perform complex real-time analysis on top of RFID event streams is a key challenge for modern applications. This provides management with a novel data analysis mechanism to allow better, tactical, on time, well-informed decisions. The two main issues in RFID data management (RFDM) concern expressibility (how to simply and concisely express stream queries) and performance (how to efficiently evaluate stream queries).

In this chapter we described a decision support system incorporating a simple and powerful extension of SQL to express spreadsheet-like continuous computations. We argued that such an extension is particularly useful in RFID data management and presented it in the context of real-time supply chain decisions. However, this extension can be useful in other data stream application domains, such as analysis of financial streams. Finally we presented a fully functional prototype that implements this extension in a user-friendly and efficient manner.

Overall, the proposed decision support system links continuous RFID data streams with product information stored in a relational database in order to support real-time supply chain decisions. This decision support system is applied in a distributed system architecture that enables information sharing between retailers and suppliers in order to enable real-time decisions in an open supply chain environment [17]. While current RFID deployment efforts mainly deal with integrating RFID-based relational databases with existing legacy/ERP systems, this research

moves beyond these efforts in supporting continuous queries and real-time decisions. It further contributes to transforming RFID data streams into meaningful real-time information, thus unveiling the information potential of this technology and justifying RFID investments for different supply chain partners.

Chapter 5

An Integration Framework for Relational and Stream Systems

5.1 Introduction

Many practical applications [27] need to process continuous flows of data in real-time. Well-known stream applications involve sensor networks, RFID product tracking, network and environmental monitoring, smart grids and others. In the era of Big Data, a wide range of analytics applications need to combine persistent and stream data in a simple and efficient way. For example, situational awareness [29] is a term used to describe the capability of an organization to become aware of what is happening in its immediate business environment and how internal or external events affect organization's daily operations. Situational aware applications require the collection of information from multiple data stream sources. These stream data must be combined with persistent data for analytic purposes.

The need for processing different types of data has led to the development of multiple and diverse systems. In the case of data streams, processing can be carried out by generic stream engines, standalone stream-handling components or custom stream applications. In the case of persistent data, relational technology has proven itself for reliably managing data for many decades now. In addition, SQL is a standard language for querying data with a wide acceptance from the IT industry and with a large base of knowledgeable users and resources. As a result, relational databases have a large market share and will continue to be used extensively, according to our view. As stream data will become more available and common, it will be important for database users to easily integrate it within the schema and transparently use it in their queries. However, there is no standard query language for streams [84] and as a result each stream system has its own features and specifications. This issue creates a three-fold problem for database users: First, they must learn a new system from scratch if they want to process and query stream data. Second, they must find a way to combine relational and stream data. Third, they need a query language to query integrated data.

In this chapter we propose a view layer defined over standard relational systems to handle this mismatch. DBAs define a special type of views (called LinkViews) which combine relational data and stream aggregates. The columns of a LinkView are either columns of the relational schema (called *base* columns) or "placeholders" to be filled-in with stream values whenever necessary (called *linked* columns). As far as naive SQL users is concerned, a LinkView can be part of an SQL statement as any other relational view. We define a Key-Value interface between relational and stream systems and an Application Protocol Interface (API) is proposed for the exchange of data (send keys, retrieve values). Our API follows a web-like approach where a web server executes programs/scripts using the parameters that receives from clients (web forms). Parameters in our case are distinct values (keys) found in the base columns of a LinkView. This is the common case in most operational business environments, as most of the time the analytic queries that utilize stream data use an identifying "persistent" value such as a tag ID, a location ID, a stock ID, or something else that is usually stored in a traditional database. Our goal is to create a framework that allows database users to be completely unaware of the implementation details and inner workings of stream systems but be able to use stream aggregates (i.e. the results of stream queries) in their database systems.

5.2 Motivation and Issues

In this subsection we provide a motivating example and describe the issues that must be addressed. In addition we present a wide variety of miscellaneous applications where our proposed framework can be used. Finally we provide several examples that will be used throughout the rest of sections.

5.2.1 Motivating Example

A financial firm maintains in a relational system historical data on stock performance (opening and closing prices, variations, volumes, etc.) At the same time, it has access to two systems (e.g. Reuters and Bloomberg), lets call them A and B, that provide real-time information on stock prices and volumes respectively. These systems A and B could be anything, for example a SQL-based DSMS such as STREAM [12] or a Java-based component using sockets. Analysts would like to utilize in their relational queries real-time data (e.g. the running average price per stock) in a stream-transparent way, i.e. without knowledge either of the presence of stream systems or the continuous nature of the stream data. For example, to select those stocks that their previous day's closing price is greater than their current running average price, one would like to write a SQL query similar to the following:

```
select      stockID
from        Historical H, Prices P
where       H.stockID = P.stockID AND
            H.closingPrice > P.price AND
            H.date = date() - 1;
```

`Historical` is a relation containing historical data for the stocks and `Prices` is a relation with schema `(stockID, price)`, where price is the running average price of the `stockID`. Whenever `Prices` is used within a query, `price` column is updated with the current value. While the transient nature of column price does not comply with the relational model, the "evaluate-whenever-used" approach reminds relational views, which are evaluated whenever used within a query. Let us now make `Prices` a little bit richer in information, by adding a couple more of these transient columns. The new schema would be:

```
Prices (stockID, price, price10, volume)
```

where `price` is the running average price, `price10` is the average price of a 10-minute sliding window and volume is the running total `volume` of the `stockID`. While column `stockID` is a column found in the relational schema, columns `price`, `price10` and `volume` do not involve relational data and represent aggregate values over stream data found in systems A and B. Figure 5.1 depicts the idea.

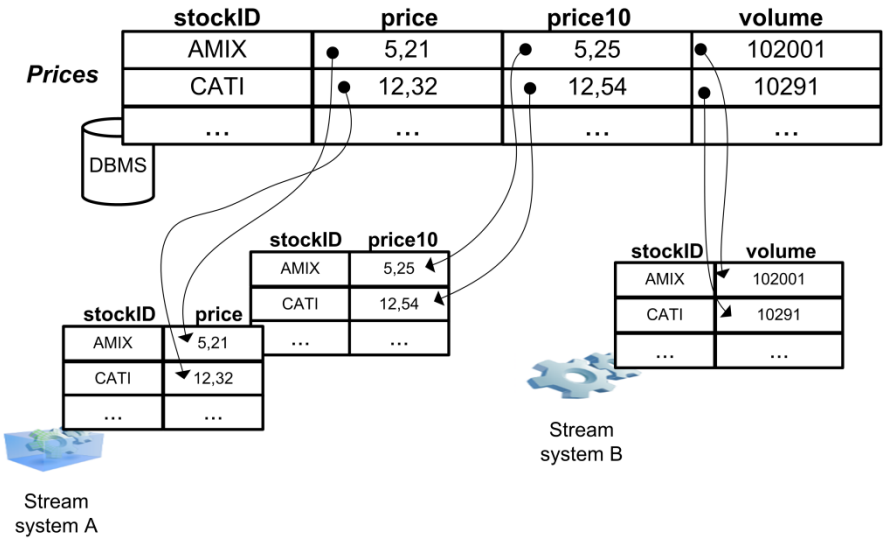


Figure 5.1: Prices view and abstract representation of the linkage with stream systems

This figure suggests that the stream structures are handled by the stream systems, while the relational system has access to these structures through a *handler*, which returns a single value to be placed into the column. While this concept is simple and quite common in interoperability and collaborative systems, there are certain subtle issues that have to be addressed – and decided. These issues are described below.

A. Stream structure definition

The first issue has to do with who provides the definition of these stream structures, the DBA or the stream programmer (stream system expert)? In the first case (DBA), this definition should be part of the SQL defining statement of the LinkView and all stream systems should adhere to, i.e. be able to map to their own native language. For example, Prices view, following syntax similar to to those given in Chapter 2 and 3 of the current thesis ([40][41][45]) would be:

```
create view Prices as
select stockID,
       X.avg(price) as price,
       Y.avg10(price) as price10,
       Z.avg(volume) as volume
from stocks
extended by X(PriceStream), Y(PriceStream), Z(VolumeStream)
such that X.stockID = stockID,
          Y.stockID = stockID AND Y.size() = 10,
          Z.stockID = stockID
```

`PriceStream` and `VolumeStream` are stream sources that provide real-time data (stock's `price` and `volume`) to stream systems A and B respectively. The idea is that for each `stockID` we define three sets of stream values *X*, *Y* and *Z*, according to their "`such that`" condition. For example, the "`Y.stockID=stockID and Y.size()=10`" condition defines that for each `stockID` the stream system must define a set `Y` which keeps the price values in a sliding 10-minute window and calculate the average price, `Y.avg10(price)` which becomes column `price10` in the output of the query. In this case, stream systems A and B must receive the SQL definitions of *X*, *Y* and *Z* and implement them in their native stream language.

The drawbacks of this approach are: (a) the case specific stream processing that applications frequently require (e.g. different window types, peculiar pattern matching, exception handling), (b) complexities involved in mapping to specific systems and languages, and (c) acceptance of the proposed SQL syntax from the stream systems community. For example, while mapping could be simple for STREAM [12], it might not be so simple for Aurora [2] or Java programs. In addition, the DBA should be aware of the stream schema, something that is not always possible. Moreover, stream sources can change dynamically and as a result the declared views may become invalid. This tight coupling between DBMSs and stream systems makes view composition and evaluation hard.

In the latter case, stream programmers define the required stream structures through programs written in the native language of their system. For example, if system A is STREAM, a CQL (Continuous Query Language) [14] statement should be issued to compute the running and window-based average prices. These programs should be able to have access to a set of input keys, supplied by the DBMS. In Figure 5.1 we see that the stream structures of systems A and B are aware of the `stockID` keys, sent by the DBMS. Symmetrically, the DBA should be aware of the name of the programs that implements these stream structures, so s/he uses them in the SQL definition of the LinkView. A web-like paradigm, where the application logic programmer provides a program P to HTML form authors – who are completely unaware of how P manipulates form parameters – becomes very suitable. DBAs are informed by stream programmers of the name of the program(s) that handle stream structures. When a LinkView is initiated, these programs execute, receiving the distinct values of the appropriate base columns as input (the keys). Each program maintains a stream structure and computes an aggregate value for each input key. For example, `Prices` view in this case could be defined as:

```
create view Prices as
  select stockID
  from stocks
  using stockID
    exec program A.P1() for price
    exec program A.P2() for price10
    exec program B.P3() for volume
```

$P_1()$, $P_2()$ and $P_3()$ are programs written by stream programmers of systems A and B implementing stream structures. The `using` keyword defines the base column (`stockID`) whose

values will be used as input parameters to programs P_1, P_2, and P_3. We assume that programs receive the complete list of stockIDs and return results for all stockIDs. Each program match the input keys with values (stream aggregates), in a key-value structure, which then becomes a new column in the view (price, price10, volume). These key-value structures correspond to the stream structures mentioned above. Note that DBAs must only know the program names to define the view (in the previous approach DBAs needed to know the actual stream schemas). Under this perspective the analogy is: one program computes stream aggregates for the keys supplied by the database. These stream aggregates are all of the same type i.e. the average price of a 10-minute sliding window (price10). However this is not an optimal design pattern for stream programs as we force stream programmer to write one program per stream aggregate. Computing multiple aggregates in one program gives better optimization possibilities: sharing database data (keys) and sharing stream aggregation computation [76]. In addition there is less overhead for stream systems as we have less number of programs. So, a better approach is to let stream programmers implement one or more stream aggregates in one stream program. The analogy is: one program implements one or more stream structures and each stream structure contains stream aggregates of the same type (e.g. avg). Each stream structure is tagged with a label that can be used in view declaration. In this way DBAs must know only the label and not the stream structure schema to access a stream aggregate. The label is useful for DBAs as they can choose which stream aggregate will use in a LinkView in case the program provides multiple aggregates. Also the label is useful if for example the stream programmer decides to extend the program to compute another stream aggregate. The label of old stream structures remains the same so the view definition is still valid. In general the stream program hides the schemas of stream sources and the actual implementation of stream aggregates while the label hides the schemas of stream structures. Stream programmers implement stream programs based on what aggregates they want to provide to database users but they do not know the exact key data i.e. they know only the key semantics and not the actual values. For example the same stream program can be used for all or only a part of stockIDs (e.g. only belonging in a category).

B. Query processing

Once Prices view has been defined, it can be used by naive SQL users as any other view in the DBMS. When used in an SQL statement, the key-value structures corresponding to columns price, price10 and volume must be accessed to materialize Prices, which then participates in query processing as any other relation. Doing that efficiently presents a number of challenges. For example, a query may ask for the stockIDs having price > 10. Apparently, the key-value structures of price10 and volume are not necessary and Prices view will only

be partially materialized. In addition, the condition (`price > 10`) can be applied either directly in the key-value structure of price or later, during query processing of the SQL statement in the DBMS.

