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 6

Conclusions and Future Research Paths*

In this thesis, we set the roots for a novel data management architecture that naturally integrates database and stream query processing inside the same query engine. As we discussed in the beginning of the thesis, there is a large demand nowadays to combine efficient and scalable stream and one-time processing. We start with a modern column-store architecture, realized in the MonetDB system, and we design our new system in this kernel. Column-store architectures offer the requirement for efficient one time processing and our main contribution here is the design of a column-store system that can do both stream and one-time processing efficiently.

The reason to choose this research direction comes from today’s application requirements to support both processing models providing advanced processing in both cases. So far the research community used to deal with this scenario in two ways. The first way is by trying to build specialized stream systems that in addition to stream processing provide simple processing of persistent/historical data. However, in this case, we are not able to reach the sophisticated techniques of mature database systems, especially when we need to support complex queries and/or big data analysis. An alternative direction is to externally con-

*Part of the material in this chapter has been presented at VLDB11 PhD Workshop paper “DataCell: Building a Data Stream Engine on top of a Relational Database Kernel.” (Liarou and Kersten, 2009) and at the PVLDB11 paper “The Researcher’s Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds” (Kersten et al., 2011).
nect and synchronize under the same middleware two specialized processing engines, i.e., a separate data stream engine and a separate DBMS, assigning different processing tasks to each one of them. The vision of an integrated processing model, has been considered in the past in the context of active databases and database triggers. However, it was soon rejected once the requirements of streaming applications became demanding for near real-time processing, multi-query optimizations and adaptive query processing; these are concepts that at this moment were new and different from the ones of the original database scenarios. In this thesis, we reconsider the path to implant on-line capabilities within a modern column-store database kernel in a way that we can efficiently synthesize and support interesting scenarios with streaming and database functionalities. In the DataCell project we exploit, reuse, redirect and extend the useful parts that the existing database technology already offers, to support a more complete query processing scenario, where the need of active and passive processing co-exist.

In this chapter, we will discuss and summarize our contributions to this research direction. We will also discuss a number of interesting future research topics towards scalable and efficient stream and one-time query systems, e.g., multi-query processing, adaptive query processing, query relaxation, distributed processing, etc.

6.1 Contributions

Basic DataCell Architecture

In this thesis, we introduced the basic DataCell architecture to exploit the notion of scalable systems that can provide both streaming and database functionality. We first showed the minimal additions that allow for stream processing within a DBMS kernel. The unique goal of DataCell is to exploit, as much as possible, all the available infrastructure offered by the underlying database kernel. In this way, we built our system using the majority of the original relational operators and optimization techniques, elevating the streaming functionality mainly at the query plan and scheduling level. The first DataCell architecture (Chapter 3) resulted in a model that allows to repeatedly run queries over incoming data as new data continuously arrives. Already this model was shown to provide substantially good streaming performance mainly by exploiting the power of modern column-store architectures. We were able to run the complete Linear Road benchmark and be well within the timing requirements set in the
6.2 LOOKING AHEAD

In this thesis, we made the first steps towards a complete data management architecture that integrates database and data stream functionalities in the same kernel. DataCell fundamentally changes the way that stream data is handled and processed, trying to exploit many traditionally core database techniques and ideas.

In this way, DataCell brings a significantly different view on how to build stream systems and radically changes the way we process data streams. Thus, it also brings the need to reconsider several of the well established techniques in the stream processing area. The road-map for DataCell research calls for innovation in many important stream processing areas. In the rest of this section, we will touch on these topics and where possible we will also provide discussion on possible research paths for solving these problems in the DataCell context.
6.2.1 Multi-Query Processing

A critical issue is that of multi-query processing and the rich scheduling opportunities that control the interaction between multiple continuous queries. In traditional stream processing, this area has received a lot of attention with several innovative solutions, e.g., (Sharaf et al., 2008). DataCell offers all the available ingredients to achieve similar levels of multi-query optimizations, while keeping the underlying generic engine intact. Below we discuss some of these directions.

Splitting and Merging Factories

Exploiting similarities at the query and data level is necessary in order to meet the real-time deadlines a stream application sets. In this way, we need to study mechanisms to efficiently and dynamically organize the queries in multiple groups based on their needs and properties. To accommodate partially overlapping queries we also need mechanisms to dynamically split and merge factories that wrap the query plans or parts of them.

