Enhanced Stream Processing in a DBMS Kernel

Erietta Liarou* Stratos Idreos† Stefan Manegold† Martin Kersten†

*EPFL, Switzerland
erietta.liarou@epfl.ch

†CWI, Amsterdam
{idreos,manegold,mk}@cwi.nl

ABSTRACT

Continuous query processing has emerged as a promising query processing paradigm with numerous applications. A recent development is the need to handle both streaming queries and typical one-time queries in the same application. For example, data warehousing can greatly benefit from the integration of stream semantics, i.e., online analysis of incoming data and combination with existing data. This is especially useful to provide low latency in data-intensive analysis in big data warehouses that are augmented with new data on a daily basis.

However, state-of-the-art database technology cannot handle streams efficiently due to their "continuous" nature. At the same time, state-of-the-art stream technology is purely focused on stream applications. The research efforts are mostly geared towards the creation of specialized stream management systems built with a different philosophy than a DBMS. The drawback of this approach is the limited opportunities to exploit successful past data processing technology, e.g., query optimization techniques.

For this new problem we need to combine the best of both worlds. Here we take a completely different route by designing a stream engine on top of an existing relational database kernel. This includes reuse of both its storage/execution engine and its optimizer infrastructure. The major challenge then becomes the efficient support for specialized stream features. This paper focuses on incremental window-based processing, arguably the most crucial stream-specific requirement. In order to maintain and reuse the generic storage and execution model of the DBMS, we elevate the problem at the query plan level. Proper optimizer rules, scheduling and intermediate result caching and reuse, allow us to modify the DBMS query plans for efficient incremental processing. We describe in detail the new approach and we demonstrate efficient performance even against specialized stream engines, especially when scalability becomes a crucial factor.

1. INTRODUCTION

Kranzberg's second law states that "invention is the mother of necessity". Though history proves that great technological innovations were given birth at certain periods to fulfill stressed human needs, the technological evolution of the recent years in many scientific areas, creates *new needs* all over again.

Scientific evolution on various research areas brought *data overloading* on many aspects of our lives. Modern applications coming from various fields, e.g., astronomy, physics, finance and web applications, require fast data analysis over data that are continuously growing. The Large Synoptic Survey Telescope (LSST) [3] is a characteristic paradigm. In 2015 astronomers will be able to scan the sky from a mountain-top in Chile, recording 30 Terabytes of data every night which incrementally will lead a 150 Petabyte database (over the operation period of ten years). It will be capturing changes to the observable universe, evaluating huge statistical calculations over the entire database. Another characteristic data-driven example is the Large Hadron Collider (LHC) [2], a particle accelerator that will revolutionize our understanding for the universe, generating 60 Terabytes of data every day (4Gb/sec). The same model stands for modern data warehouses which enrich their data on a daily basis creating a strong need for quick reaction and combination of scalable stream and traditional processing [35].

A new processing paradigm is born [27, 14, 18] where we need to quickly analyze incoming *streams* of data and possibly combine them with existing data in order to discover trends and patterns. Subsequently, we may also store the new data to the data warehouse for further analysis in the future if necessary. This new paradigm requires scalable query processing that can combine continuous querying for fast reaction to incoming data with traditional querying for access to existing data. However, neither pure database technology nor pure stream technology are designed for this purpose.

The traditional database technology typically faces a data processing problem, by first loading and organizing all data before it can analyze it. In most cases this strategy works fine, but when the requirement for fast and on-the-fly continuous analysis of high volume data becomes essential, this model becomes inefficient and slow-witted. Databases do not contain the mechanisms to support continuous query processing. In a stream application, queries respond to data arriving continuously at high rates. To achieve good processing performance, i.e., handling input data within strict time bounds, a system should provide *incremental* processing to avoid considering the same data over and over again. In addition, it should scale to handle numerous queries at a time [33] as each query stays alive for a long time. Furthermore, environment and

workload changes call for adaptive processing strategies to achieve the best query response time. The hooks for building a continuous streaming application are not commonly available in *Database Management Systems* (*DBMS*) and thus the pioneering *Data Stream Management Systems* (*DSMS*) architects naturally designed systems from scratch giving birth to interesting novel ideas and system architectures [7, 8, 12, 13, 15, 20].

On the other hand, the current generation of data stream systems are purely specialized on query processing over streaming (temporary) data. By designing from scratch completely different architectures aimed at the specifics of streaming applications, very few of the existing techniques for relational databases were reused. This became more pressing as the stream applications demanded more database functionality and scalability, which called for more generic solutions. For this reason, [10] argues towards "mimicking" how traditional relational engines work by decoupling the storage management part from the query processing part in a stream engine towards a more generic system. A simple way to achieve processing integration of persistent and temporary data, is by externally connecting a stream engine with a *foreign-body* DBMS [5, 1, 11]. However, these are not scalable solutions for high volume data.

This way, a few efforts have emerged the last few years towards a complete integration of database and streaming technology [27, 14, 18]. The past years we are developing a system, named DataCell [27, 28], that integrates both streaming and database technologies in the most natural way; a fully functional stream engine is designed on top of an extensible DBMS kernel. The goal is to fully exploit the generic storage and execution engine as well as its complete optimizer stack. Stream processing then becomes primarily a query *scheduling* task, i.e., make the proper queries see the proper portion of stream data at the proper time. A positive side-effect is that our architecture supports SQL'03, which allows stream applications to exploit sophisticated query semantics.

Numerous research and technical questions immediately arise. The most prominent issues are the ability to provide specialized stream functionality and hindrances to guarantee real-time constraints for event handling. [27] illustrates the DataCell architecture and sets the research path and critical milestones.

Contributions. In this paper, we focus on the core of streaming applications, i.e., incremental stream processing and window-based processing. Window queries form the prime programming paradigm in data streams, i.e., we break an initially unbounded stream into pieces and continuously produce results using a *focus window* as a peephole on the data content passing by. Successively considered windows may overlap significantly as the focus window *slides* over the stream. It is the cornerstone in the design of stream engines and typically specialized operators are designed to avoid work when part of the data falls outside the focus window. Most relational operators underlying traditional DBMSs *cannot* operate incrementally without a major overhaul of their implementation. Here, we show that efficient incremental stream processing is, however, possible in a DBMS kernel handling the problem *at the query plan and scheduling level*. For this to be realized the relational query plans are transformed in such a way that the stream is broken into pieces and different portions of the plan are assigned to different portions of the focus window data. DataCell takes care that this "partitioning" happens in such a way that we can exploit past computation during future windows. As the window slides, the stream data also "slides" within the continuous query plan.

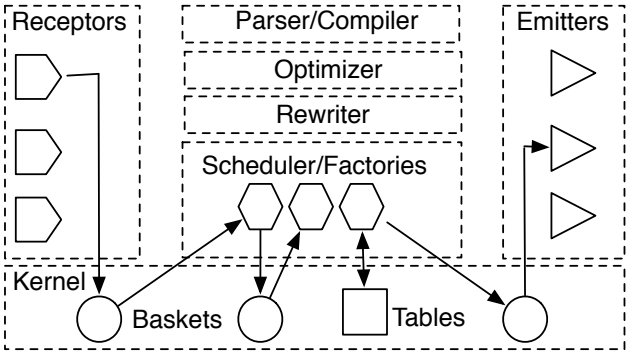


Figure 1: The DataCell Architecture

We discuss in detail our design of the DataCell prototype, which is based on the open-source column-store MonetDB. In particular, we illustrate the methods to extend the optimizer with the ability to create and rewrite them into incremental plans. A detailed experimental analysis demonstrates that DataCell supports efficient incremental processing, comparable to a specialized stream engine or even better in terms of scalability.

