The Design of Stream Database Engine in Concurrent Environment

Marcin Gorawski and Aleksander Chrószcz

Silesian University of Technology,
Institute of Computer Science,
Akademicka 16, 44-100 Gliwice Poland
Marcin.Gorawski@polsl.pl, Aleksander.Chroszcz@polsl.pl

Abstract. Quality of Service (QoS) in stream databases is strongly connected with the chosen scheduler algorithm and the architecture of Data Stream Management System (DSMS). In order to achieve better efficiency, those systems work in concurrent and distributed environments. Main difficulties in such systems are caused by a high number of messages exchanged between stream database modules. In consequence, it is hard to create a scalable stream database because the communication cost substantially reduces the benefits of the computation in concurrent environment. Therefore, in the paper, we analyze and evaluate architectures of DSMS. Especially, we focus on cooperation between stream buffers, physical operators and schedulers so as to reduce the role of synchronization in DSMS overhead. Finally, we have created a stream database architecture which substantially improves query result latencies and, besides, this architecture is easy to develop.

1 Introduction

The optimization of response times and memory usage is a key target in data stream management system development. Data stream applications continuously update results each time a new tuple arrives, whereas operators of traditional Data Base Management Systems (DBMS) are oriented to process tables of records. In consequence, the synchronization of data access is more frequent in DSMS than DBMS. Moreover, in contrast to DBMS, DSMS processes queries continuously so it needs specialized architecture to process them.

The research in this area involves schedulers [1, 2, 3], architectures of stream operators [4] and DSMS (e.g., Aurora&Borealis [5], STREAM [6], PIPES [4], Eddy [7], Nile [8], CEDR[9]). Unlike them, we focus on architectures of stream processors because the cooperation between stream operators, schedulers and stream buffers determines the final performance of such systems.

The low-level communication/concurrency aspects of stream processing haven't been adequately addressed by the stream database community, although this issue is critical to achieve low-latency results and scalable systems. In the paper, we compare the combinations of existing techniques used in concurrent applications and test them in our stream database, StreamAPAS. Having analyzed the working conditions of those experimental combinations, we have created and evaluated the architecture of the stream database engine which introduces a low communication costs and is easy to develop.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part II, LNCS 5871, pp. 1033–1049, 2009.

The rest of this paper is organized as follows: Section 2 introduces the background of DSMS; Section 3 compares architectures of workers; Section 4 thoroughly explains a tuple transport layer; Section 5 describes potential optimizations of stream buffers, next Section 6 briefly shows the architecture of physical operators; then Section 7 discusses the effects of our approach; and finally Section 8 concludes the paper.

2 Background of DSMS

A logical query plan is described by a direct acyclic graph whose nodes represent logical stream operators and edges represent streams. We distinguish source operators, which do not have input streams. Sink operators, which have only input streams and middle operators which provide both input and output streams. The operators on the path from a source operator to a sink operator with the source operator excluded create a *processing path*.

The physical query plan defines algorithms used to process the corresponding logical query plan. For instance the concrete realization of a logical join operator is a physical operator. The physical plan is also described by a direct acyclic graph whose nodes represent physical operators and edges represent algorithms of stream buffers. The query engine defines workers which compute physical query plans. Each of those workers has its own associated thread. Summing up, stream processing is a continuous process which distributes physical operators to workers. The policy of this distribution is defined by a scheduler algorithm.

An operator is not available for scheduling, as no data exists at the input at a given moment. On the contrary, the available operator for scheduling has waiting input data to be processed. In the remaining part of the paper we use the terms *unavailable operator* for scheduling and *unavailable operator* interchangeably. Similarly, the terms available operator for scheduling and available operator are also used interchangeably.

The Round-Robin algorithm is a basic example of schedulers. It activates physical operators in the sequence of their arrivals at the FIFO queue of ready-to-execute operators. An operator is executed by a worker until the operator has processed all the input tuples or the fixed limit of input tuples has been reached. The simplicity of this scheduler is achieved at the cost of no control over result latencies and memory usage. On the other hand, this algorithm guarantees starvation avoidance because each physical operator will by finally popped from queue and executed.

The FIFO algorithm [3] processes tuples in order of their arrivals at the stream application. Each input tuple is processed through its processing path until the next tuple is popped from input streams. In comparison to Round-Robin, this algorithm minimizes latencies of result tuples. On the other hand, it ignores the optimization of memory usage.

The Chain scheduler [1] is designed to reduce the size of stream buffers. This algorithm uses the average processing time and selectivity of operators in order to adapt to the changing system load. The weakness of the Chain scheduler is that it does not guarantee starvation avoidance. The Chain-flush algorithm [2, 1] is the modification which defines two separate algorithms: one optimizes the buffer size and the other optimizes result latency. This scheduler switches to the optimization of result latency when the

stay of tuples in the system is longer than a predefined limit. As a result, this scheduler guarantees starvation avoidance.

The above schedulers assume that there is a global order in which operators are executed and therefore those algorithms are not directly designed for concurrent or distributed environments. It is an open challenge to adapt them efficiently to the concurrent environment.

