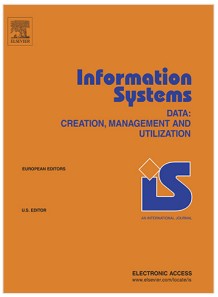
Unleashing the power of querying streaming data in a temporal database world: A relational algebra approach

Journal Pre-proof



Unleashing the power of querying streaming data in a temporal database world: A relational algebra approach

Fabio Grandi, Federica Mandreoli, Riccardo Martoglia, Wilma Penzo

PII: S0306-4379(21)00098-3
DOI: https://doi.org/10.1016/j.is.2021.101872
Reference: IS 101872

To appear in: *Information Systems*

Received date : 2 March 2020
Revised date : 18 July 2021
Accepted date : 28 July 2021

Please cite this article as: F. Grandi, F. Mandreoli, R. Martoglia et al., Unleashing the power of querying streaming data in a temporal database world: A relational algebra approach, *Information Systems* (2021), doi: https://doi.org/10.1016/j.is.2021.101872.

Unleashing the Power of Querying Streaming Data in a Temporal Database World: A Relational Algebra Approach

Fabio Grandi[a], Federica Mandreoli[b], Riccardo Martoglia[b,*], Wilma Penzo[a]

[a] *DISI, University of Bologna, viale Risorgimento, 2, I-40136 Bologna, Italy*
[b] *FIM, University of Modena and Reggio Emilia, via Campi, 213/b, I-41125 Modena, Italy*

Abstract

Modern data-intensive applications have to manage huge quantities of streaming/relational data and need advanced query capabilities involving combinations of continuous queries (CQs) and one-time queries (OTQs) also requiring the verification of complex temporal conditions.

In this paper, we go beyond the disjointed panorama of current approaches and adopt a new holistic approach to the integration of stream processing capabilities into the temporal database world based on the streaming table concept. To this end, we propose a full-fledged query interface composed of a TSQL2-like query language with an underlying algebraic framework. The algebraic framework, which is aimed at implementing the query interface on top of a working DBMS, is made up of: (a) the extended temporal algebra $\mathcal{TA}^\star$ supporting OTQs with an hybrid temporal semantics (sequenced and non-sequenced); (b) the continuous temporal algebra $\mathcal{CTA}$ that extends $\mathcal{TA}^\star$ with window expressions for CQ specification; (c) the translation of $\mathcal{CTA}$ expressions into $\mathcal{TA}^\star$ ones that can be executed by a traditional DBMS with an extended kernel.

Key words: Continuous queries, Data streams, Relational algebra, Temporal DB

1. Introduction

Modern data-intensive applications, including advanced surveillance (e.g., financial market enforcement), monitoring applications in smart cities and healthcare, network applications (e.g., intrusion detection), require an increasingly wider range of data management capabilities in order to be fully supported. On

*Corresponding author.
Email addresses: fabio.grandi@unibo.it (Fabio Grandi), federica.mandreoli@unimo.it (Federica Mandreoli), riccardo.martoglia@unimo.it (Riccardo Martoglia), wilma.penzo@unibo.it (Wilma Penzo)

the one hand, such applications need to manage very large quantities of continuously streaming data in combination with standard relational data. This kind of management goes beyond real-time processing as the incoming data streams need to be persistently stored as historical data in order to make them available to future retrospective analysis. On the other hand, applications must rely on advanced query capabilities that involve the combination of continuous queries (CQs) and one-time queries (OTQs) to satisfy complex analytics requirements, including temporal conditions and data versioning specifications.

For instance, in the context of database forensics, we can consider applications used to review on-line activities and to ex-post assess user responsibility. To this purpose, one application could verify that a financial broker did not sell the stocks of a client, although the bearish indicators and the stop-loss signal triggered on his/her computer screen suggested to do it. The analysis of the broker's past behavior following or not the real-time recommendations given by a stream processing system could be effected by means of a continuous query reconstructing the context of his/her actions whose history is recorded in a temporal database.

The fulfillment of such requirements necessitates advanced query support mixing stream processing with the evaluation of complex temporal conditions involving the different kinds of data. Existing applications cope with this issue by implementing ad-hoc solutions based on the bridge-building between stream data management and traditional database querying [1, 2]. However, they have to rely on currently available systems that usually provide a very limited support for temporal querying, with respect to the potentialities evidenced by the temporal database research. Query interfaces that are made available by most big data processing engines supporting data streams (e.g., [3]) and by stream processing frameworks (e.g., [4–7]), as well as by streaming extensions to traditional DBMSs (e.g., [8–10]), are not based on a temporal data model and do not support a temporal query language [11]. An exception is Flink [12] that acknowledges the importance of managing temporal data and provides specific constructs to support them. However, it does not provide a real temporal query language and users have to improvise on querying data with a temporal semantics by explicitly expressing conditions on time attributes, often giving rise to complex and inefficient queries. Moreover, queries dealing with temporal data have often to be flanked with application-specific procedures, resulting in this way in an overall mixed declarative-procedural query approach that is far from the uniform, transparent and user-friendly declarative query style that characterizes the query language of a real temporal database.

In order to overcome the highlighted difficulties, we embarked in a new holistic approach to the integration of stream processing into the temporal database world. By acknowledging the inherently temporal nature of streaming data, we started this project by introducing in [13] a new kind of temporal table, named *streaming table*, as the essential abstraction for representing streaming data into a temporal DBMS in a native way. In a streaming table, a stream of data entering the system is kept historical for a user-defined period. Streaming tables can be straightforwardly involved in OTQs and CQs, possibly together

with standard relational tables.

The next step, started in the preliminary work [14], is to equip such a data model with a powerful query interface that can satisfy all application needs and that is made up of two complementary components. The first component, which is the external layer of the interface, consists in a friendly SQL-based declarative language, like the reference temporal query language TSQL2 [15], that allows users to write expressive temporal queries and that can seamlessy treat streaming tables as if their were temporal tables. Moreover, since TSQL2 [15] and its evolutions have not been designed to deal with streaming data, the proposed language must be a TSQL2 extension with constructs to specify continuous queries and manage window expressions over data streams. The second component, which is the internal layer of the interface, is the definition of a temporal algebra with the twofold aim to provide a precise semantics to such a TSQL2-like query language and to pave the way for its implementation on a DBMS. In fact, the availability of an algebra has been the main foundation on which a solid relational technology could be built. Efficient query answering in a DBMS is typically implemented by mapping the query into an algebraic expression, being the evaluation of algebraic operators natively supported by the query execution engine. In fact, manipulation of equivalent algebraic expressions provides the basis for cost-based query optimization.

Moreover, the temporal algebra we want to define as main objective should also be implementable with minimal impact in a working DBMS. In this perspective, an extremely interesting approach is presented in [16], where it is shown how to implement a temporal algebra $\mathcal{TA}$ on top of an off-the-shelf non-temporal DBMS by means of suitable kernel extensions. However, the temporal algebra $\mathcal{TA}$ has by design an almost pure *sequenced* semantics, which allows to only combine data having the same validity. On the other hand, most interesting temporal queries in the targeted advanced applications have a *non-sequenced* semantics, that is they need to combine data belonging to different temporal snapshots (e.g., find the stocks that shot up almost five points this morning; in order to evaluate the price increase, the two versions valid at the beginning and at the end of the morning must be compared).

Therefore, we define a more expressive algebra, denoted as $\mathcal{TA}^{\star}$, that extends $\mathcal{TA}$ to also support non-sequenced operations. Notice that, at query language level, the coexistence of sequenced and non-sequenced parts was actually part of the original TSQL2 design, although a precise semantics for their interaction could not be formulated. Nevertheless, the hybrid nature of TSQL2 was highly underrated as a strength so far and even considered a defect, such that offsprings like ATSQL [17] with separated sequenced and non-sequenced statements were proposed. The syntax of $\mathcal{TA}^{\star}$ is further extended to cover the continuous part of the TSQL2-like query language giving rise to a continuous algebra $\mathcal{CTA}$.

We can summarize the contributions of this work, which is an evolution and completion of the framework laid out in [14], as follows:

- We introduce in Sec. 3 a collection of queries demonstrating the expressiveness of the proposed TSQL2-like query language extensions and their

use as motivating example of a financial market surveillance application. The examples illustrate a wide range of requirements that we want to meet with our approach: use of streaming data as temporal data, joint use of streaming and temporal data in continuous queries, offline (retrospective) execution of continuous queries, hybrid sequenced/non-sequenced query semantics in OTQs and CQs. One of the qualifying features of the proposed language extension is that, differently from other CQs languages previously defined, window expressions are allowed in the `WHERE` clause in order to produce *window-based* data selection.

- We extend in Sec. 4 the standard (sequenced) temporal algebra $\mathcal{TA}$ by also allowing a controlled use of non-temporal operators, which are native in every traditional DBMS. The resulting extended temporal algebra $\mathcal{TA}^\star$ intrinsically has a hybrid (sequenced and non-sequenced) semantics and is shown to be strictly more expressive than $\mathcal{TA}$. OTQs semantics funds on $\mathcal{TA}^\star$ expressions over standard, temporal and streaming tables.

- We present in Sec. 5 the continuous temporal algebra $\mathcal{CTA}$. CQs can be formulated as $\mathcal{CTA}$ expressions that are $\mathcal{TA}^\star$ expressions over standard and temporal tables and *windowing expressions* on streaming tables. A wide range of windowing operators are available to this end, including partitioned versions that mimic the SQL group-by mechanism. We provide a formal definition of *legal sliding window expression* and we present a detailed discussion on composition rules of $\mathcal{CTA}$ operators. For a better understanding of the compositional definitions of windowing operators, we introduce some synthetic datatypes and we discuss how they could be useful for implementing windowing operators natively.

- We formally define in Sec. 6 the semantics of CQs. It relies on the adoption of a *sampling operator* that evaluates $\mathcal{CTA}$ expressions at the *time points* specified through a set of evaluation parameters. In this way, we introduce a sort of *on-demand semantics*, by means of which CQs can be executed when query results are needed and required data are available.

- We provide in Sec. 7 a solid hybrid semantics for the proposed TSQL2-like query language in terms of $\mathcal{CTA}$. In particular, we show how SPJ queries with window expressions can be translated into $\mathcal{CTA}$ expressions. Translation examples for the sample queries presented in Sec. 3 are also given in a motivating example reprise.

- We present in Sec. 8 the translation of $\mathcal{CTA}$ expressions into $\mathcal{TA}^\star$ expressions. Such equivalence guarantees the preservation of the semantics of CQs in the corresponding OTQs obtained as result of the translation process and is therefore the basis for the optimization and execution of the proposed query language on top of a traditional DBMS with a kernel extended as shown in [16]. With regard to implementation issues, we rely on a concrete temporal model [16] that supports an extended sequenced

semantics that is able to enforce snapshot equivalence also in the presence of subexpressions involving a non-sequenced semantics.

The paper is completed by Sec. 2 where we recall the necessary notions concerning temporal databases and review the definition of streaming table, by Sec. 9 where related works are discussed and compared with our approach, and by Sec. 10 where we draw our conclusions and outline future research directions.

2. Preliminaries

The continuous temporal data model we propose relies on a multi-temporal relational model [18], where temporal and non-temporal standard tables coexist, extended with streaming tables. In this section we provide some preliminaries for its specification by reviewing the notion of streaming table, first introduced in [13].

First of all, we assume a discrete, ordered and unbounded time domain $\mathcal{T} = \{0, 1, 2, \ldots, now, \ldots, \infty\}$ composed of *chronons* [19], where 0 stands for the earliest time and *now* the current time. A chronon, as defined in [19], is a non-decomposable time interval of fixed unit duration used to represent time instants in the discrete model. We further assume that $\mathcal{T}$ has the semantics of *valid time* [19]. In order to represent a duration of time, we assume *time spans* [19] belong to a domain $\mathcal{I}$ composed of all possible multiples of a chronon duration.

As far as the temporal model is concerned, a temporal relation R with explicit schema $R(A_1, \ldots, A_n)$, with $A_i \in \mathcal{A}$ $(1 \leq i \leq n)$ where $\mathcal{A}$ is the set of attribute names, is represented as $R(A_1, \ldots, A_n|T)$ where T is the implicit timestamp attribute with domain $\mathcal{T}$. If (r, t) is a tuple from R then $r.A_i$ denotes the value of A_i in r and $T(r)$ denotes the tuple timestamp t. Moreover, given any time instant $t \in \mathcal{T}$, we denote with R^t the content of R at time t, also called the *snapshot* of R valid at time t. Notice that we assume here an *abstract temporal database*, according to the terminology introduced in [20], to be used as a representation-independent data model. In its representation as a *concrete temporal database*, we assume we can distinguish two kinds of temporal relations: *state tables*, whose tuples are timestamped with time intervals $[t_B, t_E)$ and represent persistent facts that are true for each time in the interval, and *event tables*, whose tuples are timestamped with instants and represent facts that occur at a single instant of time [15, Ch. 16]. For a state table, we can pass from the abstract to its concrete representation by *coalescing* maximal sets value-equivalent tuples with consecutive timestamps $\{(r, t), (r, t + 1), (r, t + 2), \ldots, (r, t + n)\}$ into a single interval-timestamped tuple $(r, [t, t + n + 1))$ (so that there cannot exist value-equivalent tuples whose timestamps are adjacent intervals). The concrete representation of an event table coincides with its abstract representation. A concrete temporal database is composed of relations that can be stored using a finite number of tuples and, thus, can be implemented on a computer system.

As far as non-temporal tables are concerned, we assume they are virtually converted to temporal tables when they have to be interoperated with temporal

tables and streaming tables by using a suitable temporal conversion map [21]. In particular, we assume a non-temporal table R can be virtually converted to a temporal table as follows $(R)^T = \{(r, \tau) \,|\, r \in R, \tau \in \mathcal{T}\}$ (in its concrete representation, $(R)^T = \{(r, [0, \infty)) \,|\, r \in R\}$). The fact that a "valid always" timestamping is added to non-temporal data could look as an arbitrary choice. For example, a more conservative choice of assigning timestamps $\tau \in [now, \infty)$ timestamp, where *now* is the execution time of a temporal query referencing that data, was made in [21]. On the other hand, an assumption universally made in the temporal database field (e.g., which is at the base of interval-timestamping) is the *step function continuity assumption* [22], stating that temporal data after insertion keep their constant value until they are changed or deleted. Considering data found *now* in a non-temporal relation, we can assume its validity starts from a time t in the past corresponding to its insertion time. Since we have no information about the insertion time t, we make an unrestricted choice by setting $t = 0$. In this way, constant values stored in a non-temporal table can be interoperated with any data contained in (each snapshot of) a temporal table via a temporal algebra operator.

As far as streams are concerned, we adopt the definition of continuous data stream (or simply stream) provided in [23], that is a potentially infinite sequence of timestamped relational tuples having a fixed schema.

A *streaming table* is a special kind of relational table, first introduced in [13], that, unlike a standard relational table, is subject to continuous insertions of streaming data. In a streaming table, streaming data enter and turn historical by remaining stored for a user-defined long period, ideally forever. Any streaming table inherits the temporal nature of the data it stores. Specifically, it is modelled as an event table, where data are kept for a user-defined limited time span named *historical period*. As time goes by, we assume the oldest data exiting the historical period are subject to *vacuuming* [15, Ch. 23].

Definition 1 (Streaming Table). *A streaming table S with explicit 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. For each tuple $u \in S$, the expression $u.\nu$ denotes the tuple insertion number in S (i.e., $u.\nu = n$ iff u is the n^{th} inserted tuple). The content of S_{hp} at any time instant t, that is S^t_{hp}, is a set of tuples such that:*

- *for each tuple $u = (s, \tau)$ the timestamp $T(u) = \tau$ satisfies $\max(t - hp, 0) \leq \tau \leq t$; if positive, $t - hp$ represents the chronon preceding t by a time span of hp chronons.*
- *for any u_1 and u_2 in S^t_{hp}, $T(u_1) < T(u_2)$ implies $u_1.\nu < u_2.\nu$, that is u_1 was inserted before u_2.*

In practice, in order to implement this insertion semantics, systems cope with out-of-order and skewed inputs. Interested readers can refer to [24] for an in-depth discussion of this aspect. In this paper we assume an input manager that guarantees in-order tuple arrival. Notice that, in order not to burden the

OPTION_TRADES

OPTION	STOCK	CLASS	STRIKE	EXPIR	CONTRACTS	T
Opt245	Tesla	Put	300	Jan 2022	480	45
Opt237	Apple	Put	180	Jan 2022	360	44
Opt245	Tesla	Put	300	Jan 2022	350	42
Opt127	Google	Call	1100	Dec 2021	150	36
Opt666	Apple	Put	120	Nov 2021	15000	35
Opt127	Google	Call	1100	Dec 2021	200	33

LATEST_NEWS

ID	STOCK	TYPE	SOURCE	T
N1	Tesla	Merger and Acquisition	Wall Street Journal	[29, 43)
N8	Tesla	Dividend	Wall Street Journal	[43, ∞)
N5	Google	Fitch Rating	Financial Times	[32, ∞)
N2	Tesla	Dividend	Tesla Investor Relations	[27, ∞)
N3	Apple	Product Launch	Financial Times	[26, 34)
N4	Apple	Q4 Financial Results	Financial Times	[34, 38)
N7	Apple	Q4 Financial Results	Financial Times	[38, ∞)
N6	Google	Capital Increase	Wall Street Journal	[35, ∞)

Figure 1: Running example tables: The `OPTION_TRADES` streaming table (top) and the `LATEST_NEWS` temporal table (bottom)

notation, we neither represent ν as an explicit nor an implicit attribute in tuples, but we assume that its values can always be determined for each tuple stored in a streaming table (e.g., via a system-managed `ROW_NUMBER` function). Still for ease of notation, whenever possible in the following, we will use S in place of S_{hp}.

In the following, let $\mathcal{R}$ be the set of all temporal tables (also including the non-temporal ones) and $\mathcal{S}$ be the set of all streaming tables. Notice that $\mathcal{S} \subseteq \mathcal{R}$, as streaming tables are a special kind of temporal tables. Indeed, one effect of the insertion semantics is that timestamps in a streaming table are always bounded by the current time *now*, whereas temporal tables may also contain timestamps greater than *now* to represent proactively inserted future data.

3. Motivating Example and Proposed SQL Extensions

As an application example in the context of financial market surveillance, we consider tracking of *insider trading* activities and, more in general, market behavior analysis in response to available information. According to the US Securities and Exchange Commission (SEC), insider trading consists of the buying or selling of a security while in possession of nonpublic material information [25]. Examples of such material information include, for instance, a company being up/downgraded by a rating agency, unexpected revisions to earning results or projections, mergers and acquisitions news. Since insider trading undermines investor confidence in the fairness and integrity of the financial markets, control bodies like the SEC have the detection and prosecution of

insider trading violations as one of their market surveillance and enforcement priorities. Whereas early (i.e., when the material information is not yet publicly available) detection of insider trading patterns could allow to prevent frauds [26], their *a posteriori* (i.e., when the material information is of public domain) verification is most important for triggering and pursuing prosecution of illegal activities.

Sample tables for the application example are shown in Fig. 1. In particular, we assume that stock option trading data are available (cf., owing to their financial leverage, options are usually the elective way to capitalize on insider information) through the *streaming table* `OPTION_TRADES`, automatically fed by the stock market information system. The streaming table contains data concerning the negotiated option, the underlying stock, the trade timestamp (i.e., the implicit attribute T), the option class (call or put), strike price and expiration, and the number of contracts traded. For example, the top tuple of `OPTION_TRADES` in Fig. 1 represents the fact that, at time 45, a trade of 480 contracts was concluded on option Opt245, which is a put option on the Tesla stock with strike 300 and expiration January 2022. Such an option, bought to bet against the Tesla stock, will pay off if the stock at the end of January 2022 is below the strike price of $300. Moreover, we assume that relevant news are gathered from several publication sources (e.g., press releases and financial news stories) and stored in a *temporal table* `LATEST_NEWS`. Notice that such a table is a traditional temporal table that cannot be defined as an automatically fed stream indeed, as news information are manually inserted into the table by a panel of human experts after a press review activity. For each news, the stored information concerns the publication source, the involved stock and the news type. The pair `SOURCE-STOCK` is a key of the table, representing the narration of a stock behavior made by a news source. It is a state table, with tuples timestamped with a time interval, representing the period in which the tuple contents represent the latest available news within the narration, ranging from the publication time of the news itself to the publication time of a subsequent news, if any. For instance, according to the data in the first two tuples of `LATEST_NEWS` in Fig. 1, the behavior of the Tesla stock has been described by the Wall Street Journal through two news: the former, with `ID` N1 and concerning a merge and acquisition operation involving the stock, has been the latest available from time 29 to time 43; the latter, with `ID` N8 and concerning the payment of a stock dividend has become the latest from time 43 (the right timestamp boundary equal to ∞ means that the news is currently the latest available one). We assume the granularity of time in Fig. 1 is one second, so that the option trade data and news publication data in the sample table actually represent a 20-second timespan (from 26 to 45) of the application lifetime.

In the following, we propose to use a TSQL2-like temporal query language [15] and illustrate its use to express example queries on the database in Fig. 1. The first type of use of such data we want to exemplify in queries (Q1)–(Q3) concerns the use of a streaming table as it was a standard temporal table. The second type is indeed aimed at exemplifying in queries (Q4)–(Q6) the use of a temporal table in continuous queries.

(Q1) The first query retrieves the identifiers of the news and options involving the Apple stock traded during the period in which the news was the latest available one about Apple, and can be expressed as follows:

```
SELECT ID, OPTION                                          (Q1)
FROM LATEST_NEWS AS LN, OPTION_TRADES AS OT
WHERE LN.STOCK = OT.STOCK
  AND LN.STOCK = 'Apple'
```

In practice, this is a temporal join between the two tables (restricted to data involving the Apple stock). As in TSQL2, an overlap condition between the timestamps of the matching tuples is implicit and their intersection is assigned as timestamp to the joined tuples in the result. With the data in Fig. 1, the outcome of (Q1) is composed of the temporal tuples "(N4, Opt666 , 35)" and "(N7, Opt237 , 44)". This is a typical example of a *sequenced* query, as only data valid at the same time are retrieved together.