Outline. The remainder of this paper is organized as follows. Section 2 provides the necessary background. Section 3 discusses in detail how we achieve efficient incremental processing in DataCell followed by an experimental analysis in Section 4. Section 5 briefly discusses related work while Sections 6 and 7 discuss future work and conclude the paper.

2. BACKGROUND

A Column-oriented DBMS. MonetDB is a full-fledged column-store engine. Every relational table is represented as a collection of *Binary Association Tables* (*BATs*), one for each attribute. Advanced column-stores process one column at a time, using *late* tuple reconstruction, discussed in, e.g., [4, 23]. Intermediates are also in column format. This allows the engine to exploit CPU- and cache-optimized vector-like operator implementations throughout the whole query evaluation, using an efficient bulk processing model instead of the typical tuple-at-a-time volcano approach. This way, a select operator for example, operates on a single column, filtering the qualifying values and producing an intermediate that holds their tuple IDs. This intermediate can then be used to retrieve the necessary values from a different column for further actions, e.g., aggregations, further filtering, etc. The key point is that in DataCell these intermediates can be exploited for flexible incremental processing strategies, i.e., we can selectively keep around the proper intermediates at the proper places of a plan for efficient future reuse.

DataCell. DataCell [27] is positioned between the SQL compiler/ optimizer and the DBMS kernel. The SQL compiler is extended with a few orthogonal language constructs to recognize and process continuous queries. 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 plan.

Figure 1 shows a DataCell instance. It contains *receptors* and *emitters*, i.e., a set of separate processes per stream and per client, respectively, to listen for new data and to deliver results. They form the edges of the architecture and the bridges to the outside world, e.g., to sensor drivers.

The key idea is that when an event stream enters the system via a receptor, stream tuples are immediately stored in a lightweight table, called *basket*. By collecting event tuples into baskets, DataCell can evaluate the continuous queries over the baskets as if they were normal one-time queries and thus it can reuse any kind of algorithm and optimization designed for a DBMS. Once a tuple has been seen by all relevant queries/operators, it is *dropped* from its basket.

Continuous query plans are represented by *factories*, i.e., a kind of co-routine, whose semantics are extended to align with table producing SQL functions. Each factory encloses a (partial) query plan and produces a partial result at each call. For this, a factory continuously reads data from the input baskets, evaluates its query plan and creates a result set, which it then places in its output baskets. The factory remains active as long as the continuous query remains in the system, and it is always alert to consume incoming stream data when they arrive.

The execution of the factories is orchestrated by the DataCell scheduler, which implements a Petri-net model [30]. The firing condition is aligned to arrival of events; once there are tuples that may be relevant to a waiting query, we trigger its evaluation. Furthermore, the scheduler manages the time constraints attached to event handling, which leads to possibly delaying events in their baskets for some time. One important merit of the architecture, is the natural integration of baskets and tables within the same processing fabric. As we show in Figure 1, a single factory can interact both with tables and baskets. In this way, we can naturally support queries interweaving the basic components of both processing models.

By introducing the baskets, the factories and the scheduler, our architecture becomes able to handle data streams sufficiently, without losing any database functionality. This is the natural first step that covers the gap between the two originally incompatible processing models. However, numerous research and technical questions immediately arise. The most prominent issues are the ability to provide specialized stream functionality and hindrances to guarantee real-time constraints for event handling. In addition, we need to cope with (and exploit) similarities between standing queries, in order to deal with high performance requirements.

Albeit a clean and simple approach, by introducing the baskets, the factories and the DataCell scheduler, and by exploiting a column-store kernel optimized for modern hardware, DataCell is shown to perform extremely well, easily meeting the requirements of the Linear Road Benchmark in [27], without also losing any database functionality. In this paper, we focus on incremental processing for efficient and scalable window-based queries.

3. INCREMENTAL PROCESSING

Complete re-evaluation is the straightforward approach when it comes to continuous queries. The idea is simple; every time a window is complete, i.e., enough tuples have arrived, we compute the result over all tuples in the window. In fact, this is the way that any DBMS can support continuous query processing modulo the addition of certain scheduling and triggering mechanisms. In DataCell terms, this means that we let factories run every time we have enough new tuples for the window to slide and once the factory runs we remove all expired tuples from the baskets. For example, Algorithm 1 shows such a continuous re-evaluation query plan.

Although this could be sufficient for *tumbling* and *hopping* windows, i.e., windows that slide per one or more than a full window size at a time, it is far from optimal when it comes to the more common and challenging case of *overlapping sliding* windows. The drawback is that we continuously process the same data over and over again, i.e., a given stream tuple t will be considered by the same query multiple times until the window slides enough for t to expire. For this, we need efficient incremental query processing, a feature missing from DBMSs.

Algorithm 1 The factory for continuous re-evaluation of a tumbling window query that selects all values of attribute X in a range v_1-v_2.

```
input = basket.bind(X)
output = basket.bind(Y)

while true do
    while input.size < windowsize do
        suspend()

    basket.lock(input)
    basket.lock(output)
    w = basket.getLatest(input,windowsize)

    result = algebra.select(w,v1,v2)

    basket.delete(input,windowsize)
    basket.append(output,result)

    basket.unlock(input)
    basket.unlock(output)
    suspend()
```

Splitting Streams. Conceptually, DataCell achieves incremental processing by partitioning a window into n smaller parts, called *basic windows*. Each basic window is of equal size to the sliding step of the window and is processed separately. The resulting partial results are then *merged* to yield the complete window result.

Assume a window $W_i = w_1, w_2, \ldots, w_n$ split into n basic windows. After processing W_i, all windows after that can exploit past results. For example, for window $W_{i+1} = w_2, w_3, \ldots, w_{n+1}$ only the last basic window w_{n+1} contains new tuples and needs to be processed, merging its result with the past partial results. This process continues as the window slides.

Operator-level Vs Plan-level Incremental Processing. The basic strategy described above is generally considered as the standard backbone idea in any effort to achieve incremental stream processing. It has been heavily adopted by researchers and has lead to the design of numerous specialized stream operators (stream joins, stream aggregates, etc.), e.g., [17, 19, 21, 25, 36, 26].

Stream engines provide radically different architectures than a DBMS by pushing the incremental logic all the way down to the operators. Here, in the context of DataCell we design and develop the incremental logic at the query plan level, leaving the lower level intact and thus being able to reuse the complete storage and execution engine of a DBMS kernel. The motivation is to inherit all the good properties of the DBMS regarding scalability and robustness in heavy workloads as demanded by nowadays stream applications. In addition, an architecture such as DataCell is perfectly applied in scenarios that need to tightly combine both stream and database query processing model.

Algorithm 2 The plan for incremental evaluation of a simple window query that selects all values of attribute X in (v_1-v_2).

1: input = basket.bind(X)
2: output = basket.bind(Y)
3: **while** input.size $<$ windowsize **do**
4: suspend()
5: basket.lock(input)
6: basket.lock(output)
7: $w_1, w_2, \ldots, w_n$ = basket.split(input,n)
8: res_1 = algebra.select(w_1,v1,v2)
9: res_2 = algebra.select(w_2,v1,v2)
10: ...
11: res_{n-1} = algebra.select(w_{n-1},v1,v2)
12: **while** true **do**
13: **while** input.size $<$ windowsize **do**
14: suspend()
15: basket.lock(input)
16: basket.lock(output)
17: w_n = basket.getLatest(input,stepsize)
18: res_n = algebra.select(w_n,v1,v2)
19: result = algebra.concat(res_1,res_2,...,res_n)
20: $w_{exp} = w_1, w_1 = w_2, w_2 = w_3, \ldots, w_{n-1} = w_n$
21: $res_1 = res_2, res_2 = res_3, \ldots, res_{n-1} = res_n$
22: basket.delete(input,w_{exp})
23: basket.append(output,result)
24: basket.unlock(output)
25: basket.unlock(input)
26: suspend()

The questions to answer then are the following.

