ReTIN: Indexing Schema for Soft Real-Time Data Streams

No Author Given

No Institute Given

Abstract. The paper deals with the indexing of a complex type data stream where a portion of the stream that represents the content of its sliding window is stored in a database. The processing of this data must often meet some real-time constraints. We present here a novel indexing schema/framework referred to as ReTIN (Real-Time INdexing), the objective of which is to allow indexing of complex data arriving as a stream to a database with respect to soft real-time constraints for the maximum duration of insert and select operations. In contrast to hard real-time constraints, the softness means that constraints violations are allowed but their number must be minimized. In ReTIN, soft real-time constraints are met with some level of confidence. The basic idea of ReTIN is a combination of sequential access to the most recent data to less recent data that has been indexed and stored in the database. The collection of statistics makes balancing the indexed and unindexed parts of the database efficient. We have implemented ReTIN PostgreSQL DBMS and its GIN index to store and index data. Experimental results presented in the paper demonstrate some properties and advantages of our approach.

1 Introduction

During the last two decades a new class of data sources and data-intensive applications that process data of such sources has appeared. The data produced by these data sources and processed by these applications is modeled best as data streams [6]. In contrast to traditional database management systems, in which data is stored as persistent relations, a data stream is a continuous, possibly infinite, stream of changing and often of high dimensional data that must be processed under some real-time constrains. Examples of such applications include network monitoring, surveillance, multimedia, financial and other sensor data.

Our research of soft real-time data stream indexing was motivated by the need of querying metadata of moving objects produced by computer vision modules of our experimental SUNAR (Surveillance Network Augmented by Retrieval) system [12]. The goal of SUNAR is to persistently track moving objects in a space monitored by multiple smart cameras (at airports). These cameras produce multiple data streams, which are necessary to index in order to make the similarity search and spatio-temporal queries possible in near real-time. The

identity preservation queries are performed every time an object (or subject) appears in a video. Although neighboring cameras query one database only, the load is huge.

Another application domain that require our research on real-time indexing is computer security, namely intelligent intrusion detection systems that monitor the computer network traffic in real-time. We use (Argos) honeypots [ARGOS] to detect exploits on a highly monitored machine and this information is then used to stop such incidents in the whole network. This includes also identifying all computers possibly compromised up till now using complex network data queries.

A different example could be a geographic information system [18] that stores and evaluates data issued from an array of spatially referenced sensors to prevent a natural disaster. The longer the system takes to detect any incidents, the more we loose. There is barely enough time to re-index the database under these circumstances in order to enable to query other sensors' data.

In this paper, we present a novel indexing schema/framework referred to as ReTIN (Real-Time INdexing), the objective of which is to allow indexing of complex data arriving as a stream to a database with respect to soft real-time constraints. A soft real-time constraint, in contrast to a hard constraint that has to be always met, allows violations but their number is minimized and the query speed optimized.

The indexing schema is based on database partitioning. Values of the data stream are inserted into an unindexed partition that contains the most recent data. Less recent data is stored in other partitions that are indexed. Queries are processed using both full scan and an indexed-based access method. Schema maintenance is controlled by two soft real-time constraints. It includes moving data from the unindexed part to the indexed one, a simple load shedding and use of a technique similar to piggybacking to collect and update statistics. The schema maintenance and indexing runs as a completely independent process of new data insertion and query processing. We presume no changes to client SQL queries (except continuous ones, of course)[1].

The rest of the paper is organized as follows. The most influential related works are mentioned in the next section. Section 3 contains problem formulation and section 4 describes the structure of the proposed indexing schema and operations on it. Section 5 includes several comments on ReTIN implementation. Section 6 presents experimental results and section 7 conclusions and future work.

2 Related work

Data stream theory and technology has become a hot research topic since the beginning of this century. Several Data Stream Management Systems (DSMS)have

[1] The ReTIN binary and source code including experimental utilities for the PostgreSQL database system can be obtained at http://www.fit.vutbr.cz/research/view_product.php.en?id=129 under GNU GPL.

been developed [8], [1], [4]. Their motivation was to provide monitoring and tracking applications capabilities and functionality not supported directly by traditional DBMSs. They have brought techniques such as continuous querying [5], sliding window query processing [15], approximate query processing [14], sampling, sketching and synopsis construction [3]. Moreover, techniques of query optimization and Quality of Service (QoS) to guarantee a certain level of performace, as load shedding [24], [10], have been developed. In addition, some data stream specific indexing methods have appeared. Multi-granularity aggregation indexing [13] is an integrated structure managing summarized information of snapshots. Po-tree [18] is an indexing structure for spatio-temporal databases with soft real time constraints which combines two different structures for spatial and temporal dimensions. A continuous query index for RFID data streams [21] is an index that is built on queries rather than data records. Examples of other data stream indexing techniques are multi-resolution indexing scheme [13] or integrated distributed indexing architecture [9].

