A Query Language for a Data Refinery Cell

M. Kersten
CWI, Amsterdam
The Netherlands

E. Liarou
CWI, Amsterdam
The Netherlands

R. Goncalves
CWI, Amsterdam
The Netherlands

Abstract

In this paper we propose the DataCell, an event management system designed as a flexible data hub in an ambient environment and we mainly focus on its language interface. The 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 DataCell capitalizes the architectural choices and performance characteristics of a modern column-store relational DBMS. The design and implementation of the DataCell does not affect the underlying database kernel and the SQL software stack. This means that the DataCell extends the functionality of the DBMS without hampering the already given features and performance.

1. Introduction

Data Stream Management Systems (DSMS) have become an active research area in the database community. Inspiration comes from potentially large application areas, e.g., network monitoring, sensor networks, telecommunication, financial and web applications. Many DSMSs have been designed from scratch to counter the perceived sluggishness of the traditional DBMSs. The latter are considered inadequate to achieve the desired performance, because they are too heavy in terms of functionality, and they are not equipped to support continuous querying. In a relational DBMS, a submitted query is evaluated once over the existing data. On the contrary, in a stream application we need mechanisms to support long-standing queries over tables that are continuously updated from the environment.

Several DSMS solutions have been proposed, e.g., [4, 6, 7, 9, 10], and in most of them the architects follow a dedicated (bounded) main-memory approach. In addition, the application semantics are defined in a SQL-based stream query language. However, the drawback of this approach became clear as the applications grew more complex and demanding. Commercial systems, such as Streambase[1] and Coral8[2] stress the need for a better integration with a full fledged relational DBMS. Ad-hoc solutions to combine streams with proprietary persistent main-memory tables is insufficient.

In this work, we design a data stream management system, called the DataCell, adopting the other end of the spectrum. We build the DataCell on top of an extensible relational database engine, the MonetDB[3] system, an open-source, column-oriented database engine. We present our vision in the context of MonetDB, but it should be emphasized that the approach taken can be applied in other modern (SQL-based) systems as well. The database platform is used to elicit the challenges and create a working prototype quickly.

Currently we study the DataCell over the stream application scenario of an ambient home setting. The DataCell is positioned as a data refinery cell that acts as an easy programmable data hub in a multi network environment. Its task is to collect, filter and aggregate information from different sources to enable complex decision making, e.g., to control the lighting based on audio/video presentations. The challenge in an ambient environment is to hide the computer from the casual user and at the same time be proactive in steering the environment.

In this paper, we focus on the DataCell language interface. The main contributions and topics addressed are:

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

- *Bulk processing.* The DataCell processing engine is geared at bulk processing events. This favors a skewed arrival distribution, where a peak load can be handled easily, and possibly within the same time frame, as an individual event.

- *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 priory assume a sequence data type. Moreover, the complete state of the system can at any time be inspected using SQL queries. It enables reflective algorithms.

[1]http://www.streambase.com/
[2]http://www.coral8.com/
[3]http://monetdb.cwi.nl

EDA-PS'07, September 24, 2007, Vienna, Austria.

The stream behavior in the DataCell is obtained using a minimal and orthogonal extension to the SQL language. 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.

The remainder of the paper is organized as follows. In Section 2 we introduce the SQL enrichment in more detail. This is expanded in Section 3, where we give a brief overview of the DataCell Architecture. Section 4 explores the scope of the solution by modeling stream-based application concepts borrowed from dedicated stream database systems. Finally, Section 5 discusses related work and Section 6 concludes the paper and outlines future work.

2. DataCell Model

In this section we describe the DataCell model. We introduce the role of each component: *baskets*, *receptors* and *emitters*, *basket expressions*, and *continuous queries*. All components are modelled with the SQL'03 language [8] with a novel extension, the basket expression, which will also be described in this section. Together they capture and generalize the essence of streaming applications.

2.1 Receptors and Emitters

The periphery of the DataCell consists of *receptors* and *emitters*. A receptor is a separate process that picks up events from a communication channel and forwards them to the kernel for processing. The latter under an auto-commit transaction mode. Likewise, an emitter is a separate process that picks up the events that constitute the answer of the continuous queries and delivers them to clients interested, i.e., who have subscribed to the query result.

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 protocolls are TCP-IP and UDP channels, which are sufficient to create publish/subscribe systems and support the large networks in our ambient setting. We expect xml-based protocols with shredding to baskets to follow quickly.