3 The Architectures of Stream Engines

For a given physical query plan, the physical operators are divided into two groups. One group contains all the physical operators which belong to processing paths. Those operators are executed when they have an available input tuple to process and they are scheduled. The other group consist of source operators.

3.1 One-Thread Engine

This architecture implements all the engine functionalities in a one thread environment. The external tuples/events are received by the source operators which work in separate threads, next they send those tuples to the engine. Then the engine dequeues input streams, evaluates calculations and sends results to clients. The main drawback of this architecture is that it does not allow us to gain any benefits from multi-core processors. On the other hand it is easy to develop.

3.2 Basic Worker Pool

In this architecture, the engine manages a group of workers. The basic query engine, which is shown in Fig. 1, consists of a worker pool and a scheduler queue. At the beginning of a query start, the physical operators except source operators are moved to the collection of unavailable operators, and then source operators are started. When an operator has data to process, it is moved to the scheduler queue. The first operator in the scheduler queue triggers the procedure which distributes the physical operators to available workers. Workers which currently execute physical operators are moved to the collection of the working workers. When a worker finishes execution, it returns to the collection of the available workers. The physical operator which was executed has three options. When it has no tuples to process, it is moved to the collection of unavailable operators. If it still has tuples to be processed, it is moved to the scheduler query. Otherwise, the physical operator finishes working and it leaves the system.

This model requires the synchronization of three areas: 1) the movement of workers between the collections, 2) insertions and removals from the scheduler queue and 3) stream buffers. Figure 2 shows a snapshot of a working stream engine. The stream operator O1 (O3) is processed by worker W1 (W2). Because O1 is linked to O3 and those operators belong to different threads, the connecting stream buffer must be synchronized.

Let us notice that the distribution of physical operators to workers is random because a physical operator is assigned to any available worker when the operator is popped from the scheduler queue. In consequence, all the stream buffers have to be synchronized

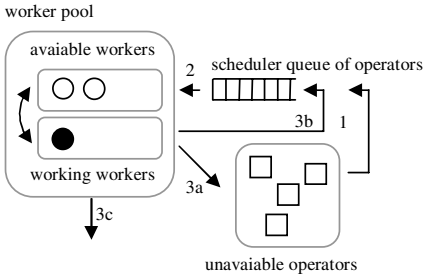

Fig. 1. The basic implementation of a worker pool and a scheduler queue

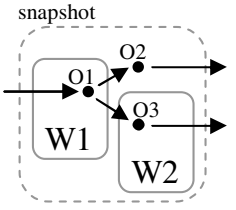

Fig. 2. Random distribution of physical operators to workers

because there can appear a configuration showed in Fig. 2. On the other hand, this architecture guarantees that the workload is distributed equally among the workers.

Let us notice that the efficiency of this worker pool is high when the time to process an operator is remarkably higher than the time to service stream buffers, the scheduler queue and collections of workers. Available operators usually have a few tuples to process per one scheduler iteration. In consequence, the synchronization of stream buffers and the synchronization of the scheduler queue play a significant role in the final performance of the stream processor.

Another weak point of this stream processor architecture becomes apparent when the Chain scheduler is chosen. Then, it is necessary to introduce a *supervisor* module which collects statistics of physical operators, calculates the priorities of those operators and updates scheduler queue parameters. As a result of that, the resources of physical operators are accessed by both workers and the *supervisor* so the resources of those operators have to be additionally synchronized, which makes the system slower.

3.3 Worker Architecture Oriented toward Tuple Transmission

The number of tuples to be processed is much greater than the number of scheduler iterations and other events processed by the engine. From this point of view, the cooperation of operators, scheduler and stream buffers should be biased toward tuple transmission.

Let us assume that two consecutive operators in a processing path are always processed by worker W1. This is illustrated in Fig. 3. In consequence, the stream buffer connecting O1 with O3 does not need to be synchronized because it works in a single-thread context. If the stream processor allows us to determine which operators are executed inside the same thread context, we can use faster, non-synchronized stream buffers

Figure 4 shows a proposed worker architecture. Workers have two types of input queues:

- The instruction queue is used to access the local resources of a worker, for instance it is used to update priorities of physical operators or insert/remove them from workers.
- The input stream buffers are used to communicate with physical operators belonging to other workers.

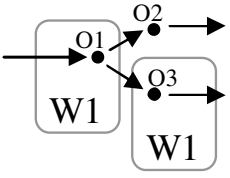

Fig. 3. Determined distribution of physical operators to workers

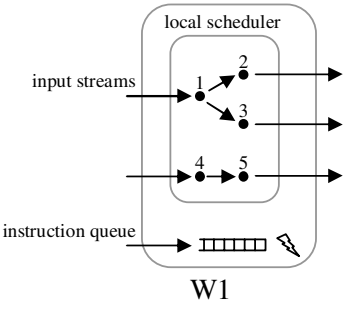

Fig. 4. A single worker

In contrast to the architecture from Sect. 3.2, each worker owns a local scheduler queue. The workers execute tasks stored in the instruction queue and the scheduler queue according to the priorities: at first the instruction queue then the scheduler queue.