1) How can we achieve this in a generic and automatic way?

2) How does it compare against state-of-the-art stream systems?

In this section, we will describe our design and implementation over the MonetDB system where we extended the optimizer to transform normal plans into incremental ones which a scheduler is responsible to trigger. In the next section, we will show the advantages of this approach over specialized stream engines as well as the possibilities to combine those two extremes.

Plan Rewriting. The key point is careful and generic query plan rewriting. DataCell takes as input the query plans that the SQL engine creates, leveraging the algebraic query optimization performed by the DBMS's query optimizer. Fully exploiting MonetDB's execution stack, the incremental plan generated by DataCell is handed back to MonetDB's optimizer stack for physical plan optimization.

To rewrite the original query plan into an incremental one, DataCell applies four basic transformations; 1) Split the input stream into n basic windows, 2) Process each (unprocessed) basic window separately, 3) Merge partial results, and 4) Slide to prepare for the next basic window. Figure 2 shows this procedure schematically. For the first window, we run part of the original plan for each basic window while intermediates are directed to the remainder of the plan to be merged and execute the rest of the operators. As the window slides we need to process only the new data avoiding to reaccess past basic windows (shown faded in Figure 2) and perform the proper merging with past intermediates. Achieving this for generic and complex SQL plans is everything but a trivial task.

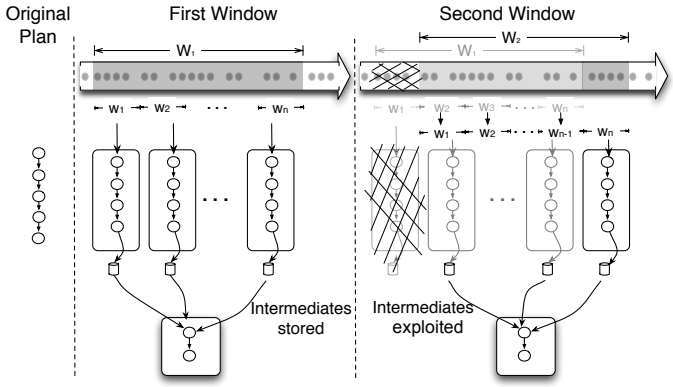


Figure 2: Incremental processing at the query plan level

Thus, we begin with an over-simplified example shown in Algorithm 2 to better describe these concepts.

Splitting. The first time the query plan runs, it will split the first window into n basic windows (line 7). This task is in practice an almost zero cost operation and results in creating a number of views over the base input basket.

Query Processing. The next part is to run the actual query operators over each of the first $n-1$ basic windows (lines 8-11), calculating their partial results. While in general more complicated (as we will see later on), for this simple single-stream, single-operator query the task boils down to simply calling the select operator for each basic window. For more complex queries, we will see that only part of the plan runs on every single basic window, while there is another part of the incremental plan that runs on merged results.

Basic Loop. The plan then enters an infinite loop where it (a) runs the query plan for the last (latest) basic window and (b) merges all partial results to compose the complete window result. The first part (line 18) is equivalent to processing each of the first $n-1$ basic windows as discussed above. For the simple select query of our example, the second part can create the complete result by simply concatenating the n partial results (line 19). We will discuss later how to handle the merge in more complex cases.

Transition Phase. Subsequently, we start the preparation for processing the next window, i.e., for when enough future tuples will have arrived. Basically, this means that we first shift the basic windows forward by one, as indicated in line 20 for this example. Then, more importantly, we make the correct correlations between the remaining intermediate results. This transition (line 21) is derived from the previous one. In the current example, both transitions are aligned, but in the case of more complex queries, e.g., joins, we need more steps to identify transitions (to be discussed later).

Intermediates Maintenance. Maintaining and reusing the proper intermediates is of key importance. In our simple example, the intermediates we maintain are the results of each select operator which are to be reused in the next window as well. In general, a query plan may have hundreds or even thousands of operators. The DataCell plan rewriter maintains the proper intermediates by following the path of operators starting from each basic window to associate the proper intermediates with the proper basic window

such as to know (a) how to reuse an intermediate and (b) when to expire it. This becomes a big challenge especially in multi-stream queries where an intermediate from one stream may be combined with multiple intermediates from other streams, e.g., for join processing (we will see more complex examples later on).

Continuous Processing. The next step is to discard the old tuples that expire (line 22) and deliver the result to the output stream (line 23). After that, the plan *pauses* (line 26) and will be resumed by the scheduler only when new tuples have arrived. Lines 13-14 ensure that the plan then runs only once there are enough new tuples to fill a complete basic window.

Discarding Input. In simple cases, as in the given example, once the intermediate results of the individual basic windows are created, the original input tuples are no longer required. Hence, to reduce storage requirements we can discard all processed tuples from the input basket, even if they are not yet expired, keeping only the respective intermediate results for further processing. Extending Algorithm 2 for achieving this is straightforward. A caveat seen shortly is that there are cases, e.g., multi-stream matching operations like joins, where we cannot apply this optimization, as we need access the original input data until it expires.

Generic Plan Rewriting. When considering more complex queries and supporting the full power of SQL, the above plan rewriting goals are far from simple to achieve. How and when we split the input, how and when we merge partial results are delicate issues that depend on numerous parameters related to both the operator semantics for a given query plan and the input data distribution.

This way, our strategy of rewriting query plans becomes as follows. The DataCell plan rewriter takes as input the optimized query plan from the DB optimizer.

(1) The first step remains intact; it splits the input stream into $n = |W|/|w|$ disjoint pieces.

(2) In a greedy manner, it then consumes one operator of the target plan at a time. For each operator it decides whether it is sufficient to replicate the operator (once per basic window) or whether more actions need to be taken.

The goal is to *split the plan as deep as possible*, i.e., allow as much of the original plan operators to operate independently on each basic window. This gives maximum flexibility and eventually performance as it requires less post processing with every new slide of the window, i.e., less effort in merging partial results.

To ease the discussions towards a generic and dynamic plan rewriting strategy, we continue by giving a number of characteristic examples where different handling is needed than the simplistic directions we have seen before. Figure 3 will help in the course of this discussion. Note, that we show only the pure SQL query expression, cutting out the full language statments of the continuous sliding window queries. In Figure 3 we represent the query plans for a variety of queries. For each query, we show the normal database query plan (non incremental) as well as the DataCell plan. The solid lines in the incremental query plan indicate the basic loop, i.e., the path that is continuously repeated as more and more tuples arrive. The rest of the incremental plan needs to be executed only the first time this plan runs.

Exploit Column-store Intermediates. Our design is on top of a column-store architecture. As we have already discussed in Section 2, column-stores exploit vector-based bulk processing, i.e., each operator processes a full column at a time to take advantage of vector-based optimizations. The result of each operator is a new column (BAT in MonetDB). In DataCell, we do not release these intermediates once they have been consumed. Instead, we selectively keep intermediates when processing one window to reuse them in future windows. This effectively allows us to put breakpoints in multiple parts of a query plan given that each operator creates a new intermediate. Subsequently, we can "restart" the query plan from this point on simply by loading the respective intermediates and performing the remaining operators given the new data. Which is the proper point to "freeze" a query plan depends on the kind of query at hand. We discuss this in more detail below.

Merging Intermediates. The point where we freeze a query plan practically means that we no longer replicate the plan. At this point we need to merge the intermediates so that we can continue with the rest of the plan. The merging is done using the `concat` operator. An example of how we use this can be seen in all instances of Figure 3. Observe, how before a `concat` operator the plan forks into multiple branches to process each basic window separately, while after the merge it goes back into a single flow. In addition, note that depending on the complexity of the query, there might be more than one flow of intermediates that we need to maintain and subsequently merge. For example, the plans in Figure 3(a), (b) and (e) have a single flow of intermediates while the plans in Figure 3(c) and (d) have two flows.

