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

Citation for published version (APA):

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

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 5

Incremental Processing in DataCell*

5.1 Introduction

In the two previous chapters, we described the basic DataCell architecture and the SQL-extended query language interface that allow us to formulate and submit continuous queries, encompassing streams and tables. However, numerous research and technical questions are still waiting for answers and solutions in the DataCell context. The most prominent issues are the ability to provide specialized stream functionality and hindrances to guarantee real-time constraints for event handling. Chapter 3 illustrates the DataCell architecture but leaves open issues related to real-time stream processing. Here, we make the next step towards a fully functional streaming DBMS kernel; we study how we can deal with incremental processing while staying faithful at the DataCell philosophy that dictates minimal changes to the underlying kernel.

*The material in this chapter has been the basis for a paper submitted for publication entitled “Enhanced Stream Processing in a DBMS Kernel” (Liarou et al., 2012a) and at the PVLDB12 paper “MonetDB/DataCell: Online Analytics in a Streaming Column-Store” (Liarou et al., 2012b).
5.1.1 Contributions

In this chapter, 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 specialized 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.

In this chapter, we illustrate the methods to extend the MonetDB/DataCell 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.

5.1.2 Outline

The rest of this chapter is organized as follows. Firstly, Section 5.2 presents a short recap of the notion of window-based stream processing. Then, Sections 5.3 and 5.4 discuss in detail how we achieve efficient incremental processing in DataCell. Section 5.6 provides a detailed experimental analysis. We compare the incrementalist DataCell kernel with our basic architecture (discussed in Chapter 3) and a specialized state-of-the-art commercial stream engine. Finally, Section 6 concludes the chapter.
5.2 Window-based Processing

Continuous computation of long standing queries in large scale streaming environments is a huge challenge from a data management perspective. Continuously considering all past data is not a scalable solution. Especially when it comes to blocking operators, e.g., a join, it is unrealistic to continuously analyze all data purely from a system resources point of view. This way, window-based queries have been introduced to assist efficient query processing in streaming environments. By windowing a continuous query, we delimit the boundaries of the initially unbounded stream and we continuously produce results on different portions of the data. Figure 5.1 shows simple examples of how window-based processing differs from “complete” stream processing.

Figure 5.1(a) shows the typical unbounded stream processing. This is often referred to as landmark window in the literature, i.e., the processing window is continuously growing. Figures 5.1(b) and (c) on the other hand, show window processing where as new data tuples arrive, some of the old ones expire. This way, a limited window of tuples is defined and the system is called to produce answers only for the tuples within the current window, ignoring the larger volume of past data preceding this window. The most straightforward type are tumbling windows (cf., Fig. 5.1(b)). Here, the size of the step, i.e., the number of tuples we move the window forward, is equal to the window size. This leads to non-overlapping windows of tuples, i.e., every tuple is considered (at most) once for a given query.

Other than making query processing possible by limiting the amount of processed data, window-based processing also raises a number of challenges. Especially, sliding window queries, i.e., queries where the step is smaller than the window such that subsequent windows overlap, lead to very interesting scenarios and processing challenges. Figure 5.1(c) shows an example of sliding
Algorithm 2 The factory for continuous re-evaluation of a tumbling window query that selects all values of attribute $X$ in range $v_1$-$v_2$.

1: input = basket.bind(X)
2: output = basket.bind(Y)
3: while true do
4:   while input.size < windowsize do
5:     suspend()
6:   basket.lock(input)
7:   basket.lock(output)
8:   $w = basket.getLatest(input,windowsize)$
9:   result = algebra.select($w,v_1,v_2$)
10:  basket.delete(input,windowsize)
11:  basket.append(output,result)
12:  basket.unlock(input)
13:  basket.unlock(output)
14:  suspend()

windows. The ideal goal is that every time we need to recompute the result of a query over the current window, we would like to analyze as little data as possible by cleverly exploiting past computation actions over previous windows that overlap with the current one.