Let us notice that this configuration associates physical operators with workers as it was shown in Fig. 3. This allows us to apply non-synchronized local stream buffers. Both instructions and physical operators are executed by the same thread. In consequence, physical operators belonging to worker W are accessed by the same worker thread. This feature simplifies the management of W resources because no additional synchronization is necessary. As a result of this, the priority updates of physical operators and the deployment of the physical query graph among workers are faster.

In summary, the proposed architecture is oriented towards a fast tuple transmission. The resources of a worker are accessed only by one thread, so their implementation is easier to develop. The control communication is established by instruction queue. As a result, this model can be easily adapted to the distributed environment. The main drawback of this approach is the workload, which needs to be monitored and balanced.

4 Tuple Transport Layer

A tuple supply layer transmits tuples between operators. When there is no tuple to processing, the tuple transport layer generates the *unavailable* signal. When there appears the first tuple to process, the *available* signal is generated. The *available* signal is generated by a thread which inserts tuples into the streams of operator O and the *unavailable* signal is generated by a thread which consumes tuples. In order to interpret those signals correctly, we have to guarantee that the *unavailable* signal follows the *available* signal in a concurrent application. The tuple supply layer is created for each operator and has the following functionalities:

1. It registers the event observers of the signals *available* and *unavailable*,
2. It implements a method which inserts tuples into input streams,
3. It implements the *getNextTuple* method which returns the next tuple to process and the index of the input stream of this tuple.

Let us assume that a stream operator has n inputs: x_1, x_2,...,x_n; each x_i is associated with register r_i and supplied with input stream s_i. Register r_i is updated with the $timestamp$ of the tuple which has been recently passed on to x_i. Tuples in streams are in chronological order [5] (in lexicographical order [10] in the case of temporal tuples). Algorithm 1 which was proposed in [11] describes the order in which tuples are popped from input streams.

Algorithm 1. Selection of the next tuple to process

1. If none of the input streams is empty, then the next tuple to process is the one with the lowest $timestamp$ value,
2. If at least one input stream is empty, then the next tuple to process is the one with $timestamp = min_{i\in[1,n]} r_i$.

Let us imagine that tuples with identical timestamps arrive at both inputs streams of a *join* operator. Thanks to registers *r* and point 2) of Alg. 1, all of those input tuples can be processed. If the algorithm consisted only of point 1), then tuple processing would be stopped after the first stream becomes empty.

Now we introduce two implementations of the tuple transport layer. The first one is based directly on the tuple transport specification. The second one additionally uses the instruction queue so as to define better separation between objects belonging to a worker and threads collaborating with the worker.

4.1 Straightforward Implementation of the Tuple Transport Layer

When the tuple supply layer works in multi-thread environment, it is necessary to synchronize signals in order to guarantee that the *unavailability* signal follows the *availability* signal. Let us assume that $minSize$ is the minimum length of all the input streams connected to operator O.

We synchronize the tuple supply layer in the following moments. Each insertion into an empty stream buffer is synchronized with monitor m because, in those moments, the *available* signal can be generated. The *getNextTuple* method call is synchronized with monitor m when $minSize = 0$ or $minSize = 1$ because, in those moments, *unavailable* signal can be generated. When $minSize > 1$, neither of the signals is generated and therefore no additional synchronization is necessary. The $minSize$ also determines which part of Alg. 1 is executed. When $minSize = 0$ then point 2) of the algorithm occurs. If $minSize > 0$ then point 1) of the algorithm occurs.

The above rules show calculating the exact value of $minSize$ is not necessary because we detect only three situations: $minSize = 0$, $minSize = 1$ and $minSize > 1$. Therefore, we can calculate only $minSize \in [0, 2]$ and treat value 2 as a situation in which the input stream buffers are of a size greater than 1.

Algorithm 2 describes how we apply those rules to implement the *getNextTuple* function so as to reduce synchronization. In this algorithm, $minSize$ is controlled with the use of Compare-And-Swap instructions (CAS) [12] because it is shared by multiple threads.

Algorithm 2. Synchronization of *getNextTuple* function

```
if(minSize.get() > 1) {
  read a next tuple to process
                point 2) of Alg. 1
  if(remaining stream buffer size = 1)
    minSize = 1
} else {
  synchronized(m) {
    switch(minSize.get()) {
      case 0:
          read next tuple to process
                point 1) of Alg. 1
        update of minSize
        if(no tuple available to process
           in next iteration step)
          send unavailable signal
        break;
      case 1:
        read next tuple to process
                point 2) of Alg. 1
        update minSize
        if(no tuple available to process
           in next iteration step)
          send unavailable signal
        break;
      default:
        read next tuple to process
                point 2) of Alg. 1
        if(stream buffer size = 1)
          minSize = 1
        break;
    }
  }
}
```

4.2 Customized Implementation of the Tuple Transport Layer

Let us notice that the tuple transport layer introduced in Sect. 4.1 is executed by threads which feed input streams and by the worker thread. *Available* signals are reported by one of the threads which feed input streams, whereas *unavailable* signals are reported by a worker thread. In consequence, an additional synchronizing object is necessary when $minSize \leq 1$. To make things worse, when this tuple supply layer directly controls the local scheduler queue, then it also has to be synchronized because it is accessed by multiple threads. Summing up, the drawback of this architecture is that objects controlled by a worker are not completely separated from threads which communicate with the worker.