(Q2) The second query we consider retrieves the identifiers of the news superseded by another news with the same type within a narration (i.e., the superseded and the superseding news must be represented by tuples with consecutive timestamps, must have the same values of the source-stock pair to be part of the same narration and must also have the same value of the type as requested). The desired query can be expressed in a TSQL2-like language as follows:

```
SELECT SNAPSHOT LN1.ID                                     (Q2)
FROM LATEST_NEWS AS LN1 LN2
WHERE LN1.SOURCE = LN2.SOURCE AND LN1.STOCK = LN2.STOCK
  AND LN1.TYPE = LN2.TYPE AND LN1 MEETS LN2
```

The temporal predicate `LN1 MEETS LN2` is verified if the interval timestamps of `LN1` and `LN2` are consecutive (i.e., `END(LN1.T)=BEGIN(LN2.T)`) [15, Ch. 13]. With the data in Fig. 1, the outcome of (Q2) is composed of the snapshot (i.e., non-temporal) tuple "(N4)" as the only qualifying pair of news with the same type is N4-N7 in the narration of Apple by the Financial Time. This is an example of a *non-sequenced* query, as data belonging to different temporal snapshots have to be matched via the temporal selection predicate `MEETS`.

(Q3) In our TSQL2-like language proposal, hybrid sequenced/non-sequenced queries can also easily be expressed. In practice, the example which follows combines parts of (Q1) and (Q2) in a single `SELECT` statement, which retrieves the identifiers of the news and options concerning the same stock traded during the period in which the news was considered the latest, but which was then superseded by another news with the same type within the same narration. The resulting query is as follows:

```
SELECT LN1.ID, OPTION                                      (Q3)
FROM LATEST_NEWS AS LN1 LN2, OPTION_TRADES AS OT
WHERE LN1.SOURCE = LN2.SOURCE AND LN1.STOCK = LN2.STOCK
  AND LN1.TYPE = LN2.TYPE AND LN1 MEETS LN2
  AND LN1.STOCK = OT.STOCK
```

In practice, bindings of tuple pairs to variables LN1-LN2 are determined with a non-sequenced semantics as in (Q2) but then a temporal join is performed between tuples bound to variables LN1 and OT with a sequenced semantics as in (Q1). With the data in Fig. 1, the outcome is the temporal tuple "(N4, Opt666, 35)". Notice that we assume an implicit temporal join condition (overlap of timestamps) is always implied between all the relation names/variables whose attributes appear in the SELECT clause (i.e., LN1 and OT in (Q3)), while the intersection of their timestamps is assigned to the joined tuples in the result [1]. The ability to mix in the same query sequenced and non-sequenced execution semantics is a quality of the so-called *history-oriented* query languages, like TSQL2 and HoT-SQL [27]. Histories in a relation are sets of tuples with the same key representing *versions* of the same real-world objects [19]. History-oriented languages provide a friendly syntax that allows users, with a single SELECT statement, to combine different histories using a sequenced semantics and to select histories by means of conditions involving their component versions using a non-sequenced semantics.

Now we come back to the market surveillance application problems and consider examples where the proper nature of a streaming table is exploited by means of continuous queries. The TSQL2-like query syntax adopted so far will be extended to support continuous queries by means of constructs similar to the window functions previously proposed for streaming data query languages like Flink, SparkSQL or SQLStream [4, 12, 28].

(Q4) As a preliminary step, we consider a query that can be used to evaluate the impact of financial news on the option market behavior. To this purpose, for each news concerning a stock, we want to appreciate whether there is a significant difference between the trade volumes, of the options on that stock, recorded one week before and one week after the publication of the news. The higher the variation of the after volume with respect to the before volume is, the more impactful we can consider the news publication.

Using a continuous query, the weekly trade volumes can be computed as the sum of contracts traded in a one-week time window sliding over the OPTION_TRADES streaming table. To this purpose, two time-based sliding window expressions must be defined partitioned by stock (a partitioned window implies that a separate sliding window is actually constructed for each stock). For each sampling time point t, the first window W1A is defined to include all the stream tuples inserted in the week that follows t, whereas the second win-

[1] In TSQL2, the default option is an implicit temporal join condition between all the "argument relations" [15, Ch. 13], which could be interpreted as all the relations appearing either in the SELECT or in the FROM clause. As both interpretations were sometimes followed in exemplifying the use of the language [15, Ch. 3–4], the definition is actually ambiguous. If the second interpretation is chosen, in order to override the default and specify a hybrid semantics, an explicit VALID INTERSECT(LN1, OT) clause would be required in (Q3) to exclude LN2 from the intersection of timestamps to be assigned to the result. We prefer to adhere to the first interpretation and simplify the query syntax by considering as default the intersection of the timestamps of the relations involved in the SELECT clause only.

dow W1B is defined to include all the stream tuples inserted in the week that precedes t. Such tuples are then used to compute the aggregate function SUM over their CONTRACTS values; hence W1A.SUM(CONTRACTS) represents the option trade volume in the week after and W1B.SUM(CONTRACTS) represents the option trade volume in the week before. The expression 100*(W1A.SUM(CONTRACTS)/ W1B.SUM(CONTRACTS)-1) can then be used in the query target list to compute the percentage variation of the volume after with respect to the volume before. The desired continuous query can be specified as follows:

```
SELECT SOURCE, LN.STOCK,                                          (Q4)
     100*(W1A.SUM(CONTRACTS)/W1B.SUM(CONTRACTS)-1),
     VALID BEGIN(LN)
FROM OPTION_TRADES OVER
     (PARTITION BY STOCK RANGE 1 WEEK FOLLOWING) AS W1A,
     OPTION_TRADES OVER
     (PARTITION BY STOCK RANGE 1 WEEK PRECEDING) AS W1B,
     LATEST_NEWS AS LN
WHERE LN.STOCK = W1A.STOCK AND LN.STOCK = W1B.STOCK
SAMPLE INTERVAL 1 SECOND DELAY 1 WEEK
```

The sequenced semantics of the query requires a temporal join, which synchronizes the evaluation of the sliding window expressions with the news publication time. In fact, the timestamp assigned to the results is determined by the clause VALID BEGIN(LN), which extracts the publication time of the news, and must intersect it with the valid time of the sliding windows used in the SELECT clause. Considering the data in Fig. 1, for instance, for the news N8 concerning Tesla and published at time 43, the query (Q4) would have returned the temporal tuple "(Wall Street Journal, Tesla, 37.14 , 43)", as the before volume is 350 (given only by the trade on option Opt245 at time 42) and the after volume is 480 (given only by the trade on option Opt245 at time 45).

The SAMPLE clause is used to specify the sampling pattern which is used to execute the continuous query. In this way, the query is evaluated at every second: in practice, if there is some news published at that time, the sliding windows are evaluated to compute some result. The DELAY parameter forces the evaluation of the continuous query to occur one week after the time at which it should have been executed. Notice that such a query could not produce any results in real time, that is joining with the temporal relation LATEST_NEWS the data inserted into OPTION_TRADES at the time they enter the stream, because the news information is inserted by human experts retroactively, after an accurate press review job. For instance, the N4 news published at time 34 could have been inserted by the experts 4 days after its publication. Therefore, if the join was executed in real time, no results would be produced, as no news could be found in the relation LATEST_NEWS at the time of their publication. This is the reason for which standard approaches for joining a data stream with a temporal table (e.g., as considered for the Flink query language [12]) would probably fail. In order to wait for the news data to be eventually inserted, the trading data stream has to be persistently stored in a *streaming table*, from which it can then

be used retrospectively. In our example, we assume a safety delay of 7 days with respect to their publication time is sufficient to have all relevant news data correctly stored. Evaluating the continuous query (Q4) with such a sampling delay can thus produce the desired results.

For continuous queries, we also allow a syntactic variant already proposed for other streaming SQL extensions (e.g., SQLStream [4]), providing for a separate `WINDOW` clause where window expressions can be declared. Using this alternate syntax, query (Q4) becomes as follows:

```
SELECT SOURCE, LN.STOCK,                                          (Q4′)
     100*(W1A.SUM(CONTRACTS)/W1B.SUM(CONTRACTS)-1),
     VALID BEGIN(LN)
FROM OPTION_TRADES OVER W1A_WIN AS W1A,
     OPTION_TRADES OVER W1B_WIN AS W1B,
     LATEST_NEWS AS LN
WHERE LN.STOCK = W1A.STOCK AND LN.STOCK = W1B.STOCK
WINDOW W1A_WIN AS (PARTITION BY STOCK RANGE 1 WEEK FOLLOWING),
       W1B_WIN AS (PARTITION BY STOCK RANGE 1 WEEK PRECEDING)
SAMPLE INTERVAL 1 SECOND DELAY 1 WEEK
```

Moreover, there is a main difference between our SQL streaming extension and previous proposals. In previously defined streaming query languages (e.g., Flink, Spark or SQLStream) window expressions can only be used in the `SELECT` clause to produce query results by means of aggregate functions. In our proposed SQL extension, we move their declaration to the `FROM` clause as in (Q4) (or to the combination of `FROM` and `WINDOW` clauses as in (Q4′)). Then, we allow their use with aggregate functions in the `SELECT` but also in the `WHERE` clause, in order to enable window-based data selection. Although allowing aggregates in the `WHERE` clause could look like a "stretch" of the SQL standard syntax, once window expressions have been constructed, aggregates computed over them can play the role of flat table attributes and, thus, used for selection as plain attributes in the `WHERE` clause. Obviously, aggregates in the `WHERE` clause can only be used as attributes of window expressions. The theoretical underpinning of this syntax proposal is the straightforward translation of such a SQL query into the continuous algebra that will be introduced in Sec. 5.

Furthermore, query (Q4) (or (Q4′)) can also be used as the definition query of a (continuous) temporal view `NEWS_IMPACT(SOURCE, STOCK, VOL_VAR` $|T$`)`, which plays the role of a materialized collection of the results incrementally produced by the underlying continuous query. For each source-stock pair, the `NEWS_IMPACT` view collects the percentage variations after/before the news publication in the weekly trade volumes of the stock, as computed by the continuous query (Q4) (or (Q4′)). For instance, such a view could then be exploited in the query that follows:

```
SELECT SOURCE, AVG(VOL_VAR), COUNT(DISTINCT STOCK)                (Q4″)
FROM NEWS_IMPACT
WHERE VOL_VAR > 50
```

```
GROUP BY SOURCE
ORDER BY 2 DESC, 3 DESC
```

In (Q4″), first of all, only the tuples representing a significative impact for which the after/before weekly variation is above 50% are selected by the WHERE clause. Then the surviving NEWS_IMPACT tuples are grouped by publication source and aggregate functions are computed to measure the average after/before weekly variation and the number of stocks whose option trades have been significantly influenced by that source. The ORDER BY clause makes the query results to be displayed in the form of a ranking of the sources. Obviously, the returned data are biased by highly impactful news concurrently published by different sources, but could provide anyway a useful first-sight indication of how the publication sources could have been influential on the stock option market behavior.

As in traditional databases, the query (Q4″) can be answered by the SQL language processor by combining its specification with the definition query (Q4) (or (Q4′)) of the NEWS_IMPACT view, resulting in a continuous query (in Sec. 7, we will show its full algebra translation).

(Q5) Now we consider a continuous query that returns the days in which at least 10,000 options were traded in the afternoon trading session before, on the same day, the news of an Apple product launch was published. To this purpose, first of all we need a SAMPLE clause that specifies that the sampling points are spaced by one day and centered on noon (i.e., aligned to the start of the day plus 12 hours) and, as in (Q4), the continuous execution requires an evaluation delay of 1 week in order to have the news regularly stored. In order to determine the instant at which 10,000 options have possibly been traded in the afternoon, we can use a count-based sliding window N10KA defined to contain the 10,000 tuples inserted in the streaming table OPTION_TRADES after the beginning of the window. The time at which the insertion of the 10,000-th tuple occur, that is the timestamp of the last tuple belonging to the window, can be extracted from N10KA via the aggregate function LAST. The news of interest are selected if their type is product launch and they concern the Apple stock. The temporal join conditions N10KA.LAST < BEGIN(LN) AND BEGIN(LN) < N10KA.T + INTERVAL 12 HOUR ensure that the 10,000-trade target has been reached before the publication of the news and that the two events occurred on the same day. Finally, a function DAY() is assumed to be available to be used in the SELECT clause to extract the day component from the timestamp of N10KA (in TSQL2, an expression SCALE(N10KA.T AS DAY) should be used instead). The resulting continuous query can be specified as follows:

```
SELECT SNAPSHOT DAY(N10KA.T)                                    (Q5)
FROM OPTION_TRADES OVER (ROWS 10000 FOLLOWING) AS N10KA,
     LATEST_NEWS AS LN
WHERE LN.TYPE = 'Product Launch' AND LN.STOCK = 'Apple'
  AND N10KA.LAST < BEGIN(LN)
  AND BEGIN(LN) < N10KA.T + INTERVAL 12 HOUR
SAMPLE INTERVAL 1 DAY ALIGN TO START + 12 HOUR DELAY 1 WEEK
```

Query (Q5) requires a non-sequenced semantics to evaluate the temporal join conditions. Considering the data in Fig. 1, the news N3 published at time 26 announcing an Apple product launch can be found in the `LATEST_NEWS` table. Even assuming time 26 was in the afternoon, the corresponding day is not selected, as the total number of options traded after time 26 is 6 (i.e., all the tuples in the streaming table `OPTION_TRADES`), well below 10,000. Obviously, the "toy database" in the Figure only cover a few seconds of the application lifetime and the results of query (Q5) would be quite different with real data.

(Q6) Finally, we consider the insider trading detection problem. We assume that, in order to trigger an investigation procedure, an anomalous trading volume of an option, preceding the public release of some relevant news concerning the underlying stock, published in the 6 days following, needs to be identified. The investigation has to performed over the last six months of option trading data. As anomalous volume, we consider a daily trading volume ten times higher than the average daily volume over the past month. For instance, considering the data in Fig. 1, this could be the case of the Apple stock option Opt666 recorded in table `OPTION_TRADES`, which shows a seemingly very high volume with respect to the other traded options. Anyway, the data in Fig. 1 only cover about 20 seconds of the application lifetime and, thus, do not allow to test the functioning of the query, which would require at least data collected over a couple of months.

Following this specification, the suspect stock-day pairs deserving further investigations could be easily retrieved via a continuous query running at the beginning of each trading day and performing a non-sequenced temporal join between the relevant time window(s) of the streaming table `OPTION_TRADES` and the standard temporal table `LATEST_NEWS`. Notice that also such a detection query could not be executed in a "classical" stream management system, because, when the windows used for computing volumes are evaluated, the relevant news have not been published yet. Hence, stream data must be stored in a streaming table for our purpose, and time windows evaluated in a delayed mode, when all relevant news will be available. After the 1-week publication deadline, as in (Q4) and (Q5), one more delay week is added to let human experts to insert the news after their publication.

The required trade volumes per stock can be computed as the sum of contracts concluded on sliding windows defined over the `OPTION_TRADES` streaming table. To this purpose, two time-based sliding window expressions partitioned by stock and with different width are needed. For each sampling time point t, the first window `D1A` is defined to include all the stream tuples inserted in the day that follows t, whereas the second window `M1B` is defined to include all the stream tuples inserted in the month that precedes t. Such tuples are then used to compute the aggregate function `SUM` over their `CONTRACTS` values; hence `D1A.SUM(CONTRACTS)` represents the option trade volume in the day following t and `W1B.SUM(CONTRACTS)` represents the option trade volume in the month preceding t. Hence, the condition `D1A.SUM(CONTRACTS) > 10*M1B.SUM(CONTRACTS)/30` can be used in the `WHERE` clause to select the stocks for which the daily trade volume after is greater than 10 times the average daily trade volume in the month before (computed as $1/30^{\text{th}}$ of the total monthly

volume).

The desired query can be expressed in the proposed TSQL2-like language as follows:

```
SELECT D1A.STOCK VALID D1A.T                                    (Q6)
FROM OPTION_TRADES OVER
     (PARTITION BY STOCK RANGE 1 DAY FOLLOWING) AS D1A,
     OPTION_TRADES OVER
     (PARTITION BY STOCK RANGE 1 MONTH PRECEDING) AS M1B,
     LATEST_NEWS AS LN
WHERE D1A.STOCK = M1B.STOCK
  AND D1A.SUM(CONTRACTS) > 10*M1B.SUM(CONTRACTS)/30
  AND D1A.STOCK = LN.STOCK
  AND BEGIN(LN) >= D1A.T + INTERVAL 1 DAY
  AND END(LN) < D1A.T + INTERVAL 1 WEEK
SAMPLE INTERVAL 1 DAY ALIGN TO START DELAY 15 DAYS
HISTORICAL PERIOD 6 MONTH
```

The evaluation of the temporal selection condition `BEGIN(LN) >= D1A.T + INTERVAL 1 DAY AND END(LN) < D1A.T + INTERVAL 1 WEEK`, which ensures that the news has been published from the next day to 1 week following the anomalous daily volume, requires a non-sequenced semantics. The management of the validity of the two time windows and the production of temporal tuples as result (timestamped by `D1A.T`) require instead a sequenced semantics.

Notice that in the `SAMPLE` clause of (Q6) all the available options for defining the sampling pattern have been used. In particular, sampling times are defined as occurring one per day at 00:00 AM (i.e., aligned at the start of the day) and evaluation is delayed by 15 days. Notice that `ALIGN TO START` is the default option and, thus, could be omitted as done in query (Q3). The analysis is not performed using all the streaming data stored in `OPTION_TRADES` but only on the option trade data valid in the last six months, according to the `HISTORICAL PERIOD` specification.

Summing up, we can point out the following properties of the presented query examples concerning a market surveillance application:

- (Q1) requires the use of a streaming table in a temporal query (so that streaming data can be used off-line as belonging to a standard temporal table); the temporal query execution requires a sequenced semantics.
- (Q2) the temporal query execution requires a non-sequenced semantics.
- (Q3) requires the use of a streaming table in a temporal query; the temporal query execution requires a hybrid sequenced/non-sequenced semantics.
- (Q4) requires the use of a temporal table in a continuous query; the continuous query execution requires the use of a streaming table for delayed execution; the temporal query execution requires a sequenced semantics.

- (Q5) requires the use of a temporal table in a continuous query; the continuous query execution requires the use of a streaming table for delayed execution; the temporal query execution requires a non-sequenced semantics.
- (Q6) requires the use of a temporal table in a continuous query; the continuous query execution requires the use of a streaming table for delayed execution; the temporal query execution requires a hybrid sequenced/non-sequenced semantics.

The approach presented in this work is aimed at fulfilling all these requirements.

4. Querying Streaming Tables with OTQs: The Temporal Algebra $\mathcal{TA}^\star$

$\mathcal{TA}^\star$ is the temporal algebra we propose to specify queries over streaming, temporal, and standard tables. It extends the temporal algebra $\mathcal{TA}$ with the controlled use of standard relational operators and supports an hybrid semantics that is necessary to specify expressive queries like those shown in Sec. 3. $\mathcal{TA}^\star$ is used to specify OTQs where streaming tables are dealt with as event table and is the basis for the continuous temporal algebra $\mathcal{CTA}$.

4.1. The Temporal Algebra $\mathcal{TA}$

The starting point of our proposal is the temporal algebra $\mathcal{TA}$ presented in [16], equipped with the following primitive temporal operators: selection σ^T, projection π^T, Cartesian product $\times^T$, union $\cup^T$, difference $-^T$, and grouping ϑ^T. Each of these temporal operators is a generalization of a standard relational operator where the T-superscript does not appear (derived operators, like the join $\bowtie^T$, can also be considered as usual).

The semantics of the $\mathcal{TA}$ operators is shown in Table 1, where, if (r, τ) is a tuple of a temporal relation with explicit schema $R(X)$, we consider r (with schema X) its explicit part and τ the tuple timestamp; $\mathcal{P}$ is the set of all well-defined predicates p over the explicit attributes X of R; $B \subset X$ is a subset of schema attributes; $\mathcal{F} = \{f_1, \ldots, f_k\}$ is a set of aggregation functions (e.g., `COUNT`, `SUM`, `AVG`, ...); $\circ$ is a tuple concatenation operator.

Notice that the result of the Cartesian product applied to at least one streaming table returns a streaming table, the union operator returns a streaming table when both operands are streaming tables and the difference operator returns a streaming table when the first operand is a streaming table. The grouping operator performs aggregation by groups. More precisely, it partitions the tuples of a temporal table into groups according to their common values on attributes in B, and it applies the aggregate functions in $F \subseteq \mathcal{F}$ to each group. Finally, notice that, differently form the one defined in [16], τ_t is an "extended" timeslice operator that maintains the timestamps, such that its outcome is still a temporal relation representing the *snapshot* valid at time t.

Operator **Signature** **Operator semantics**
Selection $\sigma^T : \mathcal{R} \times \mathcal{P} \to \mathcal{R}$ $\sigma^T_p(R) := \{(r,\tau) \,\vert\, (r,\tau) \in R \wedge p(r)\}$
Projection $\pi^T : \mathcal{R} \times 2^{\mathcal{A}} \to \mathcal{R}$ $\pi^T_B(R) := \{(r.B,\tau) \,\vert\, (r,\tau) \in R\}$
Cartesian product $\times^T : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ $R_1 \times^T R_2 := \{(r_1 \circ r_2,\tau) \,\vert\, (r_1,\tau) \in R_1 \wedge (r_2,\tau) \in R_2\}$
Union $\cup^T : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ $R_1 \cup^T R_2 := \{(r,\tau) \,\vert\, (r,\tau) \in R_1 \vee (r,\tau) \in R_2\}$
Difference $-^T : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ $R_1 -^T R_2 := \{(r,\tau) \,\vert\, (r,\tau) \in R_1 \wedge (r,\tau) \notin R_2\}$
Grouping $\vartheta^T : \mathcal{R} \times 2^{\mathcal{A}} \times 2^{\mathcal{F}} \to \mathcal{R}$ ${}_B\vartheta^T_F(R) := \{(r.B \circ Z,\tau) \,\vert\, (r,\tau) \in R \wedge r_g = \{(r',\tau) \in R \,\vert\, r'.B = r.B\}$ $\wedge Z = (f_1(r_g),\ldots,f_h(r_g))\}$
Extend $\varepsilon : \mathcal{R} \times \mathcal{A} \to \mathcal{R}$ $\varepsilon_U(R) := \{(r \circ U,\tau) \,\vert\, (r,\tau) \in R \wedge U = \tau\}$
Timeslice $\tau : \mathcal{R} \times \mathcal{T} \to \mathcal{R}$ $\tau_t(R) := \{(r,\tau) \,\vert\, (r,\tau) \in R \wedge \tau = t\}$

Table 1: $\mathcal{TA}$ operator semantics

The definition of these operators is borrowed from [16, Sec. 2 and Appendix A] and, thus, $\mathcal{TA}$ supports a sequenced semantics in terms of extended snapshot reducibility. In particular, an extend operator ε_U is available, which can be used to copy the timestamp T to the additional explicit attribute U. As defined in [16], *extended* snapshot reducibility refers precisely to the support of queries that reference in such a way the original timestamps.

4.2. Overcoming the Limitations of a Sequenced Temporal Algebra

The work [16] has made a breakthrough contribution to the development of the temporal database technology, by showing how an interval-based temporal database can be implemented on top of an existing traditional relational database, by enabling its query engine (with all its optimization machinery) to execute sequenced queries by means of simple kernel extensions.