DataCell here can adopt part of the existing literature in multi-query processing but there is also room to investigate research opportunities that arise from the basic DataCell processing model. One of the main innovations in DataCell comes from the choice to elevate several of the stream functionalities at the query plan and scheduling level. This allows for efficient reuse of core database functionalities and optimizations. In this way, one of the most challenging directions for multi-query processing in DataCell is the choice to split the plan of a single query into multiple factories. The motivation for this comes from different angles. For example, each factory in a group of factories sharing a basket, conceptually releases the basket content only after it has completed all operators in its query plan. Assume two query plans; a lightweight query $q_1$ and a heavy query $q_2$ that needs a considerably longer processing time compared to $q_1$. With the shared baskets strategy (see Section 3.3.2), we force $q_1$ to wait until $q_2$ finishes before we allow the receptor to place more tuples in the shared basket such that $q_1$ can run again. A simple solution is to split a query plan into multiple parts, such that part of the input can be released as soon as possible, effectively eliminating the need for a fast query to wait for a slow one.

Another natural direction once we decide to split query plans into multiple factories, is the possibility to share both the baskets and the execution cost. For example, queries requiring similar ranges in selection operators can be supported by shared factories that give output to more than one query’s factories.
Auxiliary factories can be plugged in to cover overlapping requirements.

**DataCell Cracking**

Another interesting direction for multi-query processing in DataCell is to exploit the idea of database cracking (Idreos, 2010). Database cracking was proposed as an adaptive indexing technique in the context of column-stores with bulk processing. The idea is that data is continuously physically reorganized building indexes incrementally and adaptively based on the requests of incoming queries. DataCell scheduling can exploit such ideas by allowing similar queries to run over the same baskets in a particular order. These queries can then use cracking-like ideas to continuously reorganize the basket and thus allowing successive queries to operate faster and faster for a given batch of incoming tuples. Challenges here include the dynamic scheduling of queries, i.e., which queries to allow to crack which baskets and in which order. Cracking is very sensitive in the order we process queries as this affects the kind of clustering and thus optimization achieved. Other challenges include finding a good balance between investment and amortization of the investment as in normal databases any index built can be exploited “forever”, while in our case the cracked baskets will only be useful for a given window of time.

**6.2.2 Adaptation**

Adaptive query processing is another very important issue in data streams. Dynamic changes to the arrival rate of data streams and on correlations between the incoming data, drastically affect the computation value of the continuously executed operations. In addition, new continuous queries are submitted over time while some of the old ones may expire and this changes the overall query processing behavior of the system. In this context, static query optimizations made up-front may not be valid after some time. Below we discuss some interesting directions for DataCell in this context.

**Adaptive Behavior in Traditional Streams**

Many academic prototypes presented extensive work on this topic. For example, StreaMon (Babu and Widom, 2004), the adaptive query processing infrastructure of STREAM (Arasu et al., 2003), collects statistics about stream and query plan characteristics and takes the appropriate actions to always ensure that the
query plan and memory usage are optimal for the current input characteristics. TelegraphCQ (Chandrasekaran et al., 2003) constructs query plans with adaptive routing modules, called Eddies (Avnur and Hellerstein, 2000). Thus, it is able to proceed to continuous run-time optimizations, dynamically adapting to the workload. Eddies modules adaptively decide how to route data to appropriate query operators on a tuple-by-tuple basis.

**Adaptive Behavior in DataCell**

Several key steps in the DataCell architecture are already adaptive in nature. Once a query is submitted in DataCell, it is parsed, compiled, optimized and then ends up to the pool with the other continuous queries, waiting to start processing incoming stream tuples. We first see an adaptive behavior when a factory considers how to proceed to the processing of the incoming chunk of data. It dynamically decides which way to evaluate the query, choosing between incremental processing and the re-evaluation method. As we have seen, window queries in periods with a low rate of incoming tuples can by default be executed according to the re-evaluation model. Once the arrival rate of the data streams becomes extremely high or bursty, the factory proceeds to a dynamic self-adaptive solution to find the optimal chunk size and proceed to incremental processing of the partial chunks.

By default DataCell starts with full re-evaluation, considering that the processing chunk is the same as the window size. Then, we successively modify the chunk size monitoring the response time for a couple of sliding steps. As long as the response times decrease by increasing the number of chunks in a window, we keep increasing this number. Only once the increasing merging overhead out-weights the decreasing processing costs, the response times increase, again. Then, we stop increasing the number of chunks and reset it to the value that resulted in the minimum response time.