C. Stream programs execution: The third issue has to do with the implementation model at the stream system i.e. how stream programs actually work. There are two evaluation approaches for a stream structure:

1. A stream program executes for each key
2. A stream program executes for the entire keys' set

In 1st case we have as many stream programs as the number of keys. Each stream program contributes to the stream structure supplying the computed stream aggregate for the specific key. In 2nd case we have one stream program that computes a stream aggregate for each key and assign the results to the stream structure. As most stream systems enable optimizations (e.g. sharing keys and aggregation computation) for the 2nd case we choose this approach. Additionally having less stream programs is more appropriate for performance reasons. There is an advantage of the 1st approach: we can compute different stream aggregate functions per key if we pass a parameter to stream program to differentiate stream program's executions. Our theoretic framework (Subsection 5.4) is valid for both cases.

All these issues must be carried out though a well-designed API between the DBMS and stream systems.

5.2.2 Example Queries

We provide some examples containing LinkViews. The first part of them is describing LinkView definition and the second one SQL queries that contain LinkViews.

Example 5.1 - Using LinkViews in SQL queries: The motivating example (Subsection 5.2.1) uses the `Prices` view as shown in Figure 5.1. We repeat `Prices` here.

LV1. `Prices(stockID, price, price10, volume)`

`price` and `price10` are computed from stream system A, while volume from stream system B. An example SQL query using LV1 is:

Q1. Find those stocks that their previous day's closing price is greater than their current running average

Example 5.2 - Stream programs with parameters: A stream program may have parameters. For example, instead of having two programs for columns `price` and `price10`, one can write a program that gets as parameter the size of the sliding window (in minutes), with size=0 meaning a running average. In this case, there will be two execution instances of the same program. In both cases `Prices` schema is the same, but the LinkView definition is different.

LV2. `Prices2(stockID, price, price10, volume)`

Stream programs with parameters can be used for a wide range of tasks (provide filtering conditions, thresholds, window sizes, select which aggregate to output, and many others).

Example 5.3 - Querying LinkView's stream columns: We assume the following view:

LV3. `MinMaxPriceCategory(categoryID, minPrice, maxPrice)`

Stocks belong to several categories, identified by a `categoryID`. For each category, `minPrice` (`maxPrice`) is the current minimum (maximum) price of the category's stocks. Both aggregates are computed by a single program in a stream system. Some SQL examples using LV3 are:

Q2. Find the `categoryIDs` having `minPrice` greater than 10

Q3. Show the `categoryIDs` and the `maxPrice` for categories that have `minPrice` above 10 and `maxPrice` below 12

Example 5.4 - LinkViews with multiple base columns: we assume the following view:

LV4. StockCategoryPrice(stockID, categoryID, price, categoryPrice)

`stockID` and `categoryID` are columns drawn from the database. `price` and `categoryPrice` are the running average prices of the stock and the category respectively. These are computed by two distinct stream programs in a stream system.

Example 5.5 - Saving LinkView snapshots: sometimes it is useful to store in the database system an evaluated snapshot (a LinkView with all columns filled with values) of a LinkView for future use. For example a useful query is:

Q5. Daily, at a certain time, store the closing price of stocks.

The closing price is provided by a LinkView and it can be stored in a database table with a SQL insert-into-select query. This query can be scheduled to run automatically daily at a certain time.

5.2.3 Miscellaneous Applications

Stream platforms are becoming essential for many organizations due to the usefulness of stream applications and the always increasing volume of stream data. As a result, in the following years stream processing software will be used to a larger extent by organizations. Moreover stream service providers [98] can provide stream processing services to organizations. The integration of database systems currently available in many organizations with stream platforms and applications will allow analysts to have on-time information and make timely and efficient decisions. Our proposed framework enables analysts to use their relational database systems for the integration of stored and stream data. Below we describe some representative applications:

- **Radio-Frequency Identification (RFID):** technology enables the automation of several applications as inventory control, asset management and product tracking. Such applications are very common in supply chain environments [40][41]. A supply chain is a system involving people, processes, equipment and has as main goal the movement of products from suppliers to stores. RFID technology is used to provide real-time information about products allowing analysts to get better decisions. For example a useful query for a supplier is: “What is the current inventory of supplied products on stores A and B?” In this scenario each store uses RFID tags in its warehouse and a stream system monitors the product’s stock level. With our integration framework the

stream system can receive a list of products from the supplier and return the current inventory per product. Suppliers' analysts can compare real-time information with historic data existing in their databases like product sales predictions or average delivery time in order to optimize the distribution process of their products.

- **Social networks and web advertising campaigns:** companies launch online campaigns to promote their products [30]. Such campaigns make use of web and social media advertisements. A useful query for campaign applications is to see how well the campaign performs by comparing current sales per product with the average sales of an older but similar campaign. Additionally analysts may want to correlate this information with the click through rate of ads (the number of clicks of an ad divided with the number of times the same ad is shown) for each product while taking into consideration users' product reviews from social networks. Such a query requires stored and stream data from multiple systems. Product information, advertisement details, sales and historical data about previous or similar campaigns are stored in a local database while the web and social media sites run stream systems able to handle customers' click streams. Such analytics queries that contain historical and real-time data are very useful for web campaign evaluation and can be supported easily by our framework: database users send the keys to the stream systems and retrieve the available information.
- **Financial data analysis:** In many financial applications a small number of streams (e.g. NASDAQ stock price and volume streams) are used by a large number of financial analysts [76][148]. Each analyst uses the stream for his own analytic tasks and has different requirements from the stream system. For example, a financial analyst may use in his trading strategy an average price window for the last 10 minutes while another financial analyst may want to find the average price only for the last 1000 prices. Analysts want to combine this real-time information with historical or other useful data stored in their databases. This scenario can be handled easily by our proposed framework: analysts create views that compute stream aggregates by using programs provided by stream platforms. Those programs may receive parameters that define the computation behavior of each program i.e. the last 10 minutes or the last 1000 values. Then, they can execute SQL queries over these views.

5.3 Challenges and Contributions

The main goal of this work is to integrate stream aggregates, possibly from different stream systems, within a relational framework. The challenges are:

- **Sound relational semantics.** What are the appropriate constructs and semantics to integrate relational and transient data in a theoretically sound way?
- **Simplicity and stream-transparency.** We want to impose minimal changes to the relational system, have simple and intuitive syntactic constructs to define linking between stream and the relational system, and completely hide the stream presence from end SQL users.
- **Efficiency and scalability.** Modern applications require the collection and analysis of stream aggregates produced by systems very different in nature. In addition, today's relational data sets can be humongous. Query processing must be done efficiently - if possible in a distributed and parallel fashion – and stream systems should be easily and quickly added to the integration framework.

Our work contributes to a rather uninvestigated research area that deals with the integration of relational systems with heterogeneous stream systems [132]. The goal is to standardize the way a relational system interacts with several stream systems. Specifically, the main contributions of this work are:

- **LinkViews and linked columns.** We propose a new kind of view, called LinkView, where some columns are materialized with relational data and some columns are populated by external systems through a well-defined and efficient API. We argue that the semantics of LinkViews are relationally proper and can be implemented on top of any relational database system. Finally, end users can use LinkViews in their SQL statements as any other view.
- Key-value-based interfaces and web-like protocol. By introducing a key-value interface between DBMS and stream systems, we adequately handle scalability and efficiency issues. In addition, some query processing can be delegated to the key-value stores (e.g. pushing selections or even joins.) By allowing LinkViews to specify stream programs to execute at the stream system's side, we offer a clean distinction between DBAs, naive database users, and stream programmers. An approach similar to HTTP request and response protocol, but with the concept of sending keys and getting values is provided.

- **Address an overlooked class of queries.** Discuss a class of queries that hasn't been properly addressed in the past: ad hoc queries using stream data in a database-oriented (pull) fashion.
- **Prototyping.** We have built a prototype system over PostgreSQL, integrating with C/C++ programs managing synthetic stream data, to serve as a proof of concept.

5.4 LinkView Semantics

In this section we formally define LinkViews and describe query processing when LinkViews are mentioned in users' queries. We also provide LinkViews implementation semantics.

5.4.1 Rationale

We would like to extend view definitions with columns that contain aggregates of stream data, as highlighted in Section 5.2. However, to have proper relational semantics, these values should not change over time. To overcome this difficulty, we use the concept of pointers, as in programming languages. While the name of a pointer remains the same over the time of an execution, the contents of the object it points to may change. In addition, these objects may reside in different systems, which facilitates the distributed, heterogeneous nature of modern stream systems. When a LinkView is mentioned within a user's query, it has to first be evaluated and then used in an evaluation plan. Of course, there are optimization rules than can be applied.

5.4.2 LinkView Theoretical Definitions

Definition 5.1 (Containers) Given a domain D, a *container* S over D is a named object that points to some subset of D (using multi-set semantics). The latter is called the *contents* of the container. □

Essentially, this definition provides for *named* objects for stream data. D is the domain of the stream data (e.g. prices, volume). The contents of a container may change, while the name remains the same, as in regular pointer semantics in programming languages.

Definition 5.2 (Links) Given a domain D, a container S over D and an aggregate function f: $Pow(D) \rightarrow N$, where $Pow(D)$ is the power set of D and N is the set if all permissible outputs of f, then the pair (S, f) is called a *link* L over D. The *value* of the link, denoted as $val(L)$, is defined as the return value of f when applied over the contents of container S. S is called the container and f is the aggregator of the link. □

Note that although a link remains unchanged over time, it may evaluate to different values at different times, since the contents of the container may change over time.

Definition 5.3 (Linked Column) Given:

- a materialized view V having schema $\mathscr{S} = (A1, A2, ..., An)$,
- a subset A of $\mathscr{S}$, $A = (A_{i1}, A_{i2}, ..., A_{im})$, $i_1, i_2, ... i_m$ in $\{1, 2, ..., n\}$,
- a set of links $L = \{L_k : k \in \pi_A(V)\}$, i.e. a link for each distinct value k in column(s) A,

then we can extend the schema with a column A', where the value of A' at row r is the link $L_{r.A}$, i.e. the link corresponding to the value of column(s) A at row r. A' is called a *linked column* of V. A is called the base column(s) of A'. □

Observation 5.1 A functionally determines A'.

Observation 5.2 Although the value of links may change over time, the links per se remain fixed, providing for proper relational semantics.

Note that this definition is more general than how it used in next sections of this chapter. It states that for each value in A, A' contains a link. These links could have containers over different domains and different aggregators.

Definition 5.4 (LinkView) Any view V, extended with one or more linked columns is called a LinkView. □

5.4.3 Query Processing

The question is how a traditional relational query processor can be modified to handle queries that involve LinkViews. The simplest approach is to define a relational operator, called LV-Eval that gets a LinkView and transforms it to a relation with the links of the linked columns replaced by their values.

Definition 5.5 (LV-Eval Operator) Given a LinkView V with schema $(A_1, A_2, ..., A_n, A'_1, A'_2, ..., A'_k)$, where $A'_1, A'_2, ..., A'_k$ are linked columns, the LV-Eval(V) is defined as a relation with the same schema of V and constructed in the following manner: for each row r of V, we have a row $r' = (r.A_1, r.A_2, ..., r.A_n, val(r.A'_1), val(r.A'_2), ..., val(r.A'_k))$ in LV-Eval(V). □

If a user's query Q mentions one or more LinkViews, these are replaced in the query plan by their respective LV-Eval instances.

5.4.4 LinkView Implementation Structure

We now turn to the proposed architecture. We assume a single DBMS and several *stream management entities* (SME), which can range from complete data stream management systems (DSMS) to simple Java programs, possibly employing different querying paradigms (e.g. CQL [14], operators in a workflow [2][3]). A SME also incorporates integration modules (described below) specified by our framework. The idea is that LinkViews reside in the DBMS, while the contents of link containers and aggregators reside and managed by the SMEs. The DBMS is interested only on the values of the links.

The fundamental question is who defines the links of the linked columns of a LinkView, i.e. who defines the contents of link containers and select aggregators. One approach would be to let LinkView creators (e.g. DBAs) to do so, through a standardized language and set of aggregate functions. Then, these definitions are send to the SMEs to be implemented by the native language of the SME. However, this approach suffers from the drawbacks mentioned in Subsection 5.2.1 (Stream Structure Definition)

The web client/server model where a web server executes a script/program using the parameters provided by a web form (client) is simple yet flexible and efficient. Presentation layer developers (e.g. html designers) are only aware of the server's program name that executes when a form is submitted, along with the names of the parameters that the program handles. On the server side, programs have to be invocable and able to read in the submitted parameters and values, through a well-defined interface. The output of these programs is directed back to the browser. This simple model seems appropriate also for our case: LinkView creators only know the stream program's name at a SME, responsible for managing the links of a linked column; this program should be able to obtain the values of the base column(s) of the linked column (the keys), since it has to use them during its execution and associate the keys with values (the values of the links); the DBMS should be able to retrieve these values. All these are achieved through a request-response API between the DBMS and the SMEs. Note that the program executing at the SME could be written in any programming formalism (CQL [14], JAVA/C++, etc.). Also, its semantics

is completely transparent to the database person – s/he does not really know how the task is carried out.

