The VLDB Journal (2025) 34:11
https://doi.org/10.1007/s00778-024-00889-2

REGULAR PAPER

Reconciling tuple and attribute timestamping for temporal data warehouses

Waqas Ahmed[1] · **Leticia Gómez**[2] · **Alejandro Vaisman**[2] · **Esteban Zimányi**[1]

Received: 22 February 2024 / Accepted: 12 December 2024 / Published online: 21 December 2024

Abstract
Data Warehouses (DWs) requir e storing and analyzing time-varying data to reflect changes that occur in the business world. Solutions to this problem build on the field of temporal databases and adopt the *tuple-timestamping* approach, where tuples are timestamped with their validity interval. Alternatively, the *attribute timestamping* approach represents a time-varying attribute with a list of its evolving values and the time when these changes occurred. The SQL:2011 standard has favored the tuple timestamping approach, which has also been used for temporal DWs, despite that it yields very long and complex SQL queries. This paper aims at reconciling both approaches and advocates for a database that can support both models, in a way such that they complement each other. We show that, to efficiently operate with tuple timestamping, we need appropriate time data types and operations for representing and manipulating temporal elements. We also show that many applications are more naturally and efficiently modeled and implemented using attribute timestamping. To prove the feasibility of our proposal, we implemented a portion of the TPC-DS benchmark using three alternative approaches, two of them based on classic tuple timestamping (including the well-known slowly-changing dimensions model), and a third one, based on our proposal. For the latter, we used MobilityDB, a novel spatiotemporal database built on top of PostgreSQL, that integrates both models in a natural way. Experiments showed that our proposal outperformed the other two ones, in many cases, by orders of magnitude.

Keywords Temporal data warehouses · Temporal databases · Slowly changing dimensions · Attribute timestamping · Tuple timestamping

1 Introduction and motivation

Business Intelligence (BI) systems aim at transforming large volumes of business data into knowledge that can be used for decision making. Typically, BI systems are supported by a Data Warehouse (DW), which integrates data coming from different sources [37]. A DW is typically represented using a multidimensional model (MD), where data are perceived as a collection of so-called *facts* (quantified by *measures*) and *dimensions*, along which, facts are analyzed by means of aggregation across *hierarchies*. Although it is usual to consider that facts are the only dynamic part of a DW, dimensions may change over time in many different ways [18]. Thus, a MD model must be able to handle time-varying data while also allowing non-temporal objects [17]. It must also include a collection of OLAP (Online Analytical Processing) operations to analyze both non-temporal and temporal data. Finally, all of the above should be implementable using existing technologies, mainly relational databases.

In DW practice, changes in the instances of the dimensions are usually handled using the concept of *slowly changing dimensions* (SCDs) [21], which extend a dimension table with two columns usually denoted FromTime and ToTime, representing, respectively, the start and end of the interval of validity of each tuple in the table. Since SCDs are an ad-hoc solution to the problem and do not consider most of the research in the domain of *temporal databases* [31], Ahmed et al. [1] proposed a temporal MD data model that allows

✉ Alejandro Vaisman
avaisman@itba.edu.ar

Waqas Ahmed
waqas.ahmed@ulb.be

Leticia Gómez
lgomez@itba.edu.ar

Esteban Zimányi
esteban.zimanyi@ulb.be

[1] Université libre de Bruxelles, Brussels, Belgium

[2] Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina

storing non-temporal and temporal data and showed that it can be implemented in standard SQL.

Temporal relational databases usually follow the *tuple-timestamping* model [10], where tuples are timestamped with their validity interval, so that, whenever the value of some attribute changes, a new tuple is created in a table, allowing keeping track of the history of the database. This is basically the SCD approach. On the other hand, in the *attribute timestamping* approach [12], when an attribute value changes, the new value is inserted together with its timestamp, thus producing a list of (value,timestamp) pairs that represent the evolution of the value of the attribute across time. In spite of the many years of research in the field of temporal databases, tuple timestamping proved to be very difficult to put in practice so far [11]. However, adopting this solution was fueled by the fact that most databases in the market followed the first normal form (1NF) data model, and therefore, tuple timestamping was basically the only alternative for implementation. However, in a data warehousing scenario, tuple timestamping yields very complex and long SQL OLAP queries [1], as we show in Sect. 5 of this paper. On the other hand, attribute timestamping not only allows saving storage space in the database, but also promises a better performance for a wide range of queries. For example, in a stock-exchange database, the history of the value of a stock in the attribute-oriented approach is encoded in a single tuple. Thus, a table with the values of a stock in the NASDAQ-100 will just have one-hundred tuples, each one containing the historical time-series values. On the other hand, a table using tuple timestamping enconding will contain, for each stock, as many tuples as values of the stock have been registered. Clearly, the computation of the maximum value for a stock across time, would be much simpler using the first encoding. In light of the above discussion, the question that arises is: when would we prefer one approach over the other? We argue that the choice between attribute and tuple timestamping depends on the type of operation we face. We address this problem in this paper and advocate for a database that can support both models, in a way that they can complement each other.

In this paper, we show that, to efficiently operate with tuple timestamping, we need appropriate time data types for representing timestamps, sets of timestamps, periods, and sets of periods (called temporal elements). Further, we also need operations on these data types to implement a closed interval algebra allowing the representation of the results of union, intersection, and difference of time intervals. For example, suppose an employee e_1 works in a project p_1 in the intervals [1, 4] and [7, 18]. There is also employee e_2 who worked in the same project in the intervals [3, 4], [8, 9] and [12, 14]. Now, we want to ask for the periods when they worked together in the same project. With tuple timestamping, the input to the query would need five tuples, one per employee and per period. These tuples will be combined to obtain the result. Opposite to this, with attribute timestamping, the input will be composed of two tuples and the result will be comprised in a single tuple.

The data types and operations mentioned above are provided by MobilityDB [41], a novel database that builds on PostgreSQL[1] and its spatial extension PostGIS.[2] MobilityDB extends the type system of PostgreSQL and PostGIS with data types for representing spatiotemporal data. These data types are based on the notion of temporal types and their associated operations. Although MobilityDB was originally aimed at mobility analysis, its temporal types can represent the evolution over time of any kind of data, like integer, float, Boolean, and text. Thus, new data types like temporal integer (tint), temporal float (tfloat), and temporal Boolean (tbool) are defined, along with functions to manipulate them. This makes MobilityDB a temporal database supporting both, *tuple and attribute timestamping*. In this paper, we show that we can exploit this capability to express all temporal algebra operations [29, 40] in a concise, natural way, producing queries that outperform the traditional approach in many cases. This is relevant in heavy aggregation scenarios, like DW environments. Our work aims at showing that, since many applications (in particular the ones requiring temporal aggregation) can be naturally and efficiently modeled and implemented using attribute timestamping. Thus, supporting both approaches opens a wide range of design and implementation possibilities, closing the gap produced so far by the lack of alternatives to tuple timestamping. We illustrate this idea through an example, shown in Fig. 1, which we will also use in Sect. 2. For clarity, a tabular version of this figure is shown in Fig. 1.

Figure 1 shows a portion of three fictitious time series representing, for instance, the value of stocks in the NASDAQ-100. The upper part of the figure shows an attribute timestamped representation in a table Stocks(StockId, Price). Attribute Price is of type tfloat, therefore the table contains three records that represent the twenty-five values, and each record contains the time series of each stock's price. The lower part of the figure shows an equivalent tuple timestamped representation, that requires twenty-five timestamped records. Clearly, the representation in the upper part of the picture is more compact and also more appropriate to compute, for example, the maximum or average of stock prices. However, combining both approaches may be beneficial for some other operations, as we discuss next.

Suppose now that we want to partition the Stocks table, for example, to distribute records into three nodes or cores, for parallel computation. We want each partition to contain approximately the same number of values (e.g., partitions

[1] https://www.postgresql.org.

[2] https://postgis.net/.

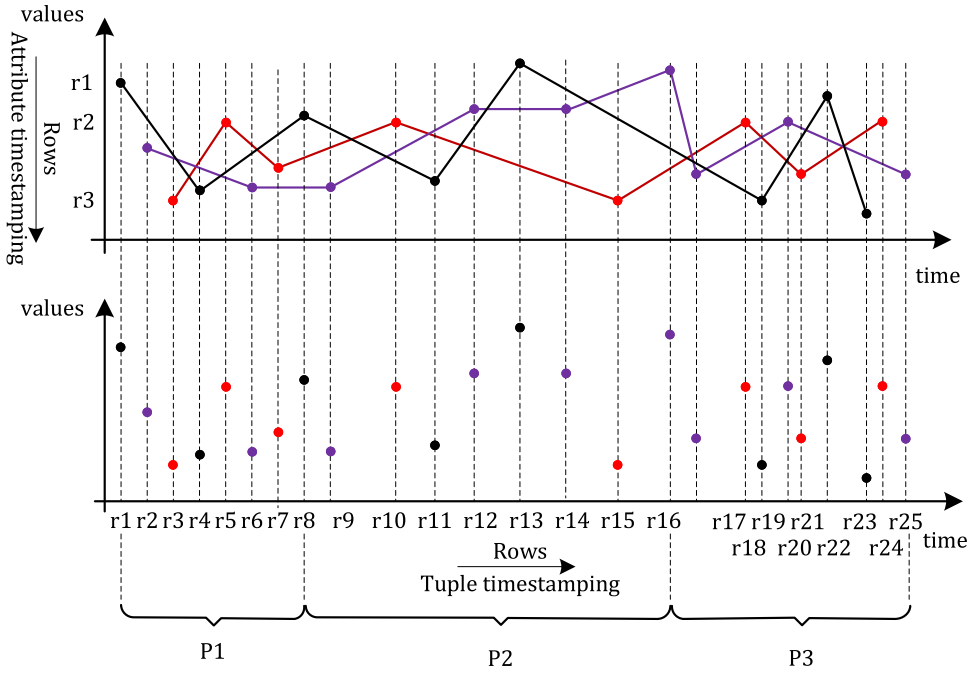


Fig. 1 Reconciling tuple and attribute timestamping

r1	v1	[t1, t4)
r2	v5	[t2, t7)
r3	v5	[t3, t5)
...	...	...

r1	v1@t1, v2@t4, v3@t8,...
r2	v5@t2, v6@t7, v4@t9,...
r3	v7@t3, v6@t5, v4@t7,...

Fig. 2 Tabular representation of Fig. 1. Left: Tuple timestamping; Right: Attribute timestamping

P1 and P2 will contain eight values and partition P3 nine). We thus need an adaptive partition, not one that considers equal time periods. This operation is called *adaptive time binning*. A key step in the binning computation finds the time limits of each partition. This computation over the attribute-timestamped Stocks table, would require a sweepline algorithm that scans all the time instants, which would be very inefficient. Instead, we can unnest the table into a tuple-timestamped representation and then compute the time limits of each partition. Finally, with the partitions already defined, we can go back to the original representation and use MobilityDB functions to efficiently find the parts of each series that fall into each partition. We leave the details to Sect. 2.

Paper Organization. Section 2 provides a theoretical framework for studying the problem, where we define a formal abstract model that supports attribute and tuple timestamping. In Sect. 3 we present the MobilityDB database as a concrete implementation of the abstract model. We also show the plausibility and benefits of the reconciliation approach. Section 4 discusses temporal DWs and presents the running example we use in the paper, which we implement in two classic temporal DW models and also using MobilityDB. In Sect. 5 we study how the classic temporal algebra operations, namely, temporal selection, projection, join, union, difference, and aggregation, are expressed over the three implementations. Since the implementation in SQL of these operations was studied in [29, 40], we considered them as a basis of our study. Section 6 presents the semantics of the temporal version of the most used classic OLAP operations namely *roll-up* and *dice*. Over the three models above, in Sect. 7 we define six representative queries, implementing the temporal OLAP operations previously mentioned. We perform experiments using the temporal algebra and OLAP queries and report the results in Sect. 8. Section 9 discusses related work in the context of the paper. We conclude in Sect. 10.

2 An abstract model for temporal databases

In this section we present an *abstract* model for temporal databases that supports both tuple and attribute timestamping. Following Güting [16], the term *abstract* refers to the fact that the definitions are made in terms of infinite sets. Therefore, we can get rid of the nuances of the implementation, which we address in the *concrete* model given in Sect. 3.

Temporal databases associate time with real-world objects and events in several ways [30]. *Valid time* (VT) refers to the time when a piece of information is considered valid in the real world. *Transaction time* (TT) represents the time when a piece of information is stored in the system. Transaction time represents the state of the database at different points in time and it is mainly used for roll-back or auditing purposes. Valid

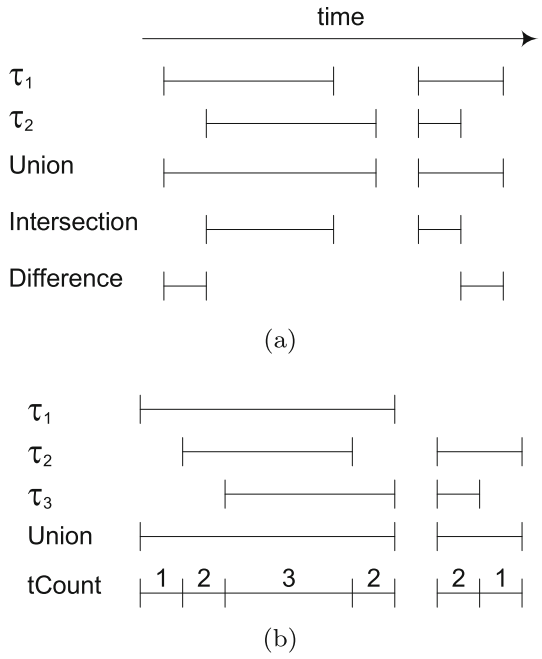


Fig. 3 **a** Union, intersection, and difference of two time values; **b** Union and temporal count aggregates

and transaction times are orthogonal, and can be used either individually or combined to define *bitemporal time* (BT). Further, *lifespan* (LS) represents the time when an object exists. The lifespan of an object can be seen as the valid time of a fact telling that the object exists. In the remainder, for the sake of simplicity and without loss of generality, we only consider valid time and lifespan.

2.1 Modeling time

Definition 1 **(Time domain)** The time domain $\mathcal{T}$ is a discrete and linearly ordered set of elements called *instants*. Instants are isomorphic (i.e., structurally similar) to the natural numbers. A special instant *now* in $\mathcal{T}$ denotes the current time instant. □

Definition 2 **(Time values)** A time value $\tau \subset \mathcal{T}$ is a finite subset of the time domain $\mathcal{T}$. □

Definition 3 **(Set operations on time values)** Let $\tau_1, \tau_2 \subset \mathcal{T}$ be time values. The union ($\cup$), intersection ($\cap$), and difference ($-$) of time values τ_1 and τ_2 are defined as the traditional set operators. For example, given a set of time values $\{\tau_1, \ldots, \tau_n\}$ their union is defined as $\bigcup_{i=1}^{n} \tau_i$. □

Figure 3a illustrates the operations on time values.

We define next a set of base data types, which have their usual interpretation except that their domain is extended with the value $\perp$ (undefined), which corresponds to the NULL value in standard SQL.

Definition 4 **(Base types)** We assume a set $\mathcal{B}$ of base types that includes the types bool, int, real, and string.[3] The domains of these base types are as follows.

$$\begin{aligned} \text{dom(bool)} &= \{\text{true}, \text{false}\} \cup \{\perp\} \\ \text{dom(int)} &= \mathbb{Z} \cup \{\perp\} \\ \text{dom(real)} &= \mathbb{R} \cup \{\perp\} \\ \text{dom(string)} &= \mathbb{S} \cup \{\perp\} \text{ where } \mathbb{S} \text{ are all finite strings} \\ &\qquad \text{over a finite alphabet } \mathcal{A}. \end{aligned}$$

□

2.2 Tuple timestamping

We now formally define tuple-timestamped relations and the relational algebra operations on them. In what follows, capital letters denote schema elements and lowercase letters refers to instances. Further, $\bar{A}$ represents schema tuples while $\bar{a}$ represents instance tuples.

