Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams

Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel, Udo Wolske
University of Marburg, Germany
{cammert,heinzch,kraemerj,seeger,sonny,wolske}@mathematik.uni-marburg.de

Abstract

A variety of real-world applications share the property that data arrives in form of transient streams. Data stream management systems (DSMS) provide convenient solutions to the problem of processing continuous queries on those streams. Within a DSMS, the scheduling of the queries and their operators has proved to be of utmost importance. Previous approaches addressing this issue can be divided into two categories: either each operator runs in its own thread or all operators, combined in one query graph, run in a single thread. Both approaches suffer from severe drawbacks concerning the thread overhead on the one hand and the stalls due to expensive operators on the other hand. To overcome these drawbacks, we propose in this work a hybrid approach that flexibly assigns threads to subgraphs of the query graph. We complement this approach with a suitable strategy to determine these subgraphs. The results of an experimental study substantiate the feasibility of our approach and its superiority to previous approaches.

1 Introduction

A variety of real world applications rely on an adequate processing of volatile data streams (DS). Consider for example traffic monitoring or intrusion detection in computer networks. The following characteristics of DS render their processing by means of standard database technology impossible: (1) the elements arrive autonomously, (2) their volume is potentially unbounded (3), and their arrival rates may abruptly change. For these reasons several prototypical data stream management systems (DSMS) have been developed, e.g. [4, 7, 12, 8, 11]. To cope with the transient and volatile nature of streams and to avoid the risk of system overload, a DSMS must be equipped with flexible adaptation mechanisms for resource management.
In this paper, we investigate an important aspect within DSMS, namely operator scheduling. Operator scheduling has proved to be a key component of a suitable resource management. Previous work on scheduling in DSMS mainly deals with the development of strategies that optimize data processing in terms of cost metrics like latency or memory usage, e.g. [3, 6, 10]. However, even though strategies for adaptive resource management exist, little work has been done to develop scalable scheduling architectures. The scheduling architectures for DSMS that have been proposed so far use, for processing, either one thread for each operator in the DSMS or one thread for one graph that unifies all queries in the DSMS. Therefore, we call these approaches *operator-threaded scheduling (OTS)* and *graph-threaded scheduling (GTS)* respectively. However, both approaches do not scale well for large query graphs.

- OTS does not scale for query graphs with a large number of operators due to the significant overhead for the thread management. In accordance with [6], we are not aware of any platform that can handle a large number of threads effectively.
- GTS faces two problems that adversely affect its scalability. Since one thread manages the complete query graph, it can only run one operator exclusively. Thus, the execution of one expensive operator can stall the processing of the whole DSMS. Furthermore, GTS cannot take advantage of new features of today's CPUs like multiple processors or hyperthreading.

We overcome these drawbacks with our flexible *hybrid multi-threaded scheduling* (HMTS) approach. It provides a proper adjustment of the number of threads for processing, in compliance with the current needs of the DSMS. This usually leads to a number of threads, which lies between the two extreme cases of OTS and GTS.

The contributions of this paper are as follows:

- Overall, we introduce a three level scheduling framework called HMTS that supports the flexible assignment of threads to subgraphs of the query graph at runtime.
- We present a sophisticated heuristic to determine a suitable partitioning of the query graph.
- We discuss the core results of an experimental study that shows the superiority of HMTS to GTS and OTS.

The rest of the paper is organized as follows. We start with a discussion of the query processing paradigms and the concept of Virtual Operators (VOs) in Section 2 and 3 respectively. The concept of VOs is essential for understanding our HMTS framework. Then, we show the motives for our HMTS framework in Section 4. The questions where to place queues is the topic of Section 5. In Section 6, we report some results of an experimental study. Section 7 provides a survey of related work. Finally, we present our conclusions as well as future work in Section 8.

2 Query processing techniques

In this section, we introduce query graphs and their use within a DSMS. We outline two data processing paradigms relevant in this context. Using one of them as point of origin, we introduce our approach for processing the query graph.

2.1 Query Graphs

To run multiple continuous queries in a DSMS, the common approach is to unify them in a query graph, in particular to gain advantage from subquery sharing. As an example, Figure 1 displays a query graph, where the results of a join are shared by the subsequent three operators.
Formally, a *query graph* is a directed acyclic graph. Its nodes are sources, operators (e.g. selection, join), and sinks; the edges between them represent the data flow. *Sources*, such as sensors, only deliver data, while *sinks* only consume data. Thus, sources are always at the bottom of the query graph whereas sinks are at the top (see Figure 3). Operators are sources and sinks as well; they receive data from their sources, process it, and deliver the results to their sinks.
In order to execute a query graph, *pull-based processing* or *push-based processing* can be used. These processing paradigms were introduced in [9] in the context of DBMS.

2.2 Pull-Based Processing

Previous approaches, e.g. [6, 14] propose to implement a query graph with pull-based operators that satisfy the open-next-close interface (ONC-operators). These operators are connected with intermediate queues. During runtime, the scheduler invokes an operator, which reads elements from its input queues, processes them, and writes the results to its output queue.

