View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CWI's Institutional Repository

Exploiting the Power of Relational Databases for Efficient Stream Processing

Erietta Liarou Romulo Goncalves Stratos Idreos
CWI Amsterdam, The Netherlands
{erietta,goncalve,idreos}@cwi.nl

ABSTRACT

Stream applications gained significant popularity over the last years that lead to the development of specialized stream engines. These systems are designed from scratch with a different philosophy than nowadays database engines in order to cope with the stream applications requirements. However, this means that they lack the power and sophisticated techniques of a full fledged database system that exploits techniques and algorithms accumulated over many years of database research.

In this paper, we take the opposite route and design a stream engine directly on top of a database kernel. Incoming tuples are directly stored upon arrival in a new kind of system tables, called baskets. A continuous query can then be evaluated over its relevant baskets as a typical one-time query exploiting the power of the relational engine. Once a tuple has been seen by all relevant queries/operators, it is dropped from its basket. A basket can be the input to a single or multiple similar query plans. Furthermore, a query plan can be split into multiple parts each one with its own input/output baskets allowing for flexible load sharing query scheduling. Contrary to traditional stream engines, that process one tuple at a time, this model allows batch processing of tuples, e.g., query a basket only after x tuples arrive or after a time threshold has passed. Furthermore, we are not restricted to process tuples in the order they arrive. Instead, we can selectively pick tuples from a basket based on the query requirements exploiting a novel query component, the basket expressions.

We investigate the opportunities and challenges that arise with such a direction and we show that it carries significant advantages. We propose a complete architecture, the DataCell, which we implemented on top of an open-source column-oriented DBMS. A detailed analysis and experimental evaluation of the core algorithms using both micro benchmarks and the standard Linear Road benchmark demonstrate the potential of this new approach.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.
EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.
Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...$5.00

1. INTRODUCTION

Data Stream Management Systems (DSMSs) have become an active research area in the database community. The motivation comes from a potentially large application domain, e.g., network monitoring, sensor networks, telecommunications, financial, web applications, etc.

In a stream application, we need mechanisms to support *long-standing/continuous* queries over data that is continuously updated from the environment. This requirement is significantly different than what happens in a relational DBMS where data is stored in static tables and then users fire *one-time* queries to be evaluated *once* over the existing data. Furthermore, a stream scenario brings a number of unique query processing challenges. For example, in order to achieve continuously high performance, the system needs to cope with (and exploit) similarities between the many standing queries, adapt to the continuously changing environment and so on.

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 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., [4, 6, 7, 8, 9, 10].

However, there is drawback with this direction. By designing completely different architectures from scratch, very little of the existing knowledge and techniques of relational databases can be exploited. This became more clear as the stream applications demanded more functionality. In this work, we start at the other end of the spectrum. We study the direction of building an efficient data stream management system on top of an extensible database kernel. With a careful design, this allows us to directly reuse all sophisticated algorithms and techniques of traditional DBMSs. We can provide support for any kind of complex functionality without having to reinvent solutions and algorithms for problems and cases with a rich database literature. Furthermore, it allows for more flexible and efficient query processing by allowing *batch processing* of stream tuples as well as *out-of-order* processing by selectively picking the tuples to process using *basket expressions*.

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 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. The above description is naturally an oversimplified one as this direction allows the exploration of quite flexible strategies. For example, throwing the same tuple into multiple different baskets where multiple queries are waiting, split query plans into multiple parts and share baskets between similar operators (or groups of operators) of different queries allowing reuse of results and so on. The query processing scheme follows the Petri-net model [18], 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.

Some questions that immediately arise are the following:

– How efficient can continuous query processing be?
– 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 paper 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 new research direction is feasible and that it can bring significant advantages. We carefully carve the research space and we discuss the opportunities and the challenges that come with this approach. It is a widely open research direction.

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

Furthermore, the DataCell introduces the following two new concepts that can be of significant importance in terms of both performance and query expressiveness.

