Query Processing Using Negative and Temporal Tuples in Stream Query Engines

Marcin Gorawski, Aleksander Chrószcz

Query Processing Using Negative and Temporal Tuples in Stream Query Engines

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. In this paper, we analyze how stream monotonicity classification can be adopted for the introduced developed model, which processes both temporal and negative events. As we show, information about stream monotonicity can be easily used to optimize individual stream operators as well as a full query plan. Comparing our stream engine with such engines as CEDR, STREAM and PIPES we demonstrate how a primary key constraint can be used in different types of the developed stream schemes. We implemented all of the above techniques in StreamAPAS.

1 Introduction

The definition of time is the foundation of all stream processing systems. There are unitemporal, bitemporal and tritemporal stream models. The unitemporal stream model, adopted by many existing stream systems, describes events with timestamps belonging to an application domain, whereas the bitemporal stream model supports both application time and local processing time ([1]).

The current unitemporal systems can be classified into two groups: one representing events whose lifetimes are known a priori (e.g. [2,3]) and the other comprising events whose lifetimes also depend on the facts that will appear later (e.g. [4,5,6,7]). The second approach to data stream processing is more general. The lifetime of a single process is decomposed into a positive and a negative tuple that represent the beginning and the end of the lifetime, respectively. It is worth noticing that a single positive tuple does not carry the information of its exact or potential duration as it carries a temporal tuple which contains a *start* timestamp and an *end* timestamp.

Our objective is to create a unitemporal stream model which incorporates positive/negative and temporal tuples. Thanks to this it is easier to develop new operators in comparison with more general models such as [8,1]. Having joined temporal and positive/negative approaches, we can more easily control internal tuple collections of stream operators using both the event approach and the temporal approach. As a result, we can monitor not only the current operator state but also predict its duration.

Another challenge in designing a continuous query system is to minimize the amount of storage and computation. In order to show the advantages of our approach, we compare the *join*, *aggregate*, *minus* and *distinct* operators defined in different stream processing systems.

Additionally, we also adapt the monotonicity classification [9] to our model in order to optimize implementations of stream operators. This information is also helpful during the optimization of the full query plan.

2 Concept of Data Stream Processing

2.1 Tuple Lifetime

In contrast to such stream processor architectures as [2,7,3,5,8,10] our stream definition incorporates *positive temporal* tuples and *negative* tuples. Let T be a discrete time domain. Let $\mathrm{I} := \{[t_\mathrm{s}, t_\mathrm{e}) \mid t_\mathrm{s}, t_\mathrm{e} \in \mathrm{T} \wedge t_\mathrm{s} \leq t_\mathrm{e}\}$ be the set of time intervals.

Definition 1. *Stream: A triple* $S = (\mathrm{M}, \leq_{t_\mathrm{s},t_\mathrm{e}}, pk)$ *is a stream, if:*

M – *is an infinite sequence of tuples* $(type, e, [t_\mathrm{s}, t_\mathrm{e}))$,
where:
type – tuple type,
e *– attributes,*
$[t_\mathrm{s}, t_\mathrm{e}) \in \mathrm{I}$ *– start timestamp and end timestamp, respectively.*

$\leq_{t_\mathrm{s},,t_\mathrm{e}}$ – *is the lexicographical order relation over* M
(primarily sorting by t_s *and secondarily by* t_e*).*

pk – *attributes of primary key.*

The tuples are of two types: positive temporal and negative. The end timestamp of a positive temporal tuple defines the upper boundary of the tuple lifetime. When lifetimes of tuples are known at the time of their generation, the start and end timestamps define the exact lifetimes. When we do not know the lifetimes of tuples at the time of their generation, the end timestamps determine the upper boundary of the tuple lifetimes. Negative tuples expire positive tuples according to the primary key value. In comparison with positive temporal tuples, negative tuples have zero lifetime periods and are represented in figures by points.

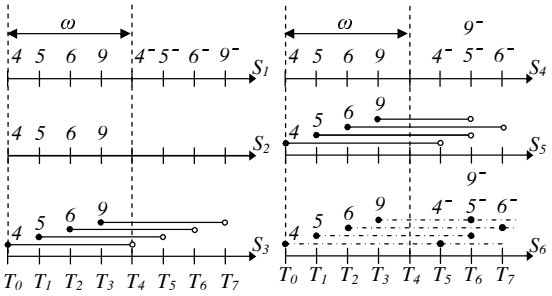

Fig. 1. Different stream models.