A problem not apparent at first sight is the changed semantic of operators if pull-based processing is used in a DSMS. The implementation of ONC operators needs a slight adaptation to be applied in a DSMS. The reason is that the semantics of the `hasNext` method is ambiguous; the result false can mean that currently no element is in the operators input queues (or no element in the queues qualifies as output) as well as that no element will be delivered anymore. This problem can be solved in different ways. For this paper, the concrete solution is not important, but the fact that this problem originates from using ONC operators in a DSMS. We assume that the problem has the following solution: the `hasNext` method returns false if no element will be delivered anymore. An empty queue is signed with a special element which only carries this information; it does not affect the results computed by the operator.

2.3 Push-based Processing

Another approach to realize a query graph is to utilize so-called push-based operators. A push-based operator receives each element delivered by one of its sources, processes it, and delivers the results to its sinks. If a query graph is implemented with intermediate queues between each pair of adjacent operators, its sinks are output queues for temporary storage. In this case, the main difference from the scheduler's point of view is that the queues must be invoked instead of the operators.

2.4 Our Approach

Let us now present our push-based approach for processing a query graph. In contrast to the approach sketched in the previous section, we do not place intermediate queues between operators by default. Instead, we let an operator invoke its successors. Therefore, an incoming element at an operator triggers a chain reaction, resulting in a depth first traversal of the graph. The processing of all elements finishes after all succeeding operators or sinks have finished their processing. We denote the ability of an operator to call its successors *direct interoperability (DI)*. Note that in case a query graph follows our approach and all operators use DI, no scheduler is required for processing. Hence, queues are optional; we can use them to decouple operators in the query graph. *Decoupling* stops DI at certain points in a query graph. For that reason, we have modeled queues as separate operators. It is worth mentioning that queues do not have an impact on the semantics, but are only introduced for performance reasons.

The question whether to place a queue between two operators or not leads to the idea of Virtual Operators, which are the topic of the next section.

3 Virtual Operators

Let us now introduce *virtual operators (VOs)*. We will outline how to implement them for pull-based and push-based processing. A comparison of both approaches reveals the convenience of a push-based approach.

3.1 Motivation

The basic idea of virtual operators is to save overhead by merging several operators into a single one. As an example, consider a chain of low-cost selections that are to be executed. If a queue is placed before each operator, the resulting `enqueue`, `dequeue`, and queue management operations may have higher cost than the subsequent operators. Thus, it would be better to unify the selections to one operation. The elements could be processed in a pipelined manner without intermediate queues. A chain of directly connected selections behaves as one virtual operator that computes the conjunction of these selections.

The concept of virtual operators can be generalized. A *virtual operator (VO)* is a subgraph that consists of at least two adjacent operators that do not store intermediate results with queues. Figure 1 shows a VO. The implementation of the virtual operator concept depends on the processing paradigm. We describe the concept for both: the pull-based approach (which underlies Aurora [1]) and our approach for push-based processing. However, let us emphasize that the concept is also suitable for other approaches.

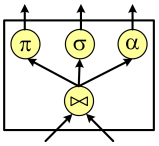

Figure 1. A virtual operator for a subgraph comprising of four operators.

3.2 Virtual Operators in Pull-based Processing

In order to build a VO in the context of pull-based processing, three steps are required. The first step is to select the operators that should build the VO. These operators must be adjacent and form a tree. The latter restriction can be tracked back to the underlying ONC operators, which do not support multiple sinks. For a given set of operators that are to build a VO, we replace in the second step all queues between them with special queues, called *proxies*. The `dequeue` method of a proxy reads the next element of its source until it either reads a data element or it reads a special element, which indicates that currently no element is available (recall Section 2.2). In the final step, we make sure that the scheduler only calls the `next` method for the root of the VO.

The general procedure is illustrated in Figure 2 for a chain of two selections. The left part shows the original chain. In the right part, the scheduler invokes only σ_2 which queries σ_1 via the proxy queue.

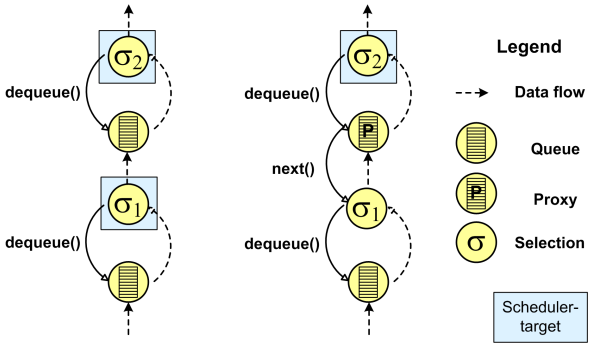

Figure 2. Realizing a VO with the pull-based paradigm. The original graph on the left is transformed into a VO by using a proxy.

3.3 Virtual Operators in Push-Based Processing

Our push-based approach offers the construction of VO in a straight forward manner. As operators without intermediate queues use DI, they automatically build a VO. For the previous example based on two selections, we can simply connect σ_1 and σ_2.

If the push-based approach relies on queues, the concept can be implemented with proxies analogously to the pull-based approach.