Predicate windows. By having tuples "waiting" to be queried in the baskets, the DataCell enables applications to selectively process the stream and prioritize event processing based on application semantics. The system does not need to process tuples in the order they arrive. Instead it can "select" part of the basket tuples allowing for more expressive queries. It generalizes the sliding window approach predominant in DSMSs to allow for arbitrary table expressions over multiple streams and persistent tables interchangeably. We do not resort to a redefinition of the WINDOW concept. Instead, we propose an orthogonal extension to SQL'03.

Batch processing. Collecting incoming tuples into baskets brings the opportunity to first collect a number of tuples and only then process the tuples in one go. This way, when the application scenario allows it, the DataCell processing engine can exploit batch processing of events to amortize overhead incurred by process management and function calls. This favors a skewed arrival distribution, where a peak load can be handled easily, and possibly within the same time frame, as an individual event.

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.

The remainder of the paper is organized as follows. In Section 2, we present the necessary background knowledge followed by a detailed introduction of the architecture at large in Section 3. Then, Section 4 discusses the query processing model and pinpoints on the wide open research possibilities. Section 5 explores the scope of the solution by modeling stream-based application concepts borrowed from dedicated stream database systems. Section 6 provides an experimental analysis using the Linear Road benchmark. Finally, in Section 7 we discuss related work and Section 8 concludes the paper and outlines future work.

2. BACKGROUND

This section briefly presents the experimentation platform, MonetDB, and the basics of the Petri-net model.

2.1 A Column-oriented DBMS

MonetDB is a full fledged column-store engine. Every relational table is represented as a collection of *Binary Association Tables* (*BATs*). Each BAT is a set of two columns, called *head* and *tail*. 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`. Key values form a dense ascending sequence representing the *position* of an attribute value in the column. Thus, for base BATs, the key column typically 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.

The system is designed as a virtual machine architecture with an assembly language, called MAL. Each MAL operator wraps a highly optimized relational primitive. The interested reader can find more details on MonetDB in [17].

2.2 The Petri-net model

We now continue with a short description of the Petri-nets model [18]. 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.

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. We will show that the Petri-net model and abstraction nicely fit the continuous query paradigm on top of a DBMS.

3. THE DATACELL ARCHITECTURE

In this section, we discuss the DataCell architecture. It is built 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 step by step build up the architecture and the possible research directions. It 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 the next section.

3.1 Receptors and Emitters

The periphery of a stream engine is formed by *adapters*, e.g., software components to interact with devices, 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 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. There can be multiple emitters each one responsible for delivering a different result to one or multiple clients.

Figure 1 demonstrates a simple interaction model between the DataCell components where a receptor and an emitter can be seen at the edges of the system listening to streams and delivering results, respectively. The interchange format between the various components is purposely kept simple using a textual interface for exchanging flat relational tuples.

3.2 Baskets

The basket is the key data structure of the DataCell. Its role is to hold a *portion* of a 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 in baskets, we can evaluate the relevant continuous queries on top of these baskets. This way,

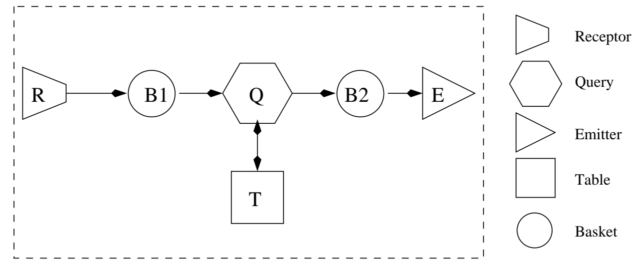

Figure 1: The DataCell model

instead of throwing each incoming tuple against its relevant queries, the DataCell does exactly the opposite by first collecting the data and then throwing the queries against the data. 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 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 "consumed" by all relevant continuous queries. This way, the baskets initiate the data flow in the stream engine.

The important differences between baskets and relational tables are summarized 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 or 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.