Example 1. 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';
```

2.2 Baskets

A *basket* is the data structure to hold stream portions. It is represented as a temporary main-memory table and acts as buffer, i.e., incoming events are just appended. 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 is aligned with the table definition in SQL'03.

Example 2. 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 SB is controlled with the statements ENABLE SB and DISABLE SB. The default is to enable the basket to enqueue and dequeue tuples. By disabling it, queries that attempt to update its content become blocked. Selective (dis)enabling basket 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.

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 [1] and unnecessary complicates applications that don't care about the arrival order.

2.3 Basket Expressions

The *basket expressions* are the novel building blocks for DataCell queries. They encompass the traditional SELECT-FROM-WHERE-GROUPBY SQL language framework. 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 tables during query evaluation. All tuples referenced in the sub-expression that contribute to the result are also immediately removed from their underlying store. This leaves a partially emptied basket or table behind. Note, a basket can also be inspected outside a basket expression. Then, it behaves as any 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 the X. All tuples selected are *immediately* removed from X, but they remain accessible through B during query execution. From this temporary table we select the payloads satisfying the predicate.

```
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. In both cases tuples are removed. However, 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.

2.4 Continuous Queries

Continuous queries are long-running queries that have to be continuously evaluated 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 the DataCell we consider every query with a basket subexpression as a continuous query. A query without basket expressions follows the standard SQL semantics. A top level query over basket is not distinguishable from querying a table, only by placing basket brackets it becomes 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;
[ 135, 2007-03-27:22:45, 123] MyFavored
[ 136, 2007-03-27:22:46, 651] MyFavored
[ 137, 2007-03-27:22:49, 133] MyFavored
```

2.5 Application Modeling

The envisioned graphical user interface closely match the network view of the flow dependencies amongst the baskets, (continuous) queries, tables, and the interface. Compared to similar tools, e.g., Borealis [1], the coarse grain approach of SQL as a specification vehicle pays off.

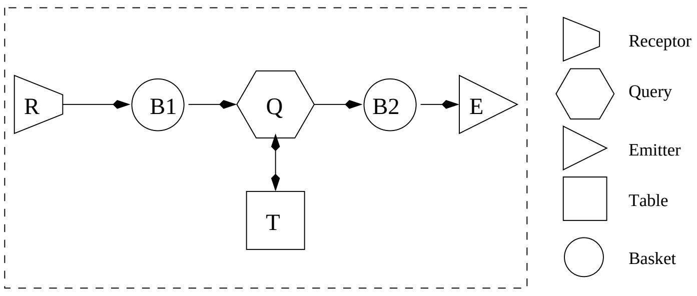

Figure 1. DataCell Application

Example 5. Figure 1 illustrates how the DataCell application scenarios can be modeled succinctly. In this simple scenario, the receptor R appends the new incoming data to a basket $B1$. When new data appears, a submitted continuous query Q obtain access to the incoming stream and it is evaluated, its results are contained in the basket $B2$ that emitter E, forwards them to the interested subscribers.

```
--An Alarm Application
create basket C1(
   tag timestamp default now(),
   pl integer);

copy into C1 from 'alarms:60000';

create basket C2( tag timestamp,
   pl integer,
   msg string);

copy from C2 into 'console';

create table T1(
   pmin integer,
   pmax integer);

