Streaming Tables: Native Support to Streaming Data in DBMSs

Luca Carafoli, Federica Mandreoli, Riccardo Martoglia, and Wilma Penzo

***Abstract*—Data Stream Management Systems (DSMSs) are conceived for running continuous queries (CQs) on the most recently streamed data. This model does not completely fit the needs of several modern data-intensive applications that require to manage recent/historical/static data and execute both CQs and OTQs joining such data. In order to cope with these new needs, some DSMSs have moved towards the integration of DBMS functionalities to augment their capabilities.**

In this paper we adopt the opposite perspective and we lay the groundwork for extending DBMSs to natively support streaming facilities. To this end, we introduce a new kind of table, the *streaming table*, as a persistent structure where streaming data enters and remains stored for a long period, ideally forever. Streaming tables feature a novel access paradigm: continuous writes and one-time as well as continuous reads. We present a streaming table implementation and two novel types of indices that efficiently support both update and scan high rates. A detailed experimental evaluation shows the effectiveness of the proposed technology.

***Index Terms*—Database design, modeling and management, Spatial/Temporal databases, Data streams, DBMS, Continuous queries.**

I. Introduction

In order to cope with large volumes of continuously streaming data, Data Stream Management Systems (DSMSs) have been introduced [1], [2]. Unlike traditional Data Base Management Systems (DBMSs) that run one-time queries (OTQs) over static data, these systems natively support continuous queries (CQs) over data streams according to windows where only the most recent data is retained. Examples of applications that adhere to these principles include publish-subscribe [3], sensor-data analysis, and financial trading systems, online gaming and geospatial services [4].

This model does not completely fit the information needs of several modern data-intensive applications that involve not only live data but also relatively past as well as historical (i.e. past streamed) information and static data [5], [6], [7], [8], [9], [10]. This is the case, for instance, of monitoring applications [5], [10] (e.g., air quality monitoring, Intelligent Transportation Systems - ITSs), military applications (e.g., platoon tracking), network applications (e.g., intrusion detection), and modern data analysis applications [8], [9].

These kinds of applications definitely need a system that is able to manage recent/historical/static data and execute both CQs and OTQs joining such data in an efficient way. For instance, in the ITS scenario we experienced in the PEGASUS project [11], vehicle reports are used both for real time traffic analysis and for long-term purposes, where they contribute to the historical information needed for data analysis. In order to fulfil the application requirements, different kinds of CQs run joining the continuous flow of vehicle reports with static tables, for instance for a continuous monitoring of critical road segments. At the same time, OTQs run over relatively recent vehicle reports, for instance to reconstruct the dynamics of an accident or to compute different insurance policy costs.

a) Current Approaches: In order to fulfill these new needs, some DSMSs have moved towards integrating DBMS functionalities within their own architectures [12], [13], [6], [7], [14]. These approaches have the main drawback of requiring to redesign from scratch a core of well-established DBMS techniques that can not be reused as such in a DSMS architecture. Further, in these solutions, historical and static data necessarily have to be converted into streams before being queried, thus producing a heavy overhead for the system.

Other works [5], [9], [10] adopt two-layered solutions where DSMS and DBMS are maintained as independent systems, each one devoted to the management of data they have been conceived for (live and historical/static data, respectively). The pure combination of these systems, each designed for specific and opposite goals, does not solve the DBMS inefficiencies in storing/retrieving data at the rate a DSMS requires. To partially reduce this problem, [15] limits the cardinality of streaming data to be stored.

Another direction deals with extending DBMSs with streaming data management capabilities [16], [17], [8], [18], [19] or with proposing new database architectures to guarantee good performance for hybrid workloads (e.g., [20], [21], [22]). As our proposal relates to this direction, a comparison with these works can be found in Section VIII.

b) Our Proposal: This paper advocates that a native support of DSMS functionalities into the DBMS is key to meeting the design and performance needs of several hybrid data-intensive applications. The DBMS is a consolidated and universally adopted technology to manage static data in tables and to run OTQs. For their execution, the DBMS has full-fledged procedures at its disposal that ensure high level of performance. However, we are aware that extending the DBMS with streaming data support is a really ambitious and long-term objective. On the other hand, a wide variety of data types such as spatial data, temporal data and XML data have been successfully integrated in the past. These positive examples suggest the path to follow to reach this goal. The outcome would be an empowered DBMS that is inherently familiar to

L. Carafoli, R. Martoglia and F. Mandreoli are with the FIM Department, University of Modena and Reggio Emilia, Italy
E-mail: name.surname@unimore.it.
W. Penzo is with the DISI, University of Bologna, Italy
E-mail: wilma.penzo@unibo.it.

experienced database developers while providing an "intuitive" yet efficient way to manage applications accessing streaming data too.

The main contribution of this paper towards this objective is to promote streaming data to "first-class citizens" of DBMSs by means of the introduction of a new kind of table, the *streaming table*, as a persistent structure where streaming data enters and remains stored for a long period, ideally forever. This straightforward solution features two main benefits.

First, streaming tables nicely fit into the relational model. The underlying principle is that historical data and streaming data are not intrinsically different, because historical data is past streamed data, whereas they differ in their management. Unlike [8] where these kinds of data are managed separately and only historical data are persistent, both historical and streaming data are in charge of streaming tables and their management is completely transparent to users. In this way, any streaming table can be straightforwardly involved in OTQs, where it participates as (historical) temporal table [23], as well as in CQs, where only its most recent tuples are considered through sliding window specifications. In both cases, streaming tables can be used in queries together with "standard" relational tables.[1] In order to speed up both kinds of queries, users are allowed to create indices on streaming table attributes, while a temporal index is always maintained on tuple timestamps. In this way, we support the whole range of queries featuring hybrid applications without requiring database users to overhaul their thinking.

Second, streaming tables nicely fit into the DBMS architecture. To implement streaming tables means to apply the DBMS loosely-coupled design principle to streaming data, as also advocated in [6]. Traditional DBMS architectures indeed found on a clear separation between the Transactional Storage Manager (TSM) and the Query Processor (QP). This design principle has the advantage that modifications on one component are transparent to the others and represents the basis for many performance optimizations. Therefore, we aim at a seamless implementation of streaming tables into the DBMS core that exploits the considerable amount of already available query processing and data management techniques for efficient purposes.

The main issue we must address to this end is that DBMSs traditionally adopt a store-first-query-later approach that does not compel with the two complementary workload characteristics of streaming tables: massive data rates and low query answering latency. As a solution to this problem, we propose a streaming table implementation at the TSM level which offers very good performance for a novel access paradigm: continuous writes and one-time as well as continuous reads. Specifically, unlike standard relational tables, streaming tables promptly answer to continuous reads over data that is continuously written and then discarded when they possibly expire. Unlike streams, streaming tables support one-time reads on current and historical data. Also, the unconventional workload supported by streaming tables sharply differs from other focused workloads recently considered for instance by NoSQL databases [24], which are built to support efficient update- and lookup-intensive online transaction processing (OLTP). As far as we know, this is the first approach that considers continuous and SQL query-intensive workloads on write-intensive OLTP.

[1]Hereinafter, we will use the term "standard" table to denote any table that is not a streaming table.

c) Paper Contributions: In summary, this paper moves the first fundamental step towards a native support to streaming data in DBMSs by making the following contributions:

- it extends the relational model with the notion of streaming table (Sect. III);
- it illustrates the main potentialities of designing and managing hybrid data-intensive applications in a DBMS extended with streaming tables and continuous query support through the reference scenario (Sects. II and IV);
- it presents the streaming table access interface bridging between QP and TSM and it sketches its use in query plans at the QP level (Sect. V);
- it introduces at the TSM level an efficient storage method for streaming tables and two novel types of indices that accelerate the execution of both OTQs and CQs on streaming tables (Sect. VI);
- it presents a comprehensive experimental evaluation showing the effectiveness of the proposed methods, also in comparison with DSMSs and standard and NoSQL DBMSs (Sect. VII).

Finally, in Section VIII we discuss literature, we conclude, and we outline future research.

II. The Reference Scenario

Our reference scenario is in the field of ITSs and stems from the experience we gained in the management of ITS data for the PEGASUS project [11], that aimed at a sustainable and safe management of people and vehicle flows in urban and peri-urban areas. In PEGASUS the main source of information comes from On Board Units that transmit vehicle reports to a Central Unit and can be integrated with urban information to answer information needs motivated by objectives of safety, urban mobility and planning, and smart navigation. Most of these requests went beyond the classical CQ-over-data-streams querying model of DSMSs and required the execution of CQs as well as OTQs over recent and historical vehicle reports and static data. Samples of requests of these kinds are the following:

OTQ-1 Reconstruct the dynamics of an accident from the trajectories of a selected list of vehicles, the ones involved in the accident, in the accident period;

OTQ-2 Retrieve detailed information of the vehicles that have passed near a given segment in a given time period, for instance to search for possible witnesses for an accident;

CQ-3 Real-time monitoring of the statistics of only those segments that use to be the most congested, according to a relatively static list of "critical" segments.

To evaluate the impact of the introduction of streaming tables in our reference scenario we extended the Linear Road benchmark [25] that has been endorsed by major DSMSs like [13] and [12] to test efficiency in ITS scenarios. The benchmark simulates an adaptive real-time toll system where a multitude of cars move on a virtual highway and the vehicle tolls are computed in real-time in order to regulate vehicular traffic. In particular, the highway has multiple expressways, each made up of lanes and divided into segments; vehicles pay a toll if and only if they drive in a congested segment, i.e. a segment where the average speed of all vehicles has been below a given threshold in the last 5 minutes.

Traffic is simulated through an input stream of car position reports. The server elaborates the information received from the vehicles to compute tolls and transmits them back to vehicles.

The benchmark includes four types of requests that must be satisfied in a strict response time deadline of 5 seconds: accident notification, toll notification, account balance and daily expenditure. In order to respond to the first three requests the server must continuously compute the answer to multiple CQs over the data stream, including accident detection, segment crossing detection and segment statistics. The last request, instead, is satisfied through an OTQ involving a static table.