In order to show the flexibility of the defined stream model we compare it with the existing models.

Let us assume that we are monitoring some real phenomenon, which consists of elementary processes. They are described by attributes and timestamps. Streams in Fig. 1 illustrate the observation of the same elementary processes but they are represented by a different stream model. Stream S_1 uses two tuples so as to define the lifetime of an event. Positive tuples signal the beginnings of events and negative ones show the ends. Knowing the application time and the lifetime period, we can define a *time window* [7] which translates the input stream into the collection of valid tuples. For instance, when the lifetimes of the above processes are constant and equal ω, we can only transmit positive tuples as it is shown on S_2. When we know the exact lifetimes of tuples at the time of their creation, we can use temporal tuples [2] which contain *start* and *end* timestamps, as it is shown on S_3. The main advantage of temporal tuples is that they reduce the amount of transmitted data twice, which is shown on S_3. On the other hand, it becomes complicated to use temporal tuples [2] when their lifetimes are not known at the time of their creation. In such a case, time is divided into smaller periods. When a period elapses, there are produced temporal tuples which prolong the lifetimes of previously sent tuples that are still valid. Nevertheless, this solution generates a lot of additional data.

The observation of another real phenomenon is illustrated by streams S_4, S_5 and S_6. Let us assume that we don't know the lifetimes of events at the time of their creations. As a result we cannot represent them directly as it is shown on stream S_5. The integration of the temporal stream model and the model of streams with positive and negative tuples allows as to solve this problem. An example of this model is stream S_6. In this scenario, if we know that a tuple will have been expired by a given time, we assign this value to the *end* timestamp. Otherwise we assign infinity. In order to distinguish between *positive temporal* tuples and *temporal* ones, the broken lines represent the lifetimes of *positive temporal* tuples.

2.2 Stream Monotonicity

Depending on the source type, streams transmit either only *positive temporal* tuples or both *positive temporal* tuples and *negative* ones. The content of streams can be classified according to their monotonicity. As we show further, this property enables the stream database to optimize the implementations of operators by using more efficient tuple collections. In the stream processing system, we borrow the stream monotonicity classification from [9]. Let Q be a query and τ a point in time. Assume that at τ, all tuples with lower or equal timestamps have already been processed. The multi-set of input tuples at time τ is denoted as $S(\tau)$, whereas all the tuples from time 0 to time τ are denoted as $S(0, \tau)$. Furthermore, let $P_S(\tau)$ be the result multi-set produced at time τ and let $E_S(\tau)$ be the multi-set of expired tuples at time τ. The equation below defines the function of the result set update.

$$\forall\, \tau\; Q(\tau + 1) = Q(\tau) \cup P_S(\tau + 1) - E_S(\tau + 1)\,. \tag{1}$$

The types of stream monotonicity are defined indirectly. Using the above symbols we define operators which generate streams of a given monotonicity:

1. The monotonic operator is an operator that produces result tuples that never expire. Formally the property is described as follows: $\forall\,\tau\,\forall\,S\colon E_S(\tau) = 0$.
2. The weakest non-monotonic operator is an operator that produces result tuples whose lifetime is known and constant. Thanks to this the order in which those tuples appear at the operator input corresponds to the order of their expiration. This can be formalized as: $\forall\,\tau\,\forall\,S\,\exists\,c \in \mathrm{N}\colon E_S(\tau) = P(\tau - c)$.
3. The weak non-monotonic operator is an operator whose result tuples have different lifetimes but they are still known at the time of their generation. Let us note that the order of tuple insertion and the order of their expiration are different. Formally represented, it looks like: $\forall\,\tau\,\forall\,S\,\forall\,S'\colon S(0,\tau) = S'(0,\tau)$, it is true that $\forall\,t \in P_S(0,\tau)\,\exists\,e\colon t \in E_S(e) \wedge t \in E'_S(e)$.
4. The strict non-monotonic operator is an operator whose expiration of tuples depends on the input tuples that will arrive in the future. The lifetimes of tuples are not known at the time of their generation. This can be formalized as follows: $\exists\,\tau\,\exists\,S\,\exists\,S'\colon S(0,\tau) = S'(0,\tau)$ and $\exists\,e\,\exists\,t \in P_S(0,\tau)$ then $t \in E_S(e) \wedge t \notin E'_S(e)$.