insert into C2
select  tag, pl, "Warning"
from T1, [select * from C1 where pl > 0] as A,
where A.pl<T1.pmin or A.pl>T1.pmax;
```

3. DataCell Architecture

In this section we give an overview of the DataCell architecture. A peek into the compilation process is given in Section 3.2 and the computational model for complex expression in Section 3.3. We conclude with a synopsis of the runtime system.

3.1 Overview

The DataCell architecture is building on top of a traditional database engine in a way that it is fully exploiting and extending the already established features and properties of a relational DBMS. In this way, we avoid to build a specialized engine from scratch. Instead, we utilize the existing DBMS kernel to provide a highly efficient stream engine. In particular, we work on

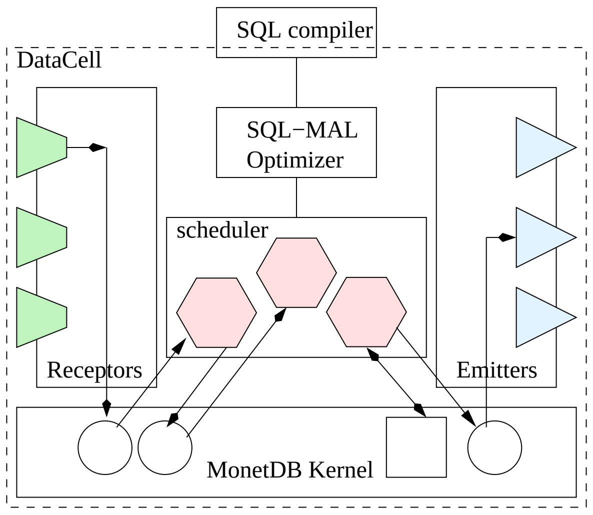

Figure 2. DataCell Architecture

top the MonetDB database system[4], a column-store database engine. MonetDB provides a virtual machine architecture based on a small assembly language MAL, which mostly wraps the highly optimized relational primitives. MAL is the target for query compilers and optimizers. The MonetDB architecture provides a clean software stack to create powerful front-ends, such as SQL'03 and XQuery[5].

The DataCell is positioned between the (enhanced) SQL to MAL compiler and the MonetDB runtime engine. In particular, the SQL runtime has been extended to deal with baskets using the BATs in the kernel and a scheduler to control activation of the continuous queries.

The multi-process architecture of DataCell is illustrated in Figure 2. Each receptor is mapped into a separate thread to handle events received and to insert them into the baskets. Likewise, each emitter aligns with a thread that handles delivery of the tuples to clients being registered.

3.2 Compilation

After the DataCell has been started, all registered continuous queries are compiled into a series of MonetDB Assembly Language (MAL) programs. In essence, a MAL program is a sequence of binary relational algebra operators that fully materialize their (intermediate) results. This algebra supports functional abstractions and a minimal set of flow-of-control primitives. It is powerful enough to represent logical plans produced by the compiler, but also precisely describe conditional and repetitive execution of subqueries. The plans received from the SQL front-end are subject to a series of code transformations, which replace portions by better evaluation sequences.

The continuous queries are translated into *factories*, MonetDB's notion of a co-routine. The factories are selectively activated by the DataCell scheduler to make another cycle through the plan. For a more in depth description of this language we refer to

```
p1:= basket.new("X_pl",:bat[:lng, :int]);
basket.group("X","X_pl");
receptor.new("X");
receptor.start("X");

p3:= basket.new("Z_pl",:bat[:lng, :int]);
basket.group("Z","Z_pl");
emitter.new("Z");
emitter.start("Z");

factory stepOne():bit;
    x:bat[:lng,:int]:= basket.bind("X","X_pl");
    z:bat[:lng,:int]:= basket.bind("Z","Z_pl");
barrier go:= true;
    basket.lock("X");
    basket.lock("Z");
    be:= algebra.select(x,0,nil);
    bat.insert(z,be);
    bat.delete(x,be);
    basket.unlock("Z");
    basket.unlock("X");
    yield qry00:=true;
  redo go:= true;
exit go;
end stepOne;
```

Figure 3. MAL program for a query step

the MonetDB website[6].

Example 6. Figure 3 illustrates a snippet a continuous query plan. The core of the plan is a regular relational algebra expression. It is surrounded by calls to the lock manager, to avoid concurrency conflicts amongst the DataCell threads. The outer loop of the plan *yield*s a result each time the loop identified by *barrier* . . . *redo* . . . *exit* is executed. The complete (sub)plan is packaged in a co-routine called by the DataCell scheduler.

3.3 Computational Model

The computational model underlying continuous query processing is based on Petri nets [12], or predicate nets in particular [11]. All baskets are considered placeholders for tokens. The basket expressions are aligned with Petri-net transitions and a transition "is fired" if input tokens are available on all input placeholders. In the DataCell context this maps to a test for non-empty basket expressions, i.e., a query automatically is executed if the basket expressions contain tuples. The result of the transition in a Petri-net is the delivery of tokens in, possibly multiple tables.

Example 7. The Petri-net model for the query below is shown in Figure 4.