Linear Road is a benchmark for standard DSMSs where vehicle reports enter, are used to solve the continuous requests and then discarded after 5 minutes. We extended the Linear Road querying model in two directions. First, we maintain the history of vehicle reports; second we consider information needs that translate into CQs as well as OTQs over recent and historical vehicle reports and static data. In this way, we can configure evaluation benchmarks that include queries like those above.

III. Adding streaming tables to the relational model

In this Section we extend the relational model to include the notion of streaming table and the semantics of OTQs and CQs when streaming tables are involved.

A. Streaming table definition

A streaming table is a relational table where streaming data enters and turns historical by remaining stored for a long period, ideally forever.

Hereafter, we adopt the definition of continuous data stream (or simply stream) provided in [13], i.e. a potentially infinite stream of timestamped relational tuples having a fixed schema. For timestamps we assume a discrete, ordered time domain $\mathcal{T} = \{0, 1, 2, 3 \ldots, now\}$ of time instants or granules where 0 stands for the earliest time instant while now is the current timestamp. For time-interval (duration of time), instead, we do not specify restrictions on its domain $\mathcal{I}$ but we assume it is made up of computable periods of execution time, including the unbound value ∞. As far as the relational model is concerned, we adopt the standard notation: if R is a relation with schema $R(A_1, \ldots, A_n)$ and t is a tuple from R then $t(A_i)$ denotes the value of A_i in t. Moreover, given any time instant $\tau \in \mathcal{T}$, we denote with R^τ the content of R at time τ.

Any streaming table inherits the temporal nature of the data it stores. Specifically, it is an event table [23], i.e. a special kind of temporal table that stores events and their occurrence time for a limited time period, named historical period, and features the access paradigm detailed below.

Definition 1 (Streaming table): A streaming table S with schema $S(A_1, \ldots, A_n)$ and historical period $hp \in \mathcal{I}$ is an event table, denoted as S_{hp}, with schema $S(A_1, \ldots, A_n|T)$, where T is the implicit timestamp attribute. The content of S_{hp} at the time instant τ, i.e. S^τ_{hp}, is the set of tuples such that the timestamp of each tuple t satisfies $t(T) \geq max(\tau - hp, 0)$.

For ease of notation, in the following, whenever possible, we will use S in place of S_{hp}.

Definition 2 (Streaming table access paradigm): Any streaming table S

- is subject to *continuous writes* in timestamp order, i.e. for t_1 and t_2 in S^τ, $t_1(T) < t_2(T)$ iff t_1 arrived before t_2;
- supports *one-time reads* at any time τ as a standard event table S^τ;
- supports *continuous reads* on the most recent tuples. S boundaries are specified by means of one of the following classes of sliding-window operators:
  - time-based window, denoted as `INTERVAL` i, with $i \in \mathcal{I}$;
  - tuple-based window, denoted as `ROWS` k, with $k \in \mathbb{N}$;
  - partitioned window, denoted as $A_{p_1}, \ldots, A_{p_m}$ `ROWS` k, with $A_{p_j} \in S(A_1, \ldots, A_n)$, for each $j \in [1, m]$.

Any sliding-window operator w over a streaming table S is denoted as $S[w]$ and outputs a standard table whose content depends on the read instant τ:

- $S[\texttt{INTERVAL } i]^\tau$ is the ordered set of tuples $t(A_1, \ldots, A_n)$ such that $t \in S$ and $t(T) \geq max(\tau - i, 0)$ projected on $A_1, \ldots, A_n$ in decreasing T values order;
- $S[\texttt{ROWS } k]^\tau$ is the ordered set of the top-k S tuples projected on $A_1, \ldots, A_n$ in decreasing T values order;
- $S[A_{p_1}, \ldots, A_{p_m}\ \texttt{ROWS } k]^\tau$ is the union of subsets of S tuples projected on $A_1, \ldots, A_n$ in decreasing T values order. These subsets are obtained by horizontally partitioning the streaming table based on equality of attributes $A_{p_1}, \ldots, A_{p_m}$ and then by applying the tuple-based sliding window `ROWS` k on each group.

This definition refers to append-only streaming context only. Indeed, the only type of continuous update allowed on streaming tables is insertion. This is very common in data stream management works [13], [12], [18]. We plan to relax this constraint in our future work. On the other hand, a streaming table is a special kind of relational table and as such it supports one-time replacements and deletions.

Moreover, based on this definition, tuples are continuously inserted in timestamp order. In practice, to implement such a semantics, systems cope with out-of-order and skewed inputs. Interested readers can refer to [26] for an in-depth discussion

VID	SPD	XWAY	LANE	DIR	SEG	POS	T
1	0	0	0	0	10	53320	0
2	32	0	0	1	10	53320	1
2	0	0	0	0	11	53320	3
4	0	0	0	0	11	53320	4
4	32	0	0	0	11	53320	6

(a) A streaming table

POSREPORTS[INTERVAL 3]^7

VID	SPD	XWAY	LANE	DIR	SEG	POS
4	32	0	0	0	11	53320
4	0	0	0	0	11	53320

POSREPORTS[ROWS 3]^7

VID	SPD	XWAY	LANE	DIR	SEG	POS
4	32	0	0	0	11	53320
4	0	0	0	0	11	53320
2	0	0	0	0	11	53320

POSREPORTS[VID ROWS 1]^7

VID	SPD	XWAY	LANE	DIR	SEG	POS
4	32	0	0	0	11	53320
2	0	0	0	0	11	53320
1	32	0	0	0	10	53320

(b) Sliding windows

Q(POSREPORTS^7)

1	0	0	0	0	10	53320	0
2	0	0	0	0	11	53320	3
4	0	0	0	0	11	53320	4

Q^c(POSREPORTS[INTERVAL 4])$_2^5$

4	0	0	0	0	11	53320	4
2	0	0	0	0	11	53320	4
1	0	0	0	0	10	53320	4

Q^c(POSREPORTS[INTERVAL 4])$_2^7$

4	0	0	0	0	11	53320	6
2	0	0	0	0	11	53320	6

(c) Query semantics

Fig. 1. A sample about streaming tables and query semantics

of this aspect. In this paper we assume an input manager that guarantees in-order tuple arrival.

Example 1: The stream of car position reports of the reference example can be stored in a streaming table with schema POSREPORTS(VID, SPD, XWAY, LANE, DIR, SEG, POS), where VID is the vehicle ID, SPD represents the reported speed, while the other attributes are related to the actual position of the vehicle in the reference road map. A sample of POSREPORTS storing a stream of car position reports generated by Linear Road up to time 6 is shown in Fig. 1(a). It concerns 3 vehicles identified by the VIDs 1,2, and 4 occupying lane 0 of the expressway 0.

The sliding window operators POSREPORTS[INTERVAL 3], POSREPORTS[ROWS 3], and POSREPORTS[VID ROWS 1] over the streaming table POSREPORTS denote a time-based window of 3 time instants, a tuple-based window requiring the last three position reports, and a partitioned window requiring the latest tuple for each VID, respectively. Their output at time 7 over the streaming table in Fig. 1(a) is shown in Fig. 1(b). It is worth noting that, according to the sliding windows semantics, tuples are projected on the explicit schema and presented in decreasing T values order.

B. Query semantics

Streaming tables can be involved both in OTQs and CQs. As to the former, we assume readers are familiar with the syntax and semantics of relational queries[2] and we only provide a concise yet informal semantics of OTQs.

Definition 3 (Semantics of one-time queries over streaming and standard tables): The result of a OTQ Q over n streaming tables $R_1, \ldots, R_n$, with $n \geq 1$, and $m - n$ standard tables, with $n \leq m$, $R_{n+1}, \ldots, R_m$, issued at time τ, is an event table $Q(R_1^\tau, \ldots, R_n^\tau, R_{n+1}^\tau, \ldots, R_m^\tau)$ of fixed arity with values from $R_1^\tau, \ldots, R_n^\tau, R_{n+1}^\tau, \ldots, R_m^\tau$.

It is worth noting that when a OTQ Q is evaluated at time τ, all the streaming tables referenced in Q undergo a one-time read at time τ. In other words they are dealt with as standard (event) tables and the content of any involved streaming table R_i at Q disposal is the set of tuples in R_i^τ.

As far as CQs are involved, a continuous query Q^c is a query that is issued once, and then logically runs continuously until Q^c is terminated. In line with many CQ specification syntaxes (e.g. [13]), we assume Q^c is always equipped with a *slide* parameter sl representing the query evaluation period. The latter can be either an interval or the special parameter REALTIME, that means that the query is re-evaluated as new tuples arrive.

Any streaming table R referenced in a continuous query must be equipped with a sliding-window w that specifies the boundaries of the most recent R tuples to be used for query evaluation. According the generally accepted definition of continuous query semantics [28], we define the semantics of a continuous query Q^c over streaming and standard tables to be equal to the semantics of the corresponding OTQ Q whose inputs are the current states of streaming and standard tables referenced in Q^c.

Definition 4 (Semantics of continuous queries over streaming and standard tables): The result of a continuous query Q^c with slide parameter sl over n streaming tables and sliding-windows $R_1[w_1], \ldots, R_n[w_n]$, with $n \geq 1$, and $m-n$ standard tables, with $n \leq m$, $R_{n+1}, \ldots, R_m$, at time τ, is a streaming table with historical period hp whose content at time τ corresponds to the set of timestamped data returned up until τ by executing the corresponding OTQ Q at sl granularity:

$$Q^c(R_1[w_1], \ldots, R_n[w_n], R_{n+1}, \ldots, R_m)^\tau_{hp} = \bigcup_{i=0}^{k:\tau_k \leq \tau} (Q(R_1[w_1]^{\tau_i}, \ldots, R_n[w_n]^{\tau_i}, R_{n+1}^{\tau_i}, \ldots, R_m^{\tau_i}) \times \{\tau_i\})$$

where $\tau_i = (\lceil \frac{\tau - hp}{sl} \rceil + i) \cdot sl$. The default historical period hp value is sl.