Definition 5.6 (Implementation Structures) Assume a LinkView V with schema $(A_1, A_2, ..., A_n, A'_1, A'_2, ..., A'_k)$, where $A'_1, A'_2, ..., A'_k$ are linked columns. For each linked column A'_i we define a quadruple $Q_i = (S, P(c_1, c_2, ..., c_m), A, KV)$, where:

- S is a stream management entity,
- P is a program that executes within S and process data streams. $P(c_1, c_2, ..., c_m)$ is a specific invocation of P with parameter values $c_1, c_2, ..., c_m$,
- A is a named output value of P, and,
- KV is a Key-Value structure, where the keys are the values of the base column(s) of A'_i.
- Q_i is called the *implementation structure* of linked column A'_i. S is the *source* of A'_i, P is the *execution* of A'_i and A is the *label* of A'_i. □

An implementation structure describes the implementation details of a linked column. The source S is the SME that provides the stream data for the link containers of the linked column. P is a program that resides in and can be executed within S, e.g. a CQL statement. This program is responsible to maintain/manage the link containers and (continuously) produce their values according to the aggregator. To do so it has to have access to a Key-Value structure KV, where the keys consists of the values of the base column(s) of the linked column. P uses the keys to define the contents of link containers and the associated values to place the output of the aggregator. In practice, P does not have to maintain link containers or apply aggregators, this is at the conceptual level. The only requirement for P is to access the keys of KV and set the corresponding values.

A program P may produce several *named* output values per key, for efficiency and/or reusability reasons. For example, P may be a CQL[14] statement, computing the min, max and average price of a sliding window of size 10 for each `stockID`. It would be inefficient[1] to have three distinct programs to separately compute min, max and average. Since a link evaluates to a single value, the output values of P must be named and the designated output value for the linked column must be specified. This is the label of the linked column. In addition, the provider of the stream data (e.g. Bloomberg, Reuters) may write generic programs with multiple output values to cover several cases of its client's requirements.

[1] To be precise, it is inefficient, unless the optimizer of the stream system is able to apply multi-query optimization techniques (in the case of SQL-oriented systems) or execution sharing (e.g. MR-Share [110] on MapReduce online [54])

Finally, an execution *P* may have parameters. For example, it may implement a sliding window of a specific size. This size could be a parameter of *P*, specified by the LinkView creator during the definition of the linked column. It may be a threshold value, if *P* manages temperature sensors. Or it could be a string representing a filtering condition that *P* applies on the stream data or the name of an aggregate function. In other words, LinkView creators use specific call instances of *P* to define linked columns.

5.5 LinkView SQL Extensions

We propose the following extension of SQL syntax to facilitate LinkView definitions in relational systems:

```
create linkview name as
SQL statement
[using BaseCol link with P(c1, …, cm) of S
    add column L1 as (data-type) A1
    ...
    add column Ln as (data-type) An
]+
```

A `create linkview` statement creates a LinkView database object. It consists of a standard SQL statement, which defines a materialized view, followed by one or more `using` statements.

A `using` statement is used to define one or more linked columns, $A_1, ..., A_n$, having the same base column(s) *BaseCol* – an arbitrary subset of the schema of the materialized view – and sharing the same execution $P(c_1, ..., c_m)$ at stream management entity *S*. $L_1, ..., L_n$ are the labels of $A_1, ..., A_n$ respectively. In other words, each `using` statement defines n linked columns, $A_1, ..., A_n$, with implementation structures $Q_i = \{S, P(c_1, ..., c_m), L_i, KV_i\}$ where i in $\{1, 2, ..., n\}$. Note that the LinkView author has to specify the datatype of the linked columns, since this information can not be retrieved by the SME as stream semantics are completely transparent to database users.

5.5.1 Example Queries

We provide below the syntactic definition of the LinkViews presented in section 5.2.2:

Example 5.1: The definition of LV1 is:

```
create linkview Prices as
select stockID
from stocks
using stockID link with pPrice() of A
  add column priceL as (real)price
using stockID link with pPrice10() of A
  add column price10L as (real)price10
using stockID link with pVol() of B
  add column volumeL as (int)volume
```

LinkView LV1 uses two stream management systems named A and B. The actual connection information (e.g. network address/port) for each system is stored on LinkViews' metadata catalog. Stream system A can invoke executions of programs `pPrice()` and `pPrice10()`. `pPrice()` computes the running average price for each stock. `pPrice10()` computes the average price within a 10-minute sliding window for each stock. System B implements `pVol()` program that computes the running total volume for each stock. The DBMS can use programs' output by referring to the named outputs (labels) of each program. The label for `pPrice()` program is `priceL`, the label for `pPrice10()` program is `price10L` and the label for `pVol()` program is volumeL. `price`, `price10` and `volume` are respectively the names of the linked columns corresponding to these outputs. The data type of each linked column is mentioned right before its name, using parentheses in the `add column` statements.

Example 5.2: The definition of LV2 is:

```
create linkview Prices2 as
select stockID
from stocks
using stockID link with pPriceV(0) of A
 add column priceL as (real)price
using stockID link with pPriceV(10) of A
 add column priceL as (real)price10
using stockID link with pVol() of B
 add column volumeL as (int)volume
```

In this case, stream system A has access to a *parameterized* program named `pPriceV(int size)`, where size denotes the size of the sliding window (size=0 means a running average.) `pPriceV(0)` computes the running average price per stock and `pPriceV(10)` the running average price per stock within a 10-minute sliding window. `pVol` () is the same as in LV1.Note that parameterized executions allow for a wide range of options in terms of functionality. Parameters may involve filtering conditions, threshold values, selecting aggregate functions, etc.

Example 5.3 The definition of LV3 is:

```
create linkview MinMaxPriceCategory as
select categoryID
from categories
using categoryID link with pMinMax() of A
 add column min_priceL as (real)minPrice
 add column max_priceL as (real)maxPrice
```

`MinMaxPriceCategory` LinkView uses `pMinMax()` program of stream system A. `pMinMax()` computes two stream aggregates – the minimum and maximum price per category – and provides its results through two labeled outputs, `min_priceL` and `max_priceL`.

Example 4.4 The definition of LV4 is:

```
create linkview StockCategoryPrice as
select stockID, categoryID
from stocks S
using stockID link with pPrice() of A
 add column priceL as (real)price
using categoryID link with pCat() of A
 add column cat_priceL as (real)categoryPrice
```

This examples simply demonstrates the usage of multiple base columns in the same `create linkview` statement.

5.6 LinkView Architecture

The proposed architecture to support the LinkView integration framework is shown in Figure 5.2. The parser and the LinkView manager sit on top of any DBMS, while Stream Management Entities (SMEs) must implement the Key-Value layer and provide a Key-Value access interface to the application layer that contains the actual stream system.

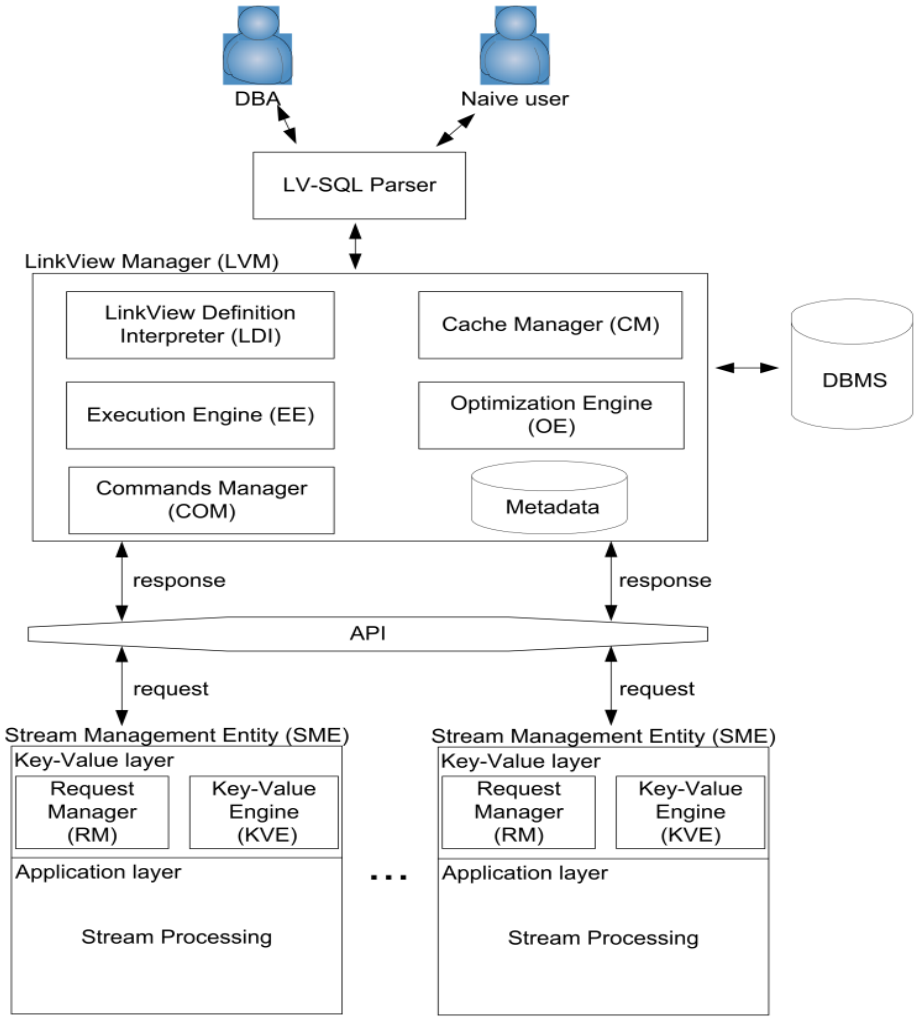


Figure 5.2: LinkViews architecture

Below we describe the system modules:

The LinkView SQL (LV-SQL) Parser: This component is responsible to parse out queries submitted by the users. A query can be either a LinkView definition, submitted by a user with administration privileges (e.g. DBA), or a standard SQL query, submitted by a naïve user. In the first case, the parsed query is passed to the LinkView Definition Interpreter (LDI) subcomponent. In the latter case, if the SQL statement involves LinkViews is directed to the Optimization Engine (OE), otherwise is directed to the database system's SQL component. LV-SQL also includes LinkViews management statements (e.g. init, cache policies) which are passed to the Commands Manager Module (COM)

The LinkView Manager (LVM): The LinkView Manager is the core component of our architecture. It stores metadata for LinkViews and implements the API for DBMS/SMEs communication. The Execution Engine (EE) of LVM is responsible for the execution of SQL queries involving LinkViews issued by naïve users. The Cache Manager (CM) is responsible for storing/caching linked columns, implementing various data refreshing policies, specified by the DBA. That means that linked columns may be refreshed with stream data at regular intervals and query processing may utilize cached linked columns instead of requesting actual stream data. Commands Manager (COM) handles statements for the management of LinkViews (e.g. init LinkView, drop LinkView, setting caching policies, etc)

Stream Management Entity (SME): A Stream Management Entity is a module that process stream data and contains subcomponents that enable the communication with a DBMS. It consists of two layers. The Key-Value layer contains the Request Manager (RM) and the Key-Value Engine (KVE). The Request Manager receives requests from the LVM and implements the API. The Key-Value Engine manages the key-value system to realize the implementation structures at the SME (Key-Value structures). The values of these structures are provided by the stream system (Application Layer). A KVE can be a Key-Value store, a custom solution, a database etc. In most cases Key-Value Engines support a simplistic query language to query keys and/or values, but it could also be a strict Key-Value store, only supporting key retrieval. The Application layer is the actual stream system that process data streams. Any stream system may exist in this layer. The only requirement is the ability of stream programs to access the keys and values of the Key-Value structures. This could be done either *natively* – i.e. the Key-Value structures reside *within* the stream system and are directly accessible by the stream programs – or *externally* – i.e. the Key-Value structures are accessible through an API between the stream programs and the Key-Value store. Both approaches have pros and cons and our architecture does not assume the one or the other. In native implementations, the obvious benefits are performance and updatability – keys and values are always up to date, since programs directly manipulate these structures. The drawback is that one has to implement Key-Value structures' functionality within the stream system. In external implementations, one can use ready-to-use Key-Value stores, offering scalability and fault-tolerance. In fact, in many real applications this is the *only* possible approach – e.g. banking systems handling streams are application-specific and closed, offering a limited API, which could be used to update Key-Value structures.

Some well-known stream systems can easily support the implementation of Key-Value structures. For example the STREAM system [12] supports the CQL [14] language for the declaration of stream queries. STREAM supports the TableSource and QueryOutput interfaces [129] to import keys and output results. AURORA [3] can use connections points to static data

sets to get keys from a DBMS and to output results to a Key-Value structure. For MapReduce Online [54] a possible solution is the usage of custom Java code to import keys from a DBMS and the usage of online aggregation snapshots functionality to output computed aggregates.

5.6.1 DBMS-SME Application Programming Interface

Communication between the DBMS and a SME is carried out through a set of primitives implementing a request-response protocol. There are four different request types in order to define implementation structures at the SME, initiate program execution at the SME and manage the Key-Value Engine (send keys and retrieve values). Table 5.1 summarizes the request types, along with the responses of the SME. Note that all communication is DBMS driven.