However, the execution of sequenced queries as it can be done with the $\mathcal{TA}$ algebra is only a part of the duties required to a temporal database. In fact, the provision of a query language with a strict snapshot-reducible semantics only is almost useless for the development of temporal applications as it lacks support of real temporal queries. Let us assume for simplicity a time granularity of one day

and consider a temporal algebra expression Q^T built with the $\sigma^T, \pi^T, \times^T, \cup^T, -^T$ base operators only. Let Q be a (non-temporal) relational algebra expression obtained from Q^T by replacing each temporal operator with its non-temporal counterpart. Being a temporal database a timestamped collection of snapshot databases, the results of the execution of Q^T is, by definition, the timestamped collection of the results that can be obtained, day by day, by executing Q on the timeslice of the relations in the database valid at that day (i.e., the results only depend on the database snapshot valid at that day). Hence, there is no need to have a temporal database and a temporal query language to extract the data to build such a result: it is sufficient to have a non-temporal database and execute the same non-temporal query Q every day on the current database instance. A temporal relation is only needed to collect in a persistent way the query results obtained day by day, timestamping them with the current date. On the contrary, a real temporal database with a real temporal query language is needed to execute queries that involve, at the same time, data belonging to different snapshots, like (Q2), (Q3), (Q5) and (Q6) in Sec.3. We call *real* temporal queries such queries, which are typical non-sequenced queries. For instance, a query like (Q2), which selects financial news that have been superseded or rectified (i.e., news for which the next news, published on the same source and concerning the same stock, has the same type), is a typical real temporal query that needs a real temporal database to be answered. In fact, the answer requires the comparison of the news belonging to two consecutive snapshots to be effected. A snapshot-reducible temporal algebra is completely useless to this purpose, as it is unable to mix data belonging to different snapshots.

Notice that the necessity of operators with a non-sequenced semantics to express real temporal queries has been rather underestimated in the recent literature regarding temporal algebras. For example, in the the concise survey [29], temporal algebras are classified as based on snapshot evaluation (viz., sequenced semantics) or on traditional evaluation (viz., non-sequenced semantics). Then, in the exemplification of algebras with a snapshot evaluation, the selection operator is defined as $\sigma^t_F(R) = \sigma_F(R_t)$ for all $t \in \mathcal{T}$, where R_t is the snapshot of R at time t and the selection formula F is said to include traditional predicates as well as temporal predicates like *Before, After, Overlaps*, etcetera. Allowing for temporal predicates in F seems clearly aimed at supporting real temporal queries, which is though incompatible with the sequenced semantics: evaluation of temporal predicated in F by $\sigma_F(R_t)$ is impossible as only one snapshot valid at t is available! Hence, only temporal algebras based on traditional or hybrid evaluation are able to express real temporal queries.

The authors of [16] partially acknowledged the limitations of a purely snapshot-reducible algebra and introduced, to overcome them, an extend operator ε and a careful definition of the temporal aggregate operator ϑ^T. The extend operator does *timestamp propagation*, that is it copies the values of the tuple timestamp into an explicit attribute in order to make the timestamp values accessible to the manipulation by other operators. The temporal aggregate operator is a snapshot-reducible operator that is able to produce a result by aggregating data belonging to more than one snapshot (i.e., to all the snapshots

spanned by interval timestamps). In particular, they showed in [16] how the use of the extend operator is needed in order to correctly compute the values of temporal aggregates from tuples adopting interval-based timestamping by *scaling* attribute values (in a way depending on their semantics). Attribute scaling is made necessary by the adjustment of intervals which is used to execute snapshot-reducible temporal queries on a traditional database kernel.

However, the support of real temporal queries in [16] is just limited to the evaluation of such temporal aggregates (and the use of the extend operator is basically aimed at the correct computation of such aggregates via the scaling mechanism). For instance, our example query (Q2) retrieving superseded news cannot be expressed in the temporal algebra proposed in [16]. In fact, in order to express such a query on data stored in the `LATEST_NEWS` relation, pairs of tuples with consecutive interval timestamps have to be found and the attributes `STOCK`, `TYPE`, `SOURCE` in them compared (i.e., a non-sequenced temporal self-join has to be actually executed). Assuming N_1 and N_2 are two aliases for the `LATEST_NEWS` relation, the desired result can be obtained by means of the following expression written in the plain relational algebra (accessing timestamps as if they were explicit attributes):

$$\pi_{N_1.\texttt{ID},N_1.\texttt{T}}(\sigma_{\substack{N_1.\texttt{SOURCE}=N_2.\texttt{SOURCE}\wedge N_1.\texttt{STOCK}=N_2.\texttt{STOCK}\\ \wedge N_1.\texttt{TYPE}=N_2.\texttt{TYPE}\wedge N_1.T\ \texttt{MEETS}\ N_2.T}}(N_1 \times N_2))$$

Using the $\mathcal{TA}$ temporal algebra, the timestamps of N_1 and N_2 can even be propagated via the extend operator to become available as explicit attributes as follows: $M_1 = \varepsilon_U(N_1)$ and $M_2 = \varepsilon_U(N_2)$. But there is no way to test the temporal join condition $M_1.U$ `MEETS` $M_2.U$ with the snapshot-reducible operators of the temporal algebra. For instance, the expression:

$$\sigma^T_{\substack{M_1.\texttt{SOURCE}=M_2.\texttt{SOURCE}\wedge M_1.\texttt{STOCK}=M_2.\texttt{STOCK}\\ \wedge M_1.\texttt{TYPE}=M_2.\texttt{TYPE}\wedge M_1.U\ \texttt{MEETS}\ M_2.U}}(M_1 \times^T M_2)$$

always produces an empty result, owing to the snapshot-reducible semantics of the temporal Cartesian product operator, which only combines tuples having an overlapping timestamp (if $M_1.T$ and $M_2.T$ overlap, also their copies $M_1.U$ and $M_2.U$ overlap and, thus, $M_1.U$ `MEETS` $M_2.U$ is always false). A snapshot-reducible temporal algebra, by definition, can only interoperate data belonging to the same snapshots, whereas our sample query require data belonging to different snapshots to be interoperated.

This deficiency of all the snapshot-reducible temporal algebras, including that of [16], can be overcome in a simple way by allowing the use of non-temporal algebra operators in temporal algebra expressions. The only requirement for their use, in order to guarantee the closure of the resulting extended temporal algebra, is that they are used, possibly in combination, to produce macro expressions that produce anyway correct temporal relations (whereas their subexpressions may do not). For instance, being M_1 and M_2 the temporal algebra expressions defined above, $\sigma_{\texttt{STOCK}\neq\texttt{'Tesla'}}(M_1)$ and $M_2 \times \pi_{\texttt{SOURCE}}(M_2)$ are correct uses of the non-temporal operators, since their result are temporal relations, whereas $\pi_{\texttt{STOCK}}(M_1)$ and $M_1 \times M_2$ are not (the former expression produces a relation without a

$E_{\mathcal{TA}^\star}$	$\rightarrow$	Q^T
Q^T	$\rightarrow$	R^T, $\mathsf{op}_1^T(Q^T)$, $R^T\ \mathsf{op}_2^T\ Q^T$, $Q^T \times Q$, $Q \times Q^T$
op_1^T	$\rightarrow$	π_X^T, σ_F^T, ${}_B\vartheta_F^T$, ε_U, τ_t
op_2^T	$\rightarrow$	$\times^T$, $\cup^T$, $-^T$
R^T	$\rightarrow$	temporal table, streaming table,
		$(Q)^T$ (non-temporal expression virtually converted to temporal)
Q	$\rightarrow$	R, $\mathsf{op}_1(Q)$, $Q\,\mathsf{op}_2\,R$,
		$\pi_Y(Q^T)$ (Y is a subset of the explicit attributes of Q^T)
op_1	$\rightarrow$	π_X, σ_F, ${}_B\vartheta_F$
op_2	$\rightarrow$	$\times$, $\cup$, $-$
R	$\rightarrow$	non-temporal table

Table 2: Complete syntax of a $\mathcal{TA}^\star$ expression $E_{\mathcal{TA}^\star}$

timestamp and the latter would produce a relation with two timestamps). Hence, the expression $M_1 \times \pi_{\texttt{ID},\texttt{STOCK},\texttt{TYPE},\texttt{SOURCE},U}(M_2)$ represents a correct use of non-temporal operators, as the result is a correct temporal relation (basically the Cartesian product between the relation M_1 and a relation containing the non-temporal part of the tuples of M_2, timestamped with the timestamps of M_1).

4.3. The Temporal Algebra $\mathcal{TA}^\star$

To overcome the limitations of a sequenced temporal algebra, we propose the temporal algebra $\mathcal{TA}^\star$ that seamlessly combine the temporal relational algebra $\mathcal{TA}$ with the standard relational algebra. The former provides implicit access to time over temporal tables while the latter can be used to query standard tables, including temporal tables where timestamp are made explicit via the extend operator ε_U.

The full syntax of any expression $E_{\mathcal{TA}^\star}$ in $\mathcal{TA}^\star$ is shown in Tab. 2.

In particular, $\mathcal{TA}^\star$ provides for a hybrid semantics leveraging the algebras it combines. In fact, both sequenced and non-sequenced semantics are supported in the expression and during the execution of a query, provided that it is clear which parts of the language must be evaluated with a sequenced or non-sequenced semantics. In fact, $\mathcal{TA}^\star$ extends $\mathcal{TA}$ to the support of the non-sequenced part of queries, which can be reduced to the minimum necessary, by means of standard relational operators in subexpressions (e.g., to join data belonging to different snapshots).

Specifically, according to the $\mathcal{TA}^\star$ production rules $Q^T \rightarrow Q^T \times Q$ and $Q^T \rightarrow Q \times Q^T$, standard algebraic sub-expressions $Q_1, \ldots, Q_n$ can be combined with one temporal sub-expression Q^T only by means of the $\times$ and the temporal result inherits the timestamps from the streaming/temporal operand Q^T.

On the other hand, any standard algebraic sub-expression Q can always be explicitly converted to a temporal one through the $\mathcal{TA}^\star$ expression $(Q)^T$ and

eventually be argument of $\mathcal{TA}$ operators according to the Q^T production rules.

In practice, like the TSQL2 language, our proposed $\mathcal{TA}^\star$ provides a snapshot-reducible algebra template allowing for *islands* of non-reducibility inside. Specifically, The first five production rules define the snapshot-reducible template, whereas the last four ones define the islands of non-reducibility.

Notice that the ATSQL language [17] can be used in a similar way by nesting non-sequenced statements into sequenced statements (e.g., to translate temporal logic expressions as shown in [30]); however, the resulting statements with possibly multiple nestings come out unnecessarily complex even for quite simple temporal SPJ queries, with respect to the TSQL2-like syntax we propose for $\mathcal{TA}^\star$ and for which a single `SELECT` statement suffices as shown in Sec. 3. To the best of our knowledge, no temporal algebras which can be used with a hybrid semantics were formalized before $\mathcal{TA}^\star$. Put in another way, $\mathcal{TA}^\star$ can also be used to give a complete hybrid semantics to ATSQL queries where mutiple SEQ and NSEQ nesting levels are employed.

In order to highlight its being an evolutionary step with respect to $\mathcal{TA}$ (for which the expression "extended snapshot reducibility" was coined), we say that $\mathcal{TA}^\star$ supports an *extended** *snapshot reducibility* semantics.

In the following, we assume readers are familiar with the syntax and semantics of relational queries[2] and we only provide a concise yet informal semantics of $\mathcal{TA}^\star$.

Definition 2 ($\mathcal{TA}^\star$ Semantics). *Given a $\mathcal{TA}^\star$ expression*

$$E_{\mathcal{TA}^\star} = Q^T(S_1, \ldots, S_n, R_1^T, \ldots, R_m^T, Q_1, \ldots, Q_\ell)$$

over n streaming tables $S_1, \ldots, S_n$, with $n \geq 1$, m temporal tables $R_1^T, \ldots, R_m^T$, with $m \geq 0$, and ℓ non-temporal sub-expressions $Q_1, \ldots, Q_\ell$, with $\ell \geq 0$, its semantics is the result of the semantics of the involved temporal operators for which extended* snapshot reducibility holds and where each Q_i operand is virtually converted to temporal, that is for each $t \in \mathcal{T}$:

$$\tau_t(E_{\mathcal{TA}^\star}) = Q^T(\tau_t(S_1), \ldots, \tau_t(S_n), \tau_t(R_1^T), \ldots, \tau_t(R_m^T), \tau_t((Q_1)^T), \ldots, \tau_t((Q_\ell)^T))$$

$\mathcal{TA}^\star$ expressions are used to specify both OTQs and CQs over streaming, temporal and standard tables. The two kinds of queries differs in the access paradigm for streaming tables and in the semantics. Streaming tables in OTQs are accessed as event tables and the semantics of any OTQ is the semantics of the corresponding $\mathcal{TA}^\star$ expression.

Example. Coming back to the last example of Sec. 4.2, it is worth noting that the algebraic expression testing the temporal join condition $M_1.U$ `MEETS` $M_2.U$ can be expressed in $\mathcal{TA}^\star$ as follows:

$$\sigma^T_{M_1.\texttt{SOURCE}=M_2.\texttt{SOURCE} \wedge M_1.\texttt{STOCK}=M_2.\texttt{STOCK} \wedge M_1.\texttt{TYPE}=M_2.\texttt{TYPE} \wedge M_1.U \texttt{ MEETS } M_2.U}(M_1 \times \pi_{\texttt{ID},\texttt{STOCK},\texttt{TYPE},\texttt{SOURCE},U}(M_2))$$

[2] Interested readers can refer to [31] for an in-depth study.

Streaming tables in CQs are instead accessed as data streams. $\mathcal{TA}^\star$ is then extended with windowing operators giving rise to the continuous temporal algebra $\mathcal{CTA}$ and its continuous evaluation semantics that are presented in Sections 5 and 6. Examples of legal use of $\mathcal{TA}^\star$ expression deriving from the translation of the queries presented in Sec. 3 will be given in Sec. 7.

As far as the expressiveness of $\mathcal{TA}^\star$ is concerned, notice that the possibility of accessing the timestamp values and of using them as if they were explicit attributes for all intents and without limitations via non-temporal algebra operators, makes our $\mathcal{TA}^\star$ language as expressive as the Temporal Relational Calculus (TRC) [32], which has been proven to be strictly more expressive than a language based on First-Order Temporal Logic (FOTL), with "implicit" access to time like the temporal algebras with snapshot evaluation [29], including $\mathcal{TA}$ [16]. For example, the TRC "snapshot-equality" query, which was proven in [33, 34] that cannot be expressed in FOTL, can be expressed in our extended temporal algebra as shown in Appendix A. These features are summarized by the Lemma that follows.

Lemma 1. *The temporal algebra $\mathcal{TA}^\star$ has the same expressiveness of the Temporal Relational Calculus (TRC) [32] and, thus, is strictly more expressive than a purely snapshot-reducible temporal algebra or an extended snapshot-reducible temporal algebra like $\mathcal{TA}$.*

Proof. TRC is the two-sorted version of first-order logic over a data domain D and a time domain T (2-FOL). Its syntax over a database schema $\rho = \{R_1, \ldots, R_k\}$ is defined by the grammar rule:

$$Q \rightarrow R(t_i, x_{i_1}, \ldots, x_{i_k}) \mid t_i < t_j \mid x_i = x_j \mid Q \wedge Q \mid \neg Q \mid \exists x_i.Q \mid \exists t_i.Q$$

It is worth noting that it refers to the schemas of atomic relations $R(t, x_1, \ldots, x_n)$ that temporally extend the relation symbols of a database schema and that all references to time are explicit.

Given the equivalence between the calculus under active domain semantics and the relational algebra, we can straightforwardly state that any 2-FOL query over any temporal relation having an equivalent representation R^T in $\mathcal{TA}^\star$ can be translated in $\mathcal{TA}^\star$ by means of the algebraic expression $\pi_Y(\varepsilon_U(R^T))$ and the standard relational symbols according to the fragment of $\mathcal{TA}^\star$ syntax headed by Q production rule in Tab. 2. □

However, unlike TRC, $\mathcal{TA}^\star$ is a temporal algebra that overcomes the limitations of a sequenced temporal algebra by supporting a rich set of temporal and non-temporal operators and the result of any $\mathcal{TA}^\star$ expression is a temporal relation evaluated under the extended* snapshot reducibility semantics. The advantage of this approach is twofold: powerful OTQs can be specified through a simple yet intuitive SQL-based language like the one shown in Sec. 3 and its implementation can largely benefit from temporally-aware query processing optimizations. Specifically, during the optimization of a $\mathcal{TA}^\star$ expression,

subexpressions Q_i have to be processed separately (also adopting custom techniques for the optimized execution of non-sequenced temporal joins). Then, $\mathcal{TA}^\star$ expressions only containing R^T, which can be either temporal tables, or streaming tables, or the result of the evaluation of a subexpression Q_i, can then be processed and optimized as described in [16].

Notice that preserving the sequenced semantics as much as possible, while allowing for minimal non-sequenced subexpressions to enrich the expressiveness of the language, was an essential design requirement for $\mathcal{TA}^\star$. In fact, the same expressiveness could even be obtained by using the non-temporal algebra over tables with explicit columns used as data timestamps, that is using a "poor man's approach" to the management of time-varying data. But this would mean to completely give up the advantages of the adoption of a temporal data model and query language, which could result in struggling with overly complex and inefficient queries, as shown for example in [11], also to accomplish simple tasks (e.g., enforcement of temporal constraints). Deprecation of such an approach was indeed the spark that ignited the powders of temporal database research.

5. Querying Streaming Tables with CQs: the Continuous Temporal Algebra $\mathcal{CTA}$

A continuous query is a query that is issued once, and then logically runs continuously until terminated by the user. Any streaming table S referenced in a continuous query must be accessed through a sliding window expression $w(S)$ that specifies the boundaries of the range of tuples in S to be used for query evaluation. The continuous temporal algebra $\mathcal{CTA}$ we propose leverages $\mathcal{TA}^\star$ and extends the set of windowing operators usually adopted in the streaming context [35], by generalizing their semantics to generate and operate on, possibly through aggregation, *sequences* of windows instead of single windows. The following definition introduces the notion of algebraic expression in the continuous temporal algebra $\mathcal{CTA}$.

Definition 3 ($\mathcal{CTA}$ Expression). *An algebraic expression*

$$E_{\mathcal{CTA}} = Q^T(w_1(S_1), \ldots, w_n(S_n), R_1^T, \ldots, R_m^T, Q_1, \ldots, Q_\ell)$$

in the continuous temporal algebra $\mathcal{CTA}$ is a $\mathcal{TA}^\star$ expression Q^T over n streaming tables $S_1, \ldots, S_n$, with $n \geq 0$, each S_i accessed through a corresponding sliding window expression w_i, m temporal tables $R_1^T, \ldots, R_m^T$, with $m \geq 0$, that may also include streaming tables accessed as event tables, and $Q_1, \ldots, Q_m$ non-temporal sub-expressions.

The complete syntax of a general $\mathcal{CTA}$ expression $E_{\mathcal{CTA}}$ is the same as in Table 2 (with $E_{\mathcal{CTA}}$ replacing $E_{\mathcal{TA}^\star}$ in the first place) augmented by the production rule $R^T \rightarrow w(S)$, where S is a streaming table.

Furthermore, in order to have a well-defined semantics of continuous queries, we must add to the $\mathcal{CTA}$ syntax the constraint that non-temporal Q subexpressions cannot contain sliding window expressions (they may contain anyway

streaming tables, which are dealt with as temporal tables). This means to forbid that Q^T subexpressions appearing in $\pi_Y(Q^T)$ expressions (where Y is a subset of the explicit attributes of Q^T) may contain sliding window expressions $w(S)$.

5.1. $\mathcal{CTA}$ Sliding Window Expressions

In order to support sliding window expressions (w) over streaming tables, $\mathcal{CTA}$ introduces two classes of operators: one includes *sliding window operators* (ω) and the other includes *window flattening and aggregation operators* (α). Sliding window expressions are then defined in a modular way as the composition of one of the former with one of the latter operators: $w = \alpha \circ \omega$. Both classes of operators include one *standard* and one *partitioned* version of each operator. Standard operators operate over streaming tables whereas partitioned operators operate over streaming table partitions, that is sets of streaming tables.

In the formalization of ω and α operators, in addition to the streaming table datatype $\mathcal{S}$, we will also make use of the following exotic datatypes:

set of streaming tables: $2^{\mathcal{S}} = \{S \mid S \in \mathcal{S}\}$

streaming table of streaming tables: $\mathcal{S}^* = \{(S, \tau) \mid S \in \mathcal{S}, \tau \in \mathcal{T}\}$

set of streaming tables of streaming tables: $2^{\mathcal{S}^*} = \{S^* \mid S^* \in \mathcal{S}^*\}$

For the sake of clarity, we give an intuition of these two classes of operators that work in a complementary fashion. All sliding window operators ω generate a timestamped sequence of portions of the input streaming table according to a sliding window specification. The new timestamps assigned to each portion correspond to times at which the windows are evaluated (whereas each portion maintains as "temporal provenance" witnesses the original timestamps of the tuples selected from the input streaming table). Specifically, the output of base sliding window operators is formally a streaming table of streaming tables whereas the output of partitioned sliding window operators is formally a set of streaming tables of streaming tables. Window flattening operators and window aggregation operators α reduce the result of a sliding window operator to a streaming table, in any case.

In the following, we provide the formal definition and semantics of the operators involved in the specification of sliding window expressions. For ease of notation, we start by introducing some utility operators. The definition of $\mathcal{CTA}$ operators and the specification of their combinations in legal expressions follow.

5.2. Utility Operators

The following operators provide useful transformations of streaming tables, which are indeed used for the definition of $\mathcal{CTA}$ operators.

Definition 4 (Time-Substreaming). *The time-substreaming operator* $\mathrm{Sub}^T : \mathcal{S} \times \mathcal{T}^2 \to \mathcal{S}$ *restricts a streaming table* $S \in \mathcal{S}$ *to only tuples such that their timestamp belongs to an interval* $[t_1, t_2]$*:*

$$\mathrm{Sub}^T_{[t_1,t_2]}(S) := \{u \mid u \in S \wedge t_1 \leq T(u) \leq t_2\}.$$

For instance, the expression $\mathrm{Sub}^{T}_{[\text{2016-01-01 00:00:00, 2016-12-31 23:59:59}]}(\texttt{OPTION\_TRADES})$ builds from OPTION_TRADES a streaming table containing the trades executed in 2016 only.