The above observation inspired us to implement a synchronization strategy in which the tuple transport layer is controlled only by its worker thread. This assumption is a key target because it guarantees that communication among physical operators, the local scheduler queue and the tuple transport layer does not need synchronization.

The main method of this tuple transport implementation is *getNextTuple* which returns a tuple and the *slot* number of the input stream from which the tuple comes. The *streamAvailable(slot)* method is called when an input stream becomes occupied. A *setAvailabilityObserver* method implements the observer programming pattern which controls insertions and removes of tuples from the local scheduler queue. The *push* method returns *true* if the buffer is empty before a tuple is inserted. This implementation of the tuple supply layer distinguishes two cases. One refers to the configuration in which O1 and O3 operators in Fig. 3 belong to the same worker. In such a situation, the *streamAvailable(slot)* method is called directly from the method which inserts a new tuple to an input stream. The other case refers to the configuration in which operators O1 and O3 in Fig. 2 belong to different workers. In such a situation, when operator O1 inserts the first tuple into the stream, it also inserts a command to the instruction queue. Next this command calls the *streamAvailable* method.

Summing up, this worker architecture contains the minimal number of synchronized objects which are the instruction queue and the stream buffers that connect operators belonging to different workers. In order to achieve software easier to maintain, we have also used the active object programming pattern [13], which simplifies the control of the instruction queue.

5 Stream Buffers

The final efficiency of the worker architecture depends on the implementation of stream buffers. We cannot compare the efficiency of stream buffer algorithms in isolation because the synchronization mechanisms which they use may not cooperate well with other synchronization mechanisms deployed in workers.

A data stream is a simple FIFO queue. One operator inserts tuples into a stream and another one reads from it. From the viewpoint of the concurrent algorithm, this queue works in the configuration: one producer - one consumer. We distinguish two types of streams. Local streams, which connect operators belonging to the same worker and synchronized streams, which work in a multi thread environment.

The basic implementation of synchronized streams uses lock/unlock operations. In the paper, we focus on the more sophisticated algorithms, which have block-free and wait-free properties.

5.1 Block-Free and Wait-Free Stream Buffers

The authors of [12] describe a fast, wait-free and non-blocking FIFO queue algorithm working in the configuration: many producers - many readers. This is a fast queue based on atomic memory modifications and CAS instructions. We decided to check if it is possible to improve this algorithm when the buffer is intended for the configuration: one producer - one consumer.

The structure of a FIFO queue [12] is illustrated in Fig. 5. The structure is initialized with the first node in the list, which is a dummy node. Let us now follow the insert operation. When a new element is inserted into the queue, we update $new.next = null$. Then, we read the reference to the last element t with the use of $tail$, next we set $t.next = new$ and update $tail = new$. When we read an element from the queue, at first we read the reference to element t with the use of $head$. Then, we save $t.next$ value to tmp variable. Next, we update $head$ with tmp and return tmp. In sum, the insert operation changes only $tail$ and $t.next$. Thanks to a dummy node, element t is always inside the queue and therefore the read operation changes only $head$. In consequence, if the queue works in the configuration: one producer - one consumer, it is not necessary to synchronize the insert and read operations.

In order to implement this algorithm correctly in Java, we have to take into consideration the specification of threads and the memory model in Java Virtual Machine (JVM) [14, 15]. The *synchronized* instruction not only locks/unlocks a monitor object but also guarantees that the variable values are consistent and up-to-date in a given thread context. Java implements a cache mechanism which optimizes access to the system memory. In consequence, the values which are written and read can be different when thread A writes to variable x, and then thread B reads from it without any synchronization.

Only variables of the *final* and *volatile* types can be safely read and written without additional synchronization. In detail, the JVM guarantees that the instructions which modify the values of *volatile* variables are atomic and their value changes are immediately visible. They do not become accessible until they refer to fully initialized objects. Additionally, the specification of threads and the memory model prevents from reordering instructions defined on the *final* and *volatile* variables during compilation.

Let us imagine that we create a new object and then assign it to a normal object variable. The Java specification allows JVM to optimize a program code. After this optimization, this code can be reordered and executed in the following way: a raw object is created, then it is assigned to the variable and finally it is initialized. If reordering of instructions defined on the *final* and *volatile* were available, the value of *volatile* an *final* variables could be inconsistent.

Having analyzed the specification of threads and the memory model of JVM, the FIFO queue working in the configuration: one producer - one consumer has the following variables: $tail$, $head$ and $next$ which are of the *volatile* type. $Item$ is of the *final* type. Alg. 3 illustrates the implementation of this queue in Java. Because the specification of the memory model of JVM is limited to the functionality which is generally available in hardware. The proposed solution can be expressed in other languages too.