Table 5.1: Request types of DBMS-SME API

Request type: *define*

Description	Defines the implementation structure of a linked column at the SME.
Parameters	*handlerID*: a unique identifier assigned to each linked column by the LinkView Manager, *execName*: a string containing the program name and the parameters' values for the call, *label*: a string containing the named output of the execution.
Response	true/false

Request type: *sendKeys*

Description	Sends to the SME a (subset of) the values of the base column(s) of one (or more) linked columns.
Parameters	*H*: a list of handlerIDs, *K*: a set of keys.
Response	true/false

Request type: *getValues*

Description	Retrieves the values of the links of a linked column.
Parameters	*handlerID*: the handlerID of the linked column, θ: a logical expression, involving only the linked column and its base column(s).
Response	A list of key-value pairs.

Request type: *delKeys*

Description	Deletes entries of Key-Value structure according to a set of keys K and/or a logical expression θ.
Parameters	*handlerID*: the handlerID of a linked column, θ: a logical expression, involving only the linked column and its base column(s), *K*: a set of keys.
Response	true/false

define request defines the implementation structure of a linked column at the SME. It also sends a unique (cross-DBMS) identifier to the SME, the *handlerID*. All further communication between the DBMS and the SMEs are carried out through this *handlerID*. At the SME, a Key-Value structure is defined for each *handlerID* and is named using the *label* parameter. Stream programs output values to these labeled Key-Value structures.

sendKeys request sends a set of keys *K* that update (append mode) the Key-Value structure of those *handlerID*s mentioned in *H*. The linked columns corresponding to the *handlerIDs* of *H* must have the same base column(s). Allowing a *sendKeys* request to affect several Key-Value structures is something useful performance-wise (bandwidth). When the *sendKeys* request is issued for the first time for a *handlerID*, it also initiates the program execution of that *handlerID*. Recall that the base columns of a LinkView correspond to a materialized view. When the base part of a LinkView is updated (i.e. during view maintenance, for example a new stock is inserted to the `Stocks` table) the *sendKeys* primitive is invoked to send the new keys to the SME.

getValues request is used during query processing for linked columns evaluation i.e. when a user submit a SQL query. It asks for the pairs of the Key-Value structure of *handlerID*. It may retrieve all key-value pairs corresponding to the submitted *handlerID* or it may retrieve key-value pairs according to a selection condition θ over the schema of the Key-Value structure (i.e. a condition mentioning keys and values.) For example, Query Q2 asks for the `categoryIDs`

having `minPrice` greater than 10 using LinkView LV3 and its linked column `minPrice`. This filtering could be pushed to the Key-Value Engine using the above-mentioned condition. The SME may or may not support the mapping of θ to the native language of the Key-Value engine. In the latter case, θ is ignored and all key-value pairs are returned.

delKeys request is useful once again during view maintenance. It deletes the entries of the Key-Value structure of *handlerID* that exists in set K and according to the condition θ. This is feasible if the SME can map θ to the native language of the Key-Value Engine, otherwise θ is ignored and only keys mentioned in K are deleted.

5.7 Implementation and Optimizations

The proposed architecture can have different implementations based on the application we want to support. For example the need for real-time acquisition of stream data can be better supported by an in-memory DBMS, an in-memory Key-Value Engine and by a high-performance cluster based stream system (e.g. financial applications). On the other hand, for less critical applications (e.g. product information in a RFID supply chain environment) the requirement for near real-time acquisition of stream data can be achieved with a distributed architecture where SMEs are in different network locations than the LVM.

In this section we provide details on our prototype system, how it operates, API implementation, and we describe SME implementation details. Also we provide a design example on how a stream program can be implemented by a stream programmer. Finally we describe how LV-Eval operator is implemented over a relational DBMS.

5.7.1 LinkView System

LinkView Manager (LVM) is implemented in C/C++ and operates over any relational database system using ODBC. In our prototype system we used the PostgreSQL DBMS. The default mode of LVM (prompt mode) accepts “create linkview” definitions and SQL queries. LVM supports a set of commands that define its operation. These commands are handled by the Commands Manager (COM) module. The commands are explained below:

- **init <*LinkView name*>:** it invokes the first *sendkeys* request for the LinkView.
- **drop <*LinkView name*>:** deletes a LinkView and stop associated stream program executions at the SMEs,
- **view <*LinkView name*>:** enables users to view the defined LinkViews and other miscellaneous statistics (e.g. linked column last update time etc)
- **readQ <*filename*>:** reads and executes from a file a LinkView definition/command or a SQL query

API requests are XML-based messages. Requests and responses involving data (key-value pairs) use comma-separated format (CSV). Communication between LVM and the implemented SME takes place via sockets. Our SME is written in C/C++. Key-Value Engine implements C++ hash maps, natively accessible by C++ threads manipulating stream data (stream programs)

In our framework stream programmers must implement stream processing queries/programs in their stream systems. A stream query/program is an invocation of a stream query/program implementation (we can have multiple invocations of the same query/program). A sample design pattern for those queries/programs is given below:

```
stream_qprogram (params, …) {
  input = get($_keys);
  queryStr =
     select I.Key, avg(S.value) as $_values
     from stream_data[window 60sec] S,
          input I
     where S.key = I.key;
  qptr = query.Prepare(queryStr);
  qptr.addLabel($_values,"label");
  queryAdapter.put(qptr);
  qptr.Exec();
}
```

`$_keys` is a variable containing the keys received from the DBMS. It can be a pointer to a Key-Value structure that contains keys and can be accessed from the stream system. Query is written in the supported language of each stream system and executed when SME receives the *sendKeys* request (init command). The program assigns labels to each possible query output and the put method defines how the computed values are stored in the Key-Value structure. This is a sample design pattern and is use a SQL-like stream language. The only requirement of a stream system is to be able to access the Key-Value structure to get keys and put values. Note that the

`queryAdapter` class can implement periodic or continuous update of aggregates in the Key-Value structure.

5.7.2 LV-Eval Operator Implementation

Since we implemented LVM on top of a DBMS instead of within, LV-Eval operator is realized through query rewriting, *prior to* actual query processing. Each LinkView in an SQL statement is rewritten as a join between its base part and its linked columns. Specifically, assume a LinkView L, which consists of a materialized view V extended by *n* linked columns L_i, i=1,2,…,n. Lets denote the base column of L_i as A_i. For each L_i we define a temporary table L'_i, i=1,2,…, n, with schema (A_i , V), where V is a column with data type the one mentioned in the `add column` statement of L_i. In essence, L'_i corresponds to the materialization of the response (i.e. key-value pairs) of a *getValues* request for linked column L_i. Each occurrence of L in an SQL statement is replaced by the following expression:

$$V \bowtie_{A1} L'_1 \bowtie_{A2} L'_2 \bowtie \ldots \bowtie_{An} L'_n$$

Since LV-Eval is not natively implemented in the DBMS, any optimization of it must be done before query rewriting by the LVM. In our prototype we implemented two simple optimization techniques:

- ***Avoiding unnecessary materializations of L'_i*** : identify the linked columns of the LinkView that do not participate in the SQL statement and exclude them from the rewriting expression. This is equivalent to pushing down projections to the LV-Eval operator. For example, Query Q1 only requires the `price` column of LinkView LV1. Linked columns `price10` and `volume` do not appear in the rewritten expression.
- ***Reducing the size of L'_i*** *:* identify selection conditions involving linked columns and apply them directly to the response of the *getValues* requests, prior to the materialization of L'_i s. This is equivalent to pushing down selections to the LV-Eval operator. For example, Query Q2 asks for the `categoryIDs` having `minPrice` greater than 10. The predicate *((real) value > 10)* can be applied to the key-value pairs returned by the *getValues* request on `minPrice` linked column of LV3.

5.7.3 Optimizations

Several techniques can be applied at various stages of query evaluation to improve performance. We briefly mention here some of the ad-hoc methods we implemented, we describe open issues and describe some optimizations that can be applied across LinkView architecture i.e. not specifically in DBMS side.

LVM can improve performance by analyzing the definitions of linked columns and try to identify interrelationships (e.g. sharing keys and/or values.) For example, *sendKeys* request can send keys for multiple *handlerIDs*. In addition, there are several open issues that LVM should handle:

- how to handle (evaluate, optimize) LinkView definitions based on previously defined LinkViews, similar to [53]
- when a linked column can be inferred by another?

Quality of Service (QoS) is a well-known concept in stream literature [3]. In our framework, linked columns can be cached at the LVM's Cache Manager and used during query evaluation, instead of issuing *getValue*s requests to the SME. This is particularly useful when a large number of queries access a small number of LinkViews. Caching policies could be set with specialized commands. In our implementation, a freshness parameter (in seconds) can be specified and is applicable to *all* linked columns of all LinkViews. Note that the CM could act proactively and issue *getValues* requests independently of SQL queries.

In section 5.7.2 we discussed how to reduce the size of the received columns, by pushing down predicates to the response of a *getValues* request. In fact, we can even push the predicate to the corresponding key-value structure of the linked column – if selectivity is low and the Key-Value Engine supports it – to avoid communication cost. The design of the *getValues* request allows something like that. This is a specific case of a more general problem, mapping relational operators to Key-Value Engine's operations. For example, if two linked columns share the same base column(s), the join could take place in the KVE and the response could be in the form (key, value1, value2). Depending on the KVE, this could be much more efficient than performing the join in the DBMS. In addition, it would result in communication cost savings. Several other optimizations can be supported across all layers of our framework. Below we describe some possibilities:

- **Optimization 5.1 - Keys re-using:** a LinkView definition can contain multiple "using" declarations. The using keyword defines the keys that the DBMS must send to a SME. These keys are put in a Key-Value structure and a SME compute aggregates using them. We can avoid sending the same keys if the keys have already be sent in a

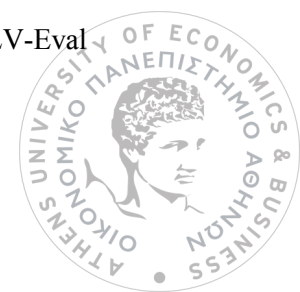
previous using declaration. The same can apply between multiple LinkViews definitions. With this optimization we can minimize the startup-time of a stream query/program and save bandwidth if the LinkView Manager and SMEs are on different machines (e.g. network, cluster)

- **Optimization 5.2 - Sharing keys in SME:** in our architecture a Key-Value structure is used to store the keys retrieved by the DBMS. Strictly a Key-Value structure is a container that for each key has a value. A different implementation approach is: for each key there exist multiple values. In this way keys are shared between multiple aggregates. In any case the LinkView Manager must be able to access any value using the assigned label i.e. the Key-Value interface must be used independently of the Key-Value structure implementation. This optimization leads to compact Key-Value structures and allow the retrieval of multiple aggregates per key.
- **Optimization 5.3 - Periodic or request based aggregate updates between Key-Value structure and stream system:** a stream system uses keys from the Key-Value Engine to compute stream aggregates. When a stream system computes a value it must put this value back to the Key-Value structure. This can be happen 1) in a best effort mode i.e. continually update the Key-Value structure 2) in a periodic mode -e.g. a Service Level Agreement may define that the values are refreshed every 5 minutes- 3) put values to the Key-Value structure per user requests. These different configurations enable Key-Value Engine to act as a caching layer. Proper usage of these techniques can lead to better utilization of SMEs from multiple users.
- **Optimization 5.4 - Linked columns caching:** when a naïve user executes a SQL query, the linked columns replaced by stream values. In a multi-user environment a large number of users issue SQL queries in a small number of LinkViews within a small time interval. This will lead to multiple LinkViews evaluations and bad performance. Cache Manager can cache linked columns values and we can define a condition if they will be reused by next SQL queries. For example we may define that the cached linked columns which are not over 1 minute old can be used as a result in a SQL query.
- **Optimization 5.5 - Links sharing:** there are cases that multiple links point to the same value of a Key-Value structure. This is frequent in LinkViews that their base table contains composite keys. For example in LV4 multiple stocks belong to a category. In this case the average price of a category is the same for multiple rows (`stocksID`). A proper implementation can update multiple pointers at once. Moreover the `using`

keyword on LinkView definition syntax can send only the distinct values of such a column.

- **Optimization 5.6 - Stream program sharing:** Stream systems can share program executions among LinkViews. If a new LinkView uses a linked column computed by an already invoked program, SME can avoid starting a new process. This is feasible by assigned the same `handlerID` to multiple LinkView definitions. Additionally, shared query execution strategies can be applied among program executions in SMEs [13]. [155] studies optimization techniques for similar stream aggregation queries differing in the grouping attributes (keys). Optimization techniques for window aggregates within a single query are given in [92]. Similarly, in [76] optimizations for multiple aggregate continuous queries are provided. Such techniques can be used in our framework for programs containing multiple stream aggregates. In our architecture, stream programs can be invoked with parameters which can define program's execution behavior. Parameterized continuous queries for complex pattern detection have been studied in [148]. The described techniques can be used in our framework.

