Linked Stream Data Processing

Danh Le-Phuoc, Josiane Xavier Parreira, and Manfred Hauswirth

Digital Enterprise Research Institute,National University of Ireland, Galway
{danh.lephuoc,josiane.parreira,manfred.hauswirth}@deri.org

Abstract. Linked Stream Data has emerged as an effort to represent dynamic, time-dependent data streams following the principles of Linked Data. Given the increasing number of available stream data sources like sensors and social network services, Linked Stream Data allows an easy and seamless integration, not only among heterogenous stream data, but also between streams and Linked Data collections, enabling a new range of real-time applications.

This tutorial gives an overview about Linked Stream Data processing. It describes the basic requirements for the processing, highlighting the challenges that are faced, such as managing the temporal aspects and memory overflow. It presents the different architectures for Linked Stream Data processing engines, their advantages and disadvantages. The tutorial also reviews the state of the art Linked Stream Data processing systems, and provide a comparison among them regarding the design choices and overall performance. A short discussion of the current challenges in open problems is given at the end.

Keywords: Linked Stream Data, Data Stream Management Systems, Linked Data, Sensors, query processing.

1 Introduction

We are witnessing a paradigm shift, where real-time, time-dependent data is becoming ubiquitous. Sensor devices were never so popular. For example, mobile phones (accelerometer, compass, GPS, camera, etc.), weather observation stations (temperature, humidity, etc.), patient monitoring systems (heart rate, blood pressure, etc.), location tracking systems (GPS, RFID, etc.), buildings management systems (energy consumption, environmental conditions, etc.), and cars (engine monitoring, driver monitoring, etc.) are continuously producing an enormous amount of information in the form of data streams. Also on the Web, services like Twitter, Facebook and blogs, deliver streams of (typically unstructured) real-time data on various topics.

Integrating these new information sources—not only among themselves, but also with other existing sources—would enable a vast range of new, real-time applications in the areas of smart cities, green IT, e-health, to name a few. However, due to the heterogeneous nature of such diverse streams, harvesting the data is still a difficult and labor-intensive task, which currently requires a lot of "hand-crafting."

Recently, there have been efforts to lift stream data to a semantic level, e.g., by the W3C Semantic Sensor Network Incubator Group[1] and [25,103,126]. The goal is to make stream data available according to the Linked Data principles [22]—a concept that is known as *Linked Stream Data* [99]. As Linked Data facilitates the data integration process among heterogenous collections, Linked Stream Data has the same goal with respect to data streams. Moreover, it also bridges the gap between stream and more static data sources.

Besides the existing work on Linked Data, Linked Stream Data also benefits from the research in the area of Data Streams Management Systems (DSMS). However, particular aspects of Linked Stream Data prevents existing work in these two areas to be directly applied. One distinguishing aspect of streams that the Linked Data principles do not consider is their temporal nature. Usually, Linked Data is considered to change infrequently. Data is first crawled and stored in a centralised repository before further processing. Updates on a dataset are usually limited to a small fraction of the dataset and occur infrequently, or the whole dataset is replaced by a new version entirely. Query processing in Linked Data databases, as in traditional relational databases, is *pull* based and *one-time*, i.e., the data is read from the disk, the query is executed against it once, and the output is a set of results for that point in time. In contrast, in Linked Stream Data, new data items are produced continuously, the data is often valid only during a time window, and it is continually *pushed* to the query processor. Queries are continuous, i.e., they are registered once and then are evaluated continuously over time against the changing dataset. The results of a continuous query are updated as new data appears. Therefore, current Linked Data query processing engines are not suitable for handling Linked Stream Data. It is interesting to notice that in recent years, there has been work that points out the dynamics of Linked Data collections [115,107]. Although at a much slower pace compared to streams, it has been observed that centralised approaches will not be suitable if *freshness* of the results is important, i.e., the query results are consistent with the actual "live" data under certain guarantees, and thus an element of "live" query execution will be needed [107]. Though this differs from stream data, some of properties and techniques for Linked Stream Data processing may also be applicable to this area.

Data Streams Management Systems, on the other hand, are designed to handle and scale with fast changing, temporal data, such as Linked Stream Data. However Linked Stream Data is usually represented as an extension of RDF—the most popular standard for Linked Data representation. This contrasts with the relational storage model used in DSMS. It has been shown that in order to efficiently process RDF data using the relational model, the data needs to be heavily replicated [125,91]. Replication fast changing RDF streams is prohibitive, therefore DSMS can't be directly used for storage and processing of Linked Stream Data.