Definition 5 **(Tuple-timestamped temporal relation)** Let $R(\bar{A} : \bar{B}) = R(A_1 : B_1, \ldots, A_n : B_n)$ be a relation scheme where values of attribute A_i are defined over $\text{dom}(B_i)$, $B_i \in \mathcal{B}$, for $i = 1, \ldots, n$. A *tuple-timestamped temporal relation* r on R is defined as:

$$r \subset \{\bar{a}/\tau \mid \bar{a} = (a_1, \ldots, a_n) \in \text{dom}(B_1) \times \ldots \times \text{dom}(B_n) \wedge \tau \text{ is a finite subset of } \mathcal{T}\}.$$

□

For a tuple $\bar{a}/\tau$, we denote the value component by $\nu(\bar{a}/\tau) = \bar{a}$ and the time component by $\tau(\bar{a}/\tau) = \tau$. Similarly, for a temporal relation r we denote $\nu(r) = \{\bar{a} \mid \bar{a}/\tau \in r\}$ and $\tau(r) = \{\tau \mid \bar{a}/\tau \in r\}$.

A temporal relation r may have a set of *value-equivalent* tuples $\{\bar{a}/\tau_1, \ldots, \bar{a}/\tau_n\}$ for $n > 1$, such that their value $\bar{a}$ coincide but their temporal part τ_i do not. In this case, a *coalescing* operation must be applied [8] to remove the redundancy by replacing the above set of tuples with a single one $\bar{a}/\tau$, where τ is the union of the time values τ_i. We define next the *COALESCE* operation, following the ideas in [39].

Definition 6 **(Coalesce operation)** Given a temporal relation r, we define $COALESCE(r) = r^c$, where

$$r^c = \{\bar{a}/\tau \mid \bar{a} \in \nu(r),\ \{\bar{a}/\tau_1, \ldots, \bar{a}/\tau_n\} \subseteq r, n \geq 1, \text{ are all the tuples in } r \text{ whose value is } \bar{a} \wedge \tau = \bigcup_{i=1}^{n} \tau_i\}.$$

□

[3] MobilityDB, the database we use later to implement the abstract model, also includes geometry and geography as base types but they are not considered in this paper.

In what follows, we assume that relations are always coalesced. We define now the algebraic operators over temporal relations, which generalize the classical ones. Redundancies in the resulting relation are removed by means of the *COALESCE* operation.

Definition 7 (Temporal selection) Let r be a temporal relation of schema $R(\bar{A})$ and φ be a Boolean selection formula on the attributes in $\bar{A}$ and/or the time component τ of the tuples in r. Then,

$$\sigma_{\varphi}(r) = COALESCE\,(\{\bar{a}/\tau \mid \bar{a}/\tau \in r \wedge \varphi(\bar{a}/\tau)\})$$

□

Definition 8 (Temporal projection) Let r be a temporal relation of schema $R(\bar{A}, \bar{B})$. Then,

$$\pi_{\bar{A}}(r) = COALESCE\,(\{\bar{a}/\tau \mid (\exists \bar{b})((\bar{a}, \bar{b})/\tau \in r)\})$$

□

Definition 9 (Temporal join) Let r_1 and r_2 be two temporal relations of schemas $R_1(\bar{A})$ and $R_2(\bar{B})$, and φ be a Boolean join formula involving the attributes in $\bar{A}$ and $\bar{B}$. Then,

$$r_1 \bowtie r_2 = COALESCE\,(\{(\bar{a}, \bar{b})/\tau \mid \bar{a}/\tau_1 \in r_1 \wedge \bar{b}/\tau_2 \in r_2 \wedge\ \tau = \tau_1 \cap \tau_2 \wedge \varphi\})$$

□

Definition 10 (Temporal union) Let r_1 and r_2 be two domain-compatible temporal relations. Then,

$$r_1 \cup r_2 = COALESCE\,(\{c/\tau \mid c/\tau \in r_1 \vee c/\tau \in r_2\})$$

□

Definition 11 (Temporal difference) Let r_1 and r_2 be two domain-compatible temporal relations. Then, $r_1 - r_2 = COALESCE\,(\{\bar{c}/\tau \mid \bar{c}/\tau_1 \in r_1 \wedge ([\bar{c}/\tau_2 \in r_2 \wedge \tau = \tau_1 - \tau_2] \vee [\bar{c} \notin \nu(r_2) \wedge \tau = \tau_1])\})$ □

We define next the notion of temporal aggregation, which extends traditional aggregation. In this paper, we consider the five typical SQL aggregation functions, namely, count, min, max, sum, and avg, and their generalization to the temporal domain. Other aggregation functions can be generalized similarly.

Consider for example, the non-temporal relation

Employee(SSN, FName, LName, Salary, DNo)

and the query "*Number of employees and maximum salary by department.*" This query can be expressed by the following aggregation

$$\mathcal{A}_{\mathsf{DNo},\mathsf{count}(\mathsf{SSN}),\mathsf{max}(\mathsf{Salary})}(\mathsf{Employee}).$$

Here, DNo is the grouping attribute, which partitions the relation Employee in as many relations as there are distinct DNo values. Then, the expressions count(SSN) and max(Salary) are computed in each partition.

The following definition formalizes the above. We assume that every base type $B \in \mathcal{B}$, has a set of aggregate functions $\mathcal{F}_B$ such as min$(\cdot)$, max$(\cdot)$, sum$(\cdot)$, etc., that operate over sets of values in dom(B).

Definition 12 (Aggregation) Let r be a non-temporal relation over a schema $R(\bar{A})$, and let $\bar{B}, \bar{C} \subset \bar{A}$, $\bar{B} \cap \bar{C} = \varnothing$. $\bar{B}$ is a set of *grouping attributes* that defines a partition $\mathcal{P}_{\bar{B}}(r) = \{P_1, \ldots, P_m\}$ of r, where

$$\pi_{\bar{B}}(r) = \{\bar{b}_1, \ldots, \bar{b}_m\}, \text{ and}$$
$$P_i = \pi_{\bar{B},\bar{C}}(\sigma_{\bar{B}=\bar{b}_i}(r)), \text{ for } i = 1, \ldots, m.$$

Also, let $\bar{f}(\bar{C}) = (f_1(C_1), \ldots, f_n(C_n))$, $f_i \in \mathcal{F}_{\mathrm{dom}(C_i)}$, be aggregate functions over the attributes C_i. An *aggregation* of r is an operation of the form $\mathcal{A}_{\bar{B},\bar{f}(\bar{C})}(r)$ defined as follows.

$$\mathcal{A}_{\bar{B},\bar{f}(\bar{C})}(r) = \textstyle\bigcup_{i=1}^{m} \mathcal{A}_{\bar{f}(\bar{C})}(P_i), \text{ and}$$
$$\mathcal{A}_{\bar{f}(\bar{C})}(P_i) = \{(\bar{b}_i, \bar{f}(\bar{c})) \mid \bar{c} \in \pi_{\bar{C}}(P_i)\}$$

□

We now generalize Def. 12 to the temporal case.

Definition 13 (Temporal aggregation) Let r be a *temporal* relation of schema $R(\bar{A})$, and let $\bar{B}$, $\bar{C}$, and $\bar{f}(\bar{C})$ be as in Def. 12, except that r is partitioned by the *value component* $\nu(r)$ of r, where each partition P_i has a time component $\tau(P_i) = \{\tau_1^i, \ldots, \tau_{m_i}^i\}$ that defines a timeline $\mathcal{T}(P_i) = \bigcup_{j=1}^{m_i} \tau_j^i$ for each partition. Then, a *temporal* aggregation $\mathcal{A}_{\bar{B},\bar{f}(\bar{C})}(r)$ is defined by

$$\mathcal{A}_{\bar{B},\bar{f}(\bar{C})}(r) = \textstyle\bigcup_{i=1}^{m} COALESCE\,(\mathcal{A}_{\bar{f}(\bar{C})}(P_i)), \text{ and}$$
$$\mathcal{A}_{\bar{f}(\bar{C})}(P_i) = \textstyle\bigcup_{j=1}^{m_i} \mathcal{A}_{\bar{f}(\bar{C})}(P_i, \tau_j) \text{ for all } \tau_j \in \mathcal{T}(P_i)$$
$$\mathcal{A}_{\bar{f}(\bar{C})}(P_i, \tau_j) = \{(\bar{b}_i, \bar{f}(\bar{c}))/\tau_j \mid \bar{c}/\tau \in \pi_{\bar{C}}(P_i) \wedge \tau_j \in \tau\}$$

□

As shown above, the temporal aggregation of each partition is obtained by applying the *non-temporal* aggregation to each instant in the timeline of the partition. This corresponds to the sequenced operations in [29].

Example 1 Consider a temporal version of the Employee table above, now with three tables:

Employee(SSN, FName, LName)
EmpSal(SSN, Salary, Time)
EmpDept(SSN, DNo, Time)

where Employee is a non-temporal table and both EmpSal and EmpDept are temporal tables whose temporality is kept in the Time column.

The query "*Time when a department has at least one employee*" can be expressed as follows:

$\mathcal{A}_{\mathsf{DNo,union(Time)}}(\mathsf{EmpDept})$.

The result of this aggregate operation for a single partition is illustrated in Fig. 3b, where the τ_i are the time components of the partition and thus each one represents the period during which an employee was affiliated to the given department. The union of the intervals is shown below τ_3. □

2.3 Attribute timestamping

We define next the attribute timestamping temporal model. The model builds on the definition of temporal attributes, which are attributes whose value evolves over time. The value of a temporal attribute is a partial function that maps instants of the time domain to values of the domain of its base type. A non-temporal attribute can be considered a particular case of a temporal one, whose values are represented as a constant function that maps all instants of the time domain to a single value in the domain of its base type. More formally, for each base type $B \in \mathcal{B}$ in Def. 4, the corresponding temporal type is defined using a temporal type constructor $\theta(\cdot)$. Examples are $\theta(\mathsf{int})$ and $\theta(\mathsf{real})$. We denote temporal types by adding the prefix 't' to their base type, such as tint or treal.

Definition 14 (Temporal domain of attributes) The temporal domain of an attribute A with base type B, denoted $\mathrm{tdom}(A)$, is defined as follows:

$$\{f \mid f : \mathcal{T} \to \mathrm{dom}(B) \text{ is a partial function}\}$$

□

Intuitively, the temporal domain of an attribute A defines all possible functions f that assign to each element in the time domain $\mathcal{T}$ a value in the domain of the base type B. The notion of temporal domain can be easily generalized for constant (i.e., non-temporal) attributes, as shown next.

Consider table Employee(SSN, Name, Salary), where Salary is the only temporal attribute. Suppose that an employee John has a salary of 30K during the interval $[t_1, t_2)$, 35K during the interval $[t_2, t_3)$, and undefined elsewhere. The value of this attribute for John can be defined as a function $f_1 : \mathcal{T} \to \mathbb{R}$ as follows

$$f_1 = \begin{cases} 30 & \text{for } t_1 \le t < t_2 \\ 35 & \text{for } t_2 \le t < t_3 \\ \bot & \text{otherwise} \end{cases}$$

On the other hand, the temporal domain of the Name attribute can be defined as the set of constant functions $\mathcal{T} \to \mathbb{S}$ that assign a single value $s \in \mathbb{S}$ for every element in $\mathcal{T}$. In the case of John, we can define his name as a constant function f_2 as follows

$f_2 =$ 'John' for all $t \in \mathcal{T}$

Therefore, without loss of generality, we define temporal relations in the attribute timestamping model considering only the temporal domains of the attributes.

Definition 15 (Attribute-timestamped temporal relation) Let $R(\bar{A} : \bar{B}) = R(A_1 : B_1, \ldots, A_n : B_n)$ be a relation scheme where attribute A_i is defined over the base type $B_i \in \mathcal{B}$, for $i = 1, \ldots, n$. Then, an *attribute-timestamped temporal relation* r on R is defined as follows:

$$r \subset \{\bar{a} \mid \bar{a} = (a_1, \ldots, a_n) \in \mathrm{tdom}(B_1) \times \ldots \times \mathrm{tdom}(B_n)\}.$$

□

Next, we define the *range* type constructor $\rho(\cdot)$ that allows representing set of disjoint intervals of time or base values. The range constructor is only applicable to types that have a total order $<$. Intervals have a lower and an upper bound, which can be either open or closed and have their usual meaning. We denote open bounds by '(' and ')', and closed bounds by '[' and ']'. Examples of intervals are $[l_1, u_1)$ or $(l_2, u_2]$.

Definition 16 (Range domain) Let $B \in \mathcal{B}$ be a base type to which the range type constructor ρ is applicable. The range domain is defined as follows.

$$\mathrm{rdom}(B) = \{\rho \mid \rho = \{\rho_1, \ldots, \rho_n\}, n \ge 1, \rho_i = \langle l_i, u_i \rangle, \\ l_i, u_i \in \mathrm{dom}(B), l_i \le u_i, \text{ and } '\langle' \text{ is either } '[' \text{ or } '(' \\ \text{and } '\rangle' \text{ is either } ']' \text{ or } ')'\}$$

□

We are now ready to define a collection of temporal operations over the abstract model. Following Güting [16], we define these operations by means of signatures over infinite sets. In Sect. 3 we will show that the MobilityDB database provides a *concrete* straightforward implementation of the operations defined in this way, where only finite representations are used.

Table 1 Classes of operations on temporal types

Class	Operations
Interaction with	GetTime, getValues, atTime, atValues
Domain/range	AtMin, atMax, valueAtTimestamp,..
Local aggregates	Integral, twAvg
Lifting	(All new operations inferred)
Conversion	Unnest

Table 2 Signature of operations on temporal types

Operation	Signature
getTime	$\theta(\alpha) \rightarrow \rho(\mathcal{T})$
getValues	$\theta(\alpha) \rightarrow \rho(\alpha)$
atTime	$\theta(\alpha) \times \rho(\mathcal{T}) \rightarrow \theta(\alpha)$
atValues	$\theta(\alpha) \times \rho(\alpha) \rightarrow \theta(\alpha)$
atMin	$\theta(\alpha) \rightarrow \theta(\alpha)$
atMax	$\theta(\alpha) \rightarrow \theta(\alpha)$
..	
+	real $\times \theta$(real) $\rightarrow \theta$(real)
+	θ(real) $\times$ real $\rightarrow \theta$(real)
+	θ(real) $\times \theta$(real) $\rightarrow \theta$(real)
..	
unnest	$\theta(\alpha) \rightarrow \alpha \times \rho(\mathcal{T})$

Table 1 shows a set of operations associated with temporal types grouped in several classes, while Table 2 depicts their signatures. Some of these operations are discussed next.

Operations getTime and getValues return, respectively, the projection of a temporal value into its domain and range, which result both in a range value. Operations atTime and atValues restrict the function to a given subset of the time or base domain defined by a range value. Operations atMin and atMax restrict the function to the points in time when its value is minimal or maximal, respectively. Operation valueAtTimestamp gets the base value of the function at a given timestamp.

Generalizing operations on base types for temporal types is called *lifting* [16]. As illustrated in Table 2, an operation for base types (e.g., +) is lifted by allowing any of the arguments to be a temporal type and return a temporal type. The semantics of lifted operations is that the result is computed at each time instant using the non-lifted operation. Lifted operations correspond to *sequenced operations* [29] in tuple timestamping.

Two interpretations may be considered when applying a lifted operation to two temporal values defined over different time extents. The first one considers that the result is defined in the intersection of both extents and it is undefined elsewhere. The second interpretation considers that the result is defined in the union of the two extents and a default value (e.g., 0 for +) is used for combining over the extents that belong to only one temporal value. We apply the first interpretation when combining *two temporal values* and apply the second one to *temporal aggregation*.

Finally, aggregate operations may also be lifted. Examples are tCount, tMin, tMax, and tAvg, which combine several temporal values, yielding a new temporal value where the traditional aggregate functions count, min, max, and avg are computed at each instant. We illustrate this with the tCount operation in Fig. 3b, explained in the next example.

Example 2 As a follow-up of Ex. 1, consider the query "*Evolution over time of the number of employees by department*." This query can be expressed by the following aggregation:

$$\mathcal{A}_{\mathsf{DNo,tCount(Time)}}(\mathsf{EmpDept}).$$

Fig. 3b shows the result of this aggregation represented as a temporal value (a tint). Since the timeline is discrete (recall Def. 1), each τ_i is actually a set of instants, thus, the traditional count is applied at each instant of the timeline of the result. □

2.4 Reconciling tuple and attribute timestamping

We explain next how the tuple- and attribute- timestamping approaches can be reconciled, so that users can choose the approach that best fits the application requirements, while being able to easily switch between them. Figure 1 shows a concrete example of two different ways of encoding the same information using either attribute- or tuple-timestamping. At a more conceptual level, Fig. 4 depicts a commutativity diagram illustrating the fact that we can encode temporal information using either tuple or attribute timestamping while leveraging efficient mechanisms to transform between the two representations. For example, we can start with a tuple-timestamped representation, transform it into an attribute-timestamped one, apply a temporal operation, and finally transform the result back into a tuple timestamped representation. In this way, users can choose the most appropriate representation on a case-by-case basis. We explain next how the tuple- and attribute- timestamping representations complement each other by showing how some operations are more efficiently computed in one of the representations.