3.4 Discussion

As the previous considerations have clarified, we can build VOs with pull-based and push-based processing without changing the operator implementation. However, the push-based approach is more general than the pull-based approach as we can build VOs for arbitrary adjacent parts of a query graph. In contrast, the pull-based processing always needs a unique root to invoke the processing and is therefore limited to trees. This limitation also means that pull-based processing can not support subquery sharing within a VO. Consider the left part of Figure 5 as an example. The results of the projections are needed by both following operators. A call of the `next` method of one of them without temporarily storing the result for the other operator leads to incorrect results. However, temporarily storing elements within a VO is not permitted. These two facts lead us to choose the push-based approach.

4 Scheduling

Let us now present our novel hybrid multi-threaded scheduling approach called *hybrid multi-threaded scheduling (HMTS)*. Prior to the discussion of our architecture for

implementing HMTS, we examine GTS and OTS.
For ease of presentation, we drop the distinction between executing queues or operators which corresponds to the pull-based or push-based paradigm (Recall this from Section 2.2 and 2.3). We concentrate in the reminder of this work on the push-based approach. HMTS can be adapted similarly for pull-based processing.

4.1 Former Scheduling Approaches

Existing scheduling approaches fall into one of the following two categories: In *graph-threaded scheduling (GTS)*, one thread executes the complete query graph. In *operator-threaded scheduling (OTS)*, each operator runs in a separate thread [2, 7, 14]. Figure 3 shows both approaches applied to the same graph.

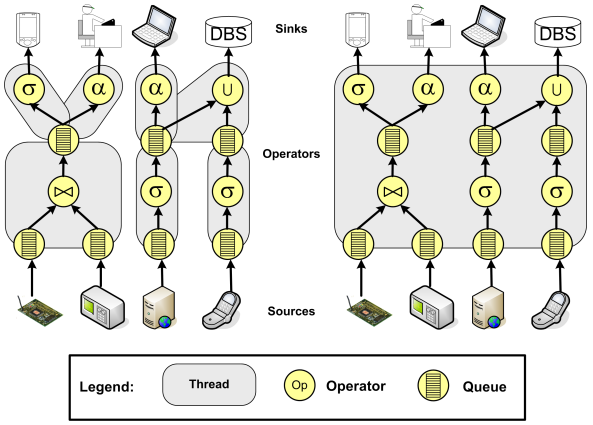

Figure 3. The same query graph realized with OTS on the left and GTS on the right.

4.1.1 GTS

A graph threaded scheduler utilizes a strategy to select the next operator to be executed. Then, the operator is given entire control; it runs for a certain time slice or as long as elements for processing are available. However, an expensive operator can exceed the given time slice as there is no guarantee that the processing of a single element is done quickly enough. This renders the resource planning in a DSMS difficult; it also adversely affects its scalability. If the query graph does not contain expensive operators, GTS performs well as there is only synchronization overhead in the input queues after the sources of an operator graph and no thread overhead occurs.
Aurora [7] schedules on the query level similar to GTS, but separately for each query. Therefore, expensive operators remain problematic on the query level.

4.1.2 OTS

In OTS, an operator thread obtains elements from its input queues, processes them, and puts the results in the output queues. In this context, the suspension of an operator's thread is an important mechanism; it is used if the operator's queues are empty or the processing takes too long. OTS takes advantage of multiprocessor machines because they can run multiple threads concurrently. However, the following two aspects limit the applicability of OTS. First, it provides no strategy for selecting the next thread. Second, the overhead of running each operator in a separate thread inhibits the scalability for query graphs with many low cost operators.

4.2 HMTS

4.2.1 Motivation

As discussed above, OTS is well-suited to avoid stalling in query graphs with expensive operators while GTS is preferable for query graphs with many low cost operators. We believe that in practice usually both cases simultaneously occur in a query graph. On account of this we propose a hybrid scheduling approach called HMTS. HMTS offers to dynamically adapt the number of threads and to assign them flexibly to partitions of the query graph. In order to keep the balance between concurrency and overhead, HMTS offers to schedule each partition with respect to a separate strategy. This feature is relevant to large query graphs and in presence of multiple QoS guidelines. HMTS also offers to dynamically switch between GTS and OTS.
The concept of VO is seamlessly integrated in HMTS with regard to an efficient resource management.

4.2.2 Architecture

We propose a three-level architecture for HMTS. Let us briefly describe these levels and the resulting benefit for processing.

Overview The first level consists of operators, queues, and VOs. On the second level, partitions of first-level operators are constructed by inserting queues if not already present between operators that belong to different partitions. The partitions can be chosen freely; we later discuss how to build them. This level provides an explicit scheduling. A thread executes a partition like a graph-threaded scheduler. It is possible to choose arbitrary strategies on the second level provided that they comply with the first level. For instance, FIFO can be used anytime while the Chain strategy [3] does not allow that operators belonging to a different segment of the lower envelope to use DI.