In this tutorial we will give an overview on Linked Stream Data processing. We will highlight the basic requirements, the different solutions and the

[1] `http://www.w3.org/2005/Incubator/ssn/`

advantages and disadvantages from each approach. We will also review existing Linked Stream Data processing system, and provide a comparison among them, regarding the design choices and overall performance.

We will start by introducing a running example that will be used throughout the tutorial (Section 1.1). The example highlights the potential benefits in integrating stream data with other sources, and it also shows what are the challenges faced when designing a Linked Stream Data Processing system. And we mentioned earlier, Linked Stream Data is closely related to Linked Data and stream processing. We assume that the attendees of this tutorial are familiar with the research in Linked Data, and in Section 2 we will focus providing a background on the fundamentals of stream processing that also applies to Linked Stream Data. We will show what are the basic models and techniques, how continuous semantics are represented, the diverse operators and processing optimisation techniques, and how issues like time management and memory overflow are handled. In the Linked Stream Data processing (Section 3), we first show the formalisation to represent the data and the continuous queries and the query operators needed. We then moved into the architecture and system design of Linked Stream Data processing engines, showing the different possible approaches and design choices. We highlight the state of the art systems in Linked Stream Data processing, and show how each of them implement the different architectures. We also provide a performance comparison in terms of query processing times and scalability. The end of this tutorial is dedicated to a short discussion of the current challenges in open problems (Section 4).

1.1 Running Example

Inspired by the experiments of Live Social Semantics [4,109], we use the following running example through the rest of the tutorial. The scenario focuses on the data integration problem between data streams given by a *tracking system* and static data sets. Similar to several real deployments in Live Social Semantics, the tracking system is used for gathering the relationship between real-world identifiers and physical spaces of conference attendees. These tracking data can then be correlated with non-stream datasets, like online information about the attendees (social network, online profiles, publication record, etc). The benefits of correlating these two sources of information are manifold. For instance, conference rooms could be automatically assigned to the talks, based on the talk's topic and the number of people that might be interested in attending it (based on their profile). Conference attendees could be notified about fellow co-authors in the same location. A service that suggest which talks to attend, based on profile, citation record, and distance between talks locations can be designed. These are just a few examples.

For the tracking service in our example, attendees of a conference wear RFID tags that constantly stream the location in a building, i.e which room/section/area they currently are. Each reading streamed from the RFID tags to RFID readers has two properties, the *tagid* and the *signal strength.*

The static datasets include data about the conference attendees and metadata about the conference building. Each attendee has a profile containing his personal information and the *tagid* given to him. The profile has also data links to the attendee's publication records in DBLP. The data about the conference building includes information such as location name, description, layout, and connection between location/rooms.

The data from streams and static datasets of this example can not be modelled by a relational data model because the data items involved are hosted in different storages. Because they do not have predefined schema and identification scheme, the integration across data sources is not easily done. Therefore, traditional DSMS can not be directly used. Thanks to the Linked Data model, heterogeneous data items can be presented as a unified data model with public vocabularies and global identifiers, i.e, URIs. To enable the seamless data integration between data stream and static data represented in Linked Data model, a stream processing that can integrate Linked Stream Data and Linked Data has to be provided.

2 Basic Concepts and Techniques for Stream Processing

2.1 Data Stream Models

A *data stream* is an unbounded, continuously arriving sequence of timestamped *stream elements*. The stream elements may arrive in some orders [114] or out of order with explicit timestamps [80]. The stream elements are continuously pushed by external stream sources, and their arrival might be unpredictable. As a result, the system processing data streams has no control over the order of the stream elements and the streaming rate. Therefore, it is only able to access stream elements sequentially in the order in which they arrive.