Consider temporal aggregation which is a costly operation[6, 7]. Example 2 shows that, starting from a tuple-timestamping representation, we can represent the result of a temporal aggregation as a temporal value. The reason for this is that temporal aggregation is more efficiently computed using attribute-timestamping. To take advantage of the above, we need to be able to switch between representations when required. Figure 5 shows how we can use the unnest opera-

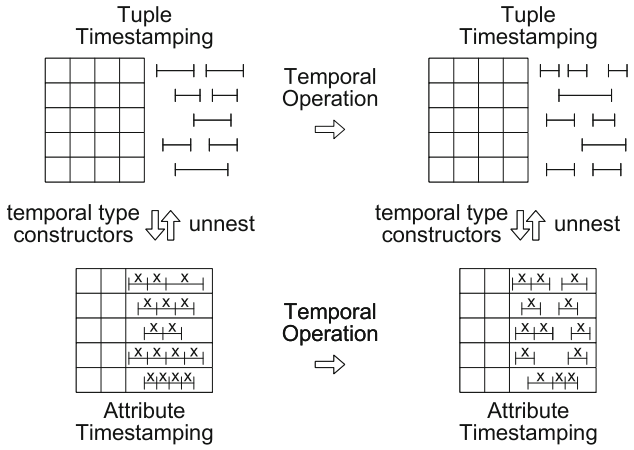


Fig. 4 Reconciling tuple and attribute timestamping

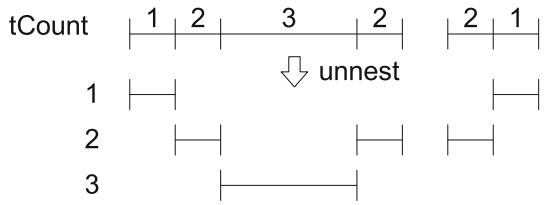


Fig. 5 Transforming from attribute to tuple timestamping the result of temporal count in Fig. 3b

tion to transform from an attribute- to a tuple-timestamping representation.

As another example, we show next how both representations can be used to solve the partition problem introduced in Sect. 1 and Fig. 1, where we have table Stocks(StockId, Price) with three time series of stock prices represented using attribute timestamping. To partition this table into three tables with approximately the same number of couples (value,timestamp), we use tuple timestamping to compute the period covered by each one of the three partitions. For this, we first create the table AdaptiveBins as follows.

```
CREATE TABLE AdaptiveBins(BinId, Period) AS
WITH Times(T) AS (
  SELECT unnest(timestamps(Price)) AS T
  FROM Stocks ),
MaxTime(MaxT) AS (
  SELECT MAX(T) FROM Times ),
Bins1(BinId, T) AS (
  SELECT NTILE(3) OVER(ORDER BY T), T
  FROM Times ),
Bins2(BinId, T, RowNo) AS (
  SELECT BinId, T, ROW_NUMBER()
    OVER (PARTITION BY BinId ORDER BY T)
  FROM Bins1 )
SELECT BinId, span(T, COALESCE(LEAD(T, 1)
  OVER (ORDER BY T), MaxT)) AS Period
FROM Bins2, MaxTime
WHERE RowNo = 1;
```

Table Times extracts all timestamps in the Stocks table. Table MaxTime obtains the maximum timestamp in the table. Table Bins1 applies the NTILE window function to divide the timestamps sorted in ascending order into three bins with approximately the same number of records. Table Bins2 adds the row number to the rows in each bin. Finally, the main query keeps the rows with row number equal to 1 to create with the span function the periods covered by each partition. For this, the LEAD window function adds to each row the timestamp of the next consecutive row and COALESCE assigns to the last row the maximum timestamp in the table. As a result, table AdaptiveBins contains three rows composed of the BinId and the associated Period. Note that most of the work is done over the timestamped representation produced by the unnest operation that creates the table Time. Now we can create the partitioned table.

```
CREATE TABLE StocksPart(BinId int, LIKE Stocks)
  PARTITION BY LIST(BinId);
CREATE TABLE StocksPart1 PARTITION OF StocksPart
  FOR VALUES IN (1);
CREATE TABLE StocksPart2 PARTITION OF StocksPart
  FOR VALUES IN (2);
CREATE TABLE StocksPart3 PARTITION OF StocksPart
  FOR VALUES IN (3);
```

Finally, we fill the partitions as follows.

```
INSERT INTO StocksPart
SELECT BinId, StockId, atTime(Price, Period) AS Price
FROM Stocks, AdaptiveBins
WHERE atTime(Price, Period) IS NOT NULL;
```

The query above uses the atTime function to restrict the temporal float attribute Price in table Stocks to the time period indicated in the second argument.

In summary, the tuple versus attribute timestamping debate, dating from the 1990 s, is still open. SQL has adopted the tuple-timestamping approach and provides data modification operations (insert, delete, update) taking into account the temporal semantics. However, SQL does not provide a temporal generalization of the relational algebra. As a consequence, to implement the relational operators the user needs to write complex SQL queries, as we discuss in Sect. 5.

3 A concrete model for temporal databases

This section presents a concrete model corresponding to the abstract model defined in Sect. 2, as implemented in MobilityDB.[4] A complete description of the model can be found in [41].

MobilityDB provides *collection types*, namely, set, span, and span set types, for representing finite subsets of the domains of base or time types. *Set types* represent a set of distinct values. Examples are intset or dateset, which are defined over the int and date types provided by PostgreSQL.

[4] https://mobilitydb.com/.

Span types represent ranges of base or time values and are defined by lower and upper bounds, that can be inclusive or exclusive. Examples are intspan and datespan. *Span set types* represent set of disjoint spans. Examples are intspanset and datespanset.[5]

3.1 Tuple timestamping

MobilityDB implements tuple timestamping by means of set operations on *time types*, which are defined at two *granularities*: date or timestamptz (timestamp with time zone). Four times types are used for defining finite subsets of the time domain at each granularity: date, dateset, datespan, and datespanset, as well as timestamptz, tstzset, tstzspan, and tstzspanset. We describe next these types using the date granularity.

A date value represents a time instant at a day granularity. A dateset value represents a set of distinct date values. It must contain at least one element and the elements must be ordered. An example is

```
SELECT dateset '{2021-01-01, 2021-01-03}';
```

A value of the datespan type has two bounds, the lower and the upper bounds, which are date values. The bounds can be inclusive (represented by '[' and ']'), or exclusive (represented by '(' and ')'). A datespan value with equal and inclusive bounds corresponds to a date value. An example of a datespan value is

```
SELECT datespan '[2021-01-01, 2021-01-03)';
```

A datespanset value represents a set of disjoint datespan values. It must contain at least one element and the elements must be ordered. An example is:

```
SELECT datespanset '{[2021-01-01, 2021-01-03),
  [2021-01-04, 2021-01-06)}';
```

MobilityDB provides an efficient implementation[6] of the time types and its operations (see Def. 3) based on a sweepline algorithm. As discussed in Sect. 2 (we will give more details in Sect. 5), the intersection and the difference of two time types are used for the temporal join and the temporal difference operation, respectively (Defs. 3, 9, and 11). We next give an example of a table representing the variation of prices of items, using the tuple timestamping model in MobilityDB.

```
CREATE TABLE Items_TS (itemId varchar(5),
  itemDesc varchar(30), unitPrice float, VT datespanset);
```

The attribute VT represents the valid time of tuples and is represented with values of type datespanset. An instance of this table is given next.

itemId	itemDesc	unitPrice	VT
'i1'	'Milk'	25.0	{[2021-01-01, 2021-07-01)}
'i1'	'Milk'	30.0	{[2022-01-01, 2022-04-01)}
'i2'	'Bread'	45.0	{[2021-04-01, 2022-01-01)}
'i2'	'Bread'	60.0	{[2022-01-01, 2022-11-01)}

3.2 Attribute timestamping

MobilityDB provides *temporal types* for representing values that evolve across time.[7] The temporal types tbool, tint, tfloat, and ttext are, respectively, based on the PostgreSQL types bool, int, float, and text.[8] Temporal types may be discrete or continuous depending on their base type. Discrete temporal types (such as tbool, tint, or ttext) evolve in a stepwise manner, while continuous temporal types (such as tfloat) evolve in either a linear or stepwise manner. The *duration* of a temporal value states the time extent at which the evolution of values is recorded. Temporal values come in three durations, namely, *instant*, *sequence*, and *sequence set*. A temporal *instant* value represents the value at a time instant, such as

```
SELECT tfloat '17.1@2022-01-01 08:00:00';
```

A temporal *sequence* value represents the evolution of the value during a sequence of time instants, where the values between these instants are interpolated using either a discrete, stepwise, or linear function. An example with *discrete* interpolation is

```
SELECT tint '{10@2022-01-01 08:00:00,
  20@2022-01-01 08:05:00, 15@2022-01-01 08:10:00}';
```

where the value is defined at the given timestamps and undefined everywhere else. An example of a temporal sequence value with *step* interpolation is

```
SELECT tint '(10@2022-01-01 08:00:00,
  20@2022-01-01 08:05:00, 15@2022-01-01 08:10:00]';
```

Finally, a temporal sequence value with *linear* interpolation is shown next:

```
SELECT tfloat '(10@2022-01-01 08:00:00,
  20@2022-01-01 08:05:00, 15@2022-01-01 08:10:00]';
```

The value of a temporal sequence is interpreted assuming that the time period defined by every pair of consecutive values v1@t1 and v2@t2 is lower inclusive and upper exclusive, unless they are the first or the last instants of the sequence and, in that case, the bounds of the whole sequence apply. Furthermore, the value of the temporal sequence between two consecutive instants depends on whether the data type is discrete or continuous. For example, the tint sequence

[5] The span and span set types in MobilityDB correspond to the range and multirange types in PostgreSQL, but they have a more efficient implementation.

[6] https://libmeos.org/documentation/datastructures.

[7] Currently, MobilityDB provides temporal types only at the timestamptz granularity.

[8] MobilityDB also provides the temporal types tgeompoint and tgeogpoint, which are based on the PostGIS types geometry and geography restricted to 2D and 3D points.

above represents that the value is 10 during (2022-01-01 08:00:00, 2022-01-01 08:05:00), 20 during [2022-01-01 08:05:00, 2022-01-01 08:10:00) and 15 at the end instant 2022-01-01 08:10:00. On the other hand, the tfloat sequence above tells that the value evolved linearly from 10 to 20 during (2022-01-01 08:00:00, 2022-01-01 08:05:00) and from 20 to 15 during [2022-01-01 08:05:00, 2022-01-01 08:10:00]. MobilityDB also allows representing sequences with stepwise interpolation when the type is continuous, for example:

```
SELECT tfloat 'Interp=Step;(10.1@2022-01-01 08:00:00,
  20.2@2022-01-01 08:05:00, 15.2@2022-01-01 08:10:00]';
```

Finally, a temporal *sequence set* value represents the evolution of the value at a set of sequences, where the values between them are unknown, for example:

```
SELECT tfloat
  '{[17.2@2022-01-01 08:00:00, 17.5@2022-01-01 08:05:00],
    [18.2@2022-01-01 08:10:00, 18.5@2022-01-01 08:15:00]}';
```

Temporal types have a rich set of operations corresponding to the ones defined for the abstract model in Sect. 2 (Tables 1 and 2). As discussed in [41], to ensure the closure of operations, when the operands of a lifted operation have a *linear* interpolation, the result of the operation must also be represented using linear interpolation.

The table containing the evolution of item prices, shown in Sect. 3.1 in the tuple-timestamped model, is modeled as follows in the attribute-timestamped approach, where the unit price of an item is represented as a temporal float (tfloat).

```
CREATE TABLE Items_AS (itemId varchar(5) PRIMARY KEY,
  itemDesc varchar(30), unitPrice tfloat);
```

The corresponding instance is shown next.

itemId	itemDesc	unitPrice
'i1'	'Milk'	{[25.0@2021-01-01, 25.0@2021-07-01), [30.0@2022-01-01, 30.0@2022-04-01)}
'i2'	'Bread'	{[45.0@2021-04-01, 45.0@2022-01-01), [60.0@2022-01-01, 60.0@2022-11-01)}

Over this table, the query

```
SELECT itemId,
  valueAtTimestamp(unitPrice, timestamptz '2021-04-15'),
  valueAtTimestamp(unitPrice, timestamptz '2021-07-15')
FROM Items_AS
```

returns the following values

```
'i1' | 25 | NULL
'i2' | 45 | 45
```

where the NULL value above represents the fact that the item's price for i1 is undefined on 2021-07-15. Consider now the following query and its result:

```
SELECT itemId, atTime(UnitPrice,
  tstzspan '[2021-04-01, 2021-11-01)')
FROM Items_AS
```

```
'i1' | {[25@2021-04-01, 25@2021-07-01)}
'i2' | {[45@2021-04-01, 45@2021-11-01)}
```

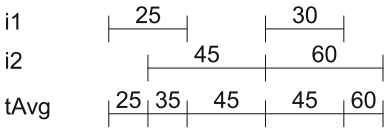


Fig. 6 Temporal aggregation in MobilityDB

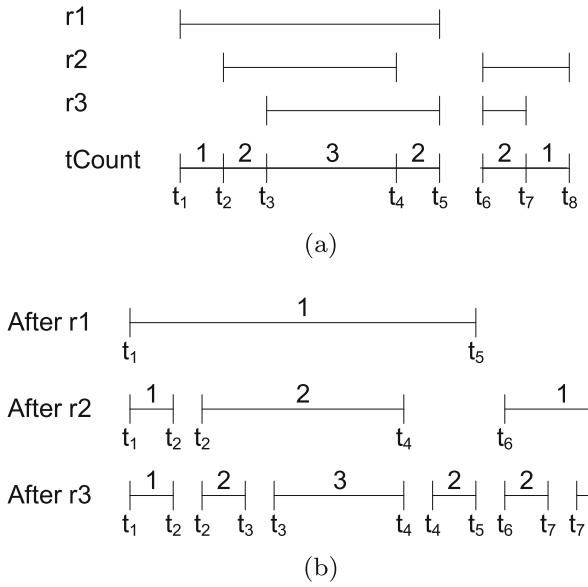


Fig. 7 **a** Temporal count of time values; **b** Content of the skiplist while processing the records iteratively

Here, the temporal attributes have been restricted to the period given in the query.

As an example of temporal aggregation, the next query asks for the average unit price across time.

```
SELECT tAvg(unitPrice)
FROM Items_AS
```

Below we show the result, and Fig. 6 shows such result graphically.

```
{[25@2021-01-01, 25@2021-04-01), [35@2021-04-01,...
  [..., 45@2022-04-01), [60@2022-04-01, 60@2022-11-01)}
```

To compute the temporal aggregate operations, a skiplist[9] is used. We explain this implementation using the example of Fig. 3b that shows how to compute the temporal count (tCount) at the abstract level. Figure 7a shows three records and their timespans, and, at the bottom, the count of the number of records at any time instant. Figure 7b shows, schematically, the iterative procedure to compute the aggregation using the skiplist. The list at the top contains the interval for record r1. When record r2 arrives, the list is split to accommodate the intervals for this record: between t_2 and t_4 there are two valid records (count = 2) and only one in the other intervals. When r3 arrives, we must make room for

[9] https://libmeos.org/documentation/aggregation/.

another element, since now there are three elements between t_3 and t_4. The interval $[t_6, t_8]$ is also split. The bottom of the figure shows the skiplist after processing r3.