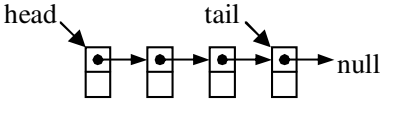

Fig. 5. A queue with a dummy node at the beginning

Algorithm 3. A FIFO Queue: one producer - one consumer

```
class TupleNode
{
   final TupleI item;
   volatile TupleNode next;
   public TupleNode(TupleI aItem,
                           TupleNode aNext) {
      item = aItem;
      next = aNext;
   }
}

volatile TupleNode head;
volatile TupleNode tail;

public PipeTupleList()
{
   head = tail = new TupleNode(null,null);
}

public final void pushNode(TupleI newNode)
{
   TupleNode node;
    node = new TupleNode(newNode, null);
   tail.next = node;
   tail = node;
}

public final TupleNode popNode()
{
   TupleNode tmp;
   if((tmp = head) == tail)
       return null;
   head = tmp.next;
   tmp.next = null;
   return head;
}
```

6 A Overview of the Architecture of Physical Operators

6.1 Operator Architecture

In contrast to PIPES, inputs (Fig. 7) and outputs (Fig. 8) of operators are represented by separate interfaces. Thanks to that we can directly refer to a given input or output. As a result we can create operators which wrap the original operators and expose the direct connection to the original operator inputs/outputs. Similarly to PIPES, the inputs of streams and operators implement the same interface (Fig. 7) in consequence operators can be linked with the help of stream buffers or interact directly without buffers. For

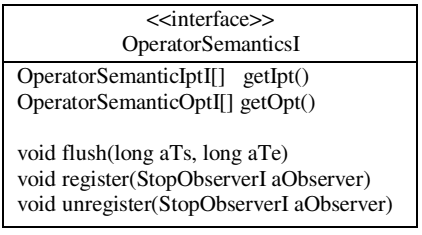

Fig. 6. The interface of a physical operator

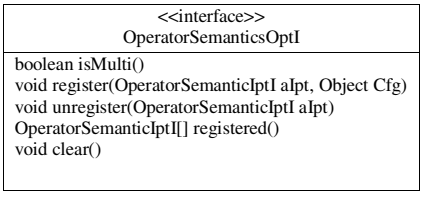

Fig. 7. The input interface

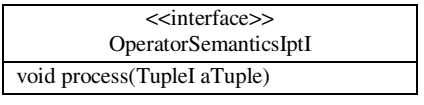

Fig. 8. The output interface

instance, those features are used by wrappers which measure the selectivity and the average time, which a physical operator takes to process a single input tuple. The interface of outputs has two implementations available. One assumes that we can register only one link to a given operator output and this version is more efficient. The other allows us to add multiple links to a given operator output. The information about the chosen implementation is available thanks to the *isMulti* method. The physical operator implements the interface showed in Fig. 6. In comparison to PIPES, this interface allows us to register observers which are notified when an operator finishes working. Thanks to those sources, sinks and middle operators can be controlled uniformly. The operator also defines a flush method which implements the algorithm triggered by BOUNDARY tuples [16]. The availability of this method allows us to trigger the flush of an operator without the need to process tuples.

6.2 Composed Operators

The architecture of operators allows us to link them directly. It is justified when the average time needed for an operator to process a tuple e.g. *filter* operator) is comparable to the average time needed to transfer a tuple by a stream. As a result, we can save the time needed to transfer tuples by streams. We refer to the sub graph of directly connected operators as a *composed operator*.

Figure 9 shows an example query. It consists of a *count type* window operator O1 and a *join* operator O2. The operators communicate directly if they are grouped. We have implemented the following grouping rules: *projection* operators are grouped with predecessors. Then, consecutive *selection* operators are grouped together. Finally, *count type* and *time type window* operators are grouped with *join* operators if they follow the *window* operators. Let us notice that grouping a *join* operator and a *count type window*

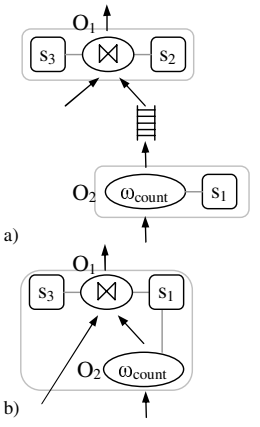

Fig. 9. Memory optimization for composed operators

operator allows us to optimize access to tuple collections. The *count type window* operator uses tuple collections of a predefined size. The *join* operator uses a tuple collection which is supplied with tuples from the *count type window* operator. When both operators work concurrently they need separate tuple collections (Fig. 9 a)): s_1, s_2. When the *join* operator is called directly by the *window* operator, the *join* operator can use the tuple collection managed by the *window* operator (Fig. 9 b)): s_1. Summing up, two separate collections *s1* and *s2* can be replaced by references to one tuple collection *s1*. This grouping rule not only reduces memory usage twice but also decreases the number of *insert* and *remove* operations.

7 Evaluation

We can divide operating time of a stream database into the time needed to calculate the physical operators and the time needed to perform DSMS operations, thread context switch operations and other system operations. In this section, we evaluate which techniques allow us to increase the contribution of the time in which the physical operators are processed. It is important to create a scalable stream database engine. We also want to compare the advantages of the better stream database engine architecture with the benefits of single physical operator optimizations. In order to achieve this aim we measured the average result latencies in the experiments. Those experiments are grouped in micro-benchmarks, which allows us to extract performance differences between different engine configurations.

