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

Citation for published version (APA):
Chapter 3

DataCell Architecture*

3.1 Introduction

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

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

*The material in this chapter has been the basis for the EDBT09 paper “Exploiting the Power of Relational Databases for Efficient Stream Processing” (Liarou et al., 2009).
query plans into multiple parts and sharing baskets between similar operators (or groups of operators) of different queries allowing reuse of results and so on. The query processing scheme of DataCell follows the Petri-net model (Peterson, 1977), i.e., each component/process/sub query plan is triggered only if it has input to process while its output is the input for other processes.

### 3.1.1 Challenges and Contributions

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

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

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

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

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

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

3.1.2 Outline

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

3.2 The DataCell Architecture

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

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

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

3.2.1 Receptors and Emitters

The periphery of a stream engine is formed by adapters, i.e., software compo-
nents to interact with devices, e.g., RSS feeds and SOAP web-services. The
communication protocols range from simple messages to complex XML docu-
ments 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. The emitter automatically removes from the kernel the delivering data. There can be multiple emitters each one responsible for delivering a different result to one or multiple clients.

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

### 3.2.2 Baskets

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

Once data is collected into baskets, we can evaluate the relevant continuous queries on top of these baskets. In this way, instead of feeding each individual tuple to the relevant query, we evaluate each query over its input basket(/s)
in one go (e.g., consuming all accumulated tuples at once). This processing model resembles the typical DBMS scenario and thus we can exploit existing algorithms and functionality of advanced DBMSs. Later in this section, we discuss in more detail the interaction between queries and baskets.

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

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

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

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

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

- **Basket Tuple Expiration**
  In a stream application scenario, tuple expiration happens on a more frequent basis than in a typical OLAP scenario. In DataCell, we handle data stream expiration immediately and in a different way than we handle updates and deletions in the underlying columnar architecture of MonetDB. In MonetDB, for each column we maintain three different arrays to represent the original persistent tuples, the updated and inserted tuples, and the deleted tuples. Once a (one-time) query is submitted, we first merge
the tuples from these three arrays to get only the valid tuple values and then continue with the actual query evaluation. When the two secondary arrays grow enough, then the merging happens automatically. Even if the data deletion in an OLAP scenario looks similar to the data stream expiration in a stream application scenario, it is inefficient to follow this processing model since the accumulated number of expired tuples is expected to grow rapidly. Instead, in DataCell we choose to do complete and instant data stream deletion, without maintaining the expired data aside. This change implies a change to the generated query plans as well. We now have only a single array instead of three (because we do deletions in place) and consequently inside the plans there is no need to do any merging of these arrays.

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

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

In this way, we exploit all column-store benefits during query processing, i.e., a query needs to read and process only the attributes required and not all attributes of a basket. For example, assume a stream $S$ that creates tuples with $k$ different attributes. In a row-oriented system, each query interested in any of the attributes in $S$ has to read the whole $S$ tuples, i.e., all $k$ attributes. In DataCell, we exploit the column-oriented structure of the underlying model,
and allow each query to bind only the attributes (of baskets) it is interested in, avoiding to access extra data and reducing their footprint. Furthermore, queries interested in different attributes of the same stream can be processed completely independently. We encountered the above scenarios for numerous queries in the Linear Road benchmark (Arasu et al., 2004) where each stream contains multiple attributes while not all queries need to access all of them.

3.2.3 Factories

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

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

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

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

In general, at any point in time, multiple receptors wait for incoming tuples and place them into the disjoint baskets. A scheduler handles multiple factories that read these input baskets and place results into multiple output baskets where multiple emitters feed the interested clients with results. It is a multi-threaded architecture, where every single component (i.e., receptors, emitters and the factory scheduler) is an independent thread. Figure 3.2 illustrates an overview of the DataCell architecture, with all the participant components de-
scribed above and the extensions of the underlying MonetDB system. DataCell components are positioned between the MonetDB SQL compiler/optimizer and the DBMS kernel. The SQL compiler is extended with a few orthogonal language constructs to recognize and process continuous queries (see Chapter 4). The query plan as generated by the SQL optimizer is rewritten to a continuous query plan and handed over to the DataCell scheduler. In turn, the scheduler handles the execution of the plans.