Simple Concatenation. The simplest case are operators where a simple concatenation of the partial results forms the correct complete result. Typical representatives are the select operator as featured in our previous examples, and any map-like operations. In this case, the plan rewriter can simply replicate the operation, apply it to each basic window, and finally concatenate the partial results. Figure 3(a) depicts such an example for a selection query.

Every time the window slides, we only have to go through the part of the plan marked with solid lines in Figure 3(a), i.e., perform the selection on the newest basic window and then concatenate the new intermediate with the old ones that are still valid. The transition phase which runs between every two subsequent windows guarantees that all needed intermediates and inputs are shifted by one position as shown in Algorithm 2.

Concatenation plus Compensation. The next category consists of operations that can be replicated as-is, but require some compensation after the concatenation of partial results to produce the correct complete result. Typical examples are aggregations like `min`, `max`, `sum`, as well as operators like `groupby`/`distinct` and `orderby`/`sort`. For these examples, the compensating action is simply applying the very operation not only on the individual basic windows, but also on the concatenated result as shown for `sum` in Figure 3(b). Other operations might require different compensating actions, though. For instance, a `count` is to be compensated by a `sum` of the partial results.

Note how Figure 3(b) actually combines the `sum` with a selection such that the selection is performed only on the basic windows, while the `sum`-compensation is required after the concatenation.

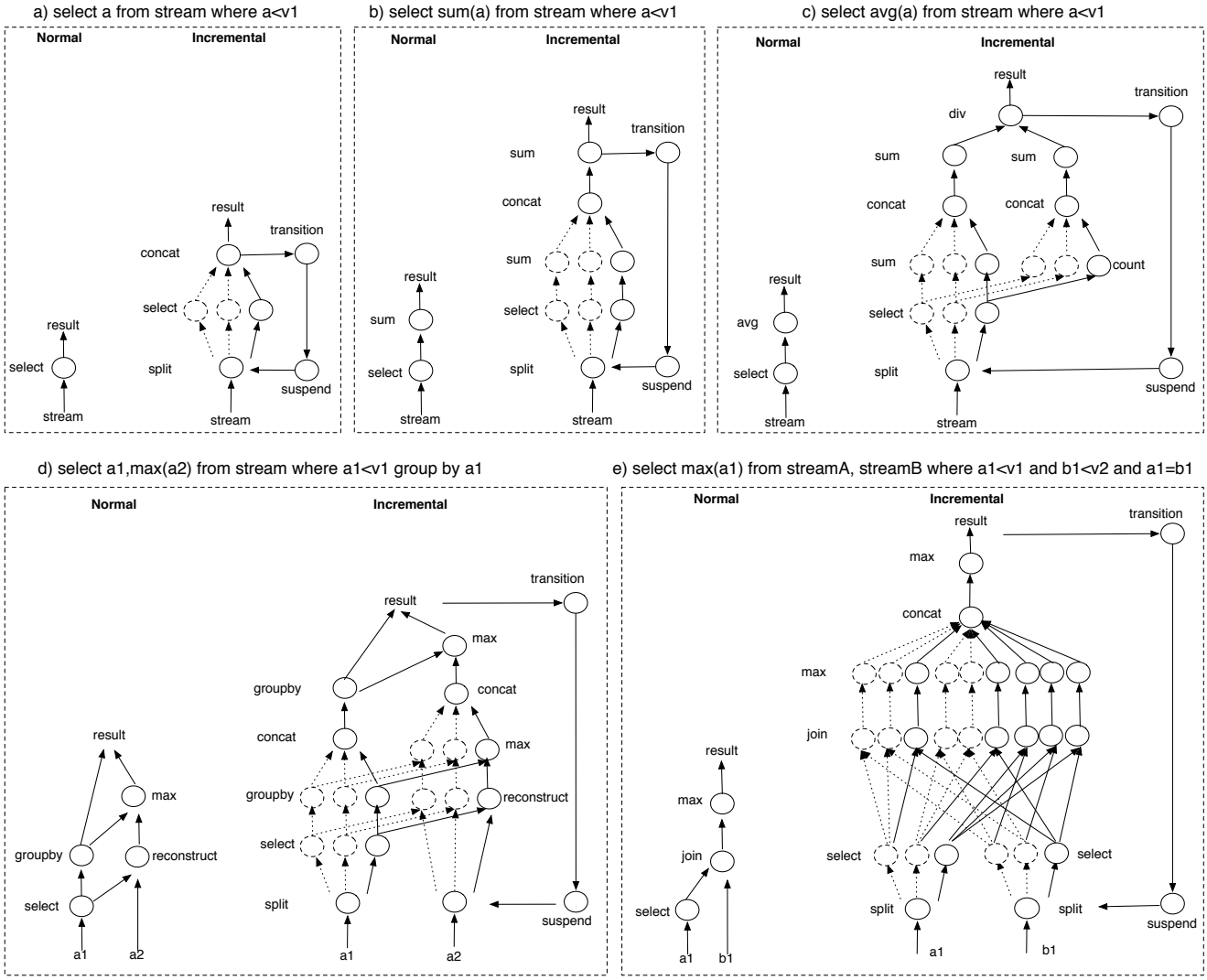


Figure 3: Examples of query plan transformations

Expanding Replication. A third category consists of operations that cannot simply be replicated to the basic windows as-is, but need to be represented by multiple different operations. For instance, Figure 3(c) sketches the incremental calculation of an `average`. Instead of simply replicating the `average` operation, we first need to calculate `sum` and `count` separately for each basic window, creating two separate data flows. Then, the global `sum` and `count` after concatenation are derived using the respective compensating actions as introduced above. Finally, dividing the global `sum` by the global `count` the two data flows are merged again to yield the requested global `average`.

Synchronous Replication. All cases discussed so far consider unary operations, either individually or in linear combinations, involving only a single attribute, and hence a single input data flow with columnar evaluation. Once multiple attributes are involved, we get multiple, possibly interconnected data flows as depicted for a grouped aggregation query in Figure 3(d). Canonically applying the rewrite rules discussed above, we can replicate the different data flows synchronously over the basic windows and use the compensating actions to merge the data flows into a single result just as in the original query plan.

Multi-stream Queries. All cases so far only consider a single data stream and (from an N-ary relational point of view) unary (i.e., single-input) operations. In these cases, it is sufficient to simply replicate the operations as often as there are basic windows. For multiple data streams and N-ary operations to combine them, the situation is more complex. Consider, for instance, the case of two streams and a join to match them as depicted in Figure 3(e). For simplicity of presentation we assume that both streams use the same window size $|W|$ and the same step size $|w|$. Given that we create the $n = |W|/|w|$ basic windows per stream as time slices, i.e., independently of the actual data (e.g., the join attribute values), we need to replicate the join operator n^2 times to join each basic window from the left stream with each basic window from the right stream. As with the other examples, the dashed operator instances in Figure 3(e) need to be evaluated only once during the initial preface. The solid operator instances need to be evaluated repeatedly, once for each step of the sliding window. Note that in this case

we cannot discard the selection results once the join has consumed them for the first time. Rather, they need to be kept and joined with newly arriving data until the respective basic windows expire.

Landmark Window Queries. Landmark queries differ from sliding window queries in that subsequent windows share the same fixed starting point ("landmark"), i.e., tuples do not expire per window step. Tuples either never expire, or at most very infrequently, and then all past tuples expire by resetting the global landmark.

Supporting such queries is straightforward in our design. Since data never expires, we do not have to keep individual intermediate results per basic windows to concatenate the active ones per step. Instead, we need to keep only one cumulative result for each `concat` operation in our DataCell plans in Figure 3. In fact, there is not even a need to split the preface in n basic windows. The initial window can be evaluated in one block; only newly arriving data is evaluated once a basic windows is filled as discussed above.