Figure 10 shows queries of the micro-benchmarks. During a single experiment, we measured latencies of result tuples. Each of the data sources gen_1 and gen_2 generated 1000 tuples at a fixed data rate in the range from 1000 to 30000 [tuple/min]. The data sources generated tuples which contain uniformly randomly distributed numbers within the range $[0, 100]$. The selection operator O4 was defined with the predicate: $value1 < 50$. Both *count type* windows contained 500 elements. The *join* operator was defined with the predicate: $value1 < value2$. Let us note that the selectivity of the *join* operator was high, which caused peaks of data rates in the output of O4.

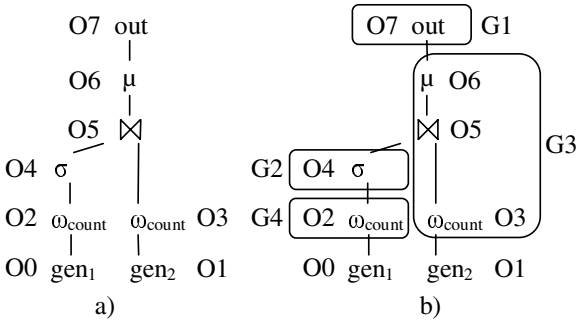

Fig. 10. The test definition: a) a query without grouping; and b) a query with enabled grouping

Table 1. Micro-benchmark configurations

Chart name	Scheduler type	Operator grouping	Tuple supply layer	Architecture type
CF1	Chain-flush	✗	synch	pool
CF2V1	Chain-flush	✓	V1	V
CF2V2	Chain-flush	✓	V2	V
CF3	Chain-flush	✗	V1	pool
CF4V1	Chain-flush	✗	V1	V
CF5V2	Chain-flush	✗	V2	V
CF6V2	Chain-flush	✗	V2	V
RR1	Round-Robin	✗	synch	pool
RR2	Round-Robin	✗	V1	pool

The experiments were carried out for the Chain-flush and Round-Robin algorithms. We tested the basic architecture presented in Sect. 3.2 with the simple tuple transport layer based on lock/unlock operations and with the tuple transport layer described in Sect. 4.1. Then, we measured latencies for the architecture (V) described in Sect. 3.3 with the Chain-flush scheduler. We distinguish two versions of this architecture. Version one (V1) is a straightforward implementation of the tuple transport layer described in Sect. 4.1. Version two (V2) is the implementation described in Sect. 4.2. The configurations of the micro-benchmarks are described in Tab. 1. The abbreviation *synch* in the column *supply layer type* means that the tuple transport layer is build upon lock/unlock operations.

The noticeable improvement is registered for the proposed worker architecture. Both tests FC2Loc and FC4Loc in Fig. 11 have lower latencies than the other experiments. It is worth noticing that the thread pool technique does not work efficiently in the stream database environment because this combination results in a high number of thread context switches.

In order to compare how the engine architecture scales when we use more workers of version two. Carried out tests with a number of workers ranging from one to four.

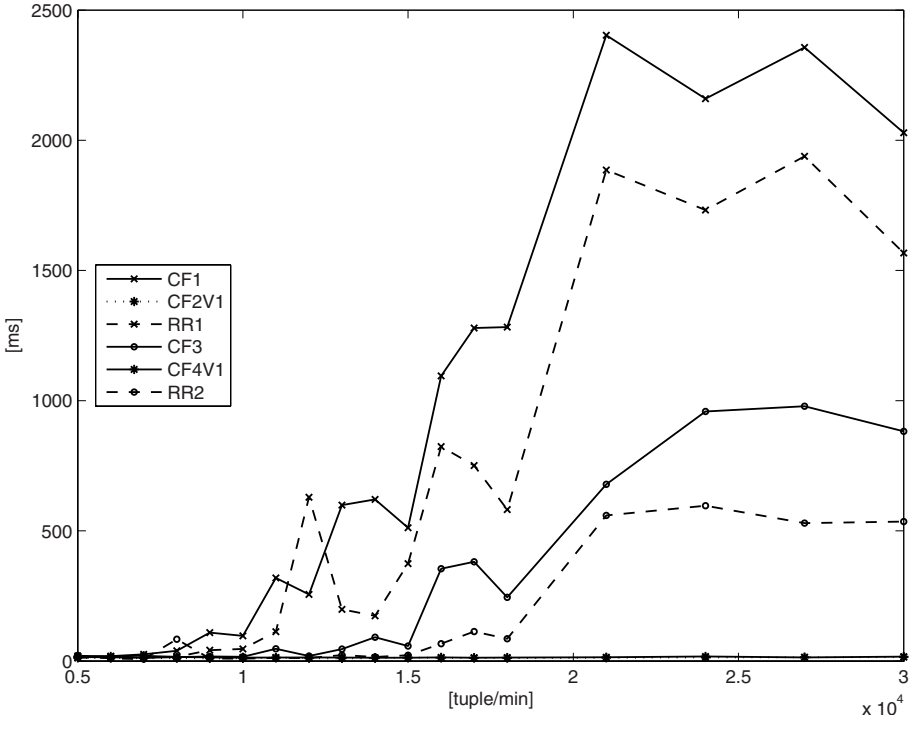

Fig. 11. The comparison of the worker pool and the introduced worker architecture

Figure 12 shows that the result latencies are almost the same for each configuration. This observation points out that the cost of keeping a higher number of threads is low when the physical operators have associated threads. What is worth emphasizing that a thread pool dose not guarantee that.