3.3 Reconciliation in the concrete model

We conclude the section showing how we can use MobilityDB to apply the most appropriate representation for different operations, as discussed in Sect. 2.4. Initially, the user chooses the design that best fits the application purposes, in this case, tuple timestamping. The query below shows how we use both approaches to compute the tMax operation, that means, the evolution of the maximum price of the items across time.

```
WITH TempPrice(itemId, itemDesc, unitPrice) AS (
  SELECT itemId, itemDesc, merge(tfloat(unitPrice,
    VT::tstzspanset, 'step') ORDER BY VT)
  FROM Items_TS
  GROUP BY itemId, itemDesc ),
TempMaxPrice(unitPrice) AS (
  SELECT tMax(unitPrice)
  FROM TempPrice )
SELECT (rec).value, (rec).time::datespanset AS VT
FROM ( SELECT unnest(unitPrice) AS rec
         FROM TempMaxPrice ) ;
```

The result of this query is shown next:

```
25 | {[2021-01-01, 2021-04-01)}
45 | {[2021-04-01, 2022-01-01)}
60 | {[2022-01-01, 2022-11-01)}
```

The query first performs a transformation from the original tuple-timestamping representation to the attribute timestamping one, yielding the table TempPrice (which is, basically, table Items_AS above. Table TempMaxPrice contains the result of the tMax temporal operation. The main query transforms back this result into a tuple timestamping form. Note that these transformations between models and the application of the temporal operation corresponds to the operations in the commutativity diagram in Fig. 4. Also note that, as in relational databases, the user may decide to physically materialize frequently used views in both representations. However, this would be redundant in most cases, since MobilityDB is very efficient in performing these transformations.

4 Temporal data warehouses

In this section we first explain the basic notions of temporal data warehousing, and then propose the three implementations over which we develop our study.

4.1 Non-temporal data warehouses

DWs are typically represented using a multidimensional model, where data are perceived as an n-dimensional space composed of facts and dimensions. A *fact* is a subject of interest, e.g., a business event. Each observation in a fact is called a *fact member*. A fact is quantified by one or more *measures*, usually numerical quantities. *Dimensions* provide context to facts, e.g., sales events can be represented as a three-dimensional Sales fact with dimensions Product, Time, and City, and a measure Amount, where the dimensions represent information on when and where a product was sold at a certain price. Dimensions are organized in *levels*, which are described by attributes. For example, a level Day provides all possible values for dimension Time. Instances of a level are called *level members*. A DW may contain multiple levels and multiple facts may share these levels. The bottom level in a dimension determines the *granularity* of the latter, that is, the level of detail at which measures are recorded. An *aggregation relationship* relates a child and a parent levels and enables the aggregation of measures at various granularites, e.g., dimension Time may have an aggregation relationship between Day and Month. These relationships define *dimension hierarchies*. As usual, the *cardinality* of a relationship between dimension levels or between a fact and a level can be one-to-one (1-1), many-to-one (m-1), and many-to-many (m-m).

4.2 Temporal data warehouses

In real-world applications, DW objects may change in content and/or structure. *Content* changes (e.g., a modification in the price of a product) are due to routine business operations or to correction of existing data, while *schema* changes may occur because of changes in the modeled reality, for example, a modification of the cardinality from m-1 to m-m in the assignment of products to categories. A DW should be able to store such time-varying data. This requires extending the classic DW model, to capture time-varying *dimensional* data, for example, along the lines of temporal database theory [31]. We explain this next, remarking that in this paper we only consider content changes, not schema changes.

A *temporal attribute* keeps track of the changes in its value and the time when these changes occurred. Note that a level may be composed of both, temporal and non-temporal attributes.

A *temporal level* is a level for which the application requires storing the lifespan of its members, that is, the time during which they exist(ed). The lifespan of a temporal member is a subset of the time domain, while the lifespan of a non-temporal member covers all the time domain. The lifespan can be an interval (a set of consecutive instants) or it

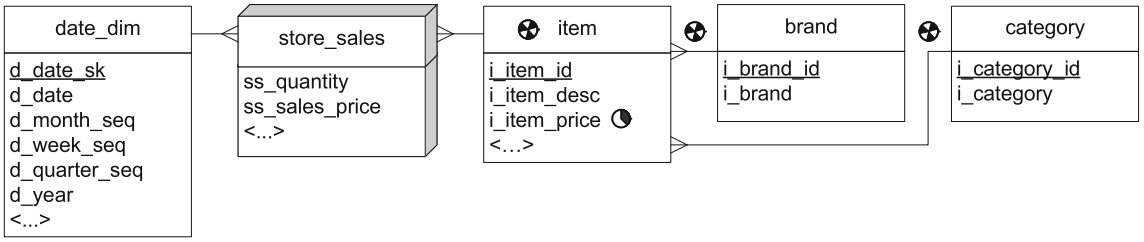


Fig. 8 Conceptual schema of the running example

may have time gaps between two intervals making it a set of intervals.

A *temporal aggregation relationship* keeps the evolution of the links between child and parent members in dimension hierarchies. It is a function that maps each instant in the time domain to a relation between the level members. For a non-temporal aggregation relationship, this function maps all the instants in the time domain to a fixed relation between level members. A *synchronization constraint* must guarantee that a parent and a child in a relationship instance coexist at each instant in the validity interval of the relationship and a child member must be assigned to some parent member throughout its lifespan.

Throughout the paper we will use, as a running example, a portion of the DW of the TPC-DS benchmark [26]. Figure 8 shows the schema of the example in the MultiDim conceptual model [25, 37] where the temporality of the elements is represented with pictograms (for brevity, not all pictograms are shown in the figure). The DW represents sales of items occurring at a certain date. The items' lifespan indicates the periods when items were available for sale. Further, the items' price, brand, and category change across time. We explain the model's notation next.

An *instant* (🕐) represents a single point in time, an *interval* (◔) denotes a set of consecutive instants between two time instants, an *instant set* (✇) represents a set of (distinct) time instants, and an *interval set* (☢) represents a set of disjoint intervals. The symbol ☢ next to the level item, indicates that it is a temporal level and thus the lifespan of its members is kept. The symbol ◔ next to attribute i_current_price indicates that this is a temporal attribute and the evolution of changes in its values is kept. The symbol ☢ next to the relationships relating item to brand and category tells that the evolution of the assignment of child members to their parent member is kept.

4.3 Temporal data warehouse implementations

This section compares alternative implementations of temporal data warehouses using as example the conceptual schema in Fig. 8. The usual approach favored by practitioners is referred to as *slowly changing dimension* (SCD), shown in Fig. 9. As an alternative, Ahmed et al. [1] proposed a temporal DW model (denoted TDW), depicted in Fig. 10, and associated OLAP operations. Finally, we also discuss the MobilityDB DW model (MobDB) based on time types shown in Fig. 11.

4.3.1 The slowly changing dimension model

In a DW, the values of some dimension attributes may change across time. Kimball and Ross [21] denoted the dimensions containing such attributes as slowly changing dimensions (SCD). They proposed three *basic* implementation techniques to track the changes in attribute values and denoted them "types."[10] Type 1 refers to changes that occur when an error is found in the data. In this case, the errors are fixed by overwriting the existing data, thus, no evolution is maintained. In Type 2 (the model most usually followed and the one we use in this paper), a new *version* of a level member is created for every change in *any* of its attribute values. A flag attribute or two time attributes representing a validity interval are used to represent the *current* version of level members. In this way, an unlimited number of versions of a level member can be created, but a surrogate key is required to uniquely identify each version. Finally, in Type 3, only the last two values of an attribute are kept, in two different columns. For example, in a Product dimension, we may have two attributes, namely Category and newCategory. When a product's category changes, the new value is stored in the newCategory column. If a subsequent change occurs, the values are shifted and the current value in the newCategory overwrites the current one in the Category column while the new value is stored in the newCategory column.

Figure 9 shows an SCD Type 2 implementation of the conceptual model of Fig. 8, based on the so-called *star schema*, where dimension tables are denormalized, to favor query performance. An alternative design, based on the *snowflake schema*, would normalize the dimension tables, leading to a higher number of joins. To keep track of the items' evolution, attributes i_rec_start_date and i_rec_end_date are added to the dimension table scd_item. These attributes rep-

[10] Kimball also proposed SCD Types 4 through 7, but we omit them since they are particular cases not related to this paper.

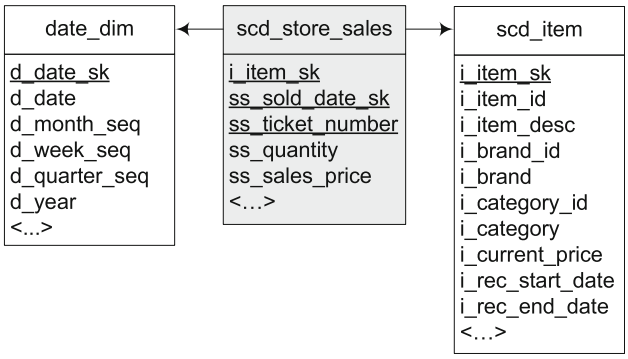


Fig. 9 Logical schema of the running example using the SCD Type 2 implementation

resent an interval when all other attribute values in a row are constant. More precisely, they represent an interval during which (1) an item exists; (2) an item's current price is valid; (3) an item–brand assignment exists; and (3) an item–category assignment exists. The value of the attribute i_rec_end_date of the current record is set to 9999-12-31. Although i_item_id is a business key of the level item, a surrogate key is required to identify the various versions of the same item. Thus, i_item_sk is added to the table and serves as its primary key.

4.3.2 The temporal data warehouse model

The Temporal DW model was proposed as an alternative to the SCD approach. Figure 10 shows the temporal DW implementation of the conceptual schema in Fig. 8. We briefly discuss the model next.

An item's lifespan is stored in table tdw_item_vt, which comprises the item's identifier and a pair of attributes FromDate and ToDate. Table tdw_item_price stores the items' price, and includes attributes FromDate and ToDate. Attributes i_item_id and FromDate compose the primary key for each table above. The evolution of the item–brand assignments is kept in table tdw_item_brand, which contains the item identifier, the brand identifier, and the period during which the assignment was valid. Finally, the assignment of items to categories is kept in table tdw_item_category, which depicts the interval during which an item was associated to a category. In this model, there is no need to create versions of records belonging to a table, and thus surrogate keys are not introduced. Attributes i_item_id, i_category_id, and FromDate compose the primary key of the table tdw_item_category. Table tdw_item_brand follows a similar rationale.

4.3.3 The MobilityDB data warehouse model

As an alternative to the previous classic temporal DW implementations, Fig. 11 shows a representation of the conceptual schema in Fig. 8, using the MobilityDB time data types explained in Sect. 3. We can see that the lifespan of items is represented in the table mobdb_item using an attribute of type datespanset. The same type is used in the three other tables that keep the evolution of items' features.

The relationship between items and brands helps us to explain the rationale of our proposed approach. Notice that table mobdb_item_brand is designed using tuple timestamping, where the periods of validity of each item–brand combination are represented by means of a datespanset type (i.e., a set of intervals with granularity date). As a first example, suppose that over this representation, we want to compute the temporal count (i.e., the evolution of the count over time) of the number of items of a brand. Figure 12a shows that this can be done by stacking over time the periods when any item belongs a brand and counting, at each point in time, how many periods overlap. The figure shows the result for brand B1 which, across time had between one and three items. The result is concisely represented in one single tuple using attribute timestamping. Opposite to this, the tuple timestamping representation of the result in this example would require at least three tuples (one for each item in the brand) or even more, if the database could not handle temporal elements. Figure 12b, shows another example, where we want to compute the evolution on time of the maximum number of items of any brand. This could be computed applying a temporal maximum operation (tMax), *over the result of the temporal count previously computed*. We remark that in Fig. 12a we have shown the computation of the temporal count for brand b1, while, for clarity, in Fig. 12b we show the temporal count of three brands, namely b1, b2 and b3. It can be intuitively seen that reusing the previous computation can be more easily done over the attribute timestamped representation of the result.

5 Temporal algebra operators in SQL

Querying and updating time-varying information using standard SQL is challenging. Snodgrass [28] proposed a temporal extension to SQL, called TSQL2. Some of its features are part of the SQL standard [22] and have also been incorporated into major DBMSs. Since database practitioners still use standard SQL for manipulating time-varying information, Snodgrass [29] has shown how most temporal relational operations can be written in standard SQL. Zimányi [40] then showed how to implement temporal aggregates and temporal universal quantifiers using standard SQL. From these works,

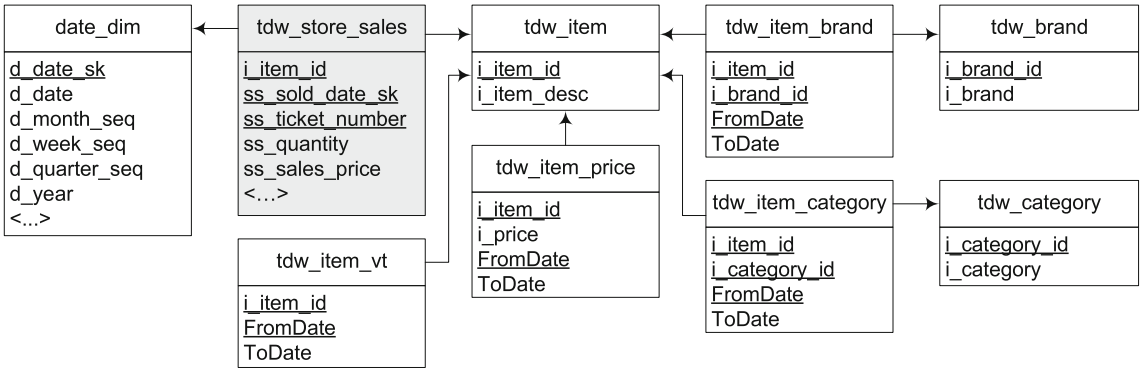


Fig. 10 Logical schema of the running example using using the traditional temporal DW model

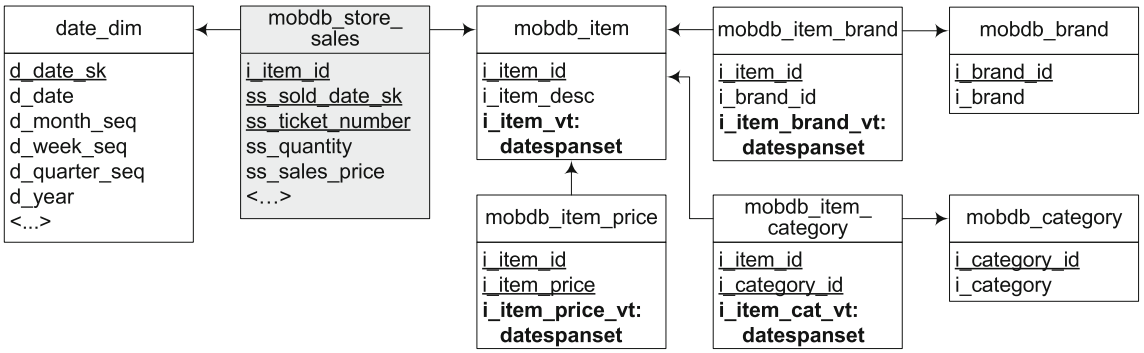


Fig. 11 Logical representation of the running example using MobilityDB time types

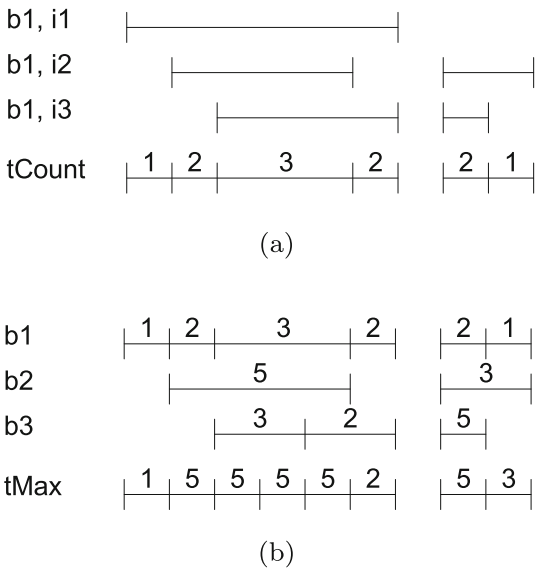


Fig. 12 **a** Evolution of the number of items for brand b1; **b** Maximum of the temporal count of items per brand across all brands

it follows that in order to foster massive use of temporal features in databases, a different approach is still needed.

In this section, we study how to express temporal relational algebra operations over the three alternative *tuple*-timestamping implementations of the TPC-DS conceptual schema depicted in Fig. 8. We remark that we start with the algebra operations because, as we will see Sect. 7, they are a core part of the OLAP queries over DWs, which typically aggregate over the results of basic algebra operations [37]. We compare the three approaches in terms of conciseness and simplicity, leaving performance to Sect. 8. For the sake of brevity, in this paper we only use the portion of the schema related to items. In Sect. 7 we apply these ideas to address OLAP queries in a DW environment.