In other words, the result of each window should be incrementally computed, by reusing valid past results. This incremental behavior is fundamental in all stream algorithms, techniques and systems. In addition, it is a functionality that is missing from a typical DBMS. Thus, it becomes a unique problem for the DataCell context as well.

5.3 Continuous Re-evaluation

Complete re-evaluation is the straight-forward approach when it comes to continuous queries for a DBMS engine. 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.

We can achieve this kind of processing by applying minimal changes to the existing DataCell architecture. Assuming, for the time being, single stream queries and tumbling windows, we only need to make sure that a query plan will “consume” \(|W|\) tuples of the input stream at a time, where \(|W|\) is the size of the window. This means that \(|W|\) tuples will be considered for query processing and subsequently \(|W|\) tuples are dropped from the input basket while of course the query plan will not run unless at least \(|W|\) tuples are present.

All this boils down to a set of simple rewriting rules for the continuous query plans of DataCell. For example, Algorithm 2 shows such a continuous re-evaluation query plan, for a simple window range query. The window semantics affect the plan only in such a way that it checks whether there are enough input tuples to fill a complete window (lines 4 and 5 in Algorithm 2). In addition, it only considers and subsequently drops \(|W|\) tuples at a time (lines 8 and 10 respectively in Algorithm 2).

To support also sliding overlapping windows with a step size of \(|w| < |W|\) tuples, only one more minor change is required, refining line 10 in Algorithm 2 as follows. Instead of deleting the complete window we would only delete the oldest \(|w|\) tuples that expire per step, namely the sliding step that encompass those tuples that are not valid in the next window.

This way, re-evaluation is quite simple to achieve in DataCell and as before the core of the query plan can be any kind of complex query, allowing DataCell to support the full strength of SQL and the complete optimizer module.

5.4 Incremental Processing

Although the direction seen in the previous section is 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 processing, a feature missing from typical DBMSs. Here, we discuss how we address this fundamental stream problem in DataCell.
5.4.1 The Goal

For ease of presentation, we begin with a high-level description of the technique at large, before we continue to discuss in more detail the various decisions and options.

The vision is to create a full-fledged stream engine without sacrificing any of the existing DBMS technology benefits. Our effort for incremental processing here successfully follows this path; without creating new specialized operators, we support sliding window queries by carefully rewriting and scheduling the existing DBMS query plans. This way, we can exploit all sophisticated query optimization techniques of a modern DBMS and all highly optimized operator implementations as well as query plan layouts.

5.4.2 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. Effectively, for each new window we only need to process the new tuples as opposed to the naive re-evaluation method that needs to process all window tuples repeatedly.

5.4.3 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 such as window stream joins and window stream aggregates, e.g., (Dobra et al., 2002; Ghanem et al., 2007; Golab, 2006; Kang et al., 2003; Zhu and Shasha, 2002; Li et al., 2005).

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.

The questions to answer then are:

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 in DataCell, where we extend the optimizer to transform normal continuous query 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.

5.4.4 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

4. Slide to prepare for the next basic window

Figure 5.2 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 5.2) and perform the proper merging with past intermediates. Achieving this for generic and complex SQL plans is everything but a trivial task. Thus, we begin with an over-simplified example shown in Algorithm 3 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 in MonetDB 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 by the previous one. In the current example both transitions are aligned, but in the case of more complicated queries (e.g., multi-stream query with join operators), we should carefully proceed this step.
Algorithm 3 The factory for incremental evaluation of a single stream 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:    $w_1, w_2, \ldots, w_n = \text{basket.split}(\text{input}, n)$
8:    res$_1 = \text{algebra.select}(w_1, v_1, v_2)$
9:    res$_2 = \text{algebra.select}(w_2, v_1, v_2)$
10:   \ldots
11:   res$_{n-1} = \text{algebra.select}(w_{n-1}, v_1, v_2)$
12: while true do
13:    while input.size < windowsize do
14:        suspend()
15:        basket.lock(input)
16:        basket.lock(output)
17:        $w_n = \text{basket.getLatest}(\text{input}, \text{steps}ize)$
18:        res$_n = \text{algebra.select}(w_n, v_1, v_2)$
19:        result = algebra.concat(res$_1$, res$_2$, \ldots, res$_n$)
20:        $w_{\text{exp}} = w_1, w_1 = w_2, w_2 = w_3, \ldots, w_{n-1} = w_n$
21:        res$_1 = \text{res$_2$, res$_2 = \text{res$_3, \ldots, res$_{n-1} = \text{res$_n$}$
22:        basket.delete(input, w_{\text{exp}})
23:        basket.append(output, result)
24:        basket.unlock(output)
25:        basket.unlock(input)
26:        suspend()

