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

Citation for published version (APA):

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

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Query Language*

4.1 Introduction

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

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

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

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

- **Predicate windows.** DataCell generalizes the (sliding) window approach, predominant in DSMSs, to allow for arbitrary table expressions over streams. It enables applications to selectively process the stream and prioritize event processing based on application semantics.

- **SQL compliance.** The language extensions proposed are orthogonal to existing SQL semantics. We do not resort to redefinition of the window concept, nor do we a priori assume a sequence data type. Moreover, the complete state of the system can at any time be inspected using SQL queries.

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

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

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

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

4.1.2 Outline

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

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

4.2.1 Baskets

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

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

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

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

A distinctive feature of a basket is its handling of integrity violations. Events that violate the constraints are silently dropped. They are not distinguishable
from those that have never arrived in the first place.

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

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

4.2.2 Receptors and Emitters

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

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

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

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

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

4.2.3 Basket Expressions

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

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

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

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

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

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

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

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

enable MyFavored;

[SELECT * FROM MyFavored];

-- part of the result set
```

4.2.5 Application Modeling

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

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

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

```sql
--An Alarm Application
CREATE BASKET B1(
    tag timestamp default now(),
    pl integer);
COPY INTO B1 FROM 'alarms:60000';

CREATE BASKET B2(
    tag timestamp,
    pl integer,
    msg string);
COPY FROM B2 INTO 'console';

CREATE TABLE T1(
    pmin integer,
    pmax integer);

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

4.3 Querying Streams

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

4.3.1 Filter and Map

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

\[
\begin{align*}
\text{INSERT INTO outliers} \\
&\quad \text{SELECT b.tag, b.payload} \\
&\quad \text{FROM [SELECT top 20} \\
&\quad \quad \text{FROM X} \\
&\quad \quad \quad \text{ORDER BY tag] as b} \\
&\quad \text{WHERE b.payload >100;}
\end{align*}
\]

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

4.3.2 Split and Merge

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

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

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

\[
\text{WITH A AS [SELECT * FROM X]}
\begin{align*}
\text{BEGIN} \\
\text{INSERT INTO Y} \\
\quad \text{SELECT * FROM A WHERE A.payload > 100;} \\
\text{INSERT INTO Z} \\
\quad \text{SELECT * FROM A WHERE A.payload <= 200;} \\
\text{END;}
\end{align*}
\]

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

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

Below we see a join between two baskets \( X \) and \( Y \) with a monotone increasing unique \( id \) sequence as the target of the join. The join basket expression produces all matching pairs. The residue in each basket are tuples that do not (yet) match. These can be removed with a controlling continuous query, e.g., using a time-out predicate. Taken together they model the gather semantics.
SELECT A.*
FROM [SELECT * FROM X,Y WHERE X.id=Y.id] as A;
INSERT INTO trash [SELECT ALL FROM X WHERE X.tag < now()-1 hour];
INSERT INTO trash [SELECT ALL FROM Y WHERE Y.tag < now()-1 hour];

4.3.3 Aggregation

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

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

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

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

4.3.4 Metronome and Heartbeat

Basket expressions can not directly be used to react to the lack of events in a basket. This is a general problem encountered in data stream management systems. A solution is to inject marker events using a separate process, called
a *metronome* function. Its argument is a time interval and it injects a value timestamp into a basket.

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

```sql
CREATE FUNCTION metronome (t interval)
    RETURNS timestamp;
BEGIN
    CALL sleep(t);
    RETURN now();
END;
```

```sql
INSERT INTO into X(tag,id,payload)
[SELECT null,metronome(1 hour),null];
```

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

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

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

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

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

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

4.3.6 Bounded Baskets

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

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

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

```sql
SELECT count(B.*), 'dropped'
FROM (SELECT *
      FROM X
      WHERE id < max(SELECT id FROM X)-100)
      as B;
```
4.3. QUERYING STREAMS

4.3.7 Stream Partitioning

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

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

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

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

4.3.8 Transaction Management

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

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

4.3.9 Sliding Windows

Most DSMSs define query processing around streams seen as a linear ordered list. This naturally leads to sequence operators, such as `NEXT`, `FOLLOW`, and `WINDOW` expressions. The latter extends the semantics of the SQL WINDOW
construct to designate a portion of interest around each tuple in the stream. The \textsc{window} operator is applied to the result of a query and, combined with the iterator semantics of SQL, mimics a kind of basket expression.

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

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

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

\begin{verbatim}
SELECT * FROM [SELECT * FROM X limit 2] 
UNION 
SELECT * FROM X limit 8;
--create window Xw (size 10 seconds
-- advance 2 seconds);
SELECT * 
FROM [SELECT *
     FROM X
     WHERE tag < min(SELECT X.tag) + 2 seconds]
UNION 
SELECT * 
FROM X
WHERE tag < min(SELECT X.tag) + 8 seconds;
\end{verbatim}

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

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

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

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