- **Optimization 5.7 - LV-Eval Operator optimizations:** The LV-Eval is a unary operator that produces a table replacing the links of linked columns with stream aggregates when a LinkView is used in a SQL query. Standard relational optimization techniques [74] can be applied on queries containing LinkViews. For example heuristic based optimization techniques can be used to apply re-ordering in the operators in the query tree. The main goal is to apply first operations that reduce the size of intermediate results in the execution plan. This size decrement can be in number of rows or in number of columns or both. For the first case the *getValue* request will return fewer results if a θ condition is supported. For the second case note that the LV-Eval operator calls *getValue* request multiple times, once per linked column. If some columns projected out then *getValue* request will be called fewer times. For example a relational algebra expression for Q1 can be: $\pi_{H.stockID}(\sigma_{H.stockID=P.stockID \text{ AND } H.closingPrice>P.price \text{ AND } H.date=date()-1}(\text{Historical x Prices}))$. The Prices LinkView is evaluated to a table and is used in a cartesian product with Historical table. Selections applied in the result of the cartesian product and finally the stockID column is retrieved. An equilevant relational algebra expression is: $\pi_{H.stockID}((\sigma_{H.date=date()-1}(\text{Historical}) \bowtie_{H.stockID=P.stockID \text{ AND } H.closingPrice>P.price} (\pi_{P.price,\ P.stockID}\text{Prices}))$. In this case projection pushdown is applied to the Price Linkview. In this way the LV-Eval Operator calls getValues request only once for the stream structure with the price label. Predicate and projection pushdown is a widely used technique in relational systems and can be used with the LV-Eval



Operator. In our system we use a small number of rules for query rewriting to enable this kind of optimization.

- **Optimization 5.8 - Predicate migration to SME:** Many queries apply predicates over linked columns. For example in LV3 for both Queries 2 and 3 there exist a predicate over linked columns. As described in Section 6.4 such a query is first evaluated in a table and then the selection operations are applied. An alternative implementation is to pass this predicate as a condition θ to SME system and in particular to the Key-Value Engine. Most Key-Values Engines support fast retrieval of values if we provide the Key but they can also support predicates over values. With this optimization the key-value data file that is send back to the DBMS when a user issue a SQL query can become smaller.
- **Qptimization 5.9 - Joining base table with Key-Value structures:** For each requested linked column an outer join operation must be applied with the base table of LinkView. Having multiple columns may lead to bad performance. One option is to generate in the Key-Value Engine a key-value data file that combines multiple values per key. As a result the number of joins is decreased. Other optimization techniques that can be used are given on [88]. These techniques are based on padding the Key-Value tuples with nulls and apply a union to get the complete result of an outer join. Alternatively we can use a more efficient algorithm to implement the join on the LVM and not use the database system.

5.8 Experiments and Performance

Measuring the performance of a LinkView system has some inherent difficulties. First, there are different architecture configurations (e.g. SMEs can be in different network locations, KVE implementations can be native or external etc). Second, SMEs may represent stream systems and performance can considerably fluctuate.

Our testing platform has the following characteristics: Windows 7, 2.13 GHz Intel Core i3 Processor i3-330M and 4 GB of RAM. We conducted our experiments in a single machine i.e. both LVM and SME run on the same node. LVM uses data stored in PostgreSQL DBMS and access is via ODBC. The financial database used in our experiments contains synthetic data sets. SME is a process and is composed of stream programs implementing LinkViews LV1 to LV4 and an embedded in-memory Key-Value engine (C++ hash table). SME is fed with tuples generated by a custom C/C++ stream generator running as a process in the same machine. The provided tuples have the following schema: <stock_id, category_id, stock_price>. Stream generator-SME interprocess communication is achieved via shared memory. LVM-SME communication is done via sockets.

We measured the execution time for several SQL queries applied over LinkViews. We used LV3 (`MinMaxPriceCategory`) and the following SQL queries in our experiment:

QE1.
```
select categoryID, minPrice
from MinMaxPriceCategory
```

QE2.
```
select categoryID, minPrice, maxPrice
from MinMaxPriceCategory
```

QE3.
```
select M.categoryID, M.minPrice,
       H.yesterdayMinPrice
from  MinMaxPriceCategory M,
      HistoricalMinPCategory H
where M.categoryID = H.categoryID
```

`MinMaxPriceCategory` uses `pMinMax()` from the SME to get values for `minPrice` and `maxPrice` columns. `HistoricalMinPCategory` is a database table containing previous day's minimum prices per category and has schema: <categoryID, yesterdayMinPrice>. When an SQL query issued on LVM the *getValues* request is called for each linked column and the SME returns a batch of key-value pairs to the LVM. These pairs of data are stored as CSV files in LVM machine and loaded in the DBMS as two-column Key-Value tables. We used the COPY command of PostgreSQL to load the CSV files. For QE3 the `minPrice` column is memory-resident while `HistoricalMinPCategory` is a disk-resident table. We annotate queries that use disk Key-Value tables with D and queries use memory Key-Value tables with M. The execution times of SQL queries (in seconds) are shown in Table 2. We varied key size (`categoryID`) from 50000 to 300000 for `MinMaxPriceCategory` with a step of 50000. The `HistoricalMinPCategory` in QE3 has the same number of rows as the specified size.

Table 5.2: SQL queries execution time (seconds)

Query	Number of keys					
	50000	100000	150000	200000	250000	300000
$QE1_M$	0.565	1.033	1.546	2.045	2.588	3.175
$QE1_D$	0.698	1.257	1.868	2.594	3.096	4.049
$QE2_M$	1.208	2.157	3.275	4.230	5.351	6.286
$QE2_D$	1.468	2.708	3.797	5.098	6.763	7.656
QE3	0.667	1.343	1.924	2.447	3.432	4.071

Note that queries using disk Key-Value tables ($QE1_D$, $QE2_D$) are slower than queries with in memory Key-Value tables ($QE1_M$, $QE2_M$). QE3 requires only one linked column from SME and its performance is similar to QE1. Also both versions of QE2 are slower than QE1. QE1 can be executed without a join i.e. all columns mentioned in the select clause can be retrieved by the created Key-Value table with schema <categoryID, minPrice>. On the other hand for QE2 a join between the Key-Value table with schema <categoryID, minPrice> and the Key-Value table with schema <categoryID, maxPrice> must be performed. In particular, QE2 evaluation requires the following tasks: 1) request of key-value data from SMEs (*getValues* API call) and storage as CSV files 2) create Key-Value tables and load the corresponding CSV files, 3) join of Key-Value tables and execution of the issued SQL query. Details on how these tasks affect linked columns evaluation can give better insights about performance bottlenecks in a LinkView system. Figures 5.3, 5.4, and 5.5 show the execution time of queries $QE2_D$, $QE2_M$ and QE3, split by task.

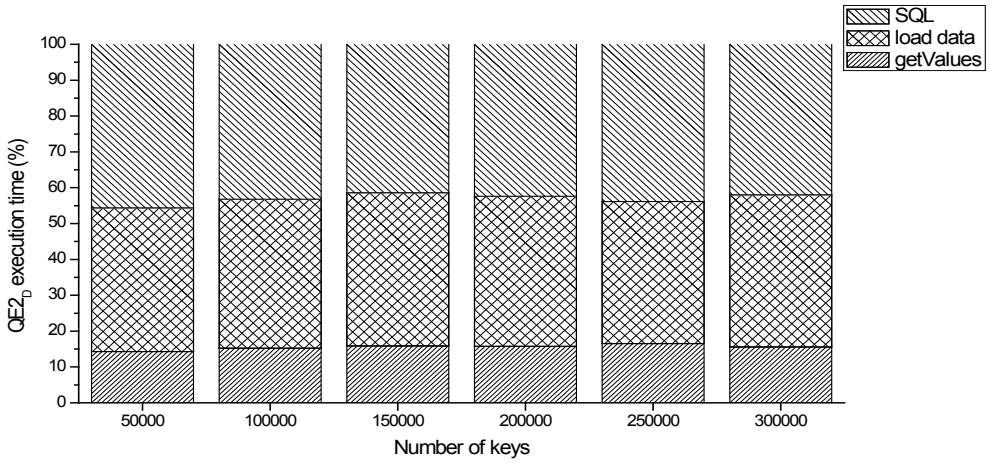


Figure 5.3: $QE2_D$ tasks

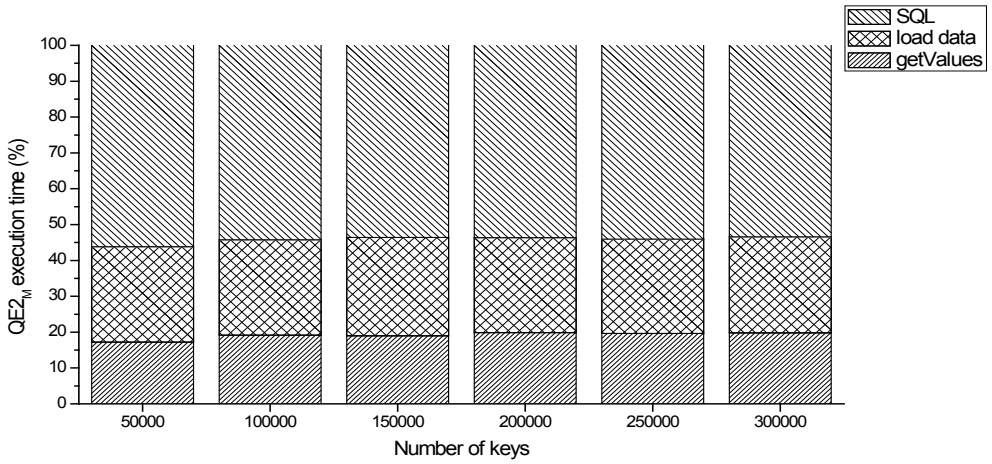


Figure 5.4: $QE2_M$ tasks

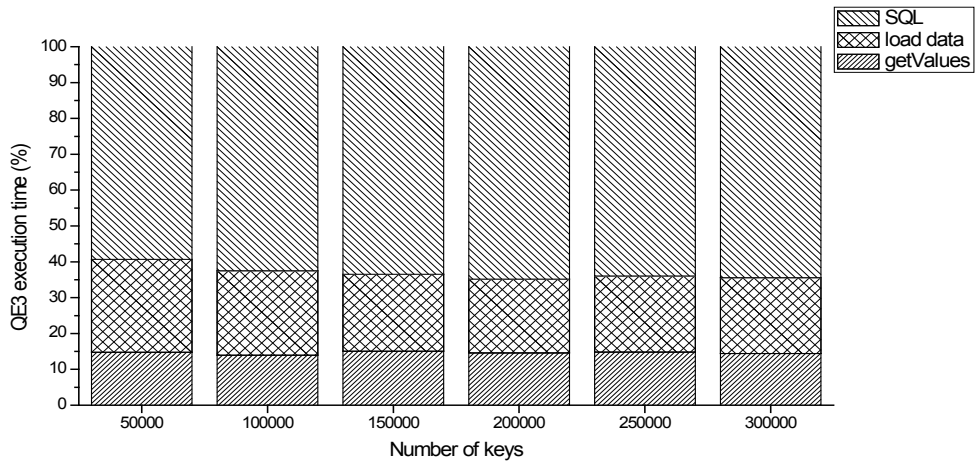


Figure 5.5: QE3 tasks

getValues request shows small variations in execution time for each query as the data transferred from SME to LVM is particular small: each key-value pair consists of an integer and a real value. For QE3 we transfer key-value pairs for one column (`minPrice`) and for QE2 for two columns (`minPrice`, `maxPrice`). $QE2_M$ spends less time than $QE2_D$ on loading data as it stores the received key-value pairs from SME in memory, avoiding disk writing. QE3 spends less time on loading data than $QE2_M$ as it creates and loads only one in-memory Key-Value table. QE3 uses a hash index on `categoryID` of `HistoricalMinPCategory` table which enables faster query execution than $QE2_M$ that does not use any index. Such plots can help users to diagnose bottlenecks in their LinkView system and apply optimizations as those given in subsection 5.7.3.

We test our system in case we push a predicate to the implemented SME. *getValues* may retrieve key-value pairs according to a selection condition θ if supported by the KVE. We used a modified version of $QE1_M$ to study the performance of this optimization:

$QE1_{M\theta}$.
```
select categoryID, minPrice
from MinMaxPriceCategory
where categoryID<x
```

where x ranges from 20000 to 100000 with a step of 20000. The number of keys used for the definition of `MinMaxPriceCategory` is 100000. Figure 5.6 shows execution time (in seconds) when (a) the predicate can be pushed to the SME, and (b) when selection takes place in the DBMS.