It is worth noting that the default result set of a continuous query Q^c at time τ is the result set of the last Q execution, in accordance with the sl granularity. Any historical period extension can be performed through a materialized view specification in such a way to store the result sets of consecutive Q executions in a persistent streaming table.

Example 2: Let us assume we are interested in identifying stopped cars in the streaming table POSREPORTS, i.e. tuples

[2]Interested readers can refer to [27] for an in-depth study.

with SPD=0. If an OTQ with such a selection condition is issued then we will obtain all the cars that stopped in the time interval of the recorded position reports. The output of such OTQ Q(POSREPORTS^7) over POSREPORTS issued at time 7 is the event table shown in Fig. 1(c).

If, instead, we are interested in a continuous monitoring of stopped cars we must issue a continuous query. For instance the continuous query Q^c(POSREPORTS[INTERVAL 4]) with slide parameter $sl = 2$ will deliver every 2 instants (i.e. at time 2, 4, and 6) the stopped cars in the last 4 instants. The query results at the two time instants 5 and 7, Q^c(POSREPORTS[INTERVAL 4])$_2^5$ and Q^c(POSREPORTS[INTERVAL 4])$_2^7$, are shown in Fig. 1(c).

IV. Sketch of extensions to SQL

In this Section we aim at illustrating the main benefits and potentialities of designing and managing hybrid data-intensive applications in a DBMS extended with streaming tables and continuous query support. To this end, we need to extend SQL, the declarative DBMS language database designers use to define and access data in their applications. Introducing a complete set of SQL constructs to put at designers disposal is a demanding issue that will be dealt with in our future work. In this paper we limit ourselves to outline some possible minimal extensions that SQL should undergo to effectively support our reference scenario (see Section II).

Streaming table POSREPORTS, that will store the stream of position reports, can be declared through the statement CREATE STREAMING TABLE shown in Table. I(a). It extends the standard SQL statement CREATE TABLE with the RANGE construct that states the historical period through a SQL INTERVAL literal. Moreover the standard CREATE INDEX construct can be used to build B$^+$-trees as well as hash indices on explicit streaming table attributes. Finally, the INSERT INTO command (Table I(b)) connects POSREPORTS to the Linear Road stream for continuous data insertion.

The upper part of Table I shows three CQs (c)-(e) that implement the Linear Road benchmark. They follow a syntax similar to the one introduced in the CQL continuous query language [13] and semantics introduced in Subsec. III-B. As to the syntax, note that SELECT and WHERE clauses are standard SQL expressions, the WHERE clause includes the streaming and standard tables referenced in the query, and the SAMPLE parameter defines the slide parameter. For instance CQ (c) continuously detects accidents by identifying stopped vehicles. A vehicle is considered stopped when the last four consecutive position reports have null speed. To this end, the query specifies a partitioned window on the vehicle ID attribute, VID, that outputs the last four reports for each vehicle and uses the newly added function, PREV, that returns the tuple preceding the current tuple t in the window. CQ (d), instead, computes each minute the segment statistics from the last minute traffic information and stores them in the materialized streaming view SEGSTAT, maintaining statistics for the last 5 minutes as required. Finally, CQ (e) continuously detects segment crossings. It is worth noting that in two out of the three CQs shown in the table, the slide parameter is REALTIME that means that both queries are re-evaluated whenever a new POSREPORTS tuple arrives.

Thanks to the minimal extensions to SQL depicted in the upper part of Tab. I, Linear Road would be implemented on a full-function database system in a very compact and intuitive way: 1 streaming table, 4 standard tables, 2 views, 3 CQs, and 2 stored procedures[3]. As a matter of fact, only the CQs shown in Tab. I are necessary while the rest of the benchmark can be implemented through standard SQL statements and stored procedures. Linear Road represents the portion of our reference scenario standard DSMSs usually support. However, their implementations are significantly more involved. Indeed, these systems have query languages that differ sharply in semantics and capabilities from standard SQL. Specifically, since they rely on stream-oriented data types and only a limited number of constructs are supported, a very large number of data definitions are usually required. For instance, the STREAM implementation of Linear Road[4] defines more than 35 CQL data stores. With reference to Table I, query (c) is implemented with 14 streams, while the ones in queries (a) and (b) correspond to more than 12 streams. Please also note that static data like expenditure data should enter the system as streaming data, making main memory requirements burdensome. The solution we advocate in this paper, instead, straightforwardly responds to this kind of requests since SQL was specifically conceived for this purpose.

The lower part of Table I implements requests OTQ-1, OTQ-2, and CQ-3 presented in Section II (denoted as (f)-(h), respectively). They go beyond the capabilities of standard DSMS querying models. According to the OTQ semantics presented in Section III, OTQs (f) and (g) fetch the streaming data in POSREPORTS in one-read and output standard tables. TRJ() is a custom aggregate function returning the trajectories of the list of vehicle IDs it receives as input. Similarly, the stored procedure NEAR() returns the vehicles that have passed near a given segment. In both cases, temporal predicates follow the TSQL syntax [23]. Specifically, predicate VALID extracts tuple timestamps while OVERLAPS checks whether the tuple timestamps fall in the required time intervals. Finally, CQ (h) joins a streaming table with the standard table CRITICALSEGS(SEG) storing "critical" segments.

V. Dealing with streaming tables in a DBMS

In this paper, we build the bases for streaming table management in a DBMS. To this end, we propose a streaming table implementation that includes the algorithms and the data structures for organizing and accessing streaming data.

According to the DBMS architectures, streaming table implementation affects the TSM and is accessible through an access interface. The latter would enable the QP to perform access and manipulation calls on streaming table data in different ways during query processing by means of basic operations to be used for query plan generation.

[3]The full version is available at http://www.isgroup.unimo.it/files/LRQueries.pdf

[4]Downloadable together with the Stream server source code from http://infolab.stanford.edu/stream/code/

a
```
CREATE STREAMING TABLE POSREPORTS ( VID INT, SPD
      INT, XWAY INT, LANE INT, DIR INT, SEG INT, POS INT)
RANGE INTERVAL '2' WEEK

CREATE INDEX IDX_VID ON POSREPORTS (VID)
```

b
```
INSERT INTO POSREPORTS
VALUES FROM 'LINEARROAD.STR'
```

c
```
SELECT P1.VID, P1.XWAY, P1.LANE, P1.DIR, P1.POS
FROM POSREPORTS [WINDOW VID ROWS 4] P1,
     POSREPORTS [WINDOW VID ROWS 4] P2,
     POSREPORTS [WINDOW VID ROWS 4] P3,
     POSREPORTS [WINDOW VID ROWS 4] P4
WHERE P1.VID = P2.VID AND P2.VID = P3.VID
AND P3.VID = P4.VID
AND P2 = PREV(P1) AND P3 = PREV(P2)
AND P4 = PREV(P3)
AND P1.SPD = 0 AND P2.SPD = 0 AND P3.SPD=0
AND P4.SPD = 0
SAMPLE REALTIME
```

d
```
CREATE VIEW SEGSTAT (XWAY, DIR, SEG, CNT, AVGSPD)
RANGE INTERVAL '5' MINUTE
AS (   SELECT XWAY, DIR, SEG, COUNT(*), AVG(SPD)
       FROM POSREPORTS [ WINDOW INTERVAL '1' MINUTE ]
       GROUP BY SEG
       SAMPLE INTERVAL '1' MINUTE)
```

e
```
SELECT P1.VID, P1.XWAY, P1.DIR, P1.SEG
FROM POSREPORTS [ WINDOW VID ROWS 2 ] P1 ,
     POSREPORTS [ WINDOW VID ROWS 2 ] P2
WHERE P1.VID = P2.VID
AND P2 = PREV(P1)
AND P1.SEG != P2.SEG
SAMPLE REALTIME
```

f
```
SELECT VID, TRJ(*)
FROM POSREPORTS P1, INVOLVEDVIDS IV
WHERE P1.VID = IV.VID
AND VALID (P1) OVERLAPS (NOW() – INTERVAL '15'
      MINUTE, NOW()– INTERVAL '10' MINUTE)
GROUP BY VID
```

g
```
SELECT XWAY, DIR, SEG, VID,POS
FROM POSREPORTS P1
WHERE NEAR(SEG , XX)
AND VALID (P1) OVERLAPS (NOW()– INTERVAL '20'
      MINUTE, NOW() – INTERVAL '10' MINUTE)
```

h
```
SELECT *
FROM SEGSTAT S [ WINDOW INTERVAL '2' MINUTE ],
     CRITICALSEGS CS
WHERE S.SEG = CS.SEG
ORDER BY S.CNT, AVGSPD DESC
SAMPLE INTERVAL '1' MINUTE
```

TABLE I
A SET OF QUERY SAMPLES FOR THE LINEAR ROAD APPLICATION

Operator	Call parameters
Data manipulation interface	
`CInsert`	`table, stream`
Data access interface	
`CScan`	
`CIndexScan`	`call_id, table,window,ixid,pred,sample`
`CSeqScan`	`call_id, table,window,pred,sample`
`Resume`	`call_id`
`Kill`	`call_id`

TABLE II
THE NEWLY ADDED OPERATORS FOR STREAMING TABLE MANIPULATION AND ACCESS

Fig. 2. Continuous operator call samples (a) and a query plan sample (b)

The base streaming table access interface includes the continuous operators listed in Table II and traditional operators that apply to streaming tables too[5].

`CInsert` implements continuous writes. It is invoked when the DBMS processes an `INSERT INTO` query in order to connect the streaming table `table` to the source stream `stream`.

The `CScan` interface refers to the way data items in the window `window` are accessed by any continuous read. It is implemented through the `CIndexScan` and the `CSeqScan` operators; the former leverages on the index with OID `ixid` while the latter sequentially scans the items in the window. Finally, `pred` is a DNF formula of simple selection predicates of the form `column operator constant`.