**Adaptive DataCell Query Plans**

Most of the past work on adaptive query processing in stream systems naturally focuses on adaptive query plans, i.e., choosing different plan configurations for a given query depending on changes in the environment, the system, the data and the queries. The adaptive features discussed for DataCell above are mainly at a different level that has to do with the administration of the system and the resources.
6.2. LOOKING AHEAD

However, there is plenty of room for more optimization by considering adaptation at the query plan level too that goes beyond the choice of re-evaluation and incremental evaluation. For example, choosing different shape of query plans depending also on multi-query processing issues can be of crucial importance. Thus, again the choice of how to organize factories, how to dynamically split and merge query plans depending on the changes of the environment becomes an important issue.

At this point we should mention that given the modern column-store roots of DataCell, we already exploit some adaptive optimization at run time. Even if the query plan is static and optimized only once, at the submission time of the query, the operators are evaluated in a dynamic way. Given the bulk processing model, each operator knows exactly what is its input at execution time. For example, before executing a join we have first collected all tuples from both join inputs which means that we know their size, properties such as cardinality and possibly other data quality properties that allow us to dynamically decide the proper join algorithm.

However, full re-optimization and full adaptive query processing that allow the system to quickly adapt and continuously match the workload is a mandatory feature of modern stream engines. Here DataCell research can exploit ideas such as dynamic sampling and possible re-optimization if initial choices seem wrong, etc.

6.2.3 Dualism

In the DataCell context we have challenges that arise by combining the two query processing paradigms in one. More and more applications require this functionality and we can naturally expect that this will become a more mainstream processing model in the coming years. For example, this applies to scientific databases as well as in social networks where new data continuously arrives and needs to be combined with past data.

Once the technology of merging both continuous and one-time query processing becomes more mature, 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. There, all the choices made in respect to optimizing single continuous or one-time queries need reconsideration. Similarly for multi-query processing. Overall, DataCell opens the road for an exciting research path by looking at the stream query processing issue from a different perspective.
6.2.4 Query Relaxation

Pure stream systems traditionally focus on small scale applications with a rather small rate of incoming data. Nowadays, though, the requirements are changing towards systems that should be able to handle data streams of Terabytes on a daily basis. For example, scientific databases and large corporate databases create a huge pile of new data each day and need to run the same queries over and over again, combine past data with new ones and so on (Winter and Kostamaa, 2010).

Typical stream systems are not designed with such workloads in mind. With DataCell we make a significant step towards scalable stream processing by exploiting modern column-store features such as bulk processing and vectorized processing. However, as the data grows even more and in order to support new kinds of applications such as scientific databases we need to rethink certain query processing assumptions. For example, complete answers are often not possible due to the limited resources given the workload. Furthermore, the exploration and comprehension of data streams with a very high rate of incoming data, may lead to fundamentally different processing models.

In light of these challenges, we should rethink some of the strict requirements data stream systems adopted in the past. Next generation data stream management systems should interpret queries by their intent, rather than as a contract carved in stone for complete and correct answers. The continuously generated result sets should aid the user in understanding the stream trends and provide guidance to continue his data exploration journey as long as the stream is coming and his requirements are possibly modified. The stream engine would ideally interact with the users and help them continuously explore the streaming data in a contextualized way. In the rest of this subsection, we will discuss two possible directions towards more relaxed stream processing.

Approximate Kernels

One of the prime impediments to fast data exploration is the query execution focus on correct and complete result sets, i.e., the semantics of SQL presupposes that the user knows exactly what he expects and needs to monitor. The design and implementation of the query optimizer, execution engine, and storage engine are focused towards this goal. That is, correctness and completeness are first class citizens in modern data stream kernels. This means that when the system needs to perform a few hard and expensive unavoidable steps, it is designed to perform them such that it can produce the complete and correct re-
6.2. LOOKING AHEAD

sults. However, the query accuracy may have a significant impact on the query processing time that potentially will lead to deadline violations.