The monotonicity of type one says that tuples of a given stream never expire. This means that streams of this type transmit only *positive temporal* tuples with infinity assigned to *end* timestamps. The monotonicity of type two is illustrated by S_3 in Fig. 1. Stream S_5 exemplifies a stream with monotonicity of type three. The last type of monotonicity is illustrated by S_6.

It is worth noticing that the higher the number of stream monotonicity, the more complicated the architecture of the tuple collection which is linked to a given stream. Collections connected with streams of monotonicity: 1, 2 and 3 process no *negative* tuples. Those collections check only *end* timestamps so as to find expired tuples. If a stream is of the weakest non-monotonic type, the order of tuple expiration and their arrival order are the same. As a result, if a simple list data structure is used, then potentially expired tuples exist only at its beginning. The stream monotonicity of type three has two potential implementations. The expired tuples can be identified by testing all the elements of a collection or we can add auxiliary indexes which store tuples in the *end* timestamp order.

According to [7], positive tuples and corresponding negative ones have the same values of attributes except for the flag which distinguishes types of tuples. In order to speed up finding *positive temporal* tuples which correspond to given *negative* tuples, we have introduced the primary key (PK) into stream schemes. If a stream of strict non-monotonic type has a defined PK, then we can create an index which simplifies processing negative tuples. In a traditional DBMS, the PK not only identifies records in relations but also excludes duplicates from relations. Similarly, if an operator is attached to a strict non-monotonic stream, it has to filter out duplicates and does not process them.

3 Stream Operators

It is worth noticing that the chosen stream model determines the lowest latency of result tuples which can be theoretically achieved. The stream model with temporal tuples assumes that the lifetime of generated tuples can't be changed (to be more precise

shortened). In consequence, some state-full operators (e.g. the aggregation operator) produces only tuples which elapse by t at time t. In summary, the effect of processing temporal tuples $[t_s, t_e)$ will be shown at an output stream after processing another tuple that has a grater t_s value. In contrast to that, operators defined in a stream model with positive/negative tuples always generate the result tuples which correspond to the most recent state of an operator. Despite the fact that our stream model uses *positive temporal* tuples, it doesn't have the above drawback.

3.1 Join Operator

In Fig. 2, we compare join operators for: a) the model with positive and negative tuples; b) the temporal model; and c) the mixed model. Streams In_1 and In_2 are the input streams and *Out* is the result of a *join* operator in Fig. 2c). Stream In_1 transmits only *positive temporal* tuples and In_2 transmits both types of tuples. Let us notice that the result streams *Out* in Fig. 2b) and Fig. 2c) transmit the same tuples but they have a different interpretation. The result tuples in Fig. 2b) define their lifetimes precisely, whereas the result tuples in Fig. 2c) define the upper boundaries of their lifetimes. In contrast to the stream model with positive and negative tuples, there is no negative result tuples in *Out*. This situation occurs because the negative tuples arrive in In_2 later than the upper lifetime boundary of result tuples in *Out*.

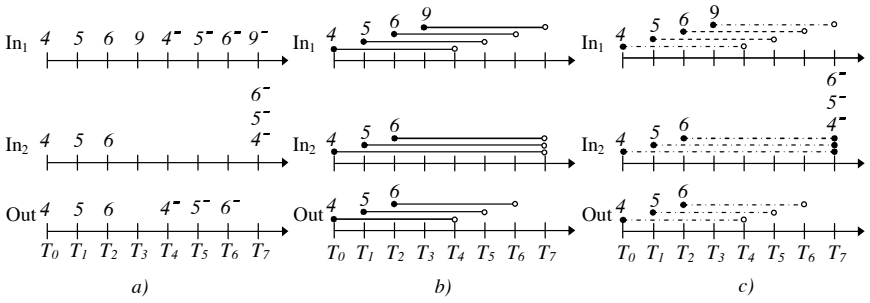

Fig. 2. Join operators for different stream models.