When a continuous query is submitted to the DBMS, `CIndexScan` and `CSeqScan` operator calls specify the access pattern on the involved streaming tables. Each call is issued to the TSM only once while updates must be delivered at the time rate that is specified in the `SAMPLE` clause (see Table I). To this end, we support two types of access models [6]: the pull and the push model. In the first case, data requests are driven by the QP that communicates to the TSM the call identifier `call_id` through the `Resume` operator. In the last case, instead, it is the TSM that notifies the QP about new available results. This model is used for instance in `SAMPLE REALTIME` queries. Therefore, `sample` is an optional parameter stating that the push model has been selected and its value represents the delivery frequency. In both cases, the `Kill` operator is invoked to stop the process.

One-time reads on streaming tables are executed through traditional `Scan` calls. Therefore when one or more streaming tables are involved in an OTQ, the execution plan is a standard one. What changes is the specific implementation for streaming tables.

Example 3: With reference to the queries shown in Tab. I, four sample calls, one for each type, are shown in Fig. 2(a) (windows are expressed in seconds). As to continuous reads, a sequential scan on the `POSREPORTS` streaming table is considered for query (d) whereas the index scan for query (h) relies on a previously defined index on the `SEG` attribute of the `SEGSTAT` streaming table. Finally, `pred` is the translation of the `WHERE` clause shown in query (g) as it can be executed at TSM level. Finally, a possible plan for the query in Table I(h) is shown in Fig. 2(b).

VI. STREAMING TABLE IMPLEMENTATION

In this Section we propose a technology for storing and indexing streaming tables at the TSM level and we give execution details of the TSM interface.

Streaming tables are subject to workloads featuring high write rate together with concurrent continuous queries with real time requirements and one-time queries often involving temporal predicates. Unfortunately, standard relational table implementations do not achieve good performance for this kind of workloads that is novel for traditional database systems.

To efficiently cope with this new kind of workload, we present an implementation for streaming tables that borrows

[5]In the absence of a standard TSM interface, we adopt the PostgresSQL operator names (http://doxygen.postgresql.org) and we introduce the continuous counterparts by prefixing `C` to the operator names, whenever possible.

the main data management principles from DBMSs and DSMSs:

- it uses main-memory storage to achieve good continuous read performance;
- it uses secondary-memory storage to store large volumes of historical tuples[6];
- it relies on indexing to provide a fast access path to tuples based on predicates;
- it employs efficient scan algorithms.

To this end, various storage methods and indices have been proposed in the past both in DSMS contexts (e.g. [6], [30]) and in standard DBMS's [31]. Nevertheless, we cannot merely juxtapose them because in our case the two memory levels are strictly interconnected by highly dynamic data transfer. Large volumes of data, continuously arriving over time, stay for a period in main memory, then gradually flow to the lower memory level, and are finally removed when the historical period slides. On the contrary, in DSMSs expired and consumed data exit the system, while DBMSs have not been conceived for frequent updates.

The technology for streaming table we propose relies on a storage structure that ensures tuple access at both memory levels while keeping low main memory consumption.

On top, we propose two new kinds of indices, one for the efficient execution of temporal predicates and the other for attribute-valued predicates, that efficiently support high update and scan rates at both memory levels, without the delays of periodic batch processing.

A. Storage structure for streaming tables and notation

Once a `CInsert` operator call is issued to the TSM, the involved streaming table starts populating with data pulled from the specified stream.

At any time, the current part of a streaming table, i.e. those items that are neither expired nor yet consumed, is stored into a *circular linked list of blocks*, while expired and consumed items are stored in disk blocks (Fig. 3(a)). Each block stores a fixed number of tuples; the tail block stores the newest tuples in the list while the head block the oldest ones, and the tuples inside are in chronological order (newest tuple at the tail). Concurrent accesses on lists are dealt with by means of a locking technique adopting single tuples as lock granularity and implements read locks (shared) and write lock (exclusive). Therefore, we exploit a state-of-the-art storage structure for stream management (see e.g. [6], [30]) with the main difference that the block size is a multiple of the database page size.

Fig. 3(a) depicts the notation for circular linked list we will adopt hereinafter: L denotes a list, b a list block and t a streaming tuple; $L.getFirst()$ and $L.getLast()$ return the head and the tail block of the list, respectively. Within each block b, $b.getFirst()$, $b.getLast()$, and $b.getNext()$ return the first and the last tuple in block b and the block next to b, respectively; $t.time()$ returns the t timestamp.

As to data transfer from main to secondary memory, we adopt a lazy disk-write approach that consists in marking whole block contents thus limiting the I/O costs of the item write flow. Outdated blocks can then be reused by replacing the old items with the new incoming ones.

Specifically, whenever a new item arrives and the current block, $L.getLast()$, is full, we have two alternatives. If the list is in a stable state, i.e. the tuples it contains enable the processing of all the continuous operator calls that are currently active on the table, then the content of the most aged block $L.getFirst()$ is ready for disk write and the block is re-used as the new current block, i.e. $L.getLast() = L.getFirst()$. Otherwise, a new block is added to the list and becomes the new current one $L.getLast()$.

The approach is correct as we follow the usual stream-based semantics, i.e. continuous operators "look" at the input only once. Moreover, it is more performant than the one adopted for instance in [6] where single items are marked as consumed and/or expired. We employ this finer-grain solution for one-time deletions only: Deleted items are marked in the blocks so that any subsequent request for them will fail. Then marked items are physically removed lazily, when their blocks are forced to disk.

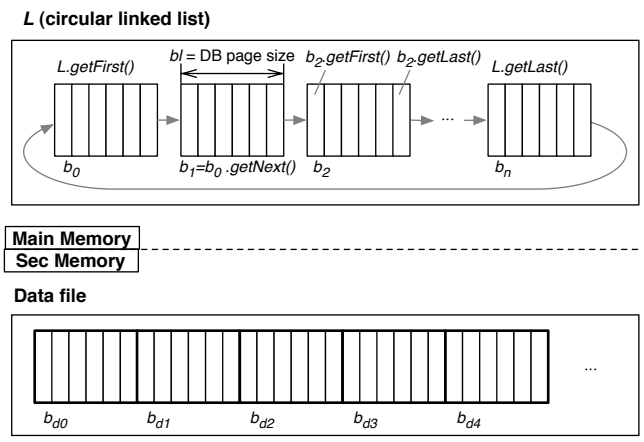
B. The interval-based index

Streaming tables store temporal data. Each streaming table is therefore always equipped with a new kind of temporal index, the interval-based index, that accelerates temporal predicate evaluation as well as temporal window computation.

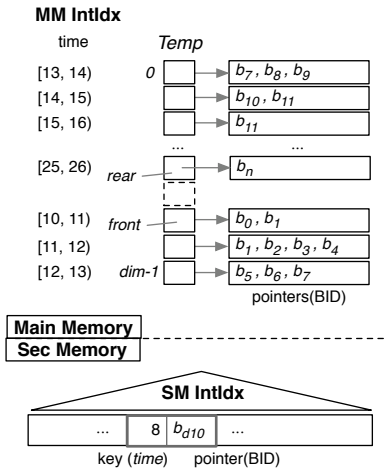
Its main memory component is a circular dynamic array that efficiently supports time-based accesses to the blocks of the circular linked list L. The array $MMIntIdx$ of dimension dim covers the list time period $[start, now]$ where $start = L.getFirst().getFirst().time()$. Each entry $MMIntIdx[i]$, for $i \in [0, dim - 1]$, corresponds to a time interval of fixed length ti and points to the queue of block identifiers (BIDs) covering the interval. More precisely, with reference to Fig. 3(b), the front element $MMIntIdx[front]$ references those blocks in L that contain items whose temporal pertinence is in the range $[start, start + ti)$, the next one, $MMIntIdx[(front + 1) \bmod dim]$, covers the time interval $[start + ti, start + 2 * ti)$, while the rear element is about $[start + ((dim + rear - front) \bmod dim) * ti, now)$. In this context, time-based accesses always amount to random accesses in a circular dynamic array.

Example 4: Let us assume that the timestamps represented in Fig. 3(b) are minutes, then the Table I(f) query predicate is solved by selecting the BIDs pointed by the array elements between $MMIntIdx[(front + (now - 15 - start) \mod ti) \bmod dim]$ and $MM[(front + (now - 10 - start) \mod ti) \bmod dim]$ where $ti = 1$, $start = 10$, and $now = 26$.
It is worth noting that the number of returned BIDs is proportional to the average number of BIDs per queue that is strictly related to the ti value. Therefore, ti is an important parameter to be set: the smaller ti, the more selective the

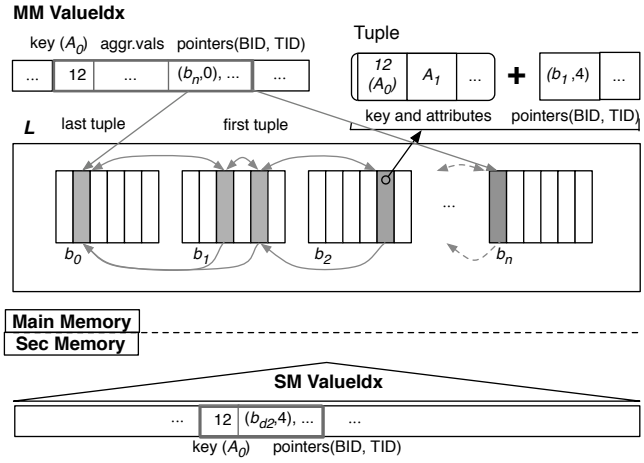
[6]Actually, large historical periods could translate into very large streaming tables. Multi-level storage is a research issue in itself [29]. This issue is out of the scope of this paper and we limit our discussion to a two-level memory hierarchy.



(a) The storage structure



(b) The interval-based index



(c) The value-based index

Fig. 3. Streaming table structures

index; on the other hand, small ti values imply a greater number of array elements.

At the same time, BID insertions and deletions are performed at the back and at the front of the array, respectively, because of the temporal order of blocks.

Summing up all operations on $MMIntIdx$ show constant time cost. This data structure therefore represents an optimal in-memory solution for the temporal indexing of a streaming table where high-speed data streams that enter and exit main memory require fast block insertions and deletions and time-based searches are very common.

