DataCell: Building a Data Stream Engine on top of a Relational Database Kernel

Erietta Liarou
(supervised by Martin Kersten)
CWI Amsterdam, The Netherlands
erietta@cwi.nl

ABSTRACT

Stream applications gained significant popularity in recent years, which lead to the development of specialized datastream engines. They often have been designed from scratch and are tuned towards the specific requirements posed by their initial target applications, e.g., network monitoring and financial services. However, this also meant that they lack the power and sophisticated techniques of a full fledged database system accumulated over many years of database research.

In this PhD work, we take the opposite route and design a stream engine, the DataCell, directly on top of a modern database kernel. To achieve this objective, we isolated the necessary and sufficient mechanism to support continuous query processing in a relational database environment. This led to a lightweight and orthogonal extension of SQL with a direct hook into the sophisticated algorithms and techniques of the DBMS. The streaming application can use any kind of complex query functionality without the need for us to reinvent a complete software stack, i.e., language parsers, optimizers, and storage structures. In this paper, we charter the roadmap of this thesis, the opportunities and challenges that arise with such a direction, and the significant advantages already achieved.

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.

VLDB '09, August 24-28, 2009, Lyon, France

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, 10, 13].

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 [22], i.e., each compo-

nent/process/sub query plan is triggered only if it has input to process while its output is the input for other processes.

Building a stream system on top of a modern relational database engine, offers freely access to the low level technology but also arises a plethora of new issues that have normally never required attention before by a standard database application scenario. First of all, how can we support the continual nature of a query? How can we organize and schedule multiple continuous queries taking also into account different query priorities, low-latency processing and possible load shedding requirements, based on the dynamic environment changes? How can we integrate queries that just arrived with existing ones? Also, how will we efficiently support window query processing, a model that is mainly invented to make feasible the infinite streaming behaviour? Our contribution is the awareness that this research direction is feasible and that it can bring significant advantages.

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.

The remainder of the paper is organized as follows. In Section 2, we present a detailed introduction of the DataCell architecture at large followed by some interesting open research directions in Section 3. Then, in Section 4, we discuss related work and Section 5 concludes the paper.

2. THE DATACELL ARCHITECTURE

In this section, we discuss the DataCell architecture. We build the DataCell on top of MonetDB [21], an open-source column-oriented DBMS.

Let us first give a brief description of our underlying database kernel. 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 [21].

The DataCell is 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

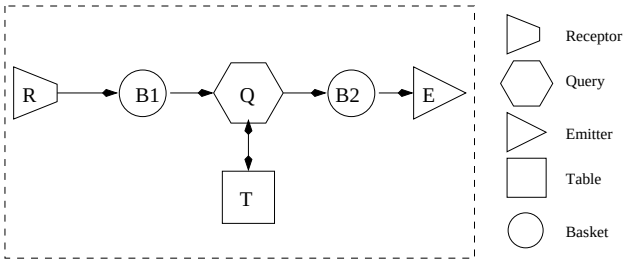

Figure 1: The DataCell model

to recognize and process continuous queries.

In the remainder of the section, we step by step build up the architecture and the possible research directions. Our architecture 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.

2.1 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.

2.2 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, 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.

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 unnecessarily complicates applications that do not depend on arrival order. On the other hand, baskets in DataCell provide maximum flexibility to perform both in-order and out-of-order processing 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. 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.

2.3 Factories

The continuous queries are cast into MonetDB's *factory* concept, i.e., a powerful co-routine method provided by the relational algebra engine. In DataCell, a factory contains the compiled continuous query plan. It has at least one *input* and one *output* basket from which it continuously reads data, processes it and creates a result to be placed in the output baskets. This process is a bulk operation, all tuples consumed are removed from their input baskets.

Having introduced the DataCell components, we can 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.

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 to 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 using the original MonetDB commands. The infinite loop is suspended to let the scheduler control its processing.