3.2 Aggregate Operator

In order to emphasize differences between implementations of *aggregate* operators, we analyze examples of calculating sum aggregates. Although the aggregate operator can have a defined grouping function on tuple attributes, in order to make figures easier to follow, we analyze aggregate operators without them.

1. Figure 3a) shows an operator designed to process temporal tuples [2] only. After a new tuple t enters the operator, the output stream is fed with result tuples whose *end* timestamp is lower or equal to the *start* timestamp of t. This operator puts forward

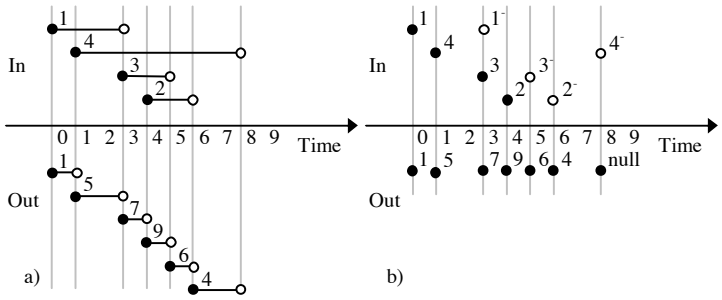

Fig. 3. Aggregate operators for different stream models: a) temporal tuples, b) positive/negeative tuples.

only aggregates whose lifetimes are fully defined. As a result the operator propagates the current values of aggregates when it receives another tuple.

2. The operator designed to process positive/negative tuples [7] is shown in Fig. 3b). In order to make this figure easier to follow, we have put the corresponding positive and negative tuples on the same level. This operator recalculates the aggregate value and pushes it to the output right after receiving le. Additionally, if we assume that the latest output value for a group overrides the previous value for the same group then this operator does not produce negative tuples. Moreover, after processing the last input tuple for a given group, the operator produces *null* tuple for this group [7].
3. The *aggregate* operator that processes streams of type 3) (Fig. 4) produces a tuple with the current aggregate value immediately after receiving a new tuple, similarly to the above operator. Aggregates are represented by *positive temporal* tuples. The lifetime of tuple t that represents the aggregate belonging to group g tells us when the aggregate value finally expires according to tuples received by g before t was produced. The output of this operator is always of strict non-monotonic type.

Now we summarize the *aggregate* operator for the above stream models:

1. The *aggregate* operator that processes temporal tuples is always delayed by one tuple. In consequence, we observe the old aggregate value until a new tuple arrives. In order to shorten the delay, the authors [2] suggest dividing the lifetime of an entering tuple into smaller periods. As a result the *aggregate* operator more frequently refreshes the output stream. However, it generates more tuples which have to be processed.
2. The aggregate operator that processes positive/negative tuples is free of the above imperfection. Figure 3 b) shows a solution which generates only positive tuples. Let us assume that the *aggregate* operator has defined grouping attributes. In this case, when new result tuple t_{new} is produced, it overrides the previous aggregate value of the same group as t_{new}. In consequence, successive operators attached to this *aggregate* operator have to know the definition of the grouping function so as to detect tuples expiration correctly. As a result, attaching subsequent operators to such an *aggregate* operator becomes complicated. If we want to avoid this drawback, the operator has to produce both positive and negative tuples.

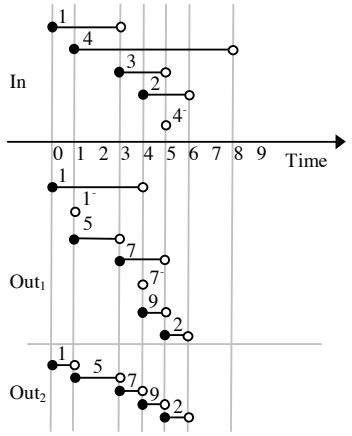

Fig. 4. Aggregate operator which process positive temporal and negative tuples.

3. Considering our stream model, lifetimes of tuples are determined by either *end* timestamps or negative ones. We do not make use of the assumption that a new tuple overwrites the previous one. Consequently, linking aggregate operators in a query plan is easy. Moreover, the result aggregates are propagated to the output when a new tuple arrives, like in the previous operator description.