Time-based sliding windows. Our approach is generic enough to support both main sliding window types, i.e., count-based and time-based queries. In the first case, the window size and the sliding function are expressed in quantity of tuples, so counting and slicing the input stream is a straightforward process. In the case of time-based queries, the window parameters are defined in terms of time, e.g., a query with window size 1 hour that slides per 10 minutes. Once a tuple arrives into the system it is tagged with a timestamp that indicates its arrival time (we could also process the window based on the generation tuple time). The splitting of the stream data now happens by taking into account the tuple timestamps. We devide the stream into time intervals; say equal to the sliding period. This means that each basic window contains as many tuples as they arrived in the corresponding time interval. In this way, we could end up with unequally filled basic windows. After that point, DataCell processes the time-based window query following the same methodology we have discussed so far. Empty basic windows are recognized and simply skipped.

Optimized Incremental Plans. The decision to split the initial window into $n = |W|/|w|$ basic windows is purely driven by the semantics of sliding window queries. Further performance considerations are not involved. Consequently, the DataCell incremental plans as described so far start processing the next step only once sufficient tuples have arrived to fill a complete basic window. The response time from the arrival of the last tuple to fill the basic window until the result is produced is hence determined by the time to process a complete basic window of $|w|$ tuples (plus merging the partial results of all n active basic windows).

However, since tuples usually arrive in a steady stream, a fraction of the basic window could be processed before the last tuple arrives. This would leave fewer tuples to be processed after the arrival of the last tuple, and could hence shorten the effective response time.

In fact, the above described DataCell approach provides all tools to implement this optimization. The idea is to process the latest basic window incrementally just as we process the whole window incrementally. Instead of waiting for $|w|$ tuples, the basic loop is triggered for every $|v| = |w|/m$ tuples, splitting the basic window in m chunks. The results of the chunks are collected, but no global result is returned, yet. Only once m chunks have been processed, the m chunk results are merged into the basic window result, just like the n basic window results are merged into the window result above. Then, the n basic window results are merged and returned. This way, once the last basic window tuple has arrived, only $|v| = |w|/m$ tuples have to be processed before the result can be returned.

Choosing m and hence $|v|$ is a non-trivial optimization task. $m = |w|$ minimizes $|v|$ and thus the pure data processing after the arrival of the last tuple, but maximizes the overhead of maintaining and merging the chunk results. $m = 1$ is obviously the original case without optimization.

Given that both processing costs and merging overhead depend on numerous hardly predictable parameters, ranging from query characteristics over data distributions to system status, we consider analytical models with reasonable accuracy hardly feasible. Instead, we propose a dynamic self-adapting solution. Starting with $m = 1$, we successively increase m, monitoring the response time for each m for a couple of sliding steps. It is to be expected that the response times initially decrease with increasing m as less data needs to be processed after the arrival of the last tuple. Only once the increasing merging overhead outweighs the decreasing processing costs, the response times increase, again. Then, we stop increasing m and reset it to the value that resulted in the minimal response time. Next to increasing m linearly or exponentially (e.g., doubling with each step), bisection in the interval $[1, |w|]$ is a viable alternative for finding the best value for m.

Row-store Processing. This paper presents incremental processing in DataCell designed over a column-store. However, we do not see any reason why our techniques could not be applied in row-stores, too, possibly with some modifications to handle N-ary relations. The key point is to be able to split the stream and then "freeze" and "resume" execution of a plan at the proper points. Using MonetDB with its operator-at-a-time bulk processing and materialization of all (narrow) intermediate result columns, we get this "for free". Row-stores usually use a tuple-at-a-time volcano-style pipelining execution model. Hence, the major extension required is to introduce intermediate result materialization for each operator that precedes a `concat` operation in the incremental plans in Figure 3. While this used to be considered an unbearable overhead, row-stores implement similar techniques for sharing intermediate results for multi-query optimization, and recently we have seen successful exploitation of intermediates in eddies [16]. These ideas combined with the DataCell incremental processing scheme provide a promising set up for incremental processing in row-stores.

4. EXPERIMENTAL ANALYSIS

In this section, we provide a detailed experimental analysis of incremental processing in our DataCell implementation over MonetDB v5.15 All experiments are on a 2.4 GHz Intel Core2 Quad CPU equipped with 8 GB RAM and running Fedora 12.

Experimental Set-up and Outline. We compare incremental processing in DataCell against the typical re-evaluation approach which reflects the straightforward way of implementing streaming over a DBMS. We refer to the latter implementation as DataCellR in the rest of the experiments. In addition, we compare DataCell against a state-of-the-art commercial stream engine, clearly demonstrating the successful design of incremental processing over an extensible DBMS kernel and the potential of blending ideas from both worlds.

4.1 Re-evaluation Vs Incremental Processing

In the first part of the experimentation we will study DataCell and DataCellR to acquire a good understanding of how a typical DBMS

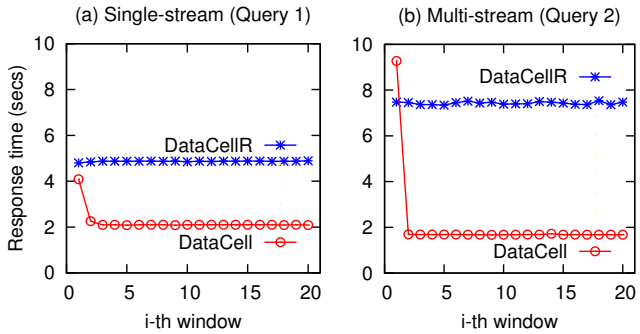


Figure 4: Basic Performance

performance can be transformed into an incremental one and the parameters that affect it. Given that these two implementations are essentially built over the same code base, this gives a clear intuition of the gains achieved by the incremental DataCell over a solid baseline. Then, with this knowledge in mind, in the second part we will see in detail how this performance compares against a specialized engine and what are the parameters that can swing the behavior in favor of one or the other approach.

We will use a single stream and a multi-stream query.

```
(Q1) SELECT x1, sum(x2)
     FROM stream
     WHERE x1 > v1 GROUP BY x1

(Q2) SELECT max(s1.x1), avg(s2.x1)
     FROM stream1 s1, stream2 s2
     WHERE s1.x2 = s2.x2
```

Basic Performance. The first experiment demonstrates the response times as the windows slide. Considering the single stream query first, we use a fixed window size, step and selectivity. Here, we use window size $|W| = 1.024 * 10^7$ tuples, window step $|w| = 2 * 10^4$ tuples, and 20% selectivity. This way, the DataCell plan rewriter splits the initial window into 512 basic windows. Each time the system gets $|w|$ new tuples, it processes them and merges the result with those of the previous 511 basic windows.

Figure 4(a) shows the response times for 20 windows. For the initial window, both DataCellR and DataCell need to process $|W|$ tuples and achieve similar performance. DataCell is slightly faster mainly because executing the group-by operation on smaller basic windows yields better locality for random access. For the subsequent sliding steps (windows 2-20), DataCellR shows the same performance as for the first one, as it needs to perform the same amount of work each time. DataCell, however, benefits from incremental processing, resulting in a significant advantage over DataCellR. Reusing the intermediate results of previously processed basic windows, DataCell only needs to process the $|w|$ tuples of the new basic window, and merge all intermediate results. This way, DataCell manages to fully exploit the ideas of incremental processing even though it is designed over a typical DBMS kernel. It nicely blends the best of the stream and the DBMS world.

For the double stream query, Query 2, we treat both streams equally, using window size $|W| = 1.024 * 10^5$ and window step $|w| = 1600$, i.e., the initial windows of both streams are split into 64 basic windows each. Figure 4(b) shows even more significant benefits for DataCell over DataCellR. The reason is that Query 2, is a complex multi-stream query that contains more expensive processing steps, i.e., join operators. DataCell effectively exploits the larger potential for avoiding redundant work.