```
select * from
  [ select * from X] as A,
  [ select * from
    [select * from Y] as B
  ] as C
```

An advantage of the Petri-net model is that it provides a clean definition of the computational state. That is, stream semantics are not hidden behind language concepts, such as sliding windows

[4]http://monetdb.cwi.nl
[5]See http://monetdb.cwi.nl for details on MAL
[6]http://monetdb.cwi.nl

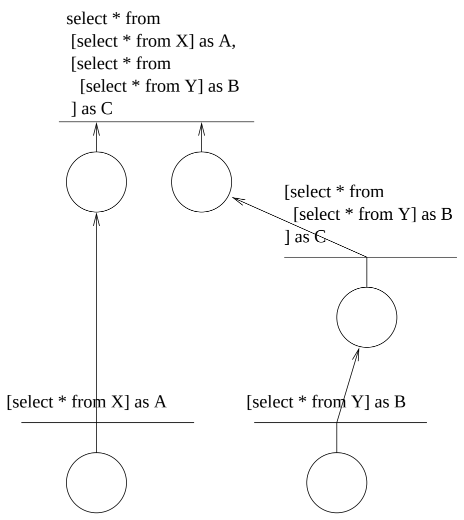

Figure 4. Petri-net Example

or a tuple-at-a-time processing scheme. Every Petri-net transition runs in isolation and has full access to all placeholders. The computational model produces a bottom up evaluation sequence for queries with nested and sibling basket expressions. That is, it enables queries whose actions depend on intermediates produced within the same (compound) queries. Petri-net analysis tools, e.g., PIPE2[7] can be used to analyze problematic cases and performance bottlenecks.

3.4 Runtime Behavior

Each continuous query proceeds in a few well-defined steps using a bottom-up traversal of the query plan. We call this collection of steps *query execution life-cycle*.

1. A snapshot is taken from all baskets of interest.
2. Basket (sub-)expressions are evaluated producing temporary tables.
3. The result is evaluated as an ordinary relational query.
4. Tuples consumed from the baskets are permanently removed.
5. Repeat the steps 1 to 5 for the next batch of tuples.

The snapshot view is a cheap operation. It inspects all baskets mentioned in the query and replaces them with a view. In MonetDB, taking a range-based view is handled in sub-micro seconds. These views fixate the database state to consider during query execution cycle. This action could be a system global atomic action. In phase two, all basket expressions are evaluated. For each we build a result table and create a list of tuples that will be removed from the basket or table upon completion. If one of the stream expressions produces an empty result, then the iteration cycle is canceled and, conceptually, all tuples are moved back to their baskets. In phase three, the query is evaluated against a database state comprised of tables for inspection and the temporaries produced. The result set is sent to the client that posed the query, a table or to an outer query.

[7]http://pipe2.sourceforge.net/

This basic cycle opens a plethora of optimization challenges. For example, if we deal with an ordered stream, cancellation of the cycle does not require to redo all work. Instead, we can keep the intermediate around until the next iteration starts and decides cheaply if the state of affairs has changed by comparing the view attributes.

4. Querying Streams

In this section we illustrate how key features of a stream query language fit the DataCell model. The state-of-the-art query language streamSQL[8], is used as a frame of reference. Its design is based on experiences gained in the Aurora [6] and CQL in STREAM [2, 4] projects. It also reflects an experience based approach, where the language design evolved based on concrete application experiments.

4.1 Filter and Map

The key operators for a streaming application are $filter$ and map. The $filter$ operator inspects individual tuples in a basket and is the most common operator. Tuples that satisfy the filter are taken out of the basket, others remain until further notice. A map operator takes an event and constructs a new one using built-in operators and calls to linked-in functions. Both operations directly map to the basket expression. There are no upfront limitations with respect to functionality, e.g., predicates over individual events or lack of access to global tables.

Example 8. A simple stream filter is shown below. It selects outlier values within batches of precisely 20 events and keeps them around in a table.

```
create view Filter
  as [select * from X
      order by id size 20 ];
insert into outliers
  select b.tag, b.payload
    from Filter as b
    where b.payload >100;
```

The SIZE clause is equivalent to the SQL LIMIT clause. In combination with the ORDER BY clause it simulates a non-overlapping tumbling window.

4.2 Stream Aggregates

The initial strong focus on aggregation networks has made stream aggregations a core language concept. 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 the 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*.

Example 9. Continuous queries are often used to accumulate a statistical property as more tuples from a basket have been consumed. 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. In this case, updates only take place after every 10 tuples.

[8]http://blogs.streamsql.org/

```
declare cnt integer;
declare tot integer;
set tot =0;
set cnt=0;
with Z as [select payload X ]
begin
  set cnt = cnt +(select count(*) from Z);
  set tot = tot +(select sum(*) from Z);
end;
```

4.3 Stream Splitting

Stream splitting enables tuple routing in the query engine. It is heavily used to support a large number of continuous queries with common interest by factoring out the common part of interest.

Unfortunately, in standard SQL all queries produce a single tabular result, i.e., there is no syntactic construct to spread the result over multiple targets. To achieve the desired effect one has to resort to the procedure abstraction offered by SQL-PSM.

Alternatively, the SQL'99 WITH construct comes closest to what we need. It defines a temporary table constructed as a prelude for a query. 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.

Example 10. Reconsider our basket X with the intend to partially replicate it into two basket Y and Z. The WITH compound block is executed for each basket binding A.