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 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 3 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.

5.4.5 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.

In 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 \( n = \frac{|W|}{|w|} \) 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. Figures 5.3, 5.4, 5.5, 5.6 and 5.7 will help in the course of this discussion through a
variety of queries. Note, that we show only the pure SQL query expression, cutting out the full language statements of the continuous sliding window queries. For each query, we show the reevaluated continuous query plan as well as the DataCell incremental 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.

5.4.6 Exploit Column-store Intermediates

As we have already discussed, our design is on top of a column-store architecture. 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.

5.4.7 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. Examples of how we use this can be seen in all instances of Figures 5.3 till 5.7. 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 5.3, 5.4 and 5.7 have a single flow of intermediates while the plans in Figure 5.5 and 5.6 have two flows.
5.4. INCREMENTAL PROCESSING

5.4.8 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 5.3 depicts such an example for a range query.

Every time the window slides, we only have to go through the part of the plan marked with solid lines in Figure 5.3, 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 intermediates needed and inputs are shifted by one position as shown in Algorithm 3.

Figure 5.3: Example of query plan transformations for range query
5.4.9 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 5.4. 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 5.4 actually combines the sum with a selection such that the
5.4. INCREMENTAL PROCESSING

Select avg(a) From stream Where a<v1

Figure 5.5: Example of query plan transformations for AVG function

selection is performed only on the basic windows, while the sum-compensation is required after the concatenation.

5.4.10 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 5.5 sketches the incremental calculation of 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
Select $a_1, \max(a_2)$ From stream Where $a_1 < v_1$ Group by $a_1$

Figure 5.6: Example of query plan transformations for GROUP BY query

the respective compensating actions as introduced above. Finally, dividing the global sum by the global count merges the two data flows, again, to yield the requested global average.

5.4.11 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 ag-
5.4. INCREMENTAL PROCESSING

select max(a1) from streamA, streamB where a1<v1 and b1<v2 and a1=b1

Figure 5.7: Example of query plan transformations for join query

ggregation query in Figure 5.6. 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.

5.4.12 Multi-stream Queries

All cases discussed above 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 5.7. For simplicity of presentation we assume that both streams use the same windows 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 5.7 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.

### 5.4.13 Landmark Window Queries

Landmark queries differ from sliding windows 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 \texttt{concat} operation in our DataCell plans in Figures 5.3 till 5.7. 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.

### 5.4.14 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., 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 input
stream now happens taking into account the tuple timestamps. We divide the stream into time intervals, let’s say equal to the sliding period. This means that each generated basic windows contains as many tuples as they arrived in the corresponding time interval, so we could end up with unequally filled in 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 skip processing.

5.4.15 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 $m$.

5.5 Optimizer Pipeline in DataCell for Incremental Query Plans

In this chapter, we presented the necessary transformation rules needed for the creation of incremental query plans for continuous sliding window queries. In this section, we discuss in more detail the optimization steps we implant in our MonetDB/DataCell experimentation platform for generic plan generation.

Recall the first DataCell implementation (see Section 3.4), where we needed to change the MonetDB optimizer, creating and adding new optimizer rules and defining a new optimizer pipeline as follows.

