MonetDB/DataCell

Leveraging the Column-store Database Technology

for

Efficient and Scalable Stream Processing

Erietta Liarou

MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Agnietenkapel
op dinsdag 22 januari 2013, te 14:00 uur

door Erietta Liarou
geboren te Athene, Griekenland

Promotiecommissie

Promotor:	Prof. dr. M.L. Kersten
Copromotor:	Dr. S. Manegold
Overige Leden:	Prof. dr. M. de Rijke
	Prof. dr. L. Hardman
	Prof. dr. M. Koubarakis
	Prof. dr. M. T. Özsu

Faculteit der Natuurwetenschapen, Wiskunde en Informatica

The research reported in this thesis has been partially carried out at *CWI*, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme *Database Architectures and Information Access*, a subdivision of the research cluster *Information Systems*.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No 2013-02.
The research reported in this thesis has been carried out under the auspices of *SIKS*, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis was partially funded by the BRICKS project.

ISBN 978-90-6196-562-6

Η Ιθάκη σ΄ έδωσε τ΄ ωραίο ταξίδι.
Χωρίς αυτήν δεν θα βγαινες στον δρόμο.
Άλλα δεν έχει να σε δώσει πια.

Κι αν πτωχική την βρείς, η Ιθάκη δεν σε γέλασε.
Έτσι σοφός που έγινες, με τόση πείρα,
ήδη θα το κατάλαβες οι Ιθάκες τι σημαίνουν.

Κ.Π. Καβάφης

Ithaka gave you the marvelous journey.
Without her you would not have set out.
She has nothing left to give you now.

And if you find her poor, Ithaka won't have fooled you.
Wise as you will have become, so full of experience,
you will have understood by then what these Ithakas
mean.

C.P. Cavafy (1863-1933)

Contents

Chapter 1

Introduction

1.1 From the Theory of Forms to the Knowledge Boom

Knowledge is a concept that the ancient philosophers studied more than 2000 years ago. Plato and Aristotle, already in 400 BC, tried to understand and define what is knowledge and how it is created and acquired. In the *Theory of Ideas* (or *Forms*), Plato argues that the knowledge is already created and given to us from a universal metaphysical level. In this way, he claims that we learn in this life by remembering and trying to imitate the principles that our soul already encloses from the world of *Ideas*. On the other hand, Aristotle, the most important student of Plato for twenty years, supported that the observation and the study of particular phenomena will lead us to the real knowledge.

Over the years, the definition of knowledge constituted an ongoing debate among philosophers and the triptych of the *true justified belief* has been challenged by modern epistemologists several times (Gettier, 1963). However, scientists, through the steps they follow in research, help us to realize how we come to the genesis of scientific knowledge. In that sense, we could say that scientific research agrees with the empirical aristotelean philosophy, since it depends on the observation, the measurement and the study of evidence. The collection of data, its process and evaluation constitute critical steps that transform the pure data into information, and in turn the latter one into scientific knowledge.

Even without a globally agreed definition of what knowledge is, it is a universal conviction that knowledge constitutes a very powerful and valuable *good*.

Apart from the science and technology, connected to knowledge by an endless two-way bond, almost all the aspects of everyday life depend on knowledge and can be improved by using the existing know-how, saving us from constantly *reinventing the wheel*.

Today, tons of information surrounds us where we can acquire from many and different sources, such as books, mass media, social networks, etc. In particular, the World Wide Web consists of a bottomless source of new information, readily available at our fingertips. The amount of data being generated every day is still growing exponentially; it seems that for the first time in history there is more information than we can even process and consume. However, information alone does not directly bring us closer to the philosopher's *true knowledge*. To this end, it becomes a matter of major importance to find ways to manage, analyze, selectively discard and exploit all this data we collect and turn it into (useful) knowledge.

1.2 Data Management

The father of history, Herodotus, aptly predicates that *"Of all men's miseries the bitterest is to know so much and to have control over nothing"*. This quote was not randomly said by Herodotus, the person who first realized the importance of collecting, confirming, writing, organizing and delivering to the next generations historic material that was taking place at his time. To have control over our knowledge thesaurus is an important issue, and it becomes even more difficult the more information we have to access and the more we need to combine multiple data sets.

Taking a closer look at the technological achievements of the last century, we see that they drastically affected the creation of knowledge. In the mid of 20th century, the technological evolution and most importantly transistor's invention, brought us closer to the information technology revolution. The reason for that was twofold; firstly it allowed the miniaturization of all modern electronics that brought on the digital information age and secondly it triggered the creation of cheaper and more powerful computational units that were able to store and process the generated data. A few decades later, the microprocessor made feasible the generation and process of such large amount of data that one hundred years ago it was hard, even impossible, to manipulate in a manual way.

To this end, data management very soon became the main concern of information technology. The big firms and organizations that were continuously generating data on a daily basis, kept asking for new technologies that would

allow them to process, analyze, visualize and manage their data in a more efficient way. Furthermore, fundamental sciences such as astronomy and biology came across even more demanding issues, since the data they started producing and want to discover patterns exceeds by far the needs of all the other fields. The geneticist Richard Lewontin, in his book titled *Biology as Ideology: The Doctrine of DNA*, characteristically states that the knowledge itself is not powerful enough, but it further empowers only those who have or can acquire the power to use it.

1.2.1 Database Management Systems

The necessity for new information technologies became very soon a clear research target for the computer science community and the first data management prototypes came already around the 1960s. These systems were mostly customized and used only in large organizations, who could afford the extremely high costs. Back then, a database was designed to be the system that would be responsible to store, organize and access enormous quantities of digital data in an automatic and efficient way.

One of the first Database Management Systems (DBMSs), called IMS, was built by IBM back in 1968 for NASA's Apollo space program. Since then, we meet the database technology almost in every aspect of our electronic life. Shopping at a store, borrowing a book from the library, making bank transactions, or requesting student transcripts are only some of the examples that imply the existence of a database. A DBMS typically consists of the appropriate software that provides the insertion of new data in the database, the modification and deletion of existing data and more importantly the efficient search and retrieval of data that qualifies the requester's constrains.

A milestone in the database research was the *relational model*, originally formulated and proposed by Edgar Codd in the 1970s (Codd, 1970). There the data is organized into a set of *tables*, which are *related* to each other in many and different ways. Each table follows a predefined schema and each record (tuple) stored in a table must also respect the same schema in order to be valid. One of the nice properties of the relational model is that we can add and access data, without reorganizing the tables every time we do so. A table can have many records and each record can have many fields (attributes). In the relational model, we distinguish the tuples of a table using a unique key, called *primary key*. Another type of keys are the *foreign keys*, used to create links between tables. In this way, the navigation among tables and the retrieval of those entries that qualify the user's request was dramatically improved, com-

pared to other previously used models, e.g., hierarchical and network database model. Consider for example the case of a university database; there we could have several different tables such as the "students", the "courses" and the "professors" tables. For each student we may keep a record, marked by a unique student ID, and other attributes that better describe the student, e.g., name, date of birth and address. Moreover, we may keep track of which courses he has successfully passed and the evaluation he has received, by linking the two different tables, i.e., the tables "students" and "courses". Also, each course is linked to the "professors" table to indicate who is teaching it in each semester. By organizing our data in that structured way, we can easily navigate through the tables and retrieve any combinative query, e.g., give me all the professors that teach a course with success rate greater than 70% and average student grades 8 out of 10, for at least 5 years in a row.

The success of the relational model, mainly comes from the fact that it works in a *declarative* way. Relational databases are extremely easy to customize to fit almost any kind of data. The user is able to access and manipulate his data without being involved in technical decisions that have to do with designing *how* the data will be stored and *how* the requests are going to be executed. Through SQL, a declarative query language, the user obtains full control over the data and is able to describe in an abstract way what kind of information he is interested in, keeping his hands clean of any internal low-level system specifications. On the contrary, the DBMS is in charge to decide *autonomously* what is the best way to organize and physically store the data, and designs the appropriate strategy for getting the user's queries answered.

Apart from the powerfulness and the flexibility that the database systems provide, another reason that contributed to their wide use is coming for the fact that they are generic enough and able to handle multiple users at the same time. A DBMS has the appropriate mechanisms to always ensure data integrity, despite multiple concurrent users or different application programs are accessing the same database. The *ACID* properties are the main rules at the database cookbook, that guarantee safe transaction executions. In short, the first rule, called *atomicity*, implies that once a transaction starts it should be fully completed otherwise it should become in a status as if it never happened. All transactions must maintain the *consistency* of the database. Two concurrent transactions must be *isolated* and not interfere with each other while happening. Finally, once a transaction is completed the DBMS guarantees that its impact to the database will be *durable* from here on. Working under these rules and balancing with mastery between reliability and performance, the database systems very soon convinced the big firms and organizations that they are trustworthy

and skilled to manage their valuable data.

Database management systems were created to provide persistent data storage and an efficient and reliable answering mechanism. It is the suggested complete software solution, when the application scenario prerequisites that the data is a priori known and relatively static. A DBMS typically stores, organizes, indexes and prepares the stored data to the best of its knowledge and it becomes ready to accept and immediately answer the potential queries that will be posed in the future. Once a request comes, the database system syntactically and semantically analyzes it, and based on a predefined set of rules, as well as previously acquired query processing experience (e.g., statistics, indices), it decides what is the best query plan to use for deriving the matching answers. The execution engine precisely follows the designed query plan, and evaluates the query over the data that is currently stored in the database.

Database systems constitute an *alive* evolving research field for the past 50 years. Their quick commercial exploitation, challenged their initial capabilities and brought out their potential weak points. Many research subfields have been created to fill in the gaps and strengthen their features; some focus at the core level of query processing and optimization, and others cope with higher level topics such as language interfaces, distributed and parallel processing, privacy and security issues or research related to web applications. The diverse market needs motivate the expansion of different database architectures. Both a small business and an astronomical data center may use a database system, but their fundamentally different requirements drove researchers and developers to design different database architectures and solutions. Many open source prototypes, as such PostgreSQL, MySQL and MonetDB not only survived through the years but also keep leading the database research. Moreover the big players of the commercial arena, such as Oracle, IBM and Microsoft, are continuously investing in the ongoing database technology evolution.

Half a century after the first prototypes, database systems are still the center of attention of information technology. It seems that there are still more to research, since new data sources challenge their capabilities and performance every day.

1.3 Data Stream Management

Plenty of application scenarios fit in the traditional database processing scheme. However, a new type of applications, called *data streams*, that came a few decades after the establishment of the DBMSs, could not be satisfied by that

model. In the data stream scenario, we have to deal with the *continual* generation and processing of an infinite flood of data (stream). Queries on the other hand, appear to be persistent, namely once they are submitted they remain active forever or for at least a long period of time. These two fundamental differences on the queries and data lifetime, became enough to make very soon clear that database systems were not skilled to handle such applications. This way, the computer scientists started looking for new system architectures that could fulfill the new requirements.

A potentially large application domain stimulates the creation of data stream management systems (DSMSs). Sensors, organized in wireless networks, that continuously measure physical, biological or chemical input, nicely fit in the data stream model. The sensors produce streams of data that continuously should be analyzed in real-time to keep track of environmental conditions and detect anomalies in case they happened. Smoke detectors, health-care monitors and traffic controllers are only some simple examples that fall in that application scenario. Furthermore, sensor networks take control over smart building design, or can be used to wildlife tracking systems to give rich information to animal biologists.

In the same line, network monitoring systems continuously need to analyze the network traffic to catch potential problems, such as unusual activity, delays, server crashes and bottlenecks. They derive information enclosed in IP packets, while they are passing through the network, and generate the appropriate alerts when they diagnose a problematic behavior.

Financial trading applications is another scenario that meets the data stream requirements. The idea is the same also here, continuous fast updated information coming from different sources, should be analyzed and combined to accomplish profitable transactions. The list of applications that inspired the creation of data stream systems is long. For example, consider that the World Wide Web provides a plethora of streaming opportunities through web feeds. Users are able to subscribe to interesting sources of information and they are automatically notified when new data is available.

DSMS vs. DBMS

Let us now see in more detail what are the main fundamental differences between the database and the data stream application scenario.

- **Continuous query processing.**

  In a stream application, we need mechanisms to support *long-standing/*

continuous queries over data that is continuously updated from the environment. The queries are issued once and then they stay active for a long time, monitoring the incoming data. On the contrary in a database scenario, the user poses a query and receives the corresponding answers only once. If he wants to check for potentially different answers later, he should re-submit the same query to the database. The database traditionally, evaluates that queries all over again, without taking into account the previous evaluation.

- **Data lifetime.**

  In the database scenario, the data is characterized as persistent. Updates over the data are expected but their rate is less frequent than the incoming rate of queries. On the other side, in a stream application, we should be prepared to handle an infinite sequence of data in real-time. Typically once the data comes it is analyzed against waiting queries and then it is forgotten.

- **Pull vs. Push model**

  Taking into account the way that a DBMS treats data and queries, we could say that it follows a pure pull-based model, since each time a new query arrives, the engine pulls the data from the disk to search for answers. On the contrary, a typical DSMS works in a push-based way, pushing the incoming streams to meet the interested waiting queries.

- **Real-time processing**

  In data stream applications, it is very important to achieve real-time processing. Delays may affect answer's validity and also could produce system bottlenecks, since more data will be continuously collected. A data stream engine should be alert and process the incoming data in real-time.

- **Workload fluctuations**

  Data stream arrival may vary dramatically. There are application scenarios with low data input rates, such as sensors that update their measurements every one minute, or other cases where we have to deal with extremely high input stream rates. For example, most recent generation of satellites provides ground reception rates of 300 Mbit/sec and 800 Mbit/sec. Environment and workload changes call for adaptive processing strategies at the query evaluation level to achieve the best query response time. In databases we have to deal with workload variation too, but in

terms of queries. In that case, it may become mandatory to update our indices over the stored data.

- **Window processing**

  As we have already mentioned a stream is an infinite sequence of data. Given that hardware and software have limitations, we also need to limit the maximum amount of data we can gather and process within a given time budget. The initial stream processing models were very simple; they were producing answers by considering only one incoming tuple at a time. The window processing model came as an intermediate solution between single tuple and database processing. In this case, the system produces answers considering a number of collected stream tuples, instead of just a single one. The window processing model increases the expressiveness of stream systems, allowing for aggregations and joins in addition to simple filtering queries.

- **Query languages**

  Taking all the previous factors into account, the existence of new data models and query languages was necessary for the establishments of the DSMSs. The language used in relational databases, was not sufficient to represent the different nature and semantics of stream data and queries.

Given these differences, and the unique characteristics and needs of continuous query processing, the pioneering DSMS architects naturally considered that the existing DBMS architectures are inadequate to support stream processing and achieve the desired performance. Another aspect is that the initial stream applications had quite simple requirements in terms of query processing. This made the existing DBMS systems look overloaded with functionalities. These factors led researchers to design and build new architectures from scratch and several DSMS solutions have been proposed over the last years giving birth to very interesting ideas and system architectures, e.g., (Babcock et al., 2004; Balakrishnan et al., 2004; Chandrasekaran et al., 2003; Chen et al., 2000; Cranor et al., 2003; Girod et al., 2007).

1.4 The DataCell: a DSMS into the heart of a DBMS

1.4.1 Motivation

As we discussed earlier, the diverse needs for persistent data management and continuous queries processing, brought two different system architectures. However, the data management evolution does not seem to stop here. The last few years a new processing paradigm is born, where incoming data (stream) needs to quickly be analyzed and possibly combined with existing data to discover trends and patterns. Subsequently, the new data enters the data warehouse and is stored as normal for further analysis if necessary.

Natural sciences such as astronomy and biology that deal with large amounts of data, also motivate that paradigm. In 2015 the astronomers will be able to scan and catalog the entire night sky from a mountain-top in Chile, recording 30 Terabytes of data every night which incrementally will result in an absolutely massive 150 Petabyte database (over the operation period of ten years). It will be capturing changes to the observable universe evaluating huge statistical calculations over the entire database. Another characteristic data-driven example is the Large Hadron Collider (LHC) (LHC, 2010), a particle accelerator that will revolutionize our understanding for the universe, generating almost 40 Terabytes of data every day and collecting 15 petabytes of data annually. The same model stands for modern data warehouses which enrich their data on a daily basis creating a strong need for quick reaction and combination of scalable stream and traditional processing (Winter and Kostamaa, 2010).

In this new paradigm incoming streams of data need to quickly be analyzed and possibly combined with existing data to discover trends and patterns. We need scalable query processing that can combine continuous querying for fast reaction to incoming data with traditional querying for access to the existing data. However, neither pure database technology nor pure stream technology are designed for this purpose.

In this thesis, we propose that a complete integration of database and streaming technology is the way to go. We focus on the design, study and development of such a system that integrates both streaming and database technologies in the most natural way. A fully functional stream engine, called DataCell, is designed on top of an extensible DBMS kernel. Our goal is to fully exploit the generic storage and execution engine of the DBMS as well as its complete optimizer stack. The end goal is a single system that does combine properties and

features from both the database world and the stream world, and thus achieves efficient performance for both one-time and continuous queries.

1.4.2 The Basics

The ultimate goal of this thesis is to support full data management of persistent and streaming data within an integrated processing kernel. Instead of building a new system from scratch we opt to work over an extensible DBMS kernel such that we can exploit mature techniques and algorithms in the area of database systems. The challenge becomes how to extend such a scalable system such that it supports stream processing in addition to one-time processing.

Stream researchers in the past argued that this is not feasible as it would be very inefficient. DataCell shows that this is not true anymore, and it successfully combines both paradigms.

Our design and implementation is over the MonetDB system. MonetDB is an open source column-store database management system, developed and maintain at CWI. Several aspects of MonetDB make this research possible. For instance, MonetDB allows for easy manipulation and extension of its optimizer module which allows us to easily introduce new optimizer rules specific for DataCell while at the same time exploiting all existing optimizer rules a DBMS has to offer. In addition, MonetDB is one of the leading column-store systems. We heavily exploit its column-store nature in our techniques to speed up stream processing exploiting critical column-store features such as vectorization.

The main idea is that when stream tuples arrive into the system, they are immediately stored in (appended to) a new kind of lightweight tables, called *baskets*. By temporarily collecting tuples into baskets, we can evaluate the continuous queries over the baskets as if they were normal one-time queries and thus we can reuse any kind of algorithm and optimization designed for a modern DBMS. Once a tuple has been seen by all relevant queries/operators, it is *dropped* from its basket.

Continuous query plans are represented by *factories*, i.e., a kind of co-routine, whose semantics are extended to align with table producing SQL functions. Each factory encloses a query plan that once it is evaluated it produces a partial result at each call. For this, a factory continuously reads data from the input baskets, evaluates its query plan and creates a result set, which it then places in its output baskets. The factory remains active as long as the continuous query remains in the systems, and it is always ready to consume incoming stream data.

The execution of the factories is orchestrated by the DataCell *scheduler*. The

firing condition is aligned to arrival of events, once there are tuples that may be relevant to a waiting query we trigger its evaluation over these tuples. Furthermore, the scheduler manages the time constraints attached to event handling, which leads to possibly delaying events in their baskets for some time. One important merit of the DataCell architecture, is the natural integration of baskets and tables within the same processing fabric. A single factory can interact both with tables and baskets, this way we can naturally support queries interweaving the basic components of both models.

By introducing the baskets, the factories and the DataCell scheduler, our architecture becomes able to proceed sufficiently data streams, without also losing any database functionality. That is the natural first step that covers the gap between the two incompatible processing models. However, numerous research and technical questions immediately arise. The most prominent issues are the ability to provide specialized stream functionality and hindrances to guarantee real-time constraints for event handling. Also, we need to cope with (and exploit) similarities between the many standing queries, in order to deal with high performance requirements.

1.4.3 Research Challenges

It is a major challenge for the DataCell architecture to efficiently support and integrate all specialized stream features. The above description gives the first directions that allow the exploration of quite flexible strategies, once we have to deal with low latency deadlines or multi-query processing.

The road-map for DataCell research calls for innovation in many important aspects of stream processing and the combination with already stored data. Thus, one can distinguish between challenges that come from the fact that stream processing is performed in a DBMS and challenges that arise by combining the two query processing paradigms in one.

Regarding the first challenge, the goal is to provide all essential streaming functionality and features without losing the DBMS's strong storage and querying capabilities. We draw a path where most of the streaming functionality is provided via plan rewriting and minimal lower level operator changes. For example, resource management, scheduling, and optimization in the presence of numerous queries is a critical topic. Similarly to incremental processing, this area has received a lot of attention with innovative solutions, e.g., (Sharaf et al., 2008). DataCell offers all the available ingredients to achieve similar levels of multi-query optimizations while keeping the underlying generic engine intact. For example, a single factory (i.e., plan) may dynamically split into multiple

pieces or merge with other relevant factories to allow for efficient sharing of processing costs leading to very interesting scenarios in how the network of factories and baskets is organized and adapts. Again, these issues can be resolved at a higher level through plan rewriting. The intermediates created for incremental processing can be reused by many queries, while partitioning and scheduling decisions can also adapt to the new parameters. We have the appropriate technology to make multiple queries to cache and exploit intermediates in a column-store kernel.

Regarding the second challenge, a plethora of rich topics arise especially when optimization becomes an issue. For example, query plans that touch both streaming data and regular tables might require new optimizer rules or adaptations of the current ones. Overall, DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective.

1.4.4 Contributions

The particular contributions of this thesis can be summarized as follows:

(1) **A new Stream Paradigm.** We show that the past belief that stream query processing requires a specialized engine only for stream processing is not sufficient anymore, especially due to the increasing scalability requirements.

(2) **DataCell architecture.** We introduce the basic DataCell architecture, to exploit the notion of scalable systems that can provide both streaming and database functionality. We describe what are the minimal additions that allow for stream processing within a DBMS kernel.

(3) **Incremental processing.** We show how to efficiently support core streaming functionalities in DataCell, i.e., incremental stream processing and window-based processing.

(4) **Multi-query processing.** We investigate multi-query processing opportunities, another critical feature required in stream processing. Sharing access of common basket, splitting, merging and dynamically reorganizing the factories content are some cards we use on the performance hunting game.

(5) **A Query Language for DataCell.** We propose a language for DataCell that extends SQL. It can be used to access both streaming and database data at the same time.

(6) **Research Path.** We discuss in detail the new research area that opens with the notion of DataCell and what are the future challenges towards systems that can handle streams of multiple Terabytes on a daily basis.

1.4.5 Published Papers

The content of this thesis is built based on a number of publications in major international conferences in the area of database management systems, of computer science.

(1) Martin Kersten, Erietta Liarou and Romulo Goncalves. A Query Language for a Data Refinery Cell. In Proceedings of the 2nd International Workshop on Event Driven Architecture and Event Processing Systems, (**EDA-PS**), in conjunction with VLDB'07, Vienna, Austria, September 2007

(2) Erietta Liarou, Romulo Goncalves and Stratos Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In Proceedings of the 12th International Conference on Extending Database Technology (**EDBT**), Saint-Petersburg, Russia, March 2009

(3) Erietta Liarou and Martin Kersten. DataCell: Building a Data Stream Engine on top of a Relational Database Kernel. In Proceedings of the 35th International Conference on Very Large Data Bases, **VLDB PhD Workshop**, Lyon, France, August 2009

(4) Erietta Liarou, Stratos Idreos, Stefan Manegold and Martin Kersten. Enhanced Stream Processing in a DBMS Kernel. Submitted for publication at the moment of printing this thesis.

(5) Martin Kersten, Stefan Manegold, Stratos Idreos and Erietta Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. In Proceedings of the Very Large Databases Endowment (**PVLDB**) and in the 37th VLDB Conference, Seattle, WA, August 2011. *Challenges and Visions best paper award.*

(6) Erietta Liarou, Stratos Idreos, Stefan Manegold, Martin Kersten. MonetDB/DataCell: Online Analytics in a Streaming Column-Store. In Proceedings of the 38th International Conference on Very Large Data Bases (**PVLDB**), Istanbul, Turkey, August 2012.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we extensively discuss related work. First, we search the roots of data stream management systems in the heart of database applications and technology. We dedicate enough space to describe and compare our work with other successful pure data stream management systems, from the academic and commercial world. Finally, we provide the mandatory background of our development platform, the MonetDB system.

In Chapter 3 we describe the basic DataCell architecture. First, we explain how we represent continuous queries and data streams in DataCell, the two main stream concepts that until now were unknown in conventional databases. Then we describe the full architecture and the newly introduced components that give full stream functionality to our system.

In Chapter 4 we present the DataCell language interface. We propose a semi-procedural language as a small extension of SQL, that can be used to access both streaming and database data at the same time. The language concepts introduced are compared against building blocks found in "pure" stream management systems. They can all be expressed in a concise way and demonstrate the power of starting the design from a full-fledged SQL implementation.

Chapter 5 presents how we handle in DataCell one of the most important specialized stream processing requirements, i.e., incremental window processing. Even with the conventional underlying infrastructure that MonetDB offers to DataCell, we manage to compete against a specialized stream engine, elevating incremental processing at the query plan level, instead of building specialized stream operators.

Chapter 6 concludes the thesis and discusses a number of future interesting open topics and possible research directions towards a complete data management architecture that integrates database and stream functionalities in the same kernel. DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective and by taking into account the needs of modern data management for scalable stream processing combined with traditional query processing. Topics we discuss in

this chapter include multi-query processing, adaptive query processing, query relaxation, distributed processing, DataCell in different architectures, etc.

Chapter 2

Background and Related Work

Kranzberg's second law states that "invention is the mother of necessity". Though history proves that great technological innovations were given birth at certain periods to fulfill stressed human needs, the technological evolution of the recent years in many scientific areas, creates *new needs* all over again. Scientific evolution on various research areas brought *data overloading* on many aspects of our lives. Modern applications coming from various fields, e.g., finance, telecommunications, networking, sensor and web applications, require fast data analysis over data that are continuously updated.

In this new kind of applications, called data stream applications, we first of all need mechanisms to support *long-standing/continuous* queries over data that is continuously, and at high rate, updated by the environment. To achieve good processing performance, i.e., handling input data within strict time bounds, a system should provide *incremental* processing where query results are frequently and instantly updated as new data arrives. Systems should scale to handle numerous co-existing queries at a time and exploit potential similarities between the large number of standing queries. Furthermore, environment and workload changes may call for adaptive processing strategies to achieve the best query response time. Even if conventional DBMSs are powerful data management systems, the hooks for building a continuous streaming application are not commonly available in such systems.

DataCell balances *at the edge* between the database management systems

world and the data stream systems world. Recognizing all the nice features of a modern database system, we decided to reconsider the effort to implant streaming capabilities within it. In the DataCell project we exploit, reuse, redirect and extend the useful parts of the existing database technology, to support a more complete query processing scenario, where the need of *active* and *passive* processing co-exist.

In this chapter, we discuss relevant background knowledge and related work. The reader can roughly go through the major research efforts of the past twenty years that aimed to define and cope with the active query processing scenario. We point out the main contributions of previous research works that originally introduced the concept of continuous query processing and we compare and place our contributions in the proper context. The interested reader can further explore the rich background research area given the hints this chapter provides.

Particularly in this chapter, we recap the first attempts to define the new needs of data streams and continuous query processing and the early works that were oriented in this direction. The majority of the classical (as we know it today) pure data stream processing systems started developing two decades after the establishment of the database technology. We dedicate ample space to discuss the most important and characteristic stream processing systems from the academic and commercial world, emphasizing on their architecture and comparing their main characteristics with our work. In addition, we touch upon some of the most interesting and important issues of stream processing, including discussion on specialized stream operators, incremental processing and multi-query optimization techniques. We discuss recent work that shares a vision similar to the one of DataCell, namely works that ideally want to tightly integrate on-line and passive data processing. At the end of this chapter, we provide the necessary background for the DataCell philosophy and implementation, describing the backbone of our architecture, the MonetDB database system.

2.1 First Steps towards Real-time Processing

The continuously evolving database technology successfully undertakes a major part of the information technology (IT) duties during the last few decades. Database systems traditionally have been focusing on organizing and storing structured data providing consistent and accurate query processing. These characteristics helped to expand and establish their omnipresence in most of the data management domains. However, the technological innovations naturally bring new application requirements that eventually impose further functional

requirements on database systems.

An example constitutes the real-time control applications that started emerging together with the establishment of the conventional database technology. In this kind of application scenarios, the user desires the data management system to actively control, monitor, and manage his data whenever a change is performed. He expects that the system remains alert and automatically proceeds to the appropriate operations, that he has already defined, when a specific condition becomes satisfied. General database integrity constraint enforcement and business rules motivated the requirement for this new processing model. However, conventional database management systems are built to act *passively*. They offer the appropriate mechanism, to the users and the application programs, to create, modify and retrieve the stored data only after an explicit request. The effort to transform the *passive*, query-driven database system into an *active* one, was the first notable attempt to address the requirements of the monitoring applications, e.g., (Schreier et al., 1991; Sellis et al., 1989; Dayal et al., 1988), etc.

Already in the early 1970s, the Data Base Task Group (DBTG) demonstrated remarkable work in the development of database technology, by proposing the CODASYL (Olle, 1978) data model. CODASYL is the network model for databases, developed to handle many of the problems associated with flat-file systems. The CODASYL data manipulation language (CODASYL Data Description Language Committee, 1973) is one of the first to address the monitoring requirements, adding a *reactive* feature that was not included in the conventional database philosophy up to that time. It provides the appropriate mechanism to automatically invoke the corresponding predefined stored procedures when a specific situation arises. The *ON clause* below encapsulates this functionality:

```
ON <command list> CALL <database procedure>
```

The *database procedure*, could be any arbitrary stored procedure, written in the programming language COBOL. It is called and executed immediately after the execution of the *command list* statement.

Query-By-Example (QBE) (Zloof, 1975; Zloof, 1977), a database query language for relational database systems, is another popular work developed by IBM in the mid of 1970s that provides a trigger facility for integrity constraints checking. QBE allows users to define conditions associated with data modification operations, such as insert, delete and update operations on tables or tuples. If the condition is valid, the operation will commit, otherwise if it is false, the

effects of the operation are undone. In addition, the *time* trigger conditions are evaluated at a specified time point or at specified time frequency (Zloof, 1981).

2.1.1 Triggers

One of the first data control mechanism, i.e., *triggers*, had already been encapsulated early in the relational DBMSs. A *trigger subsystem* was proposed in the mid of 1970s for the pioneer System R relational database research project (Eswaran and Chamberlin, 1975; Eswaran, 1976), that influenced the follow-up database research and technology. The SQL standard committee made a major effort to support triggers and constraints (ISO-ANSI, 1990). Almost all the (commercial) database systems, such as, Oracle, Microsoft SQL Server and DB2 include trigger mechanisms.

A trigger is a user defined stored procedure attached to a single database table or view that is called implicitly and automatically executes when the underlying data is modified in a specific way, i.e., when an INSERT, UPDATE, or DELETE statement is issued against the associated table. The user should also specify whether the trigger must occur BEFORE or AFTER the triggering event or transaction bounds. The DBMS actively monitors the arrival of the desired information and applies it to the database state.

The trigger mechanism was introduced to express and implement complex business rules, which could not be expressed using integrity constraints directly. Initially it was considered as a promising technology to address the requirements of new monitoring applications. However, it quickly proved inadequate to support more complex scenarios; for instance, most DBMSs in their early versions allowed only one trigger for each INSERT, UPDATE, or DELETE data modification event for each table, while triggers over views were not allowed at all. Triggers most likely was limited to one level, where the trigger actions do not cause other triggers to be fired (even today, the modern DBMSs can support only a specific depth of nested triggers, e.g., Oracle and Microsoft SQL Server support nesting depth of 32 triggers, while Sybase supports nesting depth of 16 triggers). Also, the existing systems of that period considered to be weak on preventing errors coming from mutable tables. Scaling to millions or just thousands of trigger conditions in a database, it becomes inefficient to poll the database periodically and check if any of the conditions are satisfied, e.g., (Abiteboul et al., 2005).

Taking all these factors into account, the plan to fully express the demanding monitoring applications through immature triggers was soon abandoned and researchers kept looking for new methods to support richer expressiveness and

improved scalability.

2.1.2 Active Databases

The database research in the mid of 1980s started seriously looking at extending the database technology with powerful rule-processing capabilities, leading to the emergence of a new type of database systems, called active database systems (ADBMSs).

ADBMSs were mainly centered around the concept of the trigger mechanism, and seemed very promising to face the new challenges that the monitoring applications introduced, e.g., (Schreier et al., 1991; Sellis et al., 1989; Dayal et al., 1988), etc. They were considered to be much more powerful than the conventional DBMSs, since they could perform all the standard functionalities that the passive databases provide, in addition to their encapsulated event-driven architecture, that allows users and application programs to specify the desired active behavior.

Active rules, also known as Event-Condition-Action (ECA-rules), traditionally consist of the three following parts:

- ***Event:*** specifies the signal that causes the rule to be triggered.
- ***Condition:*** is checked when the rule is triggered. If it is satisfied, it causes the rule's execution.
- ***Action:*** specifies which further actions (updates) should be taken over the data, and is executed when the rule is triggered and its condition is true.

The triptych "***when*** *event*, ***if*** *condition*, ***then*** *action*" describes in an oversimplified way the active databases' processing model. In active relational databases, events are modeled as changes of the state of the database, i.e., insert, delete and update operations can trigger a reaction. In object-oriented systems, we can define more general events, such as user-defined or temporal events (Bancilhon et al., 1988). The database users can define multiple active rules, that once the system accepts them, it should continuously monitor the relevant events.

In general, the goal of active databases was to avoid unnecessary and resource intensive polling in monitoring applications. Detailed surveys and books catalog in detail the major efforts of active database research, e.g., (Widom and Ceri, 1996; Paton and Díaz, 1999). In the next section we discuss an overview of a characteristic research project, Alert system (Schreier et al., 1991).

Alert

Many research projects, e.g., HiPAC (Dayal et al., 1988), Ariel (Hanson, 1996) and POSTGRES rule system (Stonebraker et al., 1988; Stonebraker et al., 1989), demonstrated that the active database technology was convenient for enforcing business rules and general integrity constraints, which are going beyond key or referential integrity constraints. One of the most notable research results is the outcome of IBM's effort to transform the relational passive Starbust database system, to an active DBMS, called Alert (Schreier et al., 1991).

Alert users are able to define *active tables*, a kind of append-only tables, in which the tuples are never updated in-place and new tuples are added at the end. *Active queries* are queries that range over active tables and their fundamental difference from passive queries is coming from cursor's behavior. Tuples can be added to an active table even after the cursor for an active query is opened and they contribute to answering the query once they are inserted. Thus, the active queries are defined over past, present, and future data, whereas the domain of the passive queries is limited to past and present data. Active queries may be associated to one or more active tables and on abstract user-defined objects, a kind of views. Furthermore, users can express more complex query scenarios by nesting and joining multiple active queries. These features make the Alert architecture much more powerful than the trigger technology at that time was encapsulated in the passive DBMSs. A nice property of the Alert system, is that its rule language achieves full expressiveness with a minimal extension of SQL. In this way, it reuses almost all of the existing semantic checking, optimization, and execution implementations of the passive DBMS that it extends. The *from* clause represents the triggering event, caused by an append to an active table, the *where* clause specifies the condition, and the *select* clause the action that should be taken.

2.1.3 DataCell vs Active Databases and Triggers

DataCell shares similar goals and concepts with triggers and active database systems. All try to extend and re-use the existing powerful conventional database technology by embedding a reactive behavior. In particular active tables and queries share commonalities with DataCell's fundamental units, i.e., baskets and factories (to be further explained in Chapter 3). However, the DataCell model aim to be much more generic by allowing continuous queries to share (i.e., access and modify) multiple baskets (as will be shown in Chapter 3), take their input from other queries and so on, creating a network of queries inside

the kernel where a stream of data and intermediate results flows through the various queries.

DataCell adds support for specialized stream functionalities, i.e., incremental processing. Such functionality is *crucial*, especially for modern high data volume stream applications. The lack of efficient incremental processing in most active databases and databases with triggers, severely affected query latency, and was actually one of the reasons to convince architects to move from the database model to the pure data stream processing model (Abiteboul et al., 2005).

Furthermore, even though active databases address and formulate the requirement for reactive behavior and continuous monitoring, they did not after all provide a *scalable* enough architecture to deal with frequent data updates, as the pure data stream applications later demanded. DataCell is designed and built on top of the extensible MonetDB kernel; the simple and clean stream-oriented design of our architecture helps us inherit and maintain the original DBMS scalability while at the same time combining it with conventional database features. The end result is a stream system that scales well and it can do both continuous queries and one-time queries (Liarou et al., 2009).

MonetDB exploits several modern database architecture trends in its design and DataCell exploits and enhances these features for efficient stream processing. MonetDB is a column-store system that relies on operator-at-a-time bulk processing and materialization of all (narrow) intermediate result columns. This is a convenient and crucial feature to support DataCell's incremental processing requirement. On the contrary, relational active databases were built by extending traditional row store databases. This means that they used a tuple-at-a-time volcano-style pipelining execution model, which at first glance seemed inefficient in providing intermediate result materialization for each query operator.

Furthermore, the internal DataCell scheduler, that handles and controls multiple co-existing (active) queries, dealing also with concurrency issues, is an advanced model that scales much better than the plain trigger mechanism in DBMSs. In the past, it was already observed that triggers do not scale in terms of the number of triggers that an active database can support, leaving as an alternative to implement scalable triggers outside the DBMS. The DataCell scheduler on the other hand is an integral part of the kernel and thus can better co-ordinate and exploit scheduling opportunities and at a lower cost.