```
with A as [select * from X]
begin
  insert into Y
    select * from A
    where A.payload>100;
  insert into Z
    select * from A
    where A.payload<=200;
end;
```

The non-deterministic behavior due to parallelism and relative speed of the continuous queries require special care. In the DataCell stream splitting induces parallel processing too. However, the effect of relative speed is considered an application issue. Not a language semantic issue. The ability to inspect the complete system state and to precisely control emptying of the baskets is sufficient to deal with the problem raised.

4.4 Stream Joins

The $join$, $gather$ and $merge$ operators of streamSQL share the semantic problem found in all stream systems, i.e., at any time only a portion of the infinite stream is available. This complicates a straight forward mapping of the relational join. The problem is circumvented by redefinition of the join to hold for a portion of the baskets only and to exploit knowledge of the monotonicity of event tags. Then an efficient algorithm is within reach.

The approach taken in the DataCell is to encode the join algorithm over multiple baskets explicitly. It creates a more flexible setting, i.e., precisely define how long to wait for out of order events, at the cost of potential loss in performance.

Example 11. Consider the join between two baskets X and Y with a monotone increasing id sequence. Then a merge-join algorithm is called for. A crucial design issue is to determine when portions seen from either operand do not play a role in the remainder of the join construction. The simple merge-join step below illustrates a solution to this problem. It uses the WITH to collect bounds before the query block is entered.

```
with MaxX as
  select max(id) from X
with MaxY as
  select max(id) from Y
select A.*,B.*
from [select * from X where id < MaxY] as A,
     [select * from Y where id < MaxX] as B
where A.id = B.id;
```

The $gather$ operator concatenates tuples based on matching keys. It is equivalent to a basket expression involving a join over two baskets. A continuous query can be defined to deal with unmatched tuples piling up.

The $merge$ operator takes the input of multiple baskets and produces a merged sequence based on a user expression. The DataCell is based on baskets, i.e., bag of events. Merging its content translates simply into posing an ORDER BY clause in the basket expression.

4.5 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. A good example is the one shown in Figure 4. 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 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.

4.6 Metronomes

Basket expressions can not directly be used to react to lack of events in a baskets. This is a general problem encountered in all stream systems. A solution is to inject marker events using a separate process, called a *metronome* function. Its argument is a time interval and it injects a value 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 can program any decision function to inject the stream markers.

Example 12. The example below injects a marker tuple every hour.

```
create function metronome ( t interval)
  returns  timestamp
begin
```

```
    call sleep(t);
    return clock();
end;
insert into X(tag,id,payload)
select *
from [select null,metronome(1 hour),null] as Z;
```

The metronome can also be used in a basket expression to control their activation, but care should be take to achieve the desired result. Consider the following query

```
select S.*
from [metronome(1 hour)] as tick,
     [select * from X] as S;
```

This query will not ensure that we see all events received at an hourly interval. For, each sibling basket expression is evaluated in random order. Thus, if we start with the second basket and it turns out empty, then the metronome function won't be called. The effect is that we receive the events 1 hour after the first one is stored. The solution is to exploit the bottom-up evaluation of basket expressions. That is, the basket should be looked at only once per hour.

```
select count(*) from
  [ select * from X
    union
    (select null,metronome(1 hour),null)
  ];
```

We guarantee that each hour the query returns an answer, but not necessarily those that have been received in the latest hour. For, the result depends on the order in which both sub-queries are executed. If we start with the first one then we won't notice incoming events until the next metronome tick. Instead, we need a mechanism to enforce that the basket is empty after the metronome has ticked. The bottom up evaluation of basket expressions comes to rescue.

```
select count(*)
from [ select * from X
  where [metronome(1 hour)]];
```

In this example the metronome basket expression produces an answer before the outer expression is activated.

4.7 Heartbeats

A related problem addressed in stream applications is to ensure a uniform event stream, e.g., missing elements are replaced by a dummy if nothing happened in the last period. The basket is expected to be ordered on time. 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 are seen downstream before the next event is handled.

The heartbeat functionality can be simulated using a join between two baskets. The first one models the heartbeat and the second the events received. 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 always pick all relevant events from the heartbeat basket and produce a sorted list for further processing.

Example 13. The heartbeat functionality does not require special support. It can be modeled using the metronomes and the basket expressions as follows:

```
insert into HB select * from
[ select null, T, null
  from [metronome(1 second)] as T;

[ select * from X
 union
  select * from HB
  where X.tag< max(select tag from HB)]
```

4.8 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 grows 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 14. The query below illustrates a scheme to drop old events. Although this does not close the gap completely, the basket can be evaluated in micro-seconds.