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

There, DataCell receives an one-time query plan which is produced by the MonetDB optimizer and it transforms it to a continuous query plan that works according to the re-evaluation logic. Now, we need to extend our optimization phase again with a new set of rules in order to support incremental stream processing for sliding window queries. The new optimizer pipeline we configure is the following.

```
datacellInc_pipe=inline,remap,evaluate,costModel,coercions,emptySet,aliases,deadcode,constants,commonTerms,datacell,emptySet,aliases,deadcode,datacellSlicer,mergetable,deadcode,datacellIncrementalist,reduce,garbageCollector,deadcode,history,multiplex
```
Compared to the first datacell pipe, the new pipeline for incremental plans contains three new optimizer rules i.e., datacellSlicer, mergatable and datacellIncrementalist. Their main role is to transform a continuous query plan to an incremental query plan. The main actions they take are as follows.

- Traverse the plan to find the baskets on which we apply the window predicate.

- For each window, split the input window into \( n \) pieces, each piece is equal to the sliding window step of the query. Conceptually the concatenation of the \( n \) pieces constitutes the original data in the window. Replace the original MAL instruction that materializes the window stream, with \( n \) instructions that slice the window into each of the \( n \) pieces.

- Traverse the plan and find which MAL plan instructions we should replicate, due to window splitting. These are the instructions where the original materialized window stream is involved (explicitly and implicitly).

- Merge the intermediate materialized result at the proper place of the query plan.

- Identify the original MAL plan instructions that cannot be replicated. Give them the proper merged input.

- Introduce the instructions for the transition phase. Starting from the source, i.e., slices, down to the intermediate results.

- Place the instructions that the engine should evaluate only once outside the infinite loop.

- Wrap the MAL instructions that correspond to the new portion of the data stream inside the infinite loop. Wrap inside the infinite loop the merging and the transition steps; they need to run continuously.

- Traverse the plan to find which slices are needed and which are not needed for the rest of the incremental query evaluation.

This is a quite complex process, since we have to traverse and transform the plan multiple times resulting to a significant makeover of the original query plan. The benefit of developing the transformation logic at the optimization phase, is that we can compile and transform any kind of complex queries automatically while still exploiting traditional DBMS optimization strategies.
Multi-query optimization for sliding window queries is an important area of data streams research. At this level, our implementation does not prevent automatic multi query optimization at the compilation phase.

5.6 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 DataCell incremental processing against the typical re-evaluation approach which reflects the straight-forward way of implementing streaming over a DBMS. In the rest of this section, we refer to the former implementation simply as DataCell\textsubscript{I} and the latter as DataCell\textsubscript{R}. 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.

We study in detail the effects of various parameters, i.e., query and data characteristics such as window size, window step, selectivity factors, etc. The performance metric used is response time, i.e., the time the system needs to produce an answer, once the necessary tuples have arrived.

In the first part of the experimentation we will study DataCell\textsubscript{R} and DataCell\textsubscript{I} to acquire a good understanding of how a typical DBMS 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
5.6. EXPERIMENTAL ANALYSIS

5.6.1 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 \(|W| = 1.024 \times 10^7\) tuples, window step \(|w| = 2 \times 10^4\) 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 5.8(a) shows the response times for 20 windows. For the initial window, both DataCell\(_R\) and DataCell\(_I\) need to process \(|W|\) tuples and achieve

(Q2) SELECT max(s1.x1), avg(s2.x1)
    FROM stream1 s1, stream2 s2
    WHERE s1.x2 = s2.x2
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), $DataCell_R$ shows the same performance as for the first one, as it needs to perform the same amount of work each time. $DataCell_I$, however, benefits from incremental processing, resulting in a significant advantage over $DataCell_R$. Reusing the intermediate results of previously processed basic windows, $DataCell_I$ only needs to process the $|w|$ tuples of the new basic window, and merge all intermediate results. This way, $DataCell_I$ 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 $|W| = 1.024 \times 10^7$ and window step $|w| = 1600$, i.e., the initial windows of both streams are split into 64 basic windows each. Figure 5.8(b) shows even more significant benefits for $DataCell_I$ over $DataCell_R$. 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.