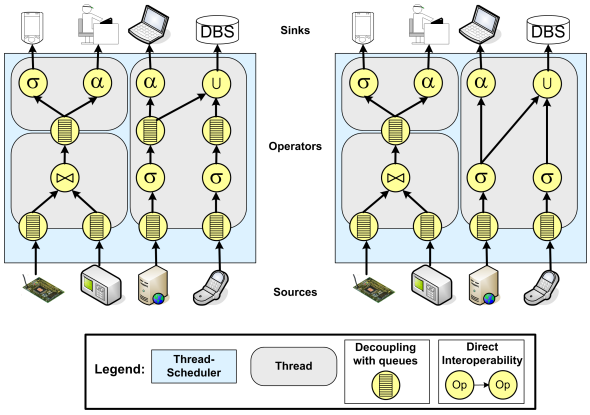

Figure 4. Query graph realized with pure HMTS on the left and HMTS in conjunction with VO on the right.

The third level runs multiple second-level units concurrently. Concurrency is managed by a specific high-priority thread termed *thread scheduler* (TS). This concept breaks up atomic execution as given in second-level units. Our default TS accomplishes a preemptive priority-based scheduling strategy. It determines the next thread to be executed so that starvation is prevented. The distribution of the available CPU resources relies on priorities that can be adapted during runtime.

Figure 4 shows the query already presented in Figure 3 with the difference that pure HMTS and HMTS in conjunction with the VO concept is used respectively. In the right query graph, some of the operators use DI and build a VO. Three threads are used to execute the operators in the graph.

Flexibility Let us now illustrate the flexibility of HMTS. It allows us to use only the levels which are needed in the context of a concrete application. For instance, it may suffice in simple cases to run a single VO without a scheduler. Moreover, we can also change the thread assignments during runtime to adapt to changing stream characteristics.

OTS and GTS are special cases of our architecture. Let us briefly describe how to simulate them with our architecture. Note that both do not need the third level although it could be used for improving OTS. For OTS, the first level consists only of physical operators with a separate thread and a queue for each of them on the second-level. (An alternative view is that each physical operator forms a first level unit.) OTS does not necessarily require a TS as threads are scheduled by the operating system and every thread has only one operator to execute. Consequently, a graph based on OTS does not need a TS. For GTS, the first-level is the same as described for OTS. On the second-level each operator has a separate queue but all operators run in one thread. Thus, a TS is not needed.
We can seamlessly switch between these approaches during runtime. For instance, we want to change from OTS to GTS. As the first level is equal in both approaches, it is not necessary to change the graph structure. The GTS scheduler is instantiated with a strategy and the entirety of operators that are assigned to OTS. Then, all OTS threads can be stopped instantly and the GTS scheduling starts.
Changes on the third level can be done in a similar way. Changes on the first level are discussed in the next section, where we elaborate the method for building VOs.

5 Queue Placement

The crucial question in the construction of VOs is the placement of the queues. From a formal point of view, this is a graph partitioning problem, where each partition corresponds to a VO. The computation of an optimal partitioning for an arbitrary graph is NP-complete. For that reason, we focus on the development of a suitable heuristic. More precisely, our objective is to partition the query graph so that stalled operators are avoided.

5.1 Stall-avoiding Partitioning

5.1.1 Motivation

The aim of the strategy is to avoid stalling in VOs. The idea is to add operators to a VO as long as it can keep pace with the input rates. To give a feeling for this idea, let us consider the VO on the left side of Figure 5. We assume that the three unary stateless operators have low processing costs while the aggregation is expensive. In Section 5 we argued that we can save resources if we can merge low cost operators into a VO. It is advantageous to leave out the queues. However, it may be disadvantageous if the aggregation is also directly connected to the projection. Let us assume we build a VO from all operators. If an element arrives at the input of the VO, the processing of the aggregation on the former element may not be finished yet. Consequently, the new element can not be processed. The VO stalls and delays the processing of new elements. Decoupling the aggregation by inserting a queue between the aggregation and the projection as shown in the right part of Figure 5 would avoid this problem. In this constellation, the aggregation cannot decelerate the output rate of the unary operator chain anymore.

5.1.2 Problem Formalization

To formalize the problem, we need some notation. Let $G = (V, E)$ be a query graph without queues. For each operator $v \in V$, we define

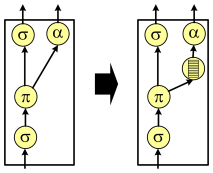

Figure 5. Removing an expensive aggregation from the VO to avoid stalling the other operators.

- $c(v)$ the average time of v to process an element,
- $d(v)$ the average interarrival time of elements in v's inputs. $d(v)$ is the reciprocal of the input rate of v.

Let us consider a partitioning P of G, which consists of disjoint subgraphs P_i. As a partition shall correspond to a VO, we additionally require that all nodes in a partition are connected. We define for a partition P_i

- $c(P_i) = \sum_{v \in P_i} c(v)$
- $d(P_i) = \frac{1}{\sum_{v \in P_i} \frac{1}{d(v)}}$
- $cap(P_i) = d(P_i) - c(P_i)$ the capacity of P_i

The goal is to minimize the number of partitions under the constraint that the capacity of each VO is not negative.

5.1.3 Static Queue Placement

Parameter We assume that the required values $c(v)$ and $d(v)$, $v \in V$, are meta data provided by the DSMS during runtime. An alternative that saves overhead is to estimate them with respect to a suitable model, e.g. [5].