Now let us compare the aggregate operators with the defined grouping attributes. The input stream is divided into N_{gr} groups.

The aggregate operator for model 1) generates result tuples for all N_{gr} groups after processing a new input tuple because all aggregates of groups have to be propagated to the output. The explanation of it is as follows: when a new aggregate is sent to the output it has to have a defined exact lifetime. In consequence it is generated when another tuple arrives at the input. Having N_{gr} groups of aggregates, the next input tuple triggers the calculation for each group.

The aggregate operator for model 2) updates the aggregate value of the group affected by an input tuple. The number of result tuples depends on the operator configuration. It can generate either positive and negative tuples or only positive tuples like in Fig. 3b). Even if we consider the configuration which generates positive and negative tuples, it generates less tuples than the operator defined for mode 1).

The aggregate operator in model 3) updates the aggregate value of the group affected by an input tuple, similarly to the operator defined on model 2). *Positive temporal* tuples determine lifetime boundaries. Thanks to this the operator does not produce as many tuples as the operator in model 1).

3.3 Stream Compression

When DSMS is loaded, more tuples are stored in stream buffers. In consequence, a stream operator processes a packet of tuples in one round of a scheduler assignment

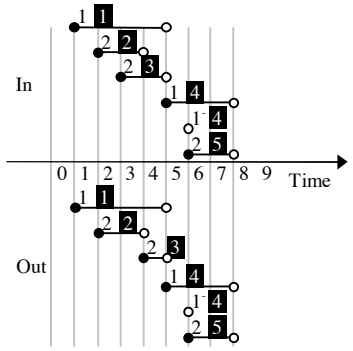

Fig. 5. A sample output of the distinct operator attached to a strict non-monotonic stream.

[11]. Let us assume that the *aggregate* operator buffers the result tuples while processing this packet. Having processed the last tuple of the packet, the buffer consists of *positive temporal* tuples and *negative* ones. Now we can check if each negative tuple t_n corresponds to a positive temporal tuple t_p which also appears in this buffer. If the above conditions are met, we can reduce those tuples as follows: a *positive temporal* tuple t_p has updated $t_p.t_e = t_n.t_s$ and a *negative* tuple t_n is removed. Finally, the buffer is compressed and subsequently transmitted to the output stream.

3.4 Distinct Operator

The *distinct* operator uses an internal collection, which contains input tuples as long as they are valid. The input tuple has attributes DST_{Attr} which represent distinct values. When a *positive temporal* tuple t_1 arrives, the internal operator collection is checked for the existence of DST_{Attr} values of t_1. If not, t_1 is propagated to the output and added to the collection. Otherwise it is only added to the collection. When t_1 is terminated by either the lifetime limit or the corresponding negative tuple, the *distinct operator* checks if there is a valid tuple with the same DST_{Attr} values as t_1. If there is such a tuple t_{next}, the operator produces a new result which has attribute values from t_{next} and the lifetime: $start = t_1.end$, $end = t_{next}.end$. Let us consider the monotonicity of the result stream. If the operator is fed with a strict non-monotonic stream, then the result stream is also of that type. Otherwise it is of the weak non-monotonic type. A sample output of the *distinct* operator is shown in Fig. 5. The numbers printed in black represent DST_{Attr}, and the numbers in white are PK. Let us consider a scenario in which negative tuple 2^- arrives at $t = 4$: $< 2, [4, 4) >^-$. Without knowing the PK value, we cannot chose the right tuple to remove because both tuples $< 2, [2, 4) >$ and $< 2, [3, 5) >$ have the same attribute values. On the other hand, they have different lifetimes. Depending on which tuple was removed, a different result tuple will be produced. In contrast to this operator, the *distinct* operator defined for streams with positive and negative tuples [7] does not have to use the PK. All positive tuples represent an infinite lifetime so that if there exist duplicates, the operator removes just one of them.

The algorithm below gives the pseudo code of the distinct operator which is attached to the strict non-monotonic stream. This algorithm uses the following collections:

- Hash table, *HPK*, which stores input tuples sorted by PK.
- Collection, *HDST*, which contains buckets of tuples sorted by DST_{Attr}.
- Collection, *LExp*, which stores input tuples sorted by *start* and *end* timestamp.
- Collection, *DL*, which stores tuples that generate distinct tuples.