The fact that incremental processing beats re-evaluation is not surprising of course (although later we will demonstrate the opposite behavior as well). What is interesting to keep from this experiment is that by applying the incremental logic at the query plan level we achieve a significant performance boost achieving efficient incremental processing within a DBMS.

Varying Query Parameters. The processing costs of a query depend on a number of parameters related to the semantics of the query, e.g., selectivity, window size, step size, etc. These are not tuning parameters, but reflect the requirements of the user. In general, the more data a query needs to handle (less selective/bigger windows, etc.), the more incremental processing benefits as it avoids processing the same data over and over again.

Selectivity. We start with Query 1, using a window size of $1.024 * 10^7$ tuples and a step of $2 * 10^4$ tuples. By varying the selectivity of the selection predicate from 10% to 90%, we increase the amount of data that has to be processed by the group-by and aggregation. Figure 5(a) shows the results. For both DataCellR and DataCell, the response times for a sliding step increase close to linear with the increasing data volume. However, the gradient for DataCellR is much steeper as it needs to process the whole window. Incremental processing allows DataCell to process only the last basic window, resulting in a less steep slope, and hence, an increasing advantage over DataCellR.

A similar effect can be seen with the join query in Figure 5(b). We use $|W| = 1.024 * 10^5$ and $|w| = 1600$ and vary the join selectivity from 10^{-5}% to 10^{-2}%. Due to the more expensive operators in this plan, the benefits of DataCell are stronger than before.

Window Size. For our next experiment, we use Query 1 with selectivity 20% and vary the window size. Keeping the number of basic windows invariant at 512, the step size increases with the total window size. Figure 6(a) reports the response time required for a sliding step using three different window sizes. The bigger the window, i.e., the more data we need to process, the bigger the benefits of incremental processing with DataCell over DataCellR materializing more than a 50% improvement. Again this clearly demonstrates the effectiveness of our incremental design using a generic storage and execution engine.

Landmark Queries. By definition, the window size of landmark queries increases with each sliding step, the step size is invariant. We run the following single-stream query as landmark query, using $|w| = 2.5 * 10^6$ and 20% selectivity.

```
(Q3) select max(x1),sum(x2)
     from stream where x1>v1
```

Figure 6(b) shows the response time for 40 successive windows. As in Figure 4, DataCellR and DataCell yield very similar performance for the initial window, where both need to process all data.

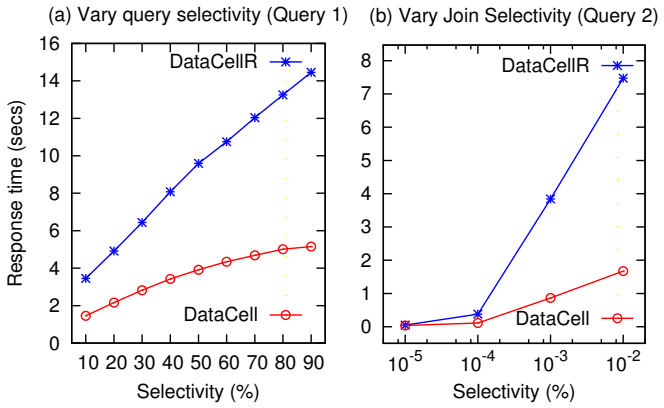


Figure 5: Varying Selectivity

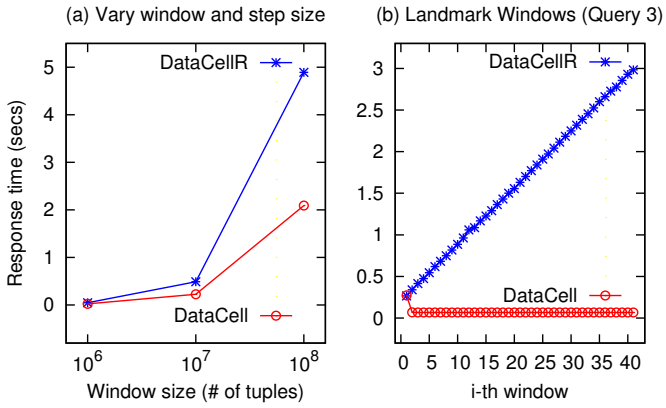


Figure 6: Varying Window and Step Size

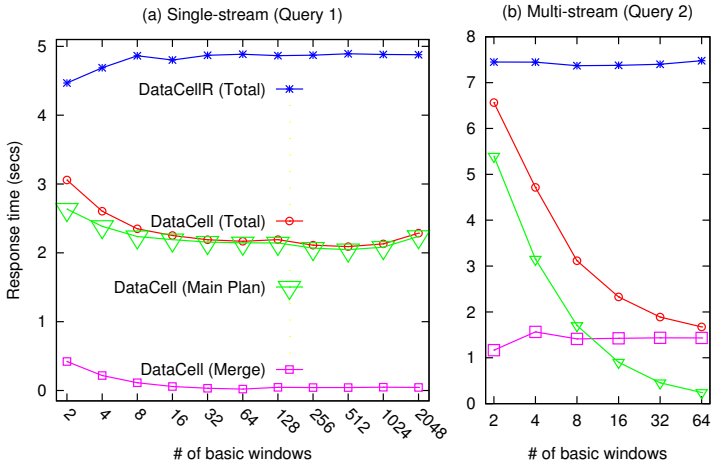


Figure 7: Decreasing Step (Incr. Number of Basic Windows)

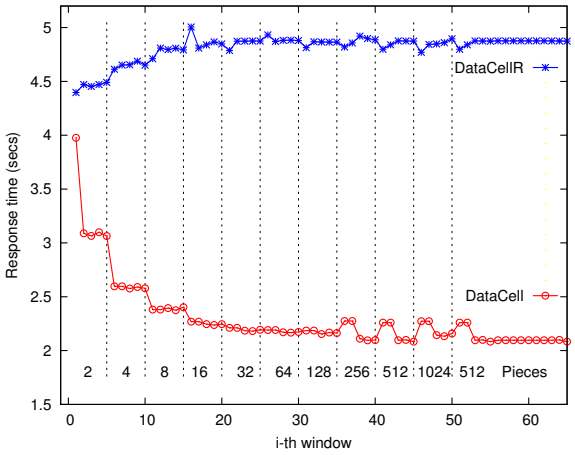


Figure 8: Query Plan Adaptation

The re-evaluation approach of DataCellR then makes the response time grow linearly with the growing window size. With DataCell, the response time for the second query drops to what is required to process only the new basic windows, and then stays constant at that level, exploiting the benefits of incremental processing.

With invariant window size, decreasing the step size in turn means increasing the number of basic windows per window, i.e., the number of intermediate results that need to be combined per step.

Figure 7(a) shows the results for Query 1. We use window size $w = 1.024 * 10^7$ tuples and a selectivity of 20%. With a small number of basic windows, i.e., with a big window step, we still need to process a relatively big amount of data each time a window is completed. Thus, response times are still quite high, e.g., for 2 basic windows. However, as the number of basic windows increases, DataCell improves quickly until it stabilizes once fixed initialization costs dominate over data-dependent processing costs.

Figure 7(a) also breaks down the cost of DataCell into two components. First, is the actual query processing cost, i.e., the cost spent on the main operators of the plan that represent the original plan flow. Second is the merging cost, i.e., all additional operators needed to support incremental processing, i.e., operators for merging intermediates, performing the transitions at the end of a query plan and so on. Figure 7(a) shows that the cost of merging becomes negligible. The main component is the query processing cost required for the original plan operators.