On the other hand, DBMSs improved their capabilities too. SQL:2003 introduced logical and physical windows and several windowing agregate functions. These features are also supported in many commercial DBMSs. This frameworkhe has been extended in the Expresive Stream Language (ESL) of the Stream Mill system [7]. Although push-based processing is usually presented as a prefered processing model for data streams there are applications where pull-based processing and DBMSs can be successfuly applied. Examples have been mentioned in the previous section. Our approach is focused on such applications. It is based on modified techniques that reduce the problem of high index maintenance overhead for updates (inserts of new data stream values in our case) and techniques that help to guarantee QoS measured by real-time constraints. Similarly as in differential files [23], we confine the recent data stream values to a separate partition. In addition, their indexing is deferred and done in batches, which is the idea of differential indexing.

Our approach also profits from several techniques known from real-time databases. Their research received a lot of attention, but the primary objective of real-time support in these databases was different compared to data streams [16]. First of all, our concept of a soft real-time constraint is similar to one known from real-time databases [2]. Moreover, the idea of our approach is similar to a real-time index/cache consistency maintenance technique Codir for text retrieval systems presented in [11] or to a differential indexing with LSM-tree [19]. Codir builds a transient index for new document updates and queries are processed using both permanent and transient index. To minimize performance overhead associated with document database updates, Codir integrates transient index with permanent index lazily using piggybacking [25] or batch processing.

3 Indexing schema concepts

The ReTIN indexing schema supports the most important real-time data stream operations on a single table in the database, in which a portion of the stream is

stored – insert (inserts a new data stream element) and select (executes a query). There are three parameters that control the behavior of the indexing schema: a maximum time of insert operation T^{MAX}_{INSERT}, a maximum time of select operation T^{MAX}_{SELECT} and a confidence factor r. Then, the schema meets the soft constraints T^{MAX}_{INSERT} and T^{MAX}_{SELECT} with confidence $r \times \sigma$ where σ is a standard deviation of the execution times distribution and r is a selected confidence factor as described below.

3.1 Problem formulation

Let ds be a data stream of data elements of a type dt, which is complex in general – composite and/or multiple-valued. The data stream is processed using a sliding window. Let us assume that the window is larger than it fits in the main memory. The content of the window is stored in a database table D, which is not necessarily normalized. The size of the window is not specified in advance. Instead, real-time constraints T^{MAX}_{INSERT} and T^{MAX}_{SELECT} are specified for durations of insert and select operations on table D. Thus, the size of the window is dependent on the duration of these operations.

It is required to minimize the number of violations of the timing constraints, so they can be considered to be soft real-time constraints. The softness of the constraints is dependent on the probability of their violation. We can introduce estimates for maximum processing times of insert and select operations on D: $M[T_{INSERT}]$ and $M[T_{SELECT}]$, respectively:

$$M[T_{OPERATION}] = \mu(T_{OPERATION}) + r \times \sigma(T_{OPERATION}) \qquad (1)$$

where `OPERATION` is either `INSERT` or `SELECT`. The estimates are derived from the expected (duration of the operation) $E[T_{OPERATION}]$. It is given by the average processing time of the operation $\mu(T_{OPERATION})$, and its standard deviation $\sigma(T_{OPERATION})$. The real value r is a confidence factor that determines the confidence interval or the allowable probability of the constraint violation together with the standard deviation σ. For example, provided *Gaussian normal distribution* the value $r = 3.0$ results in 99.73 % probability of not exceeding the $T^{MAX}_{OPERATION}$.

We have chosen operations select and insert because they are typical for data streams and data of temporal character. Data modification operations are usually not defined on streams and they have a minor effect on the system performance.

4 Proposed solution

The idea of the ReTIN indexing schema is that table D consists of two logical partitions, namely DS and DI that differ in access methods and DX that stores obsolete data that are not accessed using queries on D. Data in DS is accessed by means of full scan whereas data in DI is indexed. All incoming data of the data stream ds is inserted into DS. DI contains less recent data of the stream that

was relocated there from DS partitions during indexing schema maintenance operations in the past. The objective of the schema maintenance operation is to improve performance in order to meet the soft constraints T^{MAX}_{INSERT} and T^{MAX}_{SELECT}. There are two cases that result in accomplishing the schema maintenance operation:

- *CASE 1:* duration of insert or select operation that is to be executed would violate T^{MAX}_{INSERT} or T^{MAX}_{SELECT} with high probability,
- *CASE 2:* full scan access of DS takes more time than access to data in DI.