Let us notice that the collection *HPK* can be removed when the operator is attached to the streams of other monotonicity types.

Operator distinct; main procedure.
Input: t - an incoming tuple

```
1  RemoveExpiredTuples(t)
2  If t is a positive temporal tuple
3    If t is not found in HPK
4      Add t to HPK
5      Add t to LExp
6      If HDST doesn't contain bucket for t
7        Create bucket b; add b to HDST
8        Add t to b
9        Add t to DL
10       Produce positive temporal tuple to Output
11     Else
12       Add t to that bucket
13 If t is a negative tuple
14   If t is found in HPK
15     Remove t from HPK
16     Remove t from LExp
17     Get bucket b containing tuple t
18     Remove t from b
19     If b.size() == 0
20       Remove t from DL; produce negative tuple to Output
21       Disconnect b from HDST
22     Else If t ∈ DL
23       Remove t from DL; produce negative tuple to Output
24       Add the next tuple from b to DL,
25       Produce new distinct tuple to Output
```

3.5 Minus Operator

The *minus* operator calculates $S - R$ and sends the result to stream *Out*. This operator uses internal collections one which contains valid tuples from *S* and the other one contains valid tuples from *R*. Tuple *t* at input *S* (*R*) has a counterpart in the *R* (*S*) collection

Procedure: *RemoveExpiredTuples(Tuple t)* definition.
It removes expired tuples from the collection of the operator.

```
1  Get next tuple t_exp to expire from LExp
2  While (t_exp <> null) and (t_exp precedes or coexists with t)
3    Get bucket b containing tuple t_exp
4    Remove t from LExp
5    Remove t from HPK
6    Remove t_exp from b
7    If b.size() == 0
8      Remove t from DL; produce negative tuple to Output
9      Disconnect b from HDST
10   Else If t ∈ DL
11     Remove t from DL; produce negative tuple to Output
12     Add the next tuple from b to DL
13     Produce new distinct tuple to Output
14   Get next tuple t_exp to expire from LExp
```

if there exists tuple t_R (t_S) whose values of attributes $Attr_R$ ($Attr_S$) are equal to corresponding values of attributes $Attr_S$ ($Attr_R$) of t. Let us consider the following scenario. Tuple t_R arrives at input R and it has its counterpart t_S in the S collection. In consequence, the operator produces a *negative* tuple which terminates corresponding t_S. This scenario shows that even if the input contains only *positive temporal* tuples the result may contain *negative* tuples. Summing up, the *minus* operator always generates a result stream of strict non-monotonic type. Fig. 6 shows a sample execution of such an operator.

4 Tests

We tested the proposed stream compression algorithm in order to check how this algorithm works on the output of an aggregate operator with a defined grouping function. In this configuration, the distance between the corresponding *positive temporal* tuple and a *negative* one is higher. The number of result tuples which are generated after processing one input tuple increases as the number of groups increases. Moreover, aggregate operators produce negative tuples when the tuples that belong to the same group overlap as it is illustrated in Fig. 4. The above features make this sample query a good benchmark. Figure 7 shows the query plan, which is a chain composing of a generator, a *range* window, a *filter* operator, an *aggregate* operator, an *map* operator and a sink, respectively. We defined this test in StreamAPAS using the query statement that is shown on the right side of Fig. 7.

In order to measure the compression efficiency and the latency time of the results, we present the configurations in which the *aggregate* operator has enabled or disabled compression. A single experiment consists of processing 1000 tuples generated by gen_1

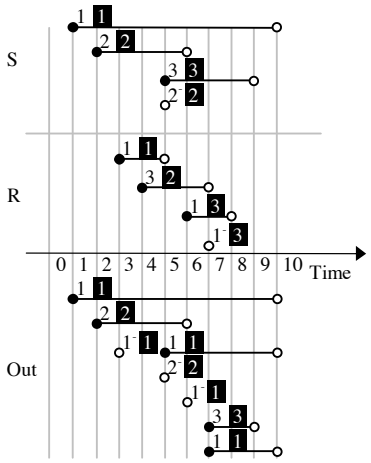

Fig. 6. The example result stream of the operator minus.