Definition 5 (Order-Substreaming). *The order-substreaming operator* $\mathrm{Sub}^{\nu} : \mathcal{S} \times \mathcal{T}^2 \to \mathcal{S}$ *restricts a streaming table* $S \in \mathcal{S}$ *to only tuples such that their insertion number belongs to an interval* $[\nu_1, \nu_2]$*:*

$$\mathrm{Sub}^{\nu}_{[\nu_1,\nu_2]}(S) := \{u \,|\, u \in S \wedge \nu_1 \leq u.\nu \leq \nu_2\}.$$

For instance, the expression $\mathrm{Sub}^{\nu}_{[201,\ 300]}(\texttt{OPTION\_TRADES})$ builds from OPTION_TRADES a streaming table containing the 100 trades inserted after the first 200 ones.

Definition 6 (Streaming Table Partition). *The partitioning operator* $\zeta : \mathcal{S} \times 2^{\mathcal{A}} \to 2^{\mathcal{S}}$ *partitions the streaming table* $S \in \mathcal{S}$ *into a set of streaming tables containing the tuples of* $\mathcal{S}$ *grouped by their attributes in* B *(as for the SQL group by mechanism, a partition is created for each combination of the values of the attributes* $B_1, \ldots, B_k$ *in* B*):*

$$\zeta_B(S) := \{S' \,|\, \exists (s,\tau) \in S \wedge S' = \{(s',\tau') \,|\, (s',\tau') \in S \wedge s'.B = s.B\}\}.$$

For instance, the expression $\zeta_{\texttt{CLASS}}(\texttt{OPTION\_TRADES})$, partitioning OPTION_TRADES with respect to the values of CLASS, evaluates to a set containing two streaming tables, one containing all the trades on call-type options and the other containing all the trades on put-type options present in OPTION_TRADES.

5.3. Sliding Window Operators (ω)

In accordance with commonly adopted definitions of sliding windows [35], sliding window operators in $\mathcal{CTA}$ are time-based and count-based. Standard sliding window operators apply to a streaming table S and formally generate a streaming table of streaming tables. Definitions of (backward and/or forward) standard sliding window operators are provided below.

Definition 7 (Time-based Sliding Window). *The time-based sliding window operator* $w^{\text{time}} : \mathcal{S} \times \mathcal{I}^2 \to \mathcal{S}^*$ *over a streaming table* $S \in \mathcal{S}$ *creates a streaming table of streaming tables with a window size of duration* $d_1 \geq 0$ *before and* $d_2 \geq 0$ *after the timestamps around which it is computed:*

$$w^{\text{time}}_{[d_1,d_2]}(S) := \{(S',\tau) \,|\, \exists \tau \in \mathcal{T} \wedge S' = \mathrm{Sub}^{T}_{[\tau-d_1,\tau+d_2]}(S)\}.$$

For instance, the expression $w^{\text{time}}_{[1\text{week},0]}(\texttt{OPTION\_TRADES})$ defines a streaming table whose tuples, for each time τ, are in turn streaming tables extracted from OPTION_TRADES by restricting it to the 1-week wide time window preceding τ.

Definition 8 (Count-based Sliding Window). *The count-based sliding window operator* $w^{\text{count}} : \mathcal{S} \times \mathbb{N}^2 \to \mathcal{S}^*$ *over a streaming table* $S \in \mathcal{S}$ *creates a streaming table of streaming tables with a window containing the temporally closest* $n_1 \geq 0$ *tuples valid before and the temporally closest* $n_2 \geq 0$ *tuples valid*

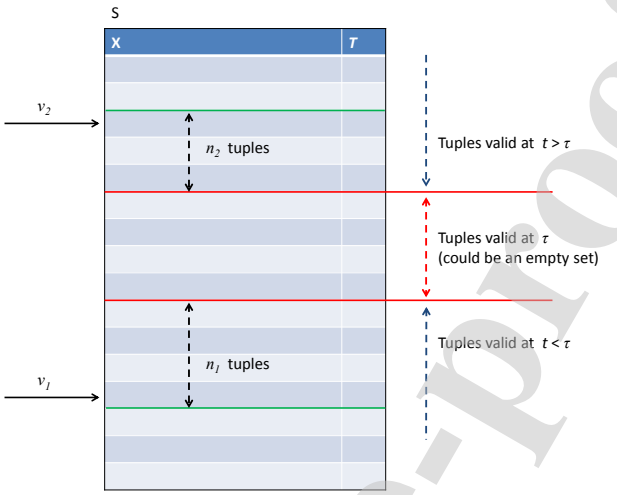


Figure 2: Construction of (S', τ) by $w^{\text{count}}_{[n_1,n_2]}(S)$. The tuples to be included in S' have an insertion number $\nu \in [\nu_1, \nu_2]$ in S.

after the time around which it is computed (the tuples possibly valid at the time around which it is computed are also included):

$$w^{\text{count}}_{[n_1,n_2]}(S) := \{(S', \tau) \,|\, \exists \tau \in \mathcal{T} \wedge S' = \text{Sub}^{\nu}_{[\nu_1,\nu_2]}(S) \\ \wedge \nu_1 = \text{prev}(S, \tau, n_1) \wedge \nu_2 = \text{next}(S, \tau, n_2)\}.$$

In the definition of (S', τ) above, the function $\text{prev}(S, \tau, n_1)$ computes the insertion number ν_1 of the n_1^{th} closest tuple of S inserted before τ, whereas the function $\text{next}(S, \tau, n_2)$ computes the insertion number ν_2 of the n_2^{th} closest tuple of S inserted after τ. The functioning of prev and next is exemplified in Fig. 2.

For instance, the expression $w^{\text{count}}_{[1,1]}(\texttt{OPTION\_TRADES})$ defines a streaming table whose tuples, for each timestamp τ, are in turn streaming tables extracted from `OPTION_TRADES` and containing one tuple immediately preceding τ, the tuples valid at τ, and one tuple immediately following τ.

The partitioned version of each sliding window operator applies to a streaming table S and formally generates a set of streaming tables of streaming tables, resulting from the application of the corresponding standard sliding window operator to each streaming table obtained by the partition of S according to a given set of attributes B. The operators' definitions follow.

Definition 9 (Time-based Partitioned Window). *The time-based partitioned sliding window operator $W^{\text{time}} : \mathcal{S} \times 2^{\mathcal{A}} \times \mathcal{I}^2 \to 2^{\mathcal{S}^*}$ over a streaming table $S \in \mathcal{S}$ creates a set (of streaming tables of streaming tables) composed of the time-based sliding windows (with a window size of duration $d_1 \geq 0$ before and $d_2 \geq 0$ after the timestamps around which it is computed) computed over the streaming tables into which S is partitioned according to the attributes in B:*

$$W^{\text{time},B}_{[d_1,d_2]}(S) := \{w^{\text{time}}_{[d_1,d_2]}(S') \mid S' \in \zeta_B(S)\}.$$

For instance, the expression $W^{\text{time},\texttt{OPTION}}_{[0,1\text{hour}]}(\texttt{OPTION\_TRADES})$, involving a time-based partitioned window, defines a set of streaming tables, each one corresponding to a different option, composed of the streaming tables whose tuples timestamped with τ are the streaming tables containing the trades involving that option negotiated in the hour that follows τ.

Definition 10 (Count-based Partitioned Window). *The count-based partitioned sliding window operator $W^{\text{count}} : \mathcal{S} \times 2^{\mathcal{A}} \times \mathbb{N}^2 \to 2^{\mathcal{S}^*}$ over a streaming table $S \in \mathcal{S}$ creates a set (of streaming tables of streaming tables) composed of the count-based sliding window (with a window containing the temporally closest $n_1 \geq 0$ tuples valid before and the temporally closest $n_2 \geq 0$ tuples valid after the timestamps around which it is computed) computed over the streaming tables into which S is partitioned according to the attributes in B:*

$$W^{\text{count},B}_{[n_1,n_2]}(S) := \{w^{\text{count}}_{[n_1,n_2]}(S') \mid S' \in \zeta_B(S)\}.$$

For instance, the expression $W^{\text{count},\texttt{STOCK}}_{[10,0]}(\texttt{OPTION\_TRADES})$, involving a count-based partitioned window definition, denotes a set of streaming tables, each for a different stock, composed of the streaming tables whose tuples timestamped with τ are the streaming tables containing the 10 most recent option trades preceding τ, in addition to all the trades possibly concluded at time τ, concerning that stock.

5.4. Window Flattening and Aggregation Operators (α)

The definition of window flattening and aggregation operators is introduced in the following.

5.4.1. Window Flattening Operators

Window flattening operators allow for *normalizing* the output of a time-based or a count-based sliding window operator in $\mathcal{S}^*$ or $2^{\mathcal{S}^*}$, respectively, to a streaming table in $\mathcal{S}$. As for sliding window operators, the window flattening operator is introduced both in its standard and in its partitioned version.

Definition 11 (Window Flattening). *The window flattening operator $\varphi : \mathcal{S}^* \to \mathcal{S}$ over a streaming table of streaming tables w creates a streaming table composed of the tuples belonging to the streaming tables in w valid at the time at which the flattening is computed:*

$$\varphi(w) := \{(\varepsilon_U(s), \tau) \mid \exists (S, \tau) \in w \wedge s \in S\}.$$

For instance, the expression $\varphi(w^{\text{time}}_{[1\text{hour},0]}(\texttt{OPTION\_TRADES}))$, involving a window flattening operator, builds a streaming table whose tuples with timestamp τ are all the tuples belonging to the streaming table $w^{\text{time}}_{[1\text{hour},0]}(\texttt{OPTION\_TRADES})$ valid at τ, that is belonging to the 1-hour wide time window of OPTION_TRADES preceding τ.

Definition 12 (Partitioned Window Flattening). *The partitioned window flattening operator* $\Phi : 2^{\mathcal{S}^*} \rightarrow \mathcal{S}$ *over a set of streaming tables of streaming tables* W *creates a streaming table composed of the tuples belonging to the streaming tables in* $w \in W$ *valid at the time at which the flattening is computed:*

$$\Phi(w) := \{(\varepsilon_U(s), \tau) \mid \exists w \in W \wedge \exists (S, \tau) \in w \wedge s \in S\}.$$

For instance, the expression $\Phi(W^{\text{count},\texttt{OPTION},\texttt{CLASS}}_{[4,0]}(\texttt{OPTION\_TRADES}))$, involving a partitioned flattening operator, retrieves the data necessary to display, for each time point and for each stock option, a book with the five latest put trades and the five latest call trades.

It is worth noting that, for both flattening operators, tuples come out all timestamped with τ in the result but preserve the value of the original timestamp they had in `OPTION_TRADES` converted into an explicit attribute U.

5.4.2. Window Aggregation Operators

Window aggregation operators are defined to compute aggregate data over time-based or count-based sliding windows, according to a set of aggregation functions $\mathcal{F}$. As for operators above, both standard and partitioned versions of the window aggregation operator are provided.

Definition 13 (Window Aggregation). *The sliding window aggregation operator* $\theta : \mathcal{S}^* \times 2^{\mathcal{F}} \rightarrow \mathcal{S}$ *over a streaming table of streaming tables* w *creates a streaming table having as attributes the values of the aggregates in* $F = \{f_1, \ldots, f_h\}$ *calculated over the streaming table in* w *valid at the time at which the aggregation is computed:*

$$\theta_F(w) := \{(Z, \tau) \mid \exists (S, \tau) \in w \wedge Z = (f_1(S), \ldots, f_h(S))\}.$$

The window aggregation operator θ can be used in queries for computing aggregate data over time-based or count-based sliding windows. For each time point τ, aggregates can be computed over the timestamped tuples belonging to the streaming table S in w valid at time τ; aggregate functions `MIN`, `MAX`, `COUNT` (and `SUM`, `AVG` if numeric), `FIRST_VALUE`, `LAST_VALUE`, `NTH_VALUE(n)`[3], can be used on the explicit attributes of S, whereas aggregate functions `FIRST`, `LAST`, `DURATION` can be used on the timestamps of S.

For instance, the expression $\theta_{\texttt{DURATION}}(w^{\text{count}}_{[9,0]}(\texttt{OPTION\_TRADES}))$ returns, for each time point, the width of the time window containing the 10 most recent option trades.

Definition 14 (Partitioned Window Aggregation). *The partitioned sliding window aggregation operator* $\Theta : 2^{\mathcal{S}^*} \times 2^{\mathcal{A}} \times 2^{\mathcal{F}} \rightarrow \mathcal{S}$ *over a set of streaming tables of streaming tables* W *creates a streaming table having as attributes the grouping attributes in* B *and the values of the aggregates in* $F = \{f_1, \ldots, f_h\}$

[3] `FIRST_VALUE`, `LAST_VALUE`, and `NTH_VALUE(n)` are part of the SQL:2011 window functions [36].

calculated over the streaming tables w belonging to W valid at the time at which the aggregation is computed:

$$_B\Theta_F(W) := \{(S.B \circ Z, \tau) \mid \exists w \in W \wedge (S, \tau) \in w \wedge Z = (f_1(S), \ldots, f_h(S))\}$$

The partitioned window aggregation operator Θ can be used in queries for computing aggregate data over partitioned time-based or count-based sliding windows. Also in this case, aggregate functions acting on explicit attributes or timestamps can be used.

For instance, the expression $_{\texttt{EXPIR}}\Theta_{\texttt{COUNT(OPTION)}}(W^{\text{time,EXPIR}}_{[0.5\text{hour},0.5\text{hour}]}(\texttt{OPTION\_TRADES}))$, at each timepoint and for each expiration date, returns the number of options with that expiration date traded in a 1-hour wide time window centered around the timepoint.

5.5. Legal Sliding Window Expressions

α	ω	w
$\varphi \mid \theta$	$w^{\text{time}} \mid w^{\text{count}}$	$\varphi(w^{\text{time}}_{[d_1,d_2]}(S))$, $\varphi(w^{\text{count}}_{[n_1,n_2]}(S))$, $\theta_F(w^{\text{time}}_{[d_1,d_2]}(S))$, $\theta_F(w^{\text{count}}_{[n_1,n_2]}(S))$
$\Phi \mid \Theta$	$W^{\text{time}} \mid W^{\text{count}}$	$\Phi(W^{\text{time},B}_{[d_1,d_2]}(S))$, $\Phi(W^{\text{count},B}_{[n_1,n_2]}(S))$, $_B\Theta_F(W^{\text{time},B}_{[d_1,d_2]}(S))$, $_B\Theta_F(W^{\text{count},B}_{[n_1,n_2]}(S))$

Table 3: $\mathcal{CTA}$ operators combinations in legal sliding window expressions w

The modularity of α and ω operators is regulated by the following definition:

Definition 15 (Legal Sliding Window Expression). *A $\mathcal{CTA}$* legal sliding window expression *$w(S)$ over the streaming table S is of the form $\alpha(\omega(S))$ where standard w^{time} and w^{count} (resp., partitioned W^{time} and W^{count}) sliding window operators can only be combined with their standard φ and θ (resp., partitioned Φ and Θ) window flattening or window aggregation counterparts.*

The admitted combinations are shown in Table 3 (combinations of standard operators in the first row, of partitioned operators in the second row). These constraints ensure that the value of continuous expressions augmenting $\mathcal{TA}^\star$ is always a streaming table, so that the resulting continuous algebra $\mathcal{CTA}$ is closed with respect to (streaming and) temporal tables. Nevertheless, the exotic datatypes $2^{\mathcal{S}}$, $\mathcal{S}^*$ and $2^{\mathcal{S}^*}$, that are used to formalize the operators and are not part of the model, could be used as guidelines for a *native* implementation of the $\mathcal{CTA}$ operators, for which the materialization of sets of streaming tables or (sets of) streaming tables of streaming tables (which could be stored in a $\neg$1NF datastore, e.g., in XML or JSON format) could be used to store intermediate results during the evaluation of $\alpha(\omega(S))$ window expressions. In our approach, materialization of such intermediate steps is unnecessary, as the translation that

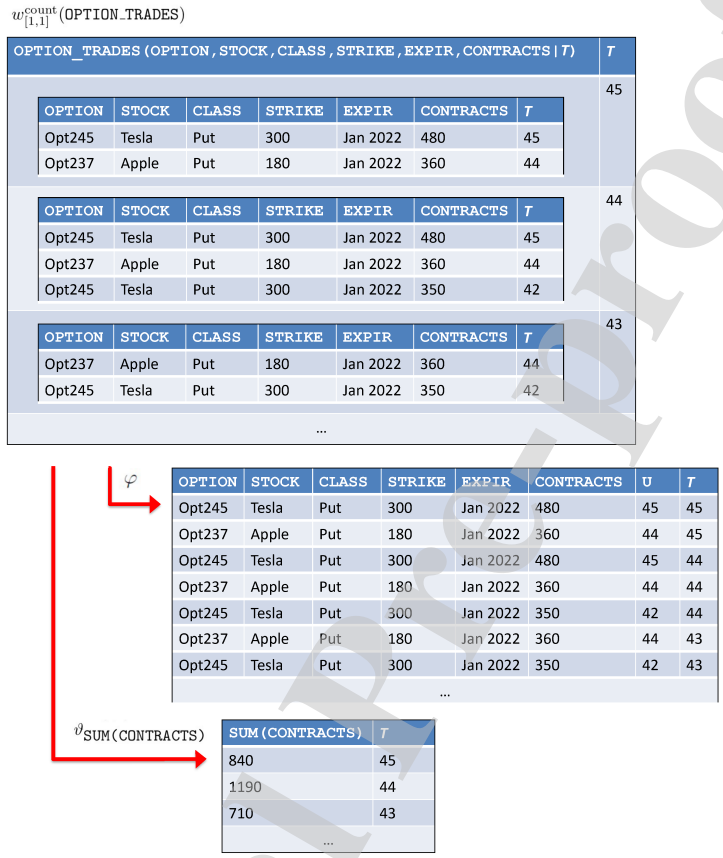


Figure 3: Examples of $\alpha(\omega(S))$ expressions: $\varphi(w^{\text{count}}_{[1,1]}(\texttt{OPTION\_TRADES}))$ and $\theta_{\texttt{SUM(CONTRACTS)}}(w^{\text{count}}_{[1,1]}(\texttt{OPTION\_TRADES}))$.

will be proposed in Sec. 8 maps any $\mathcal{CTA}$ expressions on $\mathcal{TA}^{\star}$ expressions to be executed on top of a temporal DBMS via manipulation of temporal relations only.

From a semantic point of view, the use of streaming tables of streaming tables as intermediate results in the evaluation of $\alpha(\omega(S))$ expressions allowed the propagation of original timestamps of tuples making up a streaming table as provenance witnesses, while whole streaming tables are assigned by the ω operator a new global timestamp corresponding to the window evaluation time. Provenance witness timestamps are then used by the α operator to correctly produce the desired final result in (1NF) streaming table format. In this respect, such provenance witness timestamps play a role similar to the lineage sets used in [16] to define change-preserving operators.

Example. Examples of how the results of $\alpha(\omega(S))$ expressions are conceptually

constructed via the composition of a sliding window operator with a flattening or aggregation operator are shown in Fig. 3. The input streaming table S is OPTION_TRADES, the value of the intermediate sliding window expression $w^{\text{count}}_{[1,1]}(\texttt{OPTION\_TRADES})$ is displayed on the top and the values of the final expressions $\varphi(w^{\text{count}}_{[1,1]}(\texttt{OPTION\_TRADES}))$ and $\theta_{\texttt{SUM(CONTRACTS)}}(w^{\text{count}}_{[1,1]}(\texttt{OPTION\_TRADES}))$ are displayed on the bottom of the figure.

6. CQs Evaluation

In order to support continuous queries, a sampling operator is formally introduced to evaluate an algebraic expression expressed in the continuous temporal algebra $\mathcal{CTA}$ at the required time points. In line with many CQ specification syntaxes (e.g., [23]), we assume a continuous query is always equipped with a *slide* parameter *sl* representing the query evaluation period, and with a further optional *alignment* parameter *a* specifying the position of the evaluation point within the evaluation period. The slide parameter can be either a user-supplied time span or the special parameter REALTIME, that means that the query is re-evaluated as new tuples arrive. The alignment value is expressed as a period of time to be counted from the beginning of the time granules representing the evaluation periods (and is ignored in case *sl*=REALTIME). Moreover, we also consider a *delay* parameter δ specifying that the evaluation of the query at time t has actually to be executed at time $t + \delta$. Parameters *sl*, *a* and δ used for sampling $\mathcal{CTA}$ expressions allow to generalize the usage of the so-called *tumbling windows* (and *hopping windows*) for producing continuous query results [35].

Definition 16 (Sampling Operator). *At execution time t, the sampling operator $\xi : \mathcal{CTA} \times \mathcal{T} \times \mathcal{I}^4 \rightarrow \mathcal{S}$, with an historical period parameter hp, a sliding parameter sl, an alignment parameter a, causes the evaluation of the algebraic expression $E \in \mathcal{CTA}$ at time points $t_0, t_1, \ldots, t_k$ only, where $t_i = (\lceil \frac{t-hp-a}{sl} \rceil + i) \cdot sl + a$ and k is the largest natural number such that $t_k \leq t$. If a delay parameter δ is specified, it forces the evaluation of the expression E to be actually executed at time $t + \delta$:*

$$\xi^{t,\delta}_{hp,sl,a}(E) := \bigcup_{i=0}^{\max\{k \in \mathbb{N} | t_k \leq t\}} \tau_{t_i}(E^{t+\delta})$$

For example, if *sl*="1 day", the continuous execution must produce one result per day: if the alignment parameter is *a*="30 minutes", the results are produced each day at "00:30" in the morning, whereas if the alignment parameter is *a*="16 hours", the results are produced each day at 4 p.m.. Notice that different results are produced with respect to the desired alignment, since time windows are defined with reference to the execution times, which depend on the alignment. For instance, assuming daily trading hours range from 9 a.m. to 4 p.m., the sliding window $w^{\text{time}}_{[\texttt{1day},0]}(\texttt{OPTION\_TRADES})$ executed via a sampling with *sl*="1

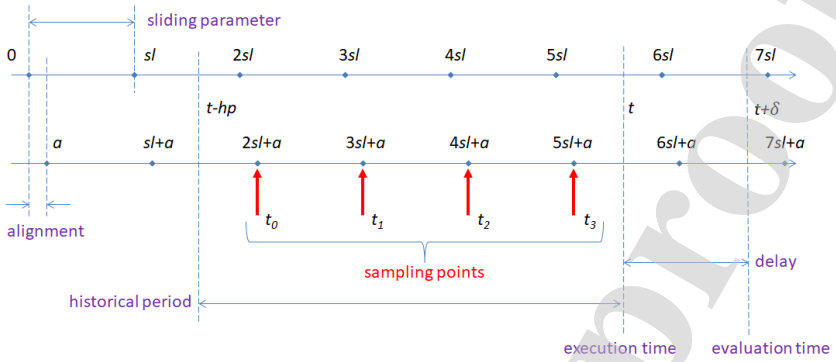


Figure 4: The sampling points defined at execution time t by the parameters hp, sl, a, δ.

day" and a="6 hours" includes all the trades executed the day before (from 9 a.m. to 4 p.m.) to contribute to a result produced daily at 6 a.m., but if it were executed with a="12 hours" it would include all the trades executed in the afternoon of the day before (from noon to 4 p.m.) and in the morning of the current day (from 9 a.m. to noon) to contribute to a result produced daily at noon.