In order to be able to check for these situations, some temporal statistics must be gathered during the execution of operations on ReTIN. In *CASE 1*, reduction of the DI part may be necessary. It is done by moving the less resent data, which is considered to be obsolete to DX storage, or by deleting it. This data will not be available further in ReTIN by querying table D.

The schema maintenance operation should not block and significantly delay an insertion of new stream data and querying the data in D. We solve it in such a way that the maintenance operation is performed asynchronously as a background process (in time and space) to insert and select operations. In addition, the maintenance operations use partitions and must be atomic.

Both T^{MAX}_{INSERT} and T^{MAX}_{SELECT} should be met, but it is not acceptable to rebuild the index on DI whenever after insert or before a select operation (the lazy approach), because it may break the time constraints.

Thus, our approach is advantageous in at least two situations. First, when the duration of a sequential scan for a select operation on D would take much longer than a corresponding index scan or when it would violate the constraint T^{MAX}_{SELECT}. Second, when updating an index would take much longer than a simple insertion of data or it would violate the constraint T^{MAX}_{INSERT}:

$$E[T_{SELECT\,DS}] \gg E[T_{SELECT\,DI}] \quad \text{or} \quad E[T_{SELECT}] > E[T^{MAX}_{SELECT}] \tag{2}$$

$$E[T_{INDEX}] \gg E[T_{INSERT}] \quad \text{or} \quad E[T_{INDEX}] > E[T^{MAX}_{INSERT}] \tag{3}$$

where $E[T_{OPERATION}]$ stands for expected duration of a corresponding operation.

Having in mind the above conditions, all the data are inserted to the unindexed subtable DS, in which they are kept until it stops being advantagous. This means, until the query duration on DS exceeds either the query time of the indexed part or it exceeds the soft constraint T^{MAX}_{SELECT}. If this condition (2) is approaching, there is time to relocate the data from DS into DI asynchronously. The asynchronous transfer is important not to influence the queries in progress. Thus it must be processed in parallel way, both in time and memory. In this way we can guarantee the soft query time constraint T^{MAX}_{SELECT}.

4.1 Structure

The basic elements of the ReTIN indexing schema are shown in Figure 1. It consists of the hierarchy of three tables. All the tables have the same schema

(t : `TIMESTAMP`, d : dt), where `TIMESTAMP` is an underlying DBMSs data type for timestamp values and dt is the type of an element of the data stream ds. Each row of tables contains a value of one element of ds in column d and the time of insertion into the database in column t. Because the data type dt can be a composite and/or multiple-valued, the column d can contain subcolumns or nested collections.

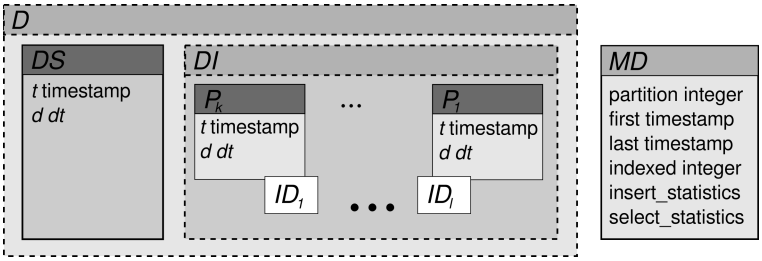

Fig. 1. Basic elements of the ReTIN indexing schema. Table D encapsulates partition DS, and indexed one DI. DI consists of partitions P_i. Metadata related to all partitions P_i and DS are stored in the MD metadata table.

- Table D – a virtual table that encapsulates tables DS and DI. All schema clients' insert and select operations run on it.
- Table DS – a base table containing the most recent data of the data stream ds that has been inserted into D. There is no index on column d or its subcolumns or nested collections. The data is accessed by means of a full scan.
- Table DI – a virtual table that encapsulates one or more base tables $P_i (i = 1, \ldots, k, k \geq 1)$ referred to as partitions. The number of partitions k changes in time. There is one or more indices $ID_j (j = 1, \ldots, l, l \geq 1)$ on column d of DI or its subcolumns or nested collections. DI encapsulates the indexed part of table D.
- Table MD – a base table that contains some temporal statistics concerning the insert and select operations on tables D, DS and DI, as illustrated in Figure 1. It will be described in more details in the following section.