2.2 Real-time Databases

Real-time database systems (RTDBSs), as their name implies, also address the requirement for real-time query processing, e.g., (Kao and Garcia-Molina, 1993; Abbott and Garcia-Molina, 1989; Abbott and Garcia-Molina, 1992; Haritsa et al., 1990). RTDBSs can be viewed as a fusion between real-time systems and DBMSs; they extend traditional database technology, adding time constraints and deadlines to transactions. Apart from such features, RTDBSs also introduce and deal with transaction time constraints and temporal validity of data.

In an RTDBS the user specifies when a transaction could start and more importantly when it should finish. Thus, we should process time-sensitive queries and temporally valid data, dealing with priority query scheduling and concurrency control issues. In such an environment, it is difficult to guarantee all time constraints. Thus, the scheduling policy tries to minimize the number of violated time constraints. In real-time databases, it is very important to consider and specify what the system should do when transaction deadlines are not met. The transaction scheduler should allocate available system resources, e.g., CPU cycles, in order to try to meet the specified transaction constraints. However, in many cases the knowledge of resource requirements may not be available up-front and dynamic changes on the workload may occur. In this case, the system should prevent the forthcoming threat of missing multiple transaction deadlines, and should proceed with adaptive decisions and overloading techniques. Different policies then are applied, e.g., rejection of new transactions, early termination of already running ones, etc.

Data stream management systems share similar concerns and goals with RTDBSs. In a typical data stream application, we should evaluate the waiting continuous queries, as soon as possible, trying to minimize the query latency. Scheduling proposals for real-time databases, that are based either on static criteria, e.g., priority-driven, or on dynamic criteria, can also be applied in stream processing policies. Real-time transactions differentiate from continuous queries in data stream systems, to the degree that the latter only allow read-only operations over data streams, while a real-time transaction may involve both read and write operations. This functionality complicates the processing policy once concurrent transactions co-exist. In real-time databases, transactions are usually sporadic while in data streams systems we expect that the continuous queries may stay for a long time in the system. In case we are dealing with *hard* real-time transactions, we may end up aborting entire transaction units, when we come through overloading conditions, while stream applications setting *firm* deadlines could allow us to proceed with data volume minimizing (and thus

resorting to approximate answers).

As modern data stream systems developed over the years, they evolved to specialized stream engines with features missing from traditional real time databases. Such an example is the feature of incremental processing, i.e., window based queries. Such queries allow a system to keep answering queries without blocking the query processing for an "infinite" amount of time. For example, this is useful for blocking operators, or simply for long running queries over large amounts of data. A whole research area was developed then in order to study how to define the proper semantics over such window queries and how to efficiently answer such queries at run time, with multiple concurrent continuous queries, etc. We also explore incremental query processing in the context of window queries, in the DataCell architecture at Chapter 5.

2.3 Publish-Subscribe Systems

Publish/subscribe (in short *pub/sub*) systems are also addressing the monitoring requirements of modern applications and to some extent are related to the area of data stream processing systems. They are mainly applied on a distributed setting and allow simple data and query models.

In pub/sub systems, subscribers register their interest in an event or pattern of events, while publishers, publish available information without addressing it to specific recipients. Typically, a very large number of autonomous computing nodes pool together their resources and rely on each other for data and services. The coordinator messaging infrastructure is responsible to propagate the appropriate messages and notifications to all interested waiting subscribers, once a related resource becomes available. The information to be shared are stored at the publisher's side, and after being discovered by an interested party, they are downloaded using a protocol similar to HTTP. This asynchronous and loosely coupled messaging scheme is a far more scalable architecture than point-to-point alternatives.

Publish/subscribe systems share the same goal: to scale in terms of subscription management, and to assure efficient request-event matching. But beyond this basic goal, there are important differences among the various proposed systems regarding the metadata kept at each network node, the topology of the network, the placement of the shared files, the routing algorithms for queries and replies, the degree of privacy offered to its users, etc.

Different architectures and pub/sub processing models have been proposed; for instance, there are subject-based or content-based systems, following a push-

based, pull-based, or both models, and being implemented in a client-server or peer-to-peer (P2P) architecture. Prominent examples of publish/subscribe applications constitute peer-to-peer databases (Huebsch et al., 2003; Gedik and Liu, 2003; Loo et al., 2004; Fausto et al., 2002), e-learning systems like EDUTELLA (Nejdl et al., 2002) and ELENA (Simon et al., 2003), semantic blogging systems like (Karger and Quan, 2005) and RSS feeds, and parallelized systems like the SETI@home (SETI@home, 1999), the Folding@home (Folding@home, 2000) and the most recent LHC@home (LHC@home, 2004) where a large task is broken into small subtasks and each one is assigned to a different node that offers computing cycles. File-sharing systems such as Napster (Napster, 1999), Gnutella (Gnutella, 2000) and KazaA (KazaA, 2001) have made this model of interaction very popular.

2.4 The New Era of Data Stream Management Systems

In the previous sections, we discussed several designs and trends towards continuous query processing. Active databases, real time databases and trigger mechanisms have all been essential towards developing the streaming technology. None of them, though, was fully prepared for the new requirements of modern streaming query processing applications. Data stream management systems nowadays should handle input data within strict time bounds, and provide instant answers and reactions as new data arrives. Incremental query processing, window-based query processing, scaling to thousands of co-existing queries, etc. are important in a stream system. Even if conventional DBMSs are powerful data management systems, the hooks for building a continuous streaming application are not commonly available in that systems.

Given these differences, and the unique characteristics and needs of continuous query processing, the pioneering *Data Stream Management Systems* (*DSMS*) architects naturally considered that the existing DBMS architectures were inadequate to achieve the desired performance. Another aspect is that the initial stream applications had quite simple requirements in terms of query processing. This made the existing DBMS systems considered overloaded with functionalities. These factors led researchers to design and build new architectures from scratch. Several DSMS solutions have been proposed over the last years giving birth to very interesting ideas and system architectures. In this section, we present some characteristic DSMSs research prototypes and we

compare the main points of these class of systems with our work.

2.4.1 Aurora

Aurora (Carney et al., 2002; Abadi et al., 2003a; Abadi et al., 2003b; Babcock et al., 2004; Balakrishnan et al., 2004) is a data stream management system, that was developed between 2001 to 2004, as a result from the collaboration of three research groups from MIT, Brown University and Brandeis University.

Aurora uses the *boxes and arrows* paradigm, followed in most workflow systems. Each *box* represents a query operator and each *arc* represents a data flow or a queue between the operators. Each query is built out of a set of operators and all submitted queries constitute the Aurora query network. SQuAl is Aurora's query algebra that provides nine stream-oriented primitive operations, i.e., Filter, Map, Union, Aggregate, Join, BSort, Resample, Read, and Update. Out of these operators users construct queries. Each operator may have multiple input streams (i.e., union), and could give its output to multiple boxes (i.e., split). Tuples flow through an acyclic, directed graph of processing operations. At the end, each query converges to a single output stream, presented to the corresponding application. Aurora can also maintain historical storage, to support ad hoc queries.

The query network is divided into a collection of n sub-networks. The decision is taken by the application administrator, who decides where to insert the connection points. Connection points indicate the network modification points and specify the query optimization limits. Thus, new boxes can only be added to or deleted from the connection network points over time. The Aurora optimizer, instead of trying to optimize the whole query graph at once, it optimizes it piece-by-piece. It isolates each sub-network, surrounded by connection points, individually from the rest of the network and optimizes it in a periodic manner.

Figure 2.1 illustrates the high-level system model of Aurora system, as it was originally presented by the authors in their publications (Carney et al., 2002). The *router* connects the system to the outside world. It receives the input data stream from the external data sources, e.g., sensors, and from inside boxes, and if the query processing is completed it forwards the tuples to external waiting sources, otherwise it re-feeds them to the *storage manager* for further processing. The storage manager stores and retrieves the data streams on in-memory buffers between query operators. Also it maintains historical storage, to serve potential ad-hoc queries. A persistence specification indicates exactly for how long the data is kept.

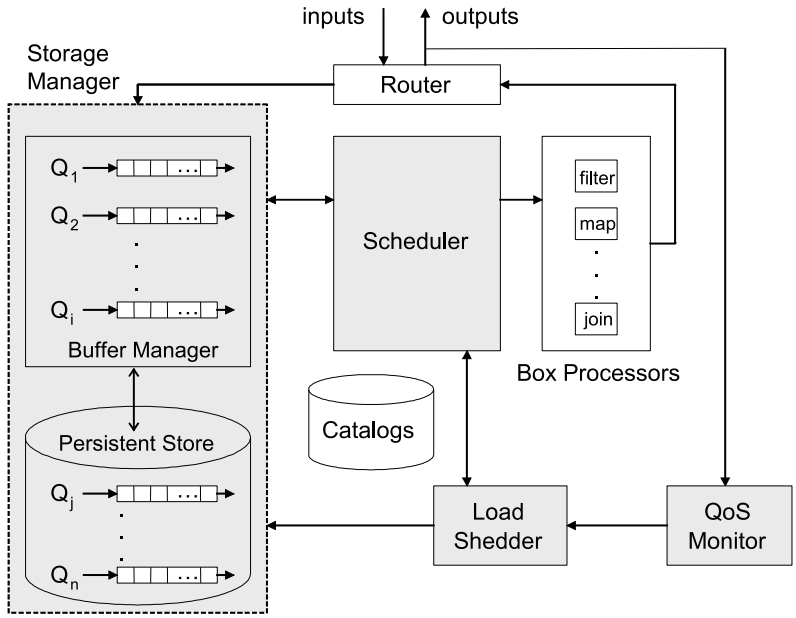

Figure 2.1: Aurora System Architecture (Carney et al., 2002)

The *scheduler* is the core Aurora component (Babcock et al., 2004). It decides when an operator should be executed, feeding it with the appropriate number of queuing tuples. In Aurora, there is one *box processor* per operator type, this part is responsible for executing a particular operator when the scheduler calls it. Then, the box operator forwards the output tuples to the router. The scheduler continuously monitors the state of the operators and the buffers and repeats this procedure periodically.

The designers of Aurora dedicated a big part of their research on addressing methods that guarantee Quality of Service (QoS) requirements when the system becomes overloaded (Tatbul, 2007). They proposed load shedding techniques that attach to the query network a kind of system-level operators that selectively drop tuples. Aurora applies such operators when the rate on incoming streams overwhelm the stream engine, trying to balance between the

expected side-effect on result accuracy meeting QoS application requirements. Later Medusa (Zdonik et al., 2003) and Borealis (Abadi et al., 2005) extended the single-site Aurora architecture to a distributed setting. In 2003, the original research prototype was commercialized into a start-up company named StreamBase Systems (StreamBase Systems, Inc, 2003).

2.4.2 STREAM (STanford stREam datA Management)

STREAM system (Motwani et al., 2003; Arasu et al., 2003) is another data stream processing research prototype that was designed and developed at Stanford university from 2001 to 2006. STREAM provides a declarative query language, called CQL (DBL,), that allows queries which can handle data both from continuous data streams and conventional relations. CQL extends SQL by allowing stream and relational expressions and introducing window operators. In CQL there are three classes of operators, (a) the *stream-to-relation* operators, that produce a relation from a stream (sliding windows), (b) the *relation-to-relation* operators, that produce a relation from one or more other relations, such as in relational algebra and SQL and (c) *relation-to-stream* operators, i.e., *Istream*, *Dstream*, and *Rstream*, that produce a stream from a relation. There are also three classes of sliding window operators, i.e., time-based, tuple-based, and partitioned. However, in practice it does not support sliding windows with a slide bigger than a single tuple.

Also in STREAM, operators read from and write to a single or multiple *queues*. Furthermore, synopses are attached to operators and store their intermediate state. This is useful when a given operator needs to continue its evaluation over an already processed input. For instance, when we need to maintain intermediate results, i.e., the content of a sliding window or the relation produced by a subquery. Synopses are also used to summarize a stream or a relation when approximate query processing is required. Scheduling in STREAM also happens at the operator level as it used to in stream systems; either it is simple scheduling strategy (Motwani et al., 2003) like round-robin or FIFO or the more sophisticated Chain algorithm (Babcock et al., 2003). The scheduling methods in STREAM focus on providing run-time memory minimization. STREAM also includes a monitoring and adaptive query processing infrastructure called StreaMon (Babu and Widom, 2004), which consists of three components, i.e., the *Executor*, that runs query plans and produces results, the *Profiler*, that collects statistics about stream and query plan characteristics, and the *Reoptimizer*, that takes the appropriate actions to always ensure that the query plan and memory usage are optimal for the current input characteristics.

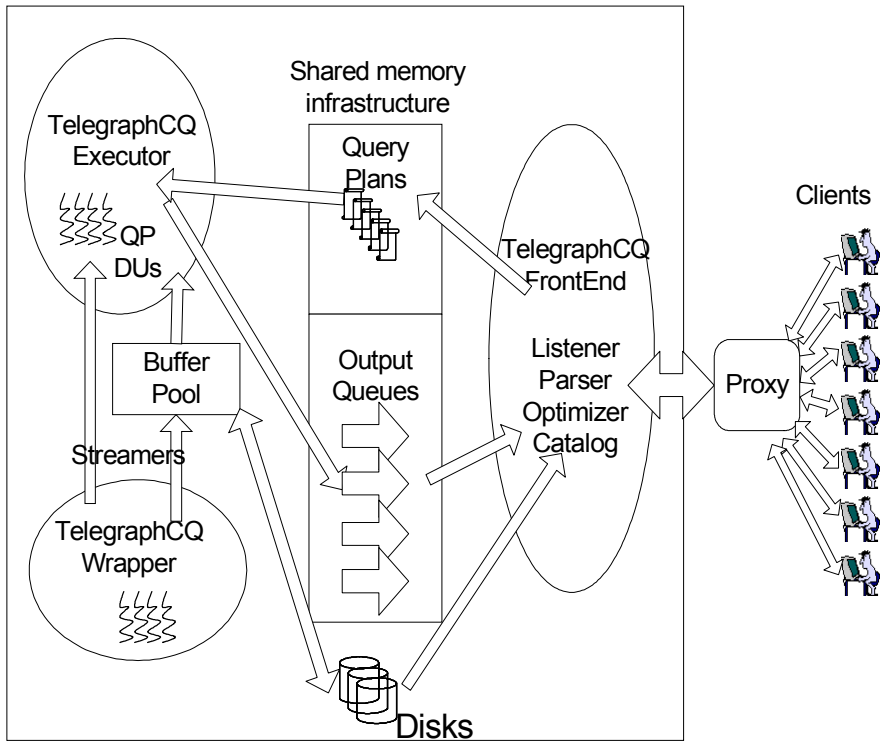

Figure 2.2: TelegraphCQ System Architecture (Chandrasekaran et al., 2003)

Once there is not enough CPU or memory available, the system proceeds with approximate query processing, trying to handle the query load by sacrificing accuracy. It introduces random sampling operators into all query plans, in a way that the relative error is the same for all queries. STREAM deals with memory-limitations also by discarding older tuples from the window joins operators, leaving free space for new data. The goal here is to maximize the size of the resulting subset.

2.4.3 Telegraph-CQ

TelegraphCQ (Chandrasekaran et al., 2003) is a continuous query processing system built at University of California, Berkeley. The main focus is on adaptive and shared continuous query processing over query and data streams. The team in Berkeley, built TelegraphCQ based on previous experience obtained while developing the preliminary prototypes, CACQ (Madden et al., 2002) and PSoup (Chandrasekaran and Franklin, 2002).

PSoup addresses the need for treating data and queries symmetrically. Thus, it allows new queries to see old data and new data to see old queries. This feature is passed to the TelegraphCQ architecture as well. Furthermore, TelegraphCQ successfully addresses and resolves important limitations that were not addressed in previous prototypes, e.g., it deals with memory and resource limitations, trying to guarantee QoS over acceptable levels and focuses on scheduling and resource management of groups of queries. TelegraphCQ constructs query plans with adaptive routing modules, called Eddies (Avnur and Hellerstein, 2000). Thus, it is able to proceed to continuous run-time optimizations, dynamically adapting to the workload. Eddies modules adaptively decide how to route data to appropriate query operators on a tuple-by-tuple basis.

TelegraphCQ shares a similar goal and vision as the one of DataCell. It tries to leverage the infrastructure of a conventional DBMS, by reusing a big part of the open source PostgreSQL code base. With minimal changes at particular components, it tries to use the front-end piece of code that PotegreSQL already offers, the Postmaster, the Listener, the System Catalog, the Query Parser and the PostgreSQL Optimizer. However, the TelegraphCQ developers proceeded to significant changes on the deeper PostgreSQL parts, such as the Executor, the Buffer Manager and the Access Methods, to make them compatible with the unique requirements of stream processing. Figure 2.2 illustrates an overview of the TelegraphCQ architecture as it is originally presented in (Chandrasekaran et al., 2003). The rightmost oval part is the most solid contribution of PostgreSQL to the new system architecture. The processes included in there, are connected using a shared memory infrastructure, and the generated query plans are placed in a query plan queue. From there, the *Executor* picks them up to proceed with the actual processing, trying first to classify the plans into groups for sharing work. The query results are continuously placed in the output queues. The *Wrapper* mechanism allows data to be streamed into the system.

As we already mentioned, TelegraphCQ follows a similar approach to DataCell, by trying to exploit the PostgreSQL infrastructure. However, there are significant differences between PostgreSQL and MonetDB that significantly af-

fect the whole streaming architectures. DataCell reuses the original storage and execution engine of the MonetDB kernel, elevating the streaming behavior at the embedded scheduler module. In contrast, TelegraphCQ needs new storage and access methods. In addition, in DataCell we do not follow a tuple-at-a-time processing method, instead we favor batch execution which brings high performance and scalability. Tuple-at-a-time has a significant functional overhead that severely hinders scalability. On the other hand, bulk processing for streams is a new area which brings performance and additional research questions as to how to properly tune the degree of batch processing throughout query plans. Furthermore, DataCell exploits array based processing as it builds on top of the pure column-store infrastructure of MonetDB. Arrays together with bulk processing are heavily exploited for efficient incremental window-based processing in DataCell. In contrast, TelegraphCQ is built on top of a typical row-store infrastructure.

2.4.4 Other Data Stream Management Systems

The unique requirements of monitoring applications, establish a new research field that demonstrates interesting results on new system architectures, query languages, specialized algorithms and optimizations. So far, we presented three characteristic efforts from the academic world. However the research efforts do not stop there; plenty of other interesting stream systems have been presented to related journals and conferences, and some of them found their way to the commercial world.

A noteworthy result is Gigascope (Cranor et al., 2003), a lightweight stream processing system that was developed in AT&T to serve network applications. It emerged from requirements of the company itself, e.g., traffic analysis, network monitoring and debugging.

NiagaraCQ (Chen et al., 2000) is an XML-based continuous query system that focuses on query optimization to improve scalability. This system tries to exploit query similarities to group queries and potentially save processing cost. The grouping process happens incrementally and once new queries are added to the system, they find their place in the appropriate groups.

Different language semantics are introduces in the Cayuage system, developed in Cornell university. Cayuage is a stateful publish/subscribe system based on a non-deterministic finite state automata (NFA) model.

Big vendors like Microsoft (Ali et al., 2009), IBM (Gedik et al., 2008) and Aleri/Coral8 (Coral8, 2007) have also become active in the data stream area during the last few years, developing high performance complex event processing

systems. Their focus is on pure stream processing, providing additional external access to historical data. Furthermore, they have moved their architectures in distributed settings to cope with the increasing data requirements.

2.4.5 DataCell vs Traditional Data Stream Architectures

In this chapter, we presented some well known data stream management systems. Each one contributed in a unique way to the broad research area of data streams. However all of them follow the same philosophy; they are built from scratch, dismissing the conventional database technology. DataCell fundamentally differs from existing stream technology, by building the complete stream functionalities on top of the storage and execution engine of a modern DBMS kernel. In this way, it opens an interesting path towards exploiting and merging technologies from both worlds.

The design of DataCell allows to exploit batch processing when the application allows it. Tuple-at-a-time processing, used in most stream systems, incurs a significant overhead while batch processing provides the flexibility for better query scheduling, and exploitation of the system resources. This point has also been nicely exploited in (Lim et al.,) but in the context of the DataCell, building on top of a modern DBMS, it brings much more power as it can be combined with algorithms and techniques of relational databases.

In addition, DataCell exploits the batch processing logic during scheduling. It tries to keep *together* as many query operators as possible. In this way, it wraps in a *single factory* all or a subset of the operators that belong to the same query plan for a given continuous query. In any case, it avoids scheduling one operator at a time and tries to schedule groups of operators that can be executed together. A factory may contain (parts of) query plans from more than one queries. In this way, we increase scalability by minimizing the scheduling overhead, i.e., by reducing the number of distinct units that the scheduler should monitor and orchestrate at each moment and by reducing all the side-effects that this process entails, e.g., access to storage units, switching function calls. Aurora also recognizes the overhead that its first single-operator scheduling approach causes, and introduces the notion of superboxes (Babcock et al., 2004). There, a sequence of boxes is scheduled and executed as an atomic group. However, it allows only the construction of superboxes that conclude to an output box, without giving the flexibility to group together intermediate groups of operators, as DataCell does.

Furthermore, DataCell tries to fully exploit the state-of-the-art modern database software stack that MonetDB offers. This fact brings a number of

fundamental differences between DataCell and the majority of pure data stream systems. For example, one such difference is that DataCell does not use buffers to temporary hold the flowing stream tuples, and consequently does not require the existence and maintenance of a separate storage manager component. On the contrary, it uses *baskets*, a kind of temporary main-memory tables which are more powerful than the simple buffer structure and more lightweight than the conventional database tables. In Section 3, we present the key components of our architecture and discuss in further detail what are the differences between basket and MonetDB tables.

The DataCell architecture interweaves basket and tables in the most natural way, since it develops both technologies in the same kernel. In this way, we can support queries that require data access from both streams and tables, and generate query plans having all this information in our plate already at the generation and optimization phase. Many other pure stream systems address the modern application requirements for access to both storage units. However, they reach their goal by either connecting a specialized DBMS to a stream engine, or by creating simplistic storage unites (compared to a full-blown database system) and execution mechanism that mimic the database work.

In DataCell, we manage to deal with crucial stream processing challenges, like the incremental window-based processing, by re-using most of the given database infrastructure. By introducing only small language extensions in SQL, we can re-use the SQL front-end and slightly extend the parser that MonetDB already offers. In order to maintain and reuse the generic storage and execution model of the DBMS, we elevate the stream processing at the query plan level. Proper optimizer rules, scheduling and intermediate result caching and reuse, allow us to modify the DBMS query plans for efficient incremental processing. In addition, we avoid to re-design and implement from scratch specialized stream operators as the pure stream systems do. Instead, by introducing the appropriate scheduling mechanisms we manage to achieve full stream functionalities using the efficient scalable operators of MonetDB. In this way, shared processing in our case does not happen at the operator level but also at the factory level, trying to maintain and reuse (batches of) intermediate results.

In this thesis we took a completely different route by designing a stream engine on top of an existing relational database kernel. Such an approach was considered a failure in the past due to the fact that databases where to slow in handling streams. Here, we show that DataCell achieves high performance, scales and naturally combines continuous querying for fast reaction to incoming data with traditional querying for access to existing data.

2.5 A new Stream Processing Paradigm

In the previous section, we discussed the main philosophy of the specialized stream engines that were developed to efficiently handle continuous query processing in bursty data arrival periods. However, the technological evolutions keep challenging the existing architectures with new application scenarios. In recent years, a new processing paradigm is born (Liarou et al., 2009; Qiming and Meichun, 2010; Franklin et al., 2009) where incoming data needs to quickly be analyzed and possibly be combined with existing data to discover trends and patterns. Subsequently, the new data enters the data warehouse and is stored for further analysis if necessary. This new paradigm requires scalable query processing that combines continuous and conventional processing.

The Large Synoptic Survey Telescope (LSST) (LSST, 2010) is a grandiose paradigm. In 2018 the astronomers will be able to start scanning the sky from a mountain-top in Chile, recording 30 Terabytes of data every night which incrementally will lead a 150 Petabyte database (over the operation period of ten years). LSST will be capturing changes to the observable universe evaluating huge statistical calculations over the entire database. Another characteristic data-driven example is the Large Hadron Collider (LHC) (LHC, 2010), a particle accelerator that is expected to revolutionize our understanding for the universe, generating 60 Terabytes of data every day (4GB/sec). The same model stands for modern data warehouses which enrich their data on a daily basis creating a strong need for quick reaction and combination of scalable stream and traditional processing (Winter and Kostamaa, 2010). However, neither pure database technology nor pure stream technology are designed for this purpose.

Truviso Continuous Analytics system (Franklin et al., 2009), a commercial product of Truviso, is another recent example that follows the same approach as DataCell. Part of the team that was working on the TelegraphCQ project, proceeded to the commercialized version of the original prototype. They extend the open source PostgreSQL database (PostgreSQL, 2012) to enable continuous analysis of streaming data, tackling the problem of low latency query evaluation over massive data volumes. TruCQ integrates streaming and traditional relational query processing in such a way that ends-up to a *stream-relational* database architecture. It is able to run SQL queries continuously and incrementally over data while they are still coming and before they are stored in *active database tables* (if they). TruCQ's query processing outperforms traditional *store-first-query-later* database technologies as the query evaluation has already started when the first tuples arrive. It allows evaluation of one-time and continuous queries as well as combinations of both query types.

Another recent work, coming from the HP Labs (Qiming and Meichun, 2010), confirms the strong research attraction for this trend. They define an extended SQL query model that unifies queries over both static relations and dynamic streaming data, by developing techniques to generalize the query engine. They extending the PostgreSQL database kernel (PostgreSQL, 2012), building an engine that can process persistent and streaming data in a single design. First, they convert the stream into a sequence of *chunks* and then continuously call the query over each sequential chunk. The query instance never shuts down between the chunks, in such a way that a cycle-based transaction model is formed.

The main difference of DataCell over the above two related efforts lies in the underlying architecture. DataCell builds over a column-store kernel using a columnar algebra instead of a relational one, bulk processing instead of volcano and vectorized query processing as opposed to tuple-based. Here we exploited all these architectural differences to provide efficient incremental processing by adapting the column-store query plans.

2.6 Data Stream Query Languages

The unique monitoring application requirements, brought new data management architectures and consequently the need for new querying paradigms. In the literature we distinguish three classes for query languages that define the proper data streaming semantics.

Declarative

Many stream systems define and support languages that maintain the declarative and rich expressive power of SQL. A characteristic example is CQL (for Continuous Query Language) (DBL,), which is introduced and implemented in the STREAM prototype (Motwani et al., 2003; Arasu et al., 2003). Apart from streams, CQL also includes relations. Thus, we can write queries from each category and queries that combine both data types as well. In CQL, we have three types of operators: the relation-to-relation operators, that SQL already offers, the stream-to-relation operators, that reflect the sliding windows, and the relation-to-streams operators, that produce a stream from a relation. There, we also have three classes of sliding window operators in CQL: time-based, tuple-based, and partitioned windows. We can denote a time-based sliding window of size T on a stream S, with the expression `[Range T]`. A tuple-based sliding

window of size N on a stream S is specified by following the reference to S in the query with `[Rows N]`.

GSQL is another SQL-like query language, developed for Gigascope to express queries for network monitoring application scenarios. GSQL is a stream-only language, where all inputs to a GSQL operator should be streams and the outputs are streams as well. However, relations can be created and manipulated using user-defined functions. Each stream should have an ordering attribute, e.g., timestamp. Only a subset of the operators found in SQL are supported by Gigascope, i.e., selections, aggregations and joins of two streams. In addition to these operators, GSQL includes a stream *merge* operator that works as an order-preserving union of ordered streams. In GSQL, only landmark windows are supported directly, but sliding windows may be simulated via user-defined functions.

StreaQuel is the declarative query language proposed and used in TelegraphCQ prototype. It supports continuous queries over a combination of tables and data streams. By using a *for-loop* construct with a variable t that moves over the timeline as the for-loop iterates, we can express the sequence of windows over which the user desires the answers to the query. Inside the loop we include a *WindowIs* statement that specifies the type and size of the window over each stream. This way, snapshot, landmark and sliding window queries can be easily expressed.

Procedural

A different approach to declarative SQL-like query languages, is a procedural one. For instance in Aurora, the developers proposed SQuAl (for Stream Query Algebra), a boxes-and-arrows query language. There, the user through a graphical interface draws a query plan, placing boxes (i.e., operators) and arrows (i.e., data streams) in the appropriate order, specifying how the data should flow through the system. SQuAl accepts streams as inputs and returns streams as output. However, it gives the option to the user to include historical data to query processing through explicitly defined connection points.

2.7 The MonetDB System

In this section, we provide the necessary background for the rest of our presentation, briefly describing the backbone of the DataCell architecture, the MonetDB database system. MonetDB (MonetDB, 2012) is an open-source column-

oriented DBMS, developed at the database group of CWI (Centrum Wiskunde & Informatica) in Amsterdam, the Netherlands, over the past two decades.

Row-store vs. Column-store architecture

MonetDB is a full fledged column-store engine; thus it stores and process data one column at a time as opposed to one tuple at a time that traditional row-stores do.

Let us first clarify what are the main differences between the two directions. A row-oriented database system stores all of the values per row from a given table together. The processing model in a row-store is typically based on the volcano model, i.e., the query plan consumes one tuple at a time. Each tuple goes all the way through every operator in the plan, before we move on to the next tuple.

On the contrary, column-oriented DBMSs are inspired by the Decomposition Storage Model (DSM) (Copeland and Khoshafian, 1985), storing data one column at a time. In this way, the system can benefit a lot in terms of I/O for queries that require to access only part of a table's attributes and not the whole table. Assume a table representing students in a university's database. This table will typically consist of a number of attributes, i.e., first name, last name, date of birth, student ID, address, department, etc. Now let's say that the secretary of the university wants to analyze the data by posing the following queries: find the average grades of the students per department, find the number of students that have exceeded the normal studying period, find the average age of students per department, etc. In this kind of queries we access only a part of the table "students". In order to answer such queries in a row store architecture we would need to load the whole table from disk to memory. On the other hand, in a column-store architecture we only load the data (columns) each query requests.

In general, row-store architectures are most appropriate when the database is mostly used for online transaction processing (OLTP). There, we expect a large number of short on-line transactions. On the other side, column-store architectures are most appropriate for applications that handle analytical queries for online analytical processing (OLAP). There, we expect relatively low volume of transactions while queries are often very complex and involve aggregations but usually focus on a subset of a table's attribute.

The MonetDB Storage Model

In MonetDB, every n-ary relational table is represented as a collection of *Binary Association Tables* called *BAT*s (Boncz et al., 1998). A BAT represents a mapping from an `oid`-key to a single attribute `attr`. Its tuples are stored physically adjacent to speed up its traversal, i.e., there are no holes in the data structure. For a relation R of k attributes, there exist k BATs, each BAT storing the respective attribute as (`key`,`attr`) pairs. The system-generated `key` identifies the relational tuple that attribute value `attr` belongs to, i.e., all attribute values of a single tuple are assigned the same `key`. For base tables, they form a dense ascending sequence enabling highly efficient positional lookups. Thus, for base BATs, the key column is a virtual non-materialized column. For each relational tuple t of R, all attributes of t are stored in the *same* position in their respective column representations. The position is determined by the insertion order of the tuples. This tuple-order *alignment* across all base columns allows the column-oriented system to perform tuple reconstructions efficiently in the presence of tuple order-preserving operators. Basically, the task boils down to a simple merge-like sequential scan over two BATs, resulting in low data access costs through all levels of modern hierarchical memory systems.

The MonetDB Execution Model

In MonetDB, SQL queries are translated by the compiler and the optimizer into a query execution plan that consists of a sequence of relational algebra operators. Each relational operator corresponds to one or more MAL instructions, while each MAL instruction performs a single action over one or more columns in a bulk processing mode.

MonetDB is a late tuple reconstruction column-store. Thus, when a query is fired, the relevant columns are loaded from disk to memory but are glued together in a tuple N-ary format only prior to producing the final result. Intermediate results are also materialized as temporary BATs in a column format. We can efficiently reuse intermediate results by recycling pieces of (intermediate) data that are useful for multiple queries (Ivanova et al., 2009). Also, in Chapter 5 and (Liarou et al., 2012a) we show that the bulk processing model of MonetDB and the materialized intermediate results are important components in our effort to support incremental stream processing for window-based continuous queries.

Let us now see a concrete example. Assume the following SQL query:

```
SELECT R.c
FROM R
WHERE R.a BETWEEN 5 AND 10
AND R.b BETWEEN 9 AND 20;
```

This query is translated into the following (partial) MAL plan:

```
Ra1 := algebra.select(Ra, 5, 10);
Rb1 := algebra.select(Rb, 9, 20);
Ra2 := algebra.KEYintersect(Ra1, Rb1);
Rc1 := algebra.project(Rc, Ra2);
```

The first operator, `algebra.select(Ra,v1,v2)`, searches the base BAT Ra for attributes with values between $v1$ and $v2$. For each qualifying attribute value, the respective `key` value (position) is included in the result BAT $Ra1$. Since selections happen on base BATs, intermediate results are also ordered in the insertion sequence. In MonetDB, intermediate results of selections are simply the keys of the qualifying tuples, thus the positions of where these tuples are stored among the column representations of the relation. In this way, given a key/position we can fetch/project (positional lookup) different attributes of the same relation from their base BATs very fast. Since both intermediate results and base BATs have the attributes ordered in the insertions sequence, MonetDB can very efficiently project attributes by having *cache-conscious reads*.

As we mentioned above, each MAL instruction is internally executed in a bulk processing way. The implementation at the C code level of the MAL instruction `Ra1 := algebra.select(Ra,v1,v2)` is as follows:

```
for (i = j = 0; i < n; i++)
 if (Ra.tail[i] >= v1)
  if (Ra.tail[i] =< v2)
   Ra1.tail[j++] = i;
```

With tight for-loops in BAT algebra operators, we have the advantage of high instruction locality that minimizes the instruction cache miss problem.

The MAL operator `algebra.KEYintersect(Ra1,Rb1)` is a tuple reconstruction operator that performs the conjunction of the selection results by returning the intersection of keys from $Ra1$ and $Rb1$ columns. Due to the order-preserving

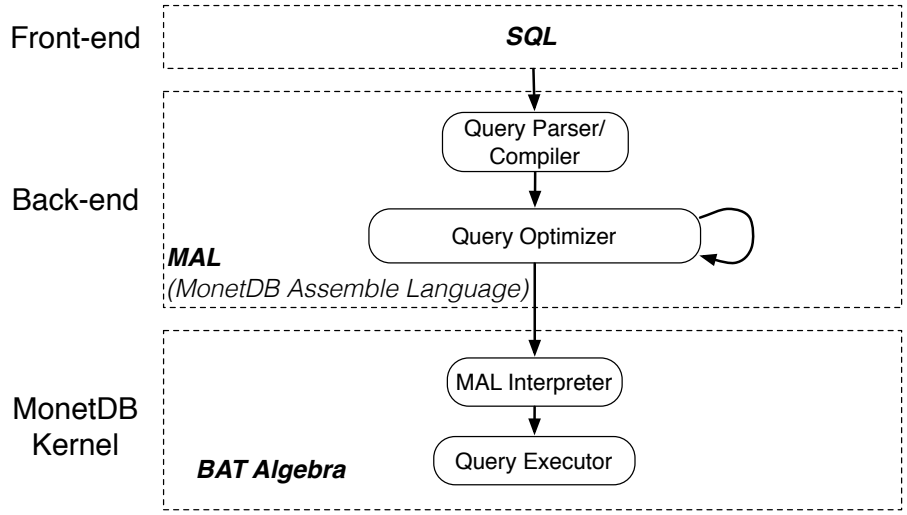

Figure 2.3: MonetDB Architecture

selection, both $Ra1$ and $Rb1$ are ordered on `key`. Thus, both intersection and union can be evaluated using cache-, memory-, and I/O-friendly sequential data access. The results are ordered on `key`, too, ensuring efficient tuple reconstructions.

Finally, the MAL operator `algebra.project(Rc,Ra2)` returns all `key-attr` pairs residing in base BAT Rc at the positions specified by $Ra2$. This is a tuple reconstruction operation. Iterating over $Ra2$, it uses cache-friendly in-order positional lookups into $Ra2$.

The MonetDB Software Stack

The MonetDB query processing scheme consists of three software layers. The top layer is formed by the query language parser that outputs a logical plan expressed in MAL. The code produced by MonetDB/SQL is passed and massaged by a series of optimization steps, denoted as an *optimizer pipeline*. The MAL plans are transformed into more efficient plans enriched with resource management directives. The pipeline to be used is identified by the SQL global variable optimizer, which can be modified using a SQL assignment.

The extensible design of MonetDB opens the traditionally closed and monolithic query optimization and execution engine, providing a modular multi-tier query optimization framework. Optimizer pipelines in MonetDB can be configured and extended to effectively exploit domain-specific data and workload characteristics.