Another important opportunity, with baskets as the central concept, is that we purposely step away from the defacto 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 [1], regulation of concurrent writes on the same stream, etc. It unnecessary 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 by allowing the system to process groups of tuples at a time.

Realizing the DataCell approach on top of a column-oriented architecture allows for even more flexibility. A basket b in MonetDB becomes a BAT (column) holding values for a single attribute A of an incoming stream. Each entry in

b holds a value of A along with a key that identifies the relational tuple in which this attribute value belongs to (see Section 2). For each relational table there exists an extra column, the timestamp column, that for each tuple it reflects the time that this tuple entered the system.

This way, we can 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 table. 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. On the other hand, in DataCell, we exploit the column-oriented structure and bind each query only to the attributes/baskets it is interested in, utilizing the available hardware to the maximum. 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 where each table contains multiple attributes while not all queries need all of them.

3.3 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 queries. For now assume for simplicity that each factory contains the full query plan of a single query. Later on we discuss in detail the opportunities that arise.

Each factory has at least one *input* and one *output* basket. It continuously reads data from the input baskets, it processes this data and creates a result which it then places in its output baskets. Each time a tuple t is being consumed from an input basket b, the factory removes t from b to avoid reading it again. We revisit these choices later on, when we discuss more complex processing schemes.

Having introduced all basic DataCell components, we can now consider them at a higher level using Figure 1 as an example. A receptor captures incoming tuples and places them in Basket B_1. Then, a factory, containing the full query plan of a continuous query, processes the data in B_1 and places all qualifying tuples in Basket B_2 where the emitter can finally collect the result and deliver it to the client.

In general, at any point in time, multiple receptors wait for incoming tuples and place them into the proper 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. Every single component is an independent thread and data streams through the threads connected by baskets.

Let us now describe factories in more detail. A factory is a function containing a set of MAL operators corresponding in the query plan of a given continuous query. A factory is specified as an ordinary function. The difference is that its execution state is *saved* between calls. The first time that the factory is called, a thread is created in the local system to handle subsequent requests. A factory is called by the scheduler (to be discussed below). 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 a factory for a simple query. The factory contains

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

```
 1: input = basket.bind(X)
 2: output = basket.bind(Y)
 3: while true do
 4:    basket.lock(input)
 5:    basket.lock(output)
 6:    result = monetdb.select(input,v1,v2)
 7:    basket.empty(input)
 8:    basket.append(output,result)
 9:    basket.unlock(input);
10:    basket.unlock(output);
11:    suspend();
12: end while
```

the full query plan (just a single operator in this case in line 6) where the original MonetDB commands are being used.

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. Careful management of the baskets ensures that one factory, receptor or emitter at a time updates a given basket. This way, as seen in Algorithm 1, the loop of the factory begins by acquiring locks on the relevant input and output baskets. 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 seen tuples 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.

3.4 Basket Expressions and Predicate Windows

Having discussed the basic building blocks of the DataCell, we now proceed with the introduction of the *basket expressions* that allow us to process *predicate windows* on a stream. They allow for more flexible/expressive queries by selectively picking the tuples to process from a basket. Every continuous query contains a basket expression. In fact, basket expressions may be part only of continuous queries, which allows the system to distinguish between continuous and normal/one-time queries.

A basket expression encompasses the traditional SELECT-FROM-WHERE-GROUPBY SQL language framework. It is syntactically a sub-query surrounded by square brackets. However, the semantics is quite different. Basket expressions have side-effects; they change the underlying tables, i.e., baskets, during query evaluation. All tuples referenced in a basket expression are *removed* from their underlying store automatically. This leaves a partially emptied basket behind. A basket can also be inspected outside a basket expression. Then, it behaves as any (temporary) table, i.e., tuples are not removed. Continuous queries q_1 and q_2 below demonstrate example usages of the basket expressions.

```
(q1) select * from [select * from R] as S
     where S.a > v1

(q2) select * from [select * from R where R.b<v2] as S
     where S.a >v1
```

In Query q_1, the basket expression requests all tuples from the relevant stream/basket R. All tuples selected are *immediately* removed from R, but they remain accessible through

S during the remainder of the query execution. From this temporary table S, we select the payloads satisfying the predicate. This query represents a typical continuous query where all tuples are considered.