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
```

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.

2.4 Processing Model

The DataCell architecture uses the abstraction of the Petri-net model [22] to facilitate continuous query processing. A Petri-net is a mathematical representation of discrete distributed systems. It uses a directed bipartite graph of *places* and *transitions* with annotations to graphically represent the structure of a distributed system. Places may contain (a) tokens to represent information and (b) transitions to model computational behavior. Edges from places to transitions model input relationships and, conversely, edges from transitions to places denote output relationships. 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. An advantage of the Petri-net model is that it provides a clean definition of the computational state.

In the DataCell world now, 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 repre-

sent 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 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. 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.

2.5 Processing Strategies

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. The current DataCell prototype implements a number of alternatives.

Our first strategy, called *separate baskets*, provides the maximum independence to each query and stream. Each query becomes a single factory and has its own input/output baskets meaning that each query can be processed independently at the expense of copying the same input to the basket of each relevant query.

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.

The shared baskets strategy 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. This strategy opens the road for even more advanced ways of exploiting query commonalities.

2.6 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 [17].

3. THESIS OUTLINE

Having discussed the basic architecture of DataCell and the minimum additional building blocks necessary, let us now briefly describe some of the interesting research directions that arise.

One of the major issues is how to support window-based processing in DataCell. Window queries were invented to make possible the evaluation of continuous queries over unbounded streaming data, especially in the presence of blocking operators. The pure database technology does not include special window-based operators or any kind of optimized algorithms for this kind of queries. Following the DataCell approach, our goal is not to rebuild a new special class of windowed operators. Instead, we study a scheme that achieves window processing based on careful high level scheduling and dynamic query plan rewriting.

A second interesting issue is that of multi-query processing and the rich scheduling opportunities that control the interaction between the multiple waiting queries. Exploiting similarities at the query and data level is necessary in order to meet the real time deadlines a stream application sets. The essential difference in the DataCell model, is that we are looking for similarities at the query plan level. This way, we need to study mechanisms to efficiently and dynamically organize the queries in multiple groups based on their needs and properties. To accommodate partially overlapping queries we also need mechanisms to dynamically *split* and *merge* factories that wrap the query plans (or parts of them).

3.1 Windowed Query Processing

Continuous computation of long standing queries in large scale streaming environments is a huge challenge from a data management perspective. Continuously considering all past data as candidates for answers to a continuous query is not a scalable solution. Especially when it comes to blocking operators, e.g., a join, it is unrealistic to continuously consider all data, as the stream is continuously bringing more and more data. This way, window-based queries have been introduced to assist efficient query processing in streaming environments. By windowing a continuous query, we delimit the boundaries of the initially unbounded incoming data and we continuously produce *partial* answers on different portions of the data. As the content of the window changes (e.g., new tuples are inserted, old tuples are expired), the query answer is also updated in order to reflect the data input changes.

High interest gains a special class of window queries, the sliding window queries. There, the challenging part is that we need to repeatedly access/process *partially the same* data. This fact combined with the high memory and processing cost requirements which are typical in a streaming environment, led the researchers to search and study alternative evaluation techniques that minimize the access and reevaluation of data that has been already processed within a given window. Specialized algorithms have been proposed for multiple different operators, such as window joins [14, 16, 15] and windowed aggregates, e.g., [11, 9]. The two basic high level routes we can follow in order to handle sliding window queries are the (a) *re-evaluation* and the (b) *incremental evaluation*, [12]. According to the query re-evaluation approach, data is processed one full window at a time. When a window w is complete, i.e., *all* the tuples that should be in w have arrived, we process all the tuples in w in one go. When new tuples arrive, they are appended to the window, the window slides and older tuples expire (sliding window case); then the evaluation of the query on the new window happens from scratch. On the other hand, with the incremental evaluation approach, we build the answer of a window snapshot by exploiting partial intermediate results of previous window snapshots on the same query. The goal is to avoid processing again the same data. Instead, we process only what is really necessary to reflect the changes to window, e.g., the effect of the new tuples that just arrived and the effect of the old tuples that just expired.