At the bottom of the MonetDB software stack there is the MAL interpreter. It contains the library of highly optimized implementation of the binary relational algebra operators. At the run-time the MonetDB engine takes into account collected statistics of the participant BATs and it is able to choose the best evaluation algorithm (physical operator) for each logical operator. For example, once it comes to the execution of the MAL operator

```
Re1:=algebra.join(Ra1,Rb1);
```

based on the size of $Ra1$ and $Rb1$ columns, the engine may decide to execute the hash join algorithm while in another case (with different data and statistics in the corresponding columns) it may execute the sort merge join algorithm.

In Figure 2.3, we show the MonetDB architecture as a series of abstraction layers. The interested reader can find more details on MonetDB in (MonetDB, 2012). In this thesis, we implement DataCell in the heart of MonetDB. Our implementation represents a set of new optimization rules, operators, algorithms and data structures that cooperate with the existing MonetDB features to give the desired result.

2.8 Summary

In this chapter, we briefly discussed the information technology history, touching on major attempts to define and support monitoring applications. We discussed the first efforts to support real-time processing applications which came through the conventional database technology. Then, we saw how the data explosion and the need for sophisticated near-real time analysis brought the genesis of specialized data streams management systems. Today, we are in an era where the need to tightly combine both database and data stream technologies is bigger than ever. Through this short survey we tried to highlight the major point that makes DataCell a unique and novel research path. Finally, we give the necessary background on the MonetDB system, which is the backbone of the DataCell architecture.

In the following chapters we introduce in detail the DataCell architecture, the DataCell query language and how DataCell handles specialized stream pro-

cessing requirements, i.e., incremental window processing. In the last chapter of this thesis, we summarize the major points of our architecture and discuss open research directions that deserve thorough study and will bring us closer to a scalable integrated system.

Chapter 3

DataCell Architecture*

3.1 Introduction

This chapter introduces the basic DataCell architecture. A system that naturally *integrates* database and stream query processing inside the same query engine. We start with a modern column-store architecture, realized in the MonetDB system, and we design our new system based on this kernel. Our ultimate goal is to fully exploit the generic storage and execution engine of the underlying DBMS as well as its optimizer stack. With a careful design, we can directly reuse all sophisticated algorithms and techniques of traditional DBMSs. A prime benefit is that without having to reinvent solutions and algorithms for problems and cases with a rich database literature we can support complex queries and scalable query processing in a streaming environment.

The main idea is that when stream tuples arrive into the system, they are immediately stored in (appended to) a new kind of tables, called *baskets*. By collecting tuples into baskets, we can evaluate the continuous queries over the baskets as if they were normal one-time queries. Thus, we can reuse many algorithms and optimizations designed for a modern DBMS. Once a tuple has been seen by all relevant queries/operators, it is *dropped* from its basket. The above description is naturally oversimplified as this direction allows the exploration of quite flexible strategies. For example, alternative directions include feeding the same tuple into multiple baskets where multiple queries are waiting, split

*The material in this chapter has been the basis for the EDBT09 paper "Exploiting the Power of Relational Databases for Efficient Stream Processing" (Liarou et al., 2009).

query plans into multiple parts and sharing baskets between similar operators (or groups of operators) of different queries allowing reuse of results and so on. The query processing scheme of DataCell follows the Petri-net model (Peterson, 1977), i.e., each component/process/sub query plan is triggered only if it has input to process while its output is the input for other processes.

3.1.1 Challenges and Contributions

Some questions that immediately arise, when we start thinking of and studying the DataCell approach, are the following:

- How does DataCell guarantee responsiveness?
- How efficient continuous query processing can DataCell provide?
- What is the optimal basket size?
- When do the queries see an incoming tuple?
- Can we handle queries with different priorities?
- Can we support query grouping?
- Is it feasible for all kind of stream applications (e.g., regarding time constraints)?

The above questions are just a glimpse of what one may consider. This chapter does not claim to provide an answer to all these questions, neither does it claim to have designed the perfect solution. Our contribution is the awareness that this research direction is feasible and that it can bring significant advantages. We carefully carve the research space and discuss the opportunities and the challenges that come with this approach.

This chapter presents a complete architecture of DataCell in the context of the currently emerging column-stores. We discuss our design and implementation on top of the open-source column-oriented DBMS, MonetDB. DataCell is realized as an extension to the MonetDB/SQL infrastructure and supports the standard SQL'03 allowing stream applications to support sophisticated query semantics.

Our prototype implementation demonstrates that a full-fledged database engine can support stream processing completely and efficiently. The validity of our approach is illustrated using concepts and challenges from the pure DSMS

arena. A detailed experimental analysis using both micro-benchmarks and the standard Linear Road benchmark demonstrates the feasibility and the efficiency of the approach.

3.1.2 Outline

The remainder of this chapter is organized as follows. In Section 3.2, we present a detailed introduction of the DataCell architecture at large. Section 3.3 discusses the query processing model and pinpoints on the wide open research possibilities. In Section 3.5 we provide an experimental analysis of the proposed DataCell architecture, including micro-benchmarks and using the Linear Road benchmark. Finally, Section 4.4 concludes the chapter.

3.2 The DataCell Architecture

In this section, we discuss the DataCell prototype architecture, which is based on top of MonetDB, positioned between the SQL-to-MAL compiler and the MonetDB kernel. In particular, the SQL runtime has been extended to manage the stream input using the columns provided by the kernel, while a scheduler controls activation of the continuous queries. The SQL compiler is extended with a few orthogonal language constructs to recognize and process continuous queries. We discuss the language extension in Chapter 4.

We step by step build up the architecture and the possible research directions. DataCell consists of the following components: *receptors*, *emitters*, *baskets* and *factories*. The novelty is the introduction of baskets and factories in the relational engine paradigm. Baskets and factories can, for simplicity, initially be thought as tables and continuous queries, respectively.

There is a large research landscape on how baskets and factories can interact within the DataCell kernel to provide efficient stream processing. In the rest of this section, we describe in detail the various components and their basic way of interaction. More advanced interaction models are discussed in Section 3.3.2.

3.2.1 Receptors and Emitters

The periphery of a stream engine is formed by *adapters*, i.e., software components to interact with devices, e.g., RSS feeds and SOAP web-services. The communication protocols range from simple messages to complex XML documents transported using either UDP or TCP/IP. The adapters for the DataCell

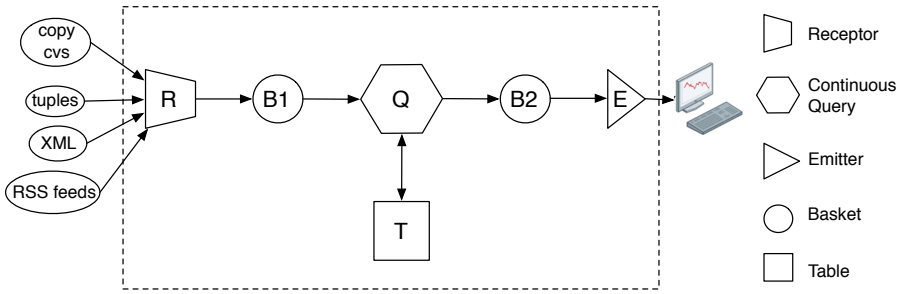

Figure 3.1: The DataCell model

consist of receptors and emitters.

A *receptor* is a separate thread that continuously picks up incoming events from a communication channel. It validates their structure and forwards their content to the DataCell kernel for processing. There can be multiple receptors, each one listening to a different communication channel/stream.

Likewise, an *emitter* is a separate thread that picks up events prepared by the DataCell kernel and delivers them to interested clients, i.e., those that have subscribed to a query result. The emitter automatically removes from the kernel the delivering data. There can be multiple emitters each one responsible for delivering a different result to one or multiple clients.

Both receptors and emitters are connected to a basket, the data structure where they write to and read from the streaming data, as we describe in the next subsection. Figure 3.1 demonstrates a simple interaction model between the DataCell components; a receptor and an emitter can be seen at the edges of the system listening to streams and delivering results, respectively.

3.2.2 Baskets

The basket is the key data structure of DataCell. Its role is to hold a *portion* of a data stream, represented as a temporary main-memory table. Every incoming tuple, received by a receptor, is immediately placed in (appended to) at least one basket and waits to be processed.

Once data is collected into baskets, we can evaluate the relevant continuous queries on top of these baskets. In this way, instead of feeding each individual tuple to the relevant query, we evaluate each query over its input basket(/s)

in one go (e.g., consuming *all* accumulated tuples at once). This processing model resembles the typical DBMS scenario and thus we can exploit existing algorithms and functionality of advanced DBMSs. Later in this section, we discuss in more detail the interaction between queries and baskets.

The commonalities between baskets and relational tables allow us to avoid a complete system redesign from scratch. Therefore, the syntax and semantics of baskets is aligned with the table definition in SQL'03 as much as possible. A prime difference is the retention period of their content and the transaction semantics. A tuple is removed from a basket when it "has been consumed" by all relevant continuous queries and it is not needed anymore. In this way, the baskets initiate the data flow in the stream engine. More advanced and flexible models are discussed in the next section.

The main differences between baskets and relational tables are as follows.

- ***Basket Integrity***
  The integrity enforcement for a basket is different from a relational table. Events that violate the constraints are silently dropped. They are not distinguishable from those that have never arrived in the first place. The integrity constraint acts as a silent filter.

- ***Basket ACID***
  The baskets are like temporary global tables, their content does not survive a crash or session boundary. However, concurrent access to their content is regulated using a locking scheme and the scheduler.

- ***Basket Control***
  The DataCell provides control over the streams through the baskets. A stream becomes blocked when the relevant basket is marked as DISABLED. The state can be changed to ENABLED once the flow is needed again. Selective (dis)enabling of baskets can be used to debug a complex stream application.

- ***Basket Tuple Expiration***
  In a stream application scenario, tuple expiration happens on a more frequent basis than in a typical OLAP scenario. In DataCell, we handle data stream expiration immediately and in a different way than we handle updates and deletions in the underlying columnar architecture of MonetDB. In MonetDB, for each column we maintain three different arrays to represent the original persistent tuples, the updated and inserted tuples, and the deleted tuples. Once a (one-time) query is submitted, we first merge

the tuples from these three arrays to get only the valid tuple values and then continue with the actual query evaluation. When the two secondary arrays grow enough, then the merging happens automatically. Even if the data deletion in an OLAP scenario looks similar to the data stream expiration in a stream application scenario, it is inefficient to follow this processing model since the accumulated number of expired tuples is expected to grow rapidly. Instead, in DataCell we choose to do complete and instant data stream deletion, without maintaining the expired data aside. This change implies a change to the generated query plans as well. We now have only a single array instead of three (because we do deletions in place) and consequently inside the plans there is no need to do any merging of these arrays.

An important opportunity, with baskets as the central concept, is that we purposely step away from the de-facto approach to process events in arrival order only. Unlike other systems there is no a priori order; a basket is simply a (multi-)set of events received from a receptor. We consider arrival order a semantic issue, which may be easy to implement on streams directly, but also raises problems, e.g., with out-of-sequence arrivals (Abadi et al., 2005), regulation of concurrent writes on the same stream, etc. It unnecessarily complicates applications that do not depend on arrival order. On the other hand, baskets in DataCell provide maximum flexibility to perform both in-order and out-of-order processing. They allow the system to select and process arbitrary groups of tuples at a time, without necessarily following their arrival order.

Realizing the DataCell approach on top of a column-oriented architecture, comes with all the benefits of the respective design. e.g., depending on the workload there may be less I/O and memory bandwidth requirements for a column-store. For a stream S of k attributes, we create a basket B that consists of k BATs (columns). Each BAT stores the respective attribute of stream S as (`key`,`attr`) pairs. In this way, the basket representation in DataCell is like the relational table representation in MonetDB (see Section 2.7). For each basket B there exists an extra column, the timestamp column, that reflects the arrival time of each tuple in the system.

In this way, we exploit all column-store benefits during query processing, i.e., a query needs to read and process only the attributes required and not all attributes of a basket. For example, assume a stream S that creates tuples with k different attributes. In a row-oriented system, each query interested in any of the attributes in S has to read the whole S tuples, i.e., all k attributes. In DataCell, we exploit the column-oriented structure of the underlying model,

and allow each query to bind only the attributes (of baskets) it is interested in, avoiding to access extra data and reducing their footprint. Furthermore, queries interested in different attributes of the same stream can be processed completely independently. We encountered the above scenarios for numerous queries in the Linear Road benchmark (Arasu et al., 2004) where each stream contains multiple attributes while not all queries need to access all of them.

3.2.3 Factories

In this section, we introduce the notion of factories. The factory is a convenient construct to model continuous queries. In DataCell, a factory contains all or just a subset of the operators of the query plan for a given continuous query. A factory may also contain (parts of) query plans from more than one query. For simplicity assume for now that each factory contains the complete plan of a single query.

Each factory has at least one *input* and one *output* basket. It continuously reads data from the input baskets, processes it and creates results which places in its output baskets. Each time a tuple t is being consumed from an input basket B (i.e., it is processed and it is not needed anymore), the factory removes t from B to avoid reading it again. We revisit these choices later on, when we discuss more complex processing schemes in Section 3.3.

A factory can also access persistent tables, deriving data from there and/or modifying their content. This feature is provided in the most natural way, since our base architecture is a DBMS. In this way, we can support query scenarios that require analysis of streaming and persistent data.

Having introduced the basic DataCell components, we can now consider how they interact at a higher level using Figure 3.1 as an example. A receptor captures incoming tuples and places them in basket B_1. Then, a factory that contains the full query plan of continuous query Q processes the streaming data in B_1 and the persistent data in table T. Subsequently, it places all qualifying tuples in the outer basket B_2 where the emitter can finally collect the results and deliver them to the client.

In general, at any point in time, multiple receptors wait for incoming tuples and place them into the disjoint baskets. A scheduler handles multiple factories that read these input baskets and place results into multiple output baskets where multiple emitters feed the interested clients with results. It is a multi-threaded architecture, where every single component (i.e., receptors, emitters and the factory scheduler) is an independent thread. Figure 3.2 illustrates an overview of the DataCell architecture, with all the participant components de-

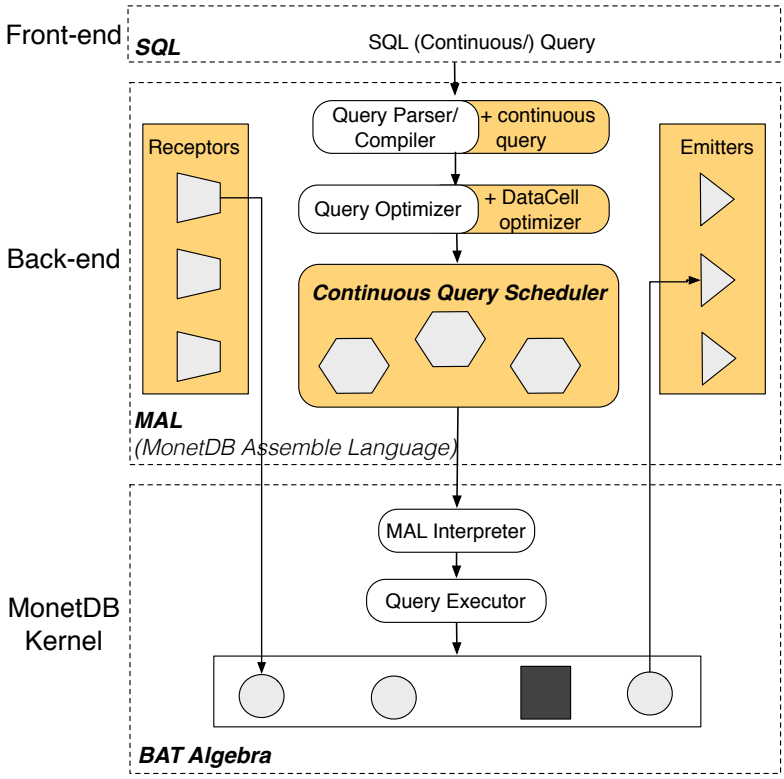

Figure 3.2: MonetDB/DataCell Architecture

scribed above and the extensions of the underlying MonetDB system. DataCell components are positioned between the MonetDB SQL compiler/optimizer and the DBMS kernel. The SQL compiler is extended with a few orthogonal language constructs to recognize and process continuous queries (see Chapter 4). The query plan as generated by the SQL optimizer is rewritten to a continuous query plan and handed over to the DataCell scheduler. In turn, the scheduler handles the execution of the plans.

Let us now describe the factories concept in more detail. A factory is a function containing a set of MAL operators corresponding to the query plan of a given continuous query. A factory is specified as an ordinary function; the

Algorithm 1 The factory for a continuous query that selects all values of attribute $X.a$ in range v_1-v_2.

```
1:  input = basket.bind(X.a)
2:  output = basket.bind(Y.a)
3:  while true do
4:      basket.lock(input)
5:      basket.lock(output)

6:      result = algebra.select(input,v1,v2)

7:      basket.empty(input)
8:      basket.append(output,result)

9:      basket.unlock(input)
10:     basket.unlock(output)
11:     suspend()
```

difference is that its execution state is *saved* between calls and that it permits re-entry other than by the first statement. Submitted queries are transformed to factories and the DataCell scheduler is responsible to trigger their execution (to be discussed below).

The first time that the factory is called, a stack frame is created in the local system to handle subsequent requests and synchronizes access. Its status is being kept around and the next time it is called it continues from the point where it stopped before. In Algorithm 1, we give an example of the factory DataCell constructs for the following simple range single stream continuous query, expressed in SQL-like syntax.

```
(q1) INSERT INTO Y (a)
     SELECT X.a
     FROM X
     WHERE X.a BETWEEN v1 and v2;
```

In query $q1$, we filter out all these tuples from stream X that their attribute value $X.a$ is between the values $(v1, v2)$. The query feeds the qualifying tuples to stream Y.

The factory in Algorithm 1 contains the full query plan (in this case just a single operator in line 6) where the original MonetDB operators are being used.

In particular, we use the `select` operator that belongs in the `algebra` module. The modules represent a logical grouping and they provide a name space to differentiate similar operations.

Essentially the factory contains an infinite loop to continuously process incoming data. Each time it is being called by the scheduler, the code within the loop executes the query plan. Then, it is put to sleep until it receives a wakeup call again from the scheduler; it continues at the point where it went to sleep.

Careful management of the baskets ensures that one factory, receptor or emitter at a time updates a given basket. In this way, as seen in Algorithm 1, the loop of the factory begins by acquiring locks on the relevant input and output baskets (line 4 and 5 respectively). The locks are released only at the end of the loop just before the factory is suspended. Both input and output baskets need to be locked exclusively as they are both updated, i.e., (a) the factory removes all tuples seen so far from the input baskets so that it does not process them again in the future to avoid duplicate notifications and (b) it adds result tuples to the output baskets. In the case of (sliding) window queries, only the tuples outside the current window are removed from the basket. In Chapter 5, we study and analyze in detail how to bring incremental stream processing for sliding window queries in the context of DataCell.

3.3 Query Processing

The previous section presented the basic components of the DataCell architecture. In this section, we focus on the interaction of these components in order to achieve efficient and scalable continuous query processing. In addition, we discuss further alternative directions that open the road for challenging research opportunities.

3.3.1 The DataCell Processing Model

The DataCell architecture uses the abstraction of the Petri-net model (Peterson, 1977) to facilitate continuous query processing. A Petri-net is a mathematical representation of discrete distributed systems. It uses a directed bipartite graph of *places* and *transitions* with annotations to graphically represent the structure of a distributed system. Places may contain (a) tokens to represent information and (b) transitions to model computational behavior. Edges from places to transitions model input relationships and, conversely, edges from transitions to places denote output relationships.

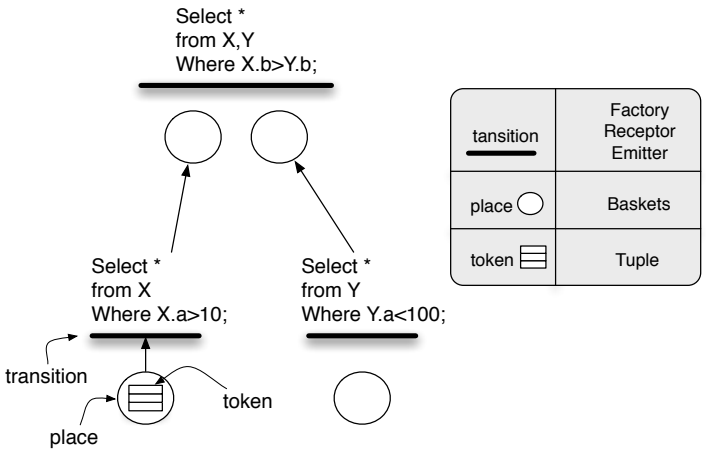

Figure 3.3: Petri-net Example

A transition fires if there are tokens in all its input places. Once fired, the transition consumes the tokens from its input places, performs some processing task, and places result tokens in its output places. This operation is atomic, i.e., it is performed in one non-interruptible step. The firing order of transitions is explicitly left undefined.

An advantage of the Petri-net model is that it provides a clean definition of the computational state. Furthermore, its hierarchical nature allows us to display and analyze large and small models at different scales and levels of detail.

In Figure 3.3, we show the mapping between the Petri-net and the Data-Cell components. Baskets are equivalent to Petri-net token place-holders while receptors, emitters and factories represent Petri-net transitions. Following the Petri-net model, each transition has at least one input and at least one output.

Each receptor has as input the stream it listens to and as output one or more baskets where it places incoming tuples. The user that sets up an application scenario, needs to specify the source of the data stream (e.g., which port the receptor listens to) and the target, where the receptor continuously appends the incoming data.

Each factory has as input one or more baskets from where it reads its input data. These baskets may be the output of one or more receptors or the output

of one or more different factories or mixed. The output of a factory is again one or more baskets where the factory places its result tuples.

Each emitter has as input one or more baskets that represent output baskets of one or more factories. The output of the emitter is the delivery of the result tuples to the clients representing the final state of the query processing chain.

The firing condition that triggers a transition (receptor, emitter or factory) to execute is the existence of input, e.g., at least one tuple exists in B, where B is the input basket of the transition. After an input tuple has been seen by all relevant transitions, it is subsequently dropped from the basket so that it is not processed again.

The DataCell kernel contains a *scheduler* to organize the execution of the various transitions. The scheduler runs an infinite loop and at every iteration it checks which of the existing transitions can be processed by analyzing their inputs. As a first approach the DataCell scheduler continuously re-evaluates the input of all transitions, implementing the round-robin algorithm; in the next section we study some alternative customized processing strategies (see Section 3.3.2).

In general, in order to accommodate more flexible processing schemes, the system may explicitly require a basket to have a minimum of n tuples before the relevant factory may run. For example, this is useful to enhance and control batch processing of tuples as well as in the case of certain window queries, e.g., a window query that calculates an average over a full window of tuples needs to run only once each window is complete. This may be achieved at the level of the scheduler for tuple-based window queries or at the level of the factory in the case of time-based queries, i.e., by plugging in auxiliary baskets that check the input for the window properties.

When a transition has multiple inputs, then *all* inputs must have tuples for the transition to run. In certain cases, to guarantee correctness and avoid unnecessary processing costs, auxiliary input/output baskets are used to regulate when a transition runs. Assume for example a sliding window join query q, with two input baskets B_1 and B_2 that reflect the join attributes. Every time q runs, we need to only *partially* delete the inputs as some of the tuples will still be valid for the next window. At the same time, we do not want to run the query again unless the window has progressed, i.e., new tuples have arrived on either input. Adding a new auxiliary input basket B_3 solves the problem. The new basket is filled with a single tuple marked *true* every time at least one new tuple is added to either B_1 or B_2 and is fully emptied every time q runs.

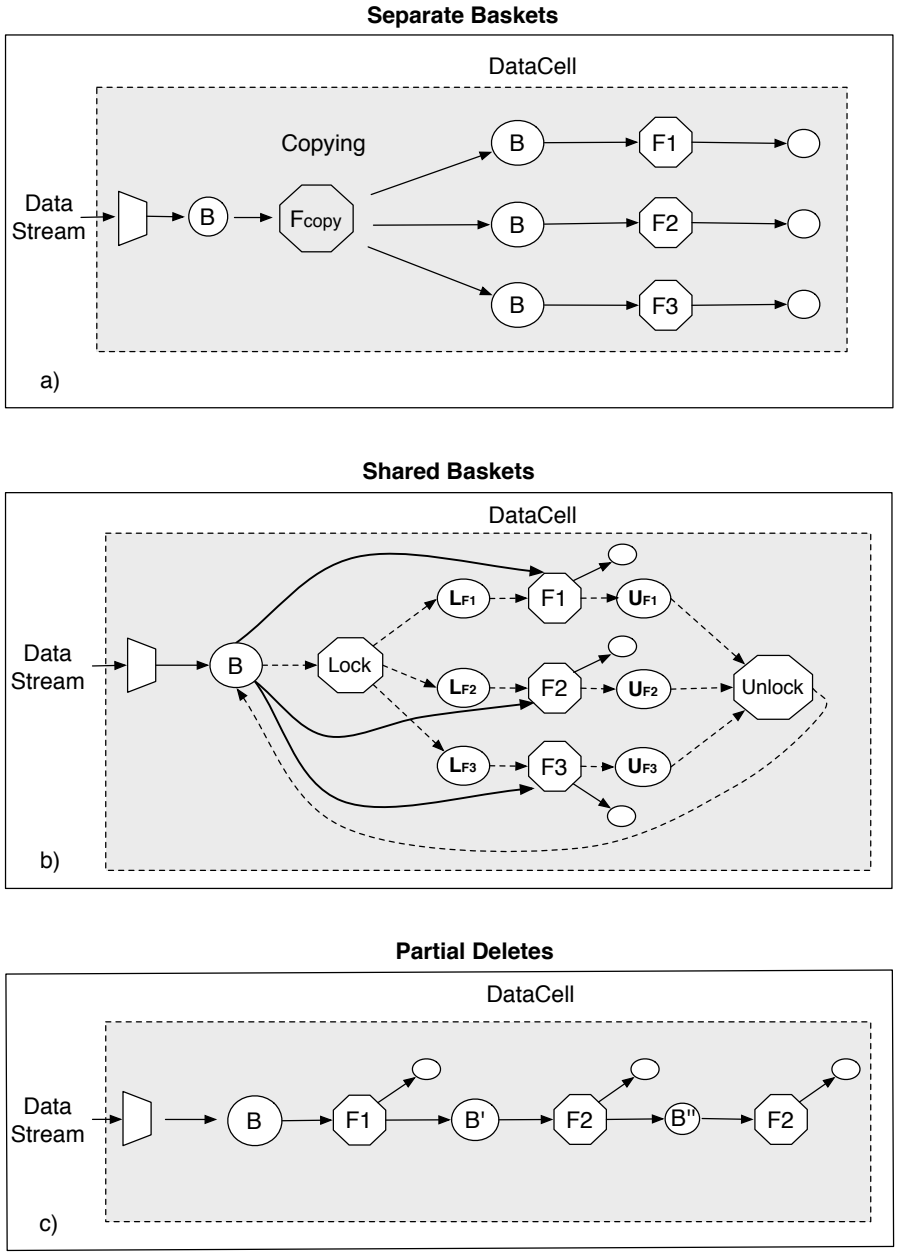

Figure 3.4: Examples of alternative processing schemes

3.3.2 Processing Strategies

Up to now, for ease of presentation, we have described the DataCell in a very generic way in terms of how the various components interact. The way factories and baskets interact within the DataCell kernel defines the query processing scheme. By choosing different ways of interaction, we can make the query processing procedure more efficient and more flexible. In this section, we discuss our first approach to validate the feasibility of the DataCell vision and subsequently we point to further challenging directions.

Separate Baskets

Our first strategy, called *separate baskets*, provides the maximum independence to each query. Each continuous query is fully encapsulated within a single factory. Furthermore, each factory F_i has its own input baskets that only F_i accesses to read and update, without the need of concurrency control. The latter has the following consequences. In the case that k factories, where $k > 1$, are interested in the same data, then this data has to be placed in more than one baskets upon arrival into the system, i.e., the data has to be replicated k times, once for each relevant factory. This is done by automatically inject a *copy* factory between the submitted factories and the original data source. The benefit is that the factories can run completely independently, avoiding any conflict of interest situation, without the need to carefully schedule their accesses on the baskets. An example is given in Figure 3.4(a).

By exploiting the flexibility of building on top of a column-store, we can minimize the overhead of the initial replication needed since the system handles and stores the data one column/attribute at a time. For example, depending on the workload there may be less I/O and memory bandwidth requirements. In this way, if a factory is interested in two attributes a, b of stream S, then we need to copy in its baskets only the columns a and b and not the full tuples of S containing all attributes of the stream.

Shared Baskets

The first strategy, described above, is the baseline to study the properties and the potential of DataCell. Our second strategy, called *shared baskets*, makes a first step towards exploiting query similarities. The motivation is to avoid the initial copying of the first strategy by *sharing* baskets between factories. Each attribute from the stream is placed in a *single* basket B and all factories interested in this attribute have B as an input basket.

Naturally, sharing baskets minimizes the overhead of replicating the stream in many baskets. In order to guarantee correct and complete results, the next step is to regulate the way factories access their input baskets such that a tuple remains in its basket until all relevant factories have seen it. Thus, the *shared basket* strategy steps away from the decision of forcing each single factory to remove the tuples it reads from an input basket after execution based on the semantics of the respective query.

To achieve the above goal, for every basket B which shared as input between a group of k factories, we add two new factories, the *locker* and the *unlocker*. An example is shown in Figure 3.4(b). The locker factory, $Lock$, is placed between B and the originally attached factories (i.e., submitted continuous queries). Once B contains a number of new tuples, $Lock$ runs. Its task is to simply lock B. The output of $Lock$ is k baskets, one for each waiting factory, i.e., L_{F1}, L_{F2}, ..., L_{FK}. In each one of these outputs, $Lock$ writes a single tuple containing a bit attribute marked "true". Then, all factories can read and process B but without removing any tuples. Every factory F_i has an extra output basket, apart from the expected result basket, where it writes a single bit attribute to mark that its execution over the locked version of the input basket B is over. In Figure 3.4(b) this is shown as U_{Fi}. These output baskets are inputs to the unlocker factory $Unlock$. The task of $Unlock$ is that once all factories have seen the content of the input basket i.e., once all output baskets U_{F1}, U_{F2}, ..., U_{FK} are marked, it removes from B all tuples covered by the semantics of the factories, and subsequently it unlocks B so that the receptor can insert new tuples.

This strategy entails that if N factories share one basket B, then DataCell needs to wait until all N factories finish reading B. Only then, we can apply deletes and move on to the next data batch. These observations make the shared baskets strategy more appropriate for "delete all" queries, or sliding window queries with the same sliding step.

Using this simple scheme, we can use shared baskets and exploit common query interests. It nicely shows that the DataCell model is generic and flexible. Furthermore opportunities may come by exploiting recent techniques and ideas for sharing retrieval and execution costs of concurrent queries in databases (Harizopoulos et al., 2005).

Further ideas for sharing data streams. Another way to achieve data stream sharing among co-existing continuous queries, is to follow a differential approach. The idea is that apart from the original basket that constitutes the common input source for multiple factories, each factory maintains a separate set of arrays, i.e., one array for each basket column. In each auxiliary basket

the factory marks the tuples it has already consumed. Thus, every time the scheduler triggers a factory, it should first merge the two different versions of its input baskets. More precisely, each factory should merge the original basket which is the same for every interested factory and the expiration basket which is unique for every factory. In this way, a factory always gets all valid tuples and then it can continue with the rest of the query evaluation steps. In this scheme, a garbage collector should have access to both the input basket and to all the factory baskets that maintain the expired tuples of the co-existing continuous queries. This is necessary such that it can periodically clean the tuples that are not useful any more by any query, lightening the total storage space.

This direction is not further explored in this thesis. We discuss it here as a valid alternative way to explore data stream sharing by slightly modifying the underlying MonetDB processing scheme. Our intuition is that this scheme would be appropriate for application scenarios with relatively low update rates of data streams and continuous queries with high commonality on tuple expiration (i.e., rate and value wise).

Partial Deletes

The shared baskets strategy, described above, removes the tuples from a shared input basket only once all relevant factories have seen it. The next strategy is motivated by the fact that not all queries on the same input are interested in the same part of this input. For example, two queries q_1 and q_2 might be interested in disjoint ranges of the same attribute. Assume q_1 runs first. Given that the queries require disjoint ranges, all tuples that qualified for q_1 are for sure not needed for q_2. This knowledge brings the following opportunity; q_1 can remove from B all the tuples that qualified its basket predicate and only then allow q_2 to read B. The effect is that q_2 has to process less tuples by avoiding seeing tuples that are already known not to qualify for q_2. All we need is an extra basket between q_1 and q_2 so that q_2 runs only after q_1. Figure 3.4(c) shows an example where three queries, encapsulated in $F1$, $F2$ and $F3$ factories respectively, create such a chain. Each factory proceeds to the query execution, appending tuples to its attached output basket, and in parallel leaves behind the left-overs of its input, e.g., $B' \subseteq B$.

This strategy opens the road for even more advanced ways of exploiting query commonalities. For example, the idea to incrementally build indices based on the particular needs of each continuous query follows the philosophy of the partial deletes mechanism we described above. There, we could choose when it is worth to build an index on streaming data, e.g., as in (Idreos, 2010) where indices are

build during the execution stage, which will also be valuable for the queries we are going to execute afterwards. We have not covered these techniques in this thesis, but it is an interesting future research direction.

3.3.3 Research Directions

In the previous subsection we introduced a number of different processing strategies and discussed how they do fit in the DataCell model. The goal of this chapter is not to propose the ultimate processing scheme. We introduce the DataCell model and argue that it is a promising direction that opens the road for a wide area of research directions under this paradigm. There is a plethora of possibilities one may consider regarding the processing strategies in data streams, e.g., (Sharaf et al., 2008).

The most challenging directions in our context come from the choice to split the query plan of a single query into multiple factories. The motivation to do this may come from multiple different reasons. For example, consider the shared baskets strategy. Each factory in a group of factories sharing a basket, will conceptually release the basket only after it has finished its full query plan. Assume two query plans, a simple (lightweight) query q_1 and a quite complex (heavy) query q_2 that needs a considerable higher amount of processing time compared to q_1. With the shared baskets strategy we force q_1 to wait for q_2 to finish before we can allow the receptor to place more tuples in the shared basket so that q_1 can run again. A simple solution is to split a query plan into multiple parts so that the part that needs to read the basket becomes a separate factory. This way, the basket can be released once a factory has loaded its tuples, effectively eliminating the need for a fast query to wait for a slow one.

Another natural direction that comes to mind once we decide to split the query plans into multiple factories is the possibility to share not only baskets, but also execution cost. For example, queries requiring similar ranges in selection operators can be supported by shared factories that give output to more than one query's factories. Auxiliary factories can be plugged in to cover overlapping requirements.

3.4 Optimizer Pipeline and DataCell Implementation

One essential part of every data management system is the optimization phase. The query optimizer is responsible for finding the most appropriate query plan, i.e., the proper way to execute a query. Then the execution engine is responsible for actually evaluating a query over the proper data.

In this section, we discuss in more detail the optimization steps in our MonetDB experimentation platform and we pinpoint on the design changes needed for DataCell. DataCell receives a *one-time* query plan which is produced by the MonetDB optimizer and it transforms it to a *continuous* query plan. It achieves this by introducing new optimization rules and transformations. The transformations required for the first reevaluation-based design of DataCell are quite simple. More advanced transformations are required to support incremental and window query processing. Those are discussed in Chapter 5.

The code produced by MonetDB/SQL is passed and massaged by a series of optimization steps, denoted as an *optimizer pipeline*, as we discussed in Section 2.7. Each pipeline consists of a sequence of MAL function calls that inspect and transform the plan. The final result of the optimizer steps is what it is submitted to the execution engine.

The basic DataCell optimizer pipeline is the following:

```
datacell_pipe=inline,remap,evaluate,costModel,coercions,emptySet,
aliases,deadcode,constants,commonTerms,datacell,emptySet,aliases,
deadcode,reduce,garbageCollector,deadcode,history,multiplex
```

The interested reader can refer to MonetDB documentation (MonetDB, 2012) for further analysis of each individual optimization rule. For example, the `costModel` optimizer inspects the SQL catalog for size information, the `deadcode` removes all code not leading to used results, the `reduce` optimizer reduces the stack space for faster calls, and the `emptySet` removes empty set expressions. Note that most of these rules in the above pipeline are optimizations we also use in the traditional OLAP scenario where we handle one-time queries.

In MonetDB, the optimizer pipelines contain dependencies. For example, it does not make much sense to call the `deadcode` optimizer too early in the pipeline, although it is not an error.

The `datacell` optimization set of rules is exclusively created to cover the needs of the continuous query scenario. Its main role is to transform a *one-time*

query plan to a *continuous* query plan. The main actions it takes are as follows.

- It wraps the MAL plan in a factory (see Section 3.2.3 and Algorithm 1).
- It adds in the proper place of the MAL plan the infinite loop that guarantees continuous query processing. Instructions that should be evaluated only once, such as basket binds, remain outside the loop.
- It plugs in the appropriate data cleaning instructions for proper tuple expiration.
- It introduces the locking and unlocking scheme for the source and target baskets of the query.
- It discards the unnecessary (secondary) arrays that by default represent the deletions and updates of each column in MonetDB. In addition, it cleans the corresponding commands that the discarded arrays participate (explicitly and implicitly).

Any optimizer in MonetDB, once it is called needs to traverse the MAL plan and collect information to local data structures that it uses to modify the input plan. In some cases, some information is passed from one optimizer to another for further analysis.