With input data sizes growing continuously, the research path of query approximation, was born such as to cope with the demanding short response times in stream applications. With huge data sizes that cannot be processed in a reasonable time load shedding has been widely adopted by the stream community as the most natural approach (Tatbul, 2007). There, we skip processing the whole input (e.g., by dropping tuples or creating tuple summarizations) aiming to save processing resources, even if this action will drastically affect our query answers. If the user accidentally chooses an expensive monitoring condition that produces a large result set, then a sample might be more informative and more feasible. Unfortunately, such a sample depends on the data distribution, the correlations, and data clustering in the data stream and the query result set. Taking a sample can still cause significant performance degradation that surface only at run time.

Current approximation techniques have only been studied for simple and small scale scenarios. Sampling and load shedding allow to drop part of the workload by completely ignoring certain incoming tuples. Summarization techniques create summaries over the data allowing to query the smaller summary and get a quick approximate response. For scientific databases though, even creating such summaries on the daily stream of Terabytes becomes a challenge on its own. Specifically, in stream processing it may not be worth creating summaries for small windows of data.

The above techniques require either a significant preprocessing step which can be prohibitive in large scale data or a strict up-front isolation of certain input parts. Here, we scrabble a novel direction where approximation becomes the responsibility of individual operators allowing a query processing kernel to self-organize and decide on-the-fly how to better exploit a given resource budget. For example, a hash-join may decide not to prompt the hash table for a given set of the inner, or after hitting a bucket where it has to follow a long list it may decide to skip this tuple of the inner.

The idea is to address the problem at its root; we envision a kernel that has rapid reactions on user’s requests. Such a kernel differs from conventional kernels by trying to identify and avoid performance degradation points on-the-fly and to answer part of the query within strict time bounds, but also without changing the query focus. Its execution plan should be organized such that a (non-empty) answer can be produced within $T$ seconds.

Although such a plan has a lot in common with a plan produced by a conventional cost-based optimizer, it may differ in execution order, it may not let
all operators run to completion, or it may even need new kinds of operators. In other words, an approximate kernel sacrifices correctness and completeness for performance. The goal is to provide a quick and fully interactive gateway to the data until the user has formulated a clear view of what he is really searching for, i.e., it is meant as the first part of the exploration process.

At this point note that the stream world has already sacrificed completeness and correctness when the window processing model was introduced in order to bound the infinite inputs. However, this has the same effect as with sampling and it cannot always guarantee good performance as the quality of the data may force expensive operations.

For example, very often during a plan we need to sort large sets of rowIDs to guarantee sequential data access. Those can be replaced by a cheaper clustering method or we can refrain from data access outside the cache. Likewise, operations dealing with building auxiliary structures over the complete columns/tables, can be broken up into their piecewise construction. Building just enough within $T$ to make progress in finding an answer. If $T$ is really short, e.g., a few seconds, the plan may actually be driven from what is already cached in the memory buffers. In a modern server, it is just too expensive to free up several Gigabytes of dirty memory buffers before a new query can start. Instead, its memory content should be used in the most effective way. In the remaining time the memory (buffer) content can be selectively replaced by cheap, yet promising, blocks. With a time budget for processing, the execution engine might either freeze individual operators when the budget has been depleted, or it might replace expensive algorithms with approximate or cheaper alternatives.

These ideas extend from high level design choices in operators and algorithms all the way to lower level (hardware conscious) implementation details. For example, during any algorithm if we reach the case where we need to extend, say, an array in a column-store with a realloc, an algorithm in that kernel may choose to skip this step if it will cause a complete copy of the original array.

This sketch is just the tip of the iceberg, i.e., numerous examples and variations can be conceived. The key challenge is to design a system architecture where budget distribution can be dynamically steered in such a way that the query still produces an informative result set. Aside from a tedious large-scale (re-)engineering effort to build a kernel on this assumption, major research questions arise. For example:

- How is the budget spread over the individual operators?
- What actions are operators allowed to take to stay within the budget?
6.2. LOOKING AHEAD

- How to harvest the system state produced by previous queries?
- How to replace the relational operators and index constructors with incremental versions?
- What all this means for dynamic and continuous adaptation of stream plans?
- How do such ideas combine with multi-query processing ideas in streaming environments?

At first sight the above ideas do not fit with the initial goal of DataCell to use existing and optimized database operators in order to exploit mature database technology. However, as we discussed in this section, the requirements of radically new applications such as scientific databases and social networks go way beyond what current technology can support which implies that drastic changes are required. Ideas such as the one discussed above apply both to traditional databases and to stream processing. For example, in the context of the DataCell the basic architecture could remain the same while the underlying core operators are updated to their approximate alternatives.