Notice also that there is a small rise in the total incremental cost with many basic windows (i.e., >1024 in Fig. 7(a)). This is attributed to the query processing cost which as we see in Figure 7(a) follows the same trend. What happens is that with more basic windows, a larger number of intermediates are maintained. Their total size remains invariant. However, with more basic windows, there are more (though smaller) intermediates to be combined and thus more operator calls required to make these combinations (the group-by) in this case. The administrative cost of simply calling these operators becomes visible with many basic windows.

Figure 7(b) shows a similar experiment for Query 2. Overall the trend is similar, i.e., cutting the stream window into smaller basic windows, brings benefits. The big difference though is that the break down costs indicate an opposite behavior than with Query 1. This time, the query processing cost becomes negligible while the merging cost is the one that dominates the total cost once the query processing part becomes small. The reason is that the intermediates this time are quite big, meaning that simply merging those interme-

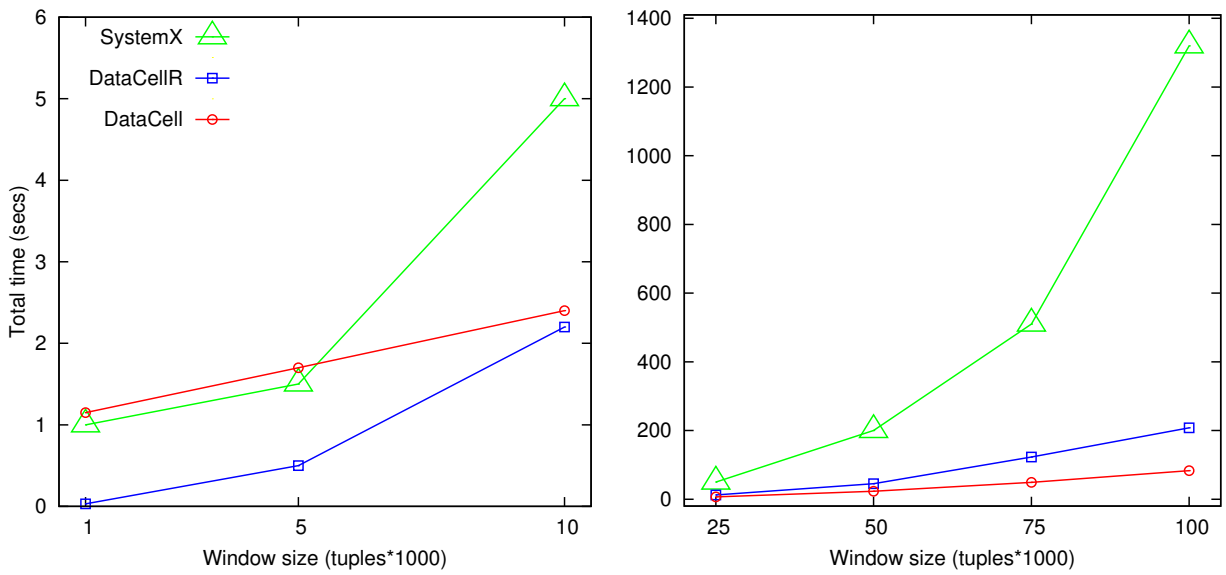


Figure 9: Against a Stream Engine

diates is significantly more expensive. This cost is rather stable given that the total size of intermediates is invariant with invariant window size, regardless of the step size.

Optimization. As discussed in Section 3 and supported by the results of the above experiments, the response time of incremental DataCell plans can further be improved by pro-actively processing the last basic window in smaller chunks than the step size defined in the query. This way, we favor a dynamic self-adapting approach over a static optimization using an analytical cost model. Figure 8 shows the results of an experiment where DataCell successively doubles the number m of chunks for a basic window every five steps as proposed in Section 3. By monitoring the response times and adjusting m, DataCell provides a steady performance improvement up to $m = 512$. With $m = 1024$, the performance starts degrading, triggering DataCell to resort to $m = 512$.

4.2 Comparison with a Specialized Engine

Here, we test our DataCell prototype against a state-of-the-art commercial specialized engine. Due to license restrictions we refrain from revealing the actual system and we will refer to it as SystemX. In addition, we tested a few open-source systems but we were not successful in installing and using them, e.g., TelegraphCQ [12] and STREAM [6]. These systems were academic projects and are not supported anymore making it very difficult to use them (in fact we are not aware of any stream papers comparing against any of these open-source stream systems). For example, TelegraphCQ compiled on our contemporary Fedora 12 system only after fixing some architecture-specific code. However, we did not manage to analyze and fix the crashes that occurred repeatedly when running continuous queries. System STREAM seemed to work correctly but the functionalities of getting the performance metrics did not work. The most important issue though is that it does not support sliding windows with a slide bigger than a single tuple. Nevertheless, we are confident that comparison against a most up-to-date version of a state-of-the-art commercial engine provides a more competitive benchmark.

For this experiment, we use the double stream Query 2. The metric reported is the total time needed for the system to consume a number of sliding windows and produce all results. Using a total of 100 windows and 64 basic windows per window, we vary the window size between $|\underline{W}| = 10^3$ and $|\overline{W}| = 10^5$ tuples with the respective step size growing from $|\underline{w}| = |\underline{W}|/64 \cong 16$ to $|\overline{w}| = |\overline{W}|/64 \cong 1600$ tuples. Thus, in total, we feed the system $|\underline{W}| + 100 * |\underline{w}| \cong 2600$ tuples in the most lightweight case and with $|\overline{W}| + 100 * |\overline{w}| \cong 260000$ tuples in the most demanding case.

Here, we test the complete software stack of DataCell, i.e., data is read from an input file in chunks. It is parsed and then it is passed into the system for query processing. The input file is organized in rows, i.e., a typical csv file. DataCell has to parse the file and load the proper column/baskets for each batch. Similarly for SystemX. For all systems, we made sure that there is no extra overhead due to tuple delays, i.e., the system never starves waiting for tuples, representing the best possible behavior.

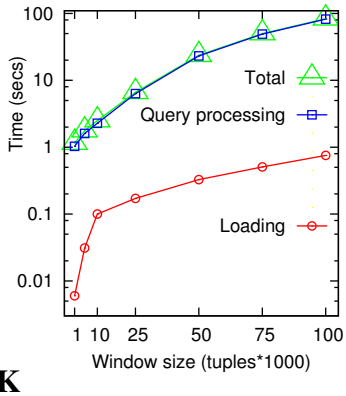
Figure 9 shows the results. It is broken down into Figure 9(a) for small windows, i.e., smaller than 10^4 tuples and into Figure 9(b) for bigger windows. For very small window sizes, we observe that plain DataCellR gives excellent results, even outperforming the stream solutions in the smaller sizes. The amount of data to be processed is so small that simply the overhead involved around the incremental logic in a stream implementation becomes visible and decreases performance. This holds for both DataCell and SystemX, with the latter having a slight edge for the very small sizes.

Response times though are practically the same for all systems as they are very small anyway. However, as the window and step size grow, we observe a very different behavior. In Figure 9(b), we see that plain DataCellR is losing ground to DataCell. This time, the amount of data and thus computation needed becomes more and more significant. The straightforward implementation of stream processing in a DBMS cannot exploit past computation leading to large total costs. In addition, we see another trend; DataCell scales nicely with the window size and now becomes the fastest system.

SystemX fails to keep up with DataCell and even plain DataCellR. When going for large amounts of data and large windows, batch processing as exploited in DataCell, gains a significant performance gain over the typical one tuple at a time processing of specialized engines. The main reason is that we amortize the continuous query processing costs over a large number of tuples as opposed to a single one. In addition, the incremental logic overhead is moved up to the query plan as opposed to each single operator.