At this level, the `datacell` optimizer is only responsible to implant streaming functionalities in a *normal* query plan. In Chapter 5, we show how we extended the optimization phase with new set of rules in order to support incremental stream processing for sliding window queries.

3.5 Experimental Analysis

In this section, we report on experiments using our DataCell implementation on top of MonetDB v5.6. All experiments are on a 2.4GHz Intel Core2 Quad CPU equipped with 8GB RAM. The operating system is Fedora 8 (Linux 2.6.24). Our analysis consists of two parts, (a) an evaluation of the individual parts of the DataCell using micro-benchmarks to assess specific costs, and (b) an evaluation of the system at large using the complete Linear Road benchmark (Arasu et al., 2004).

3.5.1 Micro-benchmarks

A stream-based application potentially involves a large number of continuous queries. To study the basic DataCell performance, we first focus on a simple topology, called *Query chain*, to simulate multi-query processing of continuous queries inside the DataCell. An example is given in Figure 3.5. It reflects a situation where the most general query is evaluated first against the incoming tuples. Then, it passes the qualifying tuples to the next query in the pipeline, which is less general and so on.

Metrics

Our metrics are the following. We measure the average *latency* per tuple, i.e., the time needed for a tuple to pass through all the stages of the stream network. Thus, the latency $L(t)$ of a tuple t is defined as $L(t) = D(t) - C(t)$, where $C(t)$ is the time on which the sensor created t, while $D(t)$ is the time on which the client received t.

In addition, we measure the *elapsed* time per batch of tuples. For a batch b of k tuples this metric is defined as $E(b) = D(t_k) - C(t_1)$ where t_1 is the first tuple created for b and t_k is the last tuple of b delivered to the client.

Finally, we measure the *throughput* of the system which is defined as the number of tuples processed by the system divided by the total time required.

Interprocess Communication Overhead

Targeting real-world application, it is not sufficient to focus only on the performance within the kernel of a stream engine. Communication costs between devices controlling the environment, e.g., sensors, clients and the kernel have a significant impact on the effectiveness and performance. For this reason, we experiment with a *complete pipeline* that includes the cost of the data shipping from and to the kernel.

We implemented two independent tools, the sensor and the actuator. The sensor module continuously creates new tuples, while the actuator module simulates a user terminal or device that posed one or more continuous queries and is waiting for answers. The sensor and the actuator connect to the DataCell through a TCP/IP connection. They run as separate processes on a single machine.

In the following experiment, we measure the elapsed time and the throughput while varying the number of queries. The sensor creates 10^5 random two-column tuples. For each tuple t, the first column contains the timestamp that this tuple

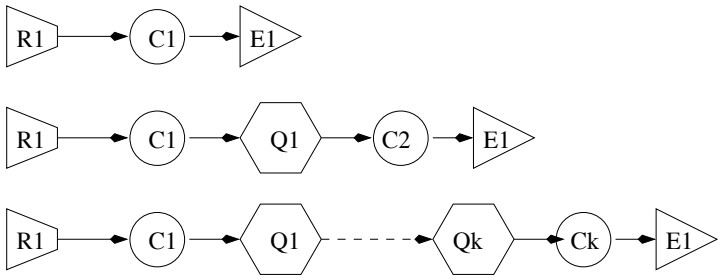

Figure 3.5: The Query Chain topology

was created by the sensor, while the second one contains a random integer value. We use simple SELECT * queries. Thus, within the kernel every query passes all tuples to the next one which reflects the worst case scenario regarding the data volume flowing through the system.

Given that we have separate sensor and actuator processes, the time metrics to be presented include (a) the communication cost for a tuple to be delivered from the sensor to the DataCell, (b) the processing time inside the engine and (c) the communication cost for the tuple to be sent from DataCell to the actuator. To assess the pure communication overhead, we also run the experiments by removing the DataCell kernel from the network. This leaves only the sensor sending tuples directly to the actuator.

Figure 3.6(a) depicts the elapsed time. It increases as we add more queries in the system and grows up to 200 milliseconds for the case of 64 queries. The flat curve of the sensor to actuator experiment demonstrates that a significant portion of this elapsed time is due to the communication overhead. The less work the kernel has to do, the higher the price of the communication overhead is, relative to the total cost.

In addition, Figure 3.6(b) shows. that the maximum throughput we achieve simply by passing tuples from the sensor to the actuator is around $2.2 * 10^4$ tuples/sec. Naturally, with the DataCell kernel included in the loop the throughput significantly decreases. Again the larger the number of queries in the system, the lower the throughput becomes.

Pure Kernel Activity

At first sight the performance figures discussed above do not seem in line with common belief. Unfortunately, the literature on performance evaluation of

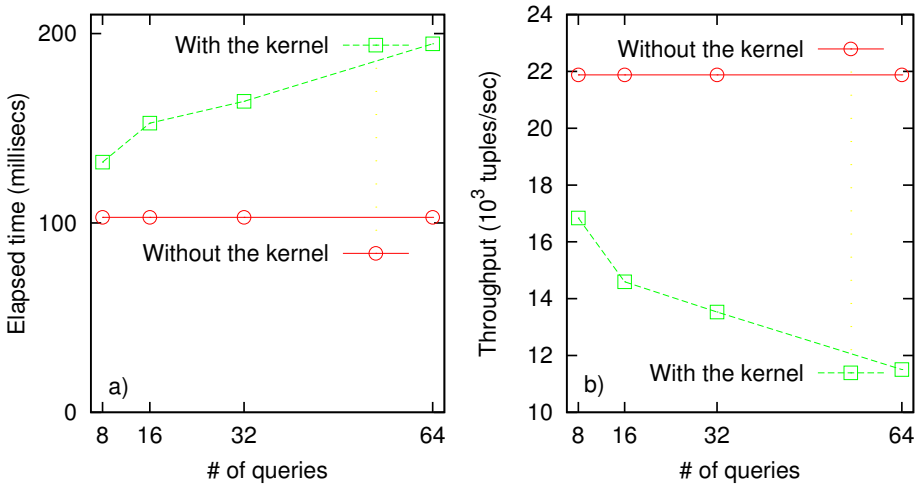

Figure 3.6: Effect of inter-process communication

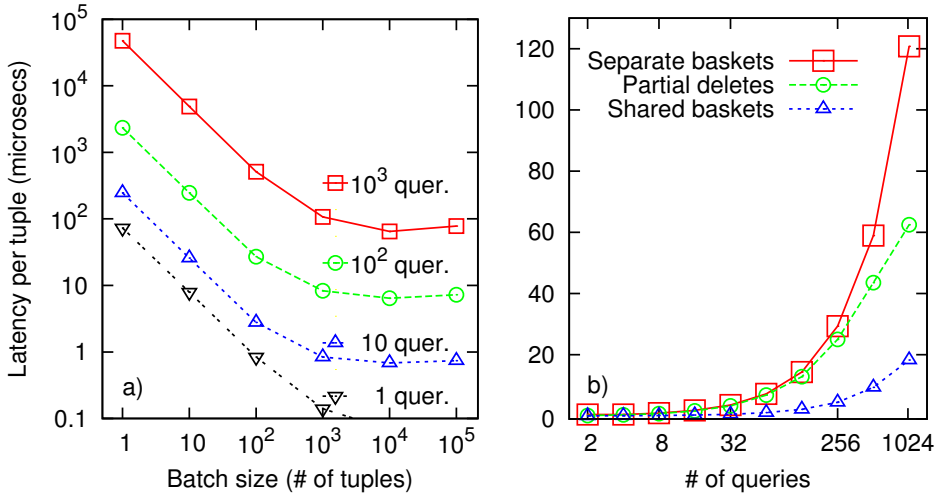

Figure 3.7: Effect of batch processing and strategies

stream engines does not yet provide many points of reference. GigaScope (Cranor et al., 2003) claims a peak performance up to a million events per second by pushing down selection conditions the Network Interface Controller. On the contrary, early presentations on Aurora report on handling around 160K msg/sec. Comparing Aurora against a commercial DBMS, systemX, the systems show the capability to handle between 100 (systemX) and 486 (Aurora) tuples/second (Arasu et al., 2004). Two solutions for systemX are given, one based on triggers and stored procedures, and another one based on polling.

Most research papers in the literature for data stream system evaluation ignore the communication overhead demonstrated above. The message throughput is largely determined by the network protocol, i.e., how quickly can we get events into the stream engine. To measure the performance of the pure DataCell kernel without taking into account any communication overheads, we use the query chain topology. Our experiments show that each factory can easily handle $7 * 10^6$ events per second. These numbers are in-line with the high-volume event handling reported by others in similar experiments, i.e., without taking into account communication costs. The interesting observation is that there is a slack time due to this overhead and the system can exploit this time in many ways, e.g., creating various indices, collecting statistics, etc.

Batch Processing

Here, we demonstrate the effect of batch processing within the DataCell engine using the separate baskets architecture. We set up the experiment as follows. 10^5 incoming tuples are randomly generated with a uniform distribution. Each tuple contains an attribute value randomly populated in [0,100] and a timestamp that reflects its creation time. All queries are single stream, continuous queries of the following form.

```
SELECT S.a
FROM S
WHERE v1 < S.a < v2
```

All queries select a random range with 10% selectivity. Figure 3.7(a) depicts the average latency per tuple for various different numbers of installed queries and while varying the batch size (T) used in query processing. The case of $T = 1$ demonstrates the impact of the traditional processing model of handling one tuple at a time. We clearly see that the latency significantly decreases as we increase the batch size materializing a benefit of roughly three orders of magnitude. An important observation is that the benefits of batch processing

increase with a higher rate up to a certain batch size and then the improvement is much less. When the batch size becomes very big, performance starts to degrade especially for the case of the maximum number of queries. This is due to the delay time needed, i.e., the average time a tuple has to wait for more tuples to arrive so that the desired batch size is reached. Only then the tuples can be processed. However, there is a point that this delay time becomes so big that overshadows the benefits of grouped processing, i.e., performance does not improve anymore or even degrades. In our experiment this point appears at $T = 10^3$. Optimally setting and adapting the batch size depending on the queries and system status is an open research problem.

Alternative Strategies

Let us now study the various query processing strategies discussed in Section 3.3.2. The previous experiment used the basic separate baskets approach. Here, we demonstrate the benefits of using alternative strategies, i.e., shared baskets and partial deletes. The set-up is similar to the previous experiment but this time the batch size is constant at $T = 10^3$.

Figure 3.7(b) presents the results for various different numbers of installed queries. Naturally, the two alternative strategies significantly outperform the basic separate baskets approach. The reason is that both these strategies avoid the procedure of creating the extra baskets which requires to replicate the stream data at multiple locations once for each query. The higher the number of queries in the system, the bigger the benefit. Furthermore, the shared baskets approach achieves much better performance, than partial deletes especially as the number of queries increases. This time the reason is that the shared baskets approach is a more lightweight one regarding basket management. With partial deletes, every query needs to *modify* its input basket to remove tuples that the next query does not need. Although the next query can execute much faster due to analyzing less data, the overhead of continuously modifying and reorganizing the baskets is significant to overshadow a large portion of this benefit. On the other hand, the shared baskets approach does not need to modify the data at all. Only once all queries are finished, then the appropriate tuples are removed from the input baskets in one simple step.

3.5.2 The Linear Road Benchmark

In this section, we analyze the performance of our system using the Linear Road benchmark (Arasu et al., 2004). This is the only well-known benchmark

developed for testing stream engines. It is a very challenging and complicated benchmark due to the complexity of the many requirements. It stresses the system and tests various aspects of its functionality, e.g., window-based queries, aggregations, various kinds of complex join queries; theta joins, self-joins, etc. It also requires the ability to evaluate not only continuous queries on the stream data, but also historical queries on past data. The system should be able to store and later query intermediate results. Due to the complexity, only a handful of implementations of the benchmark exist so far. Most of them are based on a low level implementation in C which naturally represents a specialized solution that not clearly reflects the generic potential of a system. In this chapter, we implemented the benchmark in a generic way using purely the DataCell model and SQL. We created numerous SQL queries that interact with each other via result forwarding (details are given below).

The Benchmark

Let us now give a brief description of the benchmark. It simulates a traffic management scenario where multiple cars are moving on multiple lanes and on multiple different parallel roads. In Linear City each expressway has four lanes in east and west direction. In three middle lanes of each direction cars are traveling, while the external lane is devoted to entrance and exit to the expressway. Each expressway is 100 miles-long and consists of 100 equally divided segments of 1 mile long each. Figure 3.8 illustrates an example segment, as it was originally presented by the authors of the Linear Road Benchmark. Every vehicle is equipped with a sensor that emits its exact position every 30 seconds. The system is responsible to monitor the position of each car. It collects and analyzes the incoming position reports, to create statistics about traffic conditions on each segment, of each expressway, for every minute or to immediately detect an accident when occurs. An accident is detected when two or more cars are in the same position for 4 continuous timestamps. Based on these statistics it dynamically determines the toll rates and charges each individual driver the relevant amount. In addition, the system needs to continuously monitor historical data, as it is accumulated, and report to each car the account balance and the daily expenditure. Furthermore, the benchmark poses strict time deadlines regarding the response times which must be up to X seconds, i.e., an answer must be created at most X seconds after all relevant input tuples have been created. X is 5 or 10 seconds depending on the query (details below).

The benchmark contains a tool that creates the data and verifies the results. The data of a single run reflects three hours of traffic, while there are multiple

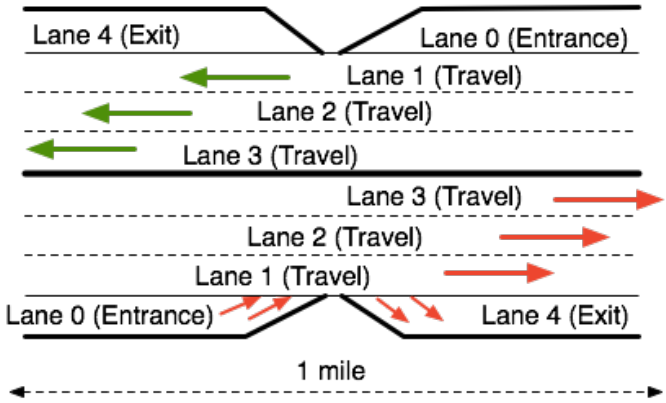

Figure 3.8: Expressway Segment in LRB (Arasu et al., 2004)

scale factors that increase the amount of data created for these three hours, e.g., for scale factor 0.5 the system needs to process $6 * 10^6$ tuples, while for scale factor 1 we need to process $1.2 * 10^7$.

Implementation in the DataCell

Our implementation of the benchmark was done completely in SQL and by exploiting the power of a modern DBMS. We translated the requirements of the benchmark in the form of a quite complex group of numerous SQL queries. The original queries can be found in the validator tool of the benchmark. We modified the queries into DataCell continuous queries. In particular there are 38 queries, logically distinguished in 7 different collections ($Q1$-$Q7$). Figure 3.9 gives a high level view of the various collections and the number of queries within each one. The interested reader could refer to the sources of the benchmark, as they are provided by the authors (Linear Road Benchmark, 2012). There are numerous complex queries, e.g., self-join queries, theta join queries, nested queries, aggregation, sliding window queries, etc. Only four of the query collections are output queries, i.e., $Q4$, $Q5$, $Q6$ and $Q7$ which create the final results requested by the benchmark. The rest process the data and create numerous intermediate results that pass from one query to another until they reach one of the output queries.

In order to verify the baseline of our approach and keep the implementation

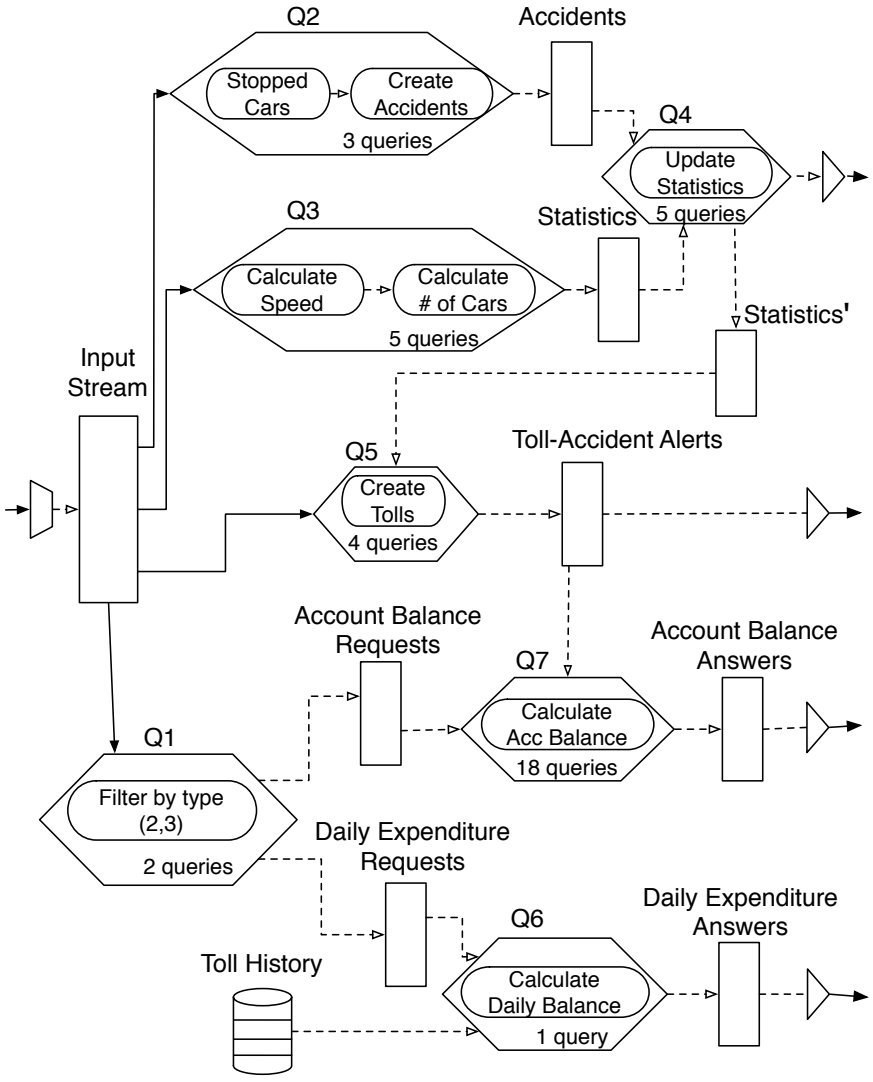

Figure 3.9: Linear Road benchmark in DataCell

simple, given the complexity of the benchmark, as a first step each collection of queries becomes a single factory. It takes its input from another query collection and gives its output to the next collection. Within each query collection the individual queries form a simple pipeline, while as seen in Figure 3.9, a query in one collection might have multiple inputs from different collections. Regarding the time deadlines, the output collections Q_4, Q_5 and Q_7 have a 5 seconds goal while Q_6 has a 10 second goal.

To verify the feasibility of the DataCell approach, as a first step, we purely exploited the functionality provided by the DBMS using operators provided by the system to handle the various columns. These operators have been developed for use in the pure DBMS arena. Early analysis showed that a number of new simple operators can increase the performance up to 20-30%. This was mostly in the cases of the operators used to remove tuples from a basket. Due to the complexity of the benchmark, there are numerous cases where we do not need to simply empty a basket. Instead we need to selectively remove tuples based on numerous restrictions, e.g., window-based queries, multiple queries needing the same data but with different restrictions, etc. To achieve the required functionality, we often had to combine 3-4 operators which introduces a significant delay by processing the same column over and over again. In most of the cases, creating a new operator, that, for example, in one go removes a set of tuples by shifting the remaining tuples in the positions of the deleted ones, gives a significant boost in performance.

Evaluation

Let us now proceed with the performance results. Figure 3.11 shows the performance during the whole duration of the benchmark (three hours) for scale factor 1. Graph 3.11(a) shows the total number of tuples entered the system at any given time while the rest of the graphs show the processing time needed for each query collection. Each time a collection of queries runs, i.e., because there was new input for its first query, then all its queries will run, one after the other, if the proper intermediate results are created. One, some or even all its queries may run in one go depending on the input. The graphs in Figure 3.11 depict the response time for each query collection Q_i, every time Q_i was activated through the three hours of the benchmark.

The first observation is that the response time is kept low for all queries. Most of the collections need much less than one second with query collection 7 being the most resource consuming. It contains 18 complex queries with multiple join and window restrictions. For most of the query collections, we observe that

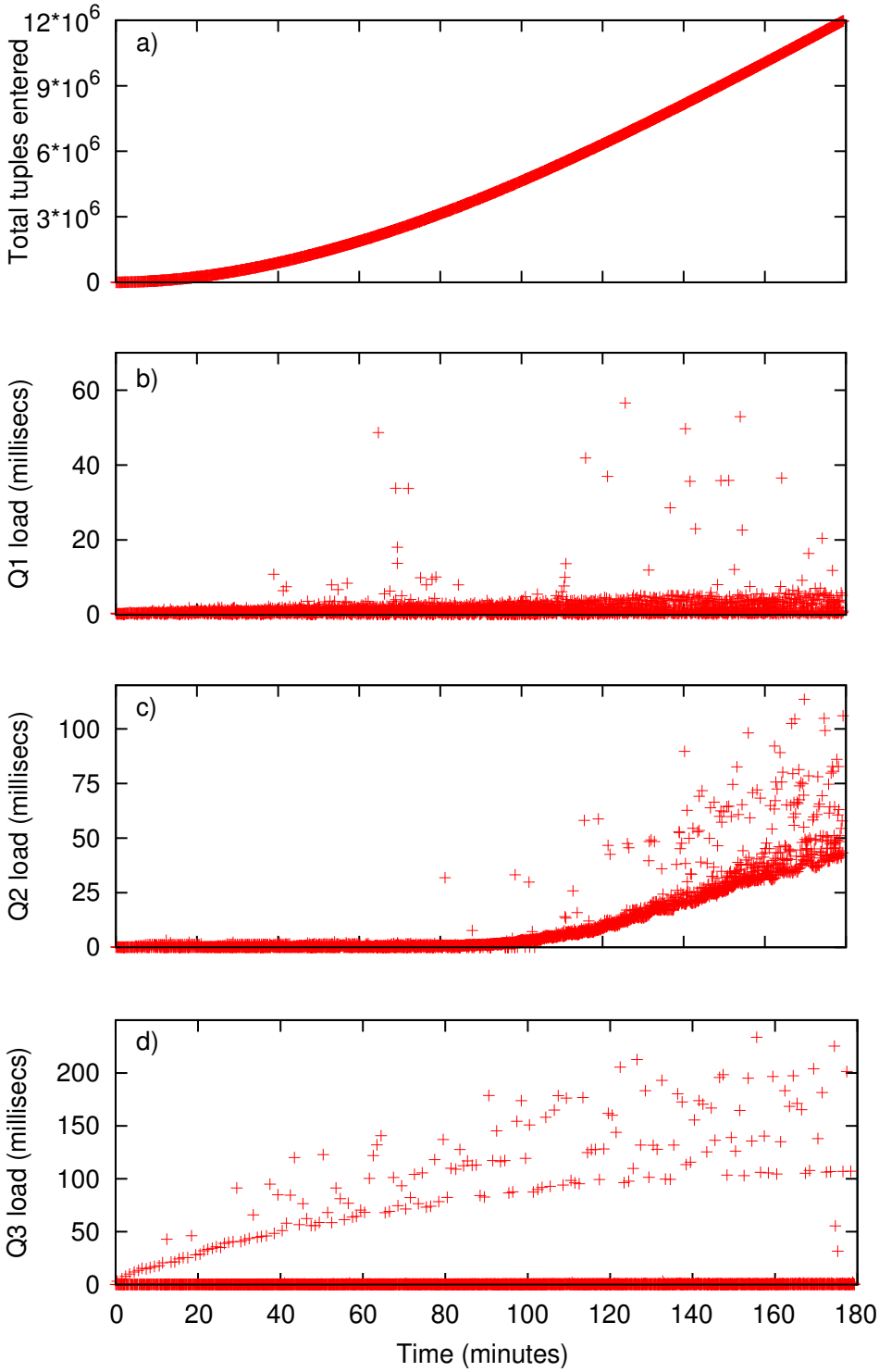

Figure 3.10: System load for each query collection (Q1-Q3)

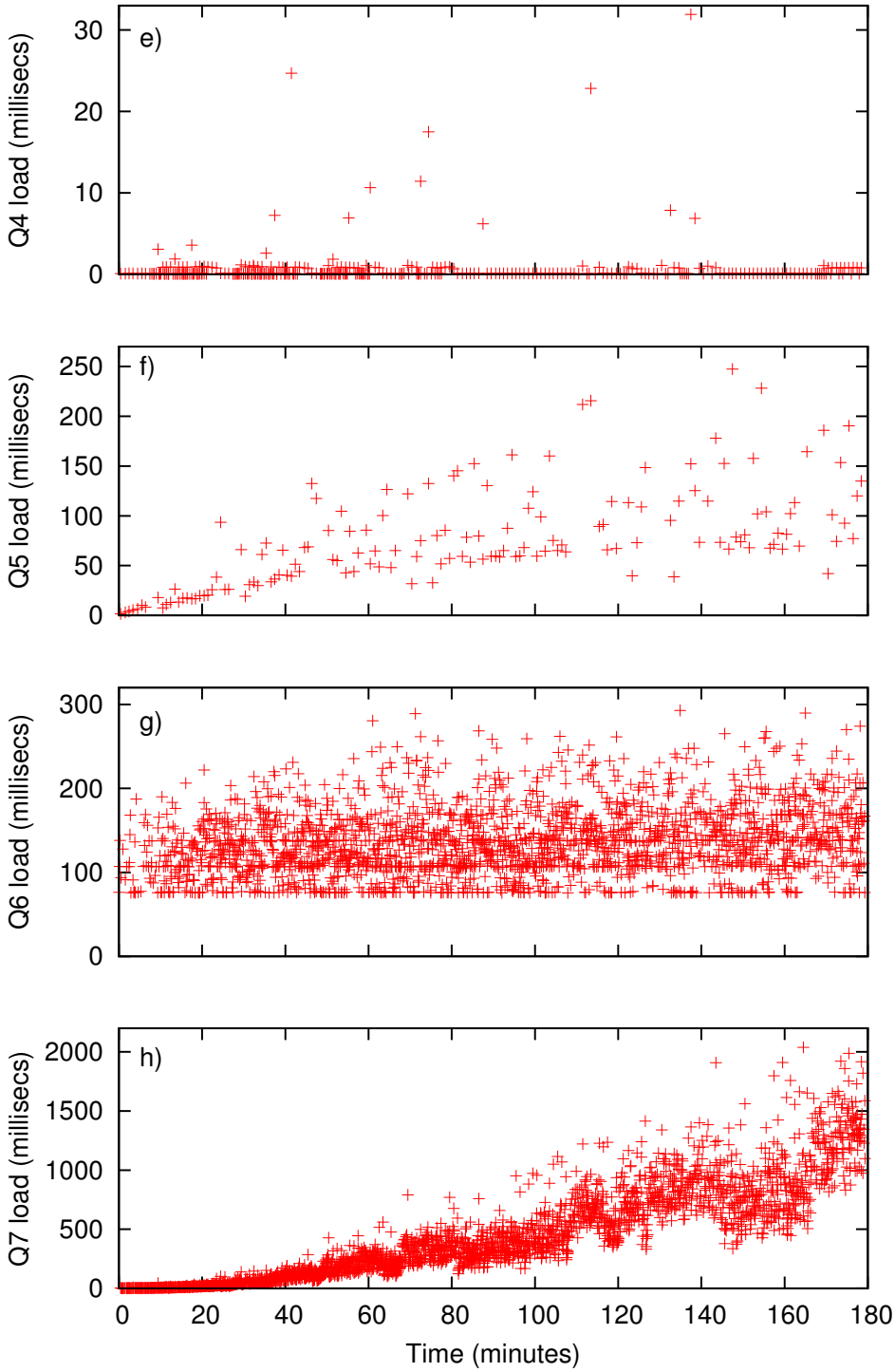

Figure 3.11: System load for each query collection (Q4-Q7)

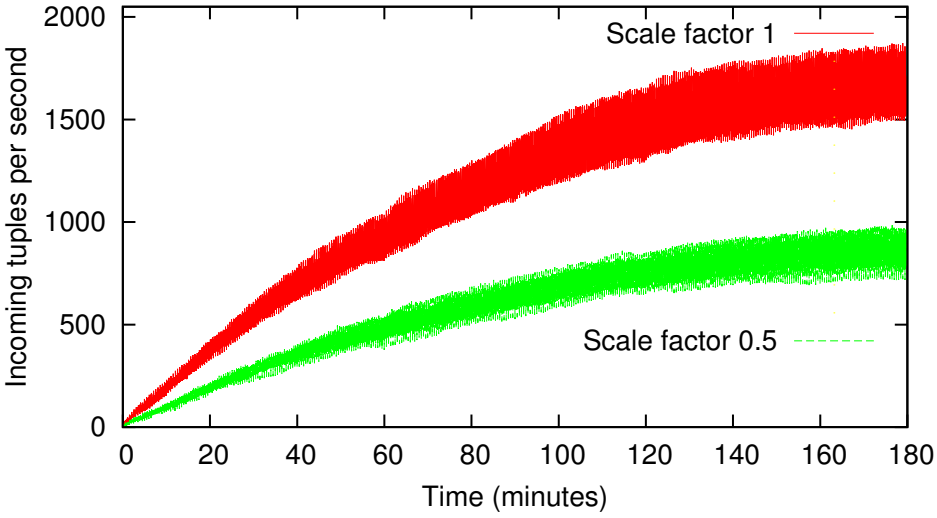

Figure 3.12: Data distribution during the benchmark

the cost is increased as more data arrives. This is due to a number of reasons. First, data and intermediate results is accumulated over time creating bigger inputs for the various queries. Most importantly, in many cases it is the content of the incoming data that triggers more work. For example, the second query collection (Figure 3.11(c)) is the one detecting the accidents. With the way data is created by the benchmark (for scale factor 1), accidents occur with a continuously increasing frequency after one hour. This is when we see the queries in Figure 3.11(c) to increase their workload as to compute the various accident situations for each car, in each lane etc. In turn, these queries create bigger inputs for the queries in the next query collections and so on.

Furthermore, the benchmark is designed in such a way that more data enters the system, the more the time goes by. This is demonstrated in Figure 3.12 where we show the number of tuples that enter the system every second. For example, for scale factor 1, 15 to 20 tuples per second arrive at the beginning, while towards the end of the three hours run we get up to 1700 tuples per second. All categories scale nicely achieving to process the extra data as the benchmark evolves. Even the most expensive query collection, $Q7$, manages to maintain performance levels below 2 seconds which is well below the 5 seconds goal.

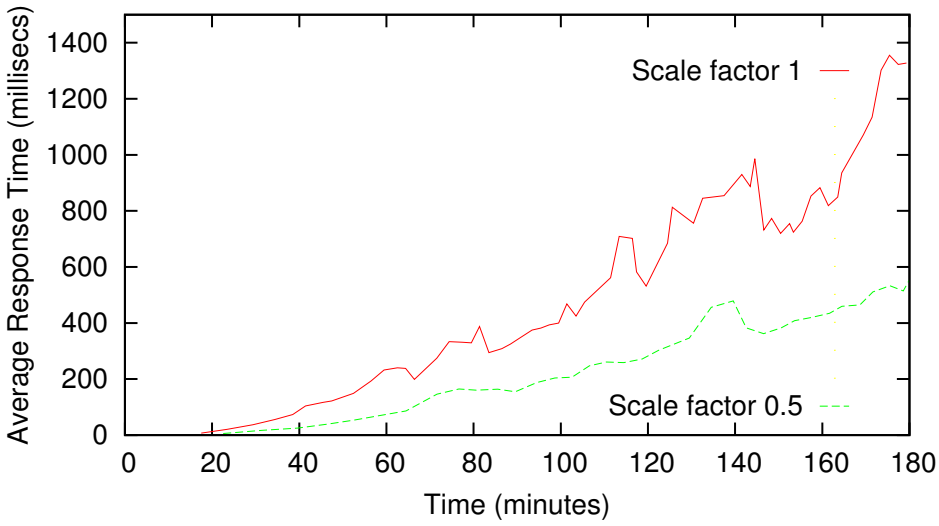

Figure 3.13: Average response time for $Q7$

Furthermore, Figure 3.13 depicts the average response time for query collection $Q7$ which is one of the output results of the benchmark. This metric is common when evaluating the benchmark, e.g., (Jain et al., 2006) as this collection defines the performance of the system by containing the most heavyweight queries, dominating the system resources (see Figure 3.11). The average response time is defined as the average processing time needed for the queries in this collection. It is measured every time 10^6 new tuples enter *this* collection by calculating the average time needed to process these 10^6 tuples.

Figure 3.13 shows that the response time is continuously kept low, below 1.5 seconds, even towards the end of the three hours run when data arrives at a much higher frequency. Going from scale factor 0.5 to 1, the performance scales nicely considering the much higher volume of incoming data.

The results observed above are similar to what specialized stream systems report, e.g., (Arasu et al., 2004). They indicate that the DataCell model can achieve competitive performance with a very generic implementation of the benchmark and with the most basic system architecture. It shows that a modern DBMS can be successfully turned into an efficient stream engine. Future research on optimization and alternative architectures is expected to bring even

more performance, exploiting the power of relational databases but also the stream properties to the maximum.

3.6 Summary

In this chapter, we introduced the basic DataCell architecture, a radically different approach in designing a stream engine. The system directly exploits all existing database knowledge by building on top of a modern column-store DBMS kernel. Incoming tuples are stored into baskets and then they are queried and removed from there by multiple factories (queries/operators) waiting in the system. Our design allows for numerous alternative ways of interaction between the basic components, opening the road for interesting and challenging research directions. This chapter presents the basic approaches and through a complete implementation of the DataCell prototype, it shows that this is a very promising direction that together with the experience gained from the existing stream literature, can lead to very interesting research opportunities.

The following chapter presents a semi-procedural query language proposed in the context of DataCell, and in Chapter 5 we study the crucial pure stream processing problem of incremental processing for window-based continuous queries.

Chapter 4

Query Language*

4.1 Introduction

In the previous chapter we presented the basic DataCell architecture. We defined the new concepts introduced in our underlying kernel in order to support efficient data stream processing. DataCell fundamentally changes the way that data streams are handled and processed, trying to exploit many traditional core database techniques and ideas. We implemented and ran the Linear Road benchmark and a number of micro-benchmarks that show that our approach to implant stream processing functionalities in the heart of a modern database kernel is not only a realistic but also a promising direction that deserves further study.

In this chapter, we focus on the DataCell language interface. We propose a semi-procedural language as a small extension of SQL, that can be used to access both streaming and database data at the same time. DataCell provides an orthogonal extension to SQL'03, called *basket expressions*, which behave as predicate windows over multiple streams and which can be bulk processed for good resource utilization. The functionality offered by basket expressions is illustrated with numerous examples to model complex event processing applications.

*The material in this chapter has been the basis for the EDA-PS paper "A Query Language for a Data Refinery Cell" (Kersten et al., 2007).

4.1.1 Contributions

The main contributions and topics addressed in this chapter are as follows.

- *Predicate windows.* DataCell generalizes the (sliding) window approach, predominant in DSMSs, to allow for arbitrary table expressions over streams. It enables applications to selectively process the stream and prioritize event processing based on application semantics.
- *SQL compliance.* The language extensions proposed are orthogonal to existing SQL semantics. We do not resort to redefinition of the WINDOW concept, nor do we a priory assume a sequence data type. Moreover, the complete state of the system can at any time be inspected using SQL queries.

The stream behavior in DataCell is obtained using a small and orthogonal extension to the SQL language. As we discussed in the previous chapter, streams are presented as ordinary temporary tables, called *baskets* which are the target for (external) sources to deposit events. Baskets carry little overhead as it comes to transaction management. Their content disappears when the system is shut down.

Subsequently, SQL table expressions can be marked as *basket expressions*, which extract portions of interest from stream baskets or ordinary tables. It creates a tuple flow between queries, independent of the implementation technique of the underlying query execution engine.

The benefit of the two language concepts is a natural integration of streaming semantics in a complete SQL framework. It does not require overloading the definition of existing language concepts, nor a focus on a subset of SQL'92. Moreover, its integration with a complete SQL software stack from the outset leverage our development investments.

The validity of our approach is illustrated using concepts and challenges from the "pure" DSMS arena where light-weight stream processing is a starting point for system design. An exhaustive list of examples provides the foundation for comparison against the DataCell approach.

4.1.2 Outline

The remainder of this chapter is organized as follows. In Section 4.2 we introduce the SQL enrichment in more detail. Section 4.3 explores the scope of the solution by modeling stream-based application concepts borrowed from dedicated stream database systems. Finally, Section 4.4 concludes the chapter.

4.2 DataCell Model

In this section we define the DataCell language components, i.e., *baskets*, *receptors* and *emitters*, *basket expressions*, and *continuous queries*, through its language interface. All components are modeled with the SQL'03 language (Eisenberg et al., 2004) with a novel extension, the basket expression, which will also be described in this section. Together they capture and generalize the essence of data stream applications.