On the other hand, in Query q_2 the basket expression sets a restriction by filtering stream tuples before the actual continuous query considers them. This restriction sets a *predicate window*, i.e., the query will continuously evaluate only the tuples that fall in the predicate window as defined by the basket expression. This effect is similar to the SQL WINDOW construct. However, the semantics is richer and more flexible.

Most DSMSs perform query processing over streams seen as a linear ordered list. This naturally leads to a sequence of operators, such as NEXT, FOLLOWS, and WINDOW expressions. The latter overloads the semantics of the SQL WINDOW construct to designate a portion of interest around each tuple in the stream. Early DSMS designs liberally extended the SQL WINDOW function to capture part of a stream, e.g., a window can be defined as a fixed sized stream fragment, a time-bounded stream fragment, or a value-bounded stream fragment only. However, in SQL'03 window semantics have been made explicit and overloading it for stream processing introduces several problems, e.g., windows 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.

The basket expressions provide a more elegant and richer ground to designate windows of interest. They can be limited in size using result set constraints, they can be explicitly defined by predicates over their content, and they can be based on predicates referring to objects in enclosing query blocks or elsewhere in the database. Their syntax and semantics seamlessly fit in an existing SQL software stack. Details of the DataCell language are presented in [14].

4. 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 continuous query processing performance. In addition, we discuss further alternative directions that open the road for challenging research opportunities.

4.1 Processing Model

The DataCell architecture uses the abstraction of the Petri-net model to facilitate continuous query processing. 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.

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. 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, i.e., 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. The scheduler continuously re-evaluates the input of all transitions.

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 the 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 queries that check the input for the window properties. The latter is how we handle window queries in the Linear Road benchmark.

Furthermore, in certain queries, e.g., sliding window queries, the system does not remove all seen tuples from input baskets . Instead, it removes only the tuples that given the query do not qualify for the next window.

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. Taking again an example from the Linear Road benchmark, assume 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.

Numerous research opportunities arise under the DataCell processing model. In this paper, our goal is to provide the motivation and description of the system at large following the basic approach while detailed analysis and optimization of the multiple possible ways on how and when baskets interact with factories depending on query and system properties is left for future work.

4.2 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 the approaches considered in this paper to validate the feasibility of the DataCell approach and subsequently we point to further chal-

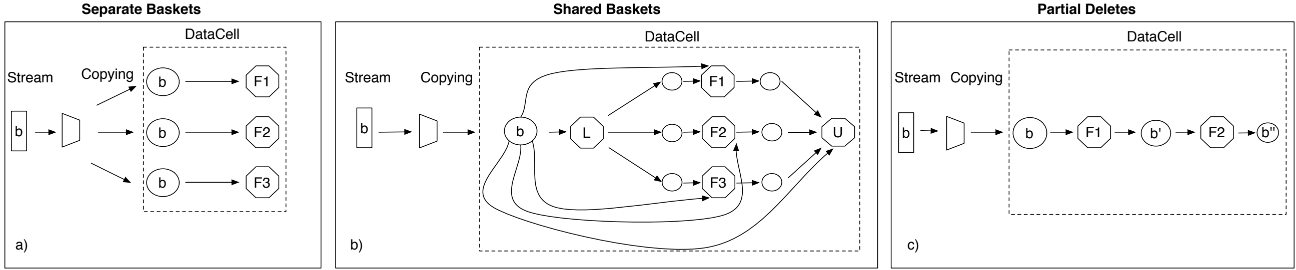

Figure 2: Examples of alternative processing schemes

lenging directions.