Similarly to its main memory counterpart, the secondary memory component of an interval-based index must be an index structure that efficiently adapts to the high insertion rate the datafile undergoes. To this end, we exploit the temporal order of the streaming tuples in the datafile to introduce a block-oriented clustered B$^+$-tree denoted as $SMIntIdx$ and built on the lower bound of the time interval covered by each block. More precisely, the i-th index entry of $SMIntIdx$ has the form $(b_i.getFirst().time(), BID_i)$ where b_i is the disk block in the datafile identified by the BID BID_i (see Fig. 3(b)). $SMIntIdx$ is a coarse-grain index that:

- greatly increases overall speed of retrieval. Indeed, any streaming tuple t lies in the block b_i such that $b_i.getFirst().time() \leq t.time() \leq b_{i+1}.getFirst().time()$. Therefore, temporal conditions are efficiently solved by accessing first $SMIntIdx$ and then by accessing data sequentially as tuples with subsequent timestamps are guaranteed to be physically adjacent;
- limits index update overhead, because it only undergoes one insertion for each block write.

The interval-based index is updated in three different block lifecycle stages:

- when a block in the main memory list is started and when it is filled up (see algorithms 1 and 2 in Fig. 4, respectively): in these cases we push its BID into the queues pointed by the elements at the back of the array and update $rear$ to the current time interval. Note that when the system starts and then the block is the first one, algorithm 1 at Line 2 initializes the $start$ value, i.e. the lower bound of the time interval that $MMIntIdx$ covers, to the timestamp of the first streaming tuple;
- when a block is moved to secondary memory (see algorithm 3 in Fig. 4): in this case the block is the ancient block in L and therefore its BID is the head of the queue pointed by the front element of $MMIntIdx$. In this case BID is popped from $MMIntIdx[front]$, while $MMIntIdx[front]$ is removed if it becomes empty. This process is iterated on the subsequent array elements. Finally, we add the BID with its key to $SMIntIdx$.

C. Value-based indices

Value-based indices can be explicitly created through the `CREATE INDEX` construct.

A value-based index for streaming tables essentially extends standard secondary memory index types with a main memory structure for fast access to the most recent tuples of the streaming table and an efficient approach for secondary memory index updates.

The secondary memory component of an index created on a table attribute, $SMValueIdx$, is a B$^+$-tree or a hash,

Algorithm 1 Block start

Require: Block id: BID, the first tuple: t_s

1: **if** $start = -1$ **then**
2: $start = t_s.time()$
3: $front = rear = 0$
4: **else**
5: $offset = \frac{t_s.time() - start}{ti}$
6: $rear = (front + offset)\ mod\ dim$
7: enqueue($MMIntIdx[rear]$,BID)

Algorithm 2 Block full

Require: Block id: BID, the last tuple: t_e

1: $offset = \frac{t_e.time() - start}{ti}$
2: $temp = (front + offset) \bmod dim$
3: **for** $i = rear + 1$ to $temp$ **do**
4: enqueue($MMIntIdx[i]$,BID)
5: $rear = temp$

Algorithm 3 Block write

1: $stop = FALSE$
2: $BID = dequeue(MMIntIdx[front])$
3: **while** NOT $stop$ **do**
4: **if** $MMIntIdx[front]$ is empty **then**
5: $front = (front + 1) \bmod dim$
6: $start = MMIntIdx[front].getFirst().time()$
7: **if** $first(MMIntIdx[front]) \neq BID$ **then**
8: $stop = TRUE$
9: **else**
10: $BID = dequeue(MMIntIdx[front])$
11: **else**
12: $stop = TRUE$
13: Let b be the block with ID BID
14: Insert $(b.getFirst().t, BID)$ into $SMIntIdx$

Fig. 4. Update algorithms for interval-based indices

depending on the indexing method specified in the `CREATE INDEX` query.

Similarly to [30], its main memory component is a linked list of entries $MMValueIdx$ where every entry contains an attribute value, some aggregate values, and points to some tuples in the block list L (see Fig. 3(c)). More precisely, all tuples with the same attribute value in L are linked through forward and backward pointers; additionally, the corresponding entry in the index points to the first and to the last tuples, thus creating a ring for each attribute value. Moreover, all the tuples in the same block point to the first tuple with the same attribute value in the backward block. Therefore, each block entry follows the structure shown in Fig. 3(c). Finally, among the aggregate values, we usually maintain at least the value frequency.

Searches in $MMValueIdx$ should in principle support the same comparison predicates as its secondary-memory counterpart. When the latter is a B$^+$-tree, besides equality comparisons, we must support in-memory range searches too. To this end, we consider two alternatives both showing a logarithmic cost: to implement the index as an ordered list or as a self-balancing search tree. If $SMValueIdx$ is an hash, the range search support is not required. On the other hand, value searches can be sped up by means of the main memory solutions specified above as well as hash tables with buckets containing linked lists of entries.

Insertions in $MMValueIdx$ proceeds as follows. When a tuple enters a list block, the TSM scans $MMValueIdx$ and a pointer is created from the index to the new tuple that becomes the first tuple of the list, while the youngest tuple in the ring, i.e. the previous first tuple, is linked to the new tuple. Similarly, the pointer to the backward block is derived from the previous first tuple in the following way: if the latter is in the same block as the new tuple then the new tuple inherits its pointer, otherwise the new tuple points to the latter's block. Pointer settings have a constant cost while index access cost depends on the implemented structure.

When the oldest block is transferred to secondary memory, the TSM must: 1) remove expired tuples from $MMValueIdx$; 2) update $SMValueIdx$ accordingly.

As to the first aspect, we start the deletion process from the youngest tuple in the block and follow the backward tuple links until the end of the block. This is repeated for each tuple in the block in order to derive the chains of the tuples that share the same attribute value. It is worth noting that the tail of each chain is the last tuple pointed by the corresponding value entry in $MMValueIdx$. Therefore, we can safely remove each chain:

- by updating the last tuple of $MMValueIdx$ entry with the tuple that is forwardly linked to the head chain and removing both the last tuple's backward pointers;
- by removing the backward block pointers of all the tuples that are in the same block as the last tuple and that share the same attribute value. This is implemented by following the chain of the forward pointers that link the last tuple with the other tuple in its block.

Each of these pointer update operations has a constant cost, therefore the deletion cost is proportional to the average number of tuples per block with the same attribute value.

As to the second aspect, the main issue is to support very high insertion rates while maintaining $SMValueIdx$ organization that is a guarantee for query performance. In this context, the primary critical aspect is that the values of the indexed attribute of the streaming tuples to be inserted are typically randomly distributed and thus each successive tuple is likely to end up in a different leaf of the B$^+$-tree. As a consequence, performing one insertion for each tuple is very costly, greatly reducing the rate at which data can be recorded [32]. Instead, we implement a buffering insertion technique [33] that founds on an in-memory data structure, e.g. a dictionary or a BST, acting as a buffer where to group streaming tuples assigned to secondary-memory. At any time, the current structure is available for read operations too and concurrent accesses are regulated by standard concurrency control mechanisms. When the ratio between the number of distinct indexed (key) values and the total number of tuples in the current structure goes under a given threshold, TSM starts a new current structure while it writes out the previous one to disk. In particular, it searches in the $SMValueIdx$ for the key values in sorted order and for each value it inserts the TID list associated with it. In this way we reduce disk I/Os and fault in the CPU cache.

D. Historical period management

Up to now, we have intentionally neglected the impact of the historical period (hp) on the permanent component of both the storage structure and the proposed indices. Indeed, while hp slides, outdated data must be thrown away.

To this end, among the different approaches that have been proposed in [34], we follow an approach that is similar to the Wait and Throw Away (WATA) scheme since it shows two important advantages for bulk deletion: it achieves the best performance and does not need complex deletion code. It consists in dividing the historical period hp into k compartments plus the current compartment. Each compartment spans an equal time interval w, where $k = \lceil \frac{hp}{w} \rceil$, and is associated with one datafile and instances of the indices previously introduced.

We maintain a circular array of pointers to compartments. Insertions are always performed on the current compartment and when the latter fills up, its pointer is inserted in the circular array by overwriting the pointer to the oldest compartment that is discarded.

Note that this approach occasionally maintains data older than the required window, as at every instant the covered time interval is between hp and $hp + w$. In our opinion, this phenomenon, named soft window in [34], is absolutely acceptable as only `Scan` calls are solved at this memory level.

E. `CScan` and `Scan` implementation

Whenever a `CScan` call is submitted, we store the call parameters for future resumes to be carried out by the TSM as well as by the QP through `Resume` calls.

When a `CScan` request is resumed, the TSM is in charge of returning the tuples in the queried streaming table that satisfy the predicates specified in the `pred` parameter and that are within the `window` parameter. In a similar vein, any `Scan` request is dealt with by satisfying the `pred` parameter. The former is solved at main memory level only, whereas the latter can also (or only) involve secondary memory.

In both cases, the scan algorithm takes advantage of the interval-based index when `pred` contains one or more temporal predicates or when a time-based window is specified.

For instance, as far as the block list in main memory is considered, the algorithm goes through two steps. First, it implements a coarse-grain search that randomly accesses $MMIntIdx$ to select the set of BIDs to be read. In the second phase, it performs a linear search on the corresponding blocks. This step retrieves every tuple in the block and tests whether its attribute values satisfy the remaining predicates in `pred`.

In case of `CIndexScan` or `IndexScan`, the value-based index specified in the `ixid` parameter is exploited to efficiently solve predicates.

When both kinds of indices can be exploited, the search is further speeded up by accessing only those tuples that both satisfy the specified value-based constraints and are in the BID list. To this end, for each selected $MMValueIdx$ entry, the algorithm starts from the first tuple and, instead of sequentially scanning the whole list through the backward pointers, whenever it accesses the first tuple of a block that is not in list, it directly "jumps" to the previous block by following the backward block pointer.

Example 5: With reference to the query in Table I(f), the first block to be accessed according to the $MMIntIdx$ shown in Fig. 3(b) is b_{11}. Therefore, assuming that $MMValueIdx$ in Fig. 3(c) is about the `VID` attribute, for each of the VIDs of the `INVOLVEDVIDS` table, the algorithm reaches b_{11} by accessing each block between b_n and b_{11} at most once.