4.2 Operations of ReTIN

The ReTIN indexing schema provides two logical operations to its clients:

- *insert(e: dt)* – inserts an element e of the data stream ds into the table D,
- *select(q: query)* – selects data from D as a result of a query q.

Both these operations rely on corresponding operations of the underlying DBMS. Here we consider `INSERT` and `SELECT` statements conforming to the

SQL standard. The only additional activities include the logging of queries and the update of temporal statistics used for the decision on whether the *indexing schema maintenance operation* should be performed. It is an internal operation of the ReTIN role, which is meant to change the content of tables DS and DI in such a way that the soft constraints T^{MAX}_{INSERT} and T^{MAX}_{SELECT} will be met for a time period. This operation is performed asynchronously to the insert and select operations. Let us assume that there is a concurrent process responsible for the indexing schema maintenance operation.

All three operations are described more formally below. Inputs, outputs (only data is considered here) and preconditions used are specified first, then the algorithm is described in the pseudocode.

```
Algorithm 1: Operation insert(e : dt)
  Input: e -- an element of the data stream
  Precondition: "INSERT INTO D VALUES(e)" performed

INSERT INTO DS VALUES (current_timestamp, e);
update_insert_statistics();
SIGNAL "check RT constraints";
```

Algorithm 1 presents the `INSERT` operation. It is ensured, that the new value will always be inserted into the table DS without the need to update any index. The operation *update_insert_statistics()* updates statistics related to the insert operation. These statistics are stored in the metadata table MD. The operation updates sums that are necessary to compute the mean (μ_{INSERT}) and the standard deviation (σ_{INSERT}) of the insert operation processing time.

The last statement `SIGNAL` represents the sending of an asynchronous message to the process responsible for the indexing schema maintenance operation. In fact, it is not appropriate to send the message after each insert. Instead, it should be send after a batch of inserts.

```
Algorithm 2: Operation select(q : query)
  Input: q -- a~select statement, retrieves data from D
  Output: rs -- a~result set of query q
  Precondition: "SELECT d FROM D" performed

rs = EXECUTE q;
update_query_statistics(q);
```

Algorithm 2 presents the `SELECT` operation. The select statement q is executed by the DBMS. Its execution is optimized by DBMS's query processing planner and optimizer. We assume the optimizer uses indexes on the table DI and a full scan on the table DS to access the data from the table D. Next, the *update_query_statistics(q)* operation logs the query in a log. The log may be later analyzed by the indexing schema maintenance process to acquaint performed queries and to analyze their durations.

Temporal statistics of a batch of *select*(q) operations are computed asynchronously based on typical or randomly logged queries in the *indexing_schema_maintenance()* process. It performs the queries on the table D in order to obtain their duration exploring both the index and sequential scan tables. It calculates the mean (μ_{SELECT}) and standard deviation (σ_{SELECT}) of the queries duration (T_{SELECT}) on D. Moreover, it calculates the mean value ($\mu_{SELECT\,DI}$) of the durations of accessing data in DI employing indexes and the mean value ($\mu_{SELECT\,DS}$) of the durations of the accessing data in DS using full scan. The standard deviations ($\sigma_{SELECT\,DI}$) and ($\sigma_{SELECT\,DS}$) are calculated analogously. In algorithm 3 below, it is represented by an *update_select_statistics()* operation for a selected or reduced set q' of client queries that were logged.

As previously mentioned, the responsibility for decisions concerning indexing schema maintenance is assigned to an asynchronous process. The process either runs if it receives a signal from the *insert()* operation or after a batch of updates according to the probability of necessity to perform the index maintenance. The *indexing_schema_maintenance()* process has no inputs and outputs, but it uses constraints T^{MAX}_{INSERT} and T^{MAX}_{SELECT} and the confidence factor r. In addition, it uses temporal statistics of insert operations.

```
Algorithm 3: Process indexing_schema_maintenance()
  Precondition: Signal "check RT constraint" or multiple updates

M[T_INSERT] = E[T_INSERT] + r * sigma_INSERT;
if M[T_INSERT] > T_INSERT_MAX raise warning "Insufficient Hardware";

M[T_SELECT] = E[T_SELECT] + r * sigma_SELECT;
if M[T_SELECT] > T_SELECT_MAX or E[T_SELECT-DS] > E[T_SELECT-DI]
  create new DI' as DI;
  if M[T_SELECT-DI] > T_SELECT_MAX exclude partition P1 from DI';
  if E[T_SELECT-DS] > E[T_SELECT-DI]
    create partition Pk+1 as DS;
    include partition Pk+1 into DI';
  endif;
  create indexes for DI'; -- may take a long time
  replace DI with DI'; -- must be atomic
  delete Pk+1 data from DS;
endif;
```