Separate Baskets. Our first strategy, called *separate baskets*, provides the maximum independence to each query and stream. Each continuous query is fully encapsulated within a single factory. Furthermore, each factory f has its own input baskets that are read *only* by f. 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. On the other hand, the benefit is that the factories can run completely independently without the need to carefully schedule their accesses on the baskets. An example is given in Figure 2(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. This way, if a factory is interested in two attributes A, B of stream R, then we need to copy in its baskets only the columns A and B and not the full tuples of R containing all attributes of the stream.

Shared Baskets. The first strategy, described above, is a very generic one that allows us to study the properties and potential of the 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 the proper baskets. In order to guarantee correct and complete results, the next step is to regulate the way the factories access their input baskets such that a tuple remains in its basket until all relevant factories have seen it. Thus, this 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 basket expression of the respective query.

To achieve the above goal, for every basket b shared as input between a group of k factories, we add two factories, as seen in Figure 2(b), the *locker* and the *unlocker* factories. The locker factory, L, is placed between b and the relevant factories. Once b contains a number of new tuples, L runs. Its task is to simply lock b. The output of L is k baskets, one for each waiting factory. In each one of these outputs, L 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 has an extra output basket where it writes a single bit attribute to mark that its execution is over. These output baskets are inputs to the unlocker factory U. The task of U is that once all factories are finished, i.e., once all output baskets are marked, it removes from b all tuples covered by the basket expressions of the factories, and subsequently it unlocks b so that the receptor can place new tuples.

Using this simple scheme, we can easily have shared baskets and exploit common query interests. It nicely shows that the DataCell model is very generic and flexible. Furthermore opportunities may come by exploiting recent techniques and ideas for sharing retrieval and execution costs of concurrent queries in databases [11].

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 2(c) shows an example where three queries create such a chain. This strategy opens the road for even more advanced ways of exploiting query commonalities.

4.3 Research Directions

The previous subsection introduced a number of different processing strategies and demonstrated the flexibility of the DataCell model. The goal of this paper is not to propose the ultimate processing scheme. We introduce the DataCell model and argue that it is a very 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 which we believe can create a stream of very interesting work.

The most challenging directions 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 lightweight query

q_1 and a quite 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 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.

Due to space restrictions we leave further analysis of the possible directions for future work. In the rest of the paper, we show that even the basic directions seen here bring high performance, opening the road for exciting future research.

5. QUERYING STREAMS

In this section, we illustrate how the key features of a stream query language are handled in the DataCell model using StreamSQL [20], as a frame of reference.

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 table.

```
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.

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];
```

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;
```

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 stream 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 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 simulated 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  select * from HB
        where X.tag < max(select tag from HB)]
```

6. 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.

6.1 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. 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.

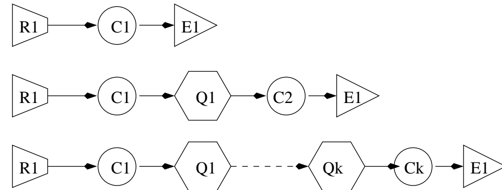

Figure 3: The Query Chain topology

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.

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 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 the 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 4(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.