VII. Experimental Evaluation

This section presents the results of the experimental study we conducted to assess the performance of our approach. Tests on our prototype running on a single-node architecture show that the proposed technology is able to cope very well with a continuous write-intensive and query-intensive workload.

We are aware that approaches relying on parallel and/or distributed architectures would help in managing heavier workloads and data volumes. However, this is not the focus of this paper, and we plan to explore this issue in future work.

A. Experimental setting

Our test settings derive from the extended Linear Road scenario described in Section II. The Linear Road input data is generated[7] at varying levels of complexity (i.e., number of simulated expressways). The data for testing each expressway consists of over 12 million position reports reaching a rate of 100000 reports per minute, 60000 account balance query requests and 12000 daily expenditure requests, all delivered in 3 hours of simulation. We recall that, in Linear Road, a system is said to achieve a so-called *L-rating* if it meets the required response time and accuracy constraints for all queries while supporting an input level encompassing L expressways of data (e.g., rating L1 for 1 expressway). This corresponds, for L1, to output 2 million toll and 28000 accident notifications, as well as to answer each of the requested account balance and expenditure queries.

In our case, we will execute all the complex real-time requests while, at the same time, demanding the system to maintain the full stream history of position reports and to make such history always queryable. We will indicate these extended one- and two-expressway scenarios as L1+ and L2+, respectively. Moreover, we will consider two additional scenarios, L1++ and L2++, where we handle the whole workload required by the L1+ and L2+ scenarios, respectively, and add a varying number of OTQs on the historical position report data acquired during the simulation. More specifically, in the L++ scenarios we instantiate additional randomly distributed OTQs over past data, involving predicates on both time and key values. We consider three different OTQ workloads (low, med and high load, with 20/min, 100/min and 200/min queries submitted to the system, respectively) and three query selectivity levels (high, med and low with query results composed of a number of tuples going from 10000 to over 60000).

The streaming tables data store structures and management code are implemented in Java 1.6. We implement both the

[7] http://www.cs.brandeis.edu/∼linearroad/tools.html

	Query	Min	Max	Avg	% over	# over
L1+	toll notification	0	3.802	0.635	0%	0
	acc notification	0	3.519	0.619	0%	0
	acc balance	0.013	3.786	0.635	0%	0
	expenditure	0.003	3.697	0.666	0%	0
L2+	toll notification	0	6.64	0.752	0.250%	22095
	acc notification	0.006	6.506	0.737	0.310%	978
	acc balance	0.005	6.527	0.76	0.311%	376
	expenditure	0.027	6.575	0.803	0%	0

TABLE III
RESPONSE TIME (SEC) FOR L1+ AND L2+

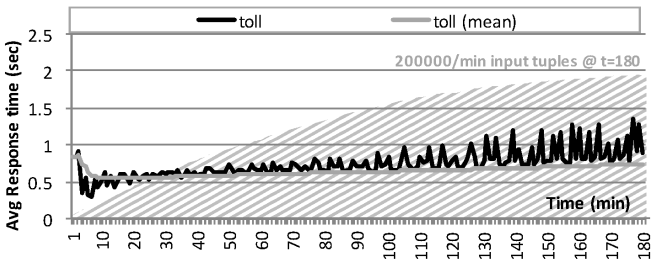


Fig. 5. Average response time detail (toll queries, L2+)

main and secondary memory components and indices by means of the ad-hoc data structures presented in Section VI; we also exploit the Oracle BerkeleyDB 11gR2[8] lightweight embedded database library for the implementation of the secondary memory permanent component of the B+-tree structures in our secondary memory indices. Standard tables (i.e. static data) are maintained in PostgreSQL 9.0. Linear Road queries are implemented in Java too and the required streaming data accesses are performed through continuous as well as one-time scan calls. Note that, at the current stage, our implementation is strictly focused on the key aspect of the paper, i.e. the streaming table data structure and the deep study of its performances on the kind of described workload. The actual integration inside a DBMS architecture, which is in line with our final objective but beyond the scope of this paper, is mainly a prototyping work that we leave for the near future and that will not directly impact on the measured performances.

All the experiments are executed on an Intel Core2 Quad Q9450 2.66Ghz Win7 Pro 64Bit workstation, equipped with 4GB RAM and a 500GB 7200rpm SATA disk. Java heap size is set at 2GB.

B. Performance analysis

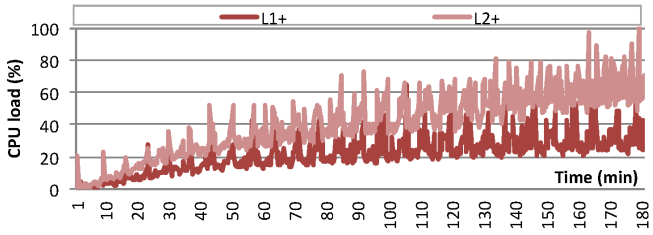
Response Time. Table III shows the minimum, maximum and average response time the system achieves for all four types of Linear Road queries, together with the percentage of query answers that exceed the maximum time threshold of 5 seconds required by the benchmark (last column). In the top part of the table, we see that in scenario L1+ all answers are delivered well ahead of the required time (a mean of nearly 0.6 secs is measured for all 4 query types). Even at the demanding L2+ scenario, the system is able to cope very well with the double input frequency: while only 0.3% (at most) of the queries are delivered slightly above the required threshold (maximum response time is 6.5 secs), for most of them average response time is almost unchanged w.r.t. L1+ performance (for all queries 0.8 secs at most).

It is important to note that, in all Linear Road scenarios, the more time goes by, the more data enter the system. For instance, at L1+, in the first minute only 1118 position reports enter the system, whereas in the last minutes of the 3 hours simulation we get up to 100000 tuples per minute. Therefore, we want to analyze the response time trend over simulation time, as shown in the graph of Fig. 5 for toll notifications (the other queries showed similar trends). In particular, the graph shows, for each simulation minute in L2+, the average response time of the queries for that minute (dark colored line), together with the average response time computed up to that minute (light colored line); the growing input workload is depicted on the background. As we can see, the system response time is stable even in the final part of the L2+ simulation (where up to 200000 tuples per minute are processed), while the average response time also increases very slowly.

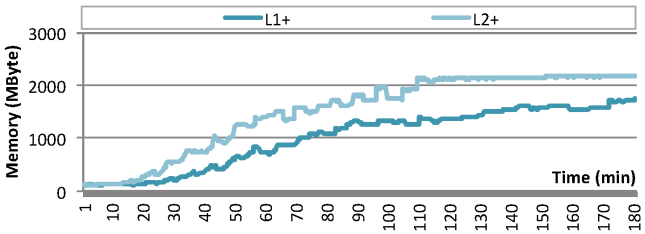
[8]http://www.oracle.com/us/products/database/berkeley-db

	Profiler	Min	Max	Avg
L1+	Mem (MByte)	110.55	1768.15	1015.84
	CPU (%)	0	67.60	21.78
L2+	Mem (MByte)	110.81	2187.96	1502.30
	CPU (%)	0	94.54	40.47

TABLE IV
MEMORY AND CPU PROFILING ANALYSIS FOR L1+ AND L2+



(a) CPU profiling



(b) Memory profiling

Fig. 6. Detailed system profiling and scalability analysis for L1+ and L2+

Profiling and scalability. Besides query response time, we profiled the system as to percentage of CPU usage and allocated memory: Table IV shows a summary of the results, with minimum, maximum and average figures, while Figs. 6(a) and 6(b) show a detailed trend over simulation time for CPU and memory, respectively. L1+ successfully completes with an average and maximum memory occupation of 1015 and 1768

	High sel			Med sel			Low sel		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
L1++									
Low load	0.035	0.112	0.056	0.108	0.266	0.132	0.219	0.672	0.268
Med load	0.041	0.082	0.053	0.125	0.160	0.133	0.254	0.358	0.283
High load	0.040	0.118	0.056	0.124	0.281	0.173	0.258	0.472	0.322
L2++									
Low load	0.038	0.566	0.060	0.117	0.376	0.154	0.248	0.832	0.287
Med load	0.035	0.551	0.056	0.126	0.753	0.149	0.226	1.042	0.302
High load	0.037	0.471	0.060	0.132	0.967	0.185	0.262	1.373	0.338
L1++ (No Index baseline)									
Low load	3.755	6.051	4.439	3.821	5.502	4.491	3.571	7.823	4.641

TABLE V
QUERY RESPONSE TIME (SEC) FOR L1++ AND L2++

MB, respectively, while the CPU is occupied by 21.78% on average. From Figs. 6(a) and 6(b) we can clearly see how the system reacts over time to the increasingly heavier input workload. Besides a good regularity and stability, the results show a very good scalability of both CPU and memory usage over time: while the input rate varies over time with an overall rate of 100:1, the required CPU/memory vary at most with a 30:1 and 20:1 rate, respectively.

L++ scenarios. We will now deepen the performance offered by the system by considering the two additional scenarios L1++ and L2++. The results are shown in Table V for the three considered workloads and selectivity levels. As we can see, the system is able to answer all queries in a very efficient way, thus proving the good performance of the proposed time and value indices: for instance, in L1++ with medium selectivity and high workload, the system is able to answer in less than 0.2 secs, on average (the maximum time is 0.28 secs). Average response times for L2++ are nearly unchanged. In the lower part of the table, we also show a comparison baseline where no indices are made available for solving the query predicates: in this case, response times are one or even two orders or magnitude larger, showing again the benefits on efficiency given by the indices we implemented on streaming tables (results are shown for low workload and L1++ only, since without indices the system does not keep up with med and high workloads). Furthermore, we also have to report that the additional OTQs in L++ are handled with a very limited additional amount of CPU and memory (5% and 30 MB more than a standard L+ execution, respectively).