Algorithm 3 presents the schema maintenance operation. Expressions $E[X]$ and $M[X]$, in accordance with the notation used in previous sections (1). The first (if) condition in the algorithm checks the insert operation durations to meet the soft real-time constraint T^{MAX}_{INSERT}. If it is violated, the situation is just reported, because the ReTIN does not use any index while inserting the data, so there is no related overhead that could be reduced.

The second condition checks the temporal constraints and defines when the indexing schema operation should be performed. Until the condition is met, the

balancing of the execution time of the full scan on DS and the index data access on DI is considered to be optimal.

The indexing schema maintenance operation can be executed if one or both of the following conditions are met:

- The duration of select operations on the indexed data part are about to break the T^{MAX}_{SELECT}. In such a case, the size of the indexed table DI has to be reduced. It is done by removing partition P_1 containing the least recent data from DI.
- Unindexed selects last longer than the indexed ones. In such a case, a new partition P_{k+1} containing data from table DS is added to DI.

After that, the data in DI is (re-)indexed using user-defined indexing operations. The (re-)indexing routine is supposed to work asynchronously in time and space. Thus a new virtual table DI' is created and the index(es) are built on this table, which can take considerably long time. The re-indexing process is accomplished by the atomic replacement of the deprecated logical index table DI with DI'.

Note: For the sake of simplicity, we have simplified the description of/ algorithm 3. For instance, we consider the range of partitions of DI from 1 *to k no matter of the fact that some can be excluded and other added. A technique similar to shadow paging known from database systems can also be used.*

5 ReTIN implementation

Our open-source implementation of ReTIN is based on DBMS PosgreSQL. Java and PL/pgSQL were used as programming languages.

There are some important issues that have to be taken into account in implementation of ReTIN. Most of them concern the DMBS, in particular:

1. *Inheritance*
   Because ReTIN indexing schema consists of the hierarchy of tables D, DS and DI, this support is necessary. Either views or object-relational extensions introduced in SQL: 1999 (inheritance) can be used. The latter seems to be more appropriate.
2. *Partitioning*
   In ReTIN, table DI is composed of partitions P_i [20]. Partitions can also be implemented as individual partition tables that are inherited by tables DI and D. We have used this approach.
3. *Triggers*
   Because data elements from the stream are inserted into the virtual table D, but are physically stored in DS, the DBMS must provide a mechanism that makes it possible (`INSTEAD OF` or `BEFORE`).
4. *Notification*
   ReTIN assumes the existence of a background process that checks soft real-time constraints and is responsible for indexing schema maintenance. This

process receives asynchronous messages that trigger these activities. Some DBMSs support the waking up of such a process, for example dbms_alert in Oracle or listen/notify concept in PostgreSQL [22, 20]. If the DBMS does not support this functionality, it is necessary to set an appropriate sleep period for the process. It can be derived from the frequency of insertions. However, notification is the primary instrument for implementing continuous queries.

5. *Indexing*
   The DBMS must provide appropriate indexing methods with respect to data types of data stream elements or their subelements that should be indexed. PostgreSQL provides two families of indexes on complex data types – GiN and GiST [22].
6. *Query planner and optimizer*
   Because ReTIN indexing schema combines full scan and indexing as access methods to data, it relies on its quality or the existence of hints that allow one to enforce a specific data access method.

There are some other important issues, such as concurrency control, query logging, query selection for temporal statistics collection, duration measurement etc., which are not discussed here.

6 Experimental Results

We used a dataset of meteorological observations *Global Surface Summary of Day Data (GSOD)* [17] for experiments. GSOD is a product archived at the *National Climatic Data Center (NCDC)* to make a wide range of climatic data available to researchers and the public. The on-line data files cover a time period from 1929. They contain data from more than 9000 stations. Each record contains the global summary of day data containing 18 surface meteorological means and maximums and other characteristics such as temperature, dew point, sea level pressure, visibility, wind speed together with precipitation amount, snow depth and indicators for occurrence of fog, rain or drizzle, snow or ice pellets, hail, thunder, and a tornado/funnel cloud summary. Because of its temporal characteristics, the data can be processed as a data stream. Although this is not typical data with critical real-time constrains, it is appropriate for our experiments.