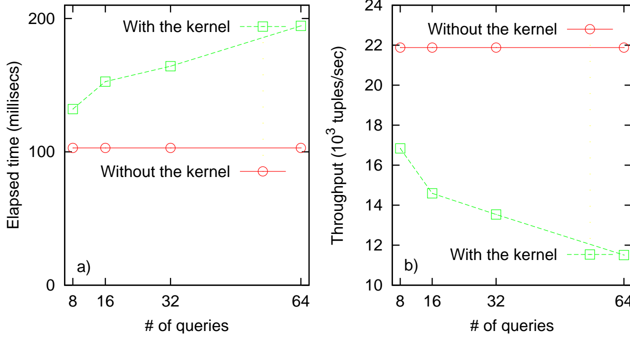

Figure 4: Effect of inter-process communication

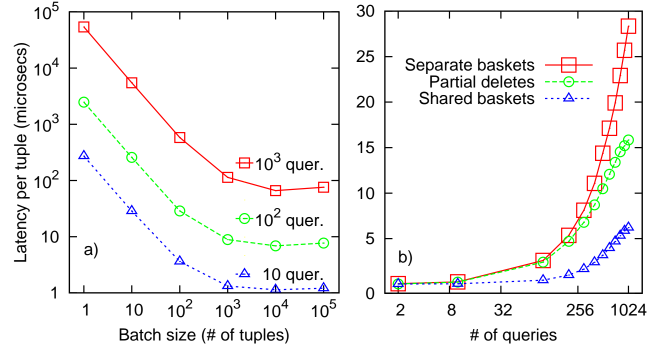

Figure 5: Effect of batch processing and strategies

In addition, Figure 4(b) shows. that the maximum throughput we can 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 stream engines does not yet provide many points of reference. GigaScope [9] claims a peak performance up to a million events per second by pushing down selection conditions into the Network Interface Controller. 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 [3]. Two solutions for systemX are given, one based on triggers and stored procedures, and another one based on polling.

However, all papers on 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 $[0, 10^4)$ and a timestamp that reflects its creation time. All queries are single stream, continuous queries of the following form.

```
Select * From [Select S.A From S Where v1<S.A<v2] as Z
```

All queries select a random range with 0.1% selectivity. Figure 5(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 4.2. 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 $T = 10^5$.

Figure 5(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.

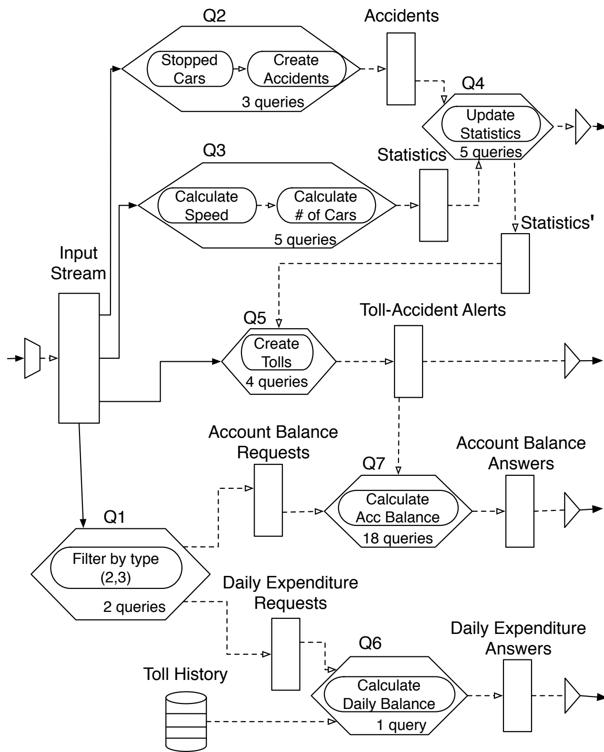

Figure 6: Linear Road benchmark in DataCell

6.2 The Linear Road Benchmark

In this section, we analyze the performance of our system using the Linear Road benchmark [3]. This is the only 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 paper, 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 roads. The system is responsible to monitor the position of each car. It continuously calculates and reports to each car the tolls it needs to pay and whether there is an accident that might affect it. An accident is detected when two or more cars are in the same position for 4 continuous timestamps. 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 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 with basket expressions. In particular there are 38 queries, logically distinguished in 7 different collections ($Q1$-$Q7$). Figure 6 gives a high level view of the various collections and the number of queries within each one. Due to space restrictions, we cannot describe in detail all 38 queries. 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 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 6, 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 7 shows the performance during the whole duration of the benchmark (three hours) for scale factor 1. Graph 7(a) shows the total number of tuples entered the