State-of-the-art performance investigation. As we have seen, the specific workload we consider in this paper requires the efficient handling of all the following aspects at once: (a) complex OTQs over recent, historical and static data; (b) continuous reads; (c) continuous writes at very high rate, with guarantee of full data persistency over the full period of time. The evaluation on the L+ and L++ scenarios clearly showed the ability of our proposal to seamlessly meet all the above mentioned requirements. We will now investigate how existing state of the art systems (i.e. DBMSs and DSMSs) behave when dealing with this kind of workload, possibly comparing their performances to the streaming table implementation.

We will start our analysis by considering state-of-the-art DBMSs. In the first test, we consider the high data load of

	Recent / just arrived tuples			Non-recent tuples		
RespTime (sec)	Delay@60	Delay@90	Delay@120	Min	Max	Avg
StrTable	0	0	0	0.093	0.156	0.105
Std DBMS	2	4	41	0.124	0.156	0.129
NoSQL DBMS	0	0	0	0.084	0.153	0.108

TABLE VI
STATE-OF-THE-ART DBMSS: RESPONSE TIME AND DELAY ANALYSIS FOR HIGH UPDATE RATES ("LIGHT" L2 SCENARIO)

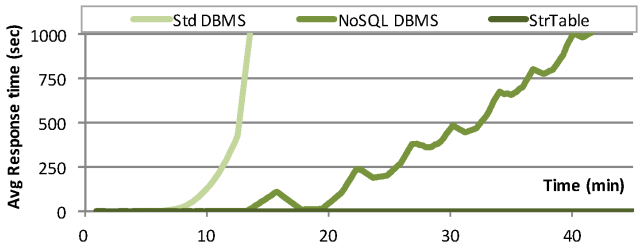


Fig. 7. State-of-the-art DBMSs: Response time analysis (L1++)

the L2 scenario, where the tuple input rate achieves nearly 200000 tuples per minute, and we are mainly interested in seeing if the considered systems are able to keep the pace of indexing the data (requirement (c)), while answering a small number of simple submitted OTQs. In this "light" L2 scenario, the complete Linear Road calculations and complex OTQs and CQs as of (a) and (b) are not considered. Table VI shows the results we obtained on our system ("StrTable" in table), on a "plain" standard DBMS (PostgreSQL 9.0, "Std DBMS" in table) and on a NoSQL DBMS (Apache Cassandra 1.1.1[9], "NoSQL DBMS" in table). In the case of our system, the main memory windows are set to a single tuple; all systems are working only in secondary memory. The buffering insertion and concurrency control techniques available in our approach provide very short response time for queries requesting non-recent data (0.1 secs on average, as shown in the right part of table). Moreover, we also have null delays for queries involving recent/just-arrived data, meaning that all the required results are always instantly available at all the high input rates at 60, 90 and 120 minutes of simulation (see central part of table). On the other hand, a standard DBMS provides comparable response time for non-recent data, but exponentially growing delays (offline time periods) for recently inserted data (up to 41 secs at 120 minutes of simulation), thus failing the test. Indeed, standard relational DBMSs would certainly prove very powerful in managing frequent and complex OTQs as in requirement (a), however the very high insertion rates on their indices as of requirement (c) are not the kind of workload they are designed for, and ultimately lead to unacceptable results. Instead, the NoSQL DBMS, as expected, succeeds, since it is able to keep up with the high write rates and simple OTQ requests of this "light" scenario.

In our second test on DBMSs, besides requirement (c), we will also consider requirements (a) and (b) in a complete L++ scenario and see which systems can withstand the whole workload. Fig. 7 shows the average response time of PostgreSQL

[9]http://cassandra.apache.org

	Avg RspTime (sec)	
	L1	L2
StrTable	0.65 (L1++)	0.78 (L2++)
Aurora	0.63	0.945
Stream	0.75 (*)	1.12 (*)
DataCell	0.8	1.71

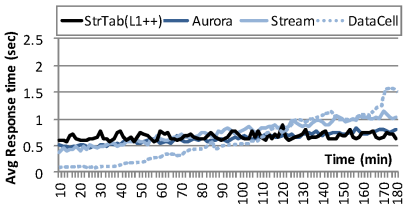


Fig. 8. State-of-the-art DSMSs: Response time for L1/L2 scenarios (left) and detailed analysis for L1 (right)

and Apache Cassandra compared to ours in the L1++ scenario. In particular, we performed all the required insertions and queries in real time in all systems, transforming each CQ into sets of OTQs in the case of the two DBMSs. In this case, all systems are configured to exploit both secondary and main memory as far as allowed by their way of working: streaming tables directly offer a gradual flow from main to secondary memory which is optimized for this kind of workload, the DBMS stores the data on disk and exploits a main memory buffer for accessing frequently requested data, while the NoSQL system exploits in-memory tables that are periodically transferred to disk. Only the first 45 minutes of simulation are shown since the performance gaps are already well evident: standard relational DBMSs, as was already clear from the previous test, are not able to sustain the write rate, and response times are higher than two minutes already at 10 minutes of simulation, exponentially growing. The performances of Cassandra are better than standard DBMSs, however they show that not even NoSQL systems (at least on our single node setting) are able to keep up with the full (a), (b) and (c) requirements of the L1++ scenario. This is possibly due to the fact that, even if they support very frequent reads and updates, their indices provide efficient support only for simple key-value requests, thus the fact that complex and continuous queries are not efficiently or natively supported ultimately hampers their results.

The previous two tests showed the relationship between our proposal and DBMSs; they also showed that the work we performed on the secondary memory structures was indeed necessary to withstand the requirements of our workload, constantly producing answers in a few fractions of seconds while sustaining very high update rates on the underlying indices. As to DSMSs, they are not designed nor able to meet requirement (a) for historical data and requirement (c). Still, there is an open question we want to answer in this evaluation: does supporting the full range of requirements introduce significant performance slowdowns in managing recent data and CQs w.r.t. standard DSMSs? We considered a number of DSMSs whose full source and Linear Road implementation were publicly available: Aurora[10] [12], STREAM[11] [13] and DataCell[12] [18]. Figure 8 shows their average response time to execute standard Linear Road scenarios (L1 and L2), and the average response time for our system to execute the extended L++ versions, thus also maintaining the full history of position reports and executing complex additional OTQs over the whole data. As we can see, the performance of our system is in line with the other DSMSs, even if performing all the additional required operations. Please note that STREAM does not cover the full benchmark specifications, since it does not handle static expenditure data (thus, we marked its response times with a "*" in the left part of the figure). Further, from our tests we also noticed some significant weak points for other systems: an ever-growing response time trend for Datacell (see the right part of Figure 8) and, in the case of STREAM, CPU figures constantly higher than 99%, maybe due to the overhead of dealing with the much larger number of streams needed to implement the benchmark in a standard DSMS.

[10]http://www.cs.brown.edu/research/aurora/

[11]http://infolab.stanford.edu/stream/code/

[12]http://github.com/snaga/monetdb/tree/master/sql/backends/monet5/datacell

VIII. Related Work and Conclusions

With the aim of supporting new kinds of modern applications that need to access both live and historical/static data, various research efforts have been devoted to the development of solutions, where three main approaches that relate DSMS and DBMS technologies can be identified.

DSMS extended with DBMS functionalities. Some DSMSs have extended towards mechanisms that store data permanently. We refer, for instance, to [12], [13], [6], [7], [14], where any flow of stored tuples is actually a stream and thus OTQs reduce to CQs. This approach suffers from the overhead of converting huge amounts of stored data to streams. As a partial solution to this problem, [14] employs data reduction techniques to retrieve only sampled data. In most of these systems, the storage manager and the query processor are tightly coupled. The decoupling of these components is instead a fundamental design principle that provides flexibility, adaptation to specific requirements, and optimizability [6]. Essentially, this approach requires to redesign from scratch functionalities that are well-established in the DBMS context.

Combination of DSMS and DBMS. Recently, some commercial DBMS vendors offer powerful platforms for the development of complex event processing and real-time analytical applications, where databases are considered as event sources and/or targets [35], [36], [37], [38]. However, these systems implement a dichotomic DSMS-DBMS vision as in other two-layered solutions (e.g. ,[5], [9], [10]) where the DSMS and the DBMS are combined as independent systems, each one devoted to the management of data they have been conceived for (live and historical/static data, respectively). The main drawback of this approach is the presence of inherent inefficiencies due to the lack of continuous update capabilities by traditional DBMSs, as shown in Sect. VII and also discussed in [8]. To overcome this problem, in [15] a-priori-known SQL queries continuously run on streams to reduce the cardinality of the streaming data to be stored, thus giving up data completeness.

DBMS extended with DSMS functionalities. Some works acknowledge the power of a core of well-established DBMS techniques and charge DSMSs with short-sightedness as to treating stream processing distinct from traditional data processing [16], [17], [8], [18], [19]. [8] advocates a stream-relational system and it is the work most similar to ours for

the underlying principles of proposing an integrated solution. However, in [8] live and past stream data are managed separately, where streams are the results of CQs and are not archived, while persistence of past streamed data is provided by means of standard SQL tables. Although a kind of "bridge" is created in between, the dichotomic vision of live and past streamed data still remains. In [18] streaming tuples are stored in temporary main-memory tables and once a tuple has been seen by all relevant queries/operators it is dropped from its table. This tuple management policy fits for CQs only. [19] focuses on handling incremental stream processing at the query plan and scheduling level. A similar approach is proposed by [17], that extends a DBMS query engine to deal with chunks of streaming data, by continuously generating an unbounded sequence of query results, one for each data chunk. The inefficiencies due to the lack of continuous update capabilities at a standard DBMS storage level still remain. In order to cope with this problem, in [17] continuous database updates are performed in a chunk-based fashion, whereas [19] adopt a column-oriented storage engine to guarantee efficient storage/access to large volumes of streaming data. Finally, [16] is a NewSQL system extended with stream processing capabilities that behaves similarly to a DSMS.

All above systems do not offer a fully native representation of streaming data in the DBMS. Streaming tables are introduced exactly with this purpose. Streaming tables allow streaming data to seamlessly flow from main-memory to secondary memory storage structures according to the specific workload to be supported, in a completely transparent way. To the best of authors' knowledge, no other system features this property for stream data management, thus offering a completely integrated solution in a DBMS. As a clear outcome, querying streaming tables is as familiar as querying standard tables, and in Sec. III we gave an intuition of how easy it would be to satisfy the Linear Road benchmark when streaming tables are available in a DBMS, with respect to the effort required, for instance, by an extended DSMS like STREAM [13].