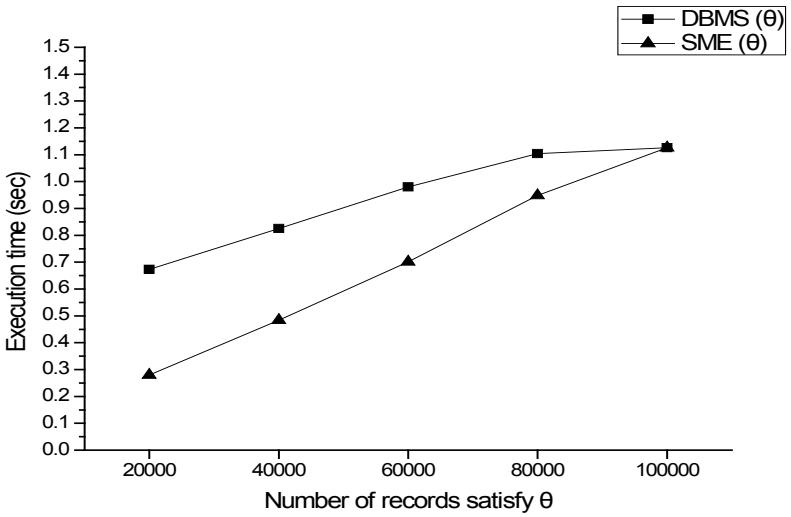


Figure 5.6: $QE1_{M\theta}$

5.9 Summary and Conclusions

Applications and utilization of stream data can be found today not only in "traditional" real-time environments, such as finance and telecommunications, but also in a wide variety of domains and settings, such as supply chain (through RFID sensors), energy management (through smart meters), social networks (through status updates) and many others. While data stream management systems (DSMS) are technologically mature and address most of the challenges in stream processing, they lack standardization in terms of modeling, querying and interoperability. So far, stream processing was confined within an organization. However, modern applications need to

integrate and manage aggregates produced by a variety of stream engines, from complete Data Stream Management Systems (DSMSs) to stand-alone stream-handling components.

We have presented an integration framework for relational database management systems and heterogeneous stream systems. In the proposed framework keys are managed by databases and stream data sets corresponding to these keys are managed by stream systems. A theoretical framework has been developed and a key-value based interface has been proposed. We argued for the advantages of such an approach. A prototype system has been implemented and can operate on top of any relational database management system. The goal of this research is to bring stream aggregates to naïve database users and analysts in a stream-transparent way. As most users are familiar with traditional relational systems and SQL, bringing stream support within such environments is of major importance.

Chapter 6

Conclusions

6.1 Summary

Modern applications require advanced data analysis over voluminous and continuous data sets. These streams of data must be processed in (near) real-time and results must be provided continuously. Applying analytics over stream data transforms passive organizations to active i.e. an organization becomes aware of what is happening in its immediate business environment and how internal or external events affect organization's daily operations in (near) real-time. Moreover considering that relational model and relational Database Management Systems (RDBMSs) are the de-facto approach for storing and processing structured data, integrating RDBMSs/relational model with stream data is of major importance. The current thesis tries to solve the above problems. In particular in this thesis we provide methods and tools to combine relational and stream data for real-time analytics. Additionally, we provide a framework that enables RDBMSs to interoperate with heterogeneous stream engines.

To support analytics over stream data SQL-like extensions are proposed in Chapter 3. The intuition is that for each relational value exists in a table we can attach a stream aggregate. Moreover we can attach multiple aggregates from multiple sources. Correlated aggregates are also feasible enabling complex data analysis over streams. The definition of such queries is simple and easily understandable by database users.

A spreadsheet-like approach for stream analytics is developed in Chapter 4. Spreadsheet tools are used by millions of users for offline data analysis. A method that resembles the common usage of spreadsheets (column-by-column) is proposed for stream query definition. The queries that can be defined using this method comprise a class of useful and practical queries that can be used for decision making.

In Chapter 5 we introduce a relational-based integration framework that sits atop any relational DBMS and mix DBMS' data and stream aggregates managed by different stream systems. We propose a special view layer defined over standard relational schemas: views in that layer, called LinkViews, consist of base- and linked- columns. A linked column is associated to a base column

in the view and a program at a stream engine. The program matches the keys of the base column with values, which then become the content of the linked column during query evaluation. We propose an SQL extension to define link views and an API to carry out the required communication between the relational and the stream systems. We claim that this framework: (a) is suitable for simple database users, (b) addresses an important and useful class of queries, overlooked so far, (c) presents numerous optimization opportunities to minimize communication and processing costs, and (d) can serve as a standard for relational-stream interoperability.

6.2 Future Work

This section describes further work on each chapter.

Chapter 3 - SQL Extensions for Real-Time Analytics: Stream variable queries modeling and evaluation assumes one machine i.e. the widened relation existed in one partition. We can horizontally partition the widened relation to several processing nodes and distribute the stream tuples to all of these. Having multiple machines requires a distributed evaluation algorithm for the computation of stream variable aggregates. This direction can be studied further. Stream variables can have a large number of function members allowing for logical or physical windows, flushing the contents of a queue at any time, pausing or restarting a queue, etc. An extensibility framework that enables developers to define their own functions could be useful. In this way a set of useful functions (library) can be created that can be used from multiple users enhancing the usefulness of stream variables. In some applications the large volume of stream data and the requirement for only one pass over stream data has give a birth to stream processing techniques that compute approximate answers. In chapter 3 stream variable queries provide exact answers assuming that each data stream element can be efficiently handled by our system. Further work can be done on how stream variables can support reporting functions that provide approximate answers (e.g. data synopses). Moreover Quality of Service (QoS) specifications can be used for the same reason. These extensions are suitable for stream variable system as reporting functions can easily enclose approximate algorithms. However correlated stream aggregates that use data synopsis need further investigation.

Chapter 4 - Spreadsheet-like Stream Processing: More theoretical work could be developed about the expressive power and the limitations of the proposed spreadsheet-like approach for stream querying. Spreadsheets can support array-style computations (e.g. compare two consecutive cells). How we can enable array-style processing for stream data is an interesting problem.

Spreadsheets provide a simple and intuitive interface to express complex relationships among cells, columns and rows. Our approach uses a column-by-column approach for the definition of stream queries. Further research can consider how spreadsheet constructs (i.e. cells, rows) can support complex stream queries. Such a spreadsheet-like query interface can be appropriate for naïve users. As a result, the development time and cost of data stream applications will decrease and stream systems will become accessible to many more people (i.e. not necessarily experts). Furthermore, spreadsheet query formulation is likely to serve as the basis for identifying efficient implementations, since succinct, concise and compact representations at the conceptual level lead to efficient optimizations at the processing level. Overall a platform that can support the declaration of stream queries using spreadsheets (i.e. define queries similar to the operations supported in Excel) can bring stream data processing to masses.

Also query-streams-by-example using the spreadsheet interface (i.e. defining a query stream example pattern in a spreadsheet on how a stream element processing is happen and how this is applied in a series of stream elements) can be studied further. Human computer interaction research for databases is an undistinguished research area despite its importance. Spreadsheets can be used to further investigate this research area.

Chapter 5 - An Integration Framework for Relational and Stream Systems: Further work includes the investigation on how LinkView definitions could be incorporated into a relational optimizer. It would be interesting to investigate and develop a cloud infrastructure for the implementation of LinkView framework. Such an infrastructure can be used by stream providers to provide services (programs) to database users. A better study is need for the interface between the streaming applications and the Key-Value store, and the policies governing how the streaming results are materialized on the Key-Value store. Moreover more work can be done for the definition of a stream program design pattern that can be used from SMEs in LinkView framework. Reusable modules and code is important for the wide acceptance of LinkView framework. Finally LinkViews can be used to link to other non-stream systems, allowing the creation of a generic integration platform for different types of data and systems (e.g. MapReduce, Big Data platforms, etc.)

Bibliography

[1] Abadi J. Daniel, Ahmad Yanif, Balazinska Magdalena, Cetintemel Ugur, Cherniack Mitch, Hwang Jeong-Hyon, Lindner Wolfgang, Maskey Anurag, Rasin Alex, Ryvkina Esther, Tatbul Nesime, Xing Ying, B. Zdonik Stanley: The Design of the Borealis Stream Processing Engine. CIDR 2005:277-289

[2] Abadi J. Daniel, Carney Donald, Cetintemel Ugur, Cherniack Mitch, Convey Christian, Lee Sangdon, Stonebraker Michael, Tatbul Nesime: a new model and architecture for data stream management. VLDB J. (VLDB) 12(2):120-139 (2003)

[3] Abadi J. Daniel, Carney Donald, Cetintemel Ugur, Cherniack Mitch, Convey Christian, Erwin C., Galvez F. Eduardo, Hatoun M., Maskey Anurag, Rasin , Singer A., Stonebraker, Michael, Tatbul Nesime, Xing Ying, Yan R., Zdonik B. Stanley: Aurora: A Data Stream Management System. SIGMOD 2003:666

[4] Abiteboul Serge, Agrawal Rakesh, Bernstein A. Philip, Carey J. Michael, Ceri Stefano, Croft W. Bruce, DeWitt J. David, Franklin J. Michael, Garcia-Molina Hector, Gawlick Dieter, Gray Jim, Haas M. Laura, Halevy Y. Alon, Hellerstein M. Joseph, Ionnidis E. Yannis, Kersten L. Martin, Pazzani J. Michael, Lesk Michael, Maier David, Naughton F. Jeffrey, Schek Hans-Jorg, Sellis K. Timos, Silberschatz Avi, Stonebraker Michael, Snodgrass T. Richard, Ullman D. Jeffrey, Weikum Gerhard, Widom Jennifer, Zdonik B. Stanley: The Lowell database research self-assessment. Commun. ACM (CACM) 48(5):111-118 (2005)

[5] Abouzeid Azza, Bajda-Pawlikowski Kamil, Abadi J. Daniel, Rasin Alexander, Silberschatz Avi: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. PVLDB 2(1):922-933 (2009)

[6] Agarwal Sameet, Agrawal Rakesh, Deshpande Prasad, Gupta Ashish, F. Naughton Jeffrey, Ramakrishnan Raghu, Sarawagi Sunita: On the Computation of Multidimensional Aggregates. VLDB 1996:506-521

[7] Ailamaki Anastassia, Faloutsos Christos, Fischbeck S. Paul, Small J. Mitchell, VanBriesen M. Jeanne: An environmental sensor network to determine drinking water quality and security. SIGMOD Record (SIGMOD) 32(4):47-52 (2003)

[8] Akinde O. Michael, Bohlen H. Michael: Generalized MD-Joins: Evaluation and Reduction to SQL. Databases in Telecommunications 2001:52-67

[9] Akinde O. Michael, Bohlen H. Michael, Johnson Theodore, Lakshmanan V. S. Laks, Srivastava Divesh: Efficient OLAP Query Processing in Distributed Data Warehouses. EDBT 2002:336-353

[10] Application Level Events (ALE) Standard, http://www.gs1.org/gsmp/kc/epcglobal/ale (last accessed February 2013)

[11] Arasu Arvind, Babcock Brian, Babu Shivnath, McAlister Jon, Widom Jennifer: Characterizing Memory Requirements for Queries over Continuous Data Streams. PODS 2002:221-232

[12] Arasu Arvind, Babcock Brian, Babu Shivnath, Datar Mayur, Ito Keith, Motwani Rajeev, Nishizawa Itaru, Srivastava Utkarsh, Thomas Dilys, Varma Rohit, Widom Jennifer: STREAM: The Stanford Stream Data Manager. IEEE Data Eng. Bull. 26(1): 19-26 (2003)

[13] Arasu Arvind, Widom Jennifer: Resource Sharing in Continuous Sliding-Window Aggregates. VLDB 2004:336-347

[14] Arasu Arvind, Babu Shivnath, Widom Jennifer: The CQL continuous query language: semantic foundations and query execution. VLDB J. (VLDB) 15(2):121-142 (2006)

[15] Avnur Ron, Hellerstein M. Joseph: Eddies: Continuously Adaptive Query Processing SIGMOD 2000:261-272

[16] Bai Yijian, Wang Fusheng, Liu Peiya, Zaniolo Carlo, Liu Shaorong: RFID Data Processing with a Data Stream Query Language. ICDE 2007:1184-1193

[17] Bardaki Cleopatra, Pramatari, Katerina, 2007: RFID-enabled supply chain collaboration services in a networked retail environment. In: Proceedings of the 20th International Bled Electronic Commerce Conference, Bled, Slovenia, June 3–6.

[18] Barua Anitesh, Lee Byungtae, 1997. An economic analysis of the introduction of an electronic data interchange system. Information Systems Research 8 (4), 398–422.