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

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

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

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

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

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

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

The factory in Algorithm 1 contains the full query plan (in this case just a single operator in line 6) where the original MonetDB operators are being used.
In particular, we use the `select` operator that belongs in the `algebra` module. The modules represent a logical grouping and they provide a name space to differentiate similar operations.

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

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

### 3.3 Query Processing

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

#### 3.3.1 The DataCell Processing Model

The DataCell architecture uses the abstraction of the Petri-net model (Peterson, 1977) to facilitate continuous query processing. A Petri-net is a mathematical representation of discrete distributed systems. It uses a directed bipartite graph of places and transitions with annotations to graphically represent the structure of a distributed system. Places may contain (a) tokens to represent information and (b) transitions to model computational behavior. Edges from places to transitions model input relationships and, conversely, edges from transitions to places denote output relationships.
A transition fires if there are tokens in all its input places. Once fired, the transition consumes the tokens from its input places, performs some processing task, and places result tokens in its output places. This operation is atomic, i.e., it is performed in one non-interruptible step. The firing order of transitions is explicitly left undefined.

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

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

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

Each factory has as input one or more baskets from where it reads its input data. These baskets may be the output of one or more receptors or the output
of one or more different factories or mixed. The output of a factory is again one or more baskets where the factory places its result tuples.

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

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

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

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

When a transition has multiple inputs, then all inputs must have tuples for the transition to run. In certain cases, to guarantee correctness and avoid unnecessary processing costs, auxiliary input/output baskets are used to regulate when a transition runs. Assume for example a sliding window join query \( q \), with two input baskets \( B_1 \) and \( B_2 \) that reflect the join attributes. Every time \( q \) runs, we need to only partially delete the inputs as some of the tuples will still be valid for the next window. At the same time, we do not want to run the query again unless the window has progressed, i.e., new tuples have arrived on either input. Adding a new auxiliary input basket \( B_3 \) solves the problem. The new basket is filled with a single tuple marked \textit{true} every time at least one new tuple is added to either \( B_1 \) or \( B_2 \) and is fully emptied every time \( q \) runs.
Figure 3.4: Examples of alternative processing schemes
3.3.2 Processing Strategies

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

Separate Baskets

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

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

Shared Baskets

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

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

To achieve the above goal, for every basket \( B \) which shared as input between a group of \( k \) factories, we add two new factories, the locker and the unlocker. An example is shown in Figure 3.4(b). The locker factory, \( \text{Lock} \), is placed between \( B \) and the originally attached factories (i.e., submitted continuous queries). Once \( B \) contains a number of new tuples, \( \text{Lock} \) runs. Its task is to simply lock \( B \). The output of \( \text{Lock} \) is \( k \) baskets, one for each waiting factory, i.e., \( L_{F_1}, L_{F_2}, \ldots, L_{F_k} \). In each one of these outputs, \( \text{Lock} \) writes a single tuple containing a bit attribute marked “true”. Then, all factories can read and process \( B \) but without removing any tuples. Every factory \( F_i \) has an extra output basket, apart from the expected result basket, where it writes a single bit attribute to mark that its execution over the locked version of the input basket \( B \) is over. In Figure 3.4(b) this is shown as \( U_{F_i} \). These output baskets are inputs to the unlocker factory \( \text{Unlock} \). The task of \( \text{Unlock} \) is that once all factories have seen the content of the input basket i.e., once all output baskets \( U_{F_1}, U_{F_2}, \ldots, U_{F_k} \) are marked, it removes from \( B \) all tuples covered by the semantics of the factories, and subsequently it unlocks \( B \) so that the receptor can insert new tuples.