The GSOD data were represented by an array of integers. Float values in the dataset were rescaled and converted into integers due to performance and memory saving reasons. Then table D, in which the data is stored in the database, has a schema $D(t : \texttt{TIMESTAMP}, d : \texttt{ARRAY OF INTEGER})$. ReTIN implementation is based on the *PostgreSQL 8.4* database management system and the *Generalized Inverted Index (GIN)* index [22] recommended for the indexing of arrays. It ran on a server with *2x AMD Opteron 2435 (6 cores, 2.6GHz), 64GB RAM and 2.5TB RAID-6*.

The goal of the first experiment was to show a dependency of the execution times of insert and select operations on the amount of data in the database for given constraints T^{MAX}_{INSERT} and T^{MAX}_{SELECT}. There were three approaches to

access data used: unindexed data, GIN indexed data and by means of the ReTIN indexing schema. The experiment was evaluated on 500, 000 records of 1950's GSOD data. The size of table D with 500, 000 records was about 240 MB, including the GIN index structure.

The methodology of the experiment was as follows: Records were sequentially inserted into the data table. Average and maximum execution times of insertions were measured for batches of 100 insertions. Average and maximum durations of queries were measured by a set of queries for batches of 1000 insertions. The same set of queries with the *contains* array operator was used in the batches. A result set of queries contained 5% to 50% of all records in the table. Execution times were measured by stored functions on the database server. They are equivalent to the `EXPLAIN ANALYZE` query. During this experiment we set both T^{MAX}_{INSERT} and T^{MAX}_{SELECT} constraints to $0.3s$ The experiment was repeated three times to avoid random noise.

Figure 2 shows dependency of average and maximum execution times on the size of table D without any index on data column d. This approach was very fast for insertion but execution times of queries increased linearly with the number of records. The time constraint 0.3s was permanently broken for more than 460000 inserted records in the table. This corresponds ito our expectation because of the full scan access to data.

Figure 3 shows the same situation when the GIN index on data column d was created. The problem of this approach is showed in figure 3(a). There are many insertion execution time peaks between 110000 and 180000 records. The cause of this phenomenon is the necessity to re-build the index structure. The maximum execution time of queries exceeded the value $0.3s$ of the T^{MAX}_{SELECT}constraint many times.

The results for ReTIN are presented in figure 4. They show the benefit of the proposed indexing schema. Maximum insertion execution times in figure 4(a) were below the value $0.3s$ of the T^{MAX}_{INSERT} constraint. Execution times were slower only when the indexing schema maintenance operation was performed. Execution times of queries shown in figure 4(b) demonstrate the benefit of our approach. The ReTIN indexing schema combines a stable time of insertion with efficient query processing.

Figure 5 provides a more detailed view of the behavior of the ReTIN with respect to tables DS and DI. Figure 5(a) shows the decomposition of the average execution times from Figure 4(b) to the times spent by partial queries accessing data in the tables DS and DI. Figure 5(b) shows the dependency of the size of the DS table on the number of executions of the indexing schema maintenance operation from the beginning of the stream. At the beginning, the full-scan search is very fast but grows linearly. As the number of records grow, more data is searched using the index and the total execution times grows logarithmically. It proved our hypotheses stated in section 2.

The second experiment was focused on the concurrency properties of ReTIN. We simulated concurrent transactions by two groups of clients. The first group generated transactions containing an insert operation, while the other group

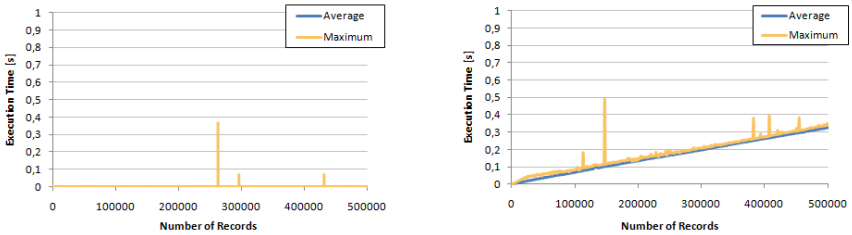

Fig. 2. a) Average and maximum execution time of insertions and b) queries on the database table using sequence scan.