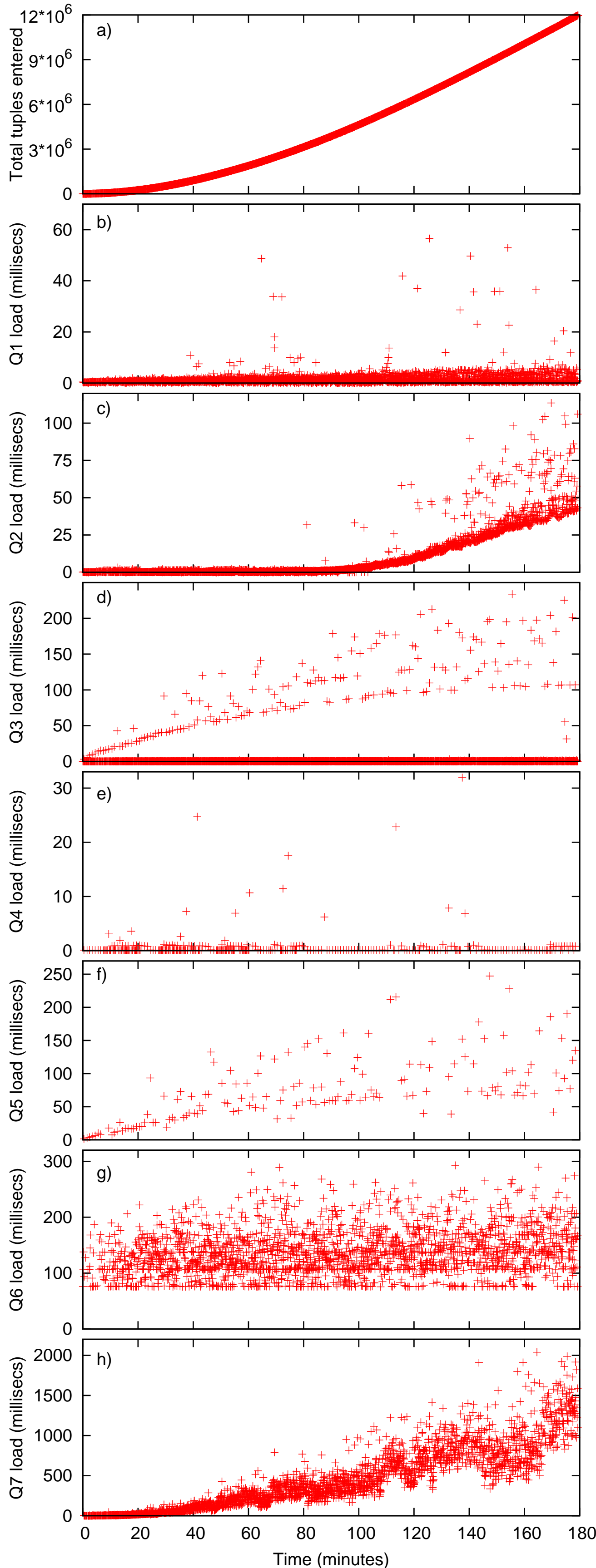

Figure 7: System load for each query collection

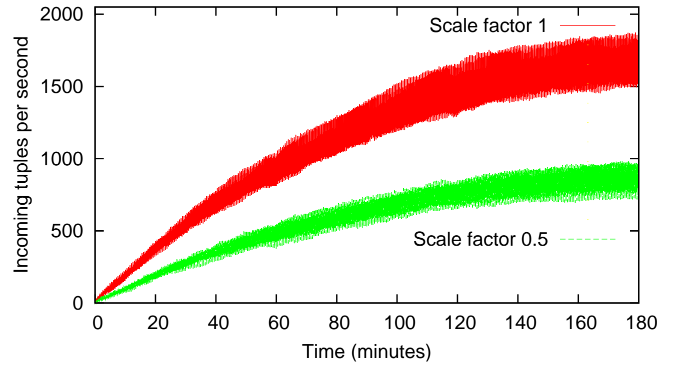

Figure 8: Data distribution during the benchmark

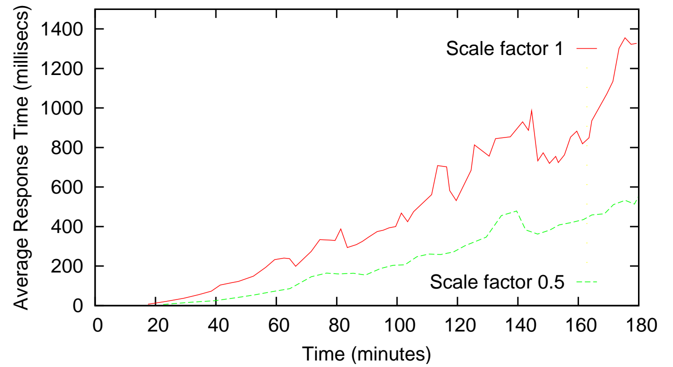

Figure 9: Average response time for $Q7$

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 7 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 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 7(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 7(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 8 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.

Furthermore, Figure 9 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., [13] as this collection defines the performance of the system by containing the most heavyweight queries, dominating the system resources (see Figure 7). 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 9 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. 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.

7. RELATED WORK

The DataCell falls in the category of stream-engines for complex event processing. Several DSMS directions have been studied in recent years, e.g., [4, 6, 7, 8, 9, 10, 12, 16], but few have reached a maturity to live outside the research labs, e.g., Borealis [1] and TelegraphCQ [7]. The DataCell is disseminated via the MonetDB product family.

The main difference of the DataCell is that it builds a completely functional DSMS on top of a modern DBMS. Contrary to the other systems, it can exploit all existing functionality of a DBMS and can support complex queries and functionalities. It proves that this is a promising direction that deserves thorough study.

Naturally, all current research on streams shares goals and concepts with the *active* databases area. Most noticeable, IBM's effort to transform a normal/passive DBMS, Starbust, to an active DBMS, called Alert [19] comes closer to the DataCell approach. Active tables and queries share commonalities with DataCell's baskets and factories. However, the DataCell model is a much more generic and powerful one by allowing continuous queries to share baskets, 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.

In addition, the design of the DataCell allows to exploit batch processing when the application allows it. Tuple-at-a-time processing, used in other 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 [15] 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.