Algorithm Algorithm 1 is a heuristic to avoid stalling within a query graph. It receives all sources of a query graph G (without queues) and traverses G bottom up. The given sources are stored in a `todo` list for further processing. The `done` list is used to ensure that every node is traversed only once (line 6-8). In line 9-14, the algorithm stores the sources that should be directly connected to a node in `partition`. `sortedSrcs` contains the sources, which are ordered descending by capacity. Thus, a partition is build with a first fit decreasing strategy because a source is selected, when the combined capacity of `source` and the actual processed `partition` is greater than or equal to zero. Finally, a queue is inserted between all nodes that are not in `partition` (line 16-18). The motivation for using the first fit decreasing strategy is that it yields for each partition a solution with an approximation guarantee of $1 + ln|\text{partition}|$.

Algorithm 1: static queue placement

Input : a query graph G = (V,E) without queues, A list inputs containing the data sources of G
Output : a query graph G' with queues

```
1  list todo;
2  list done;
3  todo.addAll(inputs);
4  while ! todo.isEmpty() do
5      node = todo.removeFirst();
6      if ! done.contains(node) then
7          done.add(node);
8          todo.add(node.getAllSinkNodes());
9      list partition;
10     partition.add(node);
11     list sortedSrcs = sortDescByCap(todo.getAllSourceNodes());
12     foreach source s from sortedSrcs do
13         if addCap(partition, s) ≥ 0 then
14             partition.add(s);
15     node.setCap(partition.getCap());
16     foreach source s from node do
17         if ! partition.contains(s) then
18             placeQueueBetween(s, node);
```

Inserting and removing queues can be done during runtime by interrupting the processing of the graph shortly. A queue can be immediately inserted without any further effort. To remove a queue all remaining elements in the queue must be entirely processed before. However, an efficient algorithm for placing queues during runtime remains to be addressed in future work.

6 Experimental Evaluation

6.1 Overview

This section presents the core results of an experimental study, showing that DI, OTS and GTS do not perform optimally. In the case the query graph is partitioned appropriately HMTS out performs these approaches. With the experiments, we addressed the following questions:
Are there scenarios where decoupling is necessary? When is it advantageous to use DI instead of OTS or GTS? What are the performance advantages of HMTS over GTS and how does our queue placement algorithm perform?
For the sake of clarity, we kept the number of operators in the query graph small. It is worth mentioning that more complex query graphs lead to larger differences in the performance of the methods.

6.2 Experimental Setup

The upper techniques were implemented in PIPES, our java-based stream processing architecture. All experiments

were executed on a Pentium DualCore 3 GHz with 3 GB main memory. This machine runs Windows XP and Sun's Standard JVM 1.5. The presented experiments rely on synthetic data streams, which facilitates the adjustment of the relevant parameter. In order to simulate bursty traffic, the inter arrival rate between two successive elements followed a Poisson distribution, analogous to the experimental setup in [3].

6.3 The Necessity of Decoupling

In our first experiment, we show that direct interoperability may induce a loss in performance. With respect to a join setting, we demonstrate the necessity of decoupling. We examined a binary symmetric hash join (SHJ) and a symmetric nested loops join (SNJ) over two data sources that delivered 180,000 elements at a rate of 1000 elements per second. Because the first source delivered elements uniformly distributed in $[0, 10^5]$ and the second in the range of $[0, 10^4]$, the join selectivity was approximately 0.1. The join algorithms used a one minute sliding window. Each join operator directly ran in the thread of its autonomous data sources.
The results in Figure 6 indicate that both joins can not keep pace with the input rate. As a consequence, their input rates decrease. This occurs for the SHJ after 58 seconds and for the SNJ after 17 seconds. We conclude that without queues placed before each join, we would inevitably lose data.

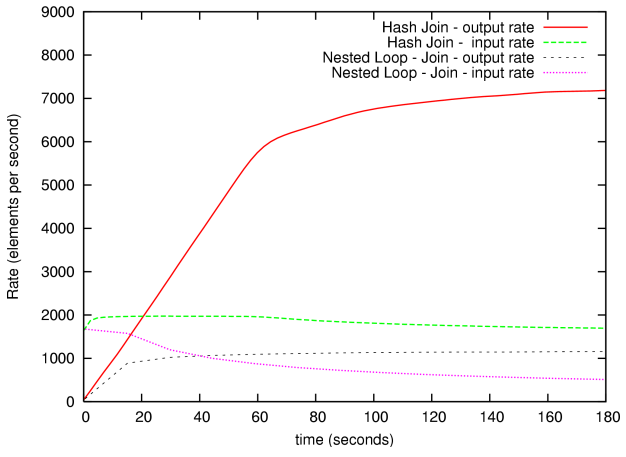

Figure 6. The necessity of decoupling.