Fig. 4 shows how sampling points can be visually determined on the time axis according to the specified ξ parameters. In practice, sl and a define a sequence of time points with period sl shifted by a with respect to the time origin. The sampling points, highlighted by red arrows in the figure, are the points of such sequence that fall into the hp-wide interval preceding the execution time t.

When a delay value δ is specified, the evaluation of the expression E valid at time t is actually computed at time $t + \delta$: in general, the results (both valid at time t) computed at time t and at time $t+\delta$ may differ as some required contents of the temporal relations may not be available at time t yet, or even because their contents may have been retroactively changed after t (and also tuples in the streaming tables might be inserted with a little delay with respect to their validity, e.g., to enforce the right timestamp order). Consider, for instance, our example of insider trading detection that, in order to produce one result per trading day, needs to compare the trading volumes evaluated using the streaming data valid on a 1-month window preceding the execution time and on a 1-day window following the execution time (to this purpose, it would be sufficient to delay the execution at the end of the day). However, it also needs to join such volumes with the news concerning the same stocks published within one week. Since we can assume relevant news are selected and inserted by human analysts, we should also consider that they are likely inserted into the `NEWS` table retroactively, with the delay of some days with respect to their publication date. Hence, we should reasonably allow for a delay of, say, 10-15 days in the

execution of the CQ in order to have all the relevant news available, otherwise the result of the join would always be empty and the insider trading cases could not be detected.

Notice that, when the streaming tables involved are defined at a finer time *granularity* than *sl* (e.g., *sl*="1 hour" but new tuples can be inserted into the streaming tables at every second), the values of E are usually defined for many more time points than required by the query. Hence, the sampling operator can be used to exactly specify the query execution timepoints of interest. This is also the reason for which we said in Sec. 2 that non-temporal tables (appearing in the expression E) are considered *virtually* converted into temporal tables that contain an infinite number of tuples: only the tuples timestamped with one of the timepoints of interest have actually to be generated.

According to the generally accepted definition of continuous query semantics [37], we define the semantics of a continuous query $Q^{c,\delta}_{hp,sl,a}$ denoted by an algebraic expression $E_{\mathcal{CTA}} \in \mathcal{CTA}$ to be equal to the sampling of $E_{\mathcal{CTA}}$ at the time points specified by the slide parameter *sl* and the alignment parameter a.

Definition 17 (CQs Semantics). *Let $E_{\mathcal{CTA}} = E_{\mathcal{TA}^\star}(w_1(S_1), \ldots, w_n(S_n), R^T_1, \ldots, R^T_m, Q_1, \ldots, Q_\ell)$ be an algebraic expression in $\mathcal{CTA}$ containing only legal sliding window expressions $w_i(S_i) = \alpha_i(\omega_i(S_i))$, with $1 \leq i \leq n$.*

The result at time t of the continuous query $Q^{c,\delta}_{hp,sl,a}$ with historical period parameter hp, slide parameter sl, alignment parameter a and delay parameter δ, expressed by $E_{\mathcal{CTA}}$ is the streaming table with historical period hp given by the sampling $\xi^{t,\delta}_{hp,sl,a}(E_{\mathcal{CTA}})$ of $E_{\mathcal{CTA}}$ at the time points specified by sl with alignment a, and evaluation delayed by δ, until t.

It is worth noting that, as the CQ semantics is founded on the sampling operator, we actually implement a CQ on-demand semantics that produces at the execution query time successive query evaluations at past query points, thus in a delayed mode (also when δ=0). Moreover, the semantics of joining streaming tables and temporal and standard tables as specified in the algebraic expression $E_{\mathcal{TA}^\star}$ refers to the standard temporal semantics. In this way, we implement different kinds of joining semantics according to the involved tables. For instance, when temporal tables are involved, the joining results will be temporally consistent according to the required time points in the past. When, instead, standard tables are involved, the joining semantics allows users to interoperate current data with past streamed data.

7. Temporal SQL Hybrid Semantics and Example Reprise

Since it is able to also support real temporal queries (and as also theoretically remarked by Lemma 1), the $\mathcal{TA}^\star$ extended algebra we introduced in Section 4.3 is strictly more expressive than a shapshot-reducible algebra like $\mathcal{TA}$. In particular, the $\mathcal{TA}^\star$ algebra has the same expressive power of a temporal SQL extension allowing for explicit reference to temporal attributes in addition to snapshot-reducible semantics for implicit temporal attributes like TSQL2, for

which it mostly represents a syntactic variant. In fact, considering the general temporal TSQL2-like SPJ query:

$$\begin{array}{l}\texttt{SELECT } A_1, A_2, \ldots, A_n \\ \texttt{FROM } R_1, R_2, \ldots, R_m \\ \texttt{WHERE } C' \texttt{ AND } C''\end{array} \quad \text{(Q0)}$$

where C' is a selection predicate only involving explicit attributes and C'' is a selection predicate explicitly involving the timestamps of the relations $R_{h+1}, \ldots, R_m$ (with $h \geq 0$). The attributes $A_1, A_2, \ldots, A_n$ appearing in the `SELECT` clause may belong to any of the $R_1, \ldots, R_h$ relations or to only one of the $R_{h+1}, \ldots, R_m$ relations, say R_k.

The query (Q0) can be translated into the $\mathcal{TA}^\star$ expression:

$$\begin{array}{l}\pi^T_{A_1,\ldots,A_n}(\sigma^T_{C'}(R_1 \times^T \cdots \times^T R_h \times^T \\ \quad \sigma^T_{\overline{C}''}(\pi_{X_{h+1},U}(\varepsilon_U(R_{h+1})) \times \cdots \times \varepsilon_U(R_k) \times \cdots \times \pi_{X_m,U}(\varepsilon_U(R_m)))))\end{array} \quad \text{(A1)}$$

where X_i is the explicit schema of $R_i(X_i|T)$ and $\overline{C}''$ it the predicate obtained by substituting references to implicit time `T` with references to its explicit copy U added by ε_U. Conjuncts involving explitic attributes of $R_{h+1}, \ldots, R_m$ olny can be moved from C' to $\overline{C}''$. The argument of $\sigma^T_{\overline{C}''}$ is the minimal subexpression containing non-temporal operators in this case. The relation R_k (with $h+1 \leq k \leq m$), appearing without projection in this subexpression, is the one whose attributes appear in the `SELECT` clause and donates its timestamps to the subexpression result. If no reference to attributes of one of the relations $R_{h+1}, \ldots, R_m$ is made in the `SELECT` clause, the relation R_k to be used without projection within the subexpression can be any one of the $R_{h+1}, \ldots, R_m$ relations. The relations $R_1, \ldots, R_h$ only involved in the C' predicate are managed instead with sequenced semantics, by preserving the main advantages of such an approach (first of all, the fact that overlap conditions between their timestamps are implicit in the translation with temporal algebra operators and do not need to be explicited by users at the query language level and the default timestamps to be assigned to the results are implicitly computed as their intersection).

As proposed for TSQL2, if a non-temporal table is desired as result, the form `SELECT SNAPSHOT` has to be used. In such a case, in the algebra translation, the outermost temporal projection π^T can be replaced by a non-temporal projection π in order to eliminate the timestamps.

As particular cases of (Q0), we can consider a pure sequenced query and a pure non-sequenced query. A pure sequenced query does not contain selection predicates involving timestamps (i.e., C'' is void and $m = h$). In this case, the translation reduces to a pure $\mathcal{TA}$ expression:

$$\pi^T_{A_1,\ldots,A_n}(\sigma^T_{C'}(R_1 \times^T \cdots \times^T R_h)) \quad \text{(A2)}$$

A pure non-sequenced query instead only involves relations whose timestamps appear in the temporal selection condition C'' (i.e., $h = 0$). In such a case, the

translation becomes:

$$\pi^T_{A_1,\ldots,A_n}(\sigma^T_{C' \wedge \overline{C}''}(\pi_{X_1,U}(\varepsilon_U(R_1)) \times \cdots \times \varepsilon_U(R_k) \times \cdots \times \pi_{X_m,U}(\varepsilon_U(R_m)))) \quad \text{(A3)}$$

Since σ^T is always equivalent to σ when the selection predicate only involve explicit attributes, if the query does not require the result to be a temporal relation (i.e., it has the form `SELECT SNAPSHOT`), π has to be used in place of π^T to remove timestamps and, thus, the whole translation can be reduced to a non-temporal algebra expression.

The correspondence between any general $\mathcal{CTA}$ expression and its SQL-like syntax as exemplified in Sec. 3 can be defined in a similar way, also taking into account grouping, window declarations with flattening/aggregate expressions and sampling specifications.

Hence, we show in the remainder of this section how the example queries presented in Sec. 3 can be automatically translated into $\mathcal{CTA}$ expressions. In order to have more compact formulas, we will always use the aliases (i.e., relational variable names) declared in the `FROM` clause of the queries instead of the full relation names. Besides immediate translations exemplifying the application of the (A1)–(A3) rules provided above and only using the $\mathcal{TA}$ base operators in Table 1, we will also provide (owing to the equivalence $\sigma_F(R_1 \times R_2) \equiv R_1 \bowtie_F R_2$, where F is a join condition between R_1 and R_2, also valid for temporal operators) the corresponding expressions containing join operations instead of Cartesian products, which is a format more friendly to SQL programmers and query optimizers.

(Q1) The query does not contain temporal selection conditions (i.e., predicates involving timestamps) and, thus, the specialized translation formula (A2) can be used:

$$\begin{aligned} Q_1 &= \pi^T_{\texttt{ID,OPTION}}(\sigma^T_{\texttt{LN.STOCK=OT.STOCK} \wedge \texttt{LN.STOCK='Apple'}}(\texttt{LN} \times^T \texttt{OT})) \\ &= \pi^T_{\texttt{ID,OPTION}}(\sigma^T_{\texttt{LN.STOCK='Apple'}}(\texttt{LN}) \bowtie^T_{\texttt{LN.STOCK=OT.STOCK}} \texttt{OT}) \end{aligned}$$

The outcome is a temporal table with schema $(\texttt{ID}, \texttt{OPTION} \mid T)$.

(Q2) The query only contains relations involved in temporal selection conditions and, thus, the specialized translation formula (A3) can be used. Since the the result is required to be in snapshot format as (i.e., a `SELECT SNAPSHOT` clause is used), the outermost projection needs to be non-temporal to remove the timestamps from the retrieved tuples. The resulting expression can be evaluated

using non-temporal algebra operators only:

$$Q_2 = \pi_{\texttt{LN1.ID}}(\sigma_{\substack{\texttt{LN1.SOURCE=LN2.SOURCE}\\ \wedge\texttt{LN1.STOCK=LN2.STOCK}\\ \wedge\texttt{LN1.TYPE=LN2.TYPE}\\ \wedge\texttt{LN1.U MEETS LN2.U}}}(\pi_{\substack{\texttt{ID,STOCK,TYPE,}\\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN1})) \times \pi_{\substack{\texttt{ID,STOCK,TYPE,}\\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN2}))))$$

$$= \pi_{\texttt{LN1.ID}}(\pi_{\texttt{ID,STOCK,TYPE,SOURCE,U}}(\varepsilon_U(\texttt{LN1})) \bowtie_{\substack{\texttt{LN1.SOURCE=LN2.SOURCE}\\ \wedge\texttt{LN1.STOCK=LN2.STOCK}\\ \wedge\texttt{LN1.TYPE=LN2.TYPE}\\ \wedge\texttt{LN1.U MEETS LN2.U}}} \pi_{\substack{\texttt{ID,STOCK,TYPE,}\\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN2})))$$

The outcome is a non-temporal table with schema (LN1.ID).

(Q3) The query contains temporal selection conditions and relations non involved in them and, thus, the general translation formula (A1) must be used (with LN1 as R_k as its attributes ID appear in the SELECT clause):

$$Q_3 = \pi^T_{\substack{\texttt{N1.ID,}\\ \texttt{OPTION}}}(\sigma^T_{\texttt{LN1.STOCK=OT.STOCK}}(\texttt{OT} \times^T (\sigma^T_{\substack{\texttt{LN1.SOURCE=LN2.SOURCE}\\ \wedge\texttt{LN1.STOCK=LN2.STOCK}\\ \wedge\texttt{LN1.TYPE=LN2.TYPE}\\ \wedge\texttt{LN1.U MEETS LN2.U}}}[\varepsilon_U(\texttt{LN1}) \times \pi_{\substack{\texttt{ID,STOCK,TYPE,}\\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN2}))])))$$

$$= \pi^T_{\substack{\texttt{N1.ID,}\\ \texttt{OPTION}}}(\texttt{OT} \bowtie^T_{\texttt{LN1.STOCK=OT.STOCK}} [\varepsilon_U(\texttt{LN1}) \bowtie_{\substack{\texttt{LN1.SOURCE=LN2.SOURCE}\\ \wedge\texttt{LN1.STOCK=LN2.STOCK}\\ \wedge\texttt{LN1.TYPE=LN2.TYPE}\\ \wedge\texttt{LN1.U MEETS LN2.U}}} \pi_{\substack{\texttt{ID,STOCK,TYPE,}\\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN2}))])$$

The part within square brackets is the minimal subexpression containing non-temporal operators. The outcome is a temporal table with schema (N1.ID, OPTION | T).

(Q4) This is a continuous query containing two window expressions, W1A and W1B, which are both partitioned time-based windows with partitioned aggregation and whose definitions can be translated, respectively, into the $\mathcal{CTA}$ expression that follow:

$$\texttt{W1A} = {}_{\texttt{STOCK}}\Theta_{\texttt{SUM(CONTRACTS)}}(W^{\texttt{time,STOCK}}_{[0,\texttt{1week}]}(\texttt{OPTION\_TRADES})),$$

$$\texttt{W1B} = {}_{\texttt{STOCK}}\Theta_{\texttt{SUM(CONTRACTS)}}(W^{\texttt{time,STOCK}}_{[\texttt{1week},0]}(\texttt{OPTION\_TRADES}))$$

The query part preceding the SAMPLE clause does not contain temporal selection conditions (i.e., predicates involving timestamps) and, thus, can be translated using the specialized translation formula (A2) as follows:

$$E_4 = \pi^T_{\substack{\texttt{SOURCE,LN.STOCK,}\\ \texttt{100*(W1A.SUM(CONTRACTS)/}\\ \texttt{W1B.SUM(CONTRACTS)-1),}\\ \texttt{BEGIN(LN)}}}(\sigma^T_{\substack{\texttt{LN.STOCK=W1A.STOCK}\\ \texttt{LN.STOCK=W1B.STOCK}}}(\texttt{LN} \times^T \texttt{W1A} \times^T \texttt{W1B}))$$

Notice the required final projection on BEGIN(LN) can be preliminary applied to the relation LN (transforming it, in practice, from a temporal *state relation* with interval timestamps $[t_B, t_E)$ to a temporal *event relation* with point timestamps

t_B). This avoids the temporary generation by the $\times^T$ or $\bowtie^T$ operators of many tuples (due to the matching of the timestamps of the window expression also with the (t_B, t_E) part of the timestamps of LN) that will be eventually discarded by the timestamp intersection implied in the projection. Hence, an equivalent but optimized formula is as follows:

$$\begin{aligned} E'_4 &= \pi^T_{\substack{\texttt{SOURCE,LN.STOCK,}\\ \texttt{100*(W1A.SUM(CONTRACTS)/}\\ \texttt{W1B.SUM(CONTRACTS)-1)}}} (\sigma^T_{\substack{\texttt{LN.STOCK=W1A.STOCK}\\ \wedge\texttt{LN.STOCK=W1B.STOCK}}} (\pi^T_{\texttt{BEGIN(LN)}}(\texttt{LN}) \times^T \texttt{W1A} \times^T \texttt{W1B})) \\ &= \pi^T_{\substack{\texttt{SOURCE,LN.STOCK,}\\ \texttt{100*(W1A.SUM(CONTRACTS)/}\\ \texttt{W1B.SUM(CONTRACTS)-1)}}} \\ &\qquad ((\pi^T_{\texttt{BEGIN(LN)}}(\texttt{LN}) \bowtie^T_{\texttt{LN.STOCK=W1A.STOCK}} \texttt{W1A}) \bowtie^T_{\texttt{LN.STOCK=W1B.STOCK}} \texttt{W1B}) \end{aligned}$$

Finally, the required sampling operator has to be applied to the resulting expression as follows:

$$Q'_4 = \xi^{\texttt{now},\texttt{1week}}_{\infty,\texttt{1sec},0}(E'_4)$$

The outcome is a temporal table with schema (SOURCE, LN.STOCK, 100*(W1A.SUM (CONTRACTS)/W1B.SUM(CONTRACTS)-1) | T), where new tuples are added at each second with a 1-week delay with respect to their timestamp values.

(Q5) This is a continuous query containing a window expressions, N10KA, which is a count-based window with aggregation and whose definition can be translated into $\mathcal{CTA}$ as follows:

$$\texttt{N10KA} = \theta_{\texttt{LAST}}(w^{\text{count}}_{[0,10000]}(\texttt{OPTION\_TRADES})),$$

The query part preceding the SAMPLE clause only contains relations/window expressions involved in temporal selection conditions and, thus, the specialized translation formula (A3) can be used:

$$\begin{aligned} E_5 &= \pi^T_{\texttt{DAY(N10KA.U)}}(\sigma^T_{\substack{\texttt{TYPE='Product Launch'}\wedge\texttt{STOCK='Apple'}\\ \wedge\texttt{N10KA.LAST<BEGIN(LN.U)}\wedge\texttt{BEGIN(LN.U)<N10KA.U+12hour}}} \\ &\qquad [\varepsilon_U(\texttt{N10KA}) \times \pi_{\substack{\texttt{ID,STOCK,TYPE,}\\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN2}))]) \\ &= \pi^T_{\texttt{DAY(N10KA.U)}}([\varepsilon_U(\texttt{N10KA}) \bowtie^T_{\substack{\texttt{N10KA.LAST<BEGIN(LN.U)}\\ \wedge\texttt{BEGIN(LN.U)<N10KA.U+12hour}}} \\ &\qquad \pi_{\substack{\texttt{ID,STOCK,TYPE,}\\ \texttt{SOURCE,U}}}(\varepsilon_U(\sigma^T_{\substack{\texttt{TYPE='Product Launch'}\\ \wedge\texttt{STOCK='Apple'}}}(\texttt{LN2})))]) \end{aligned}$$

The part within square brackets is the minimal subexpression containing non-temporal operators. Notice that, although the result is required to be in snapshot format, we cannot apply a non-temporal outermost projection as we did for (Q2), since the timestamps of N10KA are needed to apply the sampling operator. Hence, the timestamp removal required by the SELECT SNAPSHOT clause can be just performed after the application of the sampling operator:

$$Q_5 = \pi_{\texttt{DAY(N10KA.U)}}(\xi^{\texttt{now},\texttt{1week}}_{\infty,\texttt{1day},\texttt{12hour}}(E_5))$$

The outcome is a non-temporal table with schema (DAY(N10KA.U)), where new tuples are added each day at noon with a 1-week delay with respect to their timestamp values.

(Q6) This is a continuous query containing two window expressions, D1A and M1B, which are both partitioned time-based windows with partitioned aggregation and whose definitions can be translated into $\mathcal{CTA}$, respectively, as follows:

$$\begin{aligned} \texttt{D1A} &= {}_{\texttt{STOCK}}\Theta_{\texttt{SUM(CONTRACTS)}}(W^{\text{time},\texttt{STOCK}}_{[0,\texttt{1day}]}(\texttt{OPTION\_TRADES})), \\ \texttt{M1B} &= {}_{\texttt{STOCK}}\Theta_{\texttt{SUM(CONTRACTS)}}(W^{\text{time},\texttt{STOCK}}_{[\texttt{1month},0]}(\texttt{OPTION\_TRADES})) \end{aligned}$$

The query part preceding the SAMPLE clause contains temporal selection conditions and relations non involved in them and, thus, the general translation formula (A1) must be used (with the window expression D1A playing the role of R_k as its attribute STOCK appear in the SELECT clause):

$$\begin{aligned} E_6 &= \pi^T_{\texttt{D1A.STOCK}}(\sigma^T_{\substack{\texttt{D1A.STOCK=M1B.STOCK} \wedge \texttt{D1A.STOCK=LN.STOCK} \\ \wedge \texttt{D1A.SUM(CONTRACTS)>M1B.SUM(CONTRACTS)/3}}} \\ &\qquad (\texttt{M1B} \times^T \sigma^T_{\substack{\texttt{LN.U} \geqslant \texttt{D1A.U+1day} \\ \wedge \texttt{LN.U<D1A.U+1week}}} [\varepsilon_U(\texttt{D1A}) \times \pi_{\substack{\texttt{ID,STOCK,TYPE,} \\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN}))])) \\ &= \pi^T_{\texttt{D1A.STOCK}}(\texttt{M1B} \bowtie^T_{\substack{\texttt{D1A.STOCK=M1B.STOCK} \wedge \texttt{D1A.STOCK=LN.STOCK} \\ \wedge \texttt{D1A.SUM(CONTRACTS)>M1B.SUM(CONTRACTS)/3}}} \\ &\qquad [\varepsilon_U(\texttt{D1A}) \bowtie_{\substack{\texttt{LN.U} \geqslant \texttt{D1A.U+1day} \\ \wedge \texttt{LN.U<D1A.U+1week}}} \pi_{\substack{\texttt{ID,STOCK,TYPE,} \\ \texttt{SOURCE,U}}}(\varepsilon_U(\texttt{LN}))]) \end{aligned}$$

The part within square brackets is the minimal subexpression containing non-temporal operators.

Finally, the required sampling operator, for which also an HISTORICAL PERIOD of 6 months has been specified, has to be applied to the resulting expression as follows:

$$Q_6 = \xi^{\text{now},\texttt{15day}}_{\texttt{6month},\texttt{1day},0}(E_6)$$

The outcome is a temporal table with schema $(\texttt{D1A.STOCK} \mid T)$, where new tuples are added each day at midnight with a 15-day delay with respect to their timestamp values.