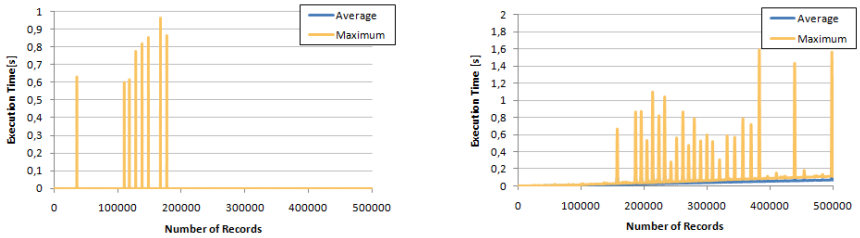

Fig. 3. a) Average and maximum execution times of insertions and b) queries on the database with the GIN index on column d.

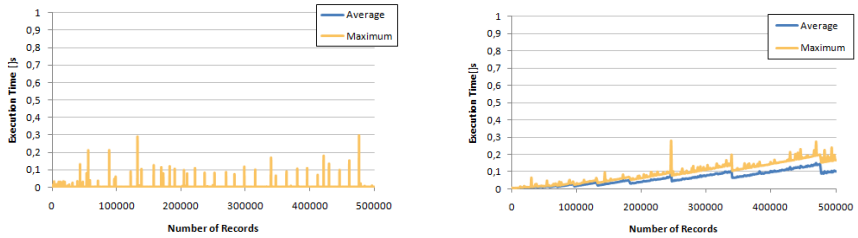

Fig. 4. a) Average and maximum execution times of insertions and b) queries on table D of the ReTIN index schema.

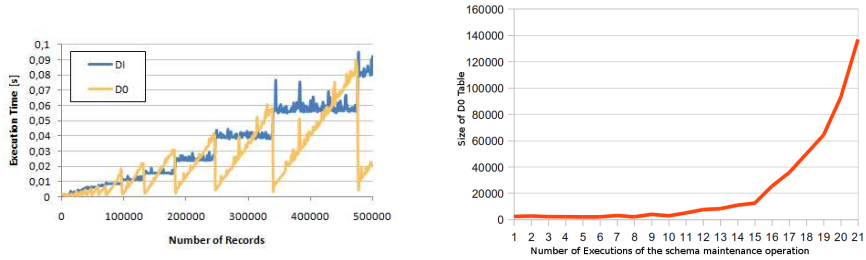

Fig. 5. a) Average times of queries on DS and DI tables based on tuples count and b) appropriate DS size each executions of the index schema maintenance operation.

generated transactions containing queries. There were several threads running in parallel. We used the same set of data as in the first experiment. The constraints T^{MAX}_{INSERT} and T^{MAX}_{SELECT} were set to $0.3s$. We evaluated the dependency of constraint violations on the number of parallel queries and insertions that were performed three times a second.

Table 1 shows the main results of the experiment. We expected a maximum violation rate of about 0.3 % by setting the confidence factor $r = 3\sigma$. The experiment showed that the ReTIN system limit for T^{MAX}_{INSERT} on this hardware is about 150 transactions a second – 25 parallel insertions and 25 parallel queries 3 times a second (the row picked in bold in Table 1). This value also determines the maximum size of the data stream sliding window. For more transactions, the number of violations is greater than 0.3 % and the window size would have to be reduced to about $200,000$ items for 300 transactions a second in our case. If we compare it to the same experiment but with data stored only in a table with a GIN index – see the last row in Table 1, which corresponds to 150 transactions a second, we can see the benefit of ReTIN. It fails about two times less for querying and 10 times less for the data insertion.

Table 1. The number of real-time constraint violations in the concurrent database access.

Threads	Queries failed	Inserts failed
ReTIN indexing schema		
10+10	0.00 %	0.03 %
25+25	**0.52 %**	**0.31 %**
50+50	1.13 %	0.54 %
GIN indexing		
25+25	**1.23 %**	**3.84 %**

7 Conclusions

We have proposed, implemented and evaluated a soft real-time indexing schema called ReTIN. It makes it possible to effectively index a portion of a data stream stored in a database and to meet real-time constraints for insert and select operations with some confidence. It combines storing the most recent data unindexed and indexing less recent data. The former is advantageous from an insert operation point of view, but results in a full scan access for select operations. The latter provides more effective access to data. The indexing schema maintenance operation that optimizes the balance between unindexed and indexed data with respect to the real-time constraints is performed asynchronously to clients' insert and select operations.

The experimental evaluation showed advantages in comparison with indexing all data stored in the database. We used the efficient PostgreSQL's GIN index both in our ReTIN implementation and as a competitive access method in

the experiments. They showed that ReTIN behaves appropriately for insertion and selection operations on both indexed and unindexed data in the database. Moreover, it changes its behavior automatically according to the system load – it changes the width of the sliding window that defines the number of data stream elements stored in the database.