6.4 Comparing GTS, OTS and DI

Our second experiment emphasizes the different runtime behavior of GTS, OTS and DI. We posed a query consisting of 5 selections with selectivities 0.998, 0996, ..., 0.990 over a autonomous source that delivered m elements with a rate of 500,000 elements per second; m varied from 100,000 to 1,000,000. In the DI setting, there was one queue after the source one thread was used for the selections. In the GTS and OTS settings, all operators were decoupled. GTS used one thread for the whole query while in the OTS setting each queue used its own thread. The GTS experiment is based on the Chain and FIFO strategies (FIFO is not presented in the chart as its performance was nearly the same as Chain).
Figure 7 shows the time that was required for processing the query. OTS and GTS heavily rely on communication via queues at the expense of a higher computational effort. OTS is significantly faster than GTS due to its efficient use of the multicore environment. However, DI is even without parallelism still 40 % faster than OTS.

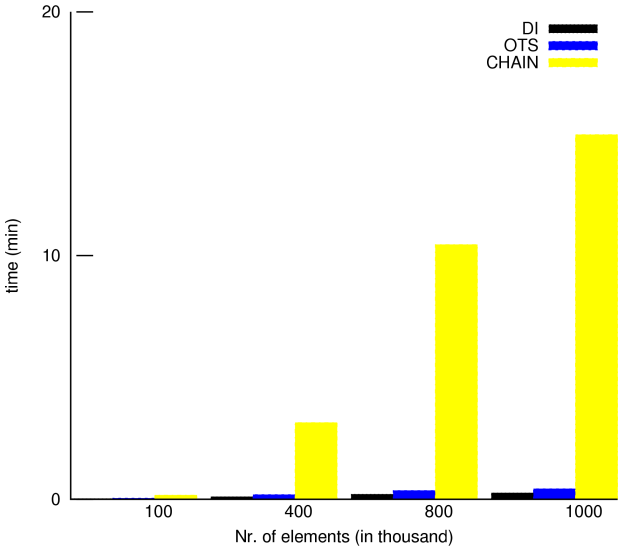

Figure 7. Runtime for a simple query using GTS, OTS and DI.

6.5 Scalability of OTS

Let us take a closer look at the different between DI and OTS. While GTS is clearly inferior, it raises the question whether it is worth to implement DI for approximately 40 percent faster processing. We show in this experiment that DI's performance advantage over OTS is not constant, but increases with number of queries.
We posed the same query with an variation from 1 to 200. The number of elements was set to 100,000 in all experiments. Figure 8 depicts the results of this experiment. We observe a significant difference between OTS and DI. The more queries are running, the better is DI. Consequently, DI offers a scalable processing in query graphs with many operators. OTS works well as long as the number of operators and their associated threads is not too high.

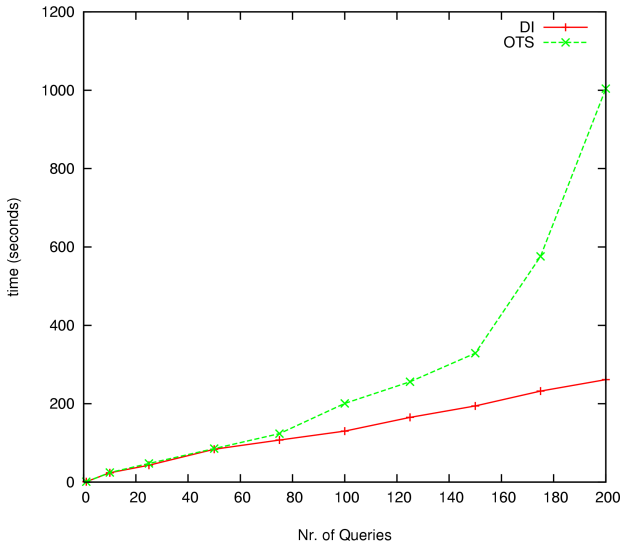

Figure 8. Varying the number of queries in the third experiment.

6.6 HMTS in comparison to GTS

The next experiment demonstrates the advantage of multi-threaded processing in HMTS in comparison to GTS. We posed a query with three successive operators. The first was a projection with processing costs of 2.7 micro seconds followed by a selection with selectivity of $9 \cdot 10^{-4}$ and processing costs of 530 nano seconds. The last was a selection with a selectivity of 0.3 and processing costs of approximately 2 seconds. Thus, the latter selection simulated a complex predicate evaluation and implied a significant stall of the other operators. The associated data source transferred $7 \cdot 10^5$ elements uniformly distributed in $[1, ..., 1 \cdot 10^7]$. The source emitted the elements 1 to 10,000 and 30,001 to 50,000 with a high rate of approximately 500,000 elements per second, which took significantly less than a second. The remaining elements (from 10,001 to 30,000 and 50,001 to 70,000) were emitted with a rate of 250 elements per second which took 80 seconds.

In our first setting (GTS), we inserted queues in-between two successive nodes and processed the queues via a scheduler, which ran in a separate thread as proposed in [3, 7, 12] As we wanted to compare the production of early results and the memory usage of the query, we chose the two GTS scheduling strategies FIFO and Chain. In the second setting (HMTS), we decoupled the data flow twice: between the source and the first filter as well as between the filters. We used two threads where one of them schedules the graph.