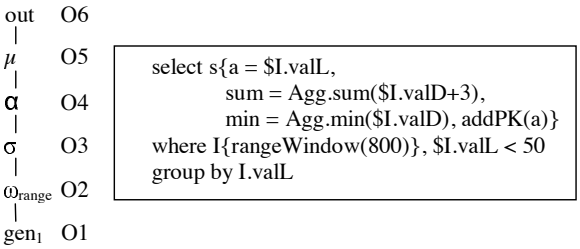

Fig. 7. The tested query.

for both configurations. Those measures were repeated for the input stream load which starts from $5000\,[tuples/min]$ to $60000\,[tuples/min]$ produced by generator gen_1. The generator produces a uniform load measured in [tuples/min]. Additionally, for each tuple the generator assigns a group $g \in 0..N$ using uniform distribution. In our experiments, value N equals 100. Figure 8 shows the changing efficiency of the compression when the system has a higher load and Fig. 9 compares the latency of result tuples. The lines named 100N and 100C represent configurations in which compression was disabled and enabled, respectively.

Figure 8 shows that the compression of the result stream is weaker than it could be expected. It is caused by the N groups which increase the distance between the corresponding *positive temporal* and *negative* tuples. As a result, there is a low probability that a single packet contains elements that can be reduced. Figure 9 shows that the impact of compression algorithm on the result latency is huge. When the system has a low load and the compression is enabled, the latency is from 20% to 50% higher. Summing up, although the stream model that is based on *positive temporal* and *negative* tuples offers a simple stream compression algorithm, its real usability depends on the distance between the corresponding positive temporal and negative tuples. A similar compres-

sion algorithm has been introduced in the systems PIPES and CEDR. In PIPES, the context of stream compression is different. Let us assume that we have a query plan with an input stream *S*. In PIPES, the tuples of stream *S* can be divided into a series of tuples 1 time unit in length so as to reduce the result latency of the query plan [2]. In this configuration, the stream compression efficiency is high.

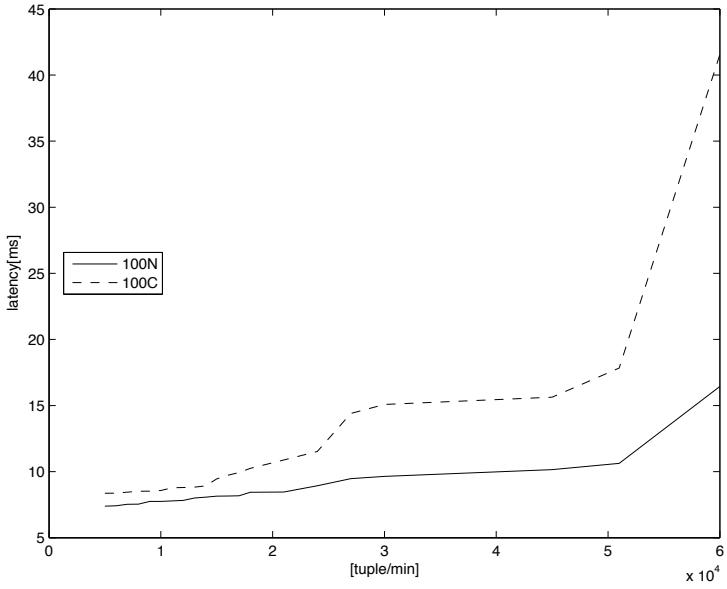

Fig. 8. Compression efficiency.

5 Conclusion

The functionality of operators which have been introduced to stream databases comes from traditional relational databases. In StreamAPAS, we develop a stream database which adapts the basis of relational databases and temporal databases [12,13,14] because they are also oriented towards processing ordered data. In contrast to other systems, our system processes both *temporal positive* tuples and the *negative* ones in a unitemporal stream model. As a result, this allows us to freely model an expiration time in a unitemporal stream system, which is useful in many application scenarios.