We conclude or example roundup by showing how also the $\mathcal{CTA}$ equivalent of query (Q4″) involving a (continuous) view can be determined. First of all, using the algebra expression Q'_4 of its definition query (Q4′), the algebra expression defining of the view NEWS_IMPACT becomes as follows (ρ is the standard renaming operator of the relational algebra):

$$\begin{aligned} \texttt{NEWS\_IMPACT} &= \rho_{2,3 \leftarrow \texttt{STOCK,VOL\_VAR}}(Q'_4) \\ &= \rho_{2,3 \leftarrow \texttt{STOCK,VOL\_VAR}}(\xi^{\text{now},\texttt{1week}}_{\infty,\texttt{1sec},0}(E'_4)) \\ &= \xi^{\text{now},\texttt{1week}}_{\infty,\texttt{1sec},0}(\rho_{2,3 \leftarrow \texttt{STOCK,VOL\_VAR}}(E'_4)) \\ &= \xi^{\text{now},\texttt{1week}}_{\infty,\texttt{1sec},0}(\tilde{E}'_4) \end{aligned}$$

where $\tilde{E}'_4$ is E'_4 with the second and third column renamed as STOCK and VOL_VAR, respectively. Then, the translation of (Q4″) into $\mathcal{TA}$ is straightforward:

$$Q''_4 = {}_{\texttt{SOURCE}}\vartheta^T_{\texttt{COUNT(DISTINCT STOCK),AVG(VOL\_VAR)}}(\sigma^T_{\texttt{VOL\_VAR}>50}(\texttt{NEWS\_IMPACT}))$$

Substituting the view definition expression above we obtain:

$$\begin{aligned} Q_4'' &= {}_{\texttt{SOURCE}}\vartheta^{T}_{\texttt{COUNT(DISTINCT STOCK)},\texttt{AVG(VOL\_VAR)}}(\sigma^{T}_{\texttt{VOL\_VAR}>50}(\xi^{\texttt{now},\texttt{1week}}_{\infty,\texttt{1sec},0}(\tilde{E}_4'))) \\ &= \xi^{\texttt{now},\texttt{1week}}_{\infty,\texttt{1sec},0}({}_{\texttt{SOURCE}}\vartheta^{T}_{\texttt{COUNT(DISTINCT STOCK)},\texttt{AVG(VOL\_VAR)}}(\sigma^{T}_{\texttt{VOL\_VAR}>50}(\tilde{E}_4'))) \end{aligned}$$

The outcome is a temporal table with schema (SOURCE, COUNT(DISTINCT STOCK), AVG(VOL_VAR) | T), where new tuples are added at each second with a 1-week delay with respect to their timestamp values.

8. Translating CQs into OTQs (with Implementation on the Horizon)

In this section we propose a translation of the continuous temporal model presented so far into a new temporal model where continuous queries are transformed into temporal one-time queries. Furthermore, whereas the continuous model has been defined as an *abstract temporal model* (point-based), the new model is intended to be a *concrete temporal model* (interval-based) [20] amenable to implementation. In particular, the new temporal model can be implemented on a traditional relational DBMS following similar directions as presented in [16]. In fact, our final aim is to build comprehensive support for the continuous temporal model through a mixed stratum/built-in approach that relies on the full potentialities of an industrial-strength relational engine, extended with novel functionalities.

For the intended translation, the source algebra is therefore $\mathcal{CTA}$ and the target algebra is $\mathcal{TA}^{\star}$ but made to work on relations employing interval-timestamping, according to an extended sequenced semantics [16], in order to enforce *snapshot equivalence* also in the presence of subexpressions supporting a non-sequenced semantics. Although working on an interval-based concrete temporal model, the target algebra represents indeed a point-based query language (in the sense of [38]) and, thus, its implementation on a traditional DBMS does not require enforcement of *change preservation* (e.g., via adjustment, alignment and scaling techniques as proposed in [16]). For ease of presentation, hereinafter, with a little abuse of notation, when we need to distinguish the same concept at the two different levels, we will use the superscript $\mathcal{CTA}$ to denote tables and algebraic operators in the continuous temporal model and the superscript $\mathcal{TA}^{\star}$ to denote the corresponding concepts in the target model.

The main issue for our goal is to mimic in a static context the behavior of $\mathcal{CTA}$ windowing operators, which are evaluated on user-specified time intervals and operate on the contents of the involved streaming tables at the evaluation instants. To this end, we first translate each streaming table $S^{\mathcal{CTA}}$ with schema $S(X|T)$ at the continuous level into an interval-based streaming table $S^{\mathcal{TA}^{\star}}$ with schema $S(X, T|T')$, where the event occurrence time T associated to tuples is made explicit and T' is an implicit interval attribute that records tuple validity, that is $[st, \infty)$, where st is the time when the tuple s enters the system (without transaction-time support, it is worth noting that st can be approximated with $s.T$). Each temporal relation with schema $R(X|T)$ in $\mathcal{CTA}$ is translated into a

relation with schema $R(X|T')$ in $\mathcal{TA}^\star$, by *coalescing* the timestamps of *value-equivalent* tuples in $R^{\mathcal{CTA}}$ into maximal intervals to be used as timestamps in $R^{\mathcal{TA}^\star}$. Non temporal relations are converted into temporal relations whose tuples are timestamped with a $[0, \infty)$ validity interval in $\mathcal{TA}^\star$.

Then, in Tab. 4 we present the semantics of the continuous operators introduced in Sec. 5.1 defined through $\mathcal{TA}$ operators, to be used as translation rules from the source $\mathcal{CTA}$ to the target $\mathcal{TA}^\star$ language (the aliases P and P_B are introduced just to disambiguate attributes T and B belonging to both join operands).

Notice that, unlike their counterpart at the $\mathcal{CTA}$ level, sliding window operators at the $\mathcal{TA}^\star$ level require a set of time instants *tset* to be evaluated and the flattening operator $\Phi^{\mathcal{TA}}$ simply undoes the effects of partitioning. Being $\mathcal{TA} \subset \mathcal{TA}^\star$, with the translation rules of Tab. 4 converting sliding window expressions into $\mathcal{TA}$, any legal $\mathcal{CTA}$ expression can be translated into $\mathcal{TA}^\star$. Moreover, it is worth stressing that, in this way, any legal $\mathcal{CTA}$ sliding window expression can be evaluated via $\mathcal{TA}$ operators working on streaming tables only. In particular, there is no need for implementing sets of streaming tables or (sets of) streaming tables of streaming tables as formally introduced in the definitions of $\mathcal{CTA}$ operators in Sec. 5.

Translation of time-based windows and partitioning can be easily carried out using the selection operator σ_p^T (with p involving the attribute T, which is explicit after the conversion to $\mathcal{TA}^\star$) and the grouping operator ${}_B\vartheta_F^T$, respectively. The translation of count-based window expressions deserves some more explanations. In general, the definition of a count-based window using plain relational algebra operators is problematic as it requires the notion of ordered data or sequence which is lacking in the relational model, where relations are defined as *sets* of tuples. To solve this problem, for instance, some authors provided for extensions of the algebra with a rank [39] or sequence [40] operator that can be applied to relations to superimpose an ordering to their tuples. In this work, we take a different approach, which exploits the intrinsic ordering of tuples in a streaming table, which is partial on the insertion time and total on the insertion order. Then, in order to define a count-based window expression, the standard grouping operator ${}_B\vartheta_F^T$ can come in help, provided that two new aggregate functions `PREV` and `NEXT` are available. Such aggregate functions correspond exactly to the functions with the same names introduced in Def. 8 (how they operate has been also illustrated in Fig. 2) and can easily and efficiently implemented thanks to the existing ordering of tuples in a streaming table. The values $\nu_1 = \texttt{PREV}(n_1)$ and $\nu_2 = \texttt{NEXT}(n_2)$, computed for each T (or for each T, B pair in the partitioned case) owing to the grouping, are then used to select tuples with insertion number $\nu_1 \leq \nu \leq \nu_2$ via the join predicate. Therefore, in order to implement count-based window expressions, we do not need to add a new algebraic operator to the kernel of a DBMS but just to implement two new aggregate functions to be used with the standard grouping operator. Notice that partitioning does not affect the definition of time-based windows (as the union of the tuples selected per partition equals the set of tuples globally selected without partitions) but

Time-based sliding windows
${}^{tset}w^{\text{time}}_{[d_1,d_2]}(S)^{\mathcal{TA}} := \bigcup_{t\in tset} \tau_t(\sigma^T_{t-d_1\leq S.T\leq t+d_2}(S))$
${}^{tset}W^{\text{time,B}}_{[d_1,d_2]}(S)^{\mathcal{TA}} := \bigcup_{t\in tset} \tau_t(\sigma^T_{t-d_1\leq S.T\leq t+d_2}(S))$
Count-based sliding windows
${}^{tset}w^{\text{count}}_{[n_1,n_2]}(S)^{\mathcal{TA}} := \bigcup_{t\in tset} \tau_t(\pi^T_{S.X}(S \bowtie^T_{\substack{S.T=P.T \\ \wedge \texttt{PREV}(n_1)\leq S.\nu\leq \texttt{NEXT}(n_2)}} P))$ where $P = {}_T\vartheta^T_{\texttt{PREV}(n_1),\texttt{NEXT}(n_2)}(S)$
${}^{tset}W^{\text{count,B}}_{[n_1,n_2]}(S)^{\mathcal{TA}} := \bigcup_{t\in tset} \tau_t(\pi^T_{S.X}(S \bowtie^T_{\substack{S.T=P_B.T\wedge S.B=P_B.B \\ \wedge \texttt{PREV}(n_1)\leq S.\nu\leq \texttt{NEXT}(n_2)}} P_B))$ where $P_B = {}_{T,B}\vartheta^T_{\texttt{PREV}(n_1),\texttt{NEXT}(n_2)}(S)$
Window flattening operators
$\varphi(S)^{\mathcal{TA}} := S$
$\Phi({}^{tset}W^{\text{time},B}_{[d_1,d_2]}(S))^{\mathcal{TA}} := {}^{tset}W^{\text{time},B}_{[d_1,d_2]}(S)^{\mathcal{TA}}$
$\Phi({}^{tset}W^{\text{count},B}_{[n_1,n_2]}(S))^{\mathcal{TA}} := {}^{tset}W^{\text{count},B}_{[n_1,n_2]}(S)^{\mathcal{TA}}$
Window aggregation operators
$\theta_F(S)^{\mathcal{TA}} := ({}_\emptyset\vartheta^T_F(S))^{\mathcal{TA}}$
${}_B\Theta_F({}^{tset}W^{\text{time},B}_{[d_1,d_2]}(S))^{\mathcal{TA}} := ({}_B\vartheta^T_F({}^{tset}w^{\text{time}}_{[d_1,d_2]}(S)))^{\mathcal{TA}}$
${}_B\Theta_F({}^{tset}W^{\text{count},B}_{[n_1,n_2]}(S))^{\mathcal{TA}} := ({}_B\vartheta^T_F({}^{tset}W^{\text{count,B}}_{[n_1,n_2]}(S)))^{\mathcal{TA}}$

Table 4: Translation of windowing operators into $\mathcal{TA}$

affects the definition of count-based windows (since tuples belonging to different partitions may be interleaved along the insertion order, the evaluation of ν_1 and ν_2 gives different results with groups defined by T or T, B; in the latter case, an additional join condition on B is required to skip tuples in the insertion number range $[\nu_1, \nu_2]$ non belonging to the partition defined by B).

Finally, we define the sampling operator at the $\mathcal{TA}^\star$ level and show that the results of the two sampling operators, the one defined at the $\mathcal{CTA}$ level and the other one defined at the $\mathcal{TA}^\star$ level, are equivalent (i.e., they provide the same results for the same continuous query).

Definition 18 (Sampling Operator$^{\mathcal{TA}^\star}$). *At execution time t, the evaluation delayed by δ of an algebraic expression $E = E_{\mathcal{TA}^\star}(\alpha_1(\omega_1(S_1)), \ldots, \alpha_n(d_n(S_n)), R_1, \ldots, R_m, Q_1, \ldots, Q_\ell) \in \mathcal{CTA}$, with an historical parameter hp, slide parameter sl and alignment parameter a, at the $\mathcal{TA}^\star$ level is defined*

by the sampling operator $\xi^{\mathcal{TA}^\star} : \mathcal{TA}^\star \times \mathcal{T} \times \mathcal{I}^4 \to \mathcal{S}^{\mathcal{TA}^\star}$ *as follows:*

$$\xi^{t,\delta}_{hp,sl,a}(E)^{\mathcal{TA}^\star} := E_{\mathcal{TA}^\star}\big((\alpha_1({}^{tset}\omega_1(S_1^{t+\delta})))^{\mathcal{TA}}, \ldots, (\alpha_n({}^{tset}\omega_n(S_n^{t+\delta})))^{\mathcal{TA}}, {}^{tset}R_1^{t+\delta}, \ldots, {}^{tset}R_m^{t+\delta}\big)$$

where $tset$ *is the evaluation time instant set:* $tset = \{t' \mid t' \leq t \wedge t' = (\lceil \frac{t-hp-a}{sl} \rceil + i) \cdot sl + a$ *for some* $i \in \mathbb{N}\}$ *and* ${}^{tset}R_j^{t+\delta} = \bigcup_{t_i \in tset} \tau_{t_i}(R_j^{t+\delta})$*, with* $1 \leq j \leq m$*.*

The following Theorem shows the correctness of the translation. It is worth noting that correctness is shown for $\mathcal{CTA}$ algebraic expressions that include legal sliding window expressions w only.

Theorem 1. *Given the continuous query* $E = E_{\mathcal{TA}^\star}(\alpha_1(\omega_1(S_1)), \ldots, \alpha_n(\omega_n(S_n)), R_1, \ldots, R_m) \in \mathcal{CTA}$ *(where* R_j*, with* $1 \leq j \leq m$*, can be a temporal table or a streaming table* S *used as a temporal table, that is without the interposition of sliding window operators, or a non-temporal* $\mathcal{TA}^\star$ *subexpression virtually converted to temporal), with slide parameter* sl *and alignment parameter* a*, then, for each execution time* t *with delay* δ*:*

$$\xi^{t,\delta}_{hp,sl,a}(E)^{\mathcal{CTA}} = \xi^{t,\delta}_{hp,sl,a}(E)^{\mathcal{TA}^\star}$$

where $\xi^{\mathcal{CTA}}$ is the sampling operator defined for $\mathcal{CTA}$ in Def. 16.

Proof. For the sake of simplicity, we assume $\delta = 0$ and replace each R^t and S^t in the operator semantics with R and S, respectively (the proof can be straightforwardly adapted to the case when $\delta > 0$). First, we can observe that:

$$\begin{aligned}
&\xi^{t,\delta}_{sl,a}(E)^{\mathcal{CTA}} \\
&= \Big(\bigcup_{i=0}^{\max\{k \in \mathbb{N} \mid t_k \leq t\}} \tau_{t_i}\big(E_{\mathcal{TA}^\star}(\alpha_1(\omega_1(S_1)), \ldots, \alpha_n(\omega_n(S_n)), R_1, \ldots, R_m)\big) \Big)^{\mathcal{CTA}} \\
&= \Big(E_{\mathcal{TA}^\star}\big(\bigcup_{i=0}^{\max\{k \in \mathbb{N} \mid t_k \leq t\}} \tau_{t_i}(\alpha_1(\omega_1(S_1))), \ldots, \\
&\qquad \bigcup_{i=0}^{\max\{k \in \mathbb{N} \mid t_k \leq t\}} \tau_{t_i}(\alpha_n(\omega_n(S_n))), {}^{tset}R_1, \ldots, {}^{tset}R_m\big) \Big)^{\mathcal{CTA}}
\end{aligned}$$

where ${}^{tset}R_j = \bigcup_{t_i \in tset} \tau_{t_i}(R_j) = \bigcup_{i=0}^{\max\{k \in \mathbb{N} \mid t_k \leq t\}} \tau_{t_i}(R_j)$ $(1 \leq j \leq m)$.

Hence, recalling that non-temporal subexpressions (which may appear among R_js) cannot contain sliding window expressions and, thus, $(R_j)^{\mathcal{CTA}} = (R_j)^{\mathcal{TA}^\star} = R_j$ $(1 \leq j \leq m)$, if we show that $\bigcup_{i=0}^{\max\{k \in \mathbb{N} \mid t_k \leq t\}} \tau_{t_i}(\alpha_j(\omega_j(S_j))^{\mathcal{CTA}}) =$

$(\alpha_j({}^{tset}\omega_j(S_j)))^{\mathcal{TA}}$, then we can write:

$$E_{\mathcal{TA}^\star}\Big(\bigcup_{i=0}^{\max\{k\in\mathbb{N}|t_k\leq t\}} \tau_{t_i}(\alpha_1(\omega_1(S_1))),\ldots,$$
$$\bigcup_{i=0}^{\max\{k\in\mathbb{N}|t_k\leq t\}} \tau_{t_i}(\alpha_n(\omega_n(S_n))),\ {}^{tset}R_1,\ldots,{}^{tset}R_m\Big)^{\mathcal{CTA}}$$
$$= E_{\mathcal{TA}^\star}\big((\alpha_1({}^{tset}\omega_1(S_1)))^{\mathcal{TA}},\ldots,(\alpha_n({}^{tset}\omega_n(S_n)))^{\mathcal{TA}},{}^{tset}R_1,\ldots,{}^{tset}R_m\big)$$

And, thus, the proof of the theorem follows: $\xi^{t,\delta}_{sl,a}(E)^{\mathcal{CTA}} = \xi^{t,\delta}_{sl,a}(E)^{\mathcal{TA}^\star}$.

To this end, as far as $\alpha_j(\omega_j(S_j))$ is concerned, all possible operator combinations should be considered. Let us first consider the case when $\alpha_j = \theta_F$ and $\omega_j = w^{\text{time}}_{[d_1,d_2]}$. Given $\theta_F(w^{\text{time}}_{[d_1,d_2]}(S_j))$, in the following we will show that $s \in \bigcup_{i=0}^{\max\{k\in\mathbb{N}|t_k\leq t\}} \tau_{t_i}(\theta_F(w^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{CTA}})$ iff $s \in (\theta_F({}^{tset}w^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{TA}}$. Let $s \in \bigcup_{i=0}^{k:t_k\leq t} \tau_{t_i}(\theta_F(w^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{CTA}})$, then $s = (X,t_i) \in \tau_{t_i}(\theta_F(w^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{CTA}})$ for some i. Being $s = (X,t_i) \in \tau_{t_i}(\theta_F(w)^{\mathcal{CTA}})$, where $w = w^{\text{time}}_{[d_1,d_2]}(S_j))$, then, according to the $\theta_F(w)$ semantics, $s \in \tau_{t_i}(\theta_F(w)^{\mathcal{CTA}})$ iff $X = (f_1(S),\ldots,f_h(S))$ and $(S,t_i) \in w$. This means that $S = \text{Sub}^T_{[t_i-d_1,t_i+d_2]}(S_j)^{\mathcal{CTA}} = \{(s,\tau)\,|\,(s,\tau) \in S_j, (t_i-d_1) \leq \tau \leq (t_i+d_2)\}$. Hence, $(s,\tau) \in \text{Sub}^T_{[t_i-d_1,t_i+d_2]}(S_j)^{\mathcal{CTA}}$ iff $(s,\tau,t_i) \in \tau_{t_i}(\text{Sub}^T_{[t_i-d_1,t_i+d_2]}(S_j)^{\mathcal{TA}})$. As $t_i \in tset$, it follows that:

$$(s,\tau,t_i) \in \bigcup_{t\in tset} \tau_t(\text{Sub}^T_{[t-d_1,t+d_2]}(S_j)^{\mathcal{TA}}) =^{tset} w^{\text{time}}_{[d_1,d_2]}(S_j)^{\mathcal{TA}}$$

From $S' = \tau_{t_i}(\text{Sub}^T_{[t_i-d_1,t_i+d_2]}(S_j)^{\mathcal{TA}})$ and $X' = (f_1(S'),\ldots,f_h(S'))$, it follows that $(X',t_i) \in {}_{\emptyset}\vartheta^T_F(S_j)^{\mathcal{TA}}$, which is equivalent to say that $(X',t_i) \in (\theta_F(w^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{TA}}$. Being $X = X'$ then $s \in \bigcup_{i=0}^{\max\{k\in\mathbb{N}|t_k\leq t\}} \tau_{t_i}(\theta_F(w^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{CTA}})$ iff $s \in (\theta_F({}^{tset}w^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{TA}}$.

When $\alpha_j = \varphi$, X and X' definitions change but the equivalence still holds. Instead, when $\omega_j = w^{\text{count}}_{[n_1,n_2]}$, notice that $S \bowtie^T_{\substack{S.T=P.T \\ \wedge \texttt{PREV}(n_1)\leq S.\nu\leq \texttt{NEXT}(n_2)}} P$ contains the n_1 closest tuples of S inserted before and the n_2 closest tuples of S inserted after each explicit timestamp in S. Therefore, the equivalence still holds.

Finally, whenever partitioned operators are concerned, let us consider the case when $\alpha_j =_B \Theta_F$ and $\omega_j = W^{\text{time}}_{[d_1,d_2]}$. Given ${}_B\Theta_F(W^{\text{time}}_{[d_1,d_2]}(S_j))$, with $B = \{B_1,\ldots,B_n\}$ grouping attributes, in the following we will show that $s \in \bigcup_{i=0}^{\max\{k\in\mathbb{N}|t_k\leq t\}} \tau_{t_i}({}_B\Theta_F(W^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{CTA}})$ iff $s \in ({}_B\Theta_F({}^{tset}W^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{TA}}$. Let $s \in \bigcup_{i=0}^{k:t_k\leq t} \tau_{t_i}({}_B\Theta_F(W^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{CTA}})$, then $s = (X,\tau) \in \tau_{t_i}({}_B\Theta_F(W^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{CTA}})$ for some i. Being $(X,\tau) \in \tau_{t_i}({}_B\Theta_F(W)^{\mathcal{CTA}})$, where $W = W^{\text{time}}_{[d_1,d_2]}(S_j))$, then, according to the ${}_B\Theta_F$ semantics, $s \in \tau_{t_i}(\theta_F(w)^{\mathcal{CTA}})$ iff it exists a streaming table of streaming tables $w \in W$ such that $(S,t_i) \in w$, for each $(s,\tau) \in S$, $s.B_i = b_i$, for $i \in [1,n]$, and $X = (b_1,\ldots,b_n,f_1(S),\ldots,f_h(S))$. Notice that,

from the definition of $W^{\text{time}}_{[d_1,d_2]}$, $w = {}^{t}\bar{w}^{\text{time}}_{[d_1,d_2]}(S'_j)$ where $S'_j \in \zeta_B(S_j)$. Then, the proof easily follows from the proof above concerning the sliding window operator ${}^{t}\bar{w}^{\text{time}}_{[d_1,d_2]}$ and the proposed translation of ${}_{B}\Theta_F(W^{\text{time}}_{[d_1,d_2]}(S_j))$ in $\mathcal{TA}$, i.e., ${}_{B}\Theta_F(W^{\text{time}}_{[d_1,d_2]}(S_j))^{\mathcal{TA}}$. The other combinations involving the partitioned operators can be dealt with in a similar way. In particular, in the case of $\omega_j = W^{\text{count},B}_{[n_1,n_2]}$ the grouping P_B and the temporal join also involves the explicit grouping attributes B, whose values are necessary to select in P_B the right ν selection range and in S the right tuples to be included in the window. □

Thanks to the above theorem ensuring that the semantics of execution is preserved, we can safely translate each continuous query in $\mathcal{CTA}$ into an equivalent expression in $\mathcal{TA}^{\star}$ and execute it on a temporal database or even on a traditional relational engine (with an extended kernel supporting the execution of $\mathcal{TA}$ expressions as shown in [16]).