4.2.1 Baskets

As we described in the previous chapter (see Section 3.2.2) the *basket* is the key data structure of DataCell, that holds a *portion* of a stream. It is represented as a temporary main-memory table. Incoming events are just appended, and tuples are removed from the basket when "consumed" by a query. The commonalities between baskets and relational tables are much more important to warrant a redesign from scratch. Therefore, their syntax and semantics are aligned with the table definition in SQL'03.

Example 1. The basket definition below models an ordered sequence of events. The *id* takes its value from a sequence generator upon insertion, a standard feature in most relational systems nowadays. It denotes the event arrival order. The default expression for the *tag*, ensures that the event is also timestamped upon arrival. The *payload* value is received from an external source.

```
CREATE BASKET X(
    tag timestamp default now(),
    id serial,
    payload integer
);
```

Important differences between a basket and a relational table are their processing state, their update semantics and their transaction behavior. The processing state of a basket X is controlled with the statements ENABLE X and DISABLE X. The default is to enable the basket to enqueue and dequeue tuples. By disabling it, queries that attempt to update its content become blocked. Selectively (dis)enabling baskets can be used to debug a complex stream application.

A distinctive feature of a basket is its handling of integrity violations. Events that violate the constraints are silently dropped. They are not distinguishable

from those that have never arrived in the first place.

Furthermore, the events do not appear in the transaction log and updates can not be "rolled-back". Baskets are subject to a rigid concurrency scheme. Access is strictly serialized between receiver/emitter and continuous queries. It all leads to a light-weight database infrastructure.

The high-volume insertion rate and the short life of an event in the system make the traditional transaction management a no-go area. With baskets as the central concept we purposely step away from the de-facto semantics of processing events in arrival order in most streaming systems. We consider arrival order a semantic issue, which may be easy to implement on streams directly, but also raises problems with out-of-sequence arrivals (Abadi et al., 2005) and unnecessary complicates applications where the arrival order is not relevant.

4.2.2 Receptors and Emitters

As we have already defined in the previous chapter, the periphery of DataCell consists of *receptors* and *emitters*. These separate processes connect DataCell with the outside world. A receptor picks up streaming events from a communication channel and forwards them to the kernel for processing. Likewise, an emitter picks up the events that constitute the answer of the continuous queries and delivers them to clients who have subscribed to the query results.

Receptors and emitters are woven into the SQL language framework as a variant of the SQL COPY statement. The communication protocol is encoded in the string literal which is interpreted internally. Currently, the supported protocols are TCP-IP and UDP channels.

Example 2. The statements below collect events from the designated IP address and deliver them to another. It is the smallest DataCell program to illustrate streaming behavior.

```
COPY INTO X(payload)
FROM 'localhost:50032';

COPY FROM X(tag,payload)
INTO 'localhost:50033'
delimiters ',','\n';
```

4.2.3 Basket Expressions

The *basket expressions* are the novel building blocks for DataCell queries. They encompass the traditional SELECT-FROM-WHERE-GROUP BY SQL language frame-

work. A basket expression is syntactically a table expression surrounded by square brackets. However, the semantics are quite different. Basket expressions have side-effects; they change the underlying baskets during query evaluation. All tuples that qualify the basket (sub-)expression are removed from the underlying store immediately after they have been processed. This may leave a partially emptied basket behind. Note that the baskets expressions exclusively express the processing requirement of a single query at the query language level. In case where multiple queries require access of the same basket, it is the obligation of the processing engine to guarantee correctness and completeness of continuous stream of answers. For example, by following the *Separate Baskets* processing model 3.3.2 we provide source independence among concurrent continuous queries. Recall that in this scheme we provide an individual basket for each continuous query, thus each one is free to modify its input based on its own needs. Note, a basket can also be inspected outside a basket expression. Then, it behaves as an append-only relational table, i.e., tuples are not removed as a side-effect of the evaluation.

Example 3. The basket expression in the query below takes precedence and extracts all tuples from basket X. All tuples selected are *immediately* removed from basket X (i.e., the basket is emptied), but they remain accessible through B during query execution. From this temporary table we select the payloads satisfying the predicate.

```
SELECT count(*)
FROM [SELECT *
      FROM X
      ORDER BY id ] as B
WHERE B.payload >100;
```

The basket expressions initiate tuple transport in the context of the query. The net effect is a stream within the query engine. X is either a basket or a table. Tuples are removes only in the case that X is a basket. Otherwise, the tuples in the base table remain intact. In MonetDB, deletion from tables is much more expensive, because it involves a transaction commit. This involves moving the tuples deleted to a persistent transaction log. Baskets avoid this overhead, no transaction log is maintained.

4.2.4 Continuous Queries

Continuous queries are long-standing queries that we should continuously evaluate while new incoming stream data arrives. Conceptually, the query is re-executed whenever the database state changes. Two cases should be distinguished. For a non-streaming database, the result presented to the user is an updated result set and it is the task of the query processor to avoid running the complete query from scratch over and over again. For a streaming database, repetitive execution produces a stream of results. The results only reflect the latest state and any persistent state variable should be explicitly encoded, e.g., using stream aggregates and singleton baskets.

In DataCell we consider every query that refers to at least one stream basket in the FROM clause, as a continuous query.

Example 4. A snippet of a console session is shown below. The continuous query can be stopped and restarted by controlling the underlying basket state.

```
CREATE BASKET MyFavored as
 [SELECT *
  FROM X
  WHERE payload>100];

enable MyFavored;

[SELECT * FROM MyFavored];

-- part of the result set
[ 135, 2007-03-27:22:45, 123]
[ 136, 2007-03-27:22:46, 651]
[ 137, 2007-03-27:22:49, 133]
```

4.2.5 Application Modeling

The graphical user interface closely matches the network view of the flow dependencies amongst the baskets, (continuous) queries, tables, and the interface (Liarou et al., 2012b). Compared to similar tools, e.g., Borealis (Abadi et al., 2005), the coarse grain approach of SQL as a specification vehicle pays off.

Example 5. In the previous chapter, when introducing the basic DataCell components (see Section 3.2), we showed how they interact and synthesize a simple

query scenario. In our basic example (Figure 3.1) a receptor R appends the new incoming data to a basket B_1. When new data appears, a submitted continuous query Q obtains access to the incoming stream and the data in the persistent table T, and it is evaluated. The produced results are placed in basket B_2, from where the emitter can finally collect them and deliver them to the client. The SQL-like syntax for this example is as follows.

```
--An Alarm Application
CREATE BASKET B1(
   tag timestamp default now(),
   pl integer);

COPY INTO B1 FROM 'alarms:60000';

CREATE BASKET B2(
   tag timestamp,
   pl integer,
   msg string);

COPY FROM B2 INTO 'console';

CREATE TABLE T1(
   pmin integer,
   pmax integer);

INSERT INTO B2
SELECT tag, pl, "Warning"
FROM T1, [SELECT * FROM C1 WHERE pl > 0] as A,
WHERE A.pl < T1.pmin or A.pl > T1.pmax;
```

4.3 Querying Streams

In this section, we illustrate how the key features of a query language for data streams are handled in the DataCell model using StreamSQL (StreamSQL, 2009), as a frame of reference. Its design is based on experiences gained in the Aurora (Balakrishnan et al., 2004) and the CQL (DBL,) in the STREAM (Arasu et al., 2003; Babcock et al., 2004) projects. It also reflects an expe-

rience based approach, where the language design evolved based on concrete applications.

4.3.1 Filter and Map

The key operations for a streaming application are the *filter* and the *map* operations. The *filter* operator inspects individual tuples in a stream removing the ones that satisfy the filter. The *map* operator takes an event and constructs a new one using built-in operators and calls to linked-in functions. Both operators directly map to the basket expression. There are no up-front limitations with respect to functionality, e.g., predicates over individual events or lack of access to global tables. A simple stream filter is shown below. It selects outlier values within batches of precisely 20 events in temporal order and keeps them in a separate basket.

```
INSERT INTO outliers
  SELECT b.tag, b.payload
  FROM [SELECT top 20
        FROM X
        ORDER BY tag] as b
  WHERE b.payload >100;
```

The TOP clause is equivalent to the SQL LIMIT clause and requires the result set of the sub-query to hold a precisely defined number of tuples. In combination with the ORDER BY clause applied to the complete basket before the TOP is applied simulates a fixed-sized sliding window over streams.

4.3.2 Split and Merge

Stream splitting enables tuple routing in the query engine. It is heavily used to support a large number of continuous queries by factoring out common parts. Likewise, stream merging, which can be a *join* or *gather*, is used to merge different results from a large number of common queries. Both were challenges for the DataCell design. The first one due to the fact that standard SQL lacks a syntactic construct to spread the result over multiple targets. The second one due to the semantic problem found in all stream systems, i.e., at any time only a portion of the infinite stream is available. This complicates a straight forward mapping of the relational join, because an infinite memory is required.

The SQL'99 WITH construct comes closer to what we need for a split operation. It defines a temporary table (or view) constructed as a prelude for query

execution. Extending its semantics to permit a compound SQL statement block gives us the means to selectively split a basket, including replication. It is an orthogonal extension to the language semantics. The statement below partially replicates a basket X into two baskets Y and Z. The WITH compound block is executed for each basket binding A.

```
WITH A AS [SELECT * FROM X]
BEGIN
  INSERT INTO Y
    SELECT * FROM A WHERE A.payload > 100;
  INSERT INTO Z
    SELECT * FROM A WHERE A.payload <= 200;
END;
```

The way out to resolve the merge operation over streams is by window-based joins. They give a limited view over the stream and any tuple outside the window can be discarded from further consideration. The boundary conditions are reflected in the join algorithm. For example, the *gather* operator needs both streams to have a uniquely identifying key to glue together tuples from different streams.

In DataCell, we elegantly circumvent the problem using the basket expression semantics and the computational power of SQL. The DataCell immediately removes tuples that contribute to a basket predicate, i.e., if the predicate is satisfied, it becomes true. In particular, the DataCell removes matching tuples used in a merge predicate. This way, merging operations over streams with uniquely tagged events are straight-forward. Delayed arrivals are also supported. Non-matched tuples remain stored in the baskets until a matching tuple arrives, or a garbage collection query takes control.

Below we see a join between two baskets X and Y with a monotone increasing unique *id* sequence as the target of the join. The join basket expression produces all matching pairs. The residue in each basket are tuples that do not (yet) match. These can be removed with a controlling continuous query, e.g., using a time-out predicate. Taken together they model the *gather* semantics.

```
SELECT A.*
FROM [SELECT * FROM X,Y WHERE X.id=Y.id] as A;
INSERT INTO trash [SELECT ALL
                   FROM X
                   WHERE X.tag < now()-1 hour];
INSERT INTO trash [SELECT ALL
                   FROM Y
                   WHERE Y.tag < now()-1 hour];
```

4.3.3 Aggregation

The initial strong focus on aggregation networks has made stream aggregations a core language requirement. In combination with the implicit serial nature of event streams, most systems have taken the route to explore a sliding window approach to ease their expressiveness.

In DataCell, we have opted not to tie the concepts that strongly. Instead, an aggregate function is simply a two phase processing structure: *aggregate initialization* followed by *incremental updates*.

The prototypical example is to calculate a running average over a single basket. Keeping track of the average payload calls for creation of two global variables and a continuous query to update them. Using batch processing the DataCell can handle such cases as shown in the following example. In this case, updates only take place after every 10 tuples.

```
DECLARE cnt integer;
DECLARE tot integer;
SET tot =0;
SET cnt=0;
WITH Z AS [SELECT top 10 payload FROM X]
BEGIN
  SET cnt = cnt +(select count(*) from Z);
  SET tot = tot +(select sum(*) from Z);
END;
```

4.3.4 Metronome and Heartbeat

Basket expressions can not directly be used to react to the lack of events in a basket. This is a general problem encountered in data stream management systems. A solution is to inject marker events using a separate process, called

a *metronome* function. Its argument is a time interval and it injects a value timestamp into a basket.

The metronome can readily be defined in an SQL engine that supports Persistent Stored Modules and provides access to linked in libraries. This way, we are not limited to time-based activation, but we can program any decision function to inject the stream markers. The example below injects a marker tuple every hour.

```
CREATE FUNCTION metronome (t interval)
  RETURNS timestamp;
BEGIN
    CALL sleep(t);
    RETURN now();
END;
INSERT INTO into X(tag,id,payload)
  [SELECT null,metronome(1 hour),null];
```

Furthermore, its functionality can be used to support another requirement from the stream world, the *heartbeat*. This component ensures a uniform stream of events, e.g., missing elements are replaced by a dummy if nothing happened in the last period. At regular intervals the heartbeat injects a null-valued tuple to mark the *epoch*. If necessary, it emits more tuples to ensure that all epochs seen downstream before the next event are handled.

The heartbeat functionality can be illustrated using a join between two baskets. The first one models the heartbeat and the second one the events received. This operation is in-expensive in a column-store. We assume that the heartbeat basket contains enough elements to fill any gap that might occur. Its clock runs ahead of those attached to the events. In this case, we can pick all relevant events from the heartbeat basket and produce a sorted list for further processing.

The heartbeat functionality can be modeled using the metronomes and the basket expressions as follows.

```
INSERT INTO HB [SELECT null, T, null
   FROM [select metronome(1 second)]];
[SELECT * FROM X
 UNION
 SELCT * FROM HB
 WHERE X.tag < max(SELECT tag FROM HB)];
```

4.3.5 Basket Nesting

A query may be composed of multiple and nested basket expressions. The Petri-net interpretation creates intermediate results as soon as a basket becomes non-empty. Each incurs an immediate side-effect of tuples movement from its source to a temporary table in the context of the query execution plan. Yet, a compound query is only executed when all basket sub-expressions have produced a result. Consequently the query result depends on their evaluation order. However, since at any point in time the database seen is complete snapshot, it is up to the programmer to resolve evaluation order dependencies using additional predicates.

A design complication arises when two continuous queries use basket expressions over the same basket and if they are interested in the same events. Then we have a potential conflict. These events will be assigned randomly to either query. If both need access to the same event, it is mandatory to split the basket and replicate the events to a private basket first.

4.3.6 Bounded Baskets

The arrival rate of stream events may surpass the capabilities of queries to handle them in time before the next one arrives. In that case, the baskets grows with a backlog of events. To tackle this problem, StreamSQL provides a mechanism to identify "slack", i.e., the number of tuples that may be waiting in the basket. The remainder is silently dropped.

Although this problem is less urgent in the bulk processing scheme of MonetDB, it might still be wise to control the maximum number of pending events in bursty environments. Of course, the semantics needed strongly depend on the application at hand. Some may benefit from a random sampling approach, others may wish to drown old events. Therefore, a hardwired solution should be avoided.

Example 6. The query below illustrates a scheme to drop old events. Although this does not close the gap completely, the basket can be evaluated in microseconds.

```
SELECT count(B.*), ' dropped'
FROM [SELECT  *
      FROM X
      WHERE id < max(SELECT id FROM X)-100)] as B;
```

4.3.7 Stream Partitioning

Stream engines use a simple value-based partitioning scheme to increase the parallelism and to group events. A partitioning generates as many copies of the down-stream plans as there are values in the partitioning column. This approach only makes sense if the number of values is limited. It is also not necessary in a system that can handle groups efficiently.

In the context of MonetDB, value-based partitioning is considered a tactical decision taking automatically by the optimizers. A similar route is foreseen in handling partitions over streams to increase parallelism. Partitioning to group events of interest still relies on the standard SQL semantics.

Example 7. A continuous query that returns a sorted list by traffic per minute become:

```
SELECT Z.tag, Z.cnt
FROM [SELECT minute(tag) as tag,
            count(*) as cnt
      FROM X
      GROUP BY tag] as Z
ORDER BY Z.tag;
```

4.3.8 Transaction Management

Transaction semantics in the context of volatile events and persistent tables is an open research area. For some applications non-serializable results should be avoided and traditional transaction primitives may be required. In StreamSQL this feature is cast in a *lock* and *unlock* primitive. It makes transaction control visible at the application level with crude blocking operators.

The approach taken in the DataCell is to rely on the (optimistic) concurrency control scheme and transaction logger as much as possible. All continuous queries have equal precedence and their actual execution order is explicitly left undefined. If necessary, it should be encoded in a control basket or explicit dependencies amongst queries.

4.3.9 Sliding Windows

Most DSMSs define query processing around streams seen as a linear ordered list. This naturally leads to sequence operators, such as NEXT, FOLLOWS, and WINDOW expressions. The latter extends the semantics of the SQL WINDOW

construct to designate a portion of interest around each tuple in the stream. The WINDOW operator is applied to the result of a query and, combined with the iterator semantics of SQL, mimics a kind of basket expression.

However, re-using SQL window semantics introduce several problems. To name a few, they are limited to expressions that aggregate only, they carry specific first/last window behavior, they are read-only queries, they rely on predicate evaluation strictly before or after the window is fixed, etc. In StreamSQL the window can be defined as a fixed sized stream fragment, a time-bounded stream fragment, or a value-bound stream fragment only.

The basket expressions provide a much richer ground to designate windows of interest. They can be bound using a sequence constraint, they can be explicitly defined by predicates over their content, and they can be based on predicates referring to objects elsewhere in the database.

Example 8. A sliding window of precisely 10 elements and a shift of two is encapsulated in the query below. A time bounded window simply requires a predicate to inspect the clock.

```
SELECT * FROM [SELECT * FROM X limit 2]
UNION
SELECT * FROM X limit 8;

--create window Xw (size 10 seconds
--                  advance 2 seconds);
SELECT *
FROM [SELECT *
      FROM X
      WHERE tag < min(SELECT X.tag) + 2 seconds]
UNION
SELECT *
FROM X
WHERE tag < min(SELECT X.tag) + 8 seconds;
```

The generality of the basket expressions come at a price. Optimization of sequence queries may be harder if the language or scheme does not provide hooks on this property. However, we still allow window functions to be used over the baskets. Their semantics is identical to applying them to an SQL table.

4.4 Summary

In this chapter, we presented the DataCell language interface. A small extension of the relational algebra engine of MonetDB is sufficient to produce a fully functional prototype DataCell implementation. The *basket expressions*, blended into the syntax and semantics of SQL 2003, provide an elegant solution to define stream-based applications. The language concepts introduced are compared against building blocks found in "pure" stream management systems. They can all be expressed in a concise way and demonstrate the power of starting the design from a full-fledged SQL implementation.

The proposed language interface is an alternative suggestion to the existing SQL-like languages for data streams, e.g., (DBL, ; StreamSQL, 2009). Basket expressions are proposed as a general way to express predicate windows over multiple streams. However, the extensible nature of MonetDB/DataCell architecture allows the complete language disconnection from the underlying engine if it is necessary. This means that with the appropriate changes in the external part of the MonetDB/DataCell software stack, i.e., in the parser and in part of the optimizer rules, we can easily set and implement a different language interface.

In the following chapter we study one of the most crucial pure stream processing problems, i.e., incremental processing for window-based continuous queries. Even with the conventional underlying infrastructure that MonetDB offers to DataCell, we manage to compete against a specialized stream engine, elevating incremental processing at the query plan level, instead of building specialized stream operators. Then, Chapter 6 concludes the thesis and discusses a number of interesting open topics and research directions towards a complete data management architecture that integrates database and stream functionalities in the same kernel.

Chapter 5

Incremental Processing in DataCell*

5.1 Introduction

In the two previous chapters, we described the basic DataCell architecture and the SQL-extended query language interface that allow us to formulate and submit continuous queries, encompassing streams and tables. However, numerous research and technical questions are still waiting for answers and solutions in the DataCell context. The most prominent issues are the ability to provide specialized stream functionality and hindrances to guarantee real-time constraints for event handling. Chapter 3 illustrates the DataCell architecture but leaves open issues related to real-time stream processing. Here, we make the next step towards a fully functional streaming DBMS kernel; we study how we can deal with incremental processing while staying faithful at the DataCell philosophy that dictates minimal changes to the underlying kernel.

*The material in this chapter has been the basis for a paper submitted for publication entitled "Enhanced Stream Processing in a DBMS Kernel" (Liarou et al., 2012a) and at the PVLDB12 paper "MonetDB/DataCell: Online Analytics in a Streaming Column-Store" (Liarou et al., 2012b).

5.1.1 Contributions

In this chapter, we focus on the core of streaming applications, i.e., incremental stream processing and window-based processing. Window queries form the prime programming paradigm in data streams, i.e., we break an initially unbounded stream into pieces and continuously produce results using a *focus window* as a peephole on the data content passing by. Successively considered windows may overlap significantly as the focus window *slides* over the stream. It is the cornerstone in the design of specialized stream engines and typically specialized operators are designed to avoid work when part of the data falls outside the focus window.

Most relational operators underlying traditional DBMSs *cannot* operate incrementally without a major overhaul of their implementation. Here, we show that efficient incremental stream processing is, however, possible in a DBMS kernel handling the problem *at the query plan and scheduling level*. For this to be realized the relational query plans are transformed in such a way that the stream is broken into pieces and different portions of the plan are assigned to different portions of the focus window data. DataCell takes care that this "partitioning" happens in such a way that we can exploit past computation during future windows. As the window slides, the stream data also "slides" within the continuous query plan.

In this chapter, we illustrate the methods to extend the MonetDB/DataCell optimizer with the ability to create and rewrite them into incremental plans. A detailed experimental analysis demonstrates that DataCell supports efficient incremental processing, comparable to a specialized stream engine or even better in terms of scalability.

5.1.2 Outline

The rest of this chapter is organized as follows. Firstly, Section 5.2 presents a short recap of the notion of window-based stream processing. Then, Sections 5.3 and 5.4 discuss in detail how we achieve efficient incremental processing in DataCell. Section 5.6 provides a detailed experimental analysis. We compare the incrementalist DataCell kernel with our basic architecture (discussed in Chapter 3) and a specialized state-of-the art commercial stream engine. Finally, Section 6 concludes the chapter.

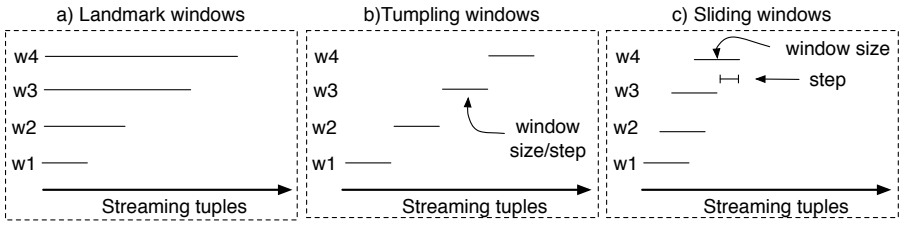

Figure 5.1: Window-based stream processing

5.2 Window-based Processing

Continuous computation of long standing queries in large scale streaming environments is a huge challenge from a data management perspective. Continuously considering all past data is not a scalable solution. Especially when it comes to blocking operators, e.g., a join, it is unrealistic to continuously analyze all data purely from a system resources point of view. This way, window-based queries have been introduced to assist efficient query processing in streaming environments. By windowing a continuous query, we delimit the boundaries of the initially unbounded stream and we continuously produce results on different portions of the data. Figure 5.1 shows simple examples of how window-based processing differs from "complete" stream processing.

Figure 5.1(a) shows the typical unbounded stream processing. This is often referred to as *landmark* window in the literature, i.e., the processing window is continuously growing. Figures 5.1(b) and (c) on the other hand, show window processing where as new data tuples arrive, some of the old ones *expire*. This way, a limited window of tuples is defined and the system is called to produce answers only for the tuples within the current window, ignoring the larger volume of past data preceding this window. The most straight-forward type are *tumbling* windows (cf., Fig. 5.1(b)). Here, the size of the *step*, i.e., the number of tuples we move the window forward, is equal to the window size. This leads to non overlapping windows of tuples, i.e., every tuple is considered (at most) once for a given query.

Other than making query processing possible by limiting the amount of processed data, window-based processing also raises a number of challenges. Especially, *sliding window* queries, i.e., queries where the step is smaller than the window such that subsequent windows *overlap*, lead to very interesting scenarios and processing challenges. Figure 5.1(c) shows an example of sliding

Algorithm 2 The factory for continuous re-evaluation of a tumbling window query that selects all values of attribute X in range v_1-v_2.

```
1: input = basket.bind(X)
2: output = basket.bind(Y)
3: while true do
4:    while input.size < windowsize do
5:       suspend()
6:    basket.lock(input)
7:    basket.lock(output)
8:    w = basket.getLatest(input,windowsize)

9:    result = algebra.select(w,v1,v2)

10:   basket.delete(input,windowsize)
11:   basket.append(output,result)

12:   basket.unlock(input)
13:   basket.unlock(output)
14:   suspend()
```

windows. The ideal goal is that every time we need to recompute the result of a query over the current window, we would like to analyze as little data as possible by cleverly exploiting past computation actions over previous windows that overlap with the current one.

In other words, the result of each window should be *incrementally* computed, by reusing valid past results. This incremental behavior is fundamental in all stream algorithms, techniques and systems. In addition, it is a functionality that is missing from a typical DBMS. Thus, it becomes a unique problem for the DataCell context as well.

5.3 Continuous Re-evaluation

Complete re-evaluation is the straight-forward approach when it comes to continuous queries for a DBMS engine. The idea is simple; every time a window is complete, i.e., enough tuples have arrived, we compute the result over all tuples in the window. In fact, this is the way that any DBMS can support continuous query processing modulo the addition of certain scheduling and triggering

mechanisms.

We can achieve this kind of processing by applying minimal changes to the existing DataCell architecture. Assuming, for the time being, single stream queries and tumbling windows, we only need to make sure that a query plan will "consume" $|W|$ tuples of the input stream at a time, where $|W|$ is the size of the window. This means that $|W|$ tuples will be considered for query processing and subsequently $|W|$ tuples are dropped from the input basket while of course the query plan will not run unless at least $|W|$ tuples are present.

All this boils down to a set of simple rewriting rules for the continuous query plans of DataCell. For example, Algorithm 2 shows such a continuous re-evaluation query plan, for a simple window range query. The window semantics affect the plan only in such a way that it checks whether there are enough input tuples to fill a complete window (lines 4 and 5 in Algorithm 2). In addition, it only considers and subsequently drops $|W|$ tuples at a time (lines 8 and 10 respectively in Algorithm 2).

To support also sliding overlapping windows with a step size of $|w| < |W|$ tuples, only one more minor change is required, refining line 10 in Algorithm 2 as follows. Instead of deleting the complete window we would only delete the oldest $|w|$ tuples that expire per step, namely the sliding step that encompass those tuples that are not valid in the next window.

This way, re-evaluation is quite simple to achieve in DataCell and as before the core of the query plan can be any kind of complex query, allowing DataCell to support the full strength of SQL and the complete optimizer module.

5.4 Incremental Processing

Although the direction seen in the previous section is sufficient for *tumbling* and *hopping* windows, i.e., windows that slide per one or more than a full window size at a time, it is far from optimal when it comes to the more common and challenging case of *overlapping sliding* windows. The drawback is that we continuously process the same data over and over again, i.e., a given stream tuple t will be considered by the same query multiple times until the window slides enough for t to expire. For this, we need efficient incremental processing, a feature missing from typical DBMSs. Here, we discuss how we address this fundamental stream problem in DataCell.

5.4.1 The Goal

For ease of presentation, we begin with a high-level description of the technique at large, before we continue to discuss in more detail the various decisions and options.

The vision is to create a full-fledged stream engine without sacrificing any of the existing DBMS technology benefits. Our effort for incremental processing here successfully follows this path; without creating new specialized operators, we support sliding window queries by carefully rewriting and scheduling the existing DBMS query plans. This way, we can exploit all sophisticated query optimization techniques of a modern DBMS and all highly optimized operator implementations as well as query plan layouts.

5.4.2 Splitting Streams

Conceptually, DataCell achieves incremental processing by partitioning a window into n smaller parts, called *basic windows*. Each basic window is of equal size to the sliding step of the window and is processed separately. The resulting partial results are then *merged* to yield the complete window result.

Assume a window $W_i = w_1, w_2, \ldots, w_n$ split into n basic windows. After processing W_i, all windows after that can exploit past results. For example, for window $W_{i+1} = w_2, w_3, \ldots, w_{n+1}$ only the last basic window w_{n+1} contains new tuples and needs to be processed, merging its result with the past partial results. This process continues as the window slides. Effectively, for each new window we only need to process the new tuples as opposed to the naive re-evaluation method that needs to process all window tuples repeatedly.

5.4.3 Operator-level vs Plan-level Incremental Processing

The basic strategy described above is generally considered as the standard backbone idea in any effort to achieve incremental stream processing. It has been heavily adopted by researchers and has lead to the design of numerous specialized stream operators such as window stream joins and window stream aggregates, e.g., (Dobra et al., 2002; Ghanem et al., 2007; Golab, 2006; Kang et al., 2003; Zhu and Shasha, 2002; Li et al., 2005).

Stream engines provide radically different architectures than a DBMS by pushing the incremental logic all the way down to the operators. Here, in the context of DataCell we design and develop the incremental logic at the query plan level, leaving the lower level intact and thus being able to reuse the complete

storage and execution engine of a DBMS kernel. The motivation is to inherit all the good properties of the DBMS regarding scalability and robustness in heavy workloads as demanded by nowadays stream applications.

The questions to answer then are:

(1) How can we achieve this in a generic and automatic way?

(2) How does it compare against state-of-the-art stream systems?

In this section, we will describe our design and implementation in DataCell, where we extend the optimizer to transform normal continuous query plans into incremental ones, which a scheduler is responsible to trigger. In the next section, we will show the advantages of this approach over specialized stream engines as well as the possibilities to combine those two extremes.

5.4.4 Plan Rewriting

The key point is careful and generic query plan rewriting. DataCell takes as input the query plans that the SQL engine creates, leveraging the algebraic query optimization performed by the DBMS's query optimizer. Fully exploiting MonetDB's execution stack, the incremental plan generated by DataCell is handed back to MonetDB's optimizer stack for physical plan optimization.

To rewrite the original query plan into an incremental one, DataCell applies four basic transformations;

(1) Split the input stream into n basic windows

(2) Process each (unprocessed) basic window separately

(3) Merge partial results

(4) Slide to prepare for the next basic window

Figure 5.2 shows this procedure schematically. For the first window, we run part of the original plan for each basic window while intermediates are directed to the remainder of the plan to be merged and execute the rest of the operators. As the window slides we need to process only the new data avoiding to reaccess past basic windows (shown faded in Figure 5.2) and perform the proper merging with past intermediates. Achieving this for generic and complex SQL plans is everything but a trivial task. Thus, we begin with an over-simplified example shown in Algorithm 3 to better describe these concepts.

Splitting

The first time the query plan runs, it will split the first window into n basic windows (line 7). This task is in practice an almost zero cost operation in MonetDB and results in creating a number of views over the base input basket.

Query Processing

The next part is to run the actual query operators over each of the first $n-1$ basic windows (lines 8-11), calculating their partial results. While in general more complicated (as we will see later on), for this simple single-stream, single-operator query the task boils down to simply calling the select operator for each basic window. For more complex queries, we will see that only part of the plan runs on every single basic window, while there is another part of the incremental plan that runs on merged results.

Basic Loop

The plan then enters an infinite loop where it (a) runs the query plan for the last (latest) basic window and (b) merges all partial results to compose the complete window result. The first part (line 18) is equivalent to processing each of the first $n-1$ basic windows as discussed above. For the simple select query of our example, the second part can create the complete result by simply concatenating the n partial results (line 19). We will discuss later how to handle the merge in more complex cases.

Transition Phase

Subsequently, we start the preparation for processing the next window, i.e., for when enough future tuples will have arrived. Basically, this means that we first shift the basic windows forward by one as indicated in line 20 for this example. Then, more importantly we make the correct correlations between the remaining intermediate results, this transition (line 21) is derived by the previous one. In the current example both transitions are aligned, but in the case of more complicated queries (e.g., multi-stream query with join operators), we should carefully proceed this step.

Algorithm 3 The factory for incremental evaluation of a single stream window query that selects all values of attribute X in v_1-v_2.

```
 1: input = basket.bind(X)
 2: output = basket.bind(Y)
 3: while input.size < windowsize do
 4:    suspend()
 5: basket.lock(input)
 6: basket.lock(output)

 7: w_1, w_2, ..., w_n = basket.split(input,n)
 8: res_1 = algebra.select(w_1,v1,v2)
 9: res_2 = algebra.select(w_2,v1,v2)
10: ...
11: res_{n-1} = algebra.select(w_{n-1},v1,v2)

12: while true do
13:    while input.size < windowsize do
14:       suspend()
15:    basket.lock(input)
16:    basket.lock(output)

17:    w_n = basket.getLatest(input,stepsize)
18:    res_n = algebra.select(w_n,v1,v2)

19:    result = algebra.concat(res_1,res_2,...,res_n)

20:    w_exp = w_1, w_1 = w_2, w_2 = w_3, ..., w_{n-1} = w_n
21:    res_1 = res_2, res_2 = res_3, ..., res_{n-1} = res_n

22:    basket.delete(input,w_exp)
23:    basket.append(output,result)

24:    basket.unlock(output)
25:    basket.unlock(input)
26:    suspend()
```

Intermediates Maintenance

Maintaining and reusing the proper intermediates is of key importance. In our simple example, the intermediates we maintain are the results of each select

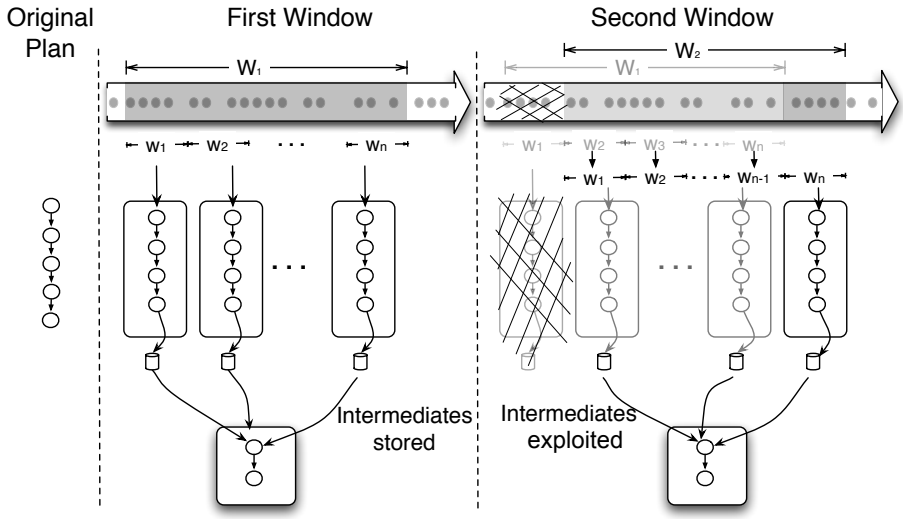

Figure 5.2: Incremental processing at the query plan level

operator which are to be reused in the next window as well. In general, a query plan may have hundreds even thousands of operators. The DataCell plan rewriter maintains the proper intermediates by following the path of operators starting from each basic window to associate the proper intermediates with the proper basic window such as to know (a) how to reuse an intermediate and (b) when to expire it. This becomes a big challenge especially in multi-stream queries where an intermediate from one stream may be combined with multiple intermediates from other streams, e.g., for join processing (we will see more complex examples later on).

Continuous Processing

The next step is to discard the old tuples that expire (line 22) and deliver the result to the output stream (line 23). After that, the plan *pauses* (line 26) and will be resumed by the scheduler only when new tuples have arrived. Lines 13-14 ensure that the plan then runs only once there are enough new tuples to fill a complete basic window.

Discarding Input

In simple cases, as in the given example, once the intermediate results of the individual basic windows are created, the original input tuples are no longer required. Hence, to reduce storage requirements we can discard all processed tuples from the input basket, even if they are not yet expired, keeping only the respective intermediate results for further processing. Extending Algorithm 3 for achieving this is straightforward. A caveat seen shortly is that there are cases, e.g., multi-stream matching operations like joins, where we cannot apply this optimization, as we need access the original input data until it expires.

5.4.5 Generic Plan Rewriting

When considering more complex queries and supporting the full power of SQL, the above plan rewriting goals are far from simple to achieve. How and when we split the input, how and when we merge partial results are delicate issues that depend on numerous parameters related to both the operator semantics for a given query plan and the input data distribution.

In this way, our strategy of rewriting query plans becomes as follows. The DataCell plan rewriter takes as input the optimized query plan from the DB optimizer.

(1) The first step remains intact; it splits the input stream into $n = |W|/|w|$ disjoint pieces.

(2) In a greedy manner, it then consumes one operator of the target plan at a time. For each operator it decides whether it is sufficient to replicate the operator (once per basic window) or whether more actions need to be taken.

The goal is to *split the plan as deep as possible*, i.e., allow as much of the original plan operators to operate independently on each basic window. This gives maximum flexibility and eventually performance as it requires less post processing with every new slide of the window, i.e., less effort in merging partial results.

To ease the discussions towards a generic and dynamic plan rewriting strategy, we continue by giving a number of characteristic examples where different handling is needed than the simplistic directions we have seen before. Figures 5.3, 5.4, 5.5, 5.6 and 5.7 will help in the course of this discussion through a

variety of queries. Note, that we show only the pure SQL query expression, cutting out the full language statments of the continuous sliding window queries. For each query, we show the reevaluated continuous query plan as well as the DataCell incremental plan. The solid lines in the incremental query plan indicate the basic loop, i.e., the path that is continuously repeated as more and more tuples arrive. The rest of the incremental plan needs to be executed only the first time this plan runs.