In contrast to CEDR, we have adopted stream monotonicity classification [9] and a primary key constraint. Depending on the type of monotonicity, the interpretation of a *positive temporal* stream can be simplified. In consequence, the stream operators do not have to maintain all collections. Additionally, the monotonicity can support the query model optimization as it was suggested in [9]. We have also introduced a primary key constraint in order to define the semantics of negative tuples. The current systems

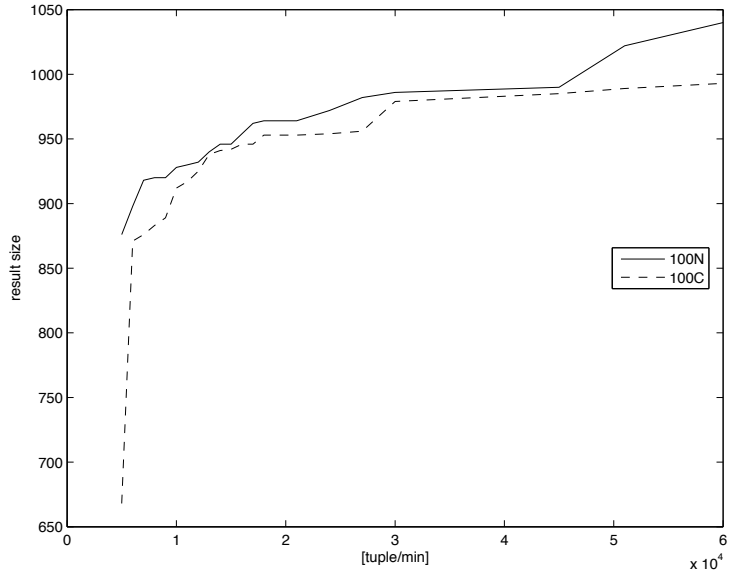

Fig. 9. The impact of compression on latency time.

that process negative tuples assume that each tuple has only one ID attribute which identifies the corresponding tuple to be removed. This causes the problem of calculating a new ID attribute for the join operator. In our approach, we have two options. Either the user can define the function which calculates a new primary key attribute upon the attributes of the joined tuples, or the user can define the primary key composed from a few attributes.

Our work aims to research new analytical systems which we name stream data warehouses. Traditional data warehouses are designed to separate the loading process (ETL) from processing analytical queries. Stream data warehouses are to enable users to monitor the analysis in a continuous way. Therefore, we have introduced a new stream data model which joins temporal information with tuples. Thanks to this we will be able to adapt the indexes of data warehouses in the streaming model.

References

1. 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)
2. Krämer, J., Seeger, B.: A Temporal Foundation for Continuous Queries Over Data Streams. In: COMAD. pp. 70–82 (2005)
3. Krämer, J.: Continuous Queries Over Data Streams Semantics and Implementation. PhD thesis, Philipps-Universität Marburg (2007)
4. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems. In: PODS '02: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 1–16. ACM Press, New York, NY, USA (2002)

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. Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, M., Convey, Ch., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., Zdonik, S.: Retrospective on Aurora. The VLDB Journal 13(4), 370–383 (2004)
7. Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G., Elmagarmid, A.K.: Query Processing Using Negative Tuples in Stream Query Engines. Technical Report 04-040, Purdue University (2005)
8. Tucker, P.: Punctuated Data Streams. PhD thesis, OGI School of Science & Technology At Oregon Heath (2005)
9. Golab, L.: Sliding Window Query Processing over Data Streams. PhD thesis, University of Waterloo (2006)
10. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., M. Datar, G.M., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Resource Management, and Approximation and in a Data Stream Management System. In: Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003), pp. 245–256. Asilomar, CA, USA (2003)
11. 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), pp. 253–264. San Diego, CA, USA (2003)
12. Ozsoyoglu, G., Snodgrass, R.T.: Temporal and Real-time Databases: A Survey. IEEE Transaction on Knowledge and Data Engineering 7(4), 513–532 (1995)
13. Slivinskas, G., Jensen, C.S., Snodgrass, R.T.: Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. In: (ICDE'00) Proceedings of the 16th International Conference on Data Engineering, p. 547–558. IEEE Computer Society, Washington, DC, USA (2000)
14. Slivinskas, G., Jensen, C.S., Snodgrass, R.T.: A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering. IEEE Transaction on Knowledge and Data Engineering 13(1), 21–49 (2001)