The functionality of the DataCell was inspired by StreamSQL [20] and CQL [5, 2]. These languages have been developed for simpler queries. Instead, the DataCell has been developed for complex queries and it supports the complete SQL-based language.

8. CONCLUSIONS

In this paper, we presented the DataCell, a radically different approach in designing a stream engine. The system directly exploits all existing database knowledge by building on top of a modern DBMS kernel. Incoming tuples are stored into baskets/tables and then they are carefully queried and removed from these tables by the multiple factories (queries/operators) waiting in the system. The design allows for numerous alternative ways of interaction between the basic components opening the road for interesting and challenging research directions. This paper studied the basic approaches and through a complete implementation, 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.

9. REFERENCES

[1] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] A. Arasu et al. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL*, 2003.

[3] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.

[4] B. Babcock et al. Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 13(4):333–353, 2004.

[5] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.

[6] H. Balakrishnan et al. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[7] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[8] J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[9] C. D. Cranor et al. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[10] L. Girod et al. The Case for a Signal-Oriented Data Stream Management System. In *CIDR*, 2007.

[11] S. Harizopoulos et al. QPipe: a simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[12] M. Ivanova and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. In *VLDB*, 2005.

[13] N. Jain et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD*, 2006.

[14] M. Kersten, E. Liarou, and R. Goncalves. A Query Language for a Data Refinery Cell. In *Int. Workshop on Event Driven Architecture and Event Processing Systems*, 2007.

[15] H. Lim et al. Continuous query processing in data streams using duality of data and queries. In *SIGMOD*, 2006.

[16] S. Madden et al. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD*, 2002.

[17] MonetDB. http://www.monetdb.com.

[18] J. L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3), 1977.

[19] U. Schreier et al. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *VLDB*, 1991.

[20] StreamSQL. http://blogs.streamsql.org/.