The re-evaluation method absolutely fits in the context of DataCell and can be easily supported by the concept of the factory. However, the incremental evaluation approach seems more promising since it avoids processing the already known stream data, but it requires additional effort that is not by-default integrated into the traditional underlying database technology. The major challenge is to support efficient window based processing without building new operators.

Our initial direction follows the *basic window* model, introduced in [25]. The basic window model is a natural way to handle sliding windows and has been heavily exploited in the literature. In the DataCell context though, the problem is very different due to the inherit design to stick with the original database operators and mainly due to the fact that the problem is elevated at the query plan level. From a high level point of view, with a basic window approach, a window w is split into more than one smaller sub-windows called *basic windows*, *bw*. For each sub-window we keep a *summary* that describes its content, in such a way that we do not need to process again the same portion of data. While window w slides, the older *bw* expire and we forget their summaries while new *bw* are added which we process to synthesize the new result.

In addition, the role of the scheduler is very important in this context to trigger the evaluation of the proper factories when there are enough tuples to fill one or more windows. For count-based windows all we need to do is to monitor the number of tuples in baskets. For time based windows the scheduler needs to monitor the timestamp of incoming stream tuples. For predicate-based windows it may become much more complicated and it requires more extensive data processing.

3.2 Processing Model

Here, we investigate research opportunities that arise from the basic DataCell processing model. The most challenging directions come from the choice to split the plan of a single query into multiple factories. The motivation for this comes from different angles. For example, each factory in a group of factories sharing a basket, conceptually releases the basket content only after it has finished its full query

plan. Assume two query plans, a lightweight query q_1 and a heavy query q_2 that needs a considerable longer processing time compared to q_1. With the shared baskets strategy we force q_1 to wait for q_2 to finish before we 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, such that part of the input can be released as soon as possible, effectively eliminating the need for a fast query to wait for a slow one.

Another natural direction that comes to mind when we decide to split the query plans into multiple factories is the possibility to share both baskets and 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.

The above directions are targeted to provide support for multi-query processing in DataCell. Naturally, most of these directions fall into the responsibility of the scheduler component of the DataCell which emerges as the most crucial component in order to provide stream capabilities in a typical DBMS. One of the most challenging issues with the scheduler is to dynamically reorganize and adapt the scheduling policy for a query or group of queries for efficient query processing triggered by environment and workload changes.

4. RELATED WORK

The DataCell falls in the category of stream-engines for complex event processing [4, 6, 7, 8, 10, 13, 20], but few have reached a maturity to live outside the research labs, e.g., Borealis [1] and TelegraphCQ [7].

Naturally, 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 [23] 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 [19] 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 [24] 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.

5. CONCLUSIONS

In this paper, we present the DataCell, a radically different approach in designing a stream engine. The system exploits all existing database technology by building directly 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 multiple *factories* (queries/operators) waiting in the system. The design allows for numerous alternative ways of interaction between the basic components, that together with the experience gained from the existing stream literature, can lead to very interesting research opportunities. We have implemented the first DataCell [18] prototype on top of MonetDB and were able to achieve out of the box good performance on the Linear Road benchmark [3]. The major research challenges for DataCell include window based processing, multi-query processing and and continuous query plan adaptation, all of which are areas where the different design of the system introduces a flow of new research problems to handle.

6. 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] E. Cohen and M. Straussat. Maintaining time-decaying stream aggregates. In *PODS*, pages 223–233, 2003.

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

[11] M. Datar et al. Maintaining stream statistics over sliding windows. In *SIAM Jour. on Computing*, pages 635–644, 2002.

[12] T. M. Ghanem et al. Incremental evaluation of sliding-window queries over data streams. *TKDE*, 19(1):57–72, 2007.

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

[14] L. Golab et al. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.

[15] M. A. Hammad et al. Scheduling for shared window joins over data streams. In *VLDB*, 2003.

[16] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[17] M. Kersten, E. Liarou, and R. Goncalves. A Query Langua- ge for a Data Refinery Cell. In *EDA-PS*, 2007.

[18] E. Liarou et al. Exploiting the Power of Relational Databases for Efficient Stream Processing. In *EDBT*, 2009.

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

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

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

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

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

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

[25] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, 2002.