The Figures 9 and 10 show the results of the experiments. As the experiment starts with a burst of 10,000 elements, all curves start with a queue memory usage of 10,000 elements. FIFO and Chain behave as expected. In the Chain setting we observe some initial delay for profiling and computing the lower envelope at the beginning indicated by the increased number of results. This computation splits the graph in two groups, the first consisting of the projection and the following selection and the second consisting of the remaining selection. Chain schedules always the first group which reduces the memory usage rapidly. After 50 seconds, the memory usage vacillates because whenever the first group's queues are empty the second is scheduled. As the second selection takes 2 seconds, the projection's queue size and the number of results grow during this period. The memory usage increases from 80 s to 160 s rapidly due to the second burst, but decreases afterward. Chain takes about 260 s to complete with very low memory usage in the period from 160 to 260 seconds.

The FIFO memory usage curve in Figure 9 decreases slower compared to Chain, because FIFO always executes both groups successively; the results are produced continuously and earlier (see Figure 10). The memory size drops faster from 160 to 260 as the input rate is zero. Like Chain, FIFO finishes after 260 seconds.

HMTS performs better than both GTS strategies. The memory usage curve is slightly below the corresponding curve from Chain. Moreover, the results are produced significantly earlier and the whole processing is finished within 160 seconds. This can be explained with the use of a multicore machine. Thus, both selections can be executed concurrently which would lead to a maximum running time of 162 seconds in the worst case: the last element would be emitted at the source after 160 seconds and would require approximately two seconds for processing the three operators. The experiment finishes shortly after 160 seconds because the last 2,000 elements are filtered out by the first selection.

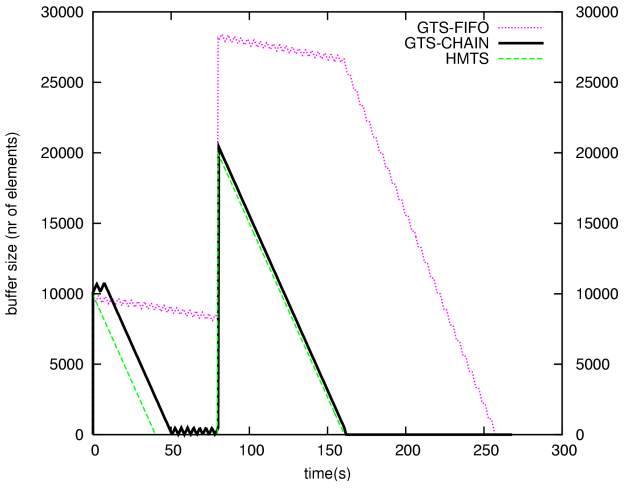

Figure 9. HMTS vs GTS (Memory size)

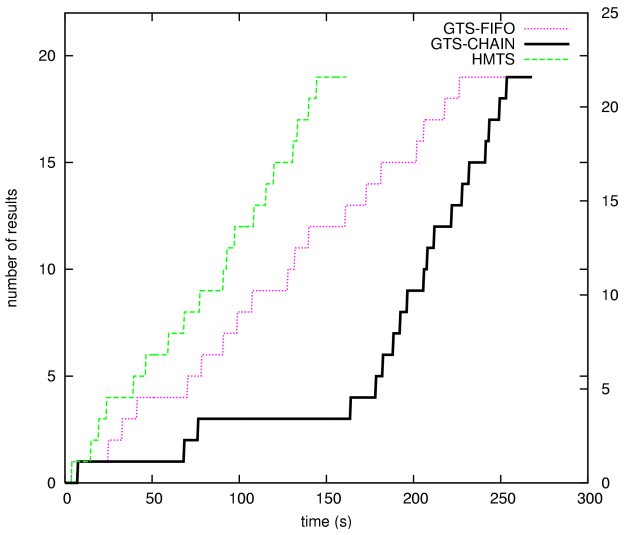

Figure 10. HMTS vs GTS (results)

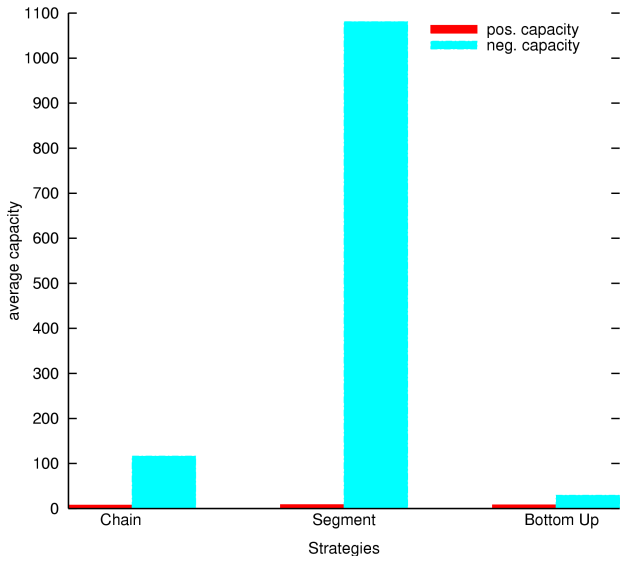

Figure 11. Differences in negative and positive capacities of three algorithms for constructing VOs.