Now let us focus on the efficiency of the straightforward implementation of the tuple transport layer and its customized version. Figure 14 shows that version two of tuple transport layers offers a better performance. We measured the result latencies achieved by version two of the tuple transport layers for different stream buffer algorithms, which is shown by Fig. 13. Those tests were done for the engine consisting of three workers. We can see that the use of CAS instructions allows us to reduce the result latencies twice. This figure also shows that the stream buffer created for the configuration one producer one consumer works as fast as the wait free and block free FIFO queue [12].

The cooperation between the data stream engine and physical operators is another area of our test. Figure 15 shows the results of experiments conducted for the configurations with enabled or disabled creation of composed operators for version one of the tuple transport layer with one worker. Contrary to our expectations, the composed operators introduce a bigger latency in comparison to the basic configuration. It is the main drawback of the straightforward implementation of the tuple transport layer. Figure 10 b) shows the query with composed operators. Synchronized stream buffers

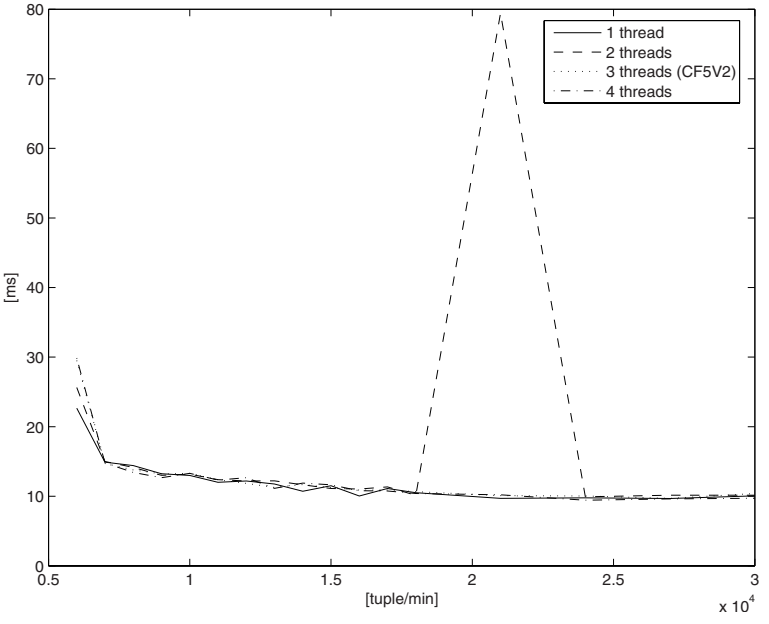

Fig. 12. The impact of the increasing number of workers

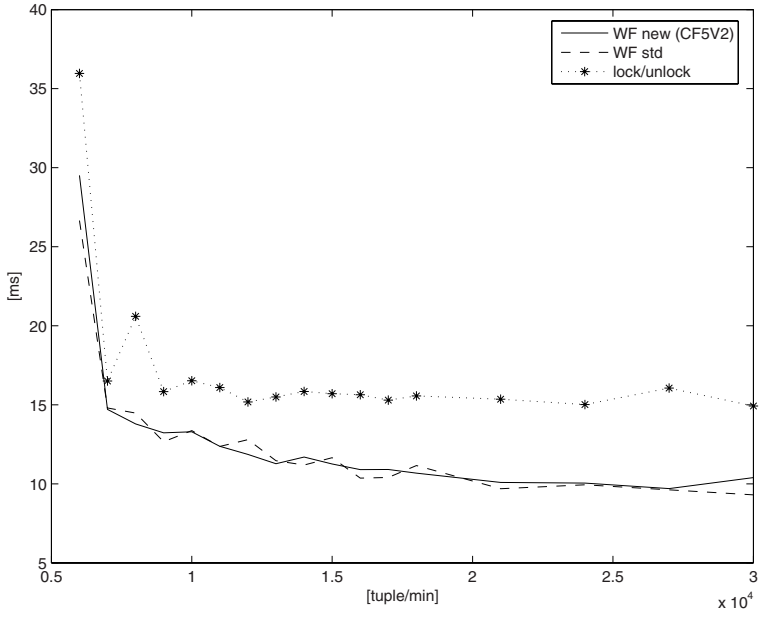

Fig. 13. The comparison of the stream buffer algorithms

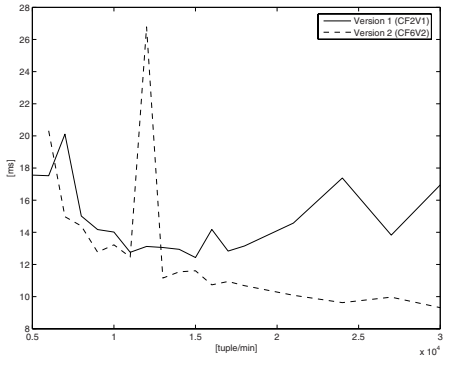

Fig. 14. The comparison of the tuple transport layers of version one and two

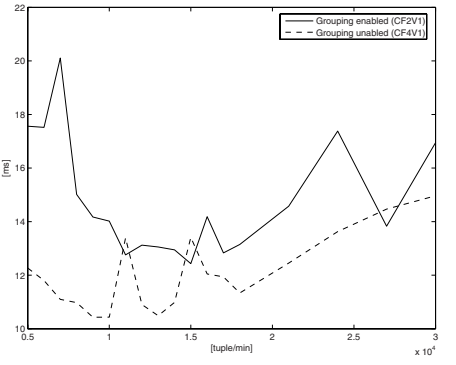

Fig. 15. Latencies measured for composed operators

have to be used when operator O1, which belongs to worker W1, is connected to an operator belonging to another worker. According to the above definition, synchronized stream buffers exist between: O1-G3, O0-G4 i G2-G3. The basic configuration of the query is shown in Fig. 10 a). There are only two synchronized streams: O0-O2 and O1-O3. Summing up, the configuration with composed operators has more synchronized stream buffers when we use the type two of tuple transport layer. Despite the fact that composed operators are faster, the final result latencies for the experiments conducted are bigger. In contrast, the type two tuple transport layer does not have such a weakness because it is controlled only by a worker thread. Thanks to that this architecture can work with both synchronized and local stream buffers.

8 Conclusions

In order to create a faster stream processor, we have analyzed the cooperation between a scheduler, physical operators, streams buffers and threads. The previous works like [5] describe stream processors on a higher level of abstraction. The tests carried out show that the architecture of the stream processing engine is crucial for achieving low latency and a scalable system. In the paper, we also proposed a possible architecture of the stream database engine and its tuple transport layers. Our test shows that the type two transport layer not only offers the best performance but also is easy to maintain.