```
select count(B.*), ' dropped'
from [select  * from X
  where id < max(select id from X)-100)] as B;
```

4.9 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 15. A continuous query that returns a sorted list by traffic per minute become:

```
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;
```

4.10 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.

4.11 Sliding Windows

Most DSMSs define query processing around streams seen as a linear order list. This naturally leads to sequence operators, such as NEXT, FOLLOWS, 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 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.

Example 16. 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.

```
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;
```

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.

5. Related work

Complex event management systems is an emerging field, which capitalizes experiences from middle-ware, publish/subscribe systems, and stream databases. The latter area has been the main driver for the query language concepts introduced in this paper.

Several DSDMs solutions have been proposed, e.g., [4, 6, 7, 9], but few have reached a maturity to live outside the research labs. Example systems that can be downloaded for experimentation are Borealis [9] and TelegraphCQ [10].

The functionality of the DataCell was inspired by streamSQL and CQL[5, 3]. The latter project has been abandoned and the software is not maintained for ease of experimentation. StreamSQL is also based on CQL, but it carries the signs of meeting the requirements of their client base. For example, it has been observed in StreamBase that the relative speed by which events flow through the network may lead to confusing information at the user interface. It led to a strong (and ad hoc) ordering of streams based on their names. The Petri-net model provides the necessary abstraction to highlight and analyze effects of concurrent behavior.

The message handling schemes offered by commercial systems, e.g. Oracle Streams and Microsoft Messaging, are primarily focused on persistent brokerage of messages between applications. Their message envelops carry the information needed for routing and authorization. Instead, the DataCell assumes a light-weight messaging system and focus on their refinery in a SQL-based application.

6. Summary and future work

The DataCell project aims to provide a complete SQL-based solution for stream data management in an ambient setting. This world is characterized with large number of sensors and mobile devices that require a stable hub to filter, aggregate, and store stream information. Moreover, event streams range from dribbles to massive event floods in case of emerging disasters.

In this paper we have chartered the direction taken by extending the MonetDB software platform. In particular, we have introduced the stream *basket*, *basket expression* and the compound WITH statement to reallies a tuple stream inside an existing SQL engine. The language extensions are orthogonal to the SQL standard. They involved minimal changes to the our SQL compiler.

Tuple movements are triggered by sub-queries that use them for producing a derived value, and continuous queries act as concurrent processes refining the tuples received from the environment. The *basket*-expressions blend into the syntax and semantics of SQL'03 and provide an elegant solution to define stream-based applications. The underlying computation model are Petri nets, which provides a sound formal basis to analyze liveliness, safety and performance of a DataCell application.

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.

A prototype DataCell kernel is operational and it is used to assess its functionality and performance. Experiments based on patching the intermediate code produced by the SQL compiler indicate that bulk processing is effective and that basket expressions nicely adapt to event arrival rates. Dealing with more than 100K events/second using the SQL framework seems feasible.

We make steady progress with the implementation of the compiler and will soon shift our focus on the plethora of optimization issues luring for attention. As soon as the functionality is fully covered, the DataCell becomes part of the MonetDB open-source package.

[9]http://www.cs.brown.edu/research/borealis/public/
[10]http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/

Acknowledgments

We wish to thank F. Groffen, M. Ivanova, and the other members of the MonetDB development team for their constructive comments on earlier versions of this paper.

7. References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. In *SIGMOD*, 2003.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL*, 2003.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 13(4):333–353, 2004.

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

[6] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uri Centintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[7] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[8] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. SQL:2003 Has been published. *SIGMOD Record*, 33(1):119–126, 2004.

[9] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. The Case for a Signal-Oriented Data Stream Management System. In *CIDR*, 2007.

[10] Milena Ivanova and Tore Risch. Customizable Parallel Execution of Scientific Stream Queries. In *VLDB*, 2005.

[11] Andrea Maggiolo-Schettini and J Winkowski. *A Generalization of Predicate/Transition Nets*. Helsinki Unversity of Technology, Digital Systems Laboratory Series A: Research Reports, 1990.

[12] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.