MonetDB/DataCell: leveraging the column-store database technology for efficient and scalable stream processing
Liarou, E.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
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 flatfile 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:

\[
\text{ON } \text{<command list> CALL <database procedure>}
\]

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 mechanisms, 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 (Eswaren and Chamberlin, 1975; Eswaren, 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).
CHAPTER 2. BACKGROUND AND RELATED WORK

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
2.1. **FIRST STEPS TOWARDS REAL-TIME PROCESSING**

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
2.4. THE NEW ERA OF DATA STREAM MANAGEMENT SYSTEMS

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.
CHAPTER 2. BACKGROUND AND RELATED WORK

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.
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 PostegreSQL 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 introduced 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 $\text{[Range } T\text{]}$. A tuple-based sliding
window of size $N$ on a stream $S$ is specified by following the reference to $S$ in the query with $[\text{Rows } N]$.

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-
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.
2.7. THE MONETDB SYSTEM

The MonetDB Storage Model

In MonetDB, every n-ary relational table is represented as a collection of Binary Association Tables called BATs (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 $(\text{key}, \text{attr})$ 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.
CHAPTER 2. BACKGROUND AND RELATED WORK

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
selection, both $Ra_1$ and $Rb_1$ 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 $\text{algebra.project}(Rc,Ra_2)$ returns all $key$-attr pairs residing in base BAT $Rc$ at the positions specified by $Ra_2$. This is a tuple reconstruction operation. Iterating over $Ra_2$, it uses cache-friendly in-order positional lookups into $Ra_2$.

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.
CHAPTER 2. BACKGROUND AND RELATED WORK

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

\[ \text{Rel1} := \text{algebra.join}(Ra1, Rb1); \]

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.