Modern trends in data warehousing and stream processing support this motivation [35] where continuous queries need to handle huge amounts of data, e.g., in the order of Terabytes while the current literature on stream processing studies only small amounts of data, i.e., 10 or 100 tuples per window in which case tuple at a time processing behaves indeed well. An interesting direction is hybrid systems, i.e., provide both low-level incremental processing as current stream engines and high level as we do here, and interchange between different paradigms depending on the environment. For example, here DataCell merely needs to switch between re-evaluation and incremental processing when the window size becomes quite small or large respectively.

Finally, the figure on the right breaks down the DataCell costs seen in the previous figure into pure query processing costs and loading costs, i.e., the costs spent in parsing and loading the input file. We see that query processing is the major component while loading represents only a minor fraction of the total cost.



5. RELATED WORK

DataCell fundamentally differs from existing stream efforts [7, 8, 12, 13, 15, 20, 29], etc. by building on top of the storage and execution engine of a DBMS kernel. It opens a very interesting path towards exploiting and merging technologies from both worlds.

Compared to even earlier efforts on *active* databases, e.g., [32], DataCell adds support for specialized stream functionalities, i.e., incremental processing. Efforts in active databases strongly resemble the simplistic re-evaluation model we studied here as well.

Incremental processing in a DBMS has been studied in the context of updating materialized views, e.g., [9, 22], but there the scenario is very different given that it targets read-mostly environments whereas a stream scenario is by definition a write-only one.

Truviso Continuous Analytics system [18], a commercial product of Truviso, is another recent example that follows the same approach as DataCell. They extend the open source PostgreSQL DBMS [31] to enable continuous analysis of streaming data, tackling the problem of low latency query evaluation over massive data volumes. TruCQ integrates streaming and traditional relational query processing. It is able to run SQL queries continuously and incrementally over data while they are still coming and before they are stored (if needed) in *active database tables*. TruCQ's query processing significantly outperforms traditional *store-first-query-later* database technologies as the query evaluation has already been initiated when the first tuples arrive. It allows evaluation of one-time queries, continuous queries, as well as combinations of both types.

Another recent work, coming from the HP Labs [14], also confirms the strong research attraction for this trend. It defines an extended SQL query model that unifies queries over both static relations and dynamic streaming data, by developing techniques to generalize the query engine. It also extends the PostgreSQL database kernel [31], building an engine that can process persistent and streaming data in a uniform design. First, they convert the stream into a sequence of *chunks* and then continuously call the query over each sequential chunk. The query instance never shuts down between the chunks, in such a way that a cycle-based transaction model is formed.

The main difference of DataCell over the above two related efforts lies in the underlying architecture. DataCell builds over a column-store kernel using a columnar algebra instead of a relational one, bulk processing instead of volcano and vectorized query processing as opposed to tuple-based. Here we exploited all these architectural differences to provide efficient incremental processing by adapting the column-store query plans.

So Does one Size Fit all? In recent years, researchers have argued that the time has come that a generic DBMS can no longer provide efficient support for all possible kinds of application scenarios [34]. The path set by DataCell, Truviso and the HP effort does not oppose such ideas; quite the contrary. DataCell and other approaches add significant components that drastically change the architecture. However, critical core functionality is kept intact, e.g., operator implementations, optimizer modules, etc. This way, DataCell does not build on top of a closed DBMS architecture, but on top of an extensible database kernel.

6. FUTURE WORK

The road-map for DataCell research calls for innovation in many other important aspects of stream processing and the combination with already stored data. Thus, one can distinguish between challenges that come from the fact that stream processing is performed in a DBMS and challenges that arise by combining the two query processing paradigms in one.

Regarding the first challenge, the goal is to provide all essential streaming functionality and features without losing the DBMS strong storage and querying capabilities. We envision that a path where most of the functionality is provided via plan rewriting and minimal lower level operator changes is a promising one. For example, resource management, scheduling, and optimization in the presence of numerous queries is a critical topic. Similarly to incremental processing, this area has received a lot of attention with innovative solutions, e.g., [33]. DataCell offers all the available ingredients to achieve similar levels of multi-query optimizations while keeping the underlying generic engine intact. For example, a single factory (i.e., plan) may dynamically split into multiple pieces or merge with other relevant factories to allow for efficient sharing of processing costs leading to very interesting scenarios in how the network of factories and baskets is organized and adapts. Again, these issues can be resolved at a higher level through plan rewriting. The intermediates created for incremental processing can be reused by many queries, while partitioning and scheduling decisions can also adapt to the new parameters. For example in [24] we have presented a scheme of how multiple queries can cache and exploit intermediates in a column-store kernel. The basic concepts for multi-query processing in DataCell have been presented in [27].

Regarding the second challenge, we expect a plethora of rich topics to arise especially when optimization becomes an issue. For example, query plans that touch both streaming data and regular tables might require new optimizer rules or adaptations of the current ones. Overall, DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective.

7. CONCLUSIONS

In this paper, we have shown that incremental continuous query processing can efficiently and elegantly be supported over an extensible DBMS kernel. These results open the road for scalable data processing that combines both stored and streaming data in an integrated environment in modern data warehouses. This is a topic with strong interest over the last few years and with a great potential impact on data management, in particular for business intelligence and science. Building over an existing modern DBMS kernel to benefit from existing scalable processing components, continuous query support is the missing link. Here, we study in this context one of the most critical problems in continuous query processing, i.e., window based incremental processing.

Essentially, incremental processing is designed and implemented at the query plan level allowing to fully reuse (a) the underlying generic storage and execution engine and (b) the complete optimizer module. In comparison with a state-of-the-art commercial DSMS, DataCell achieves similar performance with small amounts of data, but quickly gains a significant advantage with growing data volumes, bringing database-like scalability to stream processing.

8. REFERENCES

[1] IBM InfoSphere Streams. http://www-01.ibm.com/software/data/infosphere/streams/.

[2] Large Hadron Collider. http://lhc.web.cern.ch/lhc/.

[3] Large Synoptic Survey Telescope. http://www.lsst.org.

[4] D. Abadi et al. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.

[5] M. H. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. *PVLDB*, 2(2):1558–1561, 2009.

[6] A. Arasu et al. STREAM: The Stanford Stream Data Manager. In *SIGMOD*, 2003.

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

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

[9] J. A. Blakeley et al. Efficiently Updating Materialized Views. In *SIGMOD*, 1986.

[10] I. Botan et al. Flexible and scalable storage management for data-intensive stream processing. In *EDBT*, 2009.

[11] I. Botan et al. A Demonstration of the MaxStream Federated Stream Processing Architecture (Demonstration). In *ICDE*, 2010.

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

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

[14] Q. Chen and M. Hsu. Experience in Extending Query Engine for Continuous Analytics. In *DaWaK*, pages 190–202, 2010.

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

[16] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[17] A. Dobra et al. Processing complex aggregate queries over data streams. In *SIGMOD Conference*, 2002.

[18] M. J. Franklin et al. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.

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

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

[21] L. Golab. *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo, 2006.

[22] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, 1995.

[23] S. Idreos et al. Self-organizing Tuple-reconstruction in Column-stores. In *SIGMOD*, 2009.

[24] M. Ivanova et al. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD*, 2009.

[25] J. Kang et al. Evaluating Window Joins over Unbounded Streams. In *ICDE*, 2003.

[26] J. Li et al. No pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Record*, 34:2005, 2005.

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

[28] E. Liarou et al. MonetDB/DataCell: Online Analytics in a Streaming Column-Store. *PVLDB*, 5(12):1910–1913, 2012.

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

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

[31] postgreSQL. http://www.postgresql.org/.

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

[33] M. A. Sharaf et al. Algorithms and Metrics for Processing Multiple Heterogeneous Continuous Queries. *ACM TODS*, 33(1), '08.

[34] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In *ICDE*, 2005.

[35] R. Winter and P. Kostamaa. Large scale data warehousing: Trends and observations. In *ICDE*, 2010.

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