In the paper, we also introduced wait free and block free FIFO queues for the configuration consisting of one producer and one consumer. This queue is based on CAS instructions and the features of the memory model of JVM. The tests show that, in comparison with the original wait free and block free FIFO queue, the introduced optimization does not change the result latencies.

Our forthcoming research will cover tests of physical operators migration between workers in the distributed environment and algorithms which allows us to balance the workload.

References

[1] Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator scheduling in data stream systems. The VLDB Journal 13(4), 333–353 (2004)

[2] Babcock, B., Babu, S., Datar, M., Motwani, R.: Chain: Operator scheduling for memory minimization in data stream systems. In: ACM International Conference on Management of Data, SIGMOD 2003 (2003)

[3] Jiang, Q., Chakravarthy, S.: Scheduling strategies for processing continuous queries over streams. In: Williams, H., MacKinnon, L.M. (eds.) BNCOD 2004. LNCS, vol. 3112, pp. 16–30. Springer, Heidelberg (2004)

[4] Cammert, M., Heinz, C., J.K.A.M.B.S.: Pipes: A multi-threaded publish-subscribe architecture for continuous queries over streaming data sources. Technical Report 32, Department of Mathematics and Computer Science, University of Marburg (July 2003)

[5] Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal 12(2), 120–139 (2003)

[6] Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS 2002: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 1–16. ACM Press, New York (2002)

[7] Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: SIGMOD Conference (2002)

[8] Ali, M.H., Aref, W.G., Bose, R., Elmagarmid, A.K., Helal, A., Kamel, I., Mokbel, M.F.: Nile-pdt: a phenomenon detection and tracking framework for data stream management systems. In: VLDB 2005: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, pp. 1295–1298 (2005)

[9] Barga, R.S., Goldstein, J., Ali, M.H., Hong, M.: Consistent streaming through time: A vision for event stream processing. In: CIDR, pp. 363–374 (2007)

[10] Krämer, J., Seeger, B.: A temporal foundation for continuous queries over data streams. In: COMAD, pp. 70–82 (2005)

[11] Bai, Y., Thakkar, H., Wang, H., Zaniolo, C.: Optimizing timestamp management in data stream management systems data engineering. In: IEEE 23rd international conference on ICDE 2007, pp. 1334–1338 (2007)

[12] Michael, M.M., Scott, M.L.: Fast and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on Principles of Distributed Computing, pp. 267–275 (1996)

[13] Greg Lavender, R., D.C.S.: Active object an object behavioral pattern for concurrent programming (1996)

[14] Sun: JSR-133: JavaTM Memory Model and Thread Specification (2004)

[15] Manson, J., Pugh, W.: Requirements for programming language memory models. In: St. John's, N. (ed.) PODC Workshop on Concurrency and Synchronization in Java Programs (2004)

[16] Tucker: Punctuated Data Streams. PhD thesis, OGI School of Science & Technology At Oregon Heath (2005)