Streaming tables are designed to support unconventional workloads featuring massive write rate and low query answering latency on a blend of live/historical/static data. The issue of efficiently managing unconventional workloads has been addressed by other papers in the literature that redesign the database architecture to this purpose (e.g., [20], [22]). For instance, [22] presents a novel relational table implementation and [20] proposes a new database architectures to dynamically support different workloads. These proposals refer to in-memory implementations only. Other proposals (e.g., [29]) exploit database architectures that efficiently support lookup-intensive and aggregation operations and proves to be advantageous for data-intensive analytics, where queries check attributes almost in a partitioned way [24]. However, in a more general SQL query-intensive scenarios on write-intensive OLTP involving complex queries on several attributes, performances of these solutions degrade significantly, as shown by experiments in Sect. VII. To overcome this limit, [29] designs a massively parallel distributed engine. This is an orthogonal aspect and we are aware that streaming tables would certainly increase their efficiency by exploiting distributed storage and parallel and distributed query execution. We plan to explore this feature in future work by exploiting computation frameworks like [39], [40].

Streaming tables are equipped with indices that are exploited for both OTQs and CQs execution. Existing indices for streaming data are designed for main memory only and they are not burdened by the problem of transferring data in secondary memory structures (e.g. [30]). The proposed interval-based and value-based indices span both main memory and secondary memory. The value-based index is similar to [30] but it includes additional pointers whose maintenance overhead is negligible with respect to the advantages given on accesses and updates. As the experiments in Sect. VII proved, streaming tables show very promising performances on a wide spectrum of query specifications (i.e., on live and historical/static data); the supported workload goes beyond what existing state-of-the-art systems (DSMSs and DBMSs) are usually designed/able to work with.

Summing up, we believe that a DBMS extended with streaming tables and continuous query support may lead to many benefits in the design and management of hybrid data-intensive applications. We experienced the applicability of streaming tables in various smart city scenarios, where we focused on scalability issues on data acquisition by means of data reduction techniques in Vehicle-to-Infrastructure transmissions [41], and on adaptability to different traffic data and query workloads in various Intelligent Transportation Systems contexts [42]. We are aware of several research issues that need to be reviewed in order to fully integrate streaming tables in a DBMS, like query optimization, recovery, and transaction management, just to mention a few. All of these deserve a further investigation on their own and are planned in our future research agenda.

REFERENCES

[1] S. Chandrasekaran et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," in *Proc. of CIDR*, 2003.

[2] J. Chen, D. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," in *Proc. of SIGMOD*, 2000, pp. 379–390.

[3] A. Sun and M. Hu, "Query-Guided Event Detection From News and Blog Streams," *IEEE Trans. on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 41, no. 5, pp. 834–839, 2011.

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

[5] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee, "Moirae: History-Enhanced Monitoring," in *Proc. of CIDR*, 2007, pp. 375–386.

[6] I. Botan, G. Alonso, P. Fischer, D. Kossmann, and N. Tatbul, "Flexible and scalable storage management for data-intensive stream processing," in *Proc. of EDBT*, 2009, pp. 934–945.

[7] S. Chandrasekaran and M. Franklin, "PSoup: a system for streaming queries over streaming data," *VLDB J.*, vol. 12, no. 2, pp. 140–156, 2003.

[8] M. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, "Continuous Analytics: Rethinking Query Processing in a Network-Effect World," in *Proc. of CIDR*, 2009.

[9] L. Golab, T. Johnson, J. Seidel, and V. Shkapenyuk, "Stream warehousing with DataDepot," in *Proc. of SIGMOD*, 2009, pp. 847–854.

[10] K. Tufte, J. Li, D. Maier, V. Papadimos, R. Bertini, and J. Rucker, "Travel time estimation using NiagaraST and latte," in *Proc. of SIGMOD*, 2007, pp. 1091–1093.

[11] "The PEGASUS Project," http://pegasus.octotelematics.com/.

[12] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDBJ*, vol. 12, no. 2, pp. 120–139, 2003.

[13] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution"," *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006.

[14] S. Chandrasekaran and M. Franklin, "Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams," in *Proc. of VLDB*, 2004, pp. 348–359.

[15] "PipelineDB," https://www.pipelinedb.com.

[16] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. Zdonik, "S-Store: A Streaming NewSQL System for Big Velocity Applications," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1633–1636, Aug. 2014.

[17] Q. Chen and M. Hsu, "Cut-and-rewind: Extending query engine for continuous stream analytics," in *TLDKS XXI*, ser. LNCS, A. Hameurlain, J. Kueng, R. Wagner, A. Cuzzocrea, and U. Dayal, Eds., 2015, vol. 9260, pp. 94–114.

[18] E. Liarou, R. Goncalves, and S. Idreos, "Exploiting the power of relational databases for efficient stream processing," in *Proc. of EDBT*, 2009, pp. 323–334.

[19] E. Liarou, S. Idreos, S. Manegold, and M. Kersten, "Enhanced stream processing in a dbms kernel," in *Proc. of EDBT*, 2013, pp. 501–512.

[20] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads," in *Proc. of SIGMOD*, 2016, pp. 583–598.

[21] J. Dittrich and A. Jindal, "Towards a One Size Fits All Database," in *Proc. of CIDR*, 2011, pp. 195–198.

[22] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable Performance for Unpredictable Workloads," *PVLDB*, vol. 2, no. 1, pp. 706–717, 2009.

[23] R. Snodgrass, Ed., *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[24] J. Han, H. E, G. Le, and J. Du, "Survey on NoSQL Database," in *Proc. of Int. Conf. on Perv. Comp. and App. (ICPCA)*, 2011, pp. 363–366.

[25] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear Road: A Stream Data Management Benchmark," in *Proc. of VLDB*, 2004, pp. 480–491.

[26] U. Srivastava and J. Widom, "Flexible Time Management in Data Stream Systems," in *Proc. of PODS*, 2004, pp. 263–274.

[27] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases.* Addison-Wesley, 1995.

[28] L. Golab and M. T. Özsu, "Update-Pattern-Aware Modeling and Processing of Continuous Queries," in *Proc. of SIGMOD*, 2005, pp. 658–669.

[29] M. Ahuja, C. Chen, R. Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, and K. Uppala, "Petascale data warehousing at Yahoo!" in *Proc. SIGMOD*, 2009, pp. 855–862.

[30] L. Golab, S. Garg, and M. T. Özsu, "On Indexing Sliding Windows over Online Data Streams," in *Proc. of EDBT*, 2004, pp. 712–729.

[31] J. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a Database System," *Foundations and Trends in Databases*, vol. 1, no. 2, pp. 141–259, 2007.

[32] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental Organization for Data Recording and Warehousing," in *Proc. of VLDB*, 1997, pp. 16–25.

[33] G. Graefe, "B-tree indexes for high update rates," *SIGMOD Record*, vol. 35, no. 1, pp. 39–44, 2006.

[34] N. Shivakumar and H. Garcia-Molina, "Wave-Indices: Indexing Evolving Databases," in *Proc. of SIGMOD*, 1997, pp. 381–392.

[35] "InfoSphere Streams," http://www-03.ibm.com/software/products/us/en/infosphere-streams.

[36] "StreamBase," http://www.streambase.com.

[37] "StreamInsight," http://msdn.microsoft.com/en-us/sqlserver/ee476990.aspx.

[38] A. Witkowski, S. Bellamkonda, H. Li, V. Liang, L. Sheng, W. Smith, S. Subramanian, J. Terry, and T. Yu, "Continuous Queries in Oracle," in *Proc. of VLDB*, 2007, pp. 1173–1184.

[39] "Apache Spark," http://spark.apache.org.

[40] "Apache Storm," http://storm.apache.org.

[41] L. Carafoli, F. Mandreoli, R. Martoglia, and W. Penzo, "A framework for ITS data management in a smart city scenario," in *Proc. of SMARTGREENS*, 2013, pp. 215–221.

[42] ——, "A data management middleware for ITS services in smart cities," *J. UCS*, vol. 22, no. 2, pp. 228–246, 2016.

Luca Carafoli Luca Carafoli has a master's degree in Computer Engineering. Since 2011, he is a PhD student at the International Doctorate School in ICT at the University of Modena e Reggio Emilia. His current research is about studying new methodologies for efficiently and effectively querying and managing large amounts of streaming data. He applies his studies in the field of Intelligent Transportation Systems to improve their efficiency.

Federica Mandreoli Federica Mandreoli is an Assistant Professor since 2002. In December 2013 she obtained the Italian National Habilitation (Abilitazione Scientifica Nazionale) as associate professor in the Scientific Sector 09/H1 (Information Processing Systems). She has a MS degree in Computer Science and a PhD degree in Computer Engineering, both from the University of Bologna. Her research activity focuses on the management and access of non-conventional data and is currently mainly devoted to integrating streaming data in conventional DBMS, graph structured data and data sharing in heterogeneous and distributed contexts. On these topics, she participated to various financed projects and published about one hundred papers.

Riccardo Martoglia Riccardo Martoglia is an Assistant Professor in the FIM Department of the University of Modena and Reggio Emilia since 2005. He received the MS degree and the PhD degree in Computer Engineering from the same University. His current research interests are about studying new methodologies for efficiently and effectively querying and managing large amounts of non-conventional data, from semi-structured to graph-based, multiversion and streaming data. He participated to several national and European research projects; the results of his research are published in nearly ninety papers.

Wilma Penzo Wilma Penzo is an Assistant Professor in the Department of Computer Science and Engineering (DISI), University of Bologna. She has a MS degree in Computer Science and a PhD degree in Electronic and Computer Engineering, both from the University of Bologna. Her research interests are in the area of information and knowledge management in heterogeneous and distributed contexts and include query processing on graph-based data, stream data management, Semantic Web, semantic P2P systems.