5.6.2 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. In the following paragraphs, we discuss the most important of these parameters and their implications in detail.

Selectivity

We start with Query 1, using a window size of $1.024 \times 10^7$ tuples and a step of $2 \times 10^4$ 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 5.9(a) shows the results. For both DataCell\textsubscript{R} and DataCell\textsubscript{I}, the response times for a sliding step increase close to linear with the increasing data volume. However, the gradient for DataCell\textsubscript{R} 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 DataCell\textsubscript{R}.

A similar effect can be seen with the join query in Figure 5.9(b). We use $|W| = 1.024 \times 10^5$ 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 5.10(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 DataCellI 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 \( |w| = 2.5 \times 10^6 \) and 20% selectivity.

\[
(Q3) \text{select max}(x1), \text{sum}(x2) \\
\text{from stream where } x1 > v1
\]
Figure 5.10(b) shows the response time for 40 successive windows. As in Figure 5.8, MonetDB and DataCell yield very similar performance for the initial window, where both need to process all data. The re-evaluation approach of DataCell 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.

Step Size

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 5.11(a) shows the results for Query 1. We use window size $w = 1.024 \times 10^7$ 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, $DataCell_I$ improves quickly until it stabilizes once fixed initialization costs dominate over data-dependent processing costs.

Figure 5.11(a) also breaks down the cost of $DataCell_I$ 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 5.11(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. 5.11(a)). This is attributed to the query
5.6. **EXPERIMENTAL ANALYSIS**

processing cost which as we see in Figure 5.11(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 5.11(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 intermediates 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.

### 5.6.3 Optimization

As discussed in Section 5.4 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 5.12 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 5.4. Monitoring the response times, DataCell observes a steady performance improvement up to $m = 512$. With $m = 1024$, the performance starts degrading, triggering DataCell to resort to $m = 512$.

### 5.6.4 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 prototypes but we were not successful in installing and using them, e.g., TelegraphCQ and Streams. 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 Streams 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 our prototype.

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 \(|W| = 10^3|w| = 10^3|W| = 10^3|w| = 10^3|W| = 10^3\) tuples with the respective step size growing from \(|w| = W/64|w| = W/64|w| = W/64|w| = W/64\) tuples. Thus, in total, we feed the system \(|W| + 100 \cdot |w| \approx 2600 \text{ tuples in the most lightweight case and with } |W| + 100 \cdot |w| \approx 260000 \text{ tuples in the most demanding case.}

Previous experiments demonstrated purely the query processing performance. 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 5.13 shows the results. It is broken down into Figure 5.13(a) for small windows, i.e., smaller than \(10^4|w| = 10^4|w| = 10^4|w| = 10^4|w| = 10^4\) tuples and into Figure 5.13(b) for bigger windows. For very small window sizes, we observe that plain DataCell 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 \(\text{DataCell}_I\) 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 5.13(b), we see that plain DataCell is losing ground to \(\text{DataCell}_I\). This time, the amount of data and thus computation needed becomes more and more significant. The straight-forward implementa-
tion 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$_R$ and even plain DataCell. When going for large amounts of data and large windows, batch processing as exploited in DataCell$_I$, 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 (Winter and Kostamaa, 2010) 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.

Finally, Figure 5.14 breaks down the DataCellI 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.

### 5.7 Conclusions

In this chapter, 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.

The following chapter concludes the thesis and discusses a number of interesting open topics and research directions towards a complete data management architecture that integrates database and stream functionalities in the same kernel. DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective and by taking into account the needs of modern data management applications for scalable stream processing combined with traditional query processing. The range of topics discussed in this chapter include multi-query processing, adaptive query processing, query relaxation, distributed processing, and realizing DataCell in alternative architectures.