Query Morphing

In the same spirit as with the approximate kernels ideas presented above, we can also extend the ideas of approximate query processing to the actual patterns of the queries posed by the user. In this paragraph, we scrabble the vision where a stream processing kernel participates more actively in the complete query processing experience of the user, offering an additional mechanism that provides query pattern suggestions. According to the standard way a DSMS works, a user should first have a general clear idea of what to expect from the incoming data stream and then formulate and submit the corresponding continuous queries. However, when we are dealing with streams with high rates of incoming tuples, and our perspective on incoming data is not still clear or may be drastically modified depending on dynamic conditions, we could end up “missing” valuable stream data and wasting resources on analysis that ends up not being useful. For example, this can easily happen when our original continuous queries are not representative enough, of what we really wanted to monitor giving zero-hit or mega-hit result sets. This phenomenon is typical in exploratory scenarios such as scientific databases. Expensive query processing, in conjunction with the rapidly incoming data streams, trigger the vision for a data stream kernel that becomes a query consultant.
We introduce the notion of *query morphing* as an integral part of query evaluation. It works as follows, the user gives a starting query $Q$ and most of the effort $T$ is spent on finding the "best" answer for $Q$. But a small portion is set aside for the following exploratory step. The query is syntactically adjusted to create variations $Q_i$, i.e., with a small edit distance from $Q$. The process of query morphing is visualized on the left part of Figure 6.1. The user’s original request for a window stream returns the result set depicted by the small red circle. However, the processing kernel grabs the chance to explore a wider query/data spectrum in parallel, providing additional results for queries that belong in the close area, surrounding the original continuous query. The arrows that start from the red circle indicate this edit area in our example. In this way, the user also receives the orange elliptic query results that correspond to variations of his original request. In the right part of above figure, we see that the user may as a next step decide to shift his interest towards another query result, inspired by the result variations. A new query area now surrounds the user’s request, including both past and new variations of the query. This feature is very useful, once the user wants to monitor the incoming stream in a wider range and not stuck to his original request. This is not a one-time processes, as long as the input stream flows different trends could be identified in a continuously modified context.

Several kinds of adjustments can be considered to create the query variations, e.g., addition/dropping of predicate terms, varying constants, widening constants into ranges, joining with auxiliary tables through foreign key relationships, etc. The kind of adjustments can be statistically driven from the original submitted continuous queries, combinations of queries submitted by different source, or cached (intermediate partial) results. Since we have already spent part of our time on processing $Q$, the intermediates produced along the way can also help to achieve cheap evaluation of $Q_i$. 

Figure 6.1: Query Morphing
Of course, another crucial topic here is how this relates with the continuous adaptation nature that a stream system should have, i.e., how these exploratory query suggestions fit within the grand picture of continuously optimizing stream performance as the environment changes. Similarly, for multi-query processing there are opportunities to grab suggestions by exploiting multiple existing continuous queries and essentially transferring knowledge from one query pattern to the next. In other words, we can use the network of queries in order to provide suggestions about interesting queries or result sets.

The approach sketched aligns to proximity-based query processing, but it is generalized to be driven by the query edit distance in combination with statistics and re-use of intermediates. Query morphing can be realized with major adjustments to the query optimizer, because it is the single place where normalized edit distances can be easily applied. It can also use the plan generated for $Q$ to derive the morphed ones. The ultimate goal would be that morphing the query pulls it in a direction where information is available at low cost. In the ideal case, it becomes even possible to spend all time $T$ on morphed queries.

### 6.3 Distributed Stream Processing

With data volumes continuously growing, there is a pressing need to look into scalable query processing. The approximate query processing ideas described in the previous section, are a step in this direction.

However, there are more options to consider. For example, distributed query processing has always represented a good approach in supporting bigger data and query loads for any data management system. For example, some academic prototypes, e.g., Borealis (Abadi et al., 2005), and commercial systems, e.g., (Gedik et al., 2008), have focused on this topic. Many of the ideas that have been already proposed for distributed stream processing can also be applied in our context. The stream engine of DataCell can become the central processing part of each node and issues related to how we should distribute the data and the queries, to coordination, to communication protocols, and to fault tolerance can be handled in at a higher level, i.e., the core of the DataCell does not have to change.
6.4 DataCell in Different Database Kernels

The DataCell architecture is designed over modern column-store architectures. This fact raises a valid and at the same time important question whether our ultimate vision to include efficient stream processing in the heart of a traditional DBMS is limited to the underlying column-store architecture. The questions we should answer here are the following.