The reader may be asking why do we design the whole schema using the *tuple*-timestamping approach instead of attribute timestamping, for example, representing the i_item_price attribute as a temporal float data type (tfloat). Both representations would be equivalent. The choice between the two depends on the application requirements. A typical query in our case would ask for sales with respect to the product price. Such a query would take each price of an item and compute the sales of the item at that price. Implementing such operation over an attribute timestamping schema would require to perform the unnest operation depicted in Fig. 5, to disassemble the evolution of the price for an item into pairs of the form (value, datespanset) and then to aggregate the sales for that value. This query would be more efficiently evaluated using tuple timestamping that would not require unnesting. On the other hand, Fig. 12a, b show that temporal

aggregations are computed more efficiently using attribute timestamping and unnesting the result into tuple timestamping, which is straightforward, as showed in Sect. 2.4.

In what follows, we carry out our study using the classes of temporal queries proposed in [29, 40], namely, temporal selection, projection, join, union, difference, and aggregation. These queries are shown in Table 3.

Temporal Selection As an example of this operation (Def. 7), we want to obtain the time when an item has a given brand.

Query 1 "*Time when an item has brand B.*"

In the TDW implementation, the SQL expression for the query would read as follows.

```
WITH ItemBrandBAll(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, FromDate, ToDate
  FROM tdw_item_brand
  WHERE i_brand_id = 5004001 )
/* Coalesce the table above */
SELECT DISTINCT f.i_item_id, f.FromDate, l.ToDate
FROM ItemBrandBAll f, ItemBrandBAll l
WHERE f.i_item_id = l.i_item_id AND
  f.FromDate < l.ToDate AND NOT EXISTS (
  SELECT *
  FROM ItemBrandBAll m1
  WHERE f.i_item_id = m1.i_item_id AND
    f.FromDate < m1.FromDate AND
    m1.FromDate <= l.ToDate AND NOT EXISTS (
    SELECT *
    FROM ItemBrandBAll m2
    WHERE f.i_item_id = m2.i_item_id AND
      m2.FromDate < m1.FromDate AND
      m1.FromDate <= m2.ToDate ) ) AND
  NOT EXISTS (
    SELECT *
    FROM ItemBrandBAll m
    WHERE f.i_item_id = m.i_item_id AND
      ( ( m.FromDate < f.FromDate AND
          f.FromDate <= m.ToDate ) OR
      ( m.FromDate <= l.ToDate AND
        l.ToDate < m.ToDate ) ) );
```

First, the query filters the rows in the WITH clause; then, the result is coalesced in the main query. The coalesce operation combines into one row a set of *value-equivalent* rows (i.e., rows that are equal on all their columns except for FromDate and ToDate), if their time periods overlap. This is a demanding operation that requires three nested NOT EXISTS predicates. We select the period [FromDate, ToDate) obtained from two rows f and l such that there is no gap during the period; that is, for every row m1 valid within the selected period there is a row m2 that can extend m1 to the left. This is implemented by the first two NOT EXISTS predicates in the query above. The third NOT EXISTS predicate ensures that no other row m can extend the selected period to the left or to the right.

Next, we show how the query is expressed over the SCD Type 2 implementation of Fig. 9. To avoid unnecessary repetition, in what follows we only show the portions of the queries that differ from those in the TDW implementation.

```
WITH ItemBrandBAll(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, i_rec_start_date, i_rec_end_date
  FROM scd_item
  WHERE i_brand_id = 5004001 )
/* ... Main query coalescing the table above omitted ... */
```

We can see that all temporal aspects are kept in a single table scd_item. Thus, the only difference with the TDW implementation, is the table from which ItemBrandBAll obtains its content. The situation is analogous for all queries in the SCD implementation.

Finally, we show how the query is expressed over the schema in Fig. 11 (the MobilityDB DW model) using the spanUnion operation over time types, which automatically performs the coalescing operation.

```
SELECT i_item_id, spanUnion(i_item_price_vt)
FROM mobdb_item_brand
WHERE i_brand_id = 5004001
GROUP BY i_item_id;
```

To obtain *exactly* the same result as the query above, the datespanset in the second column is decomposed into two columns FromDate and ToDate as follows:

```
WITH temp(i_item_id, i_item_price_vt) AS (
  SELECT i_item_id, i_item_brand_vt
    unnest(spans(spanUnion(i_item_price_vt)))
  FROM mobdb_item_brand
  WHERE i_brand_id = 5004001
  GROUP BY i_item_id )
SELECT i_item_id, lower(i_item_price_vt) AS FromDate,
  upper(i_item_price_vt) AS ToDate
FROM temp;
```

The expression unnest(spans(·)) transforms the argument datespanset into several rows, one for each composing period, and the functions lower and upper extract the corresponding bounds from each period.

Temporal Projection We now show how the temporal projection (Def. 8) is expressed in SQL. We omit the details, since the query is similar to the previous one: first, the table ItemAnyBrandAll is computed and then this table is coalesced.

Query 2 "*Time when an item was assigned to any brand.*"

The query over the TDW implementation reads:

```
WITH ItemAnyBrandAll(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, FromDate, ToDate
  FROM tdw_item_brand )
SELECT DISTINCT f.i_item_id, f.FromDate, l.ToDate
FROM ItemAnyBrandAll f, ItemAnyBrandAll l
WHERE f.i_item_id = l.i_item_id AND
  f.FromDate < l.ToDate AND NOT EXISTS (
  /* ... Coalescing statements omitted ... */ )
```

As mentioned, the SCD implementation is similar to the TDW one, as shown next.

```
WITH ItemAnyBrandAll(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, i_rec_start_date, i_rec_end_date
  FROM scd_item )
/* ... Main query coalescing the table above omitted ... */
```

Table 3 Temporal algebra queries and the corresponding relational operator

Query	Operator
Q1: Time when an item has brand B	Temporal selection
Q2: Time when an item was assigned to any brand	Temporal projection
Q3: Time when an item has brand B and its price is greater that €80	Temporal join
Q4: Time when an item has brand B or its price is greater than €80	Temporal union
Q5:Time when an item has brand B and its price is not greater than €80	Temporal difference
Q6: Time when a category has more than three items	Temporal aggregation

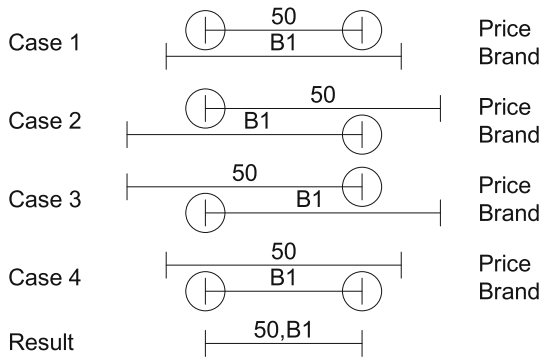


Fig. 13 The four cases for temporal join

In MobilityDB the query is written as follows.

```
SELECT i_item_id, spanUnion(i_item_brand_vt)
FROM mobdb_item_brand
GROUP BY i_item_id;
```

Temporal Join We now compute the time when an item had a given brand and a given price. This corresponds to a temporal join (Def. 9). Given one row from each table whose validity periods intersect, a temporal join will return the brand and price values together with the intersection of the two validity periods. As showed in [29], expressing a temporal join in SQL requires four SELECT statements and complex inequality predicates in order to verify that the validity periods of the rows to be combined intersect. This is illustrated in Fig. 13. Assuming that there are no duplicate rows in the tables (that is, at each point in time an item has one brand and one price), a UNION ALL operation can be used to combine the four cases. Finally, the result is, again, coalesced. The SQL:2023 standard provides two functions, greatest and least that return, respectively, the minimum and maximum of their two arguments. Thus, we can write the temporal join in a simpler way. We refer to [29, 37] for details.

Query 3 "*Time when an item has brand B and its price is greater that €80.*"

The SQL solution for this query, over the classic TDW, reads:

```
WITH BrandBAndPriceGT80All(i_item_id, FromDate,
    ToDate) AS (
  SELECT ib.i_item_id,
    greatest(ib.FromDate, ip.FromDate) AS FromDate,
    least(ib.ToDate, ip.ToDate) AS ToDate
  FROM tdw_item_brand ib, tdw_item_price ip
  WHERE ib.i_item_id = ip.i_item_id AND
    ib.i_brand_id = 5004001 AND ip.i_item_price > 80 AND
    greatest(ib.FromDate, ip.FromDate) <
    least(ib.ToDate, ip.ToDate) )
/* ... Main query coalescing the table above omitted ... */
```

The SQL query for the SCD implementation reads

```
WITH BrandBAndPriceGT80All(i_item_id, FromDate,
    ToDate) AS (
  SELECT i_item_id, i_rec_start_date, i_rec_end_date
  FROM scd_item
  WHERE i_brand_id = 5004001 AND i_current_price > 80 )
/* ... Main query coalescing the table above omitted ... */
```

In the TDW implementation, the evolution of brands and prices of items is kept in different tables and thus, a temporal join is needed to combine them, while this information is kept together in the SCD approach. Thus, the tables in the TDW model are much smaller than the ones in the SCD one, since the table scd_item keeps the temporal Cartesian product of all the corresponding tables in the TDW approach.

The same query is expressed in MobilityDB using the intersection (*) operation on two sets of intervals, represented using the datespanset type. After that, a coalescing using spanUnion is performed.

```
SELECT i_item_id, spanUnion(i_brandBAndPriceGT80_vt)
FROM (
  SELECT ib.i_item_id, ib.i_item_brand_vt *
    ip.i_item_price_vt AS i_brandBAndPriceGT80_vt
  FROM mobdb_item_brand ib, mobdb_item_price ip
  WHERE ib.i_item_id = ip.i_item_id AND
    ib.i_brand_id = 5004001 AND ip.i_item_price > 80 AND
    ib.i_item_brand_vt * ip.i_item_price_vt IS NOT NULL
    ) AS t
GROUP BY i_item_id
ORDER BY i_item_id;
```

Temporal Union Our next query asks for the times when an item has a given brand *or* has a given price. This is an example of a temporal union operation (Def. 10). To compute this operation in SQL, we first obtain the tuples that satisfy each condition and perform their union. Finally, we coalesce the result of the previous operation.

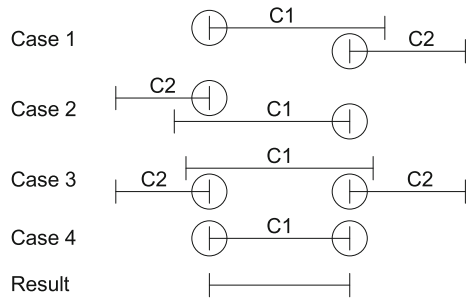


Fig. 14 The four cases for temporal difference

Query 4 "*Time when an item has brand B or its price is greater than €80.*"

The query over TDW is written as:

```
WITH BrandBOrPriceGT80All(i_item_id, FromDate,
    ToDate) AS (
  SELECT i_item_id, ib.FromDate, ib.ToDate
  FROM tdw_item_brand ib
  WHERE b.i_brand_id = 5004001
  UNION
  SELECT i_item_id, FromDate, ToDate
  FROM tdw_item_price
  WHERE i_item_price > 80 )
/* ... Main query coalescing the table above omitted ... */
```

Over SCD, the SQL would be expressed as follows, omitting, as before, repeated parts.

```
WITH BrandBOrPriceGT80All(i_item_id, FromDate,
    ToDate) AS (
  SELECT i_item_id, i_rec_start_date, i_rec_end_date
  FROM scd_item
  WHERE i_brand_id = 5004001 OR i_current_price > 80 )
/* ... Main query coalescing the table above omitted ... */
```

The corresponding MobilityDB query is given next.

```
SELECT i_item_id, spanUnion(i_brandBOrPriceGT80_vt)
FROM (
  SELECT i_item_id, i_item_brand_vt AS
    i_brandBOrPriceGT80_vt
  FROM mobdb_item_brand
  WHERE i_brand_id = 5004001
  UNION
  SELECT i_item_id, i_item_price_vt
  FROM mobdb_item_price
  WHERE i_item_price > 80 ) AS t
GROUP BY i_item_id
ORDER BY i_item_id;
```

Temporal Difference We now give an example of the temporal difference operation (Def. 11). To express this operation in SQL, we must consider the four possible cases shown in Fig. 14, where an item satisfies the first condition but not the second one.

Query 5 "*Time when an item has brand B and its price is not greater than €80.*"

Over the TDW implementation, we have:

```
/* Time when an item has brand B */
WITH ItemBrandBAll(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, FromDate, ToDate
  FROM tdw_item_brand
  WHERE i_brand_id = 5004001 ),
/* Coalesce the table above */
ItemBrandB(i_item_id, FromDate, ToDate) AS ( ... ),
/* Time when an item's price is greater than 80 */
ItemPriceGT80All(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, FromDate, ToDate
  FROM tdw_item_price
  WHERE i_item_price > 80 ),
/* Coalesce the table above */
ItemPriceGT80(i_item_id, FromDate, ToDate) AS ( ... ),
/* Temporal difference of ItemBrandB and ItemPriceGT80 */
ItemBrandBAndNotPriceGT80All(i_item_id, FromDate,
    ToDate) AS (
  /* Case 1 */
  SELECT b1.i_item_id, b1.FromDate, b2.FromDate
  FROM ItemBrandB b1, ItemPriceGT80 b2
  WHERE b1.i_item_id = b2.i_item_id AND
    b1.FromDate < b2.FromDate AND NOT EXISTS (
    SELECT *
    FROM ItemPriceGT80 b3
    WHERE b1.i_item_id = b3.i_item_id AND
      b1.FromDate < b3.FromDate AND
      b3.FromDate < b2.FromDate )
  UNION
  /* Case 2 */
  SELECT b1.i_item_id, b2.ToDate, b1.ToDate
  FROM ItemBrandB b1, ItemPriceGT80 b2
  WHERE b1.i_item_id = b2.i_item_id AND
    b2.ToDate < b1.ToDate AND NOT EXISTS (
    SELECT *
    FROM BrandTwoCat b3
    WHERE b1.i_brand_id = b3.i_brand_id AND
      b2.ToDate < b1.ToDate AND NOT EXISTS (
      SELECT *
      FROM ItemPriceGT80 b3
      WHERE b1.i_item_id = b3.i_item_id AND
        b2.ToDate < b3.ToDate AND
        b3.ToDate < b1.ToDate ) )
  UNION
  /* Case 3 */
  SELECT b1.i_item_id, b2.ToDate, b3.FromDate
  FROM ItemBrandB b1, ItemPriceGT80 b2,
    ItemPriceGT80 b3
  WHERE b2.ToDate < b3.FromDate AND
    b1.i_item_id = b2.i_item_id AND
    b1.i_item_id = b3.i_item_id AND NOT EXISTS (
    SELECT *
    FROM ItemPriceGT80 b4
    WHERE b1.i_item_id = b4.i_item_id AND
      b2.ToDate < b4.ToDate AND
      b4.ToDate < b3.FromDate )
  UNION
  /* Case 4 */
  SELECT i_item_id, FromDate, ToDate
  FROM ItemBrandB b1
  WHERE NOT EXISTS (
    SELECT *
    FROM ItemPriceGT80 b2
    WHERE b1.i_item_id = b2.i_item_id AND
      b1.FromDate < b2.ToDate AND
      b2.FromDate < b1.ToDate )
/* ... Main query coalescing the table above omitted ... */
```

The query performs a temporal difference between tables ItemBrandB and ItemPriceGT80, which are the result of coalescing ItemBrandBAll and ItemPriceGT80All. The difference is computed in table ItemBrandBAndNotPriceGT80All. The four inner queries implement (using the NOT EXISTS predicate) the four cases in Fig. 14. The result is coalesced in the main query.

Over the SCD implementation we have:

```
/* Time when an item has brand B */
WITH ItemBrandBAll(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, i_rec_start_date, i_rec_end_date
  FROM scd_item
  WHERE i_brand_id = 5004001 ),
/* Coalesce the table above */
ItemBrandB(i_item_id, FromDate, ToDate) AS ( ... ),
/* Time when an item's price is greater than 80 */
ItemPriceGT80All(i_item_id, FromDate, ToDate) AS (
  SELECT i_item_id, i_rec_start_date, i_rec_end_date,
  FROM scd_item
  WHERE i_current_price > 80 )
/* Coalesce the table above */
ItemPriceGT80(i_item_id, FromDate, ToDate) AS ( ... ),
/* Continues as the query for the TDW approach ... */
```

Again, this query and the one corresponding to the TDW implementation differ in how the tables ItemBrandBAll and ItemPriceGT80All are computed.

In the MobilityDB DW model the query is written:

```
SELECT i_item_id, spanUnion(i_brandBAndNotPriceGT80_vt)
FROM (
  SELECT ib.i_item_id, ib.i_item_brand_vt -
    ip.i_item_price_vt AS i_brandBAndNotPriceGT80_vt
  FROM mobdb_item_brand ib, mobdb_item_price ip
  WHERE ib.i_item_id = ip.i_item_id AND
    ib.i_brand_id = 5004001 AND ip.i_item_price > 80 AND
    ib.i_item_brand_vt - ip.i_item_price_vt IS NOT NULL )
    AS t
GROUP BY i_item_id
ORDER BY i_item_id;
```

The difference in this case is computed using the difference (-) operation on two interval sets in the SELECT clause of the inner query, which makes this query easier to express than in the TDW or SCD approaches.

Temporal Aggregation The temporal version of the five classic SQL aggregate operations, as stated in Def. 13, requires a three-step process: (1) split the timeline into periods of time during which all values are constant, (2) compute the aggregation over these periods, and (3) coalesce the result. This is illustrated next.

Query 6 *"Time when a category has more than three items."*

The aggregation over the TDW implementation is:

```
/* Days when the assignment of an item to a category changes */
WITH CategoryChanges(i_category_id, Day) AS (
  SELECT i_category_id, FromDate
  FROM tdw_item_category
  UNION
  SELECT i_category_id, ToDate
  FROM tdw_item_category ),
/* Per category, split the timeline using CategoryChanges */
CategoryPeriods(i_category_id, FromDate, ToDate) AS (
  SELECT * FROM (
    SELECT i_category_id, Day AS FromDate,
      LEAD(Day) OVER (PARTITION BY i_category_id
      ORDER BY Day) AS ToDate
    FROM CategoryChanges ) AS t
    WHERE ToDate IS NOT NULL ),
/* Categories with more than 3 items */
CategoryGT3ItemsAll(i_category_id, FromDate, ToDate) AS (
  SELECT ic.i_category_id, b.FromDate, b.ToDate
  FROM tdw_item_category ic, CategoryPeriods b
  WHERE ic.i_category_id = b.i_category_id AND
    ic.FromDate <= b.FromDate AND b.ToDate <= ic.ToDate
  GROUP BY ic.i_category_id, b.FromDate, b.ToDate
  HAVING COUNT(*) > 3 )
/* ... Main query coalescing the table above omitted ... */
```

Table CategoryChanges gathers, for each category, the dates at which a category assignment started or ended. Table CategoryPeriods constructs the periods from such dates. Table CategoryGT3ItemsAll filters out the previous table, selecting the categories that have more than three items assigned to it. Finally, this table is coalesced in the main query.

The SCD implementation reads:

```
WITH ItemCategoryAll(i_item_id, i_category_id, FromDate,
    ToDate) AS (
  SELECT i_item_id, i_category_id, i_rec_start_date,
    i_rec_end_date
  FROM scd_item ),
/* Coalesce the table above */
ItemCategory(i_item_id, i_category_id, FromDate,
  ToDate) AS ( ... ),
/* Days when the assignment of an item to a category changes */
CategoryChanges(i_category_id, Day) AS (
/* Continues as the query for the TDW approach ... */
```

We can see that, with respect to the query for the TDW implementation, the query above must compute the table ItemCategory before continuing with the aggregation that starts from the computation of the table CategoryChanges. We will see in Sect. 8 that this has a strong impact on the query performance.

Before showing the MobilityDB solution, we introduce the whenTrue function, which returns the time when a predicate is satisfied. For example, the query

```
SELECT whenTrue(tfloat '[1@2000-01-01, 4@2000-01-04,
  1@2000-01-07]' #> 3);
```

returns the time when the value of the temporal float is greater than three, shown next:

```
{(2000-01-03 00:00:00, 2000-01-05 00:00:00)}
```

To express our query, we first compute the temporal count (Fig. 12a) and then use the whenTrue function to obtain the periods when the temporal count is greater than three.

```
WITH ItemNoCats AS (
  SELECT i_category_id, tCount(i_item_category_vt)
```

```
  FROM mobdb_item_category
  GROUP BY i_category_id )
SELECT i_category_id, whenTrue(tCount #> 3)
FROM ItemNoCats
WHERE whenTrue(tCount #> 3) IS NOT NULL
ORDER BY i_category_id;
```

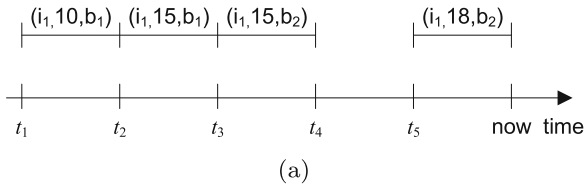
In conclusion, the queries discussed in this section illustrate the benefits of the temporal types approach, implemented in the MobilityDB DW. The temporal types allow expressing the queries very concisely. In contrast, in classic temporal databases, most queries require writing long pieces of SQL code, in particular due to the coalesce operation and the timeline splitting for temporal aggregation. Indeed, in the classic temporal approach, we need to write, in *every* query, the *implementation* of the temporal operations *and* the required coalescing operation that follows, while in the temporal types approach, this functionality is provided by the system. Sections 7 and 8 show that similar results are obtained in a DW environment, also reducing the execution times of the queries.

6 Temporal OLAP operators

We now define the temporal equivalents of the classic (non-temporal) *roll-up* and *dice* OLAP operators, for the three implementations in Sect. 4. In what follows, we assume the reader is familiar with the non-temporal operations. We will use the example illustrated in Fig. 15. There is an item i_1 introduced at instant t_1 with unit price €10 belonging to category b_1. At t_2, its unit price changed to €15. At t_3 it was assigned to brand b_2, and at t_4 it was discontinued from selling. Finally, the product was reintroduced at t_5 with unit price €18 and brand b_2, and it is available for selling until today. The evolution of i_1 and the corresponding instance of dimension Item are shown in Fig. 15a, b, respectively. We also assume that item i_2 belonged to brand b_2 throughout its entire lifespan, and item i_3 to brand b_1, also during its whole lifespan (Fig. 15b).

Temporal Roll-up A non-temporal roll-up operation aggregates fact data up to a dimension level using the parent–child relations defined in the dimension hierarchies. The temporal roll-up operation is similar, except that it must consider the state of the roll-up hierarchy at a specific instant. Depending on the time instant used to obtain the state of the hierarchy, the temporal roll-up can be *temporally-consistent* or *time-sliced*. Figure 16 depicts an example scenario to explain these operations. Figure 16b shows the current state of the aggregation hierarchy from items to brands.

In a *temporally-consistent* roll-up, the state of the hierarchy is obtained at the time when *each fact instance* occurred. For example, if a user wants to know the *total quantity sold for each brand, considering the item-brand assignment valid at the time of the sales*, she must perform a temporally-consistent roll-up. Figure 16c shows the result of a temporally-consistent roll-up on the Sales fact from Fig. 16a. The sales of each product are aggregated to the brand to which it belonged at the time indicated in column Date.



(a)

ItemID	UnitPrice	Brand
i_1	$\{10@[t_1, t_2), 15@[t_2, t_4), 18@[t_5, \text{now})\}$	$\{b_1@[t_1, t_3), b_2@[t_3, t_4), b_2@[t_5, \text{now}))\}$
i_2	$\{12@[t_1, \text{now})\}$	$\{b_2@[t_1, \text{now}))\}$
i_3	$\{8@[t_1, \text{now})\}$	$\{b_1@[t_1, \text{now}))\}$

(b)

Fig. 15 **a** Evolution of item i_1 over time; **b** Instances of level Item

A *time-sliced* roll-up aggregates measures using the state of the hierarchy at a fixed instant t. For example, to compute *the total quantity sold for each brand, using the item-brand assignment valid now*, we perform a time-slice roll-up considering the current state of the dimension (depicted in Fig. 16b). The time-slice roll-up on fact Sales from Fig. 16a is shown in Fig. 16d. For example, as depicted in Fig. 16b, the sales of item i_1 are aggregated to brand b_2. We remark that both kinds of roll-up operations are valid in any of the implementations defined in Sect. 4.3.

Temporal Dice A non-temporal dice operator filters instances of a fact based on a condition over measure values and attribute values of related dimension levels. Analogously to the case of the temporal roll-up, we can have a temporally-consistent or time-sliced dice. For example, to obtain the *sales of items that are assigned to brand b_2 and have a price equal or higher than €10*, the operator must consider whether we want to obtain the sales of the products that were assigned to brand b_2 either *at the time of the sales* or that belong to this brand now. Since the price of an item may also change over time, the same considerations apply to the price as well. For example, Fig. 16c shows the result of a time-slice dice that keeps the items that *currently* have brand b_2 and a price equal or higher than €10. Obviously, we can mix the two interpretations, for example, by asking sales of items *currently* assigned to category b_2 and that have a price equal or higher than €10 *at the time of the sale*.

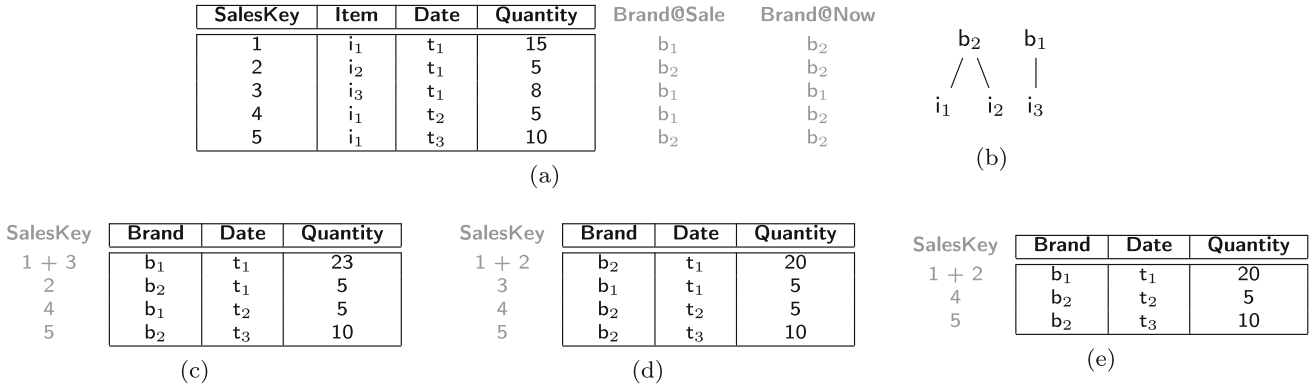


Fig. 16 **a** Fact Sales where the brands of items at the time of the sale and now are shown in gray; **b** Instances of the Brands hierarchy valid now; **c** Fact Sales after a temporally-consistent roll-up to level Brand using the state of the hierarchy valid at Sales.Date; **d** Fact Sales after a time-slice roll-up to level Brand using the state of the hierarchy valid now; **e** Result of a time-sliced dice operation on the fact Sales for current brand b_2 and current price greater than €10

7 Temporal OLAP operators in SQL

In this section, we show how to express the temporal OLAP operations defined in Sect. 6 in the three implementations presented in Sect. 4.3. For this, we extended the temporal algebra queries in Sect. 5 with the aggregation of fact data, yielding temporal OLAP queries. The queries are shown in Table 4, indicating also the query type. Notice that while algebra queries have the template "*Time when . . .*," the corresponding OLAP queries have the template "*Total sales and time when . . .*" For the sake of clarity, we have created views for each of the temporal algebra queries of Sect. 5. These views (named, e.g., Q1_MobDB, Q3_TDW, or Q5_SCD) are used in the temporal OLAP queries below. In Sect. 8 we compare their performance over the three implementations.

Temporal OLAP Selection This operator implements a temporal dice operator. As explained in Sect. 6, a dice can be temporally-consistent or time-sliced. The query below corresponds to the former type.

Query 1 "*Total sales and time when an item has brand B.*"

In the TDW approach, the query is written as:

```
SELECT q.i_item_id, q.FromDate, q.ToDate,
  SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, Q1_TDW q
WHERE s.ss_sold_date_sk = d.d_date_sk AND
  s.ss_item_id = q.i_item_id AND
  q.FromDate <= d.d_date AND d.d_date < q.ToDate
GROUP BY q.i_item_id, q.FromDate, q.ToDate
ORDER BY q.i_item_id, q.FromDate;
```

Notice that the query above uses the view Q1_TDW, which corresponds to the TDW version of the temporal selection in Query 1. Further, the WHERE clause ensures that the date of sale is included in the period obtained in the temporal selection.

In the SCD implementation, the query reads:

```
SELECT q.i_item_id, q.FromDate, q.ToDate,
  SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, scd_item i, Q1_SCD q
WHERE s.ss_sold_date_sk = d.d_date_sk AND
  s.ss_item_id = q.i_item_id AND
  q.FromDate <= d.d_date AND d.d_date < q.ToDate
GROUP BY q.i_item_id, q.FromDate, q.ToDate
ORDER BY q.i_item_id, q.FromDate;
```

In this case, since the link between the scd_store_sales and item tables is done through the surrogate key ss_item_sk, the SCD version has one join more than the TDW one, which is also the case for the MobilityDB version for this query, shown next.

```
SELECT ib.i_item_id, ib.i_item_brand_vt AS i_brandB_vt,
  SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, mobdb_item_brand ib
WHERE s.ss_sold_date_sk = d.d_date_sk AND
  s.ss_item_id = ib.i_item_id AND
  i_brand_id = 5004001 AND
  ib.i_item_brand_vt * d.d_datespan IS NOT NULL
GROUP BY ib.i_item_id, ib.i_item_brand_vt
ORDER BY ib.i_item_id;
```

Here, the intersection (*) predicate tests whether the date of the sale intersects the interval set when the item's price satisfies the query condition.

It can be seen that the three versions of the query are very similar. Thus, in what follows, we only show the TDW version of them. In Sect. 8, we study the performance of both the temporal algebra queries in Sect. 5 and the temporal OLAP queries in this section, in the three implementation schemes.

Temporal OLAP Projection In an OLAP query, a temporal projection computes the time when a condition over an evolving feature is satisfied independently of the specific values that made the condition true. This is illustrated in the next query.

Table 4 Temporal OLAP queries and the corresponding relational operator

Query	Operator
Q1: Total sales and time when an item has brand B	Temporal OLAP selection
Q2: Total sales and time when an item has *any* brand	Temporal OLAP projection
Q3: Total sales and time when an item has brand B and its price is greater than €80	Temporal OLAP join
Q4: Total sales and time when an item has brand B or its price is greater than €80	Temporal OLAP union
Q5:Total sales and time when an item has brand B and its price is not greater than €80	Temporal OLAP difference
Q6: Total sales and time when a category has more than three items	Temporal OLAP aggregation

Query 2 "*Total sales and time when an item has any brand.*"

This query implements a temporal roll-up to brand taking into account that the assignment of items to brands evolves on time. In the TDW implementation the query reads:

```
SELECT q.i_item_id, q.FromDate, q.ToDate,
  SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, Q2_TDW q
WHERE s.ss_sold_date_sk = d.d_date_sk AND
  s.ss_item_id = q.i_item_id AND
  q.FromDate <= d.d_date AND d.d_date < q.ToDate
GROUP BY q.i_item_id, q.FromDate, q.ToDate
ORDER BY q.i_item_id, q.FromDate;
```

Temporal OLAP Join A temporal join in an OLAP query computes the joint evolution on time of two evolving features stored in different tables. This is illustrated in the following query.

Query 3 "*Total sales and time when an item has brand B and its price is greater than €80.*"

Over the TDW model, the query reads as follows.

```
SELECT q.i_item_id, q.FromDate, q.ToDate,
  SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, Q3_TDW q
WHERE s.ss_sold_date_sk = d.d_date_sk AND
  s.ss_item_id = q.i_item_id AND
  q.FromDate <= d.d_date AND d.d_date < q.ToDate
GROUP BY q.i_item_id, q.FromDate, q.ToDate
ORDER BY q.i_item_id, q.FromDate;
```

Temporal OLAP Union An OLAP query requires a temporal union to compute the time when at least one of two conditions over evolving features is satisfied.

Query 4 "*Total sales and time when an item has brand B or its price is greater than €80.*"

```
SELECT q.i_item_id, q.FromDate, q.ToDate,
  SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, Q4_TDW q
WHERE s.ss_sold_date_sk = d.d_date_sk AND
  s.ss_item_id = q.i_item_id AND
  q.FromDate <= d.d_date AND d.d_date < q.ToDate
GROUP BY q.i_item_id, q.FromDate, q.ToDate
ORDER BY q.i_item_id, q.FromDate;
```

Temporal OLAP Difference An OLAP operation requires a temporal difference to obtain the time when a condition over an evolving feature is satisfied and another condition is not.

Query 5 "*Total sales and time when an item has brand B and its price is not greater than €80.*"

This query performs a temporal roll-up operation to the brand level with an additional condition that accounts for the assignment of brands to categories, which evolves over time. To express this query in the TDW model, we proceed as follows.

```
SELECT q.i_item_id, q.FromDate, q.ToDate,
  SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, Q5_TDW q
WHERE s.ss_sold_date_sk = d.d_date_sk AND
  s.ss_item_id = i.i_item_id AND
  q.FromDate <= d.d_date AND d.d_date < q.ToDate
GROUP BY q.i_item_id, q.FromDate, q.ToDate
ORDER BY q.i_item_id, q.FromDate;
```

Temporal OLAP Aggregation An OLAP operation requires a temporal aggregation to obtain summarized values from an evolving feature, as illustrated next.

Query 6 "*Total sales and time when a category has assigned to it more than 3 items.*" This is a temporal roll-up operation, with an additional condition over the number of items of a brand.

```
SELECT q.i_category_id,
    greatest(ic.FromDate, q.FromDate) AS FromDate,
    least(ic.ToDate, q.ToDate) AS ToDate,
    SUM(s.ss_net_paid) AS TotalSales
FROM store_sales s, date_dim d, tdw_item_category ic,
    Q6_TDW q
WHERE s.ss_sold_date_sk = d.d_date_sk AND
    s.ss_item_id = ic.i_item_id AND
    ic.i_category_id = q.i_category_id AND
  /* Temporal join between item_category and Q6_TDW */
    greatest(ic.FromDate, q.FromDate) <
      least(ic.ToDate, q.ToDate) AND
    q.FromDate <= d.d_date AND d.d_date < q.ToDate
GROUP BY q.i_category_id, ic.FromDate,
  ic.toDate, q.FromDate, q.toDate
ORDER BY q.i_category_id, q.FromDate;
```

8 Experiments

In this section, we compare the performance of the temporal algebra queries in Sect. 5 (Table 3) and the temporal OLAP queries in Sect. 7 (Table 4), in the three implementations, namely SCD (Fig. 9), temporal DW (Fig. 10), and MobilityDB (Fig. 11). We ran the six queries for both cases and compared their performance using the TPC-DS Scale Factors (SF) 1, 10, 50, and 100. The number of rows and the size of each table are given in Table 5. The SQL scripts and the data for reproducing the benchmark are available.[11]

The store_sales and item tables produced by the TPC-DS generator had data integrity issues, so the following actions were taken: (a) Remove sales without order date; (b) Remove sales whose date does not intersect the item's validity interval; (c) Remove sales with no item reference; (d) Remove items without a valid interval and their corresponding sales records; (e) Remove sales whose item has no price.

The three implementations were deployed on a PostgreSQL database, version 16.1, installed on a desktop computer with an AMD Ryzen 9 3900X 12-Core Processor at 3,793 Mhz and 64 GB of memory running WSL version 2 on Windows version 10.

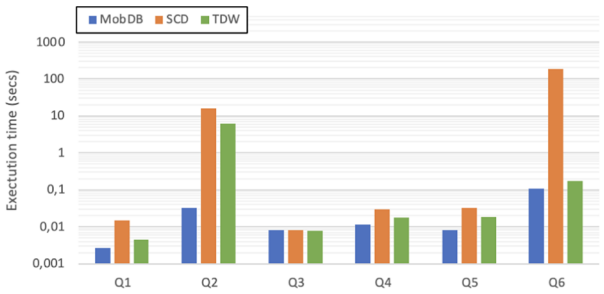
Figure 17 shows (in logarithmic scale) the execution times (in seconds) of the temporal algebra and OLAP queries at the various scale factors (SF). As a general comment, querying temporal data often involves computing the interval when an object maintains its state, e.g., the interval when an item's price remains unchanged. Since in the SCD implementation, a new record is inserted every time any of its attribute's values changes, temporal coalescing is required to obtain the interval associated with a given value, which is an expensive operation. In the TDW implementation, each temporal attribute is stored in a separate table, thus, a new record is inserted only when a change in the attribute's value occurs. Therefore, temporal coalescing is not required. Finally, the TDW implementation is highly normalized, leading to a higher number of joins compared with the SCD implementation, where all the required attributes are present in the same table.

We first discuss the results of the *temporal algebra queries*. The left-hand side of Fig. 17 shows that the MobilityDB implementation outperforms the other two ones (in some cases, by orders of magnitude), for the four SFs, except for Q3 (where performance is similar for all the cases and SFs) and Q4 (where performance is similar for SF100). The reason follows from the discussion in Sect. 5: the coalescing operation in MobilityDB takes advantage of the fact that the time intervals are temporally ordered, while in the other models, several nested NOT EXISTS predicates must be computed. We can observe the very poor performance of the TDW and SCD implementations for Query Q2 and the SCD implementation for Q6 (temporal aggregation), for all SFs. Recall from Sect. 5 that, for Q6, the SCD implementation requires additional computations. For Q3 (temporal join), the performance of the three alternatives is similar, most likely due to the effect of the performance of the greatest and least functions used in the TDW implementation, which encode, as explained in Sect. 5, the four possible temporal join cases. Also note that, except for Q3, the TDW implementation outperforms SCD for all SFs and queries.

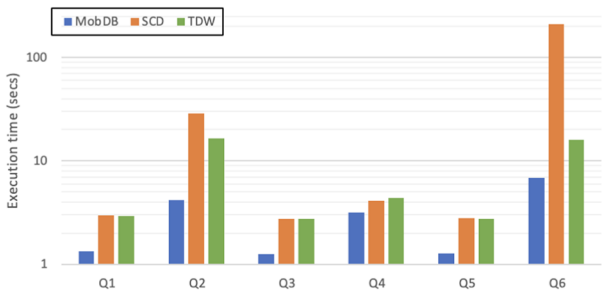
[11] https://github.com/MobilityDB/MobilityDB-TPCDS.

Table 5 Datasets used in the experiments

Table	No of rows/size SF 1	SF 10	SF 50	SF 100
All				
store_sales	2,490K	24,906K	124,537K	249,018K
	483 MB	4,834 MB	24 GB	47 GB
date_dim	2,191	2,191	2,191	2,191
	1,488 kB	1,488 kB	1,488 kB	1,552 kB
SCD				
item	17,954	101,757	61,852	203,478
	12 MB	63 MB	63 MB	125 MB
Temporal DW				
item	8,990	50,957	30,976	101,913
	1,768 kB	11 MB	6,432 kB	22 MB
item_ls	8,995	50,994	30,995	101,997
	944 kB	5,080 kB	3,096 kB	10,096 kB
item_price	17,944	101,708	61,820	203,389
	1,976 kB	11 MB	6,688 kB	21 MB
brand	949	953	953	953
	144 kB	144 kB	144 kB	144 kB
category	10	10	10	10
	24 kB	24 kB	24 kB	24 kB
item_brand	13,436	76,487	46,491	152,907
	1,488 kB	8,272 kB	5,048 kB	16 MB
item_cat	17,063	96,737	58,765	193,417
	1,880 kB	10 MB	6,368 kB	20 MB
MobilityDB				
item	8,990	50,957	30,976	101,913
	4,232 kB	23 MB	14 MB	46 MB
item_price	17,943	101,696	61,812	203,362
	2,824 kB	15 MB	9,584 kB	31 MB
brand	949	953	953	953
	512 kB	1,128 kB	1,048 kB	792 kB
category	10	10	10	10
	32 kB	96 kB	72 kB	128 kB
item_brand	13,428	76,425	46,455	152,768
	2,240 kB	12 MB	7,584 kB	24 MB
item_cat	16,805	95,156	57,796	190,239
	2,672 kB	15 MB	9,064 kB	29 MB

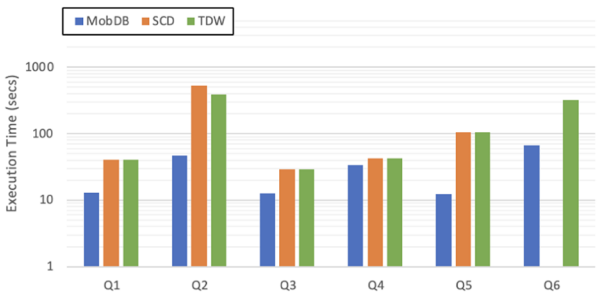


(a) Temporal algebra queries at SF1

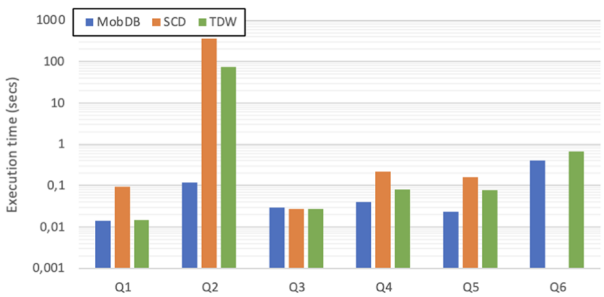


(b) Temporal OLAP queries at SF1

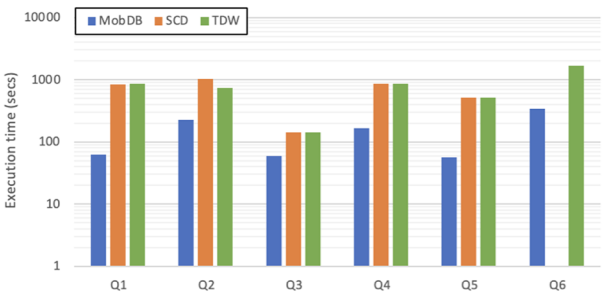
(c) Temporal algebra queries at SF10



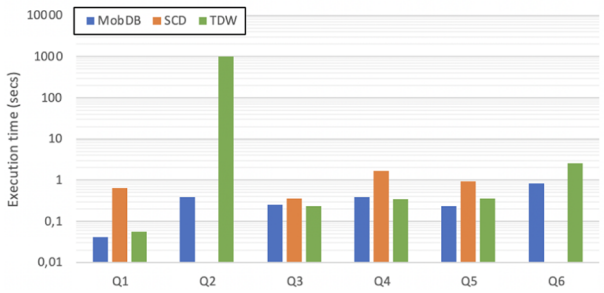
(d) Temporal OLAP queries at SF10



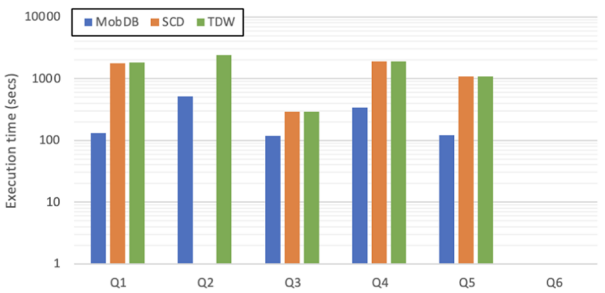
(e) Temporal algebra queries at SF50



(f) Temporal OLAP queries at SF50



(g) Temporal algebra queries at SF100



(h) Temporal OLAP queries at SF100

Fig. 17 Execution time in seconds for the temporal algebra and OLAP queries at SFs 1, 10, 50, and 100

The right-hand side of Fig. 17 shows the results for the *temporal OLAP queries*. We see that MobilityDB outperforms the other two solutions for all queries and SFs except for Q6 at SF100. Also, for all queries except for Q2 and Q6, the TDW and SCD alternatives behave similarly. For Q2 and Q6, TDW outperforms SCD. In the case of Q6, we can see that MobDB runs faster than SCD and TDW except for SF100, where the three implementations are timed out after running for one hour on our hardware. We discuss these results below.

We can see that for all algebra queries, TDW outperformed SCD. However, for the OLAP queries (which include the algebra queries as views), both implementations have, in most cases, similar performance. The reason is that the execution time of the OLAP queries is mainly the time consumed by the final OLAP aggregation. However, this is not the case for the MobDB implementation, because the aggregation is performed more efficiently taking advantage of the time types.

The case of Q6 (temporal aggregation) is worth more elaboration. First, we can see that our approach achieves efficient performance, as the results for the algebra queries on the left-hand side of Fig. 17 show, even though temporal aggregation is known to be an expensive operation [6, 7]. Second, analyzing the query processing strategies we see that PostgreSQL's query planner produces different strategies for the TDW and MobDB implementations (the query plans are available at the Github site). For the latter, a bottleneck is generated when performing two hash joins. This bottleneck does not appear in the TDW strategy that PostgreSQL proposes, since it materializes intermediate results. The reason is that PostgreSQL uses a default selectivity estimation of the result of the aggregation which, in the case of MobilityDB, deviates by orders of magnitude from the actual number of tuples in the result, and thus PostgreSQL chooses an inefficient order of the joins required for the OLAP operation. Although this occurs for all the SFs in Fig. 17, the problem only shows at SF 100 due to the size of the store_sales table. MobilityDB provides to PostgreSQL statistics for estimating the selectivity of operations on time and temporal types, in terms of histograms of lower and upper bounds. The statistics are accurately computed when at least one of the arguments of the operation is a base table column such as

```
i_item_category_vt * '[2021-01-01, 2021-02-01)' or
i_item_category_vt * i_item_brand_vt,
```

while a default selectivity applies for expressions like

```
tCount(i_item_category_vt) or
whenTrue(tCount# > 3),
```

used in Query 6. Thus, although the temporal aggregation is computed efficiently in MobilityDB, results are strongly affected by the processing strategy explained. In future work we will address this problem.

9 Related work

We now review literature on temporal data management, relevant to our proposal.

Temporal Databases There is a large corpus of work on temporal databases [31] and they still capture the interest of researchers and practitioners [13]. The two main approaches for extending relational databases with temporal semantics are tuple timestamping and attribute timestamping. The former yields first normal form relations (1NF), while the latter produces non-first normal form relations (N1NF). Attribute timestamping was initially proposed by Clifford and Tansel [10]. Later, in his seminal work, Gadia [12] introduces a temporal algebra and calculus, proving their equivalence. Jensen et al. [19] propose a temporal conceptual model that captures the temporal semantics of data in a unified way, timestamping the tuples of relations with sets of two-dimensional so-called *chronons*, thus supporting valid and transaction time. For mapping between models, the notion of *snapshot equivalence* is used. Both tuple- and attribute-timestamped representations are studied, mapping one to another using a conceptual model. To the best of our knowledge, this proposal has not been implemented. Another discussion in temporal databases refers to the representation of time, which can be point based or interval based. This is studied in [32]. A point-based temporal extension to SQL is later proposed by the same author [33]. Using the point-based approach, Chen et al. [9] propose a temporal extension to SQL. However, in practice, the interval-based approach is easier to implement and understand.

A large number of temporal query languages in the early 1990 s raised the need for a standardized temporal relational query language. As a result, TSQL2 [28] is proposed as a temporal extension to the SQL-92 standard. TSQL2 introduces a new *period* datetime data type. Valid, transaction, and user-defined times are supported in TSQL2. There are two types of tables, namely, state and event. In the former, each tuple is timestamped with a time element (a union of periods), while in the latter each tuple is timestamped with an instant set. The classic CREATE TABLE, UPDATE, and ALTER statements in standard SQL are modified to allow the specification of temporal elements. For data manipulation, the FROM clause allows coalescing the tuples that have identical values of non-temporal attributes. Finally VALID() and TRANSACTION() functions are introduced to access the valid and transaction times in the WHERE clause.

The SQL:2011 standard [22] includes some temporal features like a period data type, temporal primary keys, temporal referential integrity constraints, temporal predicates,

valid, transaction and bitemporal tables, as well as temporal insertions, updates, and deletions. However, operators like temporal coalescing [38], temporal joins, and temporal aggregation are not included in the standard, and the most popular database vendors provide a limited support for temporal features in their RDBMSs. Therefore, practitioners still use standard SQL to manipulate time-varying information. Snodgrass [29] shows how most relational operations can be written in standard SQL. Zimányi [40] then shows how to implement temporal aggregates and temporal universal quantifiers using standard SQL. Tuple timestamping is adopted in those proposals.

The work by Dignös et al. [11] is closely related to the present paper, in particular for the case of so-called sequenced temporal queries, namely queries that are evaluated at each time point (e.g., Queries 3 in Sect. 5 and 6 in Sect. 7). The idea is to reduce a temporal query to non-temporal operators. For this, the authors follow a four-step strategy composed of the following steps: (a) interval propagation; (b) interval adjustment; (c) scaling; (d) evaluation of non-temporal operators. Interval propagation refers to the duplication of the original intervals, which are later used for scaling. Interval adjustment is composed of two operations: normalization and alignment. Normalization splits the timeline according to the intervals of the tuples in a relation, and it is used for group-based operators (projection, difference, union, intersection). It requires to extend SQL with a NORMALIZE operation. Alignment is used in tuple-based operations (e.g., joins, selection, Cartesian product), and splits a tuple of a relation according to the time intervals of value-equivalent tuples in another relation. It is analogous to the cases explained in Sect. 5 (Queries 3 and 5). The implementation of the temporal alignment is similar to the MobilityDB implementation of the set operations on time values (Sect. 3) except that in MobilityDB the operations are manipulated by the time types, while in [11] a sweep plane algorithm is implemented as a PostgreSQL function. Further, the implementation using skiplists in MobilityDB makes the operations very efficient. Finally, a query plan is produced applying a set of reduction rules that transform queries expressed using temporal relational operators, to a query expressed in terms of relational operators and the two new operators. In summary, the proposal in [11] follows the classic idea of interval-based implementation and tuple timestamping. The approach requires extending SQL with the operations commented above, that must be mentioned explicitly in the queries (e.g., ALIGN, NORMALIZE, SCALE). Opposite to this, in MobilityDB they are provided built-in, through two mechanisms: extended data types and attribute timestamping. As a result, sequence queries in MobilityDB can be solved easily using temporal operations like tSum, tCount or tUnion in a very concise way, without requiring rewriting. The authors proposed a proof-of-concept implementation of their approach using PostgreSQL 9.5. As far as we know, the implementation has not been integrated into PostgreSQL.

Tsikoudis et al. [35] introduce RQL (Retrospective Query Language), a language for specifying SQL computations over collections of snapshots of past states of a data store like Berkeley DB. RQL iteratively runs an SQL query on each snapshot, collects the results and performs computations over the latter. As a follow-up, the authors extend this work presenting an optimization framework called RID [34] that detects and eliminates redundant computations in queries. RQL can address some of the queries presented here, and it is somehow related to [11] since it can be used to address sequenced queries. However, the sequenced semantics of a temporal query, which views a temporal database as a sequence of snapshot databases and evaluates the query at each of these snapshots, provides a clear semantics for theoretical studies, but a practical implementation needs additional considerations [5]. Therefore, we do not include snapshot-based databases in this work. Finally, Lu et al. [23] presents a temporal implementation in TDSQL Tencent's distributed DBMS, keeping the traditional data types and the tuple timestamping approach, focusing on data storage and query rewriting.

MobilityDB [41] is a spatiotemporal DBMS that extends the type system of PostgreSQL and PostGIS with abstract data types, to represent spatiotemporal data. MobilityDB seamlessly extends the DBMS with temporal data types, not requiring any additional software architecture. Although MobilityDB was initially designed to manage and query mobility data, since it provides temporal types over both alphanumeric and spatial base types, it is actually a temporal DBMS that can handle both tuple and attribute timestamping. To address scalability, a distributed version of MobilityDB is also available [2, 3]. We do not extend here since MobilityDB was covered in detail in Sect. 3.

Temporal Data Warehousing The most popular approach for tackling changes in the attribute values of dimension tables in a DW, is *slowly changing dimensions* (SCDs) [21]. Since this is a well-established solution adopted by practitioners, we chose it as one of the models to compare against our proposal. We have extensively covered SCDs in previous sections.

The *temporal star schema* [4] aims at overcoming the limitations of the SCDs solution. Instead of using a time dimension table, timestamps are defined for each row in every table, allowing performing time calculations without joining the fact table with the time dimension. A single timestamp in a table means that the data are event-oriented and represents the time when an event has occurred, while a pair of timestamps (called *period* timestamps) indicate that the data in the table are state-oriented and denote the period during which a state has persisted. Handling periods raises issues, which in our approach are solved by the MobilityDB time

data types. Further, omitting the time dimension makes it difficult to aggregate data over time and to define particular information on certain instants (like holidays, weekdays, and so on). Finally, temporal joins or temporal aggregation are not considered, and the discussion is limited to just four kinds of queries. Since the SQL expressions of these queries are not shown, we do not use this approach to compare against our proposal.

The *starnest schema* [15], based on the nested relational model [27], combines the advantages of the star and snowflake schemes, avoiding join operations by storing each dimension in a single nested table, preserving the dimension hierarchy. The levels of a dimension hierarchy are recursively nested, from the most generic level to the most specific one. Each parent level tuple nests the child member sub-tuples as a value of one of its attributes. If a dimension contains multiple hierarchies, tables must be duplicated. Based on this work, the *temporal starnest schema* is proposed in [14], addressing time-evolving dimension members. Timestamps are used to track the evolution of dimension members. Like in [4], the time dimension is not made explicit. Instead, valid and transaction time attributes are included in the fact table and the temporal dimensions. Temporal nested queries are processed in a non-standard query language called BTN-SQL. The main drawback of this model is its complex structure. No implementation is reported, so we do not compare this model against our proposal.

Mahlknecht et al. [24] represent facts timestamped with intervals presenting three implementations: (a) The *instant* model, where each fact is linked through a foreign key from the time dimension; (b) The *period* model, where intervals are represented as two foreign keys to the time dimension; (c) The $period^*$ model, where a fact is associated with a period stored in a separate table, linked to the fact table through a foreign key. The authors present a set of queries that perform aggregation at every time instant, showing that they are easier to express in SQL using the $period^*$ model since periods are explicitly stored. In our approach, these queries are solved by the built-in datespanset data type, without the need of implementing foreign keys for each fact. Further, our approach can handle sets of periods in a native fashion. Aggregation along the time dimension is also performed in [24]. We have shown that this operation can be natively done in our approach. The experiments show a clear edge in favor of the $period^*$ model, which in fact is aligned with our approach. Compared to our work, a limited variety of temporal queries are supported. For example, temporal joins and temporal difference are not studied. Also, unlike the TDW model mentioned below, only atelic [20] measures are addressed.

To overcome the problems of the works discussed above, Ahmed et al. [1] introduced the TDW model. We have extensively discussed this model in previous sections. Interested readers can find detailed information in [37]. Finally, Vaisman and Zimányi [36, 37] show that MobilityDB can also be used as a solution for (spatio)temporal DW.

10 Conclusion

In this paper we revisit the long-time studied tuple versus attribute timestamping dichotomy in temporal databases and DWs. Actually, the main reason for the popularity of the tuple timestamping is that 1NF databases were so far the only implementation alternative. The release of the MobilityDB database opens up the possibility of efficiently implementing attribute timestamping and, at the same time, enhancing tuple timestamping, taking advantage of the rich collection of time data types and operations on these types.

To show the feasability of the MobilityDB approach, we implement a portion of the TPC-DS benchmark using three alternative implementations: the SCD Type 2 model, the temporal DW model, and the MobilityDB DW model. We define six representative queries, implementing the temporal selection, projection, union, difference, join, and aggregation operations. Then, we use these six queries as the basis for providing a temporal extension of the classic roll-up and dice OLAP operations. Our experiments show that the MobilityDB approach outperforms the other two ones, except for the temporal OLAP aggregation query at SF 100, where the query plan produced by PostgreSQL is suboptimal and thus, although the temporal aggregation is computed efficiently in MobilityDB, the execution of the query is strongly affected by the processing strategy. We will be study this in future work.

Another future perspective is to generalize the results of this paper to a distributed setting, using a distributed version of MobilityDB based on Citus,[12] an open source distributed extension for PostgreSQL, along the lines of the approach shown in Sect. 2.4. MobilityDB partitions spatiotemporal data using multidimensional tiling, where the n-dimensional space is split into grid tiles that are assigned to cluster nodes. With respect to traditional hash-based partitioning, this preserves data locality, which is essential for location-based queries. Since metadata about the spatiotemporal extent covered by cluster nodes are kept by the query planner, only the nodes that cover the extents needed by the query are requested to work. We plan to generalize multidimensional tiling for OLAP, which requires to support the high number of dimensions in a DW, instead of one to four dimensions as is spatiotemporal data.

Acknowledgements A. Vaisman and L. Gómez were partially supported by Project PICT 2017-1054, from the Argentinian Scientific

[12] https://www.citusdata.com/.

Agency. The authors also thank Maxime Schoemans for his invaluable help in the analysis and optimization of the queries studied in this paper.

Declarations

Conflict of interest The authors declare no Conflict of interest.

References

1. Ahmed, W., Zimányi, E., Vaisman, A.A., Wrembel, R.: A temporal multidimensional model and OLAP operators. Int. J. Data Warehous. Min. **16**(4), 112–143 (2020)
2. Bakli, M., Sakr, M., Zimányi, E.: Distributed mobility data management in MobilityDB. In: Proceedings of the 21st IEEE international conference on mobile data management, MDM 2020, pp. 238–239. IEEE Computer Society Press (2020)
3. Bakli, M., Sakr, M., Zimányi, E.: Distributed spatiotemporal trajectory query processing in SQL. In: Proceedings of the 28th international conference on advances in geographic information systems, SIGSPATIAL'20. ACM (2020)
4. Bliujute, R., Saltenis, S., Slivinskas, G., Jensen, C.S.: Systematic change management in dimensional data warehousing. Technical Report TR-23, Time Center (1998)
5. Böhlen, M.H., Dignös, A., Gamper, J., Jensen, C.S.: Temporal data management: an overview. In: Proceedings of the 7th european summer school on business intelligence and big data, eBISS 2017, **324**, 51–83 (2017)
6. Böhlen, M.H., Dignös, A., Gamper, J., Jensen, C.S.: Database technology for processing temporal data. In: Proceedings of the 25th Int. Symp. on Temporal Representation and Reasoning, TIME **2018** (2018)
7. Böhlen, M., Gamper, J., Jensen, C.S.: Towards general temporal aggregation. In: Proceedings of the 25th british national conference on databases, BNCOD08, pp. 257–269 (2008)
8. Böhlen, M.H., Snodgrass, R.T., Soo, M.D.: Coalescing in temporal databases. In: Proceedings of 22th international conference on very large data bases, VLDB'96, pp. 180–191. Morgan Kaufmann (1996)
9. Chen, C.X., Kong, J., Zaniolo, C.: Design and implementation of a temporal extension of SQL. In: Proceedings of the 19th international conference on data engineering, 2003, pp. 689–691. IEEE Computer Society (2003)
10. Clifford, J., Tansel, A.U.: On an algebra for historical relational databases: two views. In: Proceedings of the 1985 ACM SIGMOD international conference on management of data, pp. 247–265. ACM Press (1985)
11. Dignös, A., Böhlen, M.H., Gamper, J., Jensen, C.S.: Extending the kernel of a relational DBMS with comprehensive support for sequenced temporal queries. ACM Trans. Database Syst. **41**(4), 26:1-26:46 (2016)
12. Gadia, S.K.: A homogeneous relational model and query languages for temporal databases. ACM Trans. Database Syst. **13**(4), 418–448 (1988)
13. Gamper, J., Ceccarello, M., Dignös, A.: What's new in temporal databases? In: Proceedings of the 26th European Conference on Advances in Databases and Information Systems, ADBIS 2022, LNCS 13389, pp. 45–58. Springer (2022)
14. Garani, G., Adam, G.K., Ventzas, D.: Temporal data warehouse logical modelling. Int. J. Data Min. Model. Manag. **8**(2), 144–159 (2016)
15. Garani, G., Helmer, S.: Integrating star and snowflake schemas in data warehouses. Int. J. Data Warehous. Min. **8**(4), 22–40 (2012)
16. Güting, R.H., Böhlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M., Vazirgiannis, M.: A foundation for representing and quering moving objects. ACM Trans. Database Syst. **25**(1), 1–42 (2000)
17. Hümmer, W., Lehner, W., Bauer, A., Schlesinger, L.: A decathlon in multidimensional modeling: Open issues and some solutions. In: Proceedings of the 4th international conference on data warehousing and knowledge discovery, DaWaK, LNCS 2454, pp. 275–285. Springer (2002)
18. Hurtado, C.A., Mendelzon, A., Vaisman, A.A.: Maintaining data cubes under dimension updates. In: Proceedings of the 15th international conference on data engineering, ICDE'99, pp. 346–355 (1999)
19. Jensen, C.S., Soo, M.D., Snodgrass, R.T.: Unifying temporal data models via a conceptual model. Inf. Syst. **19**(7), 513–547 (1994)
20. Khatri, V., Ram, S., Snodgrass, R.T., Terenziani, P.: Capturing telic/atelic temporal data semantics: generalizing conventional conceptual models. IEEE Trans. Knowl. Data Eng. **26**(3), 528–548 (2014)
21. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd edn. John Wiley & Sons (2013)
22. Kulkarni, K., Michels, J.-E.: Temporal features in SQL:2011. SIGMOD Record **41**(3), 34–43 (2012)
23. Lu, W., Zhao, Z., Wang, X., Li, H., Zhang, Z., Shui, Z., Ye, S., Pan, A., Du, X.: A lightweight and efficient temporal database management system in TDSQL. Proc. VLDB Endow. **12**(12), 2035–2046 (2019)
24. Mahlknecht, G., Dignös, A., Kozmina, N.: Modeling and querying facts with period timestamps in data warehouses. Int. J. Appl. Math. Comput. Sci. **29**(1), 31–49 (2019)
25. Malinowski, E., Zimányi, E.: A conceptual model for temporal data warehouses and its transformation to the ER and the object-relational models. Data & Knowl. Eng. **64**(1), 101–133 (2008)
26. Nambiar, R.O., Poess, M.: The making of TPC-DS. In: Proceedings of the 32nd international conference on very large data bases, VLDB, pp. 1049–1058. ACM (2006)
27. Paredaens, J., De Bra, P., Gyssens, M., Van Gucht, D.: The nested relational database model. In: The structure of the relational database model, pp. 177–201. Springer (1989)
28. Snodgrass, R.T. (ed): The TSQL2 Temporal Query Language. Kluwer Academic (1995)
29. Snodgrass, R.T.: Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann (2000)
30. Snodgrass, R.T., Ahn, I.: A taxonomy of time databases. ACM SIGMOD Rec. **14**, 236–246 (1985)
31. Tansel, A.U., Clifford, J., Gadia, S., Jajodia, S., Segev, A., Snodgrass, R.T.: Temporal Databases: Theory, Design, and Implementation. Benjamin Cummings (1993)
32. Toman, D.: Point vs. interval-based query languages for temporal databases. In: Proceedings of ACM PODS, pp. 58–67. ACM Press (1996)
33. Toman, D.: Point-based temporal extension of temporal SQL. In: Proceedings of the 5th international conference on deductive and object-oriented databases, DOOD'97, LNCS 1341, pp. 103–121. Springer (1997)
34. Tsikoudis, N., Shrira, L.: RID: deduplicating snapshot computations. In: Proc. of the 2020 Int. Conference on Management of Data, SIGMOD 2020, pp. 2087–2101. ACM (2020)
35. Tsikoudis, N., Shrira, L., Cohen, S.: RQL: retrospective computations over snapshot sets. In: Proceedings of the 21st international conference on extending database technology, EDBT 2018, pp. 600–611 (2018)
36. Vaisman, A.A., Zimányi, E.: Mobility data warehouses. ISPRS Int. J. Geo-Inf. **8**(4), 170 (2019)

37. Vaisman, A.A., Zimányi, E.: Data Warehouse Systems: Design and Implementation, 2nd edn. Springer (2022)
38. Zhou, X., Wang, F., Zaniolo, C.: Efficient temporal coalescing query support in relational database systems. In: Proceedings of the 17th conference on database and expert systems applications, DEXA, pp. 676–686 (2006)
39. Zimányi, E.: Query evaluation in probabilistic relational databases. Theoret. Comput. Sci. **171**(1–2), 179–219 (1997)
40. Zimányi, E.: Temporal aggregates and temporal universal quantifiers in standard SQL. SIGMOD Record **32**(2), 16–21 (2006)
41. Zimányi, E., Sakr, M., Lesuisse, A.: MobilityDB: a mobility database based on PostgreSQL and PostGIS. ACM Trans. Database Syst. **45**(4), 19:1-19:42 (2020)