[19] Babcock Brian, Olston Chris: Distributed Top-K Monitoring. SIGMOD 2003:28-39

[20] Babcock Brian, Babu Shivnath, Datar Mayur, Motwani Rajeev: Chain : Operator Scheduling for Memory Minimization in Data Stream Systems. SIGMOD 2003:253-264

[21] Babcock Brian, Babu Shivnath, Datar Mayur, Motwani Rajeev, Widom Jennifer: Models and Issues in Data Stream Systems. PODS 2002:1-16

[22] Babu Shivnath, Widom Jennifer: Continuous Queries over Data Streams. SIGMOD Record (SIGMOD) 30(3):109-120 (2001)

[23] Brettlecker Gert, Schuldt Heiko, Schek Hans-Jorg: Towards Reliable Data Stream Processing with OSIRIS-SE. BTW 2005:405-414

[24] Botan Irina, Cho Younggoo, Derakhshan Roozbeh, Dindar Nihal, Gupta Ankush, M. Haas Laura, Kim Kihong, Lee Chulwon, Mundada Girish, Shan Ming-Chien, Tatbul Nesime, Yan Ying, Yun Beomjin, Zhang Jin: A demonstration of the MaxStream federated stream processing system. ICDE 2010:1093-1096

[25] Botan Irina, Cho Younggoo, Derakhshan Roozbeh, Dindar Nihal, Haas M. Laura, Kim Kihong, Tatbul Nesime: Federated Stream Processing Support for Real-Time Business Intelligence Applications. BIRTE 2009:14-31

[26] Botan Irina, Derakhshan Roozbeh, Dindar Nihal, Haas M. Laura, J. Miller Renee, Tatbul Nesime: SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. PVLDB 3(1):232-243 (2010)

[27] Carney Donald, Cetintemel Ugur, Cherniack Mitch, Convey Christian, Lee Sangdon, Seidman Greg, Stonebraker Michael, Tatbul Nesime, Zdonik B. Stanley: Monitoring Streams - A New Class of Data Management Applications. VLDB 2002:215-226

[28] Castellanos Malu, Gupta Chetan, Wang Song, Dayal Umeshwar: Leveraging web streams for contractual situational awareness in operational BI. EDBT/ICDT Workshops 2010

[29] Castellanos Malu, Wang Song, Dayal Umeshwar, Gupta Chetan: SIE-OBI: a streaming information extraction platform for operational business intelligence. SIGMOD 2010:1105-1110

[30] Chan David, Ge Rong, Gershony Ori, Hesterberg Tim, Lambert Diane: Evaluating online ad campaigns in a pipeline: causal models at scale. KDD 2010:7-16

[31] Chandrasekaran Sirish, Cooper Owen, Deshpande Amol, Franklin J. Michael, Hellerstein M. Joseph, Hong Wei, Krishnamurthy Sailesh, Madden Samuel, Raman Vijayshankar, Reiss Frederick, Shah A. Mehul: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In Proceedings of the 1st Biennial Conference on Innovative Data Systems Research. CIDR'03.

[32] Chandrasekaran Sirish, Franklin J. Michael: PSoup: a system for streaming queries over streaming data. VLDB J. (VLDB) 12(2):140-156 (2003)

[33] Chatziantoniou Damianos: The PanQ Tool and EMF SQL for Complex Data Management. KDD 1999:420-424

[34] Chatziantoniou Damianos: Evaluation of Ad Hoc OLAP: In-Place Computation. SSDBM 1999:34-43

[35] Chatziantoniou Damianos: Ad Hoc OLAP: Expression and Evaluation. ICDE 1999:250

[36] Chatziantoniou Damianos: Using grouping variables to express complex decision support queries. Data Knowl. Eng. (DKE) 61(1):114-136 (2007)

[37] Chatziantoniou Damianos, Anagnostopoulos Achilleas: NESTREAM: Querying Nested Streams. SIGMOD Record (SIGMOD) 33(3):71-78 (2004)

[38] Chatziantoniou Damianos, Tzortzakakis Elias: ASSET queries: a declarative alternative to MapReduce. SIGMOD Record (SIGMOD) 38(2):35-41 (2009)

[39] Chatziantoniou Damianos, Doukidis George: Incorporating Data Stream Analysis into Decision Support Systems. Encyclopedia of Information Science and Technology (III) 2005:1431-1439

[40] Chatziantoniou Damianos, Pramatari Katerina, Sotiropoulos Yannis: COSTES: Continuous spreadsheet-like computations. ICDE Workshops. In International Workshop on RFID Data Management. ICDE Workshops, RFDM'08, 82-87.

[41] Chatziantoniou Damianos, Pramatari Katerina, Sotiropoulos Yannis: Supporting real-time supply chain decisions based on RFID data streams. Journal of Systems and Software (JSS) 84(4):700-710 (2011)

[42] Chatziantoniou Damianos, Ross A. Kenneth: Querying Multiple Features of Groups in Relational Databases. VLDB 1996:295-306

[43] Chatziantoniou Damianos, Akinde O. Michael, Johnson Theodore, Kim Samuel: The MD-join: An Operator for Complex OLAP. ICDE 2001:524-533

[44] Chatziantoniou Damianos, Sotiropoulos Yannis: ASSET Queries: A Set-Oriented and Column-Wise Approach to Modern OLAP. BIRTE 2009:66-83

[45] Chatziantoniou Damianos, Sotiropoulos Yannis: Stream Variables: A Quick but not Dirty SQL Extension for Continuous Queries. ICDE Workshops 2007:19-28

[46] Chaudhuri Surajit, Shim Kyuseok: Including Group-By in Query Optimization. VLDB 1994:354-366

[47] Chaudhuri Surajit, Dayal Umeshwar: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record (SIGMOD) 26(1):65-74 (1997)

[48] Chaudhuri Surajit, Dayal Umeshwar,. Narasayya R. Vivek: An overview of business intelligence technology. Commun. ACM (CACM) 54(8):88-98 (2011)

[49] Chen Jianjun, DeWitt J. David, Tian Feng, Wang Yuan: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD 2000:379-390

[50] Chen Yixin, Dong Guozhu, Han Jiawei, Pei Jian, W. Wah Benjamin, Wang Jianyong: Online Analytical Processing Stream Data: Is It Feasible? DMKD 2002

[51] Cherniack Mitch, Balakrishnan Hari, Balazinska Magdalena, Carney Donald, Cetintemel Ugur, Xing Ying, Zdonik B. Stanley: Scalable Distributed Stream Processing. In Proceedings of the 1st Biennial Conference on Innovative Data Systems Research. CIDR'03.

[52] Cohen Edith, Strauss Martin: Maintaining time-decaying stream aggregates. PODS 2003:223-233

[53] Colby S. Latha, Kawaguchi Akira, Lieuwen F. Daniel, Mumick Inderpal Singh, Ross A. Kenneth: Supporting Multiple View Maintenance Policies. In Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGMOD'97, 405-416.

[54] Condie Tyson, Conway Neil, Alvaro Peter, M. Hellerstein Joseph, Elmeleegy Khaled, Sears Russell: MapReduce Online. In Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation. NSDI '10:313-328

[55] Cormode Graham, Korn Flip, Muthukrishnan S., Srivastava Divesh: Finding Hierarchical Heavy Hitters in Data Streams. VLDB 2003:464-475

[56] Cortes Corinna, Fisher Kathleen, Pregibon Daryl, Rogers Anne, Smith Frederick: Hancock: A language for analyzing transactional data streams. ACM Trans. Program. Lang. Syst. (TOPLAS) 26(2):301-338 (2004)

[57] Cranor D. Charles, Johnson Theodore, Spatscheck Oliver, Shkapenyuk Vladislav: Gigascope: A Stream Database for Network Applications. SIGMOD 2003:647-651

[58] Cranor D. Charles, Gao Yuan, Johnson Theodore, Shkapenyuk Vladislav, Spatscheck Oliver: Gigascope: high performance network monitoring with an SQL interface. SIGMOD 2002:623

[59] Dean Jeffrey, Ghemawat Sanjay: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004:137-150

[60] Dean Jeffrey, Ghemawat Sanjay: MapReduce: a flexible data processing tool. Commun. ACM (CACM) 53(1):72-77 (2010)

[61] Demirbas Murat, Chow Ken Yian, Wan Chieh Shyan: INSIGHT: Internet-Sensor Integration for Habitat Monitoring. WOWMOM 2006:553-558

[62] Deshpande Amol, Nath Suman, B. Gibbons Phillip, Seshan Srinivasan: Cache-and-Query for Wide Area Sensor Databases. SIGMOD 2003:503-514

[63] Dobra Alin, Garofalakis N. Minos, Gehrke Johannes, Rastogi Rajeev: Processing complex aggregate queries over data streams. SIGMOD 2002:61-72

[64] Elmasri Ramez, Navathe B.Shamkant: Fundamentals of Database Systems, 5nd Edition. Benjamin/Cummings 2010

[65] Franklin J. Michael, Krishnamurthy Sailesh, Conway Neil, Li Alan, Russakovsky Alex, Thombre Neil: Continuous Analytics: Rethinking Query Processing in a Network-Effect World. CIDR 2009

[66] Gedik Bugra, Andrade Henrique, Wu Kun-Lung, S. Yu Philip, Doo Myungcheol: SPADE: the system s declarative stream processing engine. SIGMOD 2008:1123-1134

[67] Gehrke Johannes, Korn Flip, Srivastava Divesh: On Computing Correlated Aggregates Over Continual Data Streams. SIGMOD 2001:13-24

[68] Gilbert C. Anna, Kotidis Yannis, Muthukrishnan S., Strauss Martin: Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. VLDB 2001:79-88

[69] Gunduz Sule, Ozsu M. Tamer: A Web page prediction model based on click-stream tree representation of user behavior. KDD 2003:535-540

[70] Ganguly Sumit, Garofalakis N. Minos, Rastogi Rajeev: Processing Set Expressions over Continuous Update Streams. SIGMOD 2003:265-276

[71] Golab Lukasz, Ozsu M. Tamer: Issues in data stream management. SIGMOD Record (SIGMOD) 32(2):5-14 (2003)

[72] Golab Lukasz, Ozsu M. Tamer: Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. VLDB 2003:500-511

[73] Gonzalez Hector, Han Jiawei, Li Xiaolei: FlowCube: Constructuing RFID FlowCubes for Multi-Dimensional Analysis of Commodity Flows. VLDB 2006:834-845

[74] Graefe Goetz: Query Evaluation Techniques for Large Databases. ACM Comput. Surv. (CSUR) 25(2):73-170 (1993)

[75] Gray Jim, Chaudhuri Surajit, Bosworth Adam, Layman Andrew, Reichart Don, Venkatrao Murali, Pellow Frank, Pirahesh Hamid: Data Cube: A Relational Aggregation Operator

Generalizing Group-by, Cross-Tab, and Sub Totals. Data Min. Knowl. Discov. (DATAMINE) 1(1):29-53 (1997)

[76] Guirguis Shenoda, A. Sharaf Mohamed, K. Chrysanthis Panos, Labrinidis Alexandros: Three-Level Processing of Multiple Aggregate Continuous Queries. In 28th International Conference on Data Engineering. ICDE'12, 929-940.

[77] Guha Sudipto, Gunopulos Dimitrios, Koudas Nick: Correlating synchronous and asynchronous data streams. KDD 2003:529-534

[78] Han Jiawei, Chen Yixin, Dong Guozhu, Pei Jian, Wah W. Benjamin, Wang Jianyong, Dora Y. Cai: Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams. Distributed and Parallel Databases (DPD) 18(2):173-197 (2005)

[79] Hammad A. Moustafa, Aref G. Walid, Elmagarmid K. Ahmed: Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. SSDBM 2003:75-84

[80] Hellerstein M. Joseph, Haas J. Peter, Wang J. Helen: Online Aggregation. SIGMOD 1997:171-182

[81] Hoppe Andrzej, Gryz Jarek: Stream Processing in a Relational Database: a Case Study. In 7th International Database Engineering and Applications Symposium. IDEAS'07, 216-224.

[82] Hsu Meichun, Chen Qiming, Wu Ren, Zhang Bin, Zeller Hans: Generalized UDF for Analytics Inside Database Engine. Generalized UDF for Analytics Inside Database Engine. In 11th International Conference Web-Age Information Management. WAIM '10, 742-754.

[83] Jain Navendu, Amini Lisa, Andrade Henrique, King Richard, Park Yoonho, Selo Philippe, Venkatramani Chitra: Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGMOD'06, 431-442.

[84] Jain Namit, Mishra Shailendra, Srinivasan Anand, Gehrke Johannes, Widom Jennifer, Balakrishnan Hari, Cetintemel Ugur, Cherniack Mitch, Tibbetts Richard, Zdonik B. Stanley: aming SQL standard. Towards a streaming SQL standard. In Proceedings of the Very Large Databases. PVLDB 1, 2, (August 2008), 1379-1390.

[85] Johnson Theodore, Chatziantoniou Damianos: Extending Complex Ad-Hoc OLAP. CIKM 1999:170-179

[86] Krishnamurthy Sailesh, Chandrasekaran Sirish, Cooper Owen, Deshpande Amol, Franklin J. Michael, M. Hellerstein Joseph, Hong Wei, Madden Samuel, Reiss Frederick, Shah A. Mehul: TelegraphCQ: An Architectural Status Report. IEEE Data Eng. Bull. (DEBU) 26(1):11-18 (2003)

[87] Kumar Vibhore, Andrade Henrique, Gedik Bugra, Wu Kun-Lung: DEDUCE: at the intersection of MapReduce and stream processing. In 13th International Conference on Extending Database Technology. EDBT'10, 657-662.

[88] Larson Per-Ake, Zhou Jingren: Efficient Maintenance of Materialized Outer-Join Views. ICDE 2007:56-65

[89] Law Yan-Nei, Wang Haixun, Zaniolo Carlo: Query Languages and Data Models for Database Sequences and Data Streams. VLDB 2004:492-503

[90] Lee L. Hau, 2007. Peering through a glass darkly. International Commerce Review 7 (1), 60–78.

[91] Lerner Alberto, Shasha Dennis: The Virtues and Challenges of Ad Hoc + Streams Querying in Finance. IEEE Data Eng. Bull. (DEBU) 26(1):49-56 (2003)

[92] Li Jin, Maier David, Tufte Kristin, Papadimos Vassilis, Tucker A. Peter: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Record 34, 1, (March 2005), 39-44.

[93] Liarou Erietta, Goncalves Romulo, Idreos Stratos: Exploiting the power of relational databases for efficient stream processing. In 12th International Conference on Extending Database Technology. EDBT'09, 323-334.

[94] Lim Ee-Peng, Chen Hsinchun, Chen Guoqing: Business Intelligence and Analytics: Research Directions. ACM Trans. Management Inf. Syst. (TMIS) 3(4):17 (2013)

[95] Liu Ling, Pu Calton, Tang Wei: Continual Queries for Internet Scale Event-Driven Information Delivery. IEEE Trans. Knowl. Data Eng. (TKDE) 11(4):610-628 (1999)

[96] Liu Ling, Pu Calton, Tang Wei, Buttler David, Biggs John, Zhou Tong, Benninghoff Paul, Han Wei, Yu Fenghua: CQ: A Personalized Update Monitoring Toolkit. SIGMOD 1998:547-549

[97] Liu Bin, Jagadish H. V.: A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. ICDE 2009:417-428

[98] Loesing Simon, Hentschel Martin, Kraska Tim, Kossmann Donald: Stormy: an elastic and highly available streaming service in the cloud. In Data ANAlytics in the Cloud. EDBT/ICDT Workshop, DANAC'12, 55-60.

[99] Luo Chang, Thakkar Hetal, Wang Haixun, Zaniolo Carlo: A native extension of SQL for mining data streams. SIGMOD 2005:873-875

[100] Madden Samuel, Franklin J. Michael: Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. ICDE 2002:555-566

[101] Madden Samuel, Franklin J. Michael, Hellerstein M. Joseph, Hong Wei: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. (TODS) 30(1):122-173 (2005)

[102] Madden Samuel, Franklin J. Michael, Hellerstein M. Joseph, Hong Wei: The Design of an Acquisitional Query Processor For Sensor Networks. SIGMOD 2003:491-502

[103] Madden Samuel, Shah A. Mehul, Hellerstein M. Joseph, Raman Vijayshankar: Continuously adaptive continuous queries over streams. SIGMOD 2002:49-60

[104] Mainwaring M. Alan, Culler E. David, Polastre Joseph, Szewczyk Robert, Anderson John: Wireless sensor networks for habitat monitoring. WSNA 2002:88-97

[105] Mamoulis Nikos: Efficient Processing of Joins on Set-valued Attributes. SIGMOD 2003:157-168

[106] McCarthy R. Dennis, Dayal Umeshwar: The Architecture Of An Active Data Base Management System. SIGMOD 1989:215-224

[107] Manku Gurmeet Singh, Motwani Rajeev: Approximate Frequency Counts over Data Streams. VLDB 2002:346-357

[108] Motwani Rajeev, Widom Jennifer, Arasu Arvind, Babcock Brian, Babu Shivnath, Datar Mayur, Singh Manku Gurmeet, Olston Chris, Rosenstein Justin, Varma Rohit: Query Processing, Approximation, and Resource Management in a Data Stream Management System. CIDR 2003

[109] Muthukrishnan S.: Data Streams: Algorithms and Applications. Foundations and Trends in Theoretical Computer Science (FTTCS) 1(2) (2005)

[110] Nykiel Tomasz, Potamias Michalis, Mishra Chaitanya, Kollios George, Koudas Nick: MRShare: Sharing Across Multiple Queries in MapReduce. In Proceedings of the Very Large Databases. PVLDB 3, 1, (September 2010), 494-505.

[111] Olston Christopher, Reed Benjamin, Srivastava Utkarsh, Kumar Ravi, Tomkins Andrew: Pig latin: a not-so-foreign language for data processing. SIGMOD 2008:1099-1110

[112] Olston Christopher, Chiou Greg, Chitnis Laukik, Liu Francis, Han Yiping, Larsson Mattias, Neumann Andreas, Rao B. N. Vellanki, Sankarasubramanian Vijayanand, Seth Siddharth, Tian Chao, ZiCornell Topher, Wang Xiaodan: Nova: continuous Pig/Hadoop workflows. In

Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGMOD'11, 1081-1090.

[113] Park Jaekwan, Hong Bonghee, Ban ChaeHoon: A Continuous Query Index for Processing Queries on RFID Data Stream. RTCSA 2007:138-145

[114] Pavlo Andrew, Paulson Erik, Rasin Alexander, J. Abadi Daniel, J. DeWitt David, Madden Samuel, Stonebraker Michael: A comparison of approaches to large-scale data analysis. SIGMOD 2009:165-178

[115] Polyzotis Neoklis, Skiadopoulos Spiros, Vassiliadis Panos, Simitsis Alkis, Frantzell Nils-Erik: Supporting Streaming Updates in an Active Data Warehouse. ICDE 2007:476-485

[116] Pramatari Katerina, Doukidis George, Kourouthanassis Panos, 2005. Towards 'smarter' supply and demand-chain collaboration practices enabled by RFID technology. In: Vervest, P., Van Heck, E., Preiss, K., Pau, L.F. (Eds.), Smart Business Networks. Springer Verlag, ISBN 3-540-22840-3.

[117] Rao Jun, Doraiswamy Sangeeta, Thakkar Hetal, Colby S. Latha: A Deferred Cleansing Method for RFID Data Analytics. VLDB 2006:175-186

[118] Ross A. Kenneth, Srivastava Divesh: Fast Computation of Sparse Datacubes. VLDB 1997:116-125

[119] Ross A. Kenneth, Srivastava Divesh, Chatziantoniou Damianos: Complex Aggregation at Multiple Granularities. EDBT 1998:263-277

[120] Roth A, Mark, Korth F. Henry, Silberschatz Abraham: Extended Algebra and Calculus for Nested Relational Databases. ACM Trans. Database Syst. (TODS) 13(4):389-417 (1988)

[121] Rundensteiner A. Elke, Ding Luping, Sutherland M. Timothy, Zhu Yali, Pielech Bradford, Mehta K. Nishant: CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. VLDB 2004:1353-1356

[122] Schiefer Josef, List Beate, Bruckner M. Robert: Process Data Store: A Real-Time Data Store for Monitoring Business Processes. DEXA 2003:760-770

[123] Schreier Ulf, Pirahesh Hamid, Agrawal Rakesh, Mohan C.: Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. VLDB 1991:469-478

[124] Seligman J. Leonard, E. Lehner Paul, P. Smith Kenneth, Elsaesser Chris, Mattox David: Decision-Centric Information Monitoring. J. Intell. Inf. Syst. (JIIS) 14(1):29-50 (2000)

[125] Spring Jesper Honig, Privat Jean, Guerraoui Rachid, Vitek Jan: Streamflex: high-throughput stream programming in java. In Proceedings of the 22nd Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA'07, 211-228.

[126] Steenhagen J. Hennie, Apers M. G. Peter, Blanken M. Henk: Optimization of Nested Queries in a Complex Object Model. EDBT 1994:337-350

[127] Stonebraker Michael, Abadi J. Daniel, DeWitt J. David, Madden Samuel, Paulson Erik, Pavlo Andrew, Rasin Alexander: MapReduce and parallel DBMSs: friends or foes? Commun. ACM (CACM) 53(1):64-71 (2010)

[128] Stonebraker Michael, Cetintemel Ugur, Zdonik B. Stanley: The 8 requirements of real-time stream processing. SIGMOD Record 34, (March 2005), 42-47.

[129] STREAM: The Stanford Stream Data Manager, User Guide and Design Document. http://infolab.stanford.edu/stream/code/user.pdf (last accessed 22-May-2013).

[130] Subramani, M., 2004. How do suppliers benefit from information technology use in supply chain relationships? MIS Quarterly 28 (1), 45–73.

[131] Sullivan Mark: Tribeca: A Stream Database Manager for Network Traffic Analysis. VLDB 1996:594

[132] Tatbul Nesime: Streaming data integration: Challenges and opportunities. In 2nd International Workshop on New Trends in Information Integration. NTII'10, 155-158.

[133] Tatbul Nesime, Cetintemel Ugur, Zdonik B. Stanley, Cherniack Mitch, Stonebraker Michael: Load Shedding in a Data Stream Manager. VLDB 2003:309-320

[134] Terry B. Douglas, Goldberg David, Nichols A. David, Oki M. Brian: Continuous Queries over Append-Only Databases. SIGMOD 1992:321-330

[135] The Stanford Stream Data Management (STREAM) Project. Available at: http://www-db.stanford.edu/stream (last accessed March 2013)

[136] Thies William, Karczmarek Michal, Amarasinghe P. Saman: StreamIt: A Language for Streaming Applications. In 11th International Conference Compiler Construction. CC'02, 179-196.

[137] Thusoo Ashish, Sen Sarma Joydeep, Jain Namit, Shao Zheng, Chakka Prasad, Zhang Ning, Anthony Suresh, Liu Hao, Murthy Raghotham: Hive - a petabyte scale data warehouse using Hadoop. ICDE 2010:996-1005

[138] Tucker A. Peter, Maier David, Sheard Tim, Fegaras Leonidas: Exploiting Punctuation Semantics in Continuous Data Streams. IEEE Trans. Knowl. Data Eng. 15(3): 555-568 (2003)

[139] Tucker A. Peter, Maier David, Sheard Tim, Stephens Paul: Using Punctuation Schemes to Characterize Strategies for Querying over Data Streams. IEEE Trans. Knowl. Data Eng. 19(9): 1227-1240 (2007)

[140] Vossough Ehsan: A System for Processing Continuous Queries over Infinite Data Streams. In 15th International Conference Database and Expert Systems Applications. DEXA'04, 720-729.

[141] Wang Haixun, Zaniolo Carlo, Luo Chang: ATLAS: A Small but Complete SQL Extension for Data Mining and Data Streams. VLDB 2003:1113-1116

[142] Wang Fusheng, Liu Shaorong, Liu Peiya, Bai Yijian: Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams. EDBT 2006:588-607

[143] Witkowski Andrew, Bellamkonda Srikanth, Bozkaya Tolga, Naimat Aman, Sheng Lei, Subramanian Sankar, Waingold Allison: Query By Excel. VLDB 2005:1204-1215

[144] Witkowski Andrew, Bellamkonda Srikanth, Bozkaya Tolga, Dorman Gregory, Folkert Nathan, Gupta Abhinav, Sheng Lei, Subramanian Sankar: Spreadsheets in RDBMS for OLAP. SIGMOD 2003:52-63

[145] Wang Fusheng, Liu Peiya: Temporal Management of RFID Data. VLDB 2005:1128-1139

[146] Woo Alec, Seth Siddharth, Olson Tim, Liu Jie, Zhao Feng: A spreadsheet approach to programming and managing sensor networks. IPSN 2006:424-431

[147] Wu Eugene, Diao Yanlei, Rizvi Shariq: High-performance complex event processing over streams. In Proceedings of the ACM SIGMOD International Conference on Management of Data.SIGMOD'06,407-418.

[148] Yang Di, Rundensteiner A. Elke, Ward O. Matthew: Shared execution strategy for neighbor-based pattern mining requests over streaming windows. ACM Trans. Database Sys. 37, 1, (Feb 2012), 5.

[149] Yao Yong, Gehrke Johannes: The Cougar Approach to In-Network Query Processing in Sensor Networks. SIGMOD Record (SIGMOD) 31(3):9-18 (2002

[150] Yan P. Weipeng, Larson Per-Ake: Eager Aggregation and Lazy Aggregation. VLDB 1995:345-357

[151] Yin Xuepeng, Pedersen Torben Bach: What Can Hierarchies Do for Data Streams? BIRTE 2006:4-19

[152] Zdonik B. Stanley, Stonebraker Michael, Cherniack Mitch, Cetintemel Ugur, Balazinska Magdalena, Balakrishnan Hari: The Aurora and Medusa Projects. IEEE Data Eng. Bull. (DEBU) 26(1):3-10 (2003)

[153] Zhang Donghui, Gunopulos Dimitrios, Tsotras Vassilis J., Seeger Bernhard: Temporal Aggregation over Data Streams Using Multiple Granularities. EDBT 2002:646-663

[154] Zhang Donghui, Gunopulos Dimitrios, Tsotras J. Vassilis, Seeger Bernhard: Temporal and spatio-temporal aggregations over data streams using multiple time granularities. Inf. Syst. (IS) 28(1-2):61-84 (2003)

[155] Zhang Rui, Koudas Nick, Chin Ooi Beng, Srivastava Divesh: Multiple Aggregations Over Data Streams. In Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGMOD'05, 299-310

[156] Zhu Yunyue, Shasha Dennis: StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. VLDB 2002:358-369

[157] Zhu Yunyue, Shasha Dennis: Efficient elastic burst detection in data streams. KDD 2003:336-345