The most popular data model used for stream data is the relational model [8,30,2]. In the relational model, stream elements are relational tuples with a fixed schema. Stream elements can be modelled in an object-based model to classify the stream contents according to a type hierarchy. For example, Tribica [108] proposes hierarchical data types for representing Internet protocol layers for its network monitoring system. Another example of modelling data sources by objects is the COUGAR system for managing sensor data [24]. In COUGAR, each type of sensor is modelled as an abstract data type, whose interface consists of the supported signal processing methods. This model is also used in complex event processing (CEP) engines such as SASE [129,3], ZStream [88] and ESPER[2]. CEP is closely related to stream processing, but its focus is more on making sense of events by deriving high-level knowledge, or complex events from lower level events [43], rather than modelling and processing time-dependent information . On top of that, many dynamic applications are built upon large network infrastructures, such as social networks, communication networks, biological networks and the Web. Such applications create data that can be naturally modelled as graph streams, in which edges of the underlying graph are

[2] `http://esper.codehaus.org/`

received and updated sequentially in a form of a stream [16,47,36,133,20]. However, most of the work in Linked Stream Data reused operators and notations relevant to relation model, this paper will focus on the relational model. The terms tuple and stream element might be used alternatively in the following sections.

2.2 Continuous Semantics

A continuous query is issued once and run continuously, incrementally producing new results over time. Its inputs are one or more append-only data streams and zero or more relations. The continuous semantics of a query Q is defined by the result it returns each time instant t, denoted as Q(t). Q is *monotonic* if $Q(t) \subseteq Q(t'), \forall t \leq t'$. [8] formalised the semantics of monotonic queries and proposed how to continuously evaluate them. For the non-monotonic queries, their semantics and execution mechanisms are addressed in [60,73,77,53].

Intuitively, a continuous query provides answers at any point in time, taking into account all the data that has arrived so far. This data is commonly in the form of relations used as inputs of relational algebras. Therefore, two types of continuous query algebras based on relational counterparts have been proposed. The first one is the *stream-to-stream* algebra that was employed in defining semantics of Streaming SPARQL[23]. In a stream-to-stream algebra, each operator consumes one or more streams (and zero or more relations) and incrementally produces an output stream [35,74].

The second type is the *mixed algebra* [8,48]. Mixed algebra includes three sets of operators: *stream-to-relation operators* which produce a relation from a stream (e.g., sliding windows), *relation-to-relation operators* which produce a relation from one or more input relations (i.e., the standard relational algebraic operators), and *relation-to-stream operators* which produce a stream from a relation. Conceptually, at every time tick, an operator converts its input to relations, computes any new results, and converts the results back a stream that can be consumed by the next operator. Since the converted relations change over time, a natural way of switching back to a stream is to report the difference between the current result and the result computed one time tick ago. This is similar to computing a set of changes (insertions and/or deletions) required to update a materialised view. The mixed algebra is used in formalising semantics of C-SPARQL [17], SPARQL$_{stream}$ [27] and CQELS [94]. There also logical algebras for CEP[44,26] inspired by relational algebra and logic programming. However, CEP algebras have not been used in current Linked Stream Data processing systems, therefore they are out of the scope of this tutorial.

Stream-to-Stream Operator. A Stream-to-Stream operator continuously calls one-time queries in native SQL over physical or logical streams to produces results to a derived stream. These operators are specified by common SQL constructions such as SELECT, FROM, WHERE and GROUP BY. In [74], the window specification is defined by extending the FROM clause. Other logical standard operators are defined similar to relational algebras.

Stream-to-Relation Operator. A stream-to-relation operator takes a stream S as input and produces a relation R as output with the same schema as S. For example, CQL [8] introduced three operators: time-based, tuple-based, and partitioned windows.

1. *Time-based sliding windows.* A time-based sliding window on a stream S takes a time-interval T as a parameter and is specified by following the reference to S with [Range T]. Intuitively, a time-based window defines its output relation over time by sliding an interval of size T time units capturing the latest portion of an ordered stream. More formally, the output relation R of "S [*Range* T]" is defined as:

   $$R(t) \; = \; \{s \mid \langle s, t' \rangle \in S \wedge (t' {\leq} t) \wedge (t' {\geq} max\{t - T + 1, 0\})\} \qquad (1)$$

   When $T = 0$, R(t) consists of tuples obtained from elements with timestamp t, denoted with syntax "S [NOW]". And when $T = \infty$, R(t) consists of tuples obtained from elements with timestamps up to t, given with the SQL-99 syntax "S [Range Unbounded]".

   Example 1. "RFIDstream [Range 60 seconds]" represents a time-based sliding window of 60 seconds over a stream of RFID readings. At any time instant t, $