- Can we apply the same ideas on top of a row-store DBMS?
- How critical and unique are the column-store architecture advantages which enable DataCell?

In the basic DataCell architecture, each query is encapsulated into a factory, i.e., a function that wraps a continuous query plan in an infinite loop. Streaming data is temporarily collected into baskets and remains there until its consumption by the connected factories/operators. Baskets and tables can harmoniously coexist and interact, while the optimization algorithms are applied with minimal changes to both one-time and continuous queries. Note, that the purely stream specialized optimizations, i.e., incremental processing, are only applied to the corresponding queries.

The DataCell philosophy as briefly summarized above seems that could easily be applied in a row-store architecture, too. The existence of an intermediate scheduler, that orchestrates the waiting factories flourished by efficient scheduling policies, should certainly be implanted within the database software stack. The parser should be extended in a way to understand and differentiate the streaming from the persistent data, and the different query types. Thus, transforming a passive data management system to an active one, seems a general method that consists of a few straightforward key steps, and can easily be applied to any extensible system.

However, one of the main differences between DataCell’s underlying column-store kernel with other relational row-oriented DBMSs, is the core processing model that they obey. DataCell builds over a column-store kernel using, bulk processing instead of volcano-style pipelining execution model and vectorized query processing as opposed to tuple-based. It relies on operator-at-a-time bulk processing and materialization of all (narrow) intermediate result columns. DataCell adopts the column-at-a-time processing principle, adapting it to the streaming singularity. Thus, without waiting “forever” to fill in the (streaming attributes) columns with streaming data, it gets as many data are available into chunks when the triggering condition occurs and evaluates the query plans, in
6.4. DATACELL IN DIFFERENT DATABASE KERNELS

a Volcano-style iteration. This logic, is quite different than the original tuple-at-a-time model where the individual operations are invoked separately for each tuple. However, this fundamental difference is not a prohibitive factor to proceed to the streaming transformation of a row-oriented DBMS; all we need is to simply be able to materialize intermediates for incremental processing and introduce a mechanism for batch processing.

The tuple-at-a-time model guarantees near real-time processing in a typical stream application; it immediately processes each tuple once it arrives. However, there is a drawback coming from the need to repeatedly call all operators. This can potentially affect scalability.

Our underlying column-store architecture constitutes a crucial feature to support DataCell’s incremental processing requirement. Intermediates are also in column format. In this way, we did not need to change to original relational operators, since we keep in a natural way the required intermediate state of the partial operator evaluation into the corresponding intermediate columns/baskets inside each factory. The operators access only the newly appended streaming data and they merge the new results with the previous ones to update the result set. The key point is to be able to split the stream and then “freeze” and “resume” execution of a plan at the proper points.

Hence in a row-store implementation, the major extension required is to introduce intermediate result materialization for each operator that precedes a concat operation in the incremental plans. 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 (Deshpande and Hellerstein, 2004).

Other than design issues, using column-store or row-store as the underlying architecture comes with all the benefits or the overheads of the respective design. Row-stores and column-stores clearly represent the extremes of the database kernel architecture design space. For example, depending on the workload there may be less I/O and memory bandwidth requirements for a column-store but at the same time a row-store may have less requirements for intermediates materialization and thus less memory requirements. As such, another interesting direction for DataCell is the application of the DataCell philosophy in the more recent efforts that try to build hybrid database architectures. Again, the features of bulk processing, selective intermediates materialization and the ability to pause and resume execution, are all necessary for the core DataCell functionality.
6.5 Summary

DataCell makes the first steps towards a complete data management architecture that integrates database and stream functionalities in the same kernel. It fundamentally changes the way that stream data is handled and processed, trying to exploit many traditionally core database techniques and ideas. In this thesis, we made the strong statement that it is possible to implant stream processing functionalities in the heart of a modern database kernel and achieve both state of the art one-time query performance and stream query performance.

By relying on previous major efforts made from the database community during the last decades, we can bring several advantages on the stream processing front. So far, we made the first crucial steps to this direction. However, plenty of research challenges arise. The various open topics described in this chapter show the research path towards a fully integrated architecture where complex and hybrid stream-database scenarios will be expressed and performed. Overall, 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 for scalable stream processing combined with traditional query processing.