9. Related Work

In the following, we will provide a general overview of the status of current data management proposals with respect to the requirements we are targeting in this paper: the availability of a query language that supports the combination of CQs and OTQs on streaming and relational data, including temporal conditions to be evaluated according to different query semantics (sequenced and/or non-sequenced). In particular, Sec. 9.1 focuses on literature approaches, while Sec. 9.2 analyzes available systems and frameworks. We start with a short discussion of our contribution.

The algebra we conceived in this paper with its TSQL2-like query language counterpart is the ultimate result of a unified framework grounded on two main pillars: our contribution on streaming tables [13], bringing together for the first time the streaming and temporal database worlds, and the work in [16], presenting a first proof-of-concept that an interval-based temporal DBMS can be successfully implemented on top of standard RDBMS technology. On the one hand, we have brought under a unifying semantic umbrella streaming data as well as temporal relational data, so that they can be queried in a seamless way through the $\mathcal{CTA}$ we introduced. On the other hand, we have extended the approach in [16] by "wrapping" the temporal algebra $\mathcal{TA}$ with a formal framework that guarantees complex CQs including full-fledged temporal conditions entailing a hybrid semantics (sequenced and/or non-sequenced) to be rewritten, optimized, and executed on top of a traditional DBMS (with a kernel extended as shown in [16]).

Starting from the groundwork in [14], one main contribution of this paper consists in the extension of $\mathcal{TA}$ in [16], which supports only sequenced temporal queries, to $\mathcal{TA}^{*}$, which supports also non-sequenced operations within the execution of a sequenced temporal query through the regulated use of non-temporal operators. This extension opens a wider range of querying possibilities, most importantly the answering of *real temporal queries* as well as OTQs and CQs over streaming, relational and temporal data. In this way, this paper extends and

completes the semantics of queries introduced in [14] by regrounding the entire framework from $\mathcal{TA}$ to $\mathcal{TA}^{\star}$, and by introducing the concept of *legal sliding window expression* together with an exhaustive discussion on the composition rules of $\mathcal{CTA}$ operators. This result represents a further demonstration of the advantages provided by an algebra and advances our work in [14] by regulating how continuous queries can be properly specified. The proposal is enriched by a collection of representative TSQL2-like queries and of their translation into the continuous algebra $\mathcal{CTA}$ which exemplifies the usefulness of the proposed approach in the context of a market surveillance application.

9.1. Discussion on Query Languages and Algebras for Querying Streaming and/or Temporal Data

In this section we discuss literature approaches for querying streaming and/or temporal data together with traditional relational data. Sections 9.1.1 and 9.1.2 focus on temporal/relational and streaming/relational aspects, respectively.

9.1.1. On Temporal Data Querying and Querying Semantics

In the jungle of the dozens of proposed temporal query languages, we can consider TSQL2 [15] as a milestone. Its design was the outcome of a great consensual effort aimed at merging a rigorous theoretical foundation with a great expressive power and user friendliness, such that it could become a standard proposal for the temporal extension of SQL. TSQL2 was equipped with a mix of sequenced features (e.g., temporal intersection semantics based on syntactic defaults) and non-sequenced features (e.g. explicit rendering of implicit timestamps enabling real temporal queries). However, their coexistence was not resolved at the level of the proposed algebra [15, Ch. 27], whose operators were generic enough to support both kinds of semantics. A comprehensive mapping from the language syntax to algebraic expressions giving to the `SELECT` statement a precise semantics was lacking indeed.

The temporal query language ATSQL [17], which has been designed to supersede TSQL2, tried to solve its defects with an approach based on the separation of concerns. In ATSQL, distinct modalities of temporal execution following the sequenced or non-sequenced semantics can be selected through the use of statement modifiers `SEQ` and `NSEQ`, respectively, prefixing each query. In such a way, the advantages of a *hybrid* semantics (i.e., partly sequenced and partly non-sequenced) are lost: pure sequenced queries suffer from a very limited expressiveness and pure non-sequenced queries waive the benefits of a sequenced execution, including the efficient support of interval-timestamping (and change preservation) with the implementation techniques presented in [16]. For instance, the hybrid query (Q3) of Sec. 3 could be expressed in ATSQL only by means of two independent statements nested as follows:

```
SEQ VT
SELECT ID, OPTION
FROM LATEST_NEWS AS LN, OPTION_TRADES AS OT
WHERE LN.STOCK = OT.STOCK AND ID =
```

```
( NSEQ VT
  SELECT LN1.ID
  FROM LATEST_NEWS AS LN1 LN2
  WHERE LN1.SOURCE = LN2.SOURCE AND LN1.STOCK = LN2.STOCK
    AND LN1.TYPE = LN2.TYPE
    AND END(VTIME(LN1)) = BEGIN(VTIME(LN2)) )
```

Since the ATSQL outer and inner queries of the example have different semantics, they must be evaluated separately. The outer query with the `SEQ VT` modifier must be translated into a snapshot-reducible temporal algebra (like $\mathcal{TA}$) in order to be evaluated with a sequenced semantics, whereas the inner query with the `NSEQ VT` modifier must be translated into a non snapshot-reducible temporal algebra (like the traditional relational algebra) in order to be evaluated with a non-sequenced semantics. Since the resulting algebra expressions have different semantics and must be evaluated separately, they can only be optimized separately. On the contrary, using our hybrid $\mathcal{TA}^\star$ algebra, the whole query can be translated, as shown in Sec. 7, into a single algebraic expression which can be evaluated as a whole, and which is liable to cost-based or semantic optimization (e.g. by means of rewriting rules) as a whole.

In [41], the authors aim at reconciling the sequenced and non-sequenced viewpoints by providing a unifying framework where queries can be evaluated using different temporal semantics (also including context, periodic and preceding semantics). In such an approach, the execution semantics can be selected with lightweight annotations preceding the queries and, thus, different temporal semantics can be selected for the execution of different queries but all the parts of a single query are evaluated with the same semantics. Hence, there are no substantial differences with ATSQL as to single query expressiveness and global optimization opportunities.

In our approach, we took a different road leading to the reconciliation of the two viewpoints by allowing for individual queries to be evaluated with a hybrid semantics, as the $\mathcal{TA}^\star$ algebra allows to merge non-sequenced parts into a sequenced $\mathcal{TA}$ expression in order to extend its query expressiveness. Hence, the coexistence of the different temporal semantics is within single queries rather than between separate queries, and sequenced and non-sequenced parts can be combined in the same $\mathcal{TA}^\star$ expressions to support a wider range of temporal query types (with the same expressive power of the Temporal Relational Calculus, having explicit access to timestamp values, as shown in Lemma 1) but also preserving, as much as possible, the advantages of a sequenced approach. In this way, we finally gave a solid foundation to the hybrid semantics of user-friendly TSQL2 queries, showing how generic SPJ queries can be translated into our $\mathcal{TA}^\star$ algebra.

The present work extends the preliminary version in [14] under several directions, most importantly the specification of queries that include complex temporal conditions, the exploration of the expressiveness of the proposed algebra and the provision of a semantics for TSQL2-like hybrid queries.

Notice that also recent works on semantic aspects of temporal data man-

agement in DBMSs (such as [42]) limit their approach to different assumptions (focus on data with a single time dimension and on pure snapshot-reducible algebras).

9.1.2. On Streaming Data Querying and Querying Semantics

Many works on Data Stream Management Systems (DSMSs) exist in literature, many of them including theoretical frameworks on algebras for querying streaming data. DSMSs (among them, Stream Mill [43], STREAM [23], Aurora [44], TelegraphCQ [45], NiagaraCQ [46], Gigascope [47]) natively support CQs over continuous unbounded streams of data according to windows where only the most recent data is retained [48]. This kind of research is mainly focused on the performances of both stream data management and querying rather than on semantics. Some works specifically concentrate on semantics but in a very different context: for instance [49] addresses the problem of synopsis construction with the goal of keeping in the system the smallest amount of data w.r.t. query requirements, while our assumption is that data never leaves the system. Generally, (e.g., in CQL [23] and SyncSQL [50], but also in more recent proposals [51]) streams are transformed into instantaneous/synchronized relations that are manipulated through relational operators, and then transformed back to streams. Under a more general perspective, these approaches do not actually deal with both streaming and traditional relational data in a unified model. In this paper we prove that it is possible to exploit the full potential of a unified approach, by preserving the standard (and temporal) semantics of (T)RDBMS's algebras while overcoming the transformation overhead of stream-relation-stream approaches like [23, 50].

As a matter of fact, some works charge DSMSs with short-sightedness as to treating stream processing distinct from traditional data processing [8–10]. Thus, recent research proposals extend traditional DBMSs' query model and language towards streaming query capabilities [9, 52]. However, these works present extensions to SQL through query examples and do not offer a formal algebraic framework for a clear specification of query operators and their semantics. The only paper offering a fully native representation of streaming data in a DBMS is [13].

The work in [35] presents a comprehensive description of window types that can be specified over data streams, and proposes an algebra for continuous query specification. However, the algebraic operators introduced (e.g., windowed selection, windowed join, etc.) are tightly bound to the window types, thus resulting in windowed operators that absorb windows in their definition. In our work instead we propose a sharp distinction between standard algebraic operators, belonging to $\mathcal{TA}^{\star}$, and window operators, defined in $\mathcal{CTA}$. Such a twofold view offers two main advantages: a clearer semantics of operators, and more flexibility in formulating algebraic expressions, thus also leading to a more accurate optimization. Moreover, both in SQL:2011 and proposed stream query languages, sliding windows can only be used, via aggregation operators, to produce results in the target list of a query, whereas $\mathcal{CTA}$ allows us to use them everywhere (e.g., in a selection predicate as in our running example). To

the authors' knowledge, these features are not covered by any existing approach dealing with streaming data.

An approach similar to ours is followed in [53], where operators for window specification are kept separate from standard algebraic operators. The introduced operators are tailored to deal with a multiset representation of temporal data in a novel logical model and are implemented on ad-hoc data structures for state maintenance under a time-interval approach. Snapshot reducibility is demonstrated for standard operators. Window operators instead overwrite tuples' original timestamps with window evaluation time and thus they are not snapshot-reducible. This approach does not integrate with the theoretical and practical solutions proposed for the development of a robust temporal database technology, including [16] which presents a proposal for the implementation of a standard temporal algebra in an off-the-shelf DBMS, supporting a sequenced semantics that guarantees extended snapshot reducibility. As to $\mathcal{CTA}$ operators, we proved this fundamental requirement by showing how $\mathcal{CTA}$ expressions can be translated into equivalent expressions in the temporal algebra $\mathcal{TA}^{\star}$.

An additional note concerns the semantics of ad-hoc proposals of temporal operators that often proves to be ambiguous as far as timestamp management is concerned. A consensus is not shared among existing approaches. For instance, the timestamps of tuples resulting from a windowed join can be either the minimum of the two original timestamp values [35, 44], or the most recent one [54], or the time instant at which the join is executed [55]. Operators in $\mathcal{TA}$ follow a precise and commonly adopted sequenced semantics [16], and a temporal join is always performed between pairs of tuples having overlapping timestamp, thus being valid at the same time instants. The non-sequenced part of expressive temporal queries (e.g., necessary to interoperate data belonging to different temporal snapshots) is captured by the $\mathcal{TA}^{\star}$ extensions we proposed for $\mathcal{TA}$. Furthermore, as noticed about [53], also in [23] windowing operators overwrite the original timestamp of tuples with a new timestamp corresponding to the window evaluation time instant. $\mathcal{CTA}$ operators maintain instead both this information and the original tuple timestamp, thus not losing relevant information concerning the lineage of streaming data, to be used for further processing. Such "provenance witness" timestamps play a role similar to the lineage sets used in [16] to define change-preserving operators. This feature enables $\mathcal{CTA}$ windowing operators to be translated into $\mathcal{TA}$ and, thus, to be snapshot-reducible on valid time.

A further property featured by $\mathcal{CTA}$ is the capability of defining *forward windows*, thus opening to the possibility of evaluating queries (possibly in an approximated way) by referring tuples that will be observed after a given time instance, as proposed for SQL:2011 [36] (e.g., SQLStream Blaze [4] considers forward windows in its query syntax specifications but it does not provide an implementation for them).

9.2. Data-intensive Systems and Frameworks

From a system perspective, a great amount of software frameworks for the management of large volumes of data has flourished in recent years. However,

none of them offer a full-fledged answer to the challenging data management objectives we pointed out in the introduction and that motivated this work.

DSMSs do not support data persistency and/or temporal versioning: the common assumption is "all past tuples cannot be memorized" [43]. This means that, for instance, jointly querying current data with historical data changing over time is not directly supported and requires the overhead of interfacing with other data management systems, with performance and usability problems. The large number of available commercial DSMSs, such as SQLStream Blaze [4], IBM Infosphere Streams [5] and Oracle Fusion [6] are conceived under the same assumptions.

9.2.1. Joint Management of Streaming and Relational Data

With the advent of the Big Data era, a large number of novel data management proposals has tried to revolutionize data management, offering extensible architectures able to cope with different kinds of data, including streams and tabular data. Following the guidelines of the Lambda architecture [56], distributed realtime computation systems such as Apache Storm [57] have appeared and have been coupled with distributed computing platforms such as Apache Hadoop [58] and NoSQL databases such as Amazon Dynamo [59] in order to provide a complete solution to real-time (online) processing and batch (offline) processing of tabular and streaming data, but with big disadvantages for users: duplicative development effort in systems offering different data/querying models and languages (e.g., for defining online or offline processing pipelines, for working on tabular or streaming data, etc.), and additional overhead for reprocessing/merging data between them. Further systems have then been presented, trying to provide larger flexibility in a single data management solution, as also to the guidelines of the Kappa architecture [60]: big data processing engines such as Apache Spark [3], big data stores such as Apache HBase [61], stream processing frameworks such as Apache Flink [12, 62] and Apache Samza [7]. The extensibility and modularity of their architecture allows them to offer data models often supporting different kinds of data: for instance, Spark data model based on finite Resilient Distributed Datasets (RDDs) is extended towards streaming (infinite) data by means of the SparkStreaming [63] extension. The same applies to their querying model: users can typically implement the kind of operators they want, and, besides typical windowing operators applying to streams, some systems offer the possibility to extend their data and querying model towards the relational world, including aggregates and joins. For instance, Flink "base" data model is based on the low-level "dataset" and "datastream" data types; in order to apply relational operators (managed through Apache Calcite [64]) on such data, these need to be transformed in "table" data types (which include "dynamic tables" to manage streaming data). This is also the case of other relational add-ons such as StormSQL, SparkSQL [28], SamzaSQL [65] and Apache Phoenix [66] for HBase.

On the one hand, such variety of extensions brings great flexibility as to query answering. On the other hand, it essentially introduces fragmentation. From a practical point of view, although offering high performances, such

extensions do not provide a real synergic approach to querying streaming and relational data under a unified perspective. For instance, HBase extended with Phoenix supports SQL queries but still not streaming data; SparkStreaming and SparkSQL extensions work together only by introducing substantial overheads (stream data each time needs to be converted and registered as a temporary table to be queried using SQL); similarly, in SamzaSQL and Flink relational operators need to transform the data between the base and relational model, whereas joining a table and a stream requires reading the whole table data in memory ("bootstrap") in order to transform it to a stream. Moreover, joins between tables and streams are supported only under specific circumstances (for instance, Flink only supports joining append-only tables). This confirms that (a) the typical drawbacks of DSMSs when dealing with tabular data are still present and (b) the streaming/continuous model and the relational model are kept as two separate worlds.

With reference to the example application domain considered in this paper, also the work in [67] acknowledges the need by financial services of integrating real-time and historical analytics seamlessly on streaming and relational data. [67] presents the implementation of a framework that relies on an extended relational algebra for internal query representation as a means for translating queries expressed in the Q language (a highly domain-specific query language tailored to time series analytics) to SQL. The focus of the paper is on the efficiency of query translation, rather than on providing a comprehensive syntax and semantics of the algebra, and query translation is shown through examples.

Ultimately, these systems certainly have strong scalability and performances in specific use-cases but, since they do not rely on a traditional RDBMS architecture, they are not able to exploit the full expressive power of relational algebra and the well-established optimization power of RDBMSs' query engines. Rather, being based on novel ad-hoc implementations and extensions providing limited features in specific cases, they show substantial limitations in expressing generalized queries and inefficiencies in executing them.

9.2.2. Systems Dealing with Time

With regard to temporal capabilities, DSMSs (e.g., [23]) include a timestamp in each data tuple, representing the time the tuple is created. Some stream processing frameworks (e.g., Storm [57]) give the chance either to work without explicit timestamps (i.e., window calculations are performed based on the time when the tuple is processed), or to use source-generated timestamps (e.g., event time, for processing events based on the time when an event occurs). Flink [12] also offers a third option of working with ingestion time (the same as transaction time, i.e., when the data enters the dataflow). However, even if users can typically extend these models (and define custom operators that have to be programmed from scratch), none of the approaches provide the flexibility of automatically maintaining both the streaming data original timestamps and the window evaluation time, as we do when evaluating window expressions. By envisioning tuples' original timestamps as streaming data transaction time values, our approach permits the translation of CQs in OTQs and it enables

the implementation of the continuous temporal model on a temporal relational engine supporting an extended sequenced semantics.

When talking about time management, the support for time versioning and sequenced semantics as offered by TDBMSs is particularly crucial: while this is typically of little use in a pure DSMS, it instead becomes fundamental when supporting queries involving streams and relational data that changes over time (e.g., in temporal joins). Quite surprisingly, considering the temporal nature of a data stream, most existing data-intensive systems (i.e., both DSMSs and Big Data frameworks) do not support built-in temporal versioning for data. HBase/Phoenix and major standard RDBMSs do, but no streaming support is provided though. Flink is quite an exception and provides some kind of support to versioning by offering the "temporal table" abstraction. Temporal tables can be used in queries together with the other abstractions (e.g., dynamic tables for streaming data), but with a number of limitations similar to the ones already discussed for relational/streaming management: joining dynamic and temporal tables is again supported only under specific circumstances (e.g., joins require users to materialize portions of the temporal tables through ad-hoc programmed temporal table functions, joins with event time tables are not supported, etc.).

Most importantly, even if some systems (e.g., Flink [12] and Samza [65]) underline the importance of the so-called *replay principle* (i.e., a streaming query should produce the same result as the corresponding non-streaming query would if given the same data in a table), none of them fully support temporal database semantics. While our approach is designed to preserve as much as possible a sequenced semantics, while also allowing for non-sequenced operations, DSMSs and Big Data frameworks are not designed on this assumption and leave sequenced semantics support to the programmers' skill: in order to obtain standard sequenced semantics behavior on a query involving temporal tables, the user has to manually select relevant slices and specify how to deal with the involved times for each required operation (e.g., joins).

In short, most approaches combine streaming features with desirable (T)DBMSs features, albeit without directly exploiting well-established (T)DBMSs' research and implementations, often with uneven results.

An equally hot research topic has been extending standard DBMSs towards a flexible and efficient management of temporal and versioned data: even if some partial temporal support is becoming widespread also in commercial systems [68], much research is still needed in order to fully support such capabilities. For instance, the seminal work [16] provides several breakthroughs towards this direction, even if basically focused on the management of temporal queries with sequenced semantics. All in all, grounding our research on such recent (T)DBMS research (including extensions toward the streaming world [13]) enables our $\mathcal{CTA}$ proposal to leverage on the well-founded (temporal) query semantics and on the underlying consolidated (T)DBMS technology (e.g., regarding query optimization), provided the kernel extensions presented in [16].

10. Concluding Remarks and Future Research Directions

The ongoing research and the discussed works on the management and querying of streaming, temporal and standard relational data are certainly promising. However, there is still much work to be done in order to have an actual "all-in-one" answer to all the modern application requirements we discussed in the introduction. Following the first step provided by the streaming table concept we introduced in [13], in this paper we present a full-fledged TSQL2-like query language based on a novel temporal algebra extended with windowing and aggregation operators supporting both OTQs and CQs on streaming, standard and temporal relational data. Continuous queries are proved to be correctly translated into temporal OTQs, thus paving the way to the implementation on a relational engine extended with temporal capabilities. With the aim of avoiding any ambiguity, our approach complies to a formal scheme that guarantees correctness of query rewriting, which is known to have a crucial impact on design and implementation issues like, for instance, query optimization.

Our proposal is highly flexible in that it allows to combine in a unified framework several continuous query features proposed in the literature (e.g., real time and historical analytics, backward and forward sliding windows, tumbling and hopping windows) but for which a well-founded semantics and full implementation agenda is still lacking, and to interoperate streaming data with non-temporal data and archival data stored in temporal tables in a consistent way.

In our near future work, we plan to explore algebraic optimization issues and indexing techniques to efficiently support the implementation of $\mathcal{CTA}$ operators in a temporal DBMS or in a traditional DBMS extended with temporal capabilities. Optimization issues include the efficient support of the hybrid sequenced/non-sequenced semantics introduced in this work, requiring techniques for the combined optimization of $\mathcal{TA}^{\star}$ subexpressions containing non-temporal operators involving costly non-sequenced temporal joins and $\mathcal{TA}$ expressions which can be dealt with as shown in [16]. A further extension concerns the possibility of seamlessly maintaining two or more times for each tuple. This feature would enable a richer variety of querying possibilities, for instance through operators involving together event and transaction time, as well as out-of-order and skewed inputs' management [24]. Our final aim is to exploit all the consolidated and universally adopted technology a standard temporal framework provides, in order to complete the integration of streaming tables in the context of a temporal RDBMS, breaking the traditional barrier between the streaming and the temporal worlds.