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

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

Further ideas for sharing data streams. Another way to achieve data stream sharing among co-existing continuous queries, is to follow a differential approach. The idea is that apart from the original basket that constitutes the common input source for multiple factories, each factory maintains a separate set of arrays, i.e., one array for each basket column. In each auxiliary basket
the factory marks the tuples it has already consumed. Thus, every time the scheduler triggers a factory, it should first merge the two different versions of its input baskets. More precisely, each factory should merge the original basket which is the same for every interested factory and the expiration basket which is unique for every factory. In this way, a factory always gets all valid tuples and then it can continue with the rest of the query evaluation steps. In this scheme, a garbage collector should have access to both the input basket and to all the factory baskets that maintain the expired tuples of the co-existing continuous queries. This is necessary such that it can periodically clean the tuples that are not useful any more by any query, lightening the total storage space.

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

Partial Deletes

The shared baskets strategy, described above, removes the tuples from a shared input basket only once all relevant factories have seen it. The next strategy is motivated by the fact that not all queries on the same input are interested in the same part of this input. For example, two queries $q_1$ and $q_2$ might be interested in disjoint ranges of the same attribute. Assume $q_1$ runs first. Given that the queries require disjoint ranges, all tuples that qualified for $q_1$ are for sure not needed for $q_2$. This knowledge brings the following opportunity; $q_1$ can remove from $B$ all the tuples that qualified its basket predicate and only then allow $q_2$ to read $B$. The effect is that $q_2$ has to process less tuples by avoiding seeing tuples that are already known not to qualify for $q_2$. All we need is an extra basket between $q_1$ and $q_2$ so that $q_2$ runs only after $q_1$. Figure 3.4(c) shows an example where three queries, encapsulated in $F_1$, $F_2$ and $F_3$ factories respectively, create such a chain. Each factory proceeds to the query execution, appending tuples to its attached output basket, and in parallel leaves behind the left-overs of its input, e.g., $B' \subseteq B$.

This strategy opens the road for even more advanced ways of exploiting query commonalities. For example, the idea to incrementally build indices based on the particular needs of each continuous query follows the philosophy of the partial deletes mechanism we described above. There, we could choose when it is worth to build an index on streaming data, e.g., as in (Idreos, 2010) where indices are
build during the execution stage, which will also be valuable for the queries we are going to execute afterwards. We have not covered these techniques in this thesis, but it is an interesting future research direction.

3.3.3 Research Directions

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

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

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

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

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

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

The basic DataCell optimizer pipeline is the following:

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

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

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

The `datacell` optimization set of rules is exclusively created to cover the needs of the continuous query scenario. Its main role is to transform a one-time
query plan to a *continuous* query plan. The main actions it takes are as follows.

- It wraps the MAL plan in a factory (see Section 3.2.3 and Algorithm 1).

- It adds in the proper place of the MAL plan the infinite loop that guarantees continuous query processing. Instructions that should be evaluated only once, such as basket binds, remain outside the loop.

- It plugs in the appropriate data cleaning instructions for proper tuple expiration.

- It introduces the locking and unlocking scheme for the source and target baskets of the query.

- It discards the unnecessary (secondary) arrays that by default represent the deletions and updates of each column in MonetDB. In addition, it cleans the corresponding commands that the discarded arrays participate (explicitly and implicitly).

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

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

### 3.5 Experimental Analysis

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

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

Metrics

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

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

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

Interprocess Communication Overhead

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

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

In the following experiment, we measure the elapsed time and the throughput while varying the number of queries. The sensor creates $10^5$ random two-column tuples. For each tuple $t$, the first column contains the timestamp that this tuple
was created by the sensor, while the second one contains a random integer value. We use simple `SELECT *` queries. Thus, within the kernel every query passes all tuples to the next one which reflects the worst case scenario regarding the data volume flowing through the system.