6.7 Construction of VOs

In the last experiment, we compared the capacity usage rate of VOs, based on different algorithms for this construction. Recall that negative capacity means that a VO stalls incoming elements, while a positive capacity means that the VO is not fully utilized. We used three algorithms to build VOs: the algorithm we described in Section 5.1.3, the algorithm for the simplified segment strategy [10], and an algorithm based on the chain strategy [3]. The latter removes queues if they belong to the same chain.
We tested the algorithms by running them on random DAGs, varying the number of nodes from 10 to 1000.
Figure 11 shows for each of the strategies the average capacity of the VO over all graphs. The negative and positive capacities are shown separately. All three strategies produce only very few VOs. They are not fully utilized but they differ significantly in their average negative capacity. Our VO construction algorithm performs better than the other algorithms. However, they are not intended to build VOs with capacity close to zero.

7 Related Work

We divide this section into two parts. We start with previous work that is applicable to building virtual operators. Then we discuss scheduling architectures of related DSMS.

[15] introduces a multi way join for DSMS. Because the join does not materialize intermediate results, a join with n inputs can be seen as a VO with n inputs and one output. In Aurora [6], a query consists of boxes where each box can be a physical operator or a chain of unary operators. Thus, the boxes in Aurora are special VOs with one input and one output. Boxes can be connected to superboxes which can be viewed as a special case of the first level of our architecture for the following two reasons. Firstly, there is always a queue between two operators. Secondly, the super boxes are query trees and thus, they are limited to n inputs and one output. In [6], two approaches to build superboxes are sketched. All these approaches differ from our work as they do not discuss how to build generic VO.
[10] proposes a segment construction algorithm for operator paths. A path is split in its segments, where each segment has no queues between its operators. Thus, a segment forms a VO. Although this can lead to VO with m input and n output this work focuses on scheduling strategies to affect resource usage in a DSMS.

We compare our scheduling architecture with STREAM [14], TelegraphCQ [8] and Aurora [1]. *STREAM* is a centralized data stream management system that relies on the relational data model. A queue connects two successive operators and guarantees uni-directional communication. The architecture described in [14, 3] bases on GTS. The *TelegraphCQ* [8] system combines prior work in *Fjords*, *Eddies* and *PSoup*. Thereby, the Fjords API [12] defines a set of operators, which communicate via either ”push” (asynchronous) or ”pull” (synchronous) modalities. Fjords allows OTS as well as GTS. Eddies [2, 13] are specific operators that decide how to route elements adaptively among physical operators. Each Eddy as well as each physical operator runs in a separate thread.
None of these system approaches provides a flexible multi-

threaded scheduling architecture that adapts the number of threads and queues dynamically. As they all rely on GTS or OTS, they are limited in their application.
Aurora is a system for monitoring data streams which attempts to satisfy real-time requirements. While in [7] OTS is proposed, later work [6] uses a two level scheduler. The first level is responsible for selecting one query for execution and the second level decides in which order the operators in the selected query are executed. This information is written into a runnable queue, which is used by multiple worker threads for concurrent execution of multiple queries. Thus, this approach can be seen as GTS for every single query. In contrast to our approach, Aurora does neither provide the possibility to unify multiple queries to a single one for resource sharing reasons nor the potential to split a single query into multiple execution parts.

8 Conclusions

In this work, we addressed the problem of operator scheduling in DSMS. To overcome the drawback of two previous proposed approaches, we presented a hybrid approach termed HTMS that can be viewed as a generalization of both. HTMS is highly flexible due to a three-level architecture. The core concept of the architecture is the merge adjacent of operators into a single virtual operator. Each of these virtual operators runs in its own thread. The communication between virtual operators is established through queues. We presented a suitable algorithm for determining virtual operators. Our experimental study revealed that HMTS performs considerably faster than its competitors. It proved to keep the balance between concurrency and queuing overhead. In our future work, we will examine promising extensions of HMTS. This involves designing and improving algorithms for each level of our architecture.

Acknowledgments

This work has been supported by the German Research Foundation (DFG) under grant no. SE 553/4-3.

References

[1] D. J. Abadi and D. Carney et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of ACM SIGMOD*, pages 261–272, 2000.

[3] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. of ACM SIGMOD*, pages 253–264, 2003.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. of PODS*, pages 1–16, 2002.

[5] M. Cammert, J. Krämer, B. Seeger, and S. Vaupel. An Approach to Adaptive Memory Management in Data Stream Systems. In *Proc. of ICDE*, pages 137–139, 2006.

[6] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[7] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proc. of VLDB*, pages 215–226, 2002.

[8] S. Chandrasekaran, O. Cooper, and A. Deshpande et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of CIDR*, 2003.

[9] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[10] Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *BNCOD*, pages 16–30, 2004.

[11] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of ACM SIGMOD*, pages 925–926, 2004.

[12] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proc. of ICDE*, 2002.

[13] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *Proc. of ACM SIGMOD*. ACM Press, 2002.

[14] R. Motwani, J. Widom, and A. Arasu et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proc. of CIDR*, 2003.

[15] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Join Queries over Streaming Information Sources. In *Proc. of VLDB*, pages 285–296, 2003.