ReTIN does not specify the type of index used for indexing. Current DBMSs usually provide several types, some of them are suitable for indexing complex data, similarity search etc., for example a KD-tree or R-tree. Their disadvantage often is a high overhead of insertions. ReTIN can cushion this problem and allow for the indexing of data streams containing complex data (e.g. spatio-temporal ones).

In the future, we intend to continue the experimental evaluation of ReTIN with other types of indexes. In addition, we will focus on the ReTIN deployment and optimization for our surveillance network system SUNAR and a network security project, for which it was originally designed. The schema performance could be moreover improved by using differential indexing techniques [23] or another use of the random access memory in the database backend.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR. pp. 277–289 (2005)
2. Adelberg, B., Garcia-Molina, H., Kao, B.: Applying update streams in a soft real-time database system. In: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data. pp. 245–256. ACM, New York, NY, USA (1995)
3. Aggarwal, C.C.: Data Streams: Models and Algorithms. Springer US (2007)
4. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab (2004), http://ilpubs.stanford.edu:8090/641/
5. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–142 (2006)
6. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 1–16. ACM, New York, NY, USA (2002)
7. Bai, Y., Thakkar, H., Wang, H., Luo, C., Zaniolo, C.: A data stream language and system designed for power and extensibility. In: CIKM. pp. 337–346 (2006)
8. Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., Zdonik, S.: Retrospective on Aurora. VLDB Journal (January 2004)
9. Bulut, A., Singh, A.K., Vitenberg, R.: Distributed data streams indexing using content-based routing paradigm. Parallel and Distributed Processing Symposium, International 1, 94 (2005)

10. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring streams - a new class of data management applications. In: VLDB. pp. 215–226 (2002)
11. Chiueh, T., Huang, L.: Efficient real-time index updates in text retrieval systems (1999), technical report TR-66, SUNY at Stony Brook
12. Chmelar, P., Lanik, A., Mlich, J.: Sunar: Surveillance network augmented by retrieval. In: Advanced Concepts for Intelligent Vision Systems. p. 12. Springer, Sydney, Australia (2010)
13. Feng, J., Wang, Y., Yao, J., Watanabe, T.: Multi-granularity aggregation index for data stream. In: CW '08: Proceedings of the 2008 International Conference on Cyberworlds. pp. 767–771. IEEE Computer Society, Washington, DC, USA (2008)
14. Hsieh, M.J., Chen, M.S., Yu, P.S.: Approximate query processing in cube streams. IEEE Transactions on Knowledge and Data Engineering 19, 1557–1570 (2007)
15. Krämer, J., Seeger, B.: Semantics and implementation of continuous sliding window queries over data streams. ACM Trans. Database Syst. 34(1), 1–49 (2009)
16. Kuo, T.W., Lam, K.Y.: Real-time Database Systems: An Overview of System Characteristics and Issues, pp. 3–8. Springer US (2002)
17. NASA Official: Global Surface Summary of the Day - GSOD. [online] (2010), http://gcmd.nasa.gov/records/GCMD_gov.
noaa.ncdc.C00516.html
18. No, G., Servigne, S., Laurini, R.: The Po-tree: a Real-time Spatiotemporal Data Indexing Structure, pp. 259–270. Springer Berlin Heidelberg (2005)
19. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The log-structured merge-tree (lsm-tree). Acta Inf. 33(4), 351–385 (1996)
20. Oracle: Documentation library. [online] (2010), http://www.oracle.com/pls/db112/homepage
21. Park, J., Hong, B., Ban, C.: A continuous query index for processing queries on rfid data stream. In: RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 138–145. IEEE Computer Society, Washington, DC, USA (2007)
22. PostgreSQL Global Development Group: Postgresql 8.4.4 documentation. [online] (2010), http://www.postgresql.org/docs/8.4/static/index.html
23. Severance, D.G., Lohman, G.M.: Differential files: their application to the maintenance of large databases. ACM Trans. Database Syst. 1, 256–267 (September 1976), http://doi.acm.org/10.1145/320473.320484
24. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying fit: efficient load shedding techniques for distributed stream processing. In: Proceedings of the 33rd international conference on Very large data bases. pp. 159–170. VLDB '07, VLDB Endowment (2007), http://portal.acm.org/citation.cfm?id=1325851.1325873
25. Zhu, Q., Dunkel, B., Soparkar, N., Chen, S., Schiefer, B., Lai, T.: A piggyback method to collect statistics for query optimization in database management systems. In: CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research. p. 25. IBM Press (1998)