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

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

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

**Pure Kernel Activity**

At first sight the performance figures discussed above do not seem in line with common belief. Unfortunately, the literature on performance evaluation of
Figure 3.6: Effect of inter-process communication

Figure 3.7: Effect of batch processing and strategies
stream engines does not yet provide many points of reference. GigaScope (Cranor et al., 2003) claims a peak performance up to a million events per second by pushing down selection conditions the Network Interface Controller. On the contrary, early presentations on Aurora report on handling around 160K msg/sec. Comparing Aurora against a commercial DBMS, systemX, the systems show the capability to handle between 100 (systemX) and 486 (Aurora) tuples/second (Arasu et al., 2004). Two solutions for systemX are given, one based on triggers and stored procedures, and another one based on polling.

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

**Batch Processing**

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

\[
\text{SELECT } S.a \\
\text{FROM } S \\
\text{WHERE } v1 < S.a < v2
\]

All queries select a random range with $10\%$ selectivity. Figure 3.7(a) depicts the average latency per tuple for various different numbers of installed queries and while varying the batch size ($T$) used in query processing. The case of $T=1$ demonstrates the impact of the traditional processing model of handling one tuple at a time. We clearly see that the latency significantly decreases as we increase the batch size materializing a benefit of roughly three orders of magnitude. An important observation is that the benefits of batch processing
increase with a higher rate up to a certain batch size and then the improvement is much less. When the batch size becomes very big, performance starts to degrade especially for the case of the maximum number of queries. This is due to the delay time needed, i.e., the average time a tuple has to wait for more tuples to arrive so that the desired batch size is reached. Only then the tuples can be processed. However, there is a point that this delay time becomes so big that overshadows the benefits of grouped processing, i.e., performance does not improve anymore or even degrades. In our experiment this point appears at $T = 10^3$. Optimally setting and adapting the batch size depending on the queries and system status is an open research problem.

**Alternative Strategies**

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

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

**3.5.2 The Linear Road Benchmark**

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

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

The Benchmark

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

The benchmark contains a tool that creates the data and verifies the results. The data of a single run reflects three hours of traffic, while there are multiple
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 \(6 \times 10^6\) tuples, while for scale factor 1 we need to process \(1.2 \times 10^7\).

**Implementation in the DataCell**

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

In order to verify the baseline of our approach and keep the implementation
3.5. EXPERIMENTAL ANALYSIS

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

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

Evaluation

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

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

Figure 3.10: System load for each query collection (Q1-Q3)
Figure 3.11: System load for each query collection (Q4-Q7)
the cost is increased as more data arrives. This is due to a number of reasons. First, data and intermediate results is accumulated over time creating bigger inputs for the various queries. Most importantly, in many cases it is the content of the incoming data that triggers more work. For example, the second query collection (Figure 3.11(c)) is the one detecting the accidents. With the way data is created by the benchmark (for scale factor 1), accidents occur with a continuously increasing frequency after one hour. This is when we see the queries in Figure 3.11(c) to increase their workload as to compute the various accident situations for each car, in each lane etc. In turn, these queries create bigger inputs for the queries in the next query collections and so on.

Furthermore, the benchmark is designed in such a way that more data enters the system, the more the time goes by. This is demonstrated in Figure 3.12 where we show the number of tuples that enter the system every second. For example, for scale factor 1, 15 to 20 tuples per second arrive at the beginning, while towards the end of the three hours run we get up to 1700 tuples per second. All categories scale nicely achieving to process the extra data as the benchmark evolves. Even the most expensive query collection, Q7, manages to maintain performance levels below 2 seconds which is well below the 5 seconds goal.
Furthermore, Figure 3.13 depicts the average response time for query collection $Q_7$ which is one of the output results of the benchmark. This metric is common when evaluating the benchmark, e.g., (Jain et al., 2006) as this collection defines the performance of the system by containing the most heavyweight queries, dominating the system resources (see Figure 3.11). The average response time is defined as the average processing time needed for the queries in this collection. It is measured every time $10^6$ new tuples enter this collection by calculating the average time needed to process these $10^6$ tuples.

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

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

3.6 Summary

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

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