5.4.6 Exploit Column-store Intermediates

As we have already discussed, our design is on top of a column-store architecture. Column-stores exploit vector based bulk processing, i.e., each operator processes a full column at a time to take advantage of vector-based optimizations. The result of each operator is a new column (BAT in MonetDB). In DataCell, we do not release these intermediates once they have been consumed. Instead, we selectively keep intermediates when processing one window to reuse them in future windows. This effectively allows us to put breakpoints in multiple parts of a query plan given that each operator creates a new intermediate. Subsequently, we can "restart" the query plan from this point on simply by loading the respective intermediates and performing the remaining operators given the new data. Which is the proper point to "freeze" a query plan depends on the kind of query at hand. We discuss this in more detail below.

5.4.7 Merging Intermediates

The point where we freeze a query plan practically means that we no longer replicate the plan. At this point we need to merge the intermediates so that we can continue with the rest of the plan. The merging is done using the `concat` operator. Examples of how we use this can be seen in all instances of Figures 5.3 till 5.7. Observe, how before a `concat` operator the plan forks into multiple branches to process each basic window separately, while after the merge it goes back into a single flow. In addition, note that depending on the complexity of the query, there might be more than one flow of intermediates that we need to maintain and subsequently merge. For example, the plans in Figure 5.3, 5.4 and 5.7 have a single flow of intermediates while the plans in Figure 5.5 and 5.6 have two flows.

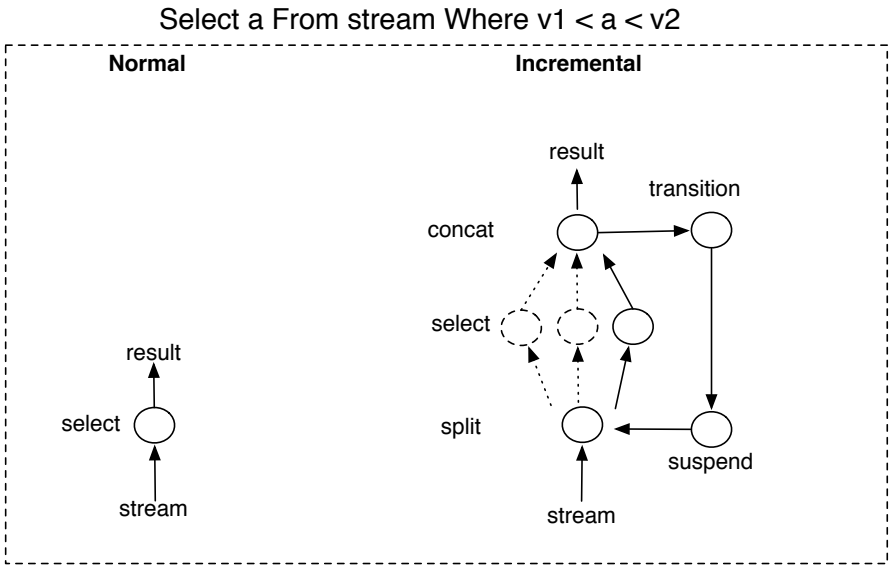

Figure 5.3: Example of query plan transformations for range query

5.4.8 Simple Concatenation

The simplest case are operators where a simple concatenation of the partial results forms the correct complete result. Typical representatives are the select operator as featured in our previous examples, and any map-like operations. In this case, the plan rewriter can simply replicate the operation, apply it to each basic window, and finally concatenate the partial results. Figure 5.3 depicts such an example for a range query.

Every time the window slides, we only have to go through the part of the plan marked with solid lines in Figure 5.3, i.e., perform the selection on the newest basic window and then concatenate the new intermediate with the old ones that are still valid. The transition phase which runs between every two subsequent windows guarantees that all intermediates needed and inputs are shifted by one position as shown in Algorithm 3.

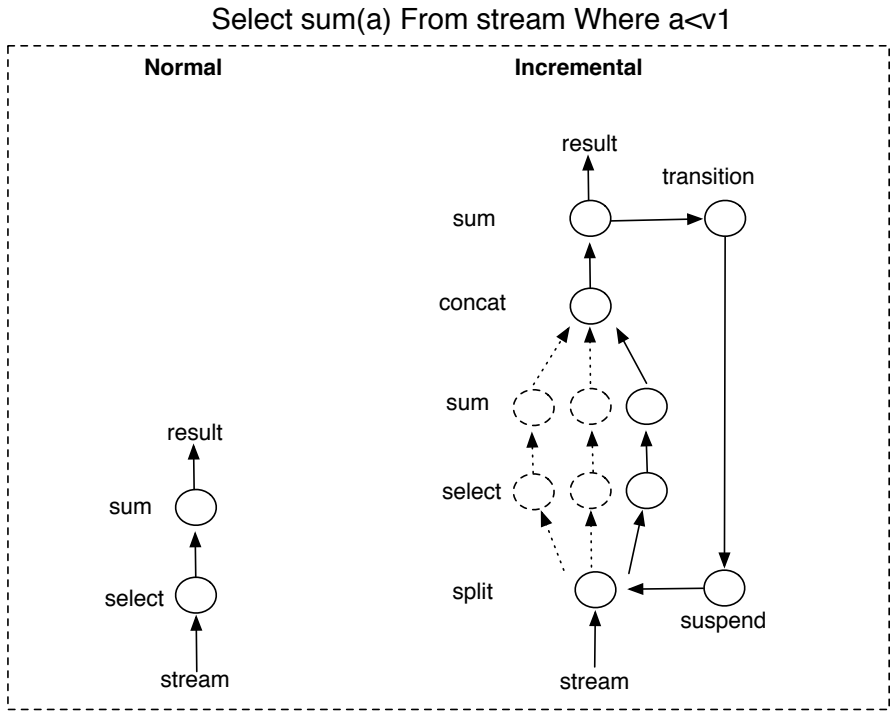

Figure 5.4: Example of query plan transformations for SUM function

5.4.9 Concatenation plus Compensation

The next category consists of operations that can be replicated as-is, but require some compensation after the concatenation of partial results to produce the correct complete result. Typical examples are aggregations like `min`, `max`, `sum`, as well as operators like `groupby/distinct` and `orderby/sort`. For these examples, the compensating action is simply applying the very operation not only on the individual basic windows, but also on the concatenated result as shown for `sum` in Figure 5.4. Other operations might require different compensating actions, though. For instance, a `count` is to be compensated by a `sum` of the partial results.

Note how Figure 5.4 actually combines the `sum` with a selection such that the

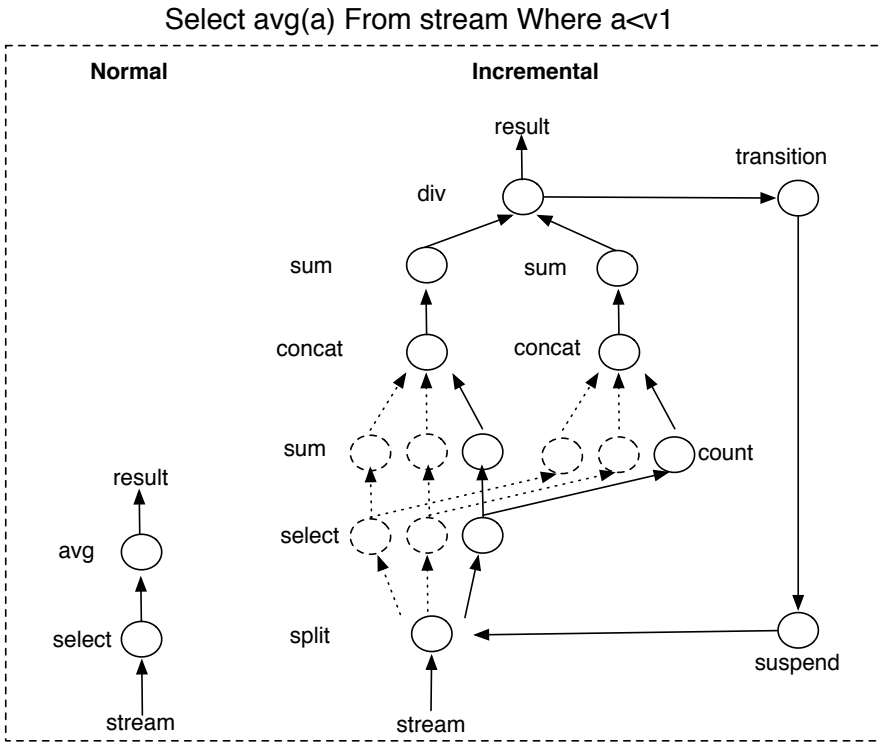

Figure 5.5: Example of query plan transformations for AVG function

selection is performed only on the basic windows, while the `sum`-compensation is required after the concatenation.

5.4.10 Expanding Replication

A third category consists of operations that cannot simply be replicated to the basic windows as-is, but need to be represented by multiple different operations. For instance, Figure 5.5 sketches the incremental calculation of `average`. Instead of simply replicating the `average` operation, we first need to calculate `sum` and `count` separately for each basic window, creating two separate data flows. Then, the global `sum` and `count` after concatenation are derived using

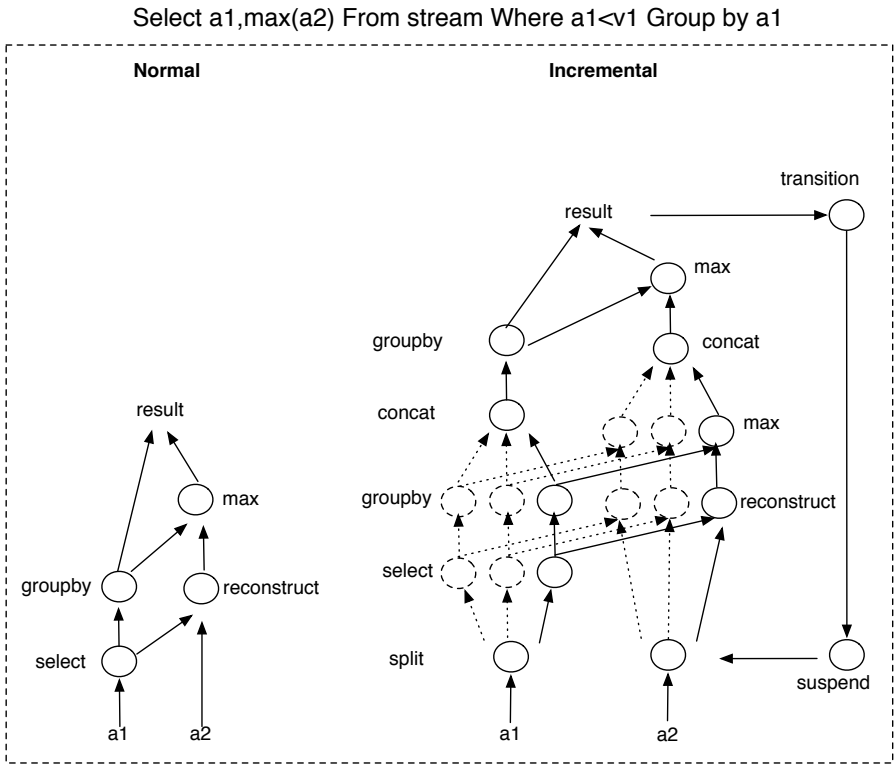

Figure 5.6: Example of query plan transformations for GROUP BY query

the respective compensating actions as introduced above. Finally, dividing the global `sum` by the global `count` merges the two data flows, again, to yield the requested global `average`.

5.4.11 Synchronous Replication

All cases discussed so far consider unary operations, either individually or in linear combinations, involving only a single attribute, and hence a single input data flow with columnar evaluation. Once multiple attributes are involved, we get multiple, possibly interconnected data flows as depicted for a grouped ag-

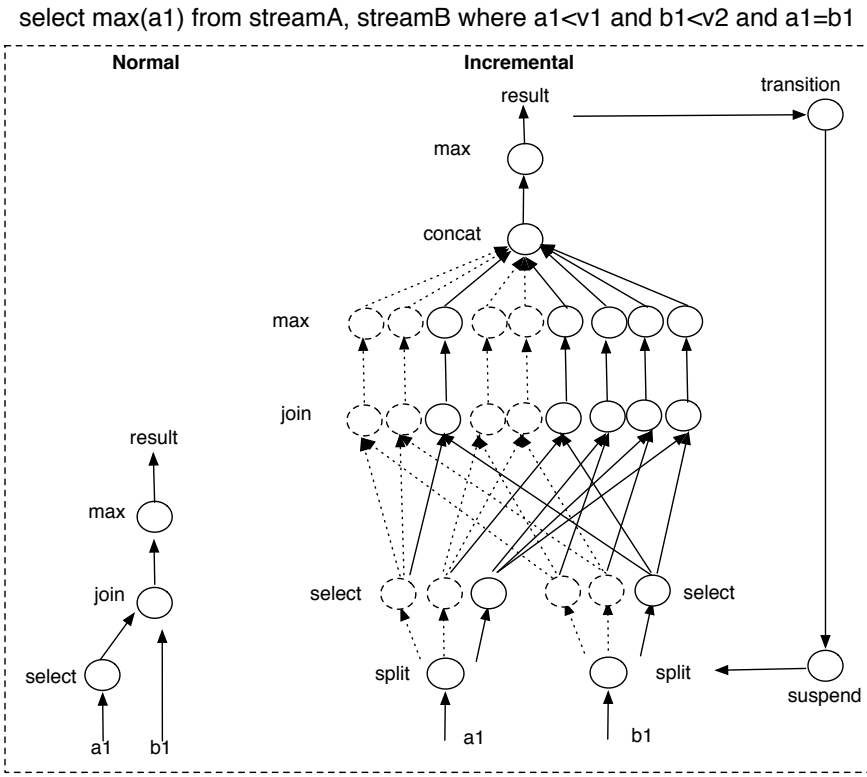

Figure 5.7: Example of query plan transformations for join query

gregation query in Figure 5.6. Canonically applying the rewrite rules discussed above, we can replicate the different data flows synchronously over the basic windows and use the compensating actions to merge the data flows into a single result just as in the original query plan.

5.4.12 Multi-stream Queries

All cases discussed above only consider a single data stream and (from an N-ary relational point of view) unary (i.e., single-input) operations. In these cases, it is sufficient to simply replicate the operations as often as there are basic windows.

For multiple data streams and N-ary operations to combine them, the situation is more complex. Consider, for instance, the case of two streams and a join to match them as depicted in Figure 5.7. For simplicity of presentation we assume that both streams use the same windows size $|W|$ and the same step size $|w|$. Given that we create the $n = |W|/|w|$ basic windows per stream as time slices, i.e., independently of the actual data (e.g., the join attribute values), we need to replicate the join operator n^2 times to join each basic window from the left stream with each basic window from the right stream. As with the other examples, the dashed operator instances in Figure 5.7 need to be evaluated only once during the initial preface. The solid operator instances need to be evaluated repeatedly, once for each step of the sliding window. Note that in this case we cannot discard the selection results once the join has consumed them for the first time. Rather, they need to be kept and joined with newly arriving data until the respective basic windows expire.

5.4.13 Landmark Window Queries

Landmark queries differ from sliding windows queries in that subsequent windows share the same fixed starting point ("landmark"), i.e., tuples do not expire per window step. Tuples either never expire, or at most very infrequently, and then all past tuples expire by resetting the global landmark.

Supporting such queries is straightforward in our design. Since data never expires, we do not have to keep individual intermediate results per basic windows to concatenate the active ones per step. Instead, we need to keep only one cumulative result for each `concat` operation in our DataCell plans in Figures 5.3 till 5.7. In fact, there is not even a need to split the preface in n basic windows. The initial window can be evaluated in one block; only newly arriving data is evaluated once a basic windows is filled as discussed above.

5.4.14 Time-based Sliding Windows

Our approach is generic enough to support both main sliding window types, i.e., count-based and time-based queries. In the first case, the window size and the sliding function are expressed in quantity of tuples, so counting and slicing the input stream is a straightforward process. In the case of time-based queries, the window parameters are defined in terms of time, e.g., query with window size 1 hour that slides per 10 minutes. Once a tuple arrives into the system it is tagged with a timestamp that indicates its arrival time (we could also process the window based on the generation tuple time). The splitting of input

stream now happens taking into account the tuple timestamps. We devide the stream into time intervals, let's say equal to the sliding period. This means that each generated basic windows contains as many tuples as they arrived in the corresponding time interval, so we could end up with unequally filled in basic windows. After that point, DataCell processes the time-based window query following the same methodology we have discussed so far. Empty basic windows are recognized and skip processing.

5.4.15 Optimized Incremental Plans

The decision to split the initial window into $n = |W|/|w|$ basic windows is purely driven by the semantics of sliding window queries. Further performance considerations are not involved. Consequently, the DataCell incremental plans as described so far start processing the next step only once sufficient tuples have arrived to fill a complete basic window. The response time from the arrival of the last tuple to fill the basic window until the result is produced is hence determined by the time to process a complete basic window of $|w|$ tuples (plus merging the partial results of all n active basic windows).

However, since tuples usually arrive in a steady stream, a fraction of the basic window could be processed before the last tuple arrives. This would leave fewer tuples to be processed after the arrival of the last tuple, and could hence shorten the effective response time.

In fact, the above described DataCell approach provides all tools to implement this optimization. The idea is to process the latest basic window incrementally just as we process the whole window incrementally. Instead of waiting for $|w|$ tuples, the basic loop is triggered for every $|v| = |w|/m$ tuples, splitting the basic window in m chunks. The results of the chunks are collected, but no global result is returned, yet. Only once m chunks have been processed, the m chunk results are merged into the basic window result, just like the n basic window results are merged into the window result above. Then, the n basic window results are merged and returned. This way, once the last basic window tuple has arrived, only $|v| = |w|/m$ tuples have to be processed before the result can be returned.

Choosing m and hence $|v|$ is a non-trivial optimization task. $m = |w|$ minimizes $|v|$ and thus the pure data processing after the arrival of the last tuple, but maximizes the overhead of maintaining and merging the chunk results. $m = 1$ is obviously the original case without optimization.

Given that both processing costs and merging overhead depend on numerous hardly predictable parameters, ranging from query characteristics over data

distributions to system status, we consider analytical models with reasonable accuracy hardly feasible. Instead, we propose a dynamic self-adapting solution. Starting with $m = 1$, we successively increase m, monitoring the response time for each m for a couple of sliding steps. It is to be expected that the response times initially decrease with increasing m as less data needs to be processed after the arrival of the last tuple. Only once the increasing merging overhead outweighs the decreasing processing costs, the response times increase, again. Then, we stop increasing m and reset it to the value that resulted in the minimal response time. Next to increasing m linearly or exponentially (e.g., doubling with each step), bisection in the interval $[1, |w|]$ is a viable alternative for finding the best value for m.

5.5 Optimizer Pipeline in DataCell for Incremental Query Plans

In this chapter, we presented the necessary transformation rules needed for the creation of incremental query plans for continuous sliding window queries. In this section, we discuss in more detail the optimization steps we implant in our MonetDB/DataCell experimentation platform for generic plan generation.

Recall the first DataCell implementation (see Section 3.4), where we needed to change the MonetDB optimizer, creating and adding new optimizer rules and defining a new optimizer pipeline as follows.

```
datacell_pipe=inline,remap,evaluate,costModel,coercions,emptySet,
aliases,deadcode,constants,commonTerms,datacell,emptySet,aliases,
deadcode,reduce,garbageCollector,deadcode,history,multiplex
```

There, DataCell receives an *one-time* query plan which is produced by the MonetDB optimizer and it transforms it to a *continuous* query plan that works according to the re-evaluation logic. Now, we need to extend our optimization phase again with a new set of rules in order to support incremental stream processing for sliding window queries. The new optimizer pipeline we configure is the following.

```
datacellInc_pipe=inline,remap,evaluate,costModel,coercions,emptySet,
aliases,deadcode,constants,commonTerms,datacell,emptySet,aliases,
deadcode,datacellSlicer,mergetable,deadcode,datacellIncrementalist,
reduce,garbageCollector,deadcode,history,multiplex
```

Compared to the first `datacell_pipe`, the new pipeline for incremental plans contains three new optimizer rules i.e., `datacellSlicer`, `mergetable` and `datacellIncrementalist`. Their main role is to transform a *continuous* query plan to a *incremental* query plan. The main actions they take are as follows.

- Traverse the plan to find the baskets on which we apply the window predicate.
- For each window, split the input window into n pieces, each piece is equal to the sliding window step of the query. Conceptually the concatenation of the n pieces constitutes the original data in the window. Replace the original MAL instruction that materializes the window stream, with n instructions that slice the window into each of the n pieces.
- Traverse the plan and find which MAL plan instructions we should replicate, due to window splitting. These are the instructions where the original materialized window stream is involved (explicitly and implicitly).
- Merge the intermediate materialized result at the *proper* place of the query plan.
- Identify the original MAL plan instructions that cannot be replicated. Give them the proper merged input.
- Introduce the instructions for the transition phase. Starting from the source, i.e., slices, down to the intermediate results.
- Place the instructions that the engine should evaluate only once outside the infinite loop.
- Wrap the MAL instructions that correspond to the new portion of the data stream inside the infinite loop. Wrap inside the infinite loop the merging and the transition steps; they need to run continuously
- Traverse the plan to find which slices are needed and which are not needed for the rest of the incremental query evaluation.

This is a quite complex process, since we have to traverse and transform the plan multiple times resulting to a significant makeover of the original query plan. The benefit of developing the transformation logic at the optimization phase, is that we can compile and transform any kind of complex queries automatically while still exploiting traditional DBMS optimization strategies.

Multi-query optimization for sliding window queries is an important area of data streams research. At this level, our implementation does not prevent automatic multi query optimization at the compilation phase.

5.6 Experimental Analysis

In this section, we provide a detailed experimental analysis of incremental processing in our DataCell implementation over MonetDB v5.15 All experiments are on a 2.4 GHz Intel Core2 Quad CPU equipped with 8 GB RAM and running Fedora 12.

Experimental Set-up and Outline

We compare DataCell incremental processing against the typical re-evaluation approach which reflects the straight-forward way of implementing streaming over a DBMS. In the rest of this section, we refer to the former implementation simply as $DataCell_I$ and the latter as $DataCell_R$. In addition, we compare DataCell against a state-of-the-art commercial stream engine, clearly demonstrating the successful design of incremental processing over an extensible DBMS kernel and the potential of blending ideas from both worlds.

We study in detail the effects of various parameters, i.e., query and data characteristics such as window size, window step, selectivity factors, etc. The performance metric used is response time, i.e., the time the system needs to produce an answer, once the necessary tuples have arrived.

In the first part of the experimentation we will study $DataCell_R$ and $DataCell_I$ to acquire a good understanding of how a typical DBMS performance can be transformed into an incremental one and the parameters that affect it. Given that these two implementations are essentially built over the same code base, this gives a clear intuition of the gains achieved by the incremental DataCell over a solid baseline. Then, with this knowledge in mind, in the second part we will see in detail how this performance compares against a specialized engine and what are the parameters that can swing the behavior in favor of one or the other approach.

We will use a single stream and a multi-stream query.

```
(Q1) SELECT x1, sum(x2)
     FROM stream
     WHERE x1 > v1
     GROUP BY x1
```

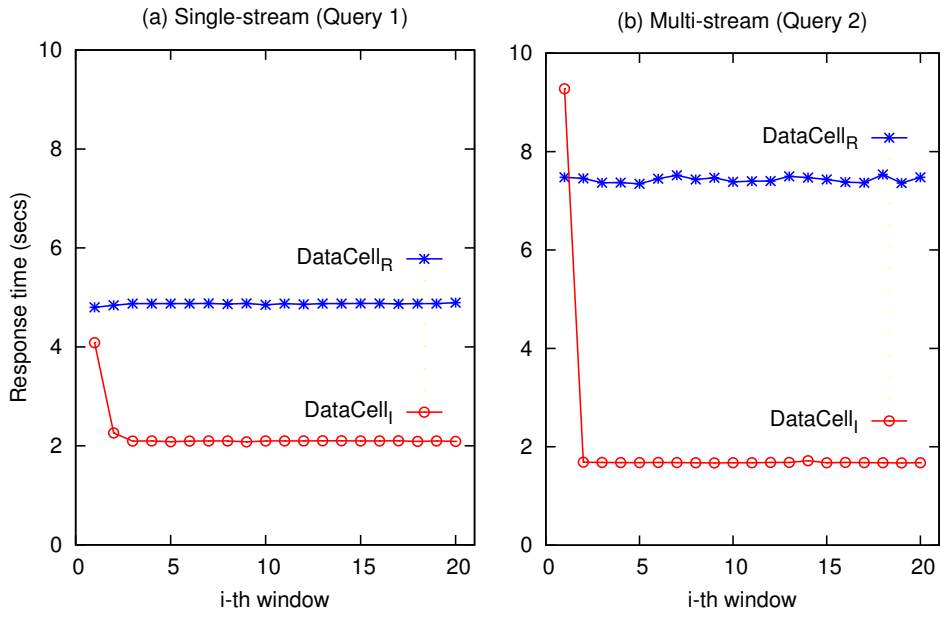

Figure 5.8: Basic Performance

```
(Q2) SELECT max(s1.x1), avg(s2.x1)
     FROM stream1 s1, stream2 s2
     WHERE s1.x2 = s2.x2
```

5.6.1 Basic Performance

The first experiment demonstrates the response times as the windows slide. Considering the single stream query first, we use a fixed window size, step and selectivity. Here, we use window size $|W| = 1.024 * 10^7$ tuples, window step $|w| = 2 * 10^4$ tuples, and 20% selectivity. This way, the DataCell plan rewriter splits the initial window into 512 basic windows. Each time the system gets $|w|$ new tuples, it processes them and merges the result with those of the previous 511 basic windows.

Figure 5.8(a) shows the response times for 20 windows. For the initial window, both $DataCell_R$ and $DataCell_I$ need to process $|W|$ tuples and achieve

similar performance. DataCell is slightly faster mainly because executing the group-by operation on smaller basic windows yields better locality for random access. For the subsequent sliding steps (windows 2-20), $DataCell_R$ shows the same performance as for the first one, as it needs to perform the same amount of work each time. $DataCell_I$, however, benefits from incremental processing, resulting in a significant advantage over $DataCell_R$. Reusing the intermediate results of previously processed basic windows, $DataCell_I$ only needs to process the $|w|$ tuples of the new basic window, and merge all intermediate results. This way, $DataCell_I$ manages to fully exploit the ideas of incremental processing even though it is designed over a typical DBMS kernel. It nicely blends the best of the stream and the DBMS world.

For the double stream query, Query 2, we treat both streams equally, using window size $|W| = 1.024 * 10^5$ and window step $|w| = 1600$, i.e., the initial windows of both streams are split into 64 basic windows each. Figure 5.8(b) shows even more significant benefits for $DataCell_I$ over $DataCell_R$. The reason is that Query 2, is a complex multi-stream query that contains more expensive processing steps, i.e., join operators. DataCell effectively exploits the larger potential for avoiding redundant work.

The fact that incremental processing beats re-evaluation is not surprising of course (although later we will demonstrate the opposite behavior as well). What is interesting to keep from this experiment is that by applying the incremental logic at the query plan level we achieve a significant performance boost achieving efficient incremental processing within a DBMS.

5.6.2 Varying Query Parameters

The processing costs of a query depend on a number of parameters related to the semantics of the query, e.g., selectivity, window size, step size, etc. These are not tuning parameters, but reflect the requirements of the user. In general, the more data a query needs to handle (less selective/bigger windows, etc.), the more incremental processing benefits as it avoids processing the same data over and over again. In the following paragraphs, we discuss the most important of these parameters and their implications in detail.

Selectivity

We start with Query 1, using a window size of $1.024 * 10^7$ tuples and a step of $2 * 10^4$ tuples. By varying the selectivity of the selection predicate from 10% to 90%, we increase the amount of data that has to be processed by the group-by

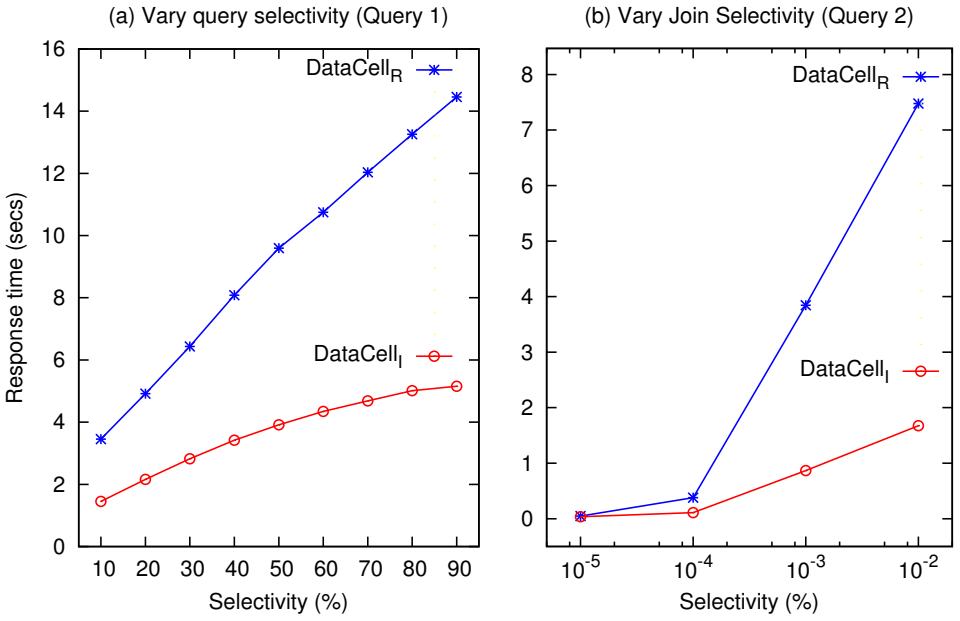

Figure 5.9: Varying Selectivity

and aggregation. Figure 5.9(a) shows the results. For both $DataCell_R$ and $DataCell_I$, the response times for a sliding step increase close to linear with the increasing data volume. However, the gradient for $DataCell_R$ is much steeper as it needs to process the whole window. Incremental processing allows DataCell to process only the last basic window, resulting in a less steep slope, and hence, an increasing advantage over $DataCell_R$.

A similar effect can be seen with the join query in Figure 5.9(b). We use $|W| = 1.024 * 10^5$ and $|w| = 1600$ and vary the join selectivity from $10^{-5}\%$ to $10^{-2}\%$. Due to the more expensive operators in this plan, the benefits of DataCell are stronger than before.

Window Size

For our next experiment, we use Query 1 with selectivity 20% and vary the window size. Keeping the number of basic windows invariant at 512, the step

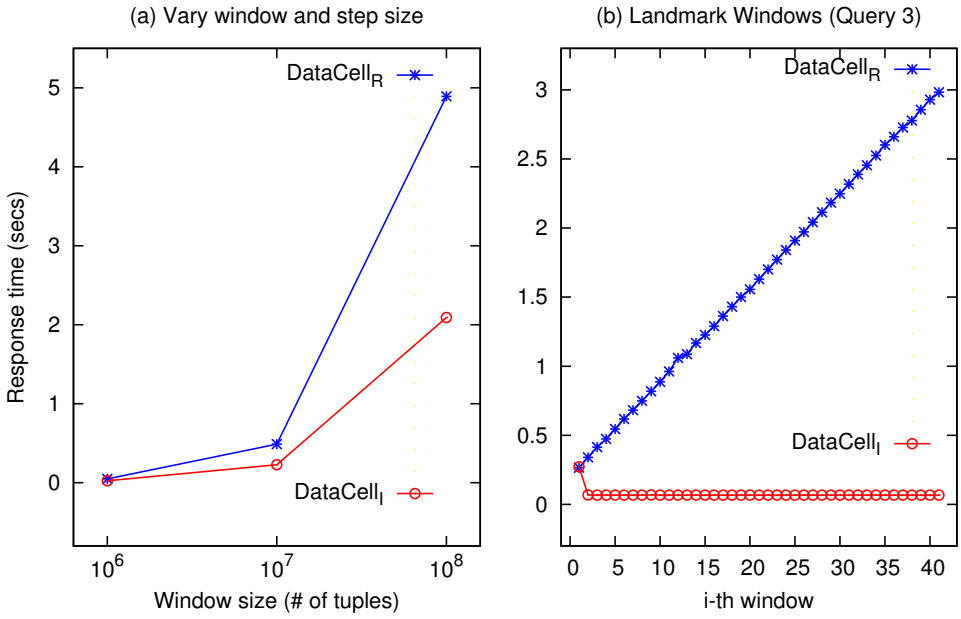

Figure 5.10: Varying Window and Step Size

size increases with the total window size. Figure 5.10(a) reports the response time required for a sliding step using three different window sizes. The bigger the window, i.e., the more data we need to process, the bigger the benefits of incremental processing with $DataCell_I$ over $DataCell_R$ materializing more than a 50% improvement. Again this clearly demonstrates the effectiveness of our incremental design using a generic storage and execution engine.

Landmark Queries

By definition, the window size of landmark queries increases with each sliding step, the step size is invariant. We run the following single-stream query as landmark query, using $|w| = 2.5 * 10^6$ and 20% selectivity.

```
(Q3) select max(x1),sum(x2)
     from stream where x1>v1
```

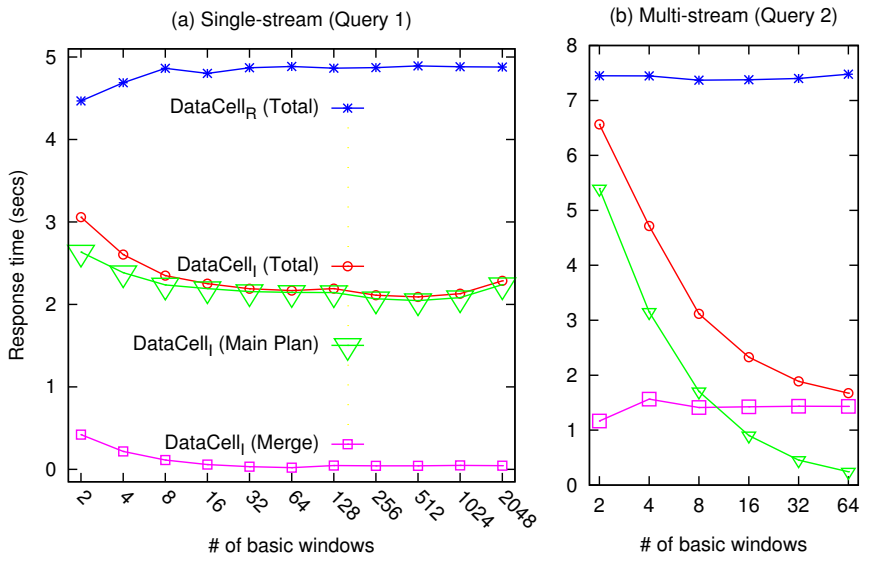

Figure 5.11: Decreasing Step (Incr. Number of Basic Windows)

Figure 5.10(b) shows the response time for 40 successive windows. As in Figure 5.8, MonetDB and DataCell yield very similar performance for the initial window, where both need to process all data. The re-evaluation approach of DataCell then makes the response time grow linearly with the growing window size. With $DataCell_I$, the response time for the second query drops to what is required to process only the new basic windows, and then stays constant at that level, exploiting the benefits of incremental processing.

Step Size

With invariant window size, decreasing the step size in turn means increasing the number of basic windows per window, i.e., the number of intermediate results that need to be combined per step.

Figure 5.11(a) shows the results for Query 1. We use window size $w = 1.024 * 10^7$ tuples and a selectivity of 20%. With a small number of basic windows, i.e., with a big window step, we still need to process a relatively big amount of data each time a window is completed. Thus, response times are still

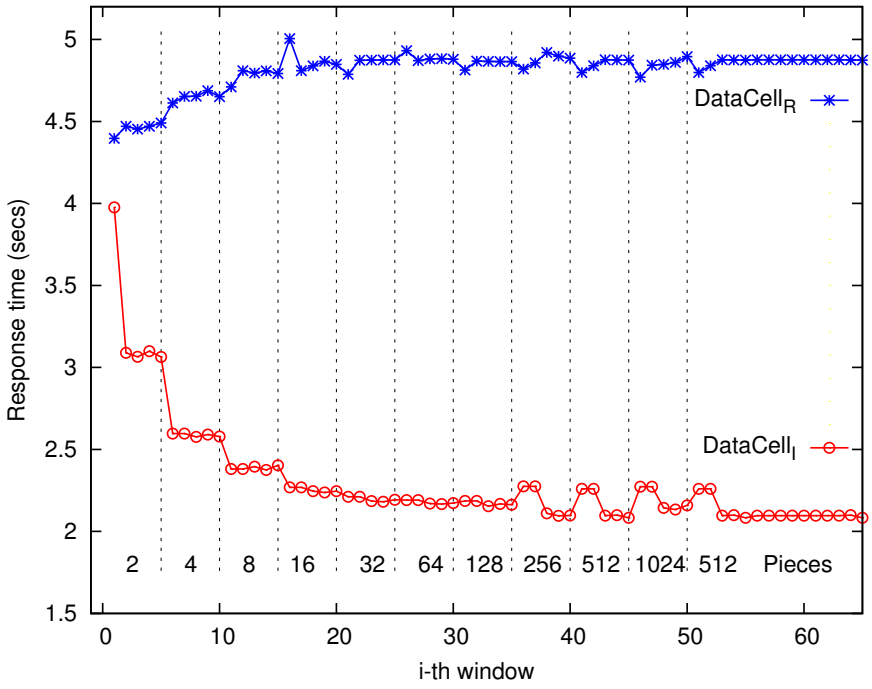

Figure 5.12: Query Plan Adaptation

quite high, e.g., for 2 basic windows. However, as the number of basic windows increases, $DataCell_I$ improves quickly until it stabilizes once fixed initialization costs dominate over data-dependent processing costs.