A medium-term research direction we intend to investigate concerns the study of metodologies and techniques for modeling/processing linked stream data, a way in which the Linked Data principles can be applied to stream data and be part of the Web of Linked Data [69]. To this end, we will blend our long-standing know-how in the field of semantic data modeling and processing [70, 71], also including temporal aspects [72, 73], with the expertise we have gathered with stream data management [13, 74, 75]. While some works discuss temporal

querying of RDF data [73, 76] as well as querying of streaming RDF data [77, 78], these issues are dealt with under different and separate perspectives, depending on their specific querying goals involving temporal or streaming query capabilities, respectively. In this context, we plan to apply the same principles underlying the comprehensive approach adopted in this paper also to this stimulating scenario. We expect that a unified querying model, like the one we proposed, will very likely transpose (most of) the advantages we discussed in this paper also to the RDF data context. We consider this topic a very promising one that would certainly boost a new range of real-time applications.

A further related topic we are interested in regards reasoning on semantic data streams. Our purpose is to take advantage of the amount of existing work on stream reasoning [79] to study the application of inference techniques to highly dynamic *semantic* data. We deem this issue as a prominent and challenging goal for the definition of innovative means for data analysis in complex environments like, for instance, smart factories in the Industry 4.0 scenario under development.

Another scenario we would like to explore is the management of data streams with intrinsic interval-based rather than point-based semantics. Intelligent sensors can be thought to produce streams of data concerning the accomplishment of some kind of complete tasks and for which an interval timestamping is needed, as they represent *telic* data [80]. Telic facts (like "the patient had one 100mg intravenous drug infusion from 10:00 to 10:45") are true on an interval but false on any superinterval or subinterval of it, and need to be manipulated with an interval temporal logic or algebra. Extending our framework to the management of interval-based data streams and telic temporal data [81] is another challenge we plan to accept.

A further extension that we intend to consider consists in providing pattern detection capabilities, which is a typical feature offered by event processing systems on data streams (for instance, Flink enables it through the `match_recognize` clause). Since this kind of operation founds on rather different principles than those underlying a DBMS's perspective, in order to efficiently implement this ability in our framework, we plan to extend our algebra with a specific pattern matching operator that, by exploiting the ordering of tuples in a streaming table, would select the desired patterns as a preprocessing operation before the application of standard algebraic operators.

In the long term, we also wish to explore the potentialities offered by the availability, at the data model and at the query language level, of complex ¬1NF data types including sets of streaming tables, streaming tables of streaming tables and sets of streaming tables of streaming tables. Such a possibility has been explicitly forbidden in the current approach as it was unnecessary for the purposes of the present work, but could reveal elements of interestingness for particular applications. In this vein, another constraint present in the current approach that could be relaxed concerns the use of sliding window expressions, which could be allowed to also have generic temporal expressions as arguments and to appear inside non-temporal algebra subexpressions.

References

[1] J. Ramnarayan, K. Bachhav, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, Snappydata: A hybrid transactional analytical store built on spark, 2016, pp. 2153–2156. doi:10.1145/2882903.2899408.

[2] D. Stripelis, J. L. Ambite, Y.-Y. Chiang, S. Eckel, R. Habre, A scalable data integration and analysis architecture for sensor data of pediatric asthma, Vol. 2017, 2017, pp. 1407–1408. doi:10.1109/ICDE.2017.198.

[3] Apache Spark, `http://spark.apache.org`.

[4] Guavus SQLStream, `http://sqlstream.com`.

[5] IBM Infosphere Streams, `http://www.ibm.com/software/products/it/ibm-streams`.

[6] Oracle Fusion, `http://www.oracle.com/it/middleware`.

[7] Apache Samza, `http://samza.apache.org`.

[8] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, S. Zdonik, S-Store: A Streaming NewSQL System for Big Velocity Applications, Proc. VLDB 7 (13) (2014) 1633–1636.

[9] Q. Chen, M. Hsu, Cut-and-rewind: Extending query engine for continuous stream analytics, in: A. Hameurlain, J. Kueng, R. Wagner, A. Cuzzocrea, U. Dayal (Eds.), TLDKS XXI, Vol. 9260 of LNCS, Springer, 2015, pp. 94–114.

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

[11] R. T. Snodgrass, Developing Time-Oriented Database Applications in SQL, Morgan Kaufmann, 1999.

[12] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.

[13] L. Carafoli, F. Mandreoli, R. Martoglia, W. Penzo, Streaming Tables: Native Support to Streaming Data in DBMSs, IEEE Trans. on Systems, Man, and Cybernetics: Systems 47 (10) (2017) 2768–2782.

[14] F. Grandi, F. Mandreoli, R. Martoglia, W. Penzo, A Relational Algebra for Streaming Tables Living in a Temporal Database World, in: Proc. of TIME Symposium, LIPICS, Mons, Belgium, 2017, pp. 15:1–15:17.

[15] R. Snodgrass (ed.), I. Ahn, G. Ariav, D. Batory, J. Clifford, C. Dyreson, R. Elmasri, F. Grandi, C. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. C. Leung, N. Lorentzos, R. Ramakrishnan, J. Roddick, A. Segev, M. Soo, S. Sripada, The TSQL2 Temporal Query Language, Kluwer, 1995.

[16] A. Dignös, M. H. Böhlen, J. Gamper, C. S. Jensen, Extending the kernel of a relational dbms with comprehensive support for sequenced temporal queries, ACM Trans. Database Syst. 41 (4) (2016) 26:1–26:46.

[17] M. H. Böhlen, C. S. Jensen, R. T. Snodgrass, Temporal statement modifiers, ACM Trans. Database Syst. 25 (4) (2000) 407–456.

[18] F. Grandi, Temporal interoperability in Multi+Temporal databases, J. Database Manag. 9 (1) (1998) 14–23.

[19] C. S. Jensen, C. E. Dyreson, M. H. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. J. Hayes, S. Jajodia, W. Käfer, N. Kline, N. A. Lorentzos, Y. G. Mitsopoulos, A. Montanari, D. A. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. U. Tansel, P. Tiberio, G. Wiederhold, The consensus glossary of temporal database concepts - february 1998 version, in: S. S. O. Etzion, S Jajodia (Ed.), Temporal Databases: Research and Practice, Springer-Verlag, 1998, pp. 367–405.

[20] J. Chomicki, Temporal query languages: A survey, in: Proceedings of the First International Conference on Temporal Logic, Springer-Verlag, 1994, pp. 506–534.

[21] C. D. Castro, F. Grandi, M. R. Scalas, Semantic interoperability of multitemporal relational databases, in: Proceedings of the 12th International Conference on the Entity-Relationship Approach, Springer-Verlag, 1993, pp. 463–474.

[22] J. Clifford, D. S. Warren, Formal semantics for time in databases, ACM Trans. Database Syst. 8 (2) (1983) 214–254.

[23] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution, VLDB J. 15 (2) (2006) 121–142.

[24] U. Srivastava, J. Widom, Flexible Time Management in Data Stream Systems, in: Proc. of ACM PODS, 2004, pp. 263–274.

[25] Insider Trading, `https://www.sec.gov/fast-answers/answersinsiderhtm.html`.

[26] S. Donoho, Early detection of insider trading in option markets, in: Proc. of the ACM KDD, ACM, 2004, pp. 420–429.

[27] F. Grandi, M. R. Scalas, P. Tiberio, A history-oriented temporal SQL extension, in: Proc. of Intl' Workshop on Next Generation Information Technologies and Systems (NGITS 95), Naharia, Israel, 1995, pp. 42–48.

[28] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, M. Zaharia, Spark sql: Relational data processing in spark, in: Proc. of ACM SIGMOD, ACM, 2015, pp. 1383–1394.

[29] A. U. Tansel, Temporal algebras, in: Encyclopedia of Database Systems, Springer US, 2009, pp. 2929–2932.

[30] J. Chomicki, D. Toman, M. H. Böhlen, Querying ATSQL databases with temporal logic, ACM Trans. Database Syst. 26 (2) (2001) 145–178.

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

[32] J. Chomicki, D. Toman, Temporal relational calculus, in: L. LIU, M. T. ÖZSU (Eds.), Encyclopedia of Database Systems, Springer US, Boston, MA, 2009, pp. 3015–3016.

[33] S. Abiteboul, L. Herr, J. V. der Busche, Temporal versus first-order logic to query temporal databases, in: Proc. of ACM PODS, ACM Press, 1996, pp. 49–57.

[34] D. Toman, D. Niwinski, First-order queries over temporal databases inexpressible in temporal logic, in: Proc. of EDBT, Springer, 1996, pp. 307–324.

[35] K. Patroumpas, T. Sellis, Window specification over data streams, in: Current Trends in Database Technology - EDBT 2006 Workshops, 2006, pp. 445–464.

[36] F. Zemke, What's new in SQL:2011, SIGMOD Rec. 41 (1) (2012) 67–73.

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

[38] D. Toman, Point vs. interval-based query languages for temporal databases, in: Proc. of ACM PODS, ACM Press, 1996, pp. 58–67.

[39] C. Li, K. C. Chang, I. F. Ilyas, S. Song, RankSQL: Query algebra and optimization for relational top-k queries, in: F. Özcan (Ed.), Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005, ACM, 2005, pp. 131–142.

[40] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, M. Krishnaprasad, SRQL: sorted relational query language, in: M. Rafanelli, M. Jarke (Eds.), 10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998, IEEE Computer Society, 1998, pp. 84–95.

[41] C. E. Dyreson, V. A. Rani, A. Shatnawi, Unifying sequenced and nonsequenced semantics, in: Proc. of TIME Symposium, IEEE Computer Society, Kassel, Germany, 2015, pp. 38–46.

[42] A. Dignös, B. Glavic, X. Niu, M. H. Böhlen, J. Gamper, Snapshot Semantics for Temporal Multiset Relations, Proceedings of the VLDB Endowment 12 (6) (2019) 639–652.

[43] Y. Bai, H. Thakkar, H. Wang, C. Luo, C. Zaniolo, A data stream language and system designed for power and extensibility, in: Proceedings of the 15th ACM International Conference on Information and Knowledge Management, Proc. of ACM CIKM '06, ACM, 2006, pp. 337–346.

[44] D. Abadi et. al., Aurora: a new model and architecture for data stream management, VLDB J. 12 (2) (2003) 120–139.

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

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

[47] C. Cranor, T. Johnson, O. Spataschek, V. Shkapenyuk, Gigascope: A stream database for network applications, in: Proc. of ACM SIGMOD, ACM, 2003, pp. 647–651.

[48] E. Panigati, F. A. Schreiber, C. Zaniolo, Data streams and data stream management systems and languages, in: F. Colace, M. De Santo, V. Moscato, A. Picariello, F. A. Schreiber, L. Tanca (Eds.), Data Management in Pervasive Systems, Springer, 2015, pp. 93–111.

[49] D. Toman, On construction of holistic synopses under the duplicate semantics of streaming queries, Vol. 174, 2007, pp. 150–162.

[50] T. Ghanem, A. Elmagarmid, P. Larson, W. Aref, Supporting Views in Data Stream Management Systems, ACM Trans. Database Syst. 35 (1) (2010) 1.

[51] M. J. Sax, G. Wang, M. Weidlich, J.-C. Freytag, Streams and tables: Two sides of the same coin, in: Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE '18, 2018, pp. 1:1–1:10.

[52] N. Laptev, B. Mozafari, H. Mousavi, H. Thakkar, H. Wang, K. Zeng, C. Zaniolo, Extending relational query languages for data streams, in: M. Garofalakis, J. Gehrke, R. Rastogi (Eds.), Data Stream Management: Processing High-Speed Data Streams, Springer, 2016, pp. 361–386.

[53] J. Krämer, B. Seeger, Semantics and Implementation of Continuous Sliding Window Queries over Data Streams, ACM Trans. Database Syst. 34 (1) (2009) 4.

[54] A. Ayad, J. Naughton, Static Optimization of Conjunctive Queries with Sliding Windows over Infinite Streams, in: Proc of ACM SIGMOD, 2004, pp. 419–430.

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

[56] N. Marz, J. Warren, Big Data. Principles and best practices of scalable realtime data systems, Manning Publications, 2015.

[57] Apache Storm, `http://storm.apache.org`.

[58] Apache Hadoop, `http://hadoop.apache.org`.

[59] Amazon Dynamo, `http://aws.amazon.com/dynamodb`.

[60] Kappa architecture, `http://www.oreilly.com/ideas/questioning-the-lambda-architecture`.

[61] Apache HBase, `http://hbase.apache.org`.

[62] Apache Flink, `http://flink.apache.org`.

[63] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: Fault-tolerant streaming computation at scale, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 423–438.

[64] Apache Calcite, `http://calcite.apache.org`.

[65] Y. P. Milinda Pathirage, Julian Hyde, B. Plale, SamzaSQL: Scalable fast data management with streaming SQL, in: Proc. of IEEE IPDP Symposium Workshops, IEEE Computer Society, 2016, pp. 1627–1636.

[66] Apache Phoenix, `http://phoenix.apache.org`.

[67] L. Antova, R. Baldwin, D. Bryant, T. Cao, M. Duller, J. Eshleman, Z. Gu, E. Shen, M. A. Soliman, F. M. Waas, Datometry hyper-q: Bridging the gap between real-time and historical analytics, in: Proc. of the 2016 Int. Conf. on Management of Data, 2016, pp. 1405–1416.

[68] D. Petkovic, Temporal data in relational database systems: A comparison, in: Á. Rocha, A. M. Correia, H. Adeli, L. P. Reis, M. Mendonça Teixeira (Eds.), New Advances in Information Systems and Technologies, Springer, 2016, pp. 13–23.

[69] J. Sequeda, O. Corcho, Linked stream data: A position paper, in: Proc. of the 2nd Int. Conf. on Semantic Sensor Networks, Vol. 522 of SSN'09, CEUR-WS.org, 2009, pp. 148–157.

[70] F. Mandreoli, R. Martoglia, W. Penzo, S. Sassatelli, Data-Sharing P2P Networks with Semantic Approximation Capabilities, IEEE Internet Computing 13 (5) (2009) 60–70.

[71] F. Mandreoli, R. Martoglia, W. Penzo, Approximating expressive queries on graph-modeled data: The GeX approach, Journal of Systems and Software 109 (Supplement C) (2015) 106 – 123.

[72] F. Grandi, Multi-temporal RDF ontology versioning, in: Proc. of Intl' Workshop on Ontology Dynamics (IWOD, in conj. with ISWC), CEUR-WS, Chantilly, VA, 2009.

[73] F. Grandi, T-SPARQL: a TSQL2-like temporal query language for RDF, in: Proc. of Intl' Workshop on on Querying Graph Structured Data (GrapQ10, in conj. with ADBIS), CEUR-WS, Novi Sad, Serbia, 2010, pp. 21–30.

[74] L. Carafoli, F. Mandreoli, R. Martoglia, W. Penzo, A Framework for ITS Data Management in a Smart City Scenario, in: Proc. of the 2nd Int. Conf. on Smart Grids and Green IT Systems SMARTGREENS, 2013, pp. 215–221.

[75] L. Carafoli, F. Mandreoli, R. Martoglia, W. Penzo, A Data Management Middleware for ITS Services in Smart Cities, Journal of Universal Computer Science 22 (2) (2016) 228 – 246.

[76] F. Zhang, K. Wang, Z. Li, J. Cheng, Temporal data representation and querying based on rdf, IEEE Access 7 (2019) 85000–85023.

[77] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, M. Grossniklaus, Querying rdf streams with c-sparql, ACM SIGMOD Record 39 (1) (2010) 20–26.

[78] J.-P. Calbimonte, O. Corcho, A. J. Gray, Enabling ontology-based access to streaming data sources, in: International Semantic Web Conference, Springer, 2010, pp. 96–111.

[79] D. Dell'Aglio, E. D. Valle, F. van Harmelen, A. Bernstein, Stream reasoning: A survey and outlook, Data Science 1 (1-2) (2017) 59–83.

[80] P. Terenziani, R. T. Snodgrass, Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction, IEEE Trans. Knowl. Data Eng. 16 (5) (2004) 540–551.

[81] P. Terenziani, The impact of the telic\atelic dichotomy on temporal databases, in: Proc. of TIME Symposium, IEEE Computer Society, Kongens Lyngby, Denmark, 2016, p. 3.

Appendix A. Expressing the Snapshot-equality Query in the Extended Temporal Algebra $\mathcal{TA}^\star$

The "snapshot-equality" query tests for the existence of distinct database snapshots with an equal content. Assuming the database schema composed

of a single unary relation $S(X|T)$ using point timestamping, it can easily be expressed in TRC as follows:

$$(\exists t)(\exists t')[t \neq t' \wedge (\forall x)(S(x,t) \equiv S(x,t'))]$$

Such a query cannot be expressed in FOTL or in a standard temporal algebra acting on relations with a single implicit timestamp [33, 34]. On the contrary, we will show in the following how it can be expressed in our extended temporal algebra $\mathcal{TA}^\star$ also employing non-temporal operators. This will prove that the following Lemma holds:

Lemma 2. *The "snapshot-equality" query can be expressed in the extended temporal algebra $\mathcal{TA}^\star$.*

Proof. The expression $\pi_{X,U_1}(\varepsilon_{U_1}(S))$ represents a non-temporal relation obtained from S by transforming the implicit timestamping attribute T into an explicit attribute U_1. Such a relation is non-temporal as the timestamp column has been projected out. Hence, if we use such a relation as argument of a $\mathcal{TA}$ operator, it is virtually converted to a temporal relation as shown in Sec. 2 (i.e., obtained as the Cartesian product between the tuples of the non-temporal relation and the whole time domain $\mathcal{T}$). Then we can consider the temporal division operator $\div^T$, defined as the temporal extension with sequenced semantics of its non-temporal counterpart $\div$ (the semantics of $\div$ is the standard one considered in relational algebra; for example, the division between $R_1(X_1X_2)$ and $R_2(X_2)$ can be defined as $R_1 \div R_2 = \pi_{X_1}(R_1) - \pi_{X_1}[(\pi_{X_1}(R_1) \times R_2) - R_1])$. In our case, the temporal division is used to express the required universal quantification on values of the explicit attribute X. In fact, we can use the expression:

$$P = \pi_{X,U_1}(\varepsilon_{U_1}(S)) \div^T S$$

whose result is a temporal relation with schema $P(U_1, T)$, to find the pairs (t_i, t_j) of timepoints, such that the snapshot of S valid at t_i contains all the X values present in the snapshot of S valid at t_j.

Notice that, owing to the sequenced semantics, the tuples valid at time t in P are the ones that can be computed by executing the non-temporal division $\div$ between $\pi_{X,U_1}(\varepsilon_{U_1}(S))$ and the explicit contents of the timeslice at time t of S (i.e., $\pi_X(\tau_t(S))$). The results of such a non-temporal division are the values t' of U_1 for which all the values x of X in $\tau_t(S)$ are present as explicit tuples (x, t') in S (actually in the snapshot valid at t of the temporal conversion of $\pi_{X,U_1}(\varepsilon_{U_1}(S))$; notice that, owing to the conversion, the explicit contents of all the snapshots are equal to the non-temporal relation $\pi_{X,U_1}(\varepsilon_{U_1}(S))$).

The timestamp T of P can also be made explicit for further manipulations as follows:

$$Q = \sigma^T_{U_1 \neq U_2}\left(\varepsilon_{U_2}[\pi_{X,U_1}(\varepsilon_{U_1}(S)) \div^T S]\right)$$

The outer selection eliminates tuples with $U_1 = U_2$, representing trivial snapshot self-containments. An example of the functioning of the temporal division (with further manipulations leading to Q) can be found in Fig. A.5.

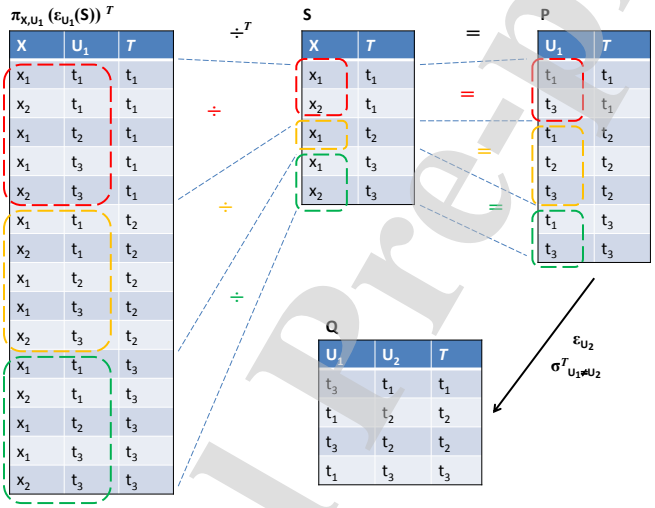


Figure A.5: Example of computation of $\pi_{X,U_1}(\varepsilon_{U_1}(S)) \div^T S$ and further manipulations leading to $Q(U_1, U_2|T)$. The table $\pi_{X,U_1}(\varepsilon_{U_1}(S))^T$ to the left represents the tuples of interest in the result of the temporal conversion of $\pi_{X,U_1}(\varepsilon_{U_1}(S))$ (all the other tuples produced by the conversion, with timestamps different form t_1, t_2 and t_3, do not contribute to the result of the temporal division owing to the sequenced semantics and, thus, have been omitted in the figure). The red, yellow and green colors highlight the $\div$ computations made on the individual snapshots valid at t_1, t_2 and t_3, respectively, to assemble the final $\div^T$ result in accordance to the sequenced semantics.

In order to find equal snapshots, we must finally test for reciprocal containment of the X values present in the snapshots valid at t_i and t_j, which means to test for the presence of both (U_1, U_2) pairs (t_i, t_j) and (t_j, t_i) in Q. This can be done in more than one way with the operators of the temporal algebra, for instance using attribute rename and intersection or (semi)join. Using rename and intersection (which in $\mathcal{TA}$ can be defined, for example, as $R_1 \cap^T R_2 = R_1 -^T (R_1 -^T R_2)$), the following expression:

$$(\pi_{U_1,U_2}(Q))^T \cap^T \rho_{U_1,U_2 \leftarrow U_2,U_1}(Q)$$

where the operator $\rho_{U_1,U_2 \leftarrow U_2,U_1}$ is used to rename the attributes U_1 to U_2 and U_2 to U_1 in Q, respectively, returns a non-empty result if and only if the TRC snapshot-equality query returns a true value. In fact, the (U_1, U_2) pairs in the tuples of the result are the timestamps of the equal snapshots. Notice that $\pi_{U_1,U_2}(Q)$ is a non-temporal relation, which is converted to temporal (with tuples valid always) since it appears as argument of a temporal operator in the expression above. □

Finally, notice that considering point timestamping in S is not a limitation, as an expression similar to the proposed solution can also be easily found for answering the snapshot-equality query on a relation using interval timestamping.

Highlights:

- Unification of streaming, temporal, and relational querying for advanced analytics
- Full acknowledgement of the inherently temporal nature of streaming data
- Definition of the Continuous Temporal Algebra
- Unified semantics for one-time and continuous queries
- Implementability on standard DBMS technology to leverage on well-founded procedures

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: