



# A new window Clause for SQL++

James Fang<sup>1</sup> · Dmitry Lychagin<sup>2</sup> · Michael J. Carey<sup>3</sup> · Vassilis J. Tsotras<sup>1</sup>

Received: 9 February 2023 / Revised: 28 July 2023 / Accepted: 14 November 2023 / Published online: 19 December 2023  
© The Author(s) 2023

## Abstract

Window queries are important analytical tools for ordered data and have been researched both in streaming and stored data environments. By incorporating ideas for window queries from existing streaming and stored data systems, we propose a new window syntax that makes a wide range of window queries easier to write and optimize. We have implemented this new window syntax in SQL++, an SQL extension that supports querying semistructured data, on top of AsterixDB, a Big Data Management System, thus allowing us to process window queries over large datasets in a parallel and efficient manner.

## 1 Introduction

Large amounts of data are being generated daily (web logs, transaction logs, sensor data, etc.). Much of this data is ordered, e.g. tweets ordered by time, and this order is important for computing analytics over the data such as running averages, etc. An important tool for computing analytics over ordered data is the *window query* which effectively breaks an ordered data sequence into smaller and possibly overlapping ordered sub-sequences (windows), whose tuples are then processed with aggregation to produce query results. Windowing allows the user to compute an answer over each of these windows. Data is either analysed as it arrives, e.g. streaming systems, or is stored for later historical analytical query processing, also called offline or batch processing systems.

This work emanated from the desire to create an alternative windowing facility for processing windows over big data that is both richer in its capabilities to support additional common

use cases, and hopefully easier for users to learn and use. In particular, we wanted a natural, usable windowing facility for the Apache AsterixDB system [3], a parallel shared-nothing big data management system that uses SQL++ [12, 30] as its query language. SQL++ is a SQL-like query language designed to work with both structured and semi-structured data. Currently SQL++ in AsterixDB (and Couchbase Analytics for that matter [21]) supports the OVER “Clause” of SQL, but as we argue in Sect. 3, it does not meet our requirements.

While SQL OVER may involve more compact queries, it is less intuitive. In particular, having the SQL OVER inside the SELECT can be confusing since the SQL OVER may introduce GROUP\_BY and ORDER\_BY within the SELECT itself. This is in contrast to the logical evaluation order that the user is familiar with from traditional SQL, where GROUP\_BY and ORDER\_BY are performed before and after SELECT, respectively. Further, OVER belongs to a window function call which is actually an *expression* which imposes various limitations. Nevertheless, the SQL OVER Clause offers various advantages, for example its simple framing.

XQuery gives a predicate-based approach to express window queries. This allows for more flexible boundaries on when a window can start or end. Additionally, the creation of predicate variables allows for easier access to information within the window. An example of a sliding window of size 3 tuples can be seen in Fig. 1 for the OVER and XQuery approaches.

By first reviewing previous window approaches from both streaming and stored data environments (namely SQL and XQuery), our proposed WindowBy Clause in SQL++

✉ James Fang  
jfang003@ucr.edu

Dmitry Lychagin  
dmitry.lychagin@snowflake.com

Michael J. Carey  
mjcarey@ics.uci.edu

Vassilis J. Tsotras  
tsotras@cs.ucr.edu

<sup>1</sup> University of California, Riverside, Riverside, USA

<sup>2</sup> Snowflake Inc. (Work Performed While at Couchbase Inc.),  
San Mateo, USA

<sup>3</sup> University of California, Irvine, Irvine, USA

```

SELECT station,
       date,
       AVG(val) OVER (ORDER BY date
                     ROWS BETWEEN CURRENT ROW
                           AND 2 FOLLOWING)
FROM Ghcn

(a)

for sliding window $w in //Ghcn,
  start s at sp when fn:true()
  end at ep when ep - sp = 2
return <window>
  <station>{$s/station} </station>
  <date>{$s/date} </date>
  <avg>{avg($w)} </avg>
</window>

(b)

```

Fig. 1 Sliding Window of Size 3 using **a** OVER and **b** XQuery

combines advantages from both worlds. The syntax of our proposed window Clause allows users to write window queries in a simpler and more intuitive way. Furthermore, the new Clause can be implemented in a horizontally scalable way in the AsterixDB big data management system.

The rest of this paper is organized as follows: Sect. 2 provides basic background on windows and summarizes related work. The desiderata for a new window Clause appear in Sect. 3.1 and are based on adapting advantages from previous window approaches. Section 3.2 examines how these desiderata can be supported in SQL++, while the details of the proposed SQL++ WindowBy Clause are presented in Sect. 3.3. Section 4 shows how various common window queries are written in a more intuitive way using the proposed WindowBy Clause as compared to previous approaches. A prototype implementation of WindowBy in AsterixDB is described in Sect. 5 while Sect. 6 shows preliminary experimental evaluation results. Conclusions and future work are described in Sect. 7.

## 2 Background and related work

### 2.1 Terminology

Each window is defined by its two boundaries: *start* (which comes first) and *end* (which comes later in the sequence order). The data sequence order is thus crucial. A window boundary occurs when one of the following events happen: (1) a new tuple arrives, or (2) some system time has elapsed (e.g. after 20 min or at the start of the hour). Case (1) refers to the *event time*, while case (2) refers to the *processing time*, i.e., the system time when a tuple is processed. In a streaming environment, window boundaries can also be cre-

ated based on a system clock (and not only based on a new tuple's attribute values).

Creating windows over stored data requires that the data is first ordered based on some attribute value; that is, data needs to be sorted first if it is not already ordered. This is typically the tuple's event time. Thus in a stored data environment, window boundaries are defined based on event time. Another difference between streaming and stored data windows is that in a streaming environment, tuples may arrive late (and thus out of sequence order) and will be eliminated from the window results.

Depending on the relative positions of the window boundaries different window types have been discussed in the literature; the most common ones are *sliding*, *tumbling*, *landmark* and *sessions* [2, 27, 32]. In sliding windows a tuple can belong to many individual windows (thus windows can overlap). In contrast, in tumbling windows each tuple can participate in at most one window (i.e., windows do not overlap). Landmark windows are considered as a special case of sliding where every window's start is fixed at a specific "landmark" (time or tuple), typically the start of the ordered sequence [27, 32]. Multiple landmarks are also possible [9]. The notion of sessions has been used in streaming systems to capture periods of specific activity in the data sequence; e.g. tuples that arrive close to each other, say within 5 min. Sessions are special cases of tumbling windows and are typically defined using a timeout (and possibly a window maximum duration) [2, 17, 36].

This paper concentrates mainly on sliding and tumbling windows as they are the most commonly used in practice. In addition, we discuss how to support typical session windows in Sect. 4.3.

### 2.2 Streaming environment

Early streaming systems such as Aurora [1], TelegraphCQ [14] and STREAM [39] were the first to define windows as a way to process the incoming data into smaller analyzable subsets. More recent streaming systems focus on using partitioning to improve latency and throughput, e.g. Spark [41], Storm [22] and Flink [11]. Other works consider hardware optimization [13, 25, 28, 36]; examples are Saber focusing on GPUs and Streambox focusing on multi-core processors. There is also work on window aggregation [34, 35, 37], how to process out of order data in a window [2, 35, 42], and watermarks which determine how late data can arrive and still be processed by a window [5].

Most streaming systems have their own way of expressing windows and this differs between systems. There are two main approaches: using dataflow languages or SQL-like languages. The dataflow languages mainly follow an approach similar to the Google Dataflow Model [2]: they create operators for the windows, add properties to these operators and

follow a Directed Acyclic Graph approach to process these windows over the streaming data. Examples include Aurora [1], Flink's Data Stream API [11], Streambox [28], Microsoft Trill [13] and IBM's SPL (Stream Processing Language) [20].

One early example of an SQL-like window language is STREAM's Continuous Query Language (CQL) [4] which supported the creation of sliding windows in the FROM Clause by allowing the user to specify either time-based windows using the RANGE keyword or row-based windows using the ROWS keyword.

Another example appears in [27] where they model windows using RANGE, SLIDE, and WATTR. RANGE defines the window length in tuples or duration, SLIDE determines how far from the start of the previous window to start a new window in tuples or duration, and WATTR is the ordering attribute to order the data. Note that while this approach is intuitive and simple for the user to define windows, it is limited to creating fixed size windows.

As more stream processing systems were created, there has been work to develop a streaming SQL standard [23]. A streaming SQL example can be found in Apache Calcite's SQL [6], used by projects such as Flink [11] and Samza [31]. In Calcite, the SQL OVER Clause is still used to create tuple-based sliding windows. However, Calcite also supports other window types such as tumbling and hopping (a sliding window using the system clock). These can be implemented through the use of the TUMBLE or HOP keywords in the GROUP BY Clause along with their parameters: the ordering attribute, the window length, and a window slide (for hopping windows). These windows are grouped together based on their ending tuple, as it is impossible to know when a window is complete before the last tuple closes it, and can be accessed in the SELECT Clause through the keywords TUMBLE\_END or HOP\_END respectively using the same parameters to find the grouping value.

Note that representing hopping windows in GROUP BY violates the relational semantics of SQL since in GROUP BY each input tuple belongs to exactly one group, which is not the case in hopping windows that are special cases of sliding windows. A proposed variation for streaming SQL [7] addresses this issue by moving the window creation (using TUMBLE or HOP) into the FROM Clause using table valued functions. Another feature of this approach is the use of EMIT that allows the user to control when results are output based on an event or system time.

### 2.3 Stored data environment

Windows in stored data environments have been expressed using SQL for relational data and XQuery for semistructured data.

Since the OVER Clause was added to the SQL standard [43], window functions have attracted increasing interest and are implemented by the major database vendors. According to a 2016 survey, SQL window queries accounted for 4% of all queries in a workload collected from a multi-year deployment of a database-as-a-service platform [24]. A characteristic of SQL OVER is that it creates a window for each tuple in the sequence. There is also work on optimizing SQL OVER in a parallel environment [8, 10, 26] and on speeding up window queries using global sampling [33], holistic aggregates [38] and reducing data transfer [15]. A disadvantage of SQL OVER is that it is not as intuitive (from a usability perspective) for writing window queries. Various SQL concepts (like ORDER BY, GROUP BY) are reused within the OVER Clause, making window query writing quite challenging.

Another approach to writing window queries in SQL is by using the MATCH\_RECOGNIZE Clause [29], which is used to identify events in a time sequence. MATCH\_RECOGNIZE accepts a set of rows as input, and returns all matches for a given data pattern. It can thus identify windows that match a specific data pattern. MATCH\_RECOGNIZE is used within FROM and effectively creates an elaborate sub-query on its own, which makes it even more complex and less intuitive than OVER for writing window queries. Note that MATCH\_RECOGNIZE can express all of the OVER and WindowBy example queries used in this paper. However, it results in queries that are very difficult to optimize (see the discussion in [16] about the optimizer). There has been some work to optimize parts of MATCH\_RECOGNIZE for queries with specific conditions and no duplicates [44].

Finally, XQuery has been extended to support window queries over stored semi-structured data [9]; here the order is the binding sequence provided by the document. XQuery windowing is different from SQL in that it uses a *predicate* based approach to define the start and end of a window. This is very powerful as it allows the user to create a window using a predicate on a tuple's attribute values (i.e. a window does not need to be based on the only sequence order as in SQL OVER). It also allows the user to choose what type of window to use: tumbling, sliding or landmark. Note that the latest XQuery 3.1 standard Window Clause [40] does not support landmark windows. There has also been work to extend XQuery to allow patterns between the window start and end [18]. To the best of our knowledge there is no work extending XQuery windowing in a scalable way for parallel systems.

## 3 The proposed WindowBy Clause

The aim of the proposed WindowBy Clause is to provide an intuitive way for users to create a variety of useful window

```

WINDOW_BY := WINDOW BY w WINDOW_PROPERTIES WINDOW_FRAMING
WINDOW_PROPERTIES := TYPE ( SLIDING | TUMBLING ) ( PARTITION BY p )? ORDER BY o
WINDOW_FRAMING := ( LABEL WINDOW_PREDICATES )? START WINDOW_BOUNDARY ( ONLY )? END WINDOW_BOUNDARY
WINDOW_PREDICATES := WINDOW_VARS ( WINDOW_CONDITION )?
WINDOW_BOUNDARY := ( UNBOUNDED | WINDOW_PREDICATES )
WINDOW_VARS := ( AS currentItem )? ( AT positionalVar )? ( PREV AS previousItem )? ( NEXT AS nextItem )?
WINDOW_CONDITION := WHEN condition

```

**Fig. 2** The proposed WINDOW BY Clause

queries. In doing so, we first describe the desiderata for a new Clause by combining the advantages of SQL OVER and XQuery while avoiding their disadvantages. We then examine how SQL++ can easily support these desiderata followed by the semantics of the WindowBy Clause.

### 3.1 SQL OVER and XQuery windows

The SQL OVER Clause defines windows using a required *window function* (or aggregate function) and three optional parts: *partitioning*, *ordering* and *framing* [26, 43]. Partitioning comes first and groups the data so as to create different sequences; one per grouping key. Ordering these creates the ordered sequences. While optional, ordering is crucial for windowing; without ordering, SQL window queries could provide different results every time they run. Finally, framing is used to build windows on top of each ordered sequence. Examples of the required window function include built-in such as AVG or user-defined aggregates, as well as built-in functions for windows such as ROW\_NUMBER). The OVER Clause can be used in the SELECT or ORDER BY Clauses. If the OVER Clause appears in SELECT, the result of the window function is appended (as an attribute) to the output tuple. If OVER is used within ORDER BY, the window function is first computed and then used to determine the order of the result.

An advantage of SQL OVER is that the use of *ordering* and *partitioning* allow the user to create ordered sequences as needed. The result of a window is appended to a single tuple in the ordered sequence. The advantage of *framing* is that it allows the user to pick window boundaries; these boundaries can be *before*, *after* or *include* the tuple that will append this window's result. Consider as an example the window in Fig. 1a, which depicts a sliding window of size 3 tuples using the OVER Clause. To do so, in the SELECT, we used AVG as our window function followed by the OVER keyword. The parameters for OVER are followed by ORDER BY to sort by date and then the ROWS keyword to specify a framing. In this specific case, the framing is between CURRENT ROW and 2 FOLLOWING (3 tuples).

Note that OVER requires that every single tuple in the sequence creates a window; as a result, OVER is effectively limited to expressing sliding windows. In [27], tumbling windows are modeled as a special case of sliding windows where the RANGE of a window is equal to the SLIDE of that win-

dow (explained in Sect. 2). However, in the case of OVER, the SLIDE is always 1 tuple; thus, a tumbling window can occur only for a window size of length 1 tuple using the ROWS keyword, assuming there are no duplicates in the ordering attribute.

XQuery windowing, as part of the XQuery 3.0 Standard, consists of three required parts: a *window type* (tumbling or sliding), *variables* (start and end), and *predicates* (start and end). Note that XQuery uses the order of the binding sequence for its windowing hence no explicit ordering is needed. *Variable access* offers a major advantage to XQuery windowing as it allows the user to create windows on more than just the ordering attribute of the sequence. The variables enable access to the first and last tuple of the window and their positional variable (their numerical position in the ordered sequence). For each of the two tuples, XQuery also provides access to the tuple preceding or following that tuple. These tuple variables and positional variables can be referred to in the predicate (Boolean expression) used to start or end a window.

Moreover, XQuery allows the user to *specify the window type* (tumbling or sliding). Another advantage of XQuery is that it provides *access to the whole window* and its tuples through a user defined variable. Since data can be nested, this allows the user to return more than just an aggregation result over this window if needed. Nevertheless, in XQuery a window result always appends the tuple that started this window; this is a limitation since it does not allow a window to consider only tuples before (or after) its boundaries. Consider as an example the window in Fig. 1b, which depicts a sliding window of size 3 tuples using XQuery. To do so, we first use the keyword SLIDING WINDOW followed by the window result variable w. The start predicates contain the starting tuple of the window s and its position sp. The start predicate condition is to always start a window after the WHEN keyword. The end predicates include its position ep and the end predicate condition is when  $ep - sp = 2$ ; which creates a window of size 3. This approach assumes that the dataset Ghcn is already ordered by time, and the user is also allowed to use the window variable w and the predicate variables s, sp and ep in forming the result.

To summarize, our proposed SQL++ WindowBy Clause should have the following *desiderata*: (1) ability to create the desired ordered sub-sequences; (2) ability to pick the window type; (3) use framing to define which tuples should



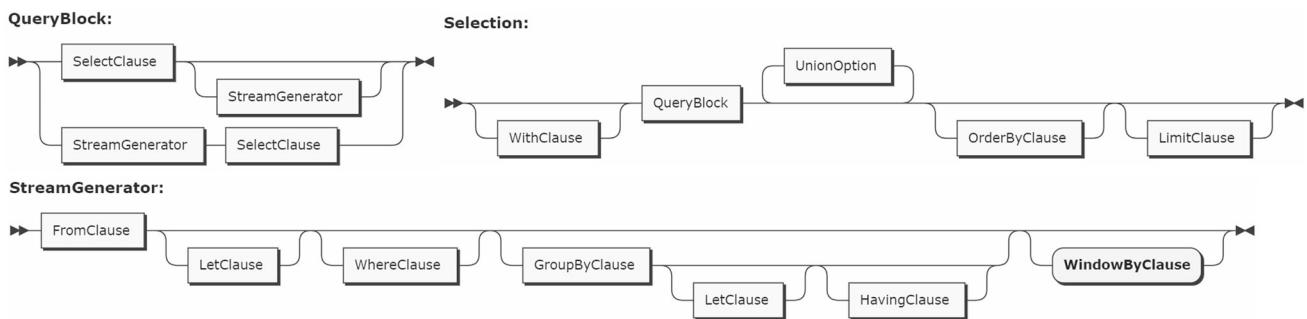


Fig. 3 SQL++ query block semantics

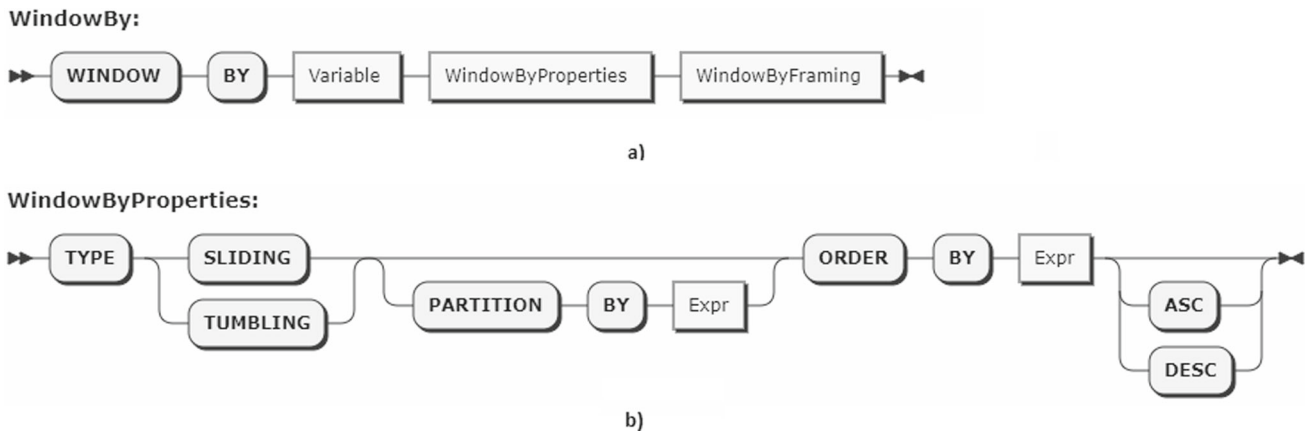


Fig. 4 a WindowBy Definition and b Properties

be included in the window; (4) provide ability to provide start/end windows using predicates; (5) access to a window's tuples through user defined variables. Here, desiderata (1) creates the input sequence, while (2) defines the window type. Desiderata (3) and (4) create the window boundaries and its position in the sequence, while (5) computes and formats the result.

A sketch of the grammar of our proposed **WindowBy** Clause appears in Fig. 2 (its semantics will be detailed in Sect. 3.3). Variables such as  $w$ ,  $p$  and  $o$  are user defined and the user can change them. Its major (required) parts are *window\_properties* (addressing desiderata (1) and (2)) and *window\_framing* (covers (3) and (4)). Inspired by XQuery, the window properties allow the user to choose the window type (sliding or tumbling), and users can create ordered sequences as needed (inspired by OVER). The framing includes predicates (as in XQuery) and the ability to append the window result to any tuple in the sequence (as in OVER). Finally, the variable  $w$  enables accessing the tuples of a window, thus supporting desiderata (5) (as will further become clear through query examples). Combining the above advantages in the proposed **WindowBY** Clause will enable users to write a variety of useful and complex windows in an intuitive way (see Sect. 4).

### 3.2 The advantages of SQL++

Having identified the desiderata, we proceed to consider what language tools are needed for supporting them. An important difference between how SQL and XQuery handle windows is that XQuery uses a *Clause* to create windows while the SQL OVER approach, despite being named a Clause, is actually an *expression*. An expression is a language fragment that can be evaluated to return a single value [12]. Examples are boolean expressions, aggregate functions, numeric expressions etc. In SQL++, a query is made of several Clauses, such as SELECT, FROM, or GROUP BY, and each Clause does a specific task by consuming input tuples and producing output tuples. Clauses like FROM, WHERE, GROUP BY and HAVING are characterized as *generators* since they result in a stream of tuples. Clauses like ORDER BY, LIMIT and OFFSET are instead *output modifiers* since they do not create any new tuples; but can only order, limit or omit some of the tuples in the result stream. SELECT is special in that it constructs an output object for each tuple in its input.

Since OVER is an expression, it returns a single value (window result) per tuple; further, in SQL it can occur either in the SELECT or in the ORDER BY Clause. Being in either of these Clauses implies that the number of results tuples in

the output stream is not changed by `OVER`; which simply attaches a window result to each tuple and in the case of `ORDER BY`, it orders the tuples using their window result. While this is fine for SQL windowing, since every tuple creates a window (i.e. each tuple is expanded with the window result), it will not suffice for `WindowBy`. As in XQuery, the `WindowBy` Clause should allow the user pick which tuples will create a window based on a predicate. An expression-based approach like `OVER` will not work; instead, a true Clause is needed for `WindowBy`.

Moreover, for each window, `WindowBy` should allow the user to access, using variables, the first and last tuple of the window as well as their positional variables, preceding tuple, and following tuple. Such variables are bounded as each window is considered. In SQL, the variables in scope (available attributes) are defined by the `FROM` Clause and stay in scope throughout the query block. Other Clauses in SQL cannot add new variables in scope; they can only place restrictions on which variables can be used in `SELECT` (e.g. as does the `GROUP BY` Clause). Hence, it would not be easy to add a new Clause to SQL Standard with our required desiderata.

We proceed by describing how the somewhat richer structure and semantics of SQL++ [12, 30] can help us avoid these shortcomings and can thus support all the desiderata of our `WindowBy` Clause. One significant difference from SQL is that SQL++ offers the ability to create or remove variables within each Clause as needed; i.e., variable creation is not limited to the `FROM` Clause.

Consider for example, `GROUP BY`. When used in SQL it restricts `SELECT` to use only the `GROUP BY` keys or aggregates. When used in SQL++, `GROUP BY` can also add new variables into the scope through the use of `GROUP AS`. As in SQL the `GROUP BY` Clause “hides” the original tuples in each group by exposing only the grouping keys and aggregation functions on the non-grouping fields; however its `GROUP AS` extension (used only within `GROUP BY`) makes the original tuples in the group visible to subsequent Clauses by creating a new variable. This allows a SQL++ query to generate output data both for the group as a whole and for the individual objects inside the group [12]. This is possible because SQL++ supports a nested data model.

As a result, SQL++ offers important advantages to support our `WindowBy` Clause as it naturally supports the variables needed to access various parts of the window and its contents. Finally note that the existing `GROUP BY` Clause even within SQL++ is not sufficient to support all of our windowing desiderata.

For example, desiderata (1) is not possible since `ORDER BY` happens after `GROUP BY`. Similarly, desiderata (2) fails since `GROUP BY` does not allow data to overlap between groups (and thus no sliding windows). Next is a detailed discussion of the `WindowBy` Clause semantics.

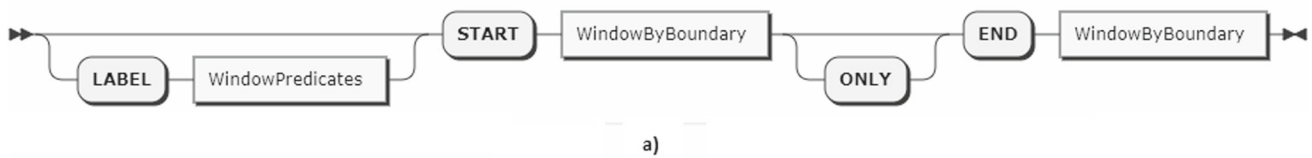
### 3.3 WindowBy Clause semantics

Figure 3 shows the top level SQL++ grammar extended to include our proposed `WindowBy` Clause. As `WindowBy` is an optional standalone Clause, the proposed extension does not affect/modify the results of any existing queries in SQL++. As can be seen from the figure, the `WindowBy` Clause is placed after `GROUP BY`. We designed `WindowBy` as a Clause to separate it from the `SELECT` Clause, which in our view reduces the complexity of writing window queries. Another important characteristic is that we allow the user to create, for each window  $w$ , predicate variables that refer to specific tuples/values in that window (e.g., starting tuple, its positional value etc.). Note that the result for each created window  $w$  is a complex tuple that contains all of the predicate variables for  $w$  as well as its data tuples as an ordered array, as can be seen in Fig. 10, box 3. The predicate variables and the window array for each  $w$  can then be used inside the `SELECT` clause so that the user can easily form the desired query result. Note that this requires a nested structure which cannot be expressed in traditional SQL; SQL++ has the advantage that it allows for nesting and makes it easy to create the desired window tuples. This allows the user to create windows over groups (if any). Note that this is similar to SQL where `OVER` is placed inside the `SELECT` or `ORDER BY` and thus applies to the results coming after `GROUP BY`.

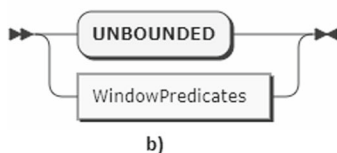
As shown, the `WindowBy` Clause starts with the `WINDOW BY` reserved keyword followed by an identifier, the window properties and framing (Fig. 4a). The identifier is a user defined variable that can be referenced in later Clauses to access the window results; it is referred to as result variable below.

According to the SQL++ grammar (Fig. 3), `ORDER BY` occurs after the `QueryBlock` and hence after the `WindowBy` Clause. Thus, `WindowBy` needs to create its own ordered sequences and this is done through the properties section (Fig. 4b). In addition to `ORDER BY`, properties specify the `TYPE` of the window to be created (currently only sliding and tumbling are supported). There is also an optional partition section where the user can partition the data before ordering, based on a user specified expression.

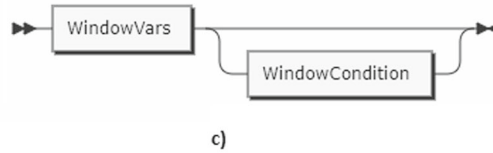
The framing section is based on XQuery’s window Clause and it uses a predicate-based approach to create the window boundaries. There are three separate parts in framing (see Fig. 5a): the optional `LABEL` (to be discussed later), `START` and `END`. The `START` refers to the first tuple in the window, while `END` refers to the last one. They are each defined by the `WindowByBoundary` (Fig. 5b). Such a boundary can either be by the keyword “`UNBOUNDED`” or a window predicate (Fig. 5c). “`UNBOUNDED`” allows the user to create windows that extend to the start or the end of the sequence of the partition. Like XQuery, the window boundary contains two parts: variables and an optional predicate (condition).

**WindowByFraming:**

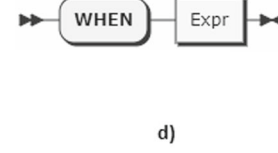
a)

**WindowByBoundary:**

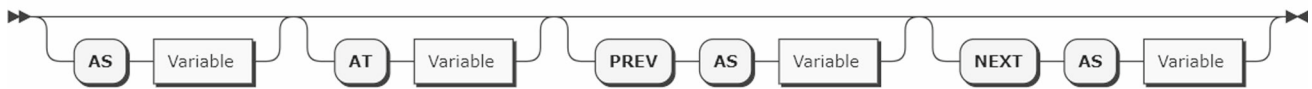
b)

**WindowPredicates:**

c)

**WindowCondition:**

d)

**Fig. 5** a WindowBy Framing, b Boundaries, c Predicates and d Conditions**WindowVars:****Fig. 6** Variables allowed in the WindowBy Predicates

The window variables (Fig. 6) for a **START** boundary correspond to the first tuple in the window (referred to by **AS**), its position in the sequence (**AT**; also termed as the positional variable), the tuple preceding the first tuple (**PREV AS**) and the tuple following the first tuple (**NEXT AS**). Similarly, for an **END** boundary these variables are relative to the last tuple of the window.

The optional window predicate (Fig. 5d), specified with the keyword **WHEN**, can use the above variables to form boolean conditions that start or end a window at a specific tuple. This removes the SQL **OVER** limitation that each tuple must start its own window. Given a finite ordered sequence, when the last tuple of the sequence is reached, there may be many windows whose **END** condition has not been met. By default, the **WindowBy** Clause will produce results for all windows whether they are ‘closed’ (these are windows for which a tuple was processed that matched the condition to **END** the window) or are still ‘open’ after the sequence is processed (no tuple in the sequence matched the **END** condition for this window). As in XQuery, when the **ONLY** keyword is used, the **WindowBy** Clause returns results only for the closed windows. After **WindowBy**, all variables preceding the **WindowBy** Clause go out of scope. Only variables introduced by **WindowBy** are available after it.

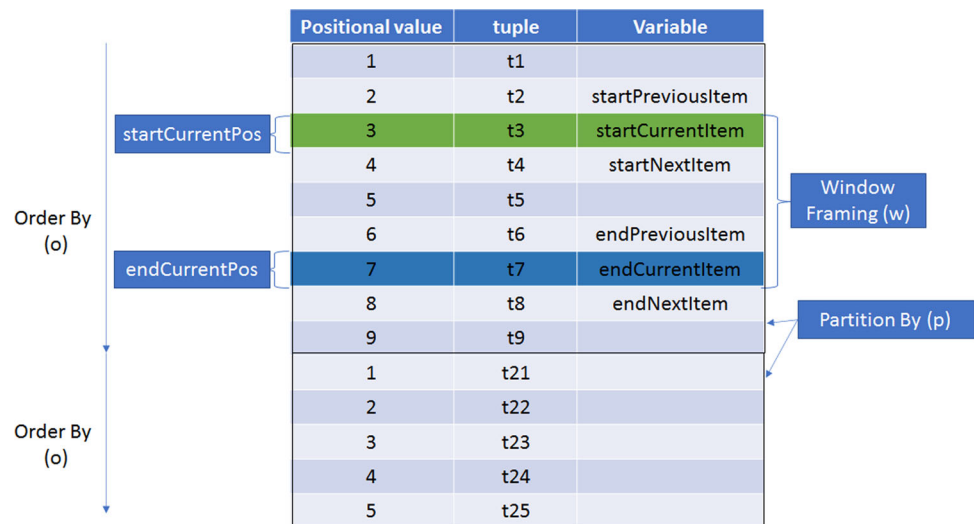
The concepts of partitioning, ordering and framing of the **WindowBy** Clause are visualized in Fig. 7 (following a similar approach from [26]). **PARTITION BY** separates the tuples into distinct groups, according to some attribute  $p$  in the Figure. Each partition is ordered by **ORDER BY** (using some

attribute  $o$ ) to create ordered sequences. The figure shows two ordered partitions with a window  $w$  created on one of these ordered partitions ( $w$  refers to the entire window framing). A user can create variables based on the start or the end tuple of a window, their positional values and their previous and next tuples. Window  $w$  starts from tuple  $t_3$  (denoted with variable **startCurrentItem** in the Figure) and ends at  $t_7$  (**endCurrentItem**). The positional variable for  $t_3$  (**startCurrentPos**) is 3 and for  $t_7$  (**endCurrentPos**) is 7. Variable **startPreviousItem** and **startNextItem** denote the previous and the next tuple of the start tuple. Similarly for **endPreviousItem** and **endNextItem** for the end tuple. A window consists of all tuples between the **startCurrentItem** and the **endCurrentItem** (within the same partition); in this example the window will consider all tuples between  $t_3$  and  $t_7$ .

One limitation that the XQuery predicate-based approach has in framing is that a window’s result will always be appended to the tuple that started the window (first tuple in the window). To remove this limitation, the optional **LABEL** part was added in **WindowBy** (Fig. 5a). The label is used to denote the tuple that will include the appended the result; doing so allows the window boundaries to be before, after, or to include this tuple. If **LABEL** is not used, the **WindowBy** Clause will default to the label being the first tuple in the window (like XQuery). The variables in the **LABEL** window predicate reference the tuple that will have the result. These variables can be referenced later in the query.

Note that **LABEL** allows the **WindowBy** Clause to support queries that are possible in **OVER** but not in XQuery.

**Fig. 7** WindowBy partitioning, ordering and framing



Consider a generic SQL OVER query with framing. This framing can be separated into BETWEEN X and Y. X and Y can be either UNBOUNDED PRECEDING/FOLLOWING, CURRENT ROW, or  $n$  PRECEDING/FOLLOWING where  $n$  is a positive integer. If X is the current ROW, then in WindowBy the label refers to the start tuple. If instead the current row is in Y, then in WindowBy the label refers to the end tuple. As discussed earlier, in the WindowBy boundaries, the UNBOUNDED choice is allowed for both start and end. With access to the label variables and the start variables, representing in WindowBy a start condition in X for  $n$  PRECEDING or  $n$  FOLLOWING is simple to do by comparing the label variables with the start variables. Consider for example a ROW framing query where X is PRECEDING 5 in SQL OVER. Let's assume that *labelCurrentPos* (*startCurrentPos*) corresponds to the label (start) positional variable (respectively) in WindowBy. The above OVER window boundary can be represented by having a WindowBy start condition of *labelCurrentPos* - *startCurrentPos* = 5, which implies that the start of the window is 5 tuples before the label. Likewise, the same can be represented for the case of Y of FOLLOWING 5 by having the end condition be *endCurrentPos* - *labelCurrentPos* = 5.

### 3.4 WindowBy bindings

We now describe the nature of WindowBy variable bindings through an SQL++ query example. All window queries in this paper will be using a dataset from the Global Historical Climatology Network (GHCN) [19]. It contains measurements of climate data collected at stations across the world. In particular we will use the AsterixDB DDL that appears in Fig. 8; which contains a subset of all the attributes in the original GHCN table. Each GHCN tuple has four attributes: station, date, val, and dataType. Station corresponds to the

```
CREATE TYPE GhcnItem AS {
  station: int,
  date: datetime,
  val: double,
  dataType: string
};
CREATE DATASET Ghcn(GhcnItem) primary key station,
  date, dataType;
```

**Fig. 8** DDL for the GHCN dataset

```
SELECT
  s.ghcn.station AS station,
  s.ghcn.date AS start_time,
  ARRAY_AVG(( SELECT VALUE ghcn.val
               FROM w )) AS average
FROM ghcn
WHERE dataType = "TMIN"
WINDOW BY w
  TYPE SLIDING
  PARTITION BY ghcn.station
  ORDER BY ghcn.date
  START AS s AT sp WHEN s.ghcn.val < 25
  END AT ep WHEN ep - sp = 2
```

**Fig. 9** Example query for bindings

id of the station that took the measurement, val refers to the reported measurement value, and dataType describes the type of the collected value. In our examples the measurement value "TMIN" corresponds to the minimum temperature collected by a station on the specific date. Other types of measurements that a station provides are maximum temperature, precipitation, snow depth, etc.

Consider the query: "When the value of TMIN in a specific station is less than 25 degrees, find the average value of TMIN over this measurement and the two following TMIN measurements of that station". This is a sliding window (i.e., overlap between windows is allowed) with a window length of three tuples (including the starting tuple). The query using our WindowBy Clause appears in Fig. 9. There are three predicate variables declared: *startCurrentItem* which corresponds



to the first tuple of the window, *startCurrentPos* which is the positional variable of this current tuple and *endCurrentPos* for the positional variable of the window's last tuple. There is also the result variable *w*. `ARRAY_AVG` is a SQL++ aggregate function that computes the average for an array of tuples while ignoring missing or null values.

Figure 10 shows the Clauses' output bindings for this WindowBy query. The FROM Clause outputs a bag of GHCN tuples (indicated as number 1 in the Figure) that becomes input to the WHERE Clause. For simplicity, we use integers to represent dates instead of the actual datetime values. Afterwards, the data is filtered to keep tuples that satisfy the expression in the WHERE Clause (`dataType = "TMIN"`). The output of WHERE becomes the input of the WindowBy Clause (number 2).

WindowBy will first partition the data by station. It will then order each partition by date (to create the ordered sequence) and then will create a (sliding) window for any input tuple that satisfies the starting condition (number 3). By construction, each window created is represented by a tuple that contains the following attributes: one attribute for each variable declared and one attribute to capture the result variable (providing the tuples of the window created). These variable's values are bound as we sequentially process the data. When a tuple satisfies the start condition (the TMIN val is less than 25), its associated start variables *startCurrentItem* and *startCurrentPos* are bound to this tuple and its sequence position, respectively. When a tuple satisfies the ending condition, its associated end variable *endCurrentPos* is bound to this tuple's sequence position. If the end of the sequence is reached, then any *endCurrentPos* variables are set to MISSING; MISSING is also appended to the result variable *w* of any unclosed window as shown in the figure.

In the example query in Fig. 10, WindowBy creates three windows, one starting at the tuple with the date 1 and val 20 for station 1, one starting at the tuple with the date 4 and val 23 for station 1, and one starting at date 1 and val 22 for station 2. In this example each window is represented by a tuple with the following four attributes: *s* (the first GHCN tuple of the window), *sp* (its position in the sequence), *ep* (the position of the last GHCN tuple in the window), and *w* (the result for this window that contains all the window's GHCN tuples).

The first window in this example contains 3 GHCN tuples in *w*, those with dates (1, 2 and 4). Note that the tuple with station 1 and date 3 was removed by the WHERE Clause which causes the tuple with station 1 and tuple 4 to be the third tuple in this sequence. The case where a window is created but its end condition is never satisfied, is denoted in the output by using the keyword 'MISSING'. This happens in the second window created for station 1 at date 4; note that the value of *ep* is MISSING. This window contains two GHCN tuples (with date 4 and 5) and since it never ended,

the value MISSING is also added). The same occurs for the third window, created for station 2.

The window tuples resulting from the WindowBy Clause will then become input to the SELECT Clause which will then construct new tuples according to the user's desired output (number 4). In particular, SELECT in the example outputs the station id of each window, the start time of the first tuple in the window and the average val of TMIN for over the tuples in the window. Note that attributes associated with variables introduced in the WindowBy (*s* and *w*) were used in the SELECT.

It is important to note that the above description provides a logical representation of how WindowBy computes the windows. Most windows (like the example shown) typically compute aggregates on the window tuples. As a result, the individual tuples within a window do not need to be materialized and carried forward (as in step 3; since step 4 only needs an average). A simple optimization will push the aggregation into the WindowBy processing thus avoiding actually materializing all the window tuples.

## 4 WindowBy query examples

To showcase the new WindowBy Clause, we proceed with query examples for various common windows. Each query uses the dataset described in Fig. 8. Queries differ in their type (sliding or tumbling) and framing (the window boundaries). For simplicity all query examples partition by the station and order by date, while the output of each window contains the station, starting time (time of the first tuple in the window), and the average TMIN value over the window for the specific station. We express each query using both in SQL++ OVER and WindowBy. For cases where SQL++ OVER cannot express the query (e.g., windows that are not based on the ordering attribute or tumbling windows) we write the query using an SQL++ correlated nested query approach.

### 4.1 Sliding windows

The length of a window, whether it is sliding or tumbling, can be constrained in two ways: either with fixed duration (**ordering attribute range**) or with a fixed number of tuples. Such windows accept tuples as long as the duration has not been met (this assumes that the ordering attribute is time-based) or as long as the number of tuples in the window is less than the threshold. We call both such windows 'bounded'. If instead the ending condition is based on a predicate involving some other attribute of a tuple (i.e., not the ordering attribute), the window will continue to accept tuples as long as the ending condition is not met. We call such windows 'unbounded' to indicate that there is no preset upper bound on the window's

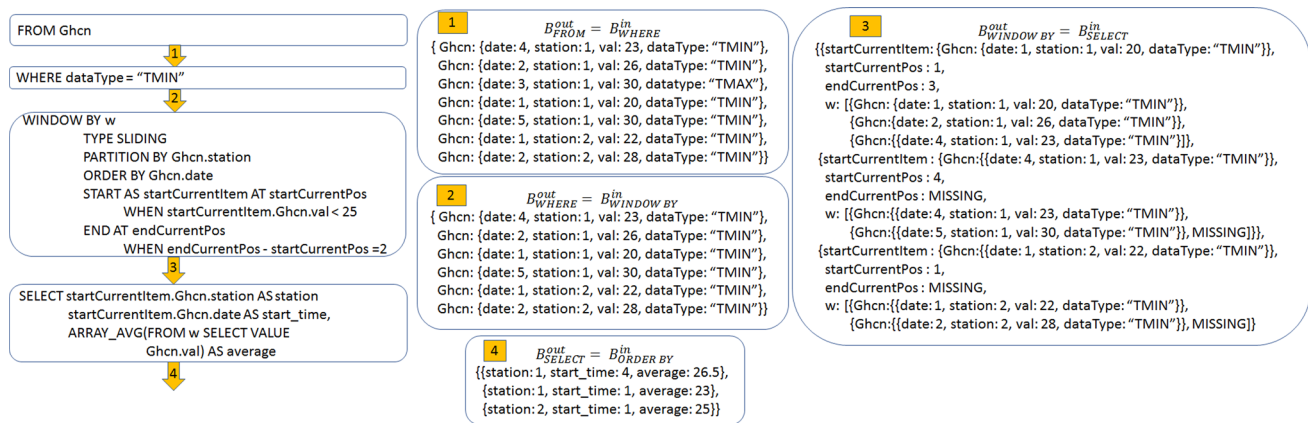


Fig. 10 The outputs of each query Clause; numbers indicate where the output was produced

length in terms of duration or number of tuples. Below we present sliding window examples for each case.

#### 4.1.1 Bounded sliding windows: duration

Consider creating a sliding window with a duration of 1 week (i.e., given a station, find the average TMIN value starting from a given tuple and ending 1 week later than the date of this tuple). This query can be written using OVER as in Fig. 11a; it partitions by station, orders by date and applies RANGE for the framing.

The same query using the WindowBy Clause appears in Fig. 11b; it creates two variables, *startCurrentItem* for the first tuple in the window, and *endNextItem* for the tuple following the last tuple of the window (using END NEXT AS). To create the sliding window of size 1 week, its ending condition ensures that the difference between the date of *startCurrentItem* and the date of *endNextItem* is within 1 week. Note that *endNextItem* is the earliest tuple in the sequence where the date is **not** within 1 week. As such, *endNextItem* should not be in the window, but it is still required for expressing the window boundaries. By using the tuple before or after the first (last) tuple of the window, it is possible to make the window boundary inclusive or exclusive of a particular condition.

#### 4.1.2 Bounded sliding windows: number of tuples

Next consider creating a sliding window of size 20 tuples (i.e., given a station, find the average TMIN value starting from a given tuple and including the next 19 tuples of the sequence). This query written using OVER appears in Fig. 12a; it partitions by station, orders by date and applies ROW for the framing.

The same query using the WindowBy Clause appears in Fig. 12b; it creates three variables, *startCurrentItem* for the first tuple in the window, *startCurrentPos* for its position in

the sequence and *endCurrentPos* for the position of the last tuple in the window. To create this sliding window of size 20 tuples, its ending condition ensures that the difference between *startCurrentPos* and *endCurrentPos* is exactly 19. The *startCurrentItem* variable allows SELECT to access both the station and date from the first tuple in the window.

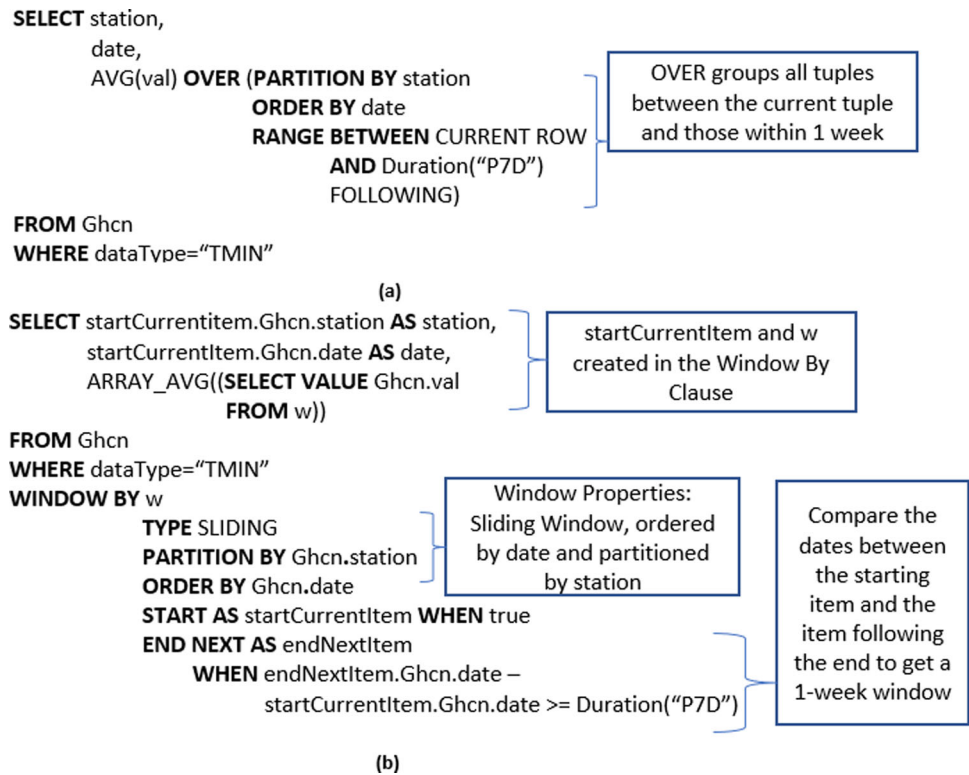
An interesting observation about the two example windows we have discussed is that each tuple in the sequence creates a window. Hence, the START condition is always satisfied (WHEN true) in the WindowBy Clause. Note that OVER does not need such a condition since every tuple starts a window.

#### 4.1.3 Unbounded sliding windows

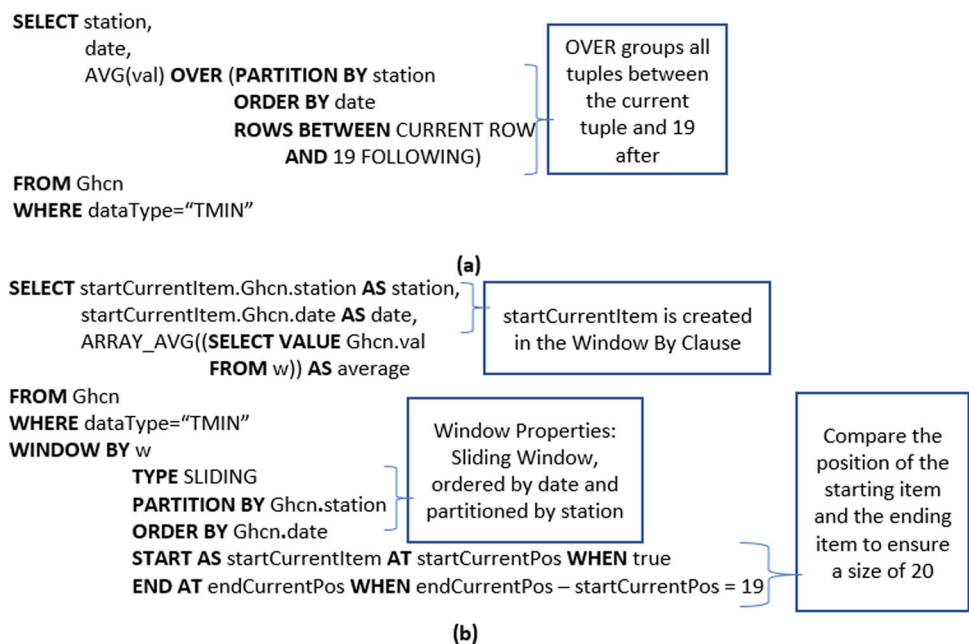
Consider a sliding window that stays open while the TMIN value of the tuple remains less than 25 (i.e., given a station, the window starts from a given tuple where the TMIN value is less than 25 and includes all the station's subsequent tuples until a tuple is reached—in sequence order—where the TMIN value is greater than or equal to 25).

This query cannot be written using OVER since the window is not based on the ordering attribute. Figure 13a provides a solution for this query in SQL++ using a nested query approach, by finding the tuples that start and end a window and creating a window using this information. Using WITH, three temporary tables are created to help find the window boundaries. The first table *wStart* finds the tuples that can start a window (i.e., their TMIN value is less than 25). The second table *wEnd* finds the tuples that can end a window (the TMIN value is greater than or equal to 25 or the last tuple in the sequence for that station). To find the last tuple in the sequence, we use the LET Clause to find the last date for that station. The third table *wBoundaries* finds pairs of actual window boundaries; in particular it pairs a starting tuple from *wStart* with the earliest tuple from *wEnd* with the same station which comes after it in sequence order (and it is from the

**Fig. 11** Example of a sliding window with 1 week duration using: **a** OVER and **b** WindowBy



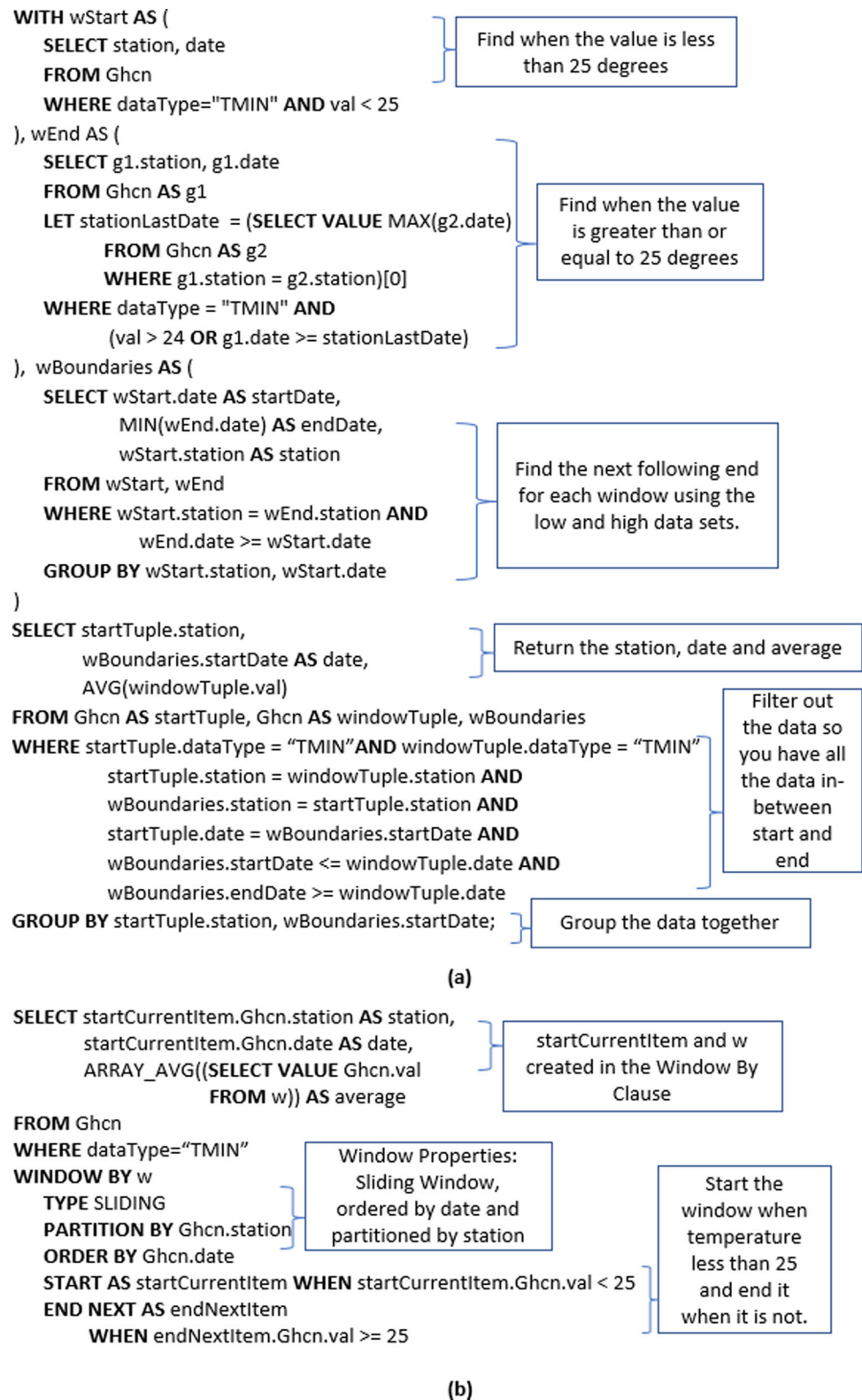
**Fig. 12** Example of a sliding window with fixed length of 20 tuples using: **a** OVER and **b** WindowBy



same station). Note that since these are sliding windows, a given tuple from *wEnd* can be the ending tuple for many windows (and thus it is paired with all tuples from *wStart* that come before it). Using two copies of GHCN and the table *wBoundaries*, a conceptual nested loop finds all tuples that are in each window. Grouping by the station and the time of the first tuple in each window (*startTuple.date*), the average TMIN is calculated for each window.

The same query using the WindowBy Clause appears in Fig. 13b. It creates two variables, *startCurrentItem* for the first tuple in the window, and *endNextItem* for the tuple following the last tuple of the window. To create this sliding window is simple: its starting condition checks if the TMIN value of *startCurrentItem* is less than 25 and the ending condition checks if the TMIN value of *endNextItem* is greater than or equal to 25. An interesting observation about this

**Fig. 13** Example of an unbounded sliding window using: **a** Nested-loop SQL++ approach and **b** WindowBy





example is that not every tuple creates a window; only tuples when the starting condition (WHEN `startCurrentItem.val < 25`) is satisfied start a window.

#### 4.1.4 Discussion

The preceding sliding window examples have revealed interesting differences between the proposed `WindowBy` Clause and SQL++'s current windowing capabilities for sliding windows. While the `OVER` Clause is more compact than `WindowBy` for sliding windows, it cannot express some predicate-based windows (those not based on the ordering attribute). Using nested loops to express these sliding windows can involve many subqueries which is very challenging for optimization. Instead, `WindowBy` is much more intuitive as the user only needs to input the window properties and the window boundaries. The properties allows the user to partition and order the sequences before processing. The window boundaries are set by the predicates and the user is also allowed to set variables for tuples and their position. Our `WindowBy` Clause also allows the user to flexibly determine which tuples create windows through the start predicate, which is not possible in `OVER`.

## 4.2 Tumbling windows

Different from sliding, tumbling windows do not allow overlap; while a window is open, no other windows can be created (i.e., a tuple can belong to at most one window). Note that `OVER` produces a window for each tuple in the input sequence. As a result, `OVER` will create sliding windows unless each window contains exactly 1 tuple (the starting tuple of the window), which is a degenerate case of tumbling windows. Below we examine two tumbling window examples that are the 'tumbling' version of sliding windows discussed in the previous section. In particular, an unbounded window and a window with fixed duration.

### 4.2.1 Unbounded tumbling windows

Consider a tumbling window that stays open while the TMIN value of the tuple is less than 25 (i.e., given a station, the window starts from a given tuple where the TMIN value is less than 25 and includes all the station's subsequent tuples until a tuple is reached—in sequence order—where the TMIN value is greater than or equal to 25). Figure 14a shows the query written using the `WindowBy` Clause. When compared with its sliding version (Sect. 4.1.3, Fig. 13b), the only difference is that keyword `TUMBLING` replaces `SLIDING` for the window type.

As with the sliding case, this query cannot be written using `OVER` (since the window boundaries are based on a non-ordering attribute). We show how this query can be written

in SQL++ using a nested loop approach (Fig. 14b). There are five temporary tables: *wStart*, *wEnd* and *wBoundaries* which perform similar tasks as in the sliding case, as well as *GhcNPos* and *currentPrev*. In particular, *GhcNPos* first partitions the data by station and then orders by time to assign each tuple its positional variable within the sequence of tuples of their respective station. This table will help identify the exact tuples that start or end a window such that there is no overlap between windows. This is done by comparing a tuple's TMIN value with the tuple preceding it coming from the same station.

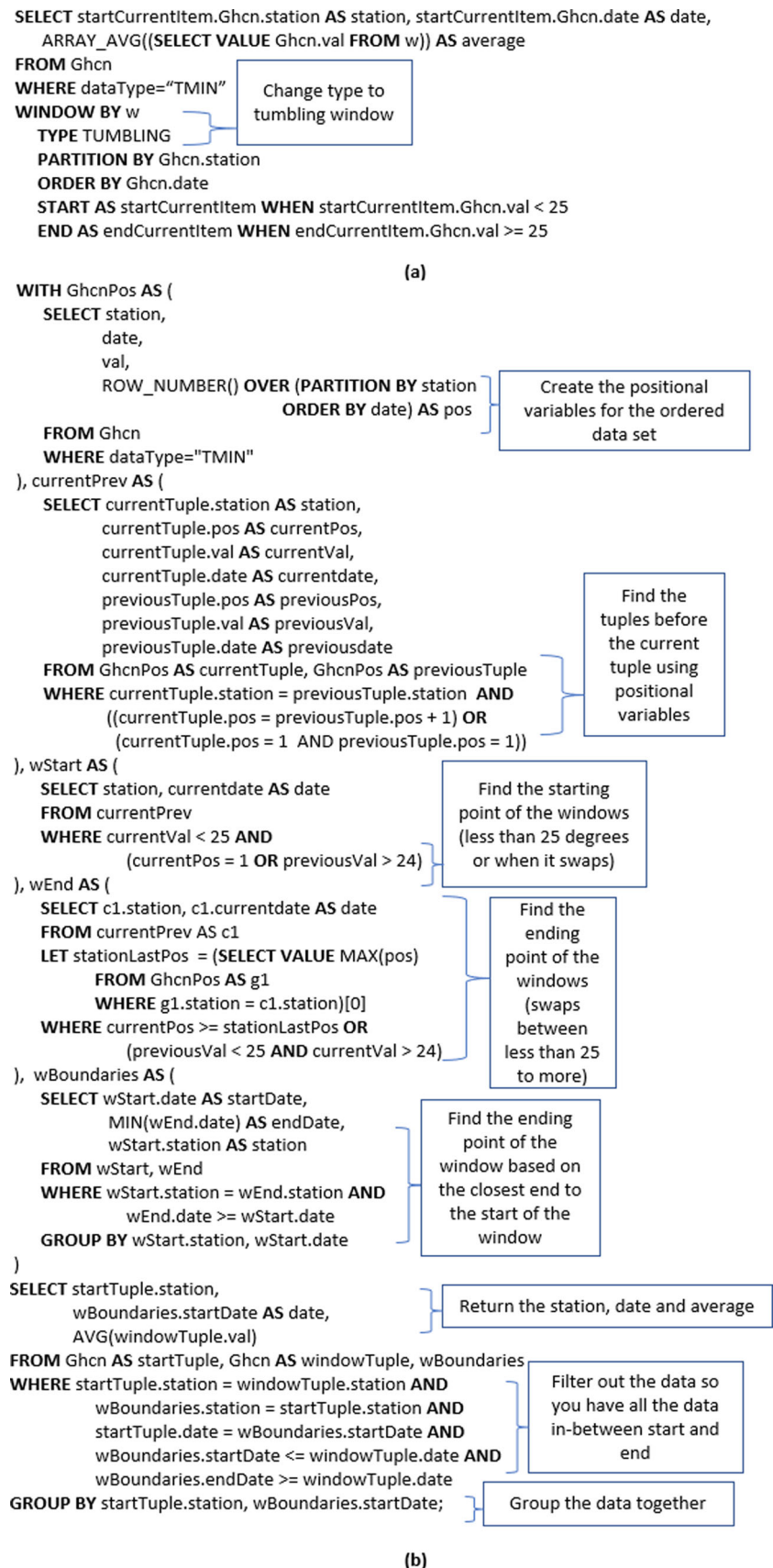
The *CurrentPrevious* table creates pairs that contain a specific tuple in the sequence (*currentTuple*) and the tuple preceding it (*previousTuple*) for the same station. The table *wStart* still finds tuples that start a window; in this example, there are two cases in which a tuple should start a window: (1) TMIN value less than 25, and (2) the *currentPos* is 1 (first tuple in the sequence) or the previous TMIN value is greater than 24. Table *wEnd* finds tuples that end a window; there are two cases for a tuple to end a window: (1) the tuple is the last tuple in the sequence, or, (2) the previous TMIN value (*previousVal*) is less than 25 and the current TMIN value (*previousVal*) is greater than 24. As before, the *wBoundaries* table finds pairs of window boundaries. The rest of the query uses a nested loop approach and `GROUP BY` to find all the tuples in each tumbling window.

### 4.2.2 Bounded tumbling windows: duration

Consider creating a tumbling window with a duration of 1 week (i.e., find the average TMIN value starting from a given tuple and ending 1 week later than the date of this tuple). Since it is a tumbling window, the next window starts in the next tuple after the previous window ended. This query written using the `WindowBy` Clause appears in Fig. 16a; it is similar to the sliding window in Fig. 11b with the only difference being that the keyword `TUMBLING` is used for the window type.

This query cannot be written in SQL without recursion because there is no way to identify the position of the starting or ending tuples. As recursion is not yet supported in SQL++, we write the query in PostgreSQL. A nested loop is first used to create all possible sliding windows between a tuple and all tuples (from the same station) within 1 week from the tuple that starts the window. Grouping by the station and the starting date, creates the boundaries and desired output for every sliding window (station, starting time, ending time and average TMIN value). The window boundaries for each sliding window are stored in a temporary table (named *w*). The next step is to identify the tumbling windows from all possible sliding windows. This is performed by using a recursive CTE (common table expression) to calculate all non-overlapping windows (denoted by 'WITH RECURSIVE

**Fig. 14** Example of an unbounded tumbling window using: **a** WindowBy, and **b** Nested-loop SQL++ approach



tumble' in Fig. 16b). CTE *tumble* produces tuples with three attributes: starting time *s*, ending time *e*, and *station*. A recursive CTE has two parts: a base query and a recursive query; when separated by UNION ALL the output contains the results of all iterations. For CTE *tumble*, the base query (first iteration) finds the boundaries of the earliest window in each station. Then the recursive query uses the output of the previous iteration and finds the next tumbling window in the sequence. This continues until there are no more tumbling windows are created.

Without using recursion, this query can also be expressed in SQL using the MATCH\_RECOGNIZE Clause; see Fig. 16c (note that SQL++ currently does not support MATCH\_RECOGNIZE). MATCH\_RECOGNIZE is made up of two required parts that define the pattern (PATTERN and DEFINE) and five optional parts for the pattern properties and the query output (PARTITION, ORDER, MEASURES, ROW PER MATCH and AFTER MATCH). For the example tumbling query, the required parts allows the user to define the pattern for the window boundaries. In this case, the pattern is similar to our WindowBy starting (always start a window) and ending condition (next tuple is over 1 week from the starting tuple). PARTITION and ORDER are used to create the ordered sequence. AFTER MATCH allows the user to choose the window type. For example, by using SKIP TO LAST DOWN the pattern matching continues from the tuple after the end of the previous window (and thus create tumbling windows). Using one ROW PER MATCH limits the output to produce one result per window. Finally MEASURE allows to format the output.

#### 4.2.3 Bounded tumbling windows: number of tuples

Next consider creating a tumbling window of size 20 tuples (e.g., given a station, find the average TMIN value starting from a given tuple and including the next 19 tuples of the sequence). This query can be written using OVER to find all sliding windows and then filtered to get just the tumbling windows in Fig. 15b.

The WindowBy version of this query can be found in Fig. 15a and it is the same query as Fig. 12b except that the keyword SLIDING has been changed to TUMBLING for the window type.

#### 4.2.4 Discussion

It is worth noting the simplicity in writing both tumbling query examples using the WindowBy Clause: since their window predicates remain the same as their sliding versions, a simple keyword change enables the previous sliding window query to now find tumbling windows. These tumbling windows cannot be expressed by OVER unless it is a fixed tuple

sized window; using MATCH\_RECOGNIZE or a nested-loop approach results in much more complex query writing.

### 4.3 Sessions

Sessions are special cases of tumbling windows modeling situations where there are periods of inactivity between subsequent windows and these periods are not included in the window results. Session windows involve the time order and group events that arrive at similar times; they have three parameters: timeout, maximum duration, and (an optional) partitioning key [17]. Partitioning is already part of our WindowBy Clause and a maximum duration is similar to the windows discussed in Sect. 4.2.2. Below we discuss how to represent windows with timeouts.

A timeout window query will continue accepting tuples until the time difference between arrivals of subsequent tuples is greater than the given timeout interval. For example, consider a window that remains open as long as subsequent tuples (from the same station) come within 5 min of each other. OVER cannot express this window because there is no mechanism to compare the difference in time between any arriving tuples; OVER only considers the time difference from the tuple that appends the result. This window is possible to express using WindowBy because the positional variables provide access to both a tuple and the tuple following it. The end condition ensures that the time difference between the last tuple of the window *endCurrentItem* and its subsequent tuple *endNextItem* is greater than 5 min (see Fig. 20).

This session query can also be expressed using a (much more complex) nested loop approach, very similar to the one taken for the unbounded tumbling window shown in Fig. 14b.

### 4.4 Using ONLY and LABEL

This section discusses two additional features in the WindowBy Clause, namely ONLY and LABEL.

#### 4.4.1 ONLY

When the optional keyword ONLY is used with the END condition, it will filter all produced windows and output only the windows that have satisfied their end condition within the sequence; i.e. there was a tuple that ended such windows. Consider for example, the sliding window query in Fig. 9; as is, the query produces three windows (shown in Fig. 10). However, if the END is changed to ONLY END, this query will produce just one window (the window starting at date 1 for station 1). The other two windows are filtered out as they have not met the ending condition (noted also by the MISSING keyword in *ep* and *w* in Fig. 10). Note that when ONLY is used with a tumbling window, it will just check

**Fig. 15** Example of an bounded tuple sized tumbling window using: **a** WindowBy, and **b** OVER

```

SELECT startCurrentItem.Ghcn.station AS station,
        startCurrentItem.Ghcn.date AS date,
        ARRAY_AVG((SELECT VALUE Ghcn.val
                    FROM w)) AS average

FROM Ghcn
WHERE dataType="TMIN"
WINDOW BY w
        TYPE SLIDING
        PARTITION BY Ghcn.station
        ORDER BY Ghcn.date
        START AS startCurrentItem AT startCurrentPos WHEN true
        END AT endCurrentPos WHEN endCurrentPos – startCurrentPos = 19
        (a)

WITH SlidingWindows AS (
        SELECT station,
            date,
            AVG(val) OVER (PARTITION BY station
                           ORDER BY date
                           ROWS BETWEEN CURRENT ROW
                           AND 19 FOLLOWING),
            ROW_NUMBER() OVER (PARTITION BY station
                               ORDER BY date
                               ROWS BETWEEN CURRENT ROW
                               AND 19 FOLLOWING) as pos
        FROM Ghcn
        WHERE dataType="TMIN"
    )
SELECT station,
        date,
        avg
FROM SlidingWindows
WHERE pos % 20 = 1
        (b)

```

Change Window Type to Tumbling

Find the windows that have position 1, 21, 41, ...

whether the last window satisfies the end condition (since by construction all previous windows have ended).

#### 4.4.2 LABEL

By default, WindowBy appends a window to the tuple that starts the window. That is, when a tuple from the sequence is processed and this tuple satisfies the window starting condition, the window created will be appended to this tuple. The optional LABEL keyword allows the user to disassociate the tuple that appends the window result (the tuple currently processed) from the first tuple of the window. When LABEL is used, it points to the currently processed tuple; if the LABEL predicate is satisfied from this tuple, a window is created and its result will be appended to this tuple (pointed by LABEL). However, the user can now use LABEL as a reference point to identify the starting/ending tuples of the window which

can thus be before or after the current tuple (e.g., the window associated with the current tuple can start 10 tuples before the current tuple and end 5 tuples after it). Not every tuple pointed by LABEL creates a window; the LABEL predicate needs to be satisfied. LABEL allows WindowBy to offer the same functionality as OVER's framing (PRECEDING, FOLLOWING etc.) (Fig. 16).

Consider the following window: when a tuple has value of TMIN that is less than 25, compute the average TMIN value of these three tuples: the current tuple, the preceding tuple and the following tuple (from the same station). This 3-tuple window query using WindowBy is shown in Fig. 17a (for reference, the same query in OVER appears in Fig. 17b). Note that this is a modified query of Fig. 9 where the window was computed starting from the current tuple and two tuples following it. As LABEL refers to the tuple currently being processed, its position *labelCurrentPos* in the sequence



**Fig. 16** Expressing a tumbling window of size 1 week using: **a** WindowBy, **b** Recursive CTE and **c** MATCH\_RECOGNIZE

```

SELECT startCurrentItem.station AS station, startCurrentItem.date AS date,
       ARRAY_AVG((SELECT VALUE val FROM w)) AS average
FROM Ghcn
WHERE dataType="TMIN"
WINDOW BY w
      TYPE TUMBLING
      PARTITION BY station
      ORDER BY date
      START AS startCurrentItem WHEN true
      END NEXT AS endNextItem
      WHEN endNextItem.date -
           startCurrentItem.date >= Duration("P1W")

```

Change Window Type to Tumbling

(a)

```

CREATE VIEW w AS (
  WITH NestedQuery AS (
    SELECT l.d AS d, r.val AS val, r.d AS end, l.station AS station
    FROM Ghcn l, Ghcn r
    WHERE r.d >= l.d AND r.station = l.station AND
          r.d < (l.d + '1 week'::interval) AND
          l.dataType="TMIN" AND r.dataType="TMIN"
    SELECT n.d AS s, MAX(n.end) AS e, AVG(n.val) AS average
    FROM NestedQuery n
    GROUP BY n.station, n.d
  )
  WITH RECURSIVE tumble(s, e, station) AS (
    SELECT w1.s AS s, w1.e AS e, w1.station AS station
    FROM w w1
    WHERE w1.s = (SELECT MIN(w2.s)
                  FROM w w2
                  WHERE w1.station = w2.station)

    UNION ALL
    SELECT w1.s AS s, w1.e AS e, w1.station AS station
    FROM tumble t, w w1
    WHERE t.station = w1.station AND
          EXISTS(SELECT MIN(w2.s) FROM w w2
                 WHERE w2.s > t.e AND w1.station=w2.station) AND
          w1.s = (SELECT MIN(w2.s) FROM w w2
                 WHERE w2.s > t.e) AND w1.station = w2.station)
    SELECT w.station, w.s, w.average
    FROM tumble, w
    WHERE w.s = tumble.s AND w.e = tumble.e;

```

This subquery finds all tuples within 1 week of a specific tuple in the sequence

Return the date, endDate and average value for the window after grouping

Recursively find the tumbling windows

Get the results based on the tumbling windows

(b)

```

SELECT station, d, average
FROM Ghcn MATCH_RECOGNIZE (
  PARTITION BY station, dataType
  ORDER BY d
  MEASURES START.station AS station
           START.d AS d,
           AVG(val) AS average,
           START.dataType AS type
  ONE ROW PER MATCH
  AFTER MATCH SKIP TO LAST DOWN
  PATTERN START END?
  DEFINE START AS START.val = START.val
         END AS NEXT(END.d) > (START.d + '1 week'::interval)
)
WHERE type="TMIN"

```

Return start date and average value

Create the tumbling window

Define the Window Start and End Conditions

(c)

```

SELECT labelCurrentItem.Ghcn.station AS station,
       labelCurrentItem.Ghcn.date AS date,
       ARRAY_AVG((SELECT VALUE Ghcn.val FROM w))
FROM Ghcn
WHERE dataType="TMIN"
WINDOW BY w
      TYPE SLIDING
      PARTITION BY Ghcn.station
      ORDER BY Ghcn.date
      LABEL AS labelCurrentItem AT labelCurrentPos
      WHEN labelCurrentItem.Ghcn.val < 25
      START AT startCurrentPos WHEN labelCurrentPos
      END AT endCurrentPos WHEN endCurrentPos - l

```

Change the starting tuple to label and create the boundaries to include the previous and following tuple

```
WITH windows AS (
    SELECT station,
           date,
           val,
           AVG(val) OVER (PARTITION BY station
                          ORDER BY date
                          ROW BETWEEN 1 PRECEDING
                          AND 1 FOLLOWING)
    FROM Ghcn
    WHERE dataType="TMIN")
SELECT station, date, avg
FROM windows
WHERE val < 25;
```

OVER groups all tuples  
between the previous tuple  
and next tuple

FROM Ghcn

WHERE dataType = "TMIN"

WINDOW BY w

TYPE SLIDING

PARTITION BY Ghcn.station

ORDER BY Ghcn.date

LABEL AS labelCurrentItem AT labelCurrentPos

WHEN labelCurrentItem.Ghcn.val < 25

START AT startCurrentPos

WHEN labelCurrentPos - startCurrentPos = 1

END AT endCurrentPos

WHEN endCurrentPos - labelCurrentPos = 1

SELECT labelCurrentItem.Ghcn.station AS station,  
labelCurrentItem.Ghcn.date AS start\_time,  
ARRAY\_AVG(FROM w SELECT VALUE  
Ghcn.val) AS average

3  $B^{out}_{WINDOW\ BY} = B^{in}_{SELECT}$

```
{labelCurrentItem : {Ghcn: {date: 1, station: 1, val: 20, dataType: "TMIN"}},
labelCurrentPos : 1,
startCurrentPos : MISSING,
endCurrentPos : 2,
w : [MISSING, {Ghcn: {date: 1, station: 1, val: 20, dataType: "TMIN"}},
{Ghcn: {date: 2, station: 1, val: 26, dataType: "TMIN"}}]},
{labelCurrentItem : {Ghcn: {{date: 4, station: 1, val: 23, dataType: "TMIN"}},
labelCurrentPos : 4,
startCurrentPos : 2,
endCurrentPos : 5,
w : [{Ghcn: {date: 2, station: 1, val: 26, dataType: "TMIN"}},
{Ghcn: {date: 4, station: 1, val: 23, dataType: "TMIN"}},
{Ghcn: {date: 5, station: 1, val: 30, dataType: "TMIN"}}]},
{labelCurrentItem : {Ghcn: {{date: 1, station: 2, val: 22, dataType: "TMIN"}},
labelCurrentPos : 1,
startCurrentPos : MISSING,
endCurrentPos : 2,
w : [MISSING, {Ghcn: {date: 1, station: 2, val: 22, dataType: "TMIN"}},
{Ghcn: {date: 2, station: 2, val: 28, dataType: "TMIN"}}]}}
```

**Fig. 18** Label bindings. The outputs of each query Clause; numbers indicate where the output was produced

**Fig. 19** WindowBy example of **a** Cumulative average and **b** window between 10 tuples before and 5 tuples before the current tuple

```

SELECT labelCurrentItem.Ghcn.station AS station,
       labelCurrentItem.Ghcn.date AS date,
       ARRAY_AVG((SELECT VALUE Ghcn.val FROM w))
FROM Ghcn
WHERE dataType="TMIN"
WINDOW BY w
  TYPE SLIDING
  PARTITION BY Ghcn.station
  ORDER BY Ghcn.date
  LABEL AS labelCurrentItem WHEN true
  START UNBOUNDED
  END AT endCurrentItem
  WHEN endCurrentItem = labelCurrentItem

```

Change the starting tuple to label and create the boundaries to all tuples from the start to the current tuple

(a)

```

SELECT labelCurrentItem.Ghcn.station AS station,
       labelCurrentItem.Ghcn.date AS date,
       ARRAY_AVG((SELECT VALUE Ghcn.val FROM w))
FROM Ghcn
WHERE dataType="TMIN"
WINDOW BY w
  TYPE SLIDING
  PARTITION BY Ghcn.station
  ORDER BY Ghcn.date
  LABEL AS labelCurrentItem
  AT labelCurrentPos WHEN true
  START AT startCurrentPos
  WHEN labelCurrentPos - startCurrentPos = 10
  END AT endCurrentPos
  WHEN labelCurrentPos - endCurrentPos = 5

```

Change the starting tuple to label and create the boundaries to between 10 tuples before and 5 tuples before

(b)

**Fig. 20** Example of a session window using WindowBy

```

SELECT startCurrentItem.Ghcn.station AS station,
       startCurrentItem.Ghcn.date AS date,
       ARRAY_AVG((SELECT VALUE Ghcn.val FROM w)) AS average
FROM Ghcn
WHERE dataType="TMIN"
WINDOW BY w
  TYPE TUMBLING
  PARTITION BY Ghcn.station
  ORDER BY Ghcn.date
  START AS startCurrentItem WHEN true
  END AS endCurrentItem NEXT AS endNextItem
  WHEN endNextItem.Ghcn.date -
        endCurrentItem.Ghcn.date > Duration("PT5M")

```

Sessions are a special type of tumbling window

Compare the last tuple of the window with the tuple following it to make sure it's within 5 minutes

can be used to start the window one tuple before the current tuple ( $labelCurrentPos - startCurrentPos = 1$ ) and end it one tuple after the current tuple ( $endCurrentPos - labelCurrentPos = 1$ ). The resulting bindings are shown in Fig. 18. When there is no tuple preceding  $labelCurrentPos$ , any start variables ( $startCurrentPos$ ) are reported as MISSING; similarly,  $w$  also shows a MISSING before the tuples in the window. The only window that satisfies the starting condition is the window that has a label at the tuple with station 1 and date 4.

As another example consider computing cumulative averages over sliding windows. That is, given a station, find the average TMIN value starting from the start of the sequence and ending on the current tuple, for each tuple for that station. Figure 19a depicts this query in WindowBy. Note that the ending tuple of the window is set to be the LABEL tuple (shown by  $endCurrentItem = labelCurrentItem$ ) which is the current tuple processed (and the tuple where the window will be appended). The START is set to UNBOUNDED, and the END condition is set to be always true.

Using LABEL allows the window not to include the current tuple. As an example, consider computing the average TMIN value for a sliding window that starts ten tuples before the given tuple and ends five tuples before the given tuple. Figure 19b depicts this query in WindowBy. Note that the LABEL predicate is set to be always true and as a result every tuple generates a window. START is set to be ten tuples before the label while END is set to be five tuples before the label.

## 5 Basic implementation

As a proof of concept, in this section we present algorithms that implement WindowBy sliding windows both for single and multi-node environments. For simplicity we focus on the default case where the label coincides with the start of the sliding window; these algorithms can be extended easily to support the case where the label is not the window's starting tuple. The algorithm can also support the creation of tumbling windows by effectively filtering (selecting) the tumbling windows out of all the sliding windows. A number of optimizations are possible, but we leave such discussions for follow-up work.

### 5.1 Sliding windows

Consider a WindowBy query creating sliding windows over an input sequence of  $n$  tuples; its output will contain between 0 and  $n$  windows (since not every tuple creates a window). As noted earlier, each window created is represented by a tuple that contains: one attribute for each variable declared (e.g. the starting and ending tuples and their positions) and one

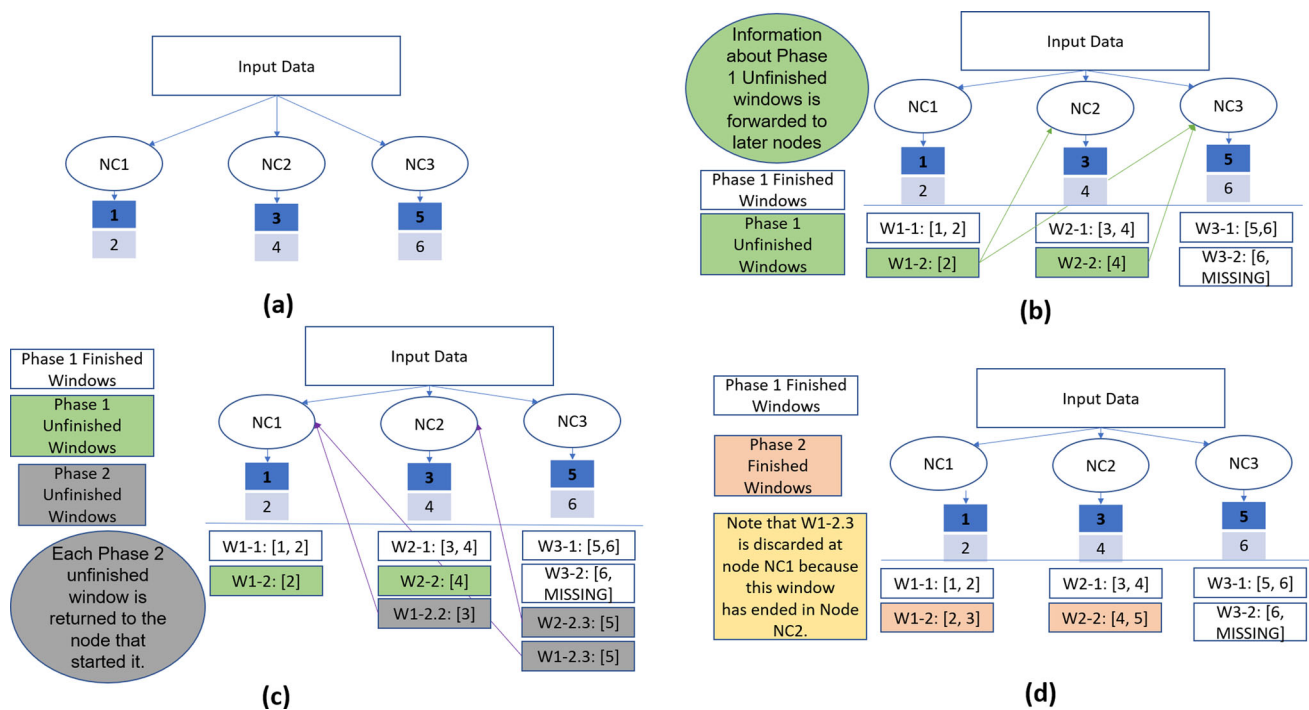
attribute to capture the result (all the tuples of the window created). For example in Fig. 10, the first window tuple (see box 3) has 4 attributes ( $startCurrentItem, startCurrentPos, endCurrentPos$  and  $w$ ).

Consider first the case where the ordered sequence can be stored in a single node. Our basic WindowBy evaluation algorithm will sequentially process the tuples of this ordered sequence. For each tuple it will check whether the starting condition is satisfied, in which case it creates a new window. It then adds this tuple to any currently open windows (by 'add' we mean that this tuple is used in the calculation of the result of a window, typically an aggregate). Afterwards it checks if this tuple ends any open window(s), in which case it closes that window. For each window the variables for its start and end are assigned when the window is created/closed. Any windows that remain open after the entire tuple sequence is processed are reported in the output with the MISSING keyword added. If the ONLY option is used, then only closed windows are reported.

If the ordered sequence is too large to fit in a single node, it will be sliced into ordered subsequences, each handled by a different node. This creates an ordered node sequence, i.e., node NC1 followed by node NC2, followed by NC3, as shown in Fig. 21a (which also shows the tuples of the subsequence assigned to each node). Note that there are three ways that the windows can occur when the ordered sequence is distributed: (1) windows that start and end in the same node, (2) windows that start in a node but end in a later node, and (3) windows that start in a node but never end. In our approach, windows will be reported at the node where they started.

In this environment the WindowBy algorithm can create sliding windows in at most two passes. The first pass will use the previous single-node algorithm to find the locally completed windows within each node (i.e., those windows that start and end within the same node). Figure 21b, shows the finished windows after the first pass assuming the data is distributed in three nodes (NC1, NC2, NC3) as in Fig. 21a, assuming that the query is a sliding window of size 2 tuples. The windows shown in the figure follow the notation  $W_i:j:[list]$  where  $i$  corresponds to the NC where this window started,  $j$  is the id of this window within that NC and  $list$  provides the result tuples currently in this window. The finished windows in the first pass are W1-1, W2-1, W3-1 and W3-2 and correspond to windows that start and end within the same node. W1-1 is the first window created by NC1 and contains tuples 1 and 2. Node NC3 created 2 windows; the second one W3-2 has the list [6, MISSING] since it started at tuple 6 in the sequence and was open when the full sequence was processed (NC3 is the last node). Note that only the last node in the node sequence will add MISSING to its unfinished windows that did not close after processing its subsequence since this marks the end of the whole sequence. If every win-





**Fig. 21** Parallel Sliding Window Example: **a** distributing the data, **b** single node sliding window results, **c** processing of unfinished windows, and **d** final calculation of results

row can be completed locally, the algorithm ends in just one pass.

If there are windows that cannot be completed within a node (that is, windows that start in a given node but are still open when this node finished processing its subsequence), each node sends information about all its unfinished windows created after the first pass, to the nodes that are later in the node sequence; this is because such windows can end in any of the subsequent nodes (or remain open after the whole sequence is processed). In the example of Fig. 21b these are windows W1-2 and W2-2. For each unfinished window the information sent includes the window's  $W_{i-j}$  identification and its *start variables* (but no window tuples are transferred).

In the second pass, each node works on any unfinished windows that it may have received from the first pass. A major difference with the first pass is that now, when processing a tuple, the algorithm does not check if this tuple starts a window; this was done in the first pass and all window starts are known. Each node will process *every* tuple in its subsequence. A given tuple will be considered by all the unfinished windows the NC received from the first pass as long as these windows are still open. Moreover, the algorithm will check if a given tuple ends any of those unfinished windows. If it does, the particular window is closed and it cannot consider any further tuples. The results of these windows are stored using the notation  $W_{i-j.k}:[list]$  where  $k$  corresponds

to the id of the NC that computed the result for unfinished window  $W_{i-j}$ . In Fig. 21c there are three unfinished window results: in W1-2.2:[3], W1-2.3:[5], and W2-1.3:[5].

During the second pass a node does not know if any of its unfinished windows may have finished at an earlier node; hence it will calculate result for any of the unfinished windows it received. (An easy optimization can mitigate this issue: when a node closes an unfinished window it can inform all later nodes in the node sequence that this unfinished window closed; however such optimization is not implemented yet). At the end of the second pass, each node sends its unfinished windows and their calculated tuples (including the *end variables* for those windows that it closed) to the node that started the respective unfinished window.

Using this information the node that started each window can now compute the final result for that window (Fig. 21d). For window W1-2, NC1 received results from NC2 (W1-2.2:[3]) and NC3 (W1-2.3:[5]) but the result from NC2 ends the window. Thus W1-2.3:[5] is discarded when calculating window W1-2.

The above distributed window algorithm can be easily updated in the case where LABEL is used; the difference is that now a node will send the information about its unfinished windows to nodes before or after it depending on the range of the window. A window will be reported by the node that contains the window's label tuple.

## 5.2 Tumbling windows

A generic algorithm for calculating tumbling windows is to first calculate the set of sliding windows as above and then identify the tumbling windows using this set. To do so, all the boundaries of the sliding window are ordered and then sent to one node. The node can then sequentially identify the sequence of tumbling windows (another window cannot be open until the previous closes). Various optimizations are possible depending on the window query but this is out of this paper's focus.

## 6 Experimental evaluation

In this section we compare the performance of our initial WindowBy implementation against the SQL++ OVER implemented on the Apache AsterixDB big data management system. We note that the SQL++ OVER Clause implementation has been tested and optimized; it is also been used and tested in an industrial strength system, Couchbase Analytics [21] (which is built on AsterixDB). The experiments here focus on the general WindowBy functionality with a largely unoptimized WindowBy implementation. Our approach focuses on a range-based partitioning scheme, so some stations' data sequences can be split across nodes. SQL OVER's hash-based partitioning ensures that each station's sequence lies within a single partition. As a result, we expect OVER to perform better because it does not require any communication or data transfers between partitions. In contrast, the only optimization that has been applied on our initial WindowBy proof-of-concept implementation is the obvious aggregate push-down (which limits how many tuples are carried between stages of the WindowBy algorithm). In our initial WindowBy implementation, we allow an ordered sequence to be split among partitions. As a result, only aggregate functions that implement a way to aggregate partial results work. Holistic functions like rank thus do not yet work in our implementation.

For our performance evaluation we used the window examples discussed in Sect. 4. For comparison purposes, for all queries we also report the performance of a SQL++ nested-loop implementation. In our experiments we used large GHCN datasets (with sizes ranging from 400 GB to 800 GB). Each tuple contains four attributes: station, date, value and dataType, using the DDL shown in Fig. 8. The experiments were run on AWS using i3.xlarge instances, each with 2.3 GHz Intel Xeon Scalable Processors, 4 vCPUs and 30.5 GB RAM. Each experiment was run using 1, 2, 4, 8 and 16 instances. The results in the charts are the result of run-

ning the experiments 3 times and taking the average of these 3 runs.

## 6.1 Sliding windows

### 6.1.1 Bounded windows

The first experiment considers the sliding window query with fixed duration of 1 week from Fig. 11. The query times appear in Fig. 22 using two datasets: **a** 400 GB and **b** 800 GB, while varying the number of nodes from 1 to 16.

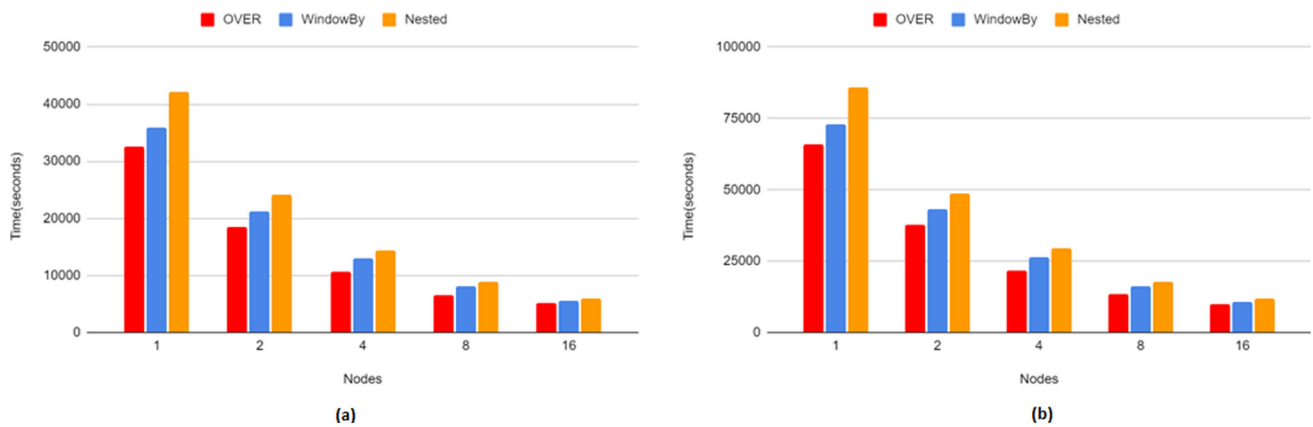
Out of the three approaches, OVER is the fastest with WindowBy close second and the nested approach being the slowest. The performance of OVER is better than WindowBy by 10–20% since the current WindowBy implementation requires a second pass over (some) data. Moreover, according to the current implementation of WindowBy (see algorithm in Sect. 5), during the second pass the last node receives unfinished windows from all nodes before it and has to process them even if they may not apply to its data. This step finishes when all nodes finish processing and communicating their results before the next step can start. One can limit the amount of communication and the number of unfinished windows sent using a range map but this optimization has not been implemented yet for WindowBy.

Another observation is that all approaches benefit from adding more nodes, experiencing good speed-up as the number of nodes is doubled. The relative performance differences between the three approaches become less apparent when 16 nodes are used; in that case even for the large file (800 GB) each partition is small enough to fully fit in main memory.

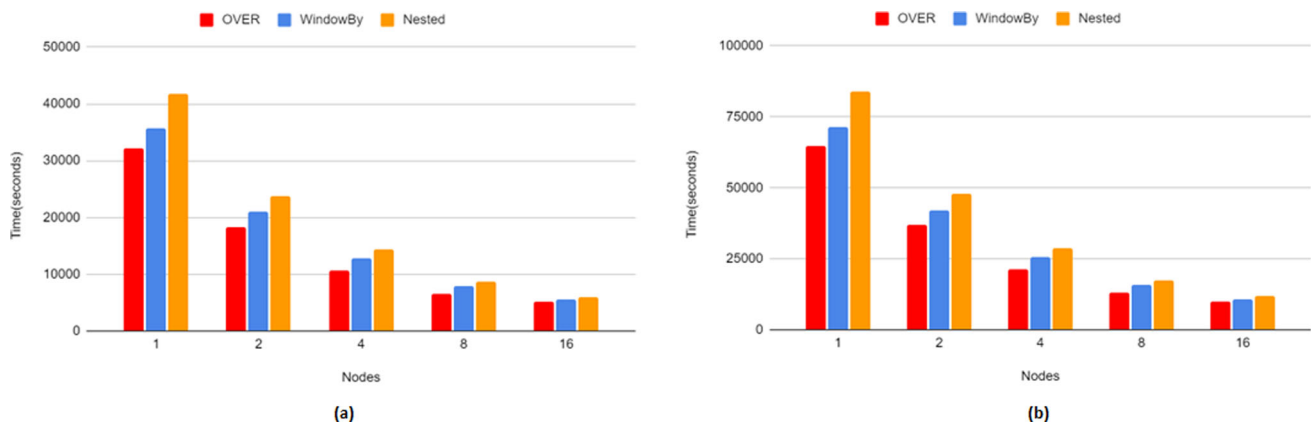
We next consider the sliding window with fixed length of twenty tuples from Fig. 12. The results are depicted in Fig. 23; all approaches behave similarly to the fixed duration sliding window query

### 6.1.2 Unbounded windows

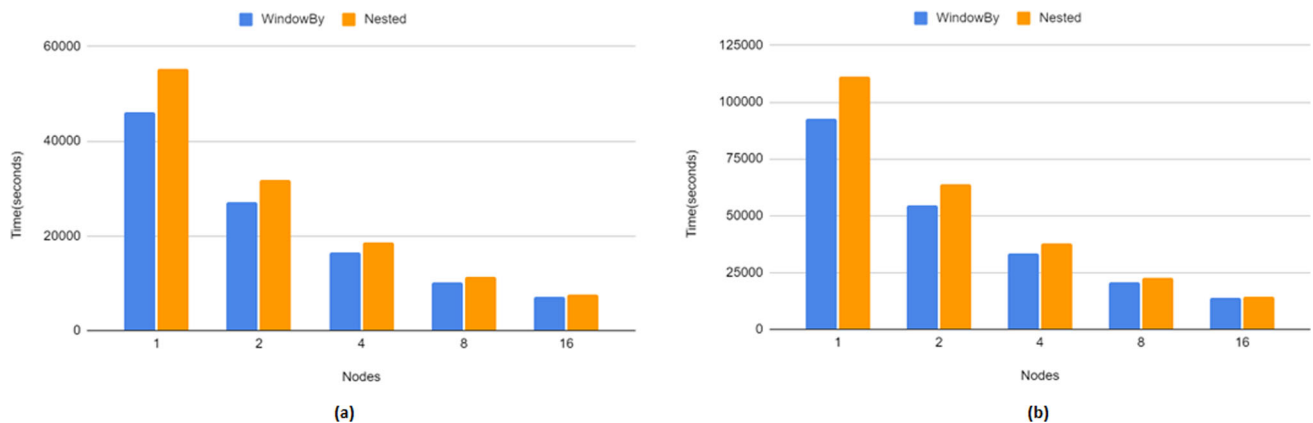
As an unbounded sliding window example we used the sliding window that stays open while the tuple's TMIN value is less than 25 (from Fig. 13). Recall that this query cannot be expressed using OVER since the window is not based on the ordering attribute. The experimental results appear in Fig. 24; again the query performance of the nested-loop approach (Fig. 13a) is slower than our WindowBy by 20% for a single node and this slowly drops to 7% at 16 nodes. As expected, the unbounded sliding window takes relatively more time than the bounded sliding examples from the previous section, since more work needs to be performed for each window.



**Fig. 22** Query performance for sliding window with fixed duration (1 week) for dataset sizes: **a** 400 GB and **b** 800 GB



**Fig. 23** Query performance for sliding window with fixed length (20 tuples) for dataset sizes: **a** 400 GB and **b** 800 GB



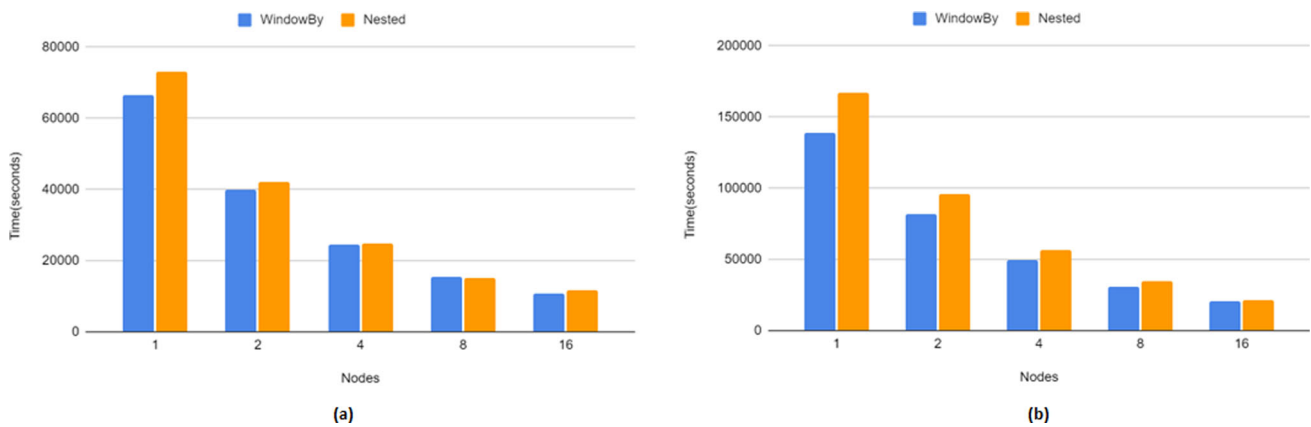
**Fig. 24** Query performance for unbounded sliding window for dataset sizes: **a** 400 GB and **b** 800 GB

## 6.2 Tumbling windows

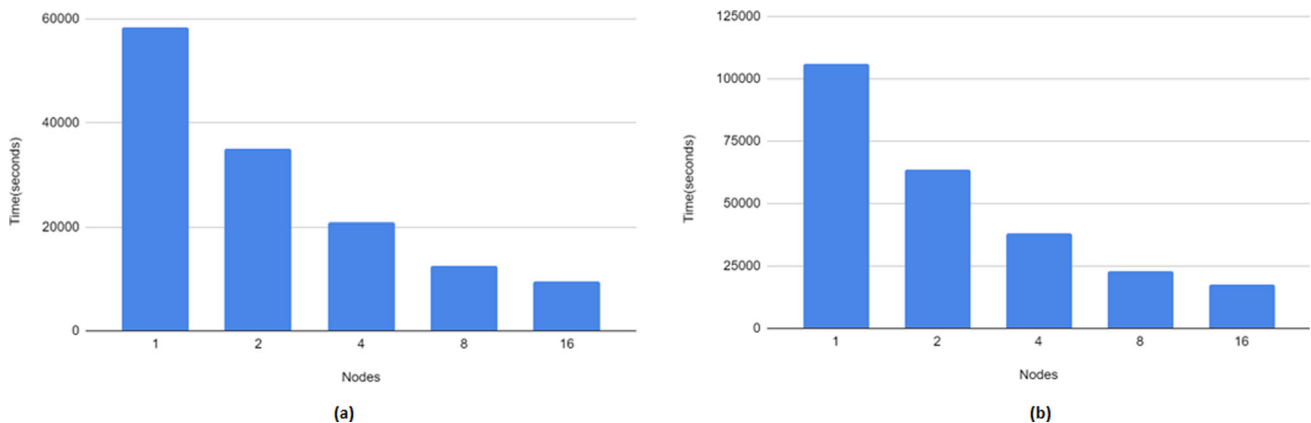
### 6.2.1 Unbounded windows

Consider first the unbounded tumbling window that stays open while the tuple's TMIN value is less than 25 (seen in Fig. 14). The performance results for the WindowBy and

nested approaches appear in Fig. 25 (recall that both the tumbling window examples used in Sect. 4.2 are not possible with OVER). The nested-loop approach still takes longer than the WindowBy query by about 19% in a single node case down to 6% in the 16 node case. Note that both the WindowBy and nested implementations of this tumbling window take longer than their respective implementations of the sliding variant



**Fig. 25** Query performance for unbounded tumbling window for dataset sizes: **a** 400 GB and **b** 800 GB



**Fig. 26** Query performance for tumbling window with fixed duration (1 week) for dataset sizes: **a** 400 GB and **b** 800 GB

of this query (from Fig. 24). This is expected since our initial tumbling window implementation uses a generic algorithm that first calculates the set of all sliding windows and then identifies the tumbling windows amongst them.

### 6.2.2 Bounded windows

Next we examine a tumbling window with fixed duration of 1 week (Fig. 16). We can only implement this query using WindowBy since AsterixDB currently does not support recursion or MATCH\_RECOGNIZE. The results appear in Fig. 26. As expected, the tumbling window query takes longer than the sliding window variant (Fig. 22), and is faster than its unbounded tumbling version (Fig. 25).

## 6.3 Label and session windows

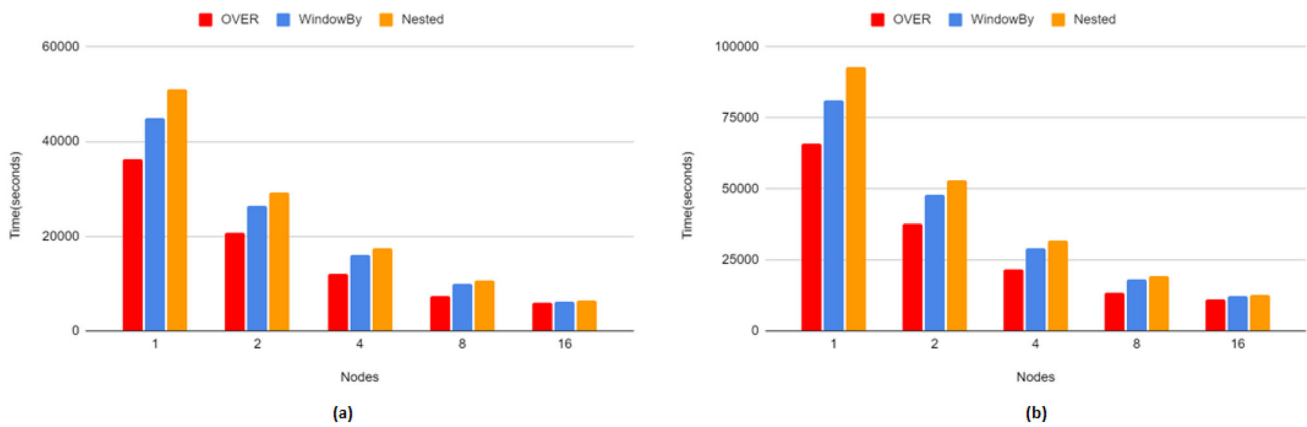
The following experiments consider the performance of windows with labels as well as session windows.

Consider the window with label example given in Fig. 17. This query represents a sliding window of length three tuples between the tuple preceding and the tuple after the current

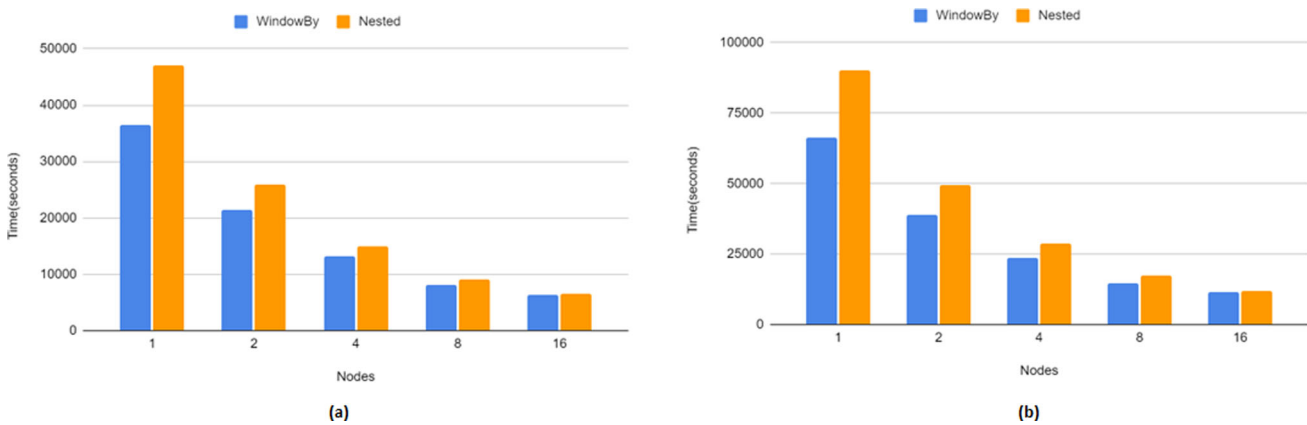
tuple, when the current tuple's TMIN value is less than 25. The experimental results appear in Fig. 27. The performance of the label sliding window in WindowBy is slightly slower by 15% than a plain (no-label) sliding window case (say Fig. 23) because additional communication is now required with nodes before the current node (in addition to nodes after it). In our initial implementation a node sends unfinished windows to all nodes that require them and in this case, it sends them to all nodes before and after itself; as before, optimizations can be applied to reduce that communication but this is left for future work.

We next experimented with a session window with timeout using the query shown in Fig. 20. This query cannot be written using OVER because there is no mechanism to compare the difference in time between arriving tuples. It can be written using a rather complex SQL++ nested approach. (see Fig. 33 in the “Appendix”). The query performance between the WindowBy and nested approaches appears in Fig. 28. As with the tumbling unbounded window performance (of which sessions is a special case) the WindowBy approach is faster than the nested one.

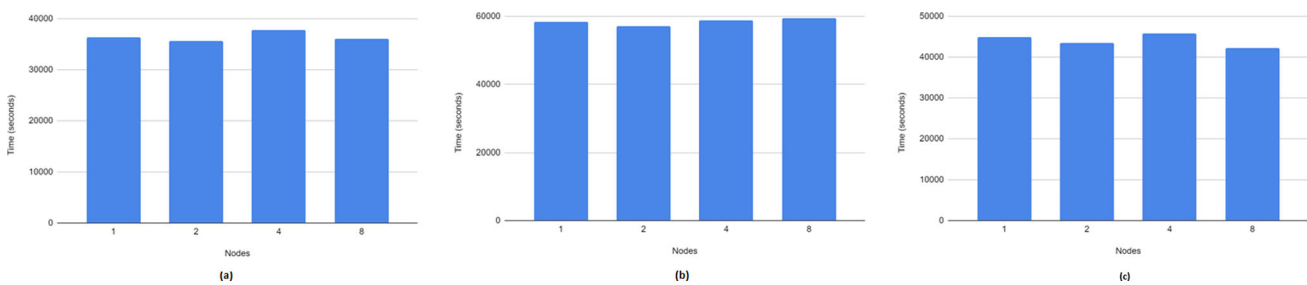




**Fig. 27** Experimental results for label window over dataset sizes: **a** 400GB and **b** 800GB



**Fig. 28** Query performance for a session window with timeout for dataset sizes: **a** 400GB and **b** 800GB



**Fig. 29** Scale-Up experiments using **a** a sliding, and **b** a tumbling window with fixed duration (1 week) and **c** a label window query. The dataset size per node is 400GB

## 6.4 Scale-up experiments

Finally we examined how well our implementation of the WindowBy Clause scales. For the scale-up experiments we used three window examples: the sliding window with fixed duration of 1 week (from Fig. 11), the tumbling version of the same window (from Fig. 16) and a label window (from Fig. 17). In particular we started with one node handling a dataset of size 400GB and continued doubling the number

of nodes and the size of the overall dataset, while keeping the dataset size per node at 400GB (that is, two nodes handled 800GB, 4 nodes 1.6TB etc.) The query performance results appear in Fig. 29. As it can be seen our WindowBy implementation achieves good scale-up performance; the query execution time remains roughly the same, which means that the additional data is processed in roughly the same amount of time.

## 7 Conclusions

One of the major advantages of the WindowBy Clause is that the window start and end can be expressed by predicates. This simplifies sliding and tumbling windows for users. As such, our WindowBy Clause is more intuitive compared to the SQL OVER Clause and many of the current streaming approaches. This makes it a potentially valuable addition to SQL++ systems. While it may not be as expressive as the MATCH\_RECOGNIZE clause that was recently added to the SQL standard, it is far easier to write a query. Various optimizations can be applied depending on the window query. For example, if the end condition is a fixed condition (e.g. TMIN value greater than or equal to 25) then the entire window can be calculated in one pass. As future research we are examining how to fully optimize WindowBy. This includes optimizations that should bring comparable performance to OVER as well as ways to finish some queries in a single pass.

**Acknowledgements** We would like to thank Yannis Papakonstantinou for early discussions on the window clause. This research was supported by NSF Grants IIS-1954644, IIS-1954962, CNS-1924694 and CNS-1925610 and by the Donald Bren Foundation (via a Bren Chair at UC Irvine).

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## Appendix

Below we present SQL++ *nested-loop* approaches for some of the window queries used in the experiments.

### A.1: Sliding window with fixed duration

The query (fixed duration of 1 week) appears in Fig. 30. While not complex, it still requires looping over the data to find the tuples in each window.

```
SELECT startTuple.station,
       startTuple.date,
       AVG(windowTuple.val)
FROM Gchn AS startTuple, Gchn AS windowTuple
WHERE startTuple.dataType = "TMIN" AND
      windowTuple.dataType = "TMIN" AND
      startTuple.station = windowTuple.station AND
      windowTuple.date >= startTuple.date AND
      windowTuple.date - startTuple.date <= Duration("P7D")
GROUP BY startTuple.station, startTuple.startDate;
```

**Fig. 30** A sliding window with fixed duration (1 week) using the SQL++ Nested-loop approach

```
WITH GchnPos AS (
  SELECT station,
         date,
         val,
         ROW_NUMBER() OVER (PARTITION BY station
                              ORDER BY date) AS pos
FROM Gchn
WHERE dataType="TMIN"
)
SELECT startTuple.station,
       startTuple.date,
       AVG(windowTuple.val)
FROM GchnPos AS startTuple, GchnPos AS windowTuple
WHERE startTuple.station = windowTuple.station AND
      windowTuple.pos >= startTuple.pos AND
      windowTuple.pos - startTuple.pos <= 19
GROUP BY startTuple.station, startTuple.startDate;
```

**Fig. 31** A sliding window with fixed length (20 tuples) using the SQL++ Nested-loop approach

### A.2: Sliding window with fixed length

The query (fixed length of 20 tuples) appears in Fig. 31. It is more complex than the fixed duration query because it requires attaching positional variables to each tuple to create a 20 tuple long window. As before, it requires looping over the data to build the windows.

```

WITH GhcnPos AS (
  SELECT station,
         date,
         val,
         ROW_NUMBER() OVER (PARTITION BY station
                             ORDER BY date) AS pos
  FROM Ghcn
  WHERE dataType="TMIN"
), wStart AS (
  SELECT station, date, val, pos
  FROM GhcnPos
  WHERE val < 25
)
SELECT startTuple.station,
       startTuple.date,
       AVG(windowTuple.val)
FROM wStart AS startTuple, GhcnPos AS windowTuple
WHERE startTuple.station = windowTuple.station AND
      windowTuple.pos >= startTuple.pos - 1 AND
      windowTuple.pos - startTuple.pos <= 1
GROUP BY startTuple.station, startTuple.date;

```

**Fig. 32** Label Query for previous and following tuple using Nested-loop SQL++ approach

### A.3: Sliding window with label

This label query represents a sliding window of length three tuples, starting with the tuple preceding and ending with the tuple after the current tuple, when the current tuple's TMIN value is less than 25; it appears in Fig. 32. The approach taken is similar with the nested-loop approach we followed for the sliding window example in Fig. 13a. In particular, the nested solution requires finding first the tuples that can start a window through the wStart table. Once these label tuples are found, the query loops over the Ghcn data and finds any Ghcn tuples in the defined window length from that label tuple. As the tuple positions are available in table GhcnPos, a startTuple's position needs to be compared with that of a windowTuple's position.

### A.4: Session window

The session window query appears in Fig. 33. Session windows are special cases of tumbling windows and are typically very complex to write using the Nested-loop SQL++ approach.

```

WITH GhcnPos AS (
  SELECT station,
         date,
         val,
         ROW_NUMBER() OVER (PARTITION BY station
                             ORDER BY date) AS pos
  FROM Ghcn
  WHERE dataType="TMIN"
), currentPrev AS (
  SELECT currentTuple.station AS station,
         currentTuple.pos AS currentPos,
         currentTuple.val AS currentVal,
         currentTuple.date AS currentdate,
         previousTuple.pos AS previousPos,
         previousTuple.val AS previousVal,
         previousTuple.date AS previousdate
  FROM GhcnPos AS currentTuple, GhcnPos AS previousTuple
  WHERE currentTuple.station = previousTuple.station AND
        ((currentTuple.pos = previousTuple.pos + 1) OR
         (currentTuple.pos = 1 AND previousTuple.pos = 1))
), wStart AS (
  SELECT station, currentdate AS date
  FROM currentPrev
  WHERE currentVal < 25 AND
        (currentPos = 1 OR previousVal > 24)
), wEnd AS (
  SELECT c1.station, c1.currentdate AS date
  FROM currentPrev AS c1
  LET stationLastPos = (SELECT VALUE MAX(pos)
                       FROM GhcnPos AS g1
                       WHERE g1.station = c1.station)[0]
  WHERE currentPos >= stationLastPos OR
        (currentdate - previousdate >
         Duration("PT5M"))
), wBoundaries AS (
  SELECT wStart.date AS startDate,
         MIN(wEnd.date) AS endDate,
         wStart.station AS station
  FROM wStart, wEnd
  WHERE wStart.station = wEnd.station AND
        wEnd.date >= wStart.date
  GROUP BY wStart.station, wStart.date
)
SELECT startTuple.station,
       wBoundaries.startDate AS date,
       AVG(windowTuple.val)
FROM GhcnPos AS startTuple, GhcnPos AS windowTuple, wBoundaries
WHERE startTuple.station = windowTuple.station AND
      wBoundaries.station = startTuple.station AND
      startTuple.date = wBoundaries.startDate AND
      wBoundaries.startDate <= windowTuple.date AND
      wBoundaries.endDate >= windowTuple.date
GROUP BY startTuple.station, wBoundaries.startDate;

```

**Fig. 33** Session window using Nested-loop SQL++ approach

## References

- Abadi, D.J., et al.: Aurora: a new model and architecture for data stream management. *VLDB J.* **12**, 120–139 (2003)
- Akidau, T., et al.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In: *Proceedings of the VLDB Endowment* (2015)
- Alsubaiee, S., et al.: AsterixDB: a scalable, open source BDMS. In: *arXiv preprint arXiv:1407.0454* (2014)
- Arasu, A., Babu, S., Widom, J.: CQL: a language for continuous queries over streams and relations. In: *Database Programming Languages: 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6–8, 2003. Revised Papers 9*. Springer, pp. 1–19 (2004)
- Awad, A., Traub, J., Sakr, S.: Adaptive watermarks: a concept drift-based approach for predicting event-time progress in data streams. In: *EDBT*, pp. 622–625 (2019)
- Begoli, E., et al.: Apache calcite: a foundational framework for optimized query processing over heterogeneous data sources. In: *Proceedings of the 2018 International Conference on Management of Data*, pp. 221–230 (2018)
- Begoli, E., et al.: One SQL to rule them all: an efficient and syntactically idiomatic approach to management of streams and tables. In: *Proceedings of the 2019 International Conference on Management of Data*, pp. 1757–1772 (2019)
- Bellamkonda, S., et al.: Adaptive and big data scale parallel execution in oracle. *Proc. VLDB Endow.* **6**(11), 1102–1113 (2013)
- Botan, I., et al.: Extending XQuery with window functions. In: *VLDB*, pp. 75–86 (2007)
- Cao, Y., et al.: Optimization of analytic window functions. In: *arXiv preprint arXiv:1208.0086* (2012)
- Carbone, P., et al.: Apache flink: stream and batch processing in a single engine. *Bull. Tech. Comm. Data Eng.* **38**(4) (2015)
- Chamberlin, D.: *SQL++ for SQL Users: A Tutorial*. Couchbase Inc., Santa Clara (2018)
- Chandramouli, B., et al.: Trill: a high performance incremental query processor for diverse analytics. *Proc. VLDB Endow.* **8**(4), 401–412 (2014)
- Chandrasekaran, S., et al.: TelegraphCQ: continuous dataflow processing. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 668–668 (2003)
- Coelho, F., et al.: Reducing data transfer in parallel processing of SQL window functions. In: *CLOSER* (1), pp. 343–347 (2016)
- Deep Dive into 12c MATCH RECOGNIZE. In: *Oracle* (2015). <https://www.oracle.com/technetwork/database/bi-datawarehousing/mr-deep-dive-3769287.pdf>
- Eiden, F., Howell, J., et al.: Session window (azure stream analytics)—stream analytics query. In: *Session Window (Azure Stream Analytics)—Stream Analytics Query—Microsoft Docs* (2013). <https://docs.microsoft.com/en-us/stream-analytics-query/session-window-azure-stream-analytics>
- Fischer, P.M., Garg, A., Sheykh, K.E.: Extending XQuery with a pattern matching facility. In: *Database and XML Technologies: 7th International XML Database Symposium, XSym 2010, Singapore, September 17, 2010. Proceedings 7*. Springer, pp. 48–57 (2010)
- Global Historical Climatology Network daily. In: *Global Historical Climatology Network daily (GHCNd)* (2012). <https://www.ncdc.noaa.gov/products/land-based-station/global-historical-climatology-network-daily>
- Hirzel, M., et al.: SPL stream processing language specification. In: *New York: IBM Research Division TJ. Watson Research Center, IBM Research Report: RC24897 (W0911 044)* (2009)
- Hubail, M.A., et al.: Couchbase analytics: NoETL for scalable NoSQL data analysis. *Proc. VLDB Endow.* **12**(12), 2275–2286 (2019)
- Iqbal, M.H., Soomro, T.R., et al.: Big data analysis: Apache storm perspective. *Int. J. Comput. Trends Technol.* **19**(1), 9–14 (2015)
- Jain, N., et al.: Towards a streaming SQL standard. *Proc. VLDB Endow.* **1**(2), 1379–1390 (2008)
- Jain, S., et al.: Sqlshare: results from a multi-year SQL-as-a-service experiment. In: *Proceedings of the 2016 International Conference on Management of Data*, pp. 281–293 (2016)
- Kolios, A., et al.: Saber: window based hybrid stream processing for heterogeneous architectures. In: *Proceedings of the 2016 International Conference on Management of Data*, pp. 555–569 (2016)
- Leis, V., et al.: Efficient processing of window functions in analytical SQL queries. *Proc. VLDB Endow.* **8**(10), 1058–1069 (2015)
- Li, J., et al.: Semantics and evaluation techniques for window aggregates in data streams. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 311–322 (2005)
- Miao, H., et al.: StreamBox: modern stream processing on a multicore machine. In: *USENIX Annual Technical Conference*, pp. 617–629 (2017)
- Michels, J.: The new improved SQL: 2016 standard. *ACM SIGMOD Rec.* **47.2**, 51–60 (2018)
- Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ query language: configurable, unifying and semi-structured. In: *arXiv preprint arXiv:1405.3631* (2014)
- Pathirage, M., et al.: SamzaSQL: scalable fast data management with streaming SQL. In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, pp. 1627–1636 (2016)
- Patroumpas, K., Sellis, T.: Window specification over data streams. In: *Current Trends in Database Technology-EDBT 2006: EDBT 2006 Workshops PhD, DataX, IIDB, IHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26–31, 2006, Revised Selected Papers 10*. Springer, pp. 445–464 (2006)
- Song, G., et al.: Approximate calculation of window aggregate functions via global random sample. *Data Sci. Eng.* **3**, 40–51 (2018)
- Tangwongsan, K., et al.: General incremental sliding-window aggregation. *Proc. VLDB Endow.* **8**(7), 702–713 (2015)
- Theodorakis, G., et al.: Lightsaber: efficient window aggregation on multi-core processors. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2505–2521 (2020)
- Traub, J., et al.: Scotty: efficient window aggregation for out-of-order stream processing. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, pp. 1300–1303 (2018)
- Traub, J., et al.: Scotty: general and efficient open-source window aggregation for stream processing systems. *ACM Trans. Database Syst. (TODS)* **46**(1), 1–46 (2021)
- Wesley, R., Xu, F.: Incremental computation of common windowed holistic aggregates. *Proc. VLDB Endow.* **9**(12), 1221–1232 (2016)
- Widom, J., et al.: The Stanford data stream management system. In: *Believed to be Prior to*, p. 24 (2007)
- XQuery 3.1: An XML query language. In: *WC3* (2019). <https://www.w3.org/TR/xquery-31>
- Zaharia, M., et al.: Apache spark: a unified engine for big data processing. *Commun. ACM* **59**(11), 56–65 (2016)
- Zaharia, M., et al.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: *HotCloud* 12.10-10 (2012)



43. Zemke, F.: What's new in SQL: 2011. *ACM SIGMOD Rec.* **41**(1), 67–73 (2012)
44. Zhu, E., Huang, S., Chaudhuri, S.: High-performance row pattern recognition using joins. *Proc. VLDB Endow.* **16**(5), 1181–1195 (2023)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.