Figure 5.11(a) also breaks down the cost of $DataCell_I$ into two components. First, is the actual query processing cost, i.e., the cost spent on the main operators of the plan that represent the original plan flow. Second is the merging cost, i.e., all additional operators needed to support incremental processing, i.e., operators for merging intermediates, performing the transitions at the end of a query plan and so on. Figure 5.11(a) shows that the cost of merging becomes negligible. The main component is the query processing cost required for the original plan operators.

Notice also that there is a small rise in the total incremental cost with many basic windows (i.e., >1024 in Fig. 5.11(a)). This is attributed to the query

processing cost which as we see in Figure 5.11(a) follows the same trend. What happens is that with more basic windows, a larger number of intermediates are maintained. Their total size remains invariant. However, with more basic windows, there are more (though smaller) intermediates to be combined and thus more operator calls required to make these combinations (the group-by) in this case. The administrative cost of simply calling these operators becomes visible with many basic windows.

Figure 5.11(b) shows a similar experiment for Query 2. Overall the trend is similar, i.e., cutting the stream window into smaller basic windows, brings benefits. The big difference though is that the break down costs indicate an opposite behavior than with Query 1. This time, the query processing cost becomes negligible while the merging cost is the one that dominates the total cost once the query processing part becomes small. The reason is that the intermediates this time are quite big, meaning that simply merging those intermediates is significantly more expensive. This cost is rather stable given that the total size of intermediates is invariant with invariant window size, regardless of the step size.

5.6.3 Optimization

As discussed in Section 5.4 and supported by the results of the above experiments, the response time of incremental DataCell plans can further be improved by pro-actively processing the last basic window in smaller chunks than the step size defined in the query. This way, we favor a dynamic self-adapting approach over a static optimization using an analytical cost model. Figure 5.12 shows the results of an experiment, where DataCell successively doubles the number m of chunks for a basic window every five steps as proposed in Section 5.4. Monitoring the response times, DataCell observes a steady performance improvement up to $m = 512$. With $m = 1024$, the performance starts degrading, triggering DataCell to resort to $m = 512$.

5.6.4 Comparison with a Specialized Engine

Here, we test our DataCell prototype against a state-of-the-art commercial specialized engine. Due to license restrictions we refrain from revealing the actual system and we will refer to it as SystemX. In addition, we tested a few open-source prototypes but we were not successful in installing and using them, e.g., TelegraphCQ and Streams. These systems were academic projects and are not supported anymore making it very difficult to use them (in fact we are not

aware of any stream papers comparing against any of these open-source stream systems). For example, TelegraphCQ compiled on our contemporary Fedora 12 system only after fixing some architecture-specific code. However, we did not manage to analyze and fix the crashes that occurred repeatedly when running continuous queries. System Streams seemed to work correctly but the functionalities of getting the performance metrics did not work. The most important issue though is that it does not support sliding windows with a slide bigger than a single tuple. Nevertheless, we are confident that comparison against a most up-to-date version of a state-of-the-art commercial engine provides a more competitive benchmark for our prototype.

For this experiment, we use the double stream Query 2. The metric reported is the total time needed for the system to consume a number of sliding windows and produce all results. Using a total of 100 windows and 64 basic windows per window, we vary the window size between $|\underline{W}| = 10^3$ and $|\overline{W}| = 10^5$ tuples with the respective step size growing from $|\underline{w}| = |\underline{W}|/64 \cong 16$ to $|\overline{w}| = |\overline{W}|/64 \cong 1600$ tuples. Thus, in total, we feed the system $|\underline{W}| + 100 * |\underline{w}| \cong 2600$ tuples in the most lightweight case and with $|\overline{W}| + 100 * |\overline{w}| \cong 260000$ tuples in the most demanding case.

Previous experiments demonstrated purely the query processing performance. Here, we test the complete software stack of DataCell, i.e., data is read from an input file in chunks. It is parsed and then it is passed into the system for query processing. The input file is organized in rows, i.e., a typical csv file. DataCell has to parse the file and load the proper column/baskets for each batch. Similarly for SystemX. For all systems, we made sure that there is no extra overhead due to tuple delays, i.e., the system never starves waiting for tuples, representing the best possible behavior.

Figure 5.13 shows the results. It is broken down into Figure 5.13(a) for small windows, i.e., smaller than 10^4 tuples and into Figure 5.13(b) for bigger windows. For very small window sizes, we observe that plain DataCell gives excellent results, even outperforming the stream solutions in the smaller sizes. The amount of data to be processed is so small that simply the overhead involved around the incremental logic in a stream implementation becomes visible and decreases performance. This holds for both $DataCell_I$ and SystemX, with the latter having a slight edge for the very small sizes.

Response times though are practically the same for all systems as they are very small anyway. However, as the window and step size grow, we observe a very different behavior. In Figure 5.13(b), we see that plain DataCell is losing ground to $DataCell_I$. This time, the amount of data and thus computation needed becomes more and more significant. The straight-forward implementa-

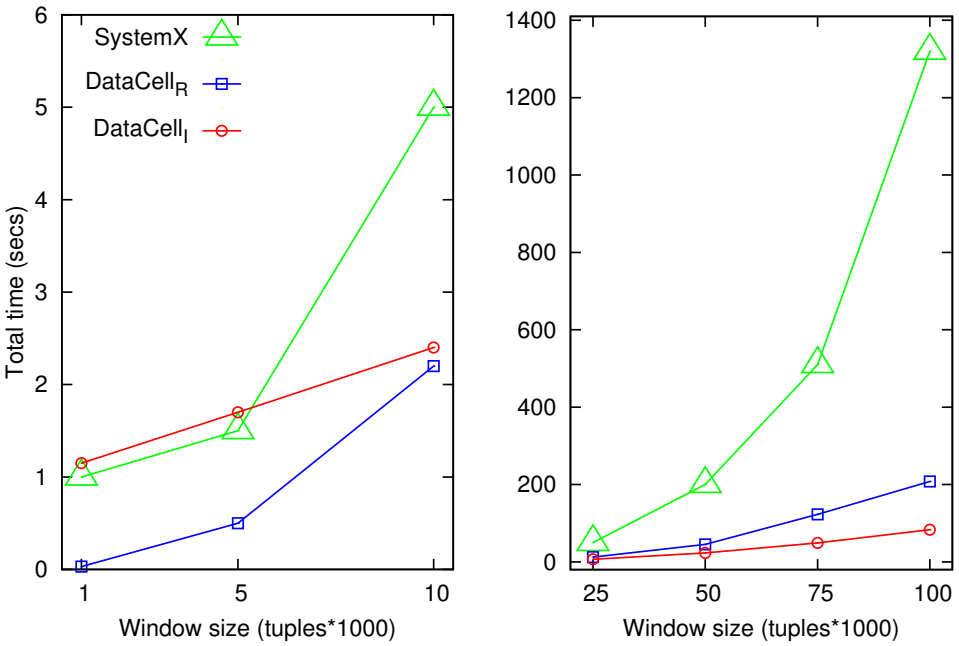

Figure 5.13: Comparison with a Stream Engine

tion of stream processing in a DBMS cannot exploit past computation leading to large total costs. In addition, we see another trend; DataCell scales nicely with the window size and now becomes the fastest system.

SystemX fails to keep up with $DataCell_I$ and even plain DataCell. When going for large amounts of data and large windows, batch processing as exploited in $DataCell_I$, gains a significant performance gain over the typical one tuple at a time processing of specialized engines. The main reason is that we amortize the continuous query processing costs over a large number of tuples as opposed to a single one. In addition, the incremental logic overhead is moved up to the query plan as opposed to each single operator.

Modern trends in data warehousing and stream processing support this motivation (Winter and Kostamaa, 2010) where continuous queries need to handle huge amounts of data, e.g., in the order of Terabytes while the current literature on stream processing studies only small amounts of data, i.e., 10 or 100

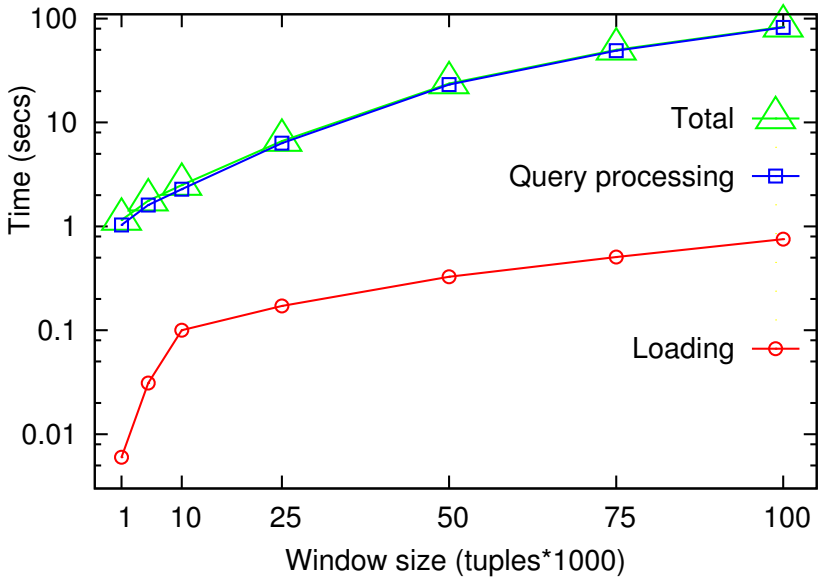

Figure 5.14: DataCell$_I$ break down costs

tuples per window in which case tuple at a time processing behaves indeed well. An interesting direction is hybrid systems, i.e., provide both low-level incremental processing as current stream engines and high level as we do here, and interchange between different paradigms depending on the environment.

Finally, Figure 5.14 breaks down the *DataCell$_I$* costs seen in the previous figure into pure query processing costs and loading costs, i.e., the costs spent in parsing and loading the input file. We see that query processing is the major component while loading represents only a minor fraction of the total cost.

5.7 Conclusions

In this chapter, we have shown that incremental continuous query processing can efficiently and elegantly be supported over an extensible DBMS kernel. These results open the road for scalable data processing that combines both stored and streaming data in an integrated environment in modern data warehouses. This is a topic with strong interest over the last few years and with a great potential

impact on data management, in particular for business intelligence and science. Building over an existing modern DBMS kernel to benefit from existing scalable processing components, continuous query support is the missing link. Here, we study in this context one of the most critical problems in continuous query processing, i.e., window based incremental processing.

Essentially, incremental processing is designed and implemented at the query plan level allowing to fully reuse (a) the underlying generic storage and execution engine and (b) the complete optimizer module. In comparison with a state-of-the-art commercial DSMS, DataCell achieves similar performance with small amounts of data, but quickly gains a significant advantage with growing data volumes, bringing database-like scalability to stream processing.

The following chapter concludes the thesis and discusses a number of interesting open topics and research directions towards a complete data management architecture that integrates database and stream functionalities in the same kernel. DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective and by taking into account the needs of modern data management applications for scalable stream processing combined with traditional query processing. The range of topics discussed in this chapter include multi-query processing, adaptive query processing, query relaxation, distributed processing, and realizing DataCell in alternative architectures.

Chapter 6

Conclusions and Future Research Paths*

In this thesis, we set the roots for a novel data management architecture that naturally *integrates* database and stream query processing inside the same query engine. As we discussed in the beginning of the thesis, there is a large demand nowadays to combine efficient and scalable stream and one-time processing. We start with a modern column-store architecture, realized in the MonetDB system, and we design our new system in this kernel. Column-store architectures offer the requirement for efficient one time processing and our main contribution here is the design of a column-store system that can do both stream and one-time processing efficiently.

The reason to choose this research direction comes from today's application requirements to support both processing models providing advanced processing in both cases. So far the research community used to deal with this scenario in two ways. The first way is by trying to build specialized stream systems that in addition to stream processing provide *simple* processing of persistent/historical data. However, in this case, we are not able to reach the sophisticated techniques of mature database systems, especially when we need to support complex queries and/or big data analysis. An alternative direction is to externally con-

*Part of the material in this chapter has been presented at VLDB11 PhD Workshop paper "DataCell: Building a Data Stream Engine on top of a Relational Database Kernel." (Liarou and Kersten, 2009) and at the PVLDB11 paper "The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds" (Kersten et al., 2011).

nect and synchronize under the same middleware two specialized processing engines, i.e., a separate data stream engine and a separate DBMS, assigning different processing tasks to each one of them. The vision of an integrated processing model, has been considered in the past in the context of active databases and database triggers. However, it was soon rejected once the requirements of streaming applications became demanding for near real-time processing, multi-query optimizations and adaptive query processing; these are concepts that at this moment were new and different from the ones of the original database scenarios. In this thesis, we reconsider the path to implant on-line capabilities within a modern column-store database kernel in a way that we can efficiently synthesize and support interesting scenarios with streaming and database functionalities. In the DataCell project we exploit, reuse, redirect and extend the useful parts that the existing database technology already offers, to support a more complete query processing scenario, where the need of active and passive processing co-exist.

In this chapter, we will discuss and summarize our contributions to this research direction. We will also discuss a number of interesting future research topics towards scalable and efficient stream and one-time query systems, e.g., multi-query processing, adaptive query processing, query relaxation, distributed processing, etc.

6.1 Contributions

Basic DataCell Architecture

In this thesis, we introduced the basic DataCell architecture to exploit the notion of scalable systems that can provide both streaming and database functionality. We first showed the minimal additions that allow for stream processing within a DBMS kernel. The unique goal of DataCell is to exploit, as much as possible, all the available infrastructure offered by the underlying database kernel. In this way, we built our system using the majority of the original relational operators and optimization techniques, elevating the streaming functionality mainly at the query plan and scheduling level. The first DataCell architecture (Chapter 3) resulted in a model that allows to repeatedly run queries over incoming data as new data continuously arrives. Already this model was shown to provide substantially good streaming performance mainly by exploiting the power of modern column-store architectures. We were able to run the complete Linear Road benchmark and be well within the timing requirements set in the

benchmark (see Section 3.5.2).

Incremental Processing

With the basic DataCell architecture at hand, the next step was to work on major stream functionalities. The topic we dealt with was incremental processing; this is necessary in order to be able to efficiently support window continuous queries over streaming data. Most relational operators underlying traditional DBMSs *cannot* operate incrementally without a major overhaul of their implementation. Here, we show that efficient incremental stream processing is possible in a DBMS kernel handling the problem *at the query plan and scheduling level*. For this to be realized, the relational query plans are transformed in such a way that the stream of data is continuously broken into pieces and different portions of the plan are assigned to different portions of the focus window data. DataCell takes care that this "partitioning" happens in such a way that we can exploit past computation during future windows. We illustrated the methods to extend a modern column-store with the ability to create and rewrite incremental query plans. The end result was efficient and crucially scalable incremental processing. As we show in this thesis, DataCell with incremental processing available can be much faster and scalable than a state of the art commercial stream system (Chapter 5).

6.2 Looking Ahead

In this thesis, we made the first steps towards a complete data management architecture that integrates database and data stream functionalities in the same kernel. DataCell fundamentally changes the way that stream data is handled and processed, trying to exploit many traditionally core database techniques and ideas.

In this way, DataCell brings a significantly different view on how to build stream systems and radically changes the way we process data streams. Thus, it also brings the need to reconsider several of the well established techniques in the stream processing area. The road-map for DataCell research calls for innovation in many important stream processing areas. In the rest of this section, we will touch on these topics and where possible we will also provide discussion on possible research paths for solving these problems in the DataCell context.

6.2.1 Multi-Query Processing

A critical issue is that of multi-query processing and the rich scheduling opportunities that control the interaction between multiple continuous queries. In traditional stream processing, this area has received a lot of attention with several innovative solutions, e.g., (Sharaf et al., 2008). DataCell offers all the available ingredients to achieve similar levels of multi-query optimizations, while keeping the underlying generic engine intact. Below we discuss some of these directions.

Splitting and Merging Factories

Exploiting similarities at the query and data level is necessary in order to meet the real-time deadlines a stream application sets. In this way, we need to study mechanisms to efficiently and dynamically organize the queries in multiple groups based on their needs and properties. To accommodate partially overlapping queries we also need mechanisms to dynamically *split* and *merge* factories that wrap the query plans or parts of them.

DataCell here can adopt part of the existing literature in multi-query processing but there is also room to investigate research opportunities that arise from the basic DataCell processing model. One of the main innovations in DataCell comes from the choice to elevate several of the stream functionalities at the query plan and scheduling level. This allows for efficient reuse of core database functionalities and optimizations. In this way, one of the most challenging directions for multi-query processing in DataCell is the choice to *split* the plan of a single query into multiple factories. The motivation for this comes from different angles. For example, each factory in a group of factories sharing a basket, conceptually releases the basket content only after it has completed all operators in its query plan. Assume two query plans; a lightweight query q_1 and a heavy query q_2 that needs a considerably longer processing time compared to q_1. With the shared baskets strategy (see Section 3.3.2), we force q_1 to wait until q_2 finishes before we allow the receptor to place more tuples in the shared basket such that q_1 can run again. A simple solution is to split a query plan into multiple parts, such that part of the input can be released as soon as possible, effectively eliminating the need for a fast query to wait for a slow one.

Another natural direction once we decide to split query plans into multiple factories, is the possibility to share both the baskets and the execution cost. For example, queries requiring similar ranges in selection operators can be supported by shared factories that give output to more than one query's factories.

Auxiliary factories can be plugged in to cover overlapping requirements.

DataCell Cracking

Another interesting direction for multi-query processing in DataCell is to exploit the idea of database cracking (Idreos, 2010). Database cracking was proposed as an adaptive indexing technique in the context of column-stores with bulk processing. The idea is that data is continuously physically reorganized building indexes incrementally and adaptively based on the requests of incoming queries. DataCell scheduling can exploit such ideas by allowing similar queries to run over the same baskets in a particular order. These queries can then use cracking-like ideas to continuously reorganize the basket and thus allowing successive queries to operate faster and faster for a given batch of incoming tuples. Challenges here include the dynamic scheduling of queries, i.e., which queries to allow to crack which baskets and in which order. Cracking is very sensitive in the order we process queries as this affects the kind of clustering and thus optimization achieved. Other challenges include finding a good balance between investment and amortization of the investment as in normal databases any index built can be exploited "forever", while in our case the cracked baskets will only be useful for a given window of time.

6.2.2 Adaptation

Adaptive query processing is another very important issue in data streams. Dynamic changes to the arrival rate of data streams and on correlations between the incoming data, drastically affect the computation value of the continuously executed operations. In addition, new continuous queries are submitted over time while some of the old ones may expire and this changes the overall query processing behavior of the system. In this context, static query optimizations made up-front may not be valid after some time. Below we discuss some interesting directions for DataCell in this context.

Adaptive Behavior in Traditional Streams

Many academic prototypes presented extensive work on this topic. For example, StreaMon (Babu and Widom, 2004), the adaptive query processing infrastructure of STREAM (Arasu et al., 2003), collects statistics about stream and query plan characteristics and takes the appropriate actions to always ensure that the

query plan and memory usage are optimal for the current input characteristics. TelegraphCQ (Chandrasekaran et al., 2003) constructs query plans with adaptive routing modules, called Eddies (Avnur and Hellerstein, 2000). Thus, it is able to proceed to continuous run-time optimizations, dynamically adapting to the workload. Eddies modules adaptively decide how to route data to appropriate query operators on a tuple-by-tuple basis.

Adaptive Behavior in DataCell

Several key steps in the DataCell architecture are already adaptive in nature. Once a query is submitted in DataCell, it is parsed, compiled, optimized and then ends up to the pool with the other continuous queries, waiting to start processing incoming stream tuples. We first see an adaptive behavior when a factory considers how to proceed to the processing of the incoming chunk of data. It dynamically decides which way to evaluate the query, choosing between incremental processing and the re-evaluation method. As we have seen, window queries in periods with a low rate of incoming tuples can by default be executed according to the re-evaluation model. Once the arrival rate of the data streams becomes extremely high or bursty, the factory proceeds to a dynamic self-adaptive solution to find the optimal chunk size and proceed to incremental processing of the partial chunks.

By default DataCell starts with full re-evaluation, considering that the processing chunk is the same as the window size. Then, we successively modify the chunk size monitoring the response time for a couple of sliding steps. As long as the response times decrease by increasing the number of chunks in a window, we keep increasing this number. Only once the increasing merging overhead out-weights the decreasing processing costs, the response times increase, again. Then, we stop increasing the number of chunks and reset it to the value that resulted in the minimum response time.

Adaptive DataCell Query Plans

Most of the past work on adaptive query processing in stream systems naturally focuses on adaptive query plans, i.e., choosing different plan configurations for a given query depending on changes in the environment, the system, the data and the queries. The adaptive features discussed for DataCell above are mainly at a different level that has to do with the administration of the system and the resources.

However, there is plenty of room for more optimization by considering adaptation at the query plan level too that goes beyond the choice of re-evaluation and incremental evaluation. For example, choosing different shape of query plans depending also on multi-query processing issues can be of crucial importance. Thus, again the choice of how to organize factories, how to dynamically split and merge query plans depending on the changes of the environment becomes an important issue.

At this point we should mention that given the modern column-store roots of DataCell, we already exploit some adaptive optimization at run time. Even if the query plan is static and optimized only once, at the submission time of the query, the operators are evaluated in a dynamic way. Given the bulk processing model, each operator knows exactly what is its input at execution time. For example, before executing a join we have first collected all tuples from both join inputs which means that we know their size, properties such as cardinality and possibly other data quality properties that allow us to dynamically decide the proper join algorithm.

However, full re-optimization and full adaptive query processing that allow the system to quickly adapt and continuously match the workload is a mandatory feature of modern stream engines. Here DataCell research can exploit ideas such as dynamic sampling and possible re-optimization if initial choices seem wrong, etc.

6.2.3 Dualism

In the DataCell context we have challenges that arise by combining the two query processing paradigms in one. More and more applications require this functionality and we can naturally expect that this will become a more mainstream processing model in the coming years. For example, this applies to scientific databases as well as in social networks where new data continuously arrives and needs to be combined with past data.

Once the technology of merging both continuous and one-time query processing becomes more mature, we expect a plethora of rich topics to arise especially when optimization becomes an issue. For example, query plans that touch both streaming data and regular tables might require new optimizer rules or adaptations of the current ones. There, all the choices made in respect to optimizing single continuous or one-time queries need reconsideration. Similarly for multi-query processing. Overall, DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective.

6.2.4 Query Relaxation

Pure stream systems traditionally focus on small scale applications with a rather *small* rate of incoming data. Nowadays, though, the requirements are changing towards systems that should be able to handle data streams of Terabytes on a daily basis. For example, scientific databases and large corporate databases create a huge pile of new data each day and need to run the same queries over and over again, combine past data with new ones and so on (Winter and Kostamaa, 2010).

Typical stream systems are not designed with such workloads in mind. With DataCell we make a significant step towards scalable stream processing by exploiting modern column-store features such as bulk processing and vectorized processing. However, as the data grows even more and in order to support new kinds of applications such as scientific databases we need to rethink certain query processing assumptions. For example, complete answers are often not possible due to the limited resources given the workload. Furthermore, the exploration and comprehension of data streams with a very high rate of incoming data, may lead to fundamentally different processing models.

In light of these challenges, we should rethink some of the strict requirements data stream systems adopted in the past. Next generation data stream management systems should *interpret queries by their intent*, rather than as a contract carved in stone for complete and correct answers. The continuously generated result sets should aid the user in understanding the stream *trends* and provide guidance to continue his data exploration journey as long as the stream is coming and his requirements are possibly modified. The stream engine would ideally interact with the users and help them continuously explore the streaming data in a contextualized way. In the rest of this subsection, we will discuss two possible directions towards more relaxed stream processing.

Approximate Kernels

One of the prime impediments to fast data exploration is the query execution focus on correct and complete result sets, i.e., the semantics of SQL presupposes that the user knows exactly what he expects and needs to monitor. The design and implementation of the query optimizer, execution engine, and storage engine are focused towards this goal. That is, correctness and completeness are first class citizens in modern data stream kernels. This means that when the system needs to perform a few hard and expensive unavoidable steps, it is designed to perform them such that it can produce the complete and correct re-

sults. However, the query accuracy may have a significant impact on the query processing time that potentially will lead to deadline violations.

With input data sizes growing continuously, the research path of *query approximation*, was born such as to cope with the demanding short response times in stream applications. With huge data sizes that cannot be processed in a reasonable time *load shedding* has been widely adopted by the stream community as the most natural approach (Tatbul, 2007). There, we skip processing the whole input (e.g., by dropping tuples or creating tuple summarizations) aiming to save processing resources, even if this action will drastically affect our query answers. If the user accidentally chooses an expensive monitoring condition that produces a large result set, then a sample might be more informative and more feasible. Unfortunately, such a sample depends on the data distribution, the correlations, and data clustering in the data stream and the query result set. Taking a sample can still cause significant performance degradation that surface only at run time.

Current approximation techniques have only been studied for simple and small scale scenarios. Sampling and load shedding allow to drop part of the workload by completely ignoring certain incoming tuples. Summarization techniques create summaries over the data allowing to query the smaller summary and get a quick approximate response. For scientific databases though, even creating such summaries on the daily stream of Terabytes becomes a challenge on its own. Specifically, in stream processing it may not be worth creating summaries for small windows of data.

The above techniques require either a significant preprocessing step which can be prohibitive in large scale data or a strict up-front isolation of certain input parts. Here, we scrabble a novel direction where approximation becomes the responsibility of individual operators allowing a query processing kernel to self-organize and decide on-the-fly how to better exploit a given resource budget. For example, a hash-join may decide not to prompt the hash table for a given set of the inner, or after hitting a bucket where it has to follow a long list it may decide to skip this tuple of the inner.

The idea is to address the problem at its root; we envision a kernel that has rapid reactions on user's requests. Such a kernel differs from conventional kernels by trying to identify and avoid performance degradation points on-the-fly and to answer part of the query within strict time bounds, but also without changing the query focus. Its execution plan should be organized such that a (non-empty) answer can be produced within T seconds.

Although such a plan has a lot in common with a plan produced by a conventional cost-based optimizer, it may differ in execution order, it may not let

all operators run to completion, or it may even need new kinds of operators. In other words, an approximate kernel sacrifices correctness and completeness for performance. The goal is to provide a quick and fully interactive gateway to the data until the user has formulated a clear view of what he is really searching for, i.e., it is meant as the first part of the exploration process.

At this point note that the stream world has already sacrificed completeness and correctness when the window processing model was introduced in order to bound the infinite inputs. However, this has the same effect as with sampling and it cannot always guarantee good performance as the quality of the data may force expensive operations.

For example, very often during a plan we need to sort large sets of rowIDs to guarantee sequential data access. Those can be replaced by a cheaper clustering method or we can refrain from data access outside the cache. Likewise, operations dealing with building auxiliary structures over the complete columns/tables, can be broken up into their piecewise construction. Building just enough within T to make progress in finding an answer. If T is really short, e.g., a few seconds, the plan may actually be driven from what is already cached in the memory buffers. In a modern server, it is just too expensive to free up several Gigabytes of dirty memory buffers before a new query can start. Instead, its memory content should be used in the most effective way. In the remaining time the memory (buffer) content can be selectively replaced by cheap, yet promising, blocks. With a time budget for processing, the execution engine might either freeze individual operators when the budget has been depleted, or it might replace expensive algorithms with approximate or cheaper alternatives.

These ideas extend from high level design choices in operators and algorithms all the way to lower level (hardware conscious) implementation details. For example, during any algorithm if we reach the case where we need to extend, say, an array in a column-store with a realloc, an algorithm in that kernel may choose to skip this step if it will cause a complete copy of the original array.

This sketch is just the tip of the iceberg, i.e., numerous examples and variations can be conceived. The key challenge is to design a system architecture where budget distribution can be dynamically steered in such a way that the query still produces an informative result set. Aside from a tedious large-scale (re-)engineering effort to build a kernel on this assumption, major research questions arise. For example:

- How is the budget spread over the individual operators?
- What actions are operators allowed to take to stay within the budget?

- How to harvest the system state produced by previous queries?
- How to replace the relational operators and index constructors with incremental versions?
- What all this means for dynamic and continuous adaptation of stream plans?
- How do such ideas combine with multi-query processing ideas in streaming environments?

At first sight the above ideas do not fit with the initial goal of DataCell to use existing and optimized database operators in order to exploit mature database technology. However, as we discussed in this section, the requirements of radically new applications such as scientific databases and social networks go way beyond what current technology can support which implies that drastic changes are required. Ideas such as the one discussed above apply both to traditional databases and to stream processing. For example, in the context of the DataCell the basic architecture could remain the same while the underlying core operators are updated to their approximate alternatives.

Query Morphing

In the same spirit as with the approximate kernels ideas presented above, we can also extend the ideas of approximate query processing to the actual patterns of the queries posed by the user. In this paragraph, we scrabble the vision where a stream processing kernel participates *more actively* in the complete query processing experience of the user, offering an additional mechanism that provides query pattern suggestions. According to the standard way a DSMS works, a user should first have a general clear idea of what to expect from the incoming data stream and then formulate and submit the corresponding continuous queries. However, when we are dealing with streams with high rates of incoming tuples, and our perspective on incoming data is not still clear or may be drastically modified depending on dynamic conditions, we could end up "missing" valuable stream data and wasting resources on analysis that ends up not being useful. For example, this can easily happen when our original continuous queries are not representative enough, of what we really wanted to monitor giving zero-hit or mega-hit result sets. This phenomenon is typical in exploratory scenarios such as scientific databases. Expensive query processing, in conjunction with the rapidly incoming data steams, trigger the vision for a data stream kernel that becomes a query consultant.

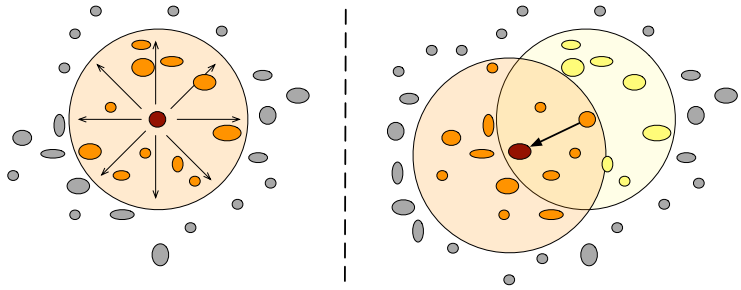

Figure 6.1: Query Morphing

We introduce the notion of *query morphing* as an integral part of query evaluation. It works as follows, the user gives a starting query Q and most of the effort T is spent on finding the "best" answer for Q. But a small portion is set aside for the following exploratory step. The query is syntactically adjusted to create variations Q_i, i.e., with a small edit distance from Q. The process of query morphing is visualized on the left part of Figure 6.1. The user's original request for a window stream returns the result set depicted by the small red circle. However, the processing kernel grabs the chance to explore a wider query/data spectrum in parallel, providing additional results for queries that belong in the *close area*, surrounding the original continuous query. The arrows that start from the red circle indicate this edit area in our example. In this way, the user also receives the orange elliptic query results that correspond to variations of his original request. In the right part of above figure, we see that the user may as a next step decide to shift his interest towards another query result, inspired by the result variations. A new query area now surrounds the user's request, including both past and new variations of the query. This feature is very useful, once the user wants to monitor the incoming stream in a wider range and not stuck to his original request. This is not a one-time processes, as long as the input stream flows different trends could be identified in a continuously modified context.

Several kinds of adjustments can be considered to create the query variations, e.g., addition/dropping of predicate terms, varying constants, widening constants into ranges, joining with auxiliary tables through foreign key relationships, etc. The kind of adjustments can be statistically driven from the original submitted continuous queries, combinations of queries submitted by different source, or cached (intermediate partial) results. Since we have already spent part of our time on processing Q, the intermediates produced along the way can also help to achieve cheap evaluation of Q_i.

Of course, another crucial topic here is how this relates with the continuous adaptation nature that a stream system should have, i.e., how these exploratory query suggestions fit within the grand picture of continuously optimizing stream performance as the environment changes. Similarly, for multi-query processing there are opportunities to grab suggestions by exploiting multiple existing continuous queries and essentially transferring knowledge from one query pattern to the next. In other words, we can use the network of queries in order to provide suggestions about interesting queries or result sets.

The approach sketched aligns to proximity-based query processing, but it is generalized to be driven by the query edit distance in combination with statistics and re-use of intermediates. Query morphing can be realized with major adjustments to the query optimizer, because it is the single place where normalized edit distances can be easily applied. It can also use the plan generated for Q to derive the morphed ones. The ultimate goal would be that morphing the query pulls it in a direction where information is available at low cost. In the ideal case, it becomes even possible to spend all time T on morphed queries.

6.3 Distributed Stream Processing

With data volumes continuously growing, there is a pressing need to look into scalable query processing. The approximate query processing ideas described in the previous section, are a step in this direction.

However, there are more options to consider. For example, distributed query processing has always represented a good approach in supporting bigger data and query loads for any data management system. For example, some academic prototypes, e.g., Borealis (Abadi et al., 2005), and commercial systems, e.g., (Gedik et al., 2008), have focused on this topic. Many of the ideas that have been already proposed for distributed stream processing can also be applied in our context. The stream engine of DataCell can become the central processing part of each node and issues related to how we should distribute the data and the queries, to coordination, to communication protocols, and to fault tolerance can be handled in at a higher level, i.e., the core of the DataCell does not have to change.

6.4 DataCell in Different Database Kernels

The DataCell architecture is designed over modern column-store architectures. This fact raises a valid and at the same time important question whether our ultimate vision to include efficient stream processing in the heart of a traditional DBMS is limited to the underlying column-store architecture. The questions we should answer here are the following.

- Can we apply the same ideas on top of a row-store DBMS?
- How critical and unique are the column-store architecture advantages which enable DataCell?

In the basic DataCell architecture, each query is encapsulated into a factory, i.e., a function that wraps a continuous query plan in an infinite loop. Streaming data is temporarily collected into baskets and remains there until its consumption by the connected factories/operators. Baskets and tables can harmoniously co-exist and interact, while the optimization algorithms are applied with minimal changes to both one-time and continuous queries. Note, that the purely stream specialized optimizations, i.e., incremental processing, are only applied to the corresponding queries.

The DataCell philosophy as briefly summarized above seems that could easily be applied in a row-store architecture, too. The existence of an intermediate scheduler, that orchestrates the waiting factories flourished by efficient scheduling polices, should certainly be implanted within the database software stack. The parser should be extended in a way to understand and differentiate the streaming from the persistent data, and the different query types. Thus, transforming a passive data management system to an active one, seems a general method that consists of a few straightforward key steps, and can easily be applied to any extensible system.

However, one of the main difference between DataCell's underlying column-store kernel with other relational row-oriented DBMSs, is the core processing model that they obey. DataCell builds over a column-store kernel using, bulk processing instead of volcano-style pipelining execution model and vectorized query processing as opposed to tuple-based. It relies on operator-at-a-time bulk processing and materialization of all (narrow) intermediate result columns. DataCell adopts the column-at-a-time processing principle, adapting it to the streaming singularity. Thus, without waiting "forever" to fill in the (streaming attributes) columns with streaming data, it gets *as many data are available* into chunks when the triggering condition occurs and evaluates the query plans, in

a Volcano-style iteration. This logic, is quite different than the original tuple-at-a-time model where the individual operations are invoked separately for each tuple. However, this fundamental difference is not a prohibitive factor to proceed to the streaming transformation of a row-oriented DBMS; all we need is to simply be able to materialize intermediates for incremental processing and introduce a mechanism for batch processing.

The tuple-at-a-time model guarantees near real-time processing in a typical stream application; it immediately processes each tuple once it arrives. However, there is a drawback coming from the need to repeatedly call all operators. This can potentially affect scalability.

Our underlying column-store architecture constitutes a crucial feature to support DataCell's incremental processing requirement. Intermediates are also in column format. In this way, we did not need to change to original relational operators, since we keep in a natural way the required intermediate state of the partial operator evaluation into the corresponding intermediate columns/baskets inside each factory. The operators access only the newly appended streaming data and they merge the new results with the previous ones to update the result set. The key point is to be able to split the stream and then "freeze" and "resume" execution of a plan at the proper points.

Hence in a row-store implementation, the major extension required is to introduce intermediate result materialization for each operator that precedes a `concat` operation in the incremental plans. While this used to be considered an unbearable overhead, row-stores implement similar techniques for sharing intermediate results for multi-query optimization, and recently we have seen successful exploitation of intermediates in eddies (Deshpande and Hellerstein, 2004).

Other than design issues, using column-store or row-store as the underlying architecture comes with all the benefits or the overheads of the respective design. Row-stores and column-stores clearly represent the extremes of the database kernel architecture design space. For example, depending on the workload there may be less I/O and memory bandwidth requirements for a column-store but at the same time a row-store may have less requirements for intermediates materialization and thus less memory requirements. As such, another interesting direction for DataCell is the application of the DataCell philosophy in the more recent efforts that try to build hybrid database architectures. Again, the features of bulk processing, selective intermediates materialization and the ability to pause and resume execution, are all necessary for the core DataCell functionality.

6.5 Summary

DataCell makes the first steps towards a complete data management architecture that integrates database and stream functionalities in the same kernel. It fundamentally changes the way that stream data is handled and processed, trying to exploit many traditionally core database techniques and ideas. In this thesis, we made the strong statement that it is possible to implant stream processing functionalities in the heart of a modern database kernel and achieve both state of the art one-time query performance and stream query performance.

By relying on previous major efforts made from the database community during the last decades, we can bring several advantages on the stream processing front. So far, we made the first crucial steps to this direction. However, plenty of research challenges arise. The various open topics described in this chapter show the research path towards a fully integrated architecture where complex and hybrid stream-database scenarios will be expressed and performed. Overall, DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective and by taking into account the needs of modern data management for scalable stream processing combined with traditional query processing.

Bibliography

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E. F., Hatoun, M., Maskey, A., Rasin, A., Singer, A., Stonebraker, M., Tatbul, N., Xing, Y., Yan, R., and Zdonik, S. B. (2003a). Aurora: A Data Stream Management System. In *ACM SIGMOD International Conference on Management of Data*.

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

Abbott, R. and Garcia-Molina, H. (1989). Scheduling Real-time Transactions with Disk Resident Data. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Abbott, R. K. and Garcia-Molina, H. (1992). Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Trans. Database Syst.*, 17:513–560.

Abiteboul, S., Agrawal, R., Bernstein, P., Carey, M., Ceri, S., Croft, B., DeWitt, D., Franklin, M., Molina, H. G., Gawlick, D., Gray, J., Haas, L., Halevy, A., Hellerstein, J., Ioannidis, Y., Kersten, M., Pazzani, M., Lesk, M., Maier, D., Naughton, J., Schek, H., Sellis, T., Silberschatz, A., Stonebraker, M., Snodgrass, R., Ullman, J., Weikum, G., Widom, J., and Zdonik, S. (2005). The Lowell Database Research Self-Assessment. *Communications of the ACM*, 48(5):111–118.

Ali, M. H., Gerea, C., Raman, B. S., Sezgin, B., Tarnavski, T., Verona, T.,

Wang, P., Zabback, P., Kirilov, A., Ananthanarayan, A., Lu, M., Raizman, A., Krishnan, R., Schindlauer, R., Grabs, T., Bjeletich, S., Chandramouli, B., Goldstein, J., Bhat, S., Li, Y., Nicola, V. D., Wang, X., Maier, D., Santos, I., Nano, O., and Grell, S. (2009). Microsoft CEP Server and Online Behavioral Targeting. *Proc. VLDB Endow. (PVLDB)*, 2(2):1558–1561.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003). STREAM: The Stanford Stream Data Manager. In *ACM SIGMOD International Conference on Management of Data*.

Arasu, A., Cherniack, M., Galvez, E. F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004). Linear Road: A Stream Data Management Benchmark. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Avnur, R. and Hellerstein, J. M. (2000). Eddies: continuously adaptive query processing. *SIGMOD Record*, 29:261–272.

Babcock, B., Babu, S., Datar, M., and Motwani, R. (2003). Chain: Operator Scheduling for Memory Minimization in Data stream systems. In *ACM SIGMOD International Conference on Management of Data*.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Thomas, D. (2004). Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 13(4):333–353.

Babu, S. and Widom, J. (2004). StreaMon: An Adaptive Engine for Stream Query Processing. In *ACM SIGMOD International Conference on Management of Data*. ACM.

Balakrishnan, H., Balazinska, M., Carney, D., Centintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., and Zdonik, S. (2004). Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383.

Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lecluse, C., Pfeffer, P., Richard, P., and Velez, F. (1988). The Design and Implementation of O2. In *Lecture notes in Computer Science on Advances in Object-Oriented Database Systems*, pages 1–22. Springer-Verlag New York, Inc.

Boncz, P., Wilschut, A., and Kersten, M. (1998). Flattening an Object Algebra to Provide Performance. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. (2002). Monitoring Streams: A New Class of Data Management Applications. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein,

J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. (2003). TelegraphCQ: Continuous Data- flow Processing for an Uncertain World. In *Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR)*.

Chandrasekaran, S. and Franklin, M. J. (2002). Streaming Queries Over Streaming Data. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Chen, J., Dewitt, D. J., Tian, F., and Wang, Y. (2000). NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *ACM SIGMOD International Conference on Management of Data*.

CODASYL Data Description Language Committee (1973). CODASYL Data Description Language. *Journal of Development*.

Codd, A. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6).

Copeland, G. and Khoshafian, S. (1985). A Decomposition Storage Model. In *ACM SIGMOD International Conference on Management of Data*.

Coral8, I. (2007). http://www.coral8.com.

Cranor, C. D., Johnson, T., Spatscheck, O., and Shkapenyuk, V. (2003). Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD International Conference on Management of Data*.

Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M. J., Livny, M., and Jauhari, R. (1988). The HiPAC project: combining active databases and timing constraints. *SIGMOD Record*, 17:51–70.

Deshpande, A. and Hellerstein, J. M. (2004). Lifting the Burden of History from Adaptive Query Processing. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Dobra, A., Garofalakis, M., Gehrke, J., and Rastogi, R. (2002). Processing Complex Aggregate Queries over Data Streams. In *ACM SIGMOD International Conference on Management of Data*.

Eisenberg, A., Melton, J., Kulkarni, K. G., Michels, J.-E., and Zemke, F. (2004). SQL:2003 Has been published. *SIGMOD Record*, 33(1):119–126.

Eswaran, K. P. (1976). Aspects of a Trigger Subsystem in an Integrated Database System. In *Proc. of the Int'l. Conf. on Software engineering (ICSE)*.

Eswaran, K. P. and Chamberlin, D. D. (1975). Functional Specifications of a Subsystem for Data Base Integrity. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Fausto, P. B., Bernstein, P. A., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. (2002). Data Management for Peer-to-

Peer Computing: A Vision. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*.

Folding@home (2000). http://folding.stanford.edu. (last accessed on April 2011).

Franklin, M. J., Krishnamurthy, S., Conway, N., Li, A., Russakovsky, A., and Thombre, N. (2009). Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR)*.

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. (2008). SPADE: the system s declarative stream processing engine. In *ACM SIGMOD International Conference on Management of Data*.

Gedik, B. and Liu, L. (2003). PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System. In *Proceedings of International Conference on Distributed Computing Systems (ICDSC)*.

Gettier, E. (1963). Is justified true belief knowledge? *Analysis*, 23:121–123.

Ghanem, T. M. et al. (2007). Incremental Evaluation of Sliding Window Queries over Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):57–72.

Girod, L. et al. (2007). The Case for a Signal-Oriented Data Stream Management System. In *Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR)*.

Gnutella (2000). http://gnutella.wego.com. (last accessed on March 2010).

Golab, L. (2006). *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo.

Hanson, E. N. (1996). The Design and Implementation of the Ariel Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8(1):157–172.

Haritsa, J. R., Carey, M. J., and Livny, M. (1990). On Being Optimistic about Real-Time Constraints. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS)*.

Harizopoulos, S., Shkapenyuk, V., and Ailamaki, A. (2005). QPipe: a simultaneously pipelined relational query engine. In *ACM SIGMOD International Conference on Management of Data*.

Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. (2003). Querying the internet with PIER. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Idreos, S. (2010). *Database Cracking: Towards Auto-tuning Database Kernel*. PhD thesis, University of Amsterdam.

ISO-ANSI (1990). ISO-ANSI Working Draft: Database Language SQL2 and

SQL3. *ISO/IEC JTC1/SC21/WG3*, X3H2/90/398.

Ivanova, M., Kersten, M., Nes, N., and Goncalves, R. (2009). An Architecture for Recycling Intermediates in a Column-store. In *ACM SIGMOD International Conference on Management of Data*.

Jain, N., Amini, L., Andrade, H., and King, R. (2006). Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *ACM SIGMOD International Conference on Management of Data*.

Kang, J. et al. (2003). Evaluating window joins over unbounded streams. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Kao, B. and Garcia-Molina, H. (1993). An overview of real-time database systems. Technical Report 1993-6, Stanford University.

Karger, D. R. and Quan, D. (2005). What Would it Mean to Blog on the Semantic Web? *Journal of Web Semantics*, 3(2-3):147–157.

KazaA (2001). http://www.kazaa.com. (last accessed on June 2004).

Kersten, M., Idreos, S., Manegold, S., and Liarou, E. (2011). The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *Proc. VLDB Endow. (PVLDB)*, 4(4):1558–1561.

Kersten, M., Liarou, E., and Goncalves, R. (2007). A Query Language for a Data Refinery Cell. In *Proceedings of the International Workshop on Event Driven Architecture and Event Processing Systems (EDA-PS)*.

LHC (2010). Large Hadron Collider. http://lhc.web.cern.ch/lhc/.

LHC@home (2004). http://lhcathome.web.cern.ch. (last accessed on April 2011).

Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. (2005). No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Record*, 34(1):39–44.

Liarou, E., Goncalves, R., and Idreos, S. (2009). Exploiting the Power of Relational Databases for Efficient Stream Processing. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*.

Liarou, E., Idreos, S., Manegold, S., and Kersten, M. (2012a). Enhanced Stream Processing in a DBMS Kernel, Under Submition.

Liarou, E., Idreos, S., Manegold, S., and Kersten, M. (2012b). MonetDB/DataCell: Online Analytics in a Streaming Column-Store. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Liarou, E. and Kersten, M. L. (2009). Datacell: Building a data stream engine on top of a relational database kernel. In *VLDB PhD Workshop*.

Lim, H.-S., Lee, J.-G., Lee, M.-J., Whang, K.-Y., and Song, I.-Y. Continuous Query Processing in Data Streams Using Duality of Data and Queries. In *ACM SIGMOD International Conference on Management of Data*.

Linear Road Benchmark (2012). http://pages.cs.brandeis.edu/ linearroad/.

Loo, B. T., Hellerstein, J. M., Huebsch, R., Shenker, S., and Stoica, I. (2004). Enhancing P2P file-sharing with an internet-scale query processor. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

LSST (2010). Large Synoptic Survey Telescope. http://www.lsst.org.

Madden, S., Shah, M., and Hellerstein, J. M. (2002). Continuously Adaptive Continuous Queries over Streams. In *ACM SIGMOD International Conference on Management of Data*.

MonetDB (2012). http://monetdb.org/.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G. S., Olston, C., Rosenstein, J., and Varma, R. (2003). Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR)*.

Napster (1999). http://www.napster.com. (last accessed on April 2011).

Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmér, M., and Risch, T. (2002). EDUTELLA: a P2P networking infrastructure based on RDF. In *Proceedings of International Conference on World Wide Web (WWW)*.

Olle, A. (1978). *The CODASYL Approach to Data Base Management*. John Wiley and Sons, New York, USA, 1 edition.

Paton, N. W. and Díaz, O. (1999). Active Database Systems. *ACM Comput. Surv.*, 31:63–103.

Peterson, J. L. (1977). Petri Nets. *ACM Comput. Surv.*, 9(3):223–252.

PostgreSQL (2012). http://www.postgresql.org/.

Qiming, C. and Meichun, H. (2010). Experience in Extending Query Engine for Continuous Analytics. Technical Report TR-44, HP Laboratories.

Schreier, U. et al. (1991). Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Sellis, T., Lin, C., and Raschid, L. (1989). Data Intensive Production Systems: the DIPS Approach. *SIGMOD Record*, 18(3):52–58.

SETI@home (1999). http://setiathome.berkeley.edu. (last accessed on April 2011).

Sharaf, M. A., Chrysanthis, P. K., Labrinidis, A., and Pruhs, K. (2008). Algorithms and Metrics for Processing Multiple Heterogeneous Continuous Queries. *ACM Trans. Database Syst.*, 33(1):5:1–5:44.

Simon, B., Miklos, Z., Neijdl, W., Sintek, M., and Salvachua, J. (2003). Smart Space for Learning: A Mediation Infrastructure for Learning Services. In

Proceedings of International Conference on World Wide Web (WWW).

Stonebraker, M., Hanson, E., and Potamianos, S. (1988). The POSTGRES Rule Manager. *IEEE Transactions on Software Engineering*, 14:897–907.

Stonebraker, M., Hearst, M. A., and Potamianos, S. (1989). A Commentary on the POSTGRES Rule System. *SIGMOD Record*, 18(3):5–11.

StreamBase Systems, Inc (2003). http://www.streambase.com.

StreamSQL (2009). http://blogs.streamsql.org/.

Tatbul, E. N. (2007). *Load shedding techniques for data stream management systems*. PhD thesis, Providence, RI, USA. AAI3272068.

Widom, J. and Ceri, S., editors (1996). *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann.

Winter, R. and Kostamaa, P. (2010). Large scale data warehousing: Trends and observations. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Zdonik, S. B., Stonebraker, M., Cherniack, M., Çetintemel, U., Balazinska, M., and Balakrishnan, H. (2003). The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin*, 26(1):3–10.

Zhu, Y. and Shasha, D. (2002). Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Zloof, M. (1981). QBE/OBE: A Language for Office and Business Automation. *Computer*, 14:13–22.

Zloof, M. M. (1975). Query by Example. In *AFIPS National Computer Conference*.

Zloof, M. M. (1977). Query-by-Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343.

List of Figures

Summary

Numerous applications nowadays require online analytics over high rate streaming data. For example, emerging applications over mobile data can exploit the big mobile data streams for advertising and traffic control. In addition, the recent and continuously expanding massive cloud infrastructures require continuous monitoring to remain in good state and prevent fraud attacks. Similarly, scientific databases create data at massive rates daily or even hourly. In addition, web log analysis requires fast analysis of big streaming data for decision support.

The need to handle queries that remain active for a long time (continuous queries) and quickly analyze big data that are coming in a streaming mode and combine it with existing data brings a new processing paradigm that can not be exclusively handled by the existing database or data stream technology. Database systems do not have support for continuous query processing, while data stream systems are not built to scale for big data analysis. For this new problem we need to combine the best of both worlds.

In this thesis, we study how to design and implant streaming functionalities in modern column-stores which targets big data analytics. In particular, we use the open source column-store, MonetDB, as our design and experimentation platform. This includes exploitation of both the storage/execution engine and the optimizer infrastructure of the underlying DBMS. We investigate the opportunities and challenges that arise with such a direction and we show that it carries significant advantages. The major challenge then becomes the efficient support for specialized stream features such as incremental window-based processing as well as exploiting standard DBMS functionalities in a streaming environment.

We demonstrate that the resulting system, MonetDB/DataCell, achieves excellent stream processing performance by gracefully handling the state of the art stream benchmark, the Linear Road Benchmark. In addition, we demon-

strate that MonetDB/DataCell outperforms state of the art commercial stream management systems as the stream data increase. These results open the road for scalable data processing that combines both persistent and streaming data in an integrated environment in modern data warehouses.

Samenvatting

Vandaag de dag moeten online analytische programmas kunnen omgaan met een snelle stroom van gegevens. Bijvoorbeeld, toepassingen in de mobiele sector proberen de stroom van gegevens te gebruiken voor advertenties en routering. In dezelfde lijn vereisen grootschalige Cloud infrastructuren een continue monitoring om stabiliteit te waarborgen en cyberaanvallen te kunnen pareren. Wetenschappelijk databanken en web-log analyses vereisen een efficiente verwerking voor decision support.

Het afhandelen van langlevende queries (continuous queries) en het snel analyseren van grote data stromen in combinatie en vergelijking met reeds opgeslagen informatie kan nog niet goed met de bestaande database en streaming technologie worden uitgevoerd. Database systemen missen de functionaliteit voor verwerking van continuous queries en data streaming systemen schalen niet. Dit nieuwe probleem vereist een oplossing die de beste eigenschappen van beide werelden combineert.

In dit proefschrift wordt een ontwerp besproken hoe data stromen kunnen worden verwerkt in een modern kolom-georienteerde database systeem. In het bijzonder richten we ons hier op het open-source systeem MonetDB als platform voor ontwerp experimentatie. Het omvat aanpassingen in zowel het opslag deel, de verwerkingskern, als ook de optimizers. De mogelijkheden worden op een rij gezet en geanalyseerd om de beste richting te kunnen bepalen. De grootste winst wordt gehaald bij 'window-based' verwerking van de data stroom.

We laten aan de hand van de Linear Road Benchmark zien dat het prototype een uitstekende performance biedt en ook in vergelijking met state-of-the-art commerciele systemen. Deze resultaten maken het mogelijk om schaalbare gegevensverwerking te verkrijgen voor zowel stromende als persistente gegevens in een geintegreerde, moderne datawarehouse omgeving.

Acknowledgments

Here, I would like to take the opportunity to thank several people who have been of great help towards the successful completion of this thesis.

First, I would like to express my gratitude to my supervisor, Prof. Martin Kersten, for his guidance throughout my Ph.D. journey. Martin, is a very inspiring person and scientist who taught me at multiple levels and not only in the strict bounds of my thesis topic; the fact the he is still an active coder, gives him a unrivaled insight of system challenges. I would also like to thank Stefan Manegold, who became part of this effort about half way in my Ph.D journey. Stefan's ability for deep and careful analysis of every little detail showed me how to study topics leaving no stone unturned. Stratos Idreos is another person that was heavily involved in my thesis research and I am indebted to him for his help. Stratos is always optimistic; he can easily detect good ideas, opportunities and problems. Working with these three people, I earned a lot not only at the context of this thesis but in the broad context of research.

In addition, I would like to thank all the members of the Database group at CWI. Without the support of the core MonetDB team, this research would be very hard to complete, if not impossible. In particular, I would like to note the contribution of Niels Nes, who is always helpful, explaining and replying quickly to any question I was bugging him with. Romulo Goncalves, my officemate for the first couple of years, also contributed significantly to the construction of my initial topic.

Manolis Koubarakis is my first advisor at the Technical University of Crete for my undergraduate and my Master's thesis; he is the person that encouraged me the most to continue my Ph.D. studies. Already during my Master's studies, Manolis spent valuable resources to teach me how to construct and defend my first publishable work. Our collaboration and academic relation still continues and I am thankful that he also accepted to become a member of my Ph.D. committee.

During my Ph.D time, I spent four great months doing a research internship in the Stream Group of IBM Research, in Watson, New York. For this highly constructive and educative experience, I would like to thank Anton Riabov, Anand Ranganathan and Octavian Udrea.

Also, I would like to thank all these inspiring people I have met during international database conferences. Many members of the Greek database "mafia" provided an invaluable network and huge motivation.

I would also like to thank Tamer Özsu, Manolis Koubarakis, Lynda Hardman and Martine de Rijke for agreeing to be part of my PhD committee.

Finally, I am grateful to my family for their continuous support and sacrifices throughout my education path.

CURRICULUM VITAE

Education

10/2006 - now	PhD candidate CWI Database group, Amsterdam, The Netherlands Supervised by Martin Kersten
9/2004 - 9/2006	Master in Computer Engineering, 9.67/10 Department of Electronic and Computer Engineering Technical University of Crete, Greece Thesis: Distributed Evaluation of Conjunctive RDF Queries over Distributed Hash Tables, 10/10 Supervised by Manolis Koubarakis Committee: Vasilis Samoladas and Euripides Petrakis
9/1997 - 6/2003	Diploma in Electronic and Computer Engineering, 7.78/10 Department of Electronic and Computer Engineering Technical University of Crete, Greece Thesis: A Hybrid Peer-to-peer System with a Schema based Routing Strategy, 10/10 Supervised by Manolis Koubarakis Committee: Stavros Christodoulakis and Vasilis Samoladas

Employment & Academic Experience

10/2006 - 09/2012	Junior researcher Database group CWI Amsterdam, The Netherlands
7/2010 - 11/2010	Research intern IBM T. J. Watson Research Center Automated Component Assembly Middleware, Stream Group New York, USA
09/2004 - 09/2006	Research assistant in projects EVERGROW and ONTOGRID (EU 6th Framework Programme IST/FET)
9/2003 - 9/2006	Research assistant Intelligent Systems Laboratory Technical University of Crete, Greece
9/2005 - 2/2006	Teaching assistant Theory of Computation (Autumn 2005) Technical University of Crete, Greece

Publications

Refereed Conference Papers

(1) Stratos Idreos and Erietta Liarou. dbTouch: Analytics at your Fingertips. In Proceedings of the 7th International Conference on Innovative Data Systems Research (**CIDR**), Asilomar, California, USA, 2013.

(2) Erietta Liarou and Stratos Idreos. Too Many Links in the Horizon; What is Next? Linked Views and Linked History. In Proceedings of the 10th International Semantic Web Conference (**ISWC**), *Outrageous Ideas track*, Bonn, Germany, October 2011.

(3) Martin Kersten, Stefan Manegold, Stratos Idreos and Erietta Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in

Just a Few Seconds. In Proceedings of the Very Large Databases Endowment (**PVLDB**) and in the 37th VLDB Conference, Seattle, WA, August 2011. *Challenges and Visions best paper award.*

(4) Erietta Liarou, Romulo Goncalves and Stratos Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In Proceedings of the 19th International Conference on Extending Database Technology (**EDBT**), Saint-Petersburg, Russia March 2009

(5) Stratos Idreos, Erietta Liarou and Manolis Koubarakis. Continuous Multi-Way Joins over Distributed Hash Tables. In Proceedings of the 11th International Conference on Extending Database Technology (**EDBT**), Nantes, France, March 2008

(6) Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Continuous RDF Query Processing over DHTs. In Proceedings of the 6th International Semantic Web Conference (**ISWC**), Busan, Korea, November 2007

(7) Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In Proceedings of the 5th International Semantic Web Conference (**ISWC**), Athens, Georgia USA, November 2006

Refereed Demo Papers

(1) Erietta Liarou, Stratos Idreos, Stefan Manegold, Martin Kersten. MonetDB/DataCell: Online Analytics in a Streaming Column-Store In Proceedings of the 38th International Conference on Very Large Data Bases (**PVLDB**), Istanbul, Turkey, August 2012

Refereed Workshop Papers

(1) Erietta Liarou and Martin Kersten. DataCell: Building a Data Stream Engine on top of a Relational Database Kernel. In Proceedings of the 35th International Conference on Very Large Data Bases (**VLDB**), **PhD workshop**, Lyon, France, August 2009

(2) Martin Kersten, Erietta Liarou and Romulo Goncalves. A Query Language for a Data Refinery Cell. In Proceedings of the 2nd International workshop on Event-Driven Architecture, Processing and Systems (**EDA-PS**), in conjunction with VLDB, Vienna, Austria, September 2007

(3) Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Publish/Subscribe with RDF Data over Large Structured Overlay Networks. In Proceedings of the 3rd International Workshop on Databases, Information Systems and Peer- to-Peer Computing (**DBISP2P**) in conjunction with VLDB, Trondheim, Norway, August 2005

Book Chapters

(1) Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou, Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Semantic Grid Resource Discovery using DHTs in Atlas. In "Knowledge and Data Management in Grids", Talia Domenico, Bilas Angelos, and Dikaiakos Marios D.(editors), Springer, 2006.

National Events

(1) Participated with the EDBT 2009 paper "Exploiting the Power of Relational Databases for Efficient Stream Processing" in the 8th Hellenic Data Management Symposium, Athens, Greece, July 2009

(2) Participated with the ISWC 2006 paper "Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks" in the 6th Hellenic Data Management Symposium, Athens, Greece, July 2007

(3) Participated with the paper "DataCell: A column oriented data stream engine" in the Dutch-Belgian Database Day, Brussels, Belgium, November 2007

Ongoing Research

The following two topics is ongoing research in the context of DataCell, an architecture that extends a column-store DBMS kernel to provide stream processing.

(1) Incremental Processing in a Column-store.
In this work we investigate how to efficiently support core streaming functionalities such as *incremental stream processing* and *window-based processing* within a column-store database architecture for handling large volumes of streaming data.

(2) Indexing in a Streaming Column-Store.
In this work we investigate indexing and multi-query processing opportunities for large volume data streams in column-stores.

Honors and Awards

(1) **VLDB Best Paper Award in Challenges and Visions**
In 37th International Conference on Very Large Databases (VLDB), Seattle, USA, September 2011.
By the Computing Community Consortium, USA, 2011

(2) **SIGMOD 2012 Travel Award**
By NSF, SAP and the SIGMOD Executive Committee, 2012

Reviewing

Reviewer	TKDE
	Ad Hoc Networks
External Reviewer	VLDB 2008, 2009, 2010
	ICDE 2009, 2010
	EDBT 2008
	DEBS 2008
	SSDBM 2008
	WISE 2012

SIKS Dissertation Series

1998-1 Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM) Memory versus Search in Games

1998-5 E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR) Classification using decision trees and neural nets

1999-3 Don Beal (UM) The Nature of Minimax Search

1999-4 Jacques Penders (UM) The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU) Re-design of compositional systems

1999-7 David Spelt (UT) Verification support for object database design

1999-8 Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1 Frank Niessink (VU) Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE) Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval

2000-6 Rogier van Eijk (UU) Programming Languages for Agent Communication

2000-7 Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR) Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI) Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA) Learning as problem solving

2001-4 Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU) Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU) Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT) Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM) Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva) Processing XML in Database Systems

2002-13 Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT) Development and specification of virtual environments

2003-07 Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM) Repair Based Scheduling

2003-09 Rens Kortmann (UM) The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT) Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT) Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM) Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM) Learning Search Decisions

2004-01 Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT) Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR) Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politile gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU) Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT) Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU) Multi-Relational Data Mining

2004-16 Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM) Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)) AI techniques for the game of Go

2005-03 Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT) Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM) Adaptive Game AI

2005-07 Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR) Distributed Simulation in Industry

2005-13 Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU) Usability of XML Query Languages

2005-17 Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU) Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM) Situated Representation

2005-20 Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE) Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU) Building Web Service Ontologies

2006-05 Cees Pierik (UU) Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT) Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA) Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU) Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT) Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN) Aptness on the Web

2006-22 Paul de Vrieze (RUN) Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT) Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU) Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA) Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU) Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM) Context-Based Image Analysis

2007-15 Joyca Lacroix (UM) NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU) Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT) On the development an management of adaptive business collaborations

2007-19 David Levy (UM) Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU) Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE) Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI) Structural Features in XML Retrieval

2007-25 Joost Schalken (VU) Empirical Investigations in Software Process Improvement

2008-01 Katalin Boer-Sorbn (EUR) Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA) Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT) Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU) Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT) Discourse oriented summarization

2008-11 Vera Kartseva (VU) Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA) Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT) Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU) Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM) Adaptive Active Vision

2008-19 Henning Rode (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer

met de overheid op de administratieve lasten van bedrijven.

2008-21 Krisztian Balog (UVA) People Search in the Enterprise

2008-22 Henk Koning (UU) Communication of IT-Architecture

2008-23 Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU) Using background knowledge in ontology matching

2008-25 Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM) Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA) Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU) Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT) Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU) Understanding Classification

2009-07 Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services

2009-13 Steven de Jong (UM) Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT) New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT) Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI) Armada, An Evolving Database System

2009-19 Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM) Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI) "RAM: Array Database Management through Relational

Mapping"

2009-26 Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT) Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN) Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT) Digital Analysis of Paintings

2009-42 Toine Bogers (UvT) Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU) Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010-01 Matthijs van Leeuwen (UU) Patterns that Matter

2010-02 Ingo Wassink (UT) Work flows in Life Science

2010-03 Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT) Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT) A Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL) Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU) Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU) Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions

2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 Marten Voulon (UL) Automatisch contracteren

2010-28 Arne Koopman (UU) Characteristic Relational Patterns

2010-29 Stratos Idreos (CWI) Database Cracking: Towards Auto-tuning Database Kernels

2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval

2010-31 Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web

2010-32 Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems

2010-33 Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 Teduh Dirgahayu (UT) Interaction Design in Service Compositions

2010-35 Dolf Trieschnigg (UT) Proof of Concept: Concept-based Biomedical Information Retrieval

2010-36 Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification

2010-37 Niels Lohmann (TUE) Correctness of services and their composition

2010-38 Dirk Fahland (TUE) From Scenarios to components

2010-39 Ghazanfar Farooq Siddiqui (VU) Integrative modeling of emotions in virtual agents

2010-40 Mark van Assem (VU) Converting and Integrating Vocabularies for the Semantic Web

2010-41 Guillaume Chaslot (UM) Monte-Carlo Tree Search

2010-42 Sybren de Kinderen (VU) Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach

2010-43 Peter van Kranenburg (UU) A Computational Approach to Content-Based Retrieval of Folk Song Melodies

2010-44 Pieter Bellekens (TUE) An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain

2010-45 Vasilios Andrikopoulos (UvT) A theory and model for the evolution of software services

2010-46 Vincent Pijpers (VU) e3alignment: Exploring Inter-Organizational Business-ICT Alignment

2010-47 Chen Li (UT) Mining Process Model Variants: Challenges, Techniques, Examples

2010-48 Milan Lovric (EUR) Behavioral Finance and Agent-Based Artificial Markets

2010-49 Jahn-Takeshi Saito (UM) Solving difficult game positions

2010-50 Bouke Huurnink (UVA) Search in Audiovisual Broadcast Archives

2010-51 Alia Khairia Amin (CWI) Understanding and supporting information seeking tasks in multiple sources

2010-52 Peter-Paul van Maanen (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention

2010-53 Edgar Meij (UVA) Combining Concepts and Language Models for Information Access

2011-01 Botond Cseke (RUN) Variational Algorithms for Bayesian Inference in Latent Gaussian Models

2011-02 Nick Tinnemeier(UU) Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

2011-03 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems

2011-04 Hado van Hasselt (UU) Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference learning algorithms

2011-05 Base van der Raadt (VU) Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

2011-06 Yiwen Wang (TUE) Semantically-Enhanced Recommendations in Cultural Heritage

2011-07 Yujia Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction

2011-08 Nieske Vergunst (UU) BDI-based Generation of Robust Task-Oriented Dialogues

2011-09 Tim de Jong (OU) Contextualised Mobile Media for Learning

2011-10 Bart Bogaert (UvT) Cloud Content Contention

2011-11 Dhaval Vyas (UT) Designing for Awareness: An Experience-focused HCI Perspective

2011-12 Carmen Bratosin (TUE) Grid Architecture for Distributed Process Mining

2011-13 Xiaoyu Mao (UvT) Airport under Control. Multiagent Scheduling for Airport Ground Handling

2011-14 Milan Lovric (EUR) Behavioral Finance and Agent-Based Artificial Markets

2011-15 Marijn Koolen (UvA) The Meaning of Structure: the Value of Link Evidence for Information Retrieval

2011-16 Maarten Schadd (UM) Selective Search in Games of Different Complexity

2011-17 Jiyin He (UVA) Exploring Topic Structure: Coherence, Diversity and Relatedness

2011-18 Mark Ponsen (UM) Strategic Decision-Making in complex games

2011-19 Ellen Rusman (OU) The Mind ' s Eye on Personal Profiles

2011-20 Qing Gu (VU) Guiding service-oriented software engineering - A view-based approach

2011-21 Linda Terlouw (TUD) Modularization and Specification of Service-Oriented Systems

2011-22 Junte Zhang (UVA) System Evaluation of Archival Description and Access

2011-23 Wouter Weerkamp (UVA) Finding People and their Utterances in Social Media

2011-24 Herwin van Welbergen (UT) Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

2011-25 Syed Waqar ul Qounain Jaffry (VU) Analysis and Validation of Models for Trust Dynamics

2011-26 Matthijs Aart Pontier (VU) Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots

2011-27 Aniel Bhulai (VU) Dynamic website optimization through autonomous management of design patterns

2011-28 Rianne Kaptein(UVA) Effective Focused Retrieval by Exploiting Query Context and Document Structure

2011-29 Faisal Kamiran (TUE) Discrimination-aware Classification

2011-30 Egon van den Broek (UT) Affective Signal Processing (ASP): Unraveling the mystery of emotions

2011-31 Ludo Waltman (EUR) Computational and Game-Theoretic Approaches for Modeling Bounded Rationality

2011-32 Nees-Jan van Eck (EUR) Methodological Advances in Bibliometric Mapping of Science

2011-33 Tom van der Weide (UU) Arguing to Motivate Decisions

2011-34 Paolo Turrini (UU) Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations

2011-35 Maaike Harbers (UU) Explaining Agent Behavior in Virtual Training

2011-36 Erik van der Spek (UU) Experiments in serious game design: a cognitive approach

2011-37 Adriana Burlutiu (RUN) Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference

2011-38 Nyree Lemmens (UM) Bee-inspired Distributed Optimization

2011-39 Joost Westra (UU) Organizing Adaptation using Agents in Serious Games

2011-40 Viktor Clerc (VU) Architectural Knowledge Management in Global Software Development

2011-41 Luan Ibraimi (UT) Cryptographically Enforced Distributed Data Access Control

2011-42 Michal Sindlar (UU) Explaining Behavior through Mental State Attribution

2011-43 Henk van der Schuur (UU) Process Improvement through Software Operation Knowledge

2011-44 Boris Reuderink (UT) Robust Brain-Computer Interfaces

2011-45 Herman Stehouwer (UvT) Statistical Language Models for Alternative Sequence Selection

2011-46 Beibei Hu (TUD) Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work

2011-47 Azizi Bin Ab Aziz(VU) Exploring Computational Models for Intelligent Support of Persons with Depression

2011-48 Mark Ter Maat (UT) Response Selection and Turn-taking for a Sensitive Artificial Listening Agent

2011-49 Andreea Niculescu (UT) Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012-01 Terry Kakeeto (UvT) Relationship Marketing for SMEs in Uganda

2012-02 Muhammad Umair(VU) Adaptivity, emotion, and Rationality in Human and Ambient Agent Models

2012-03 Adam Vanya (VU) Supporting Architecture Evolution by Mining Software Repositories

2012-04 Jurriaan Souer (UU) Development of Content Management System-based Web Applications

2012-05 Marijn Plomp (UU) Maturing Interorganisational Information Systems

2012-06 Wolfgang Reinhardt (OU) Awareness Support for Knowledge Workers in Research Networks

2012-07 Rianne van Lambalgen (VU) When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions

2012-08 Gerben de Vries (UVA) Kernel Methods for Vessel Trajectories

2012-09 Ricardo Neisse (UT) Trust and Privacy Management Support for Context-Aware Service Platforms

2012-10 David Smits (TUE) Towards a Generic Distributed Adaptive Hypermedia Environment

2012-11 J.C.B. Rantham Prabhakara (TUE) Process Mining in the Large: Preprocessing, Discovery, and Diagnostics

2012-12 Kees van der Sluijs (TUE) Model Driven Design and Data Integration in Semantic Web Information Systems

2012-13 Suleman Shahid (UvT) Fun and Face: Exploring non-verbal expressions of emotion during playful interactions

2012-14 Evgeny Knutov(TUE) Generic Adaptation Framework for Unifying Adaptive Web-based Systems

2012-15 Natalie van der Wal (VU) Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes

2012-16 Fiemke Both (VU) Helping people by understanding them - Ambient Agents supporting task execution and depression treatment

2012-17 Amal Elgammal (UvT) Towards a Comprehensive Framework for Business Process Compliance

2012-18 Eltjo Poort (VU) Improving Solution Architecting Practices

2012-19 Helen Schonenberg (TUE) What's Next? Operational Support for Business Process Execution

2012-20 Ali Bahramisharif (RUN) Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing

2012-21 Roberto Cornacchia (TUD) Querying Sparse Matrices for Information Retrieval

2012-22 Thijs Vis (UvT) Intelligence, politie en veiligheidsdienst: verenigbare grootheden?

2012-23 Christian Muehl (UT) Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction

2012-24 Laurens van der Werff (UT) Evaluation of Noisy Transcripts for Spoken Document Retrieval

2012-25 Silja Eckartz (UT) Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application

2012-26 Emile de Maat (UVA) Making Sense of Legal Text

2012-27 Hayrettin Gürkök (UT) Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games

2012-28 Nancy Pascall (UvT) Engendering Technology Empowering Women

2012-29 Almer Tigelaar (UT) Peer-to-Peer Information Retrieval

2012-30 Alina Pommeranz (TUD) Designing Human-Centered Systems for Reflective Decision Making

2012-31 Emily Bagarukayo (RUN) A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure

2012-32 Wietske Visser (TUD) Qualitative multi-criteria preference representation and reasoning

2012-33 Rory Sie (OUN) Coalitions in Cooperation Networks (COCOON)

2012-34 Pavol Jancura (RUN) Evolutionary analysis in PPI networks and applications

2012-35 Evert Haasdijk (VU) Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics

2012-36 Denis Ssebugwawo (RUN) Analysis and Evaluation of Collaborative Modeling Processes

2012-37 Agnes Nakakawa (RUN) A Collaboration Process for Enterprise Architecture Creation

2012-38 Selmar Smit (VU) Parameter Tuning and Scientific Testing in Evolutionary Algorithms

2012-39 Hassan Fatemi (UT) Risk-aware design of value and coordination networks

2012-40 Agus Gunawan (UvT) Information Access for SMEs in Indonesia

2012-41 Sebastian Kelle (OU) Game Design Patterns for Learning

2012-42 Dominique Verpoorten (OU) Reflection Amplifiers in self-regulated Learning

2012-44 Anna Tordai (VU) On Combining Alignment Techniques

2012-45 Benedikt Kratz (UvT) A Model and Language for Business-aware Transactions

2012-46 Simon Carter (UVA) Exploration and Exploitation of Multilingual Data for Statistical Machine Translation

2012-47 Manos Tsagkias (UVA) Mining Social Media: Tracking Content and Predicting Behavior

2012-48 Jorn Bakker (TUE) Handling Abrupt Changes in Evolving Time-series Data

2012-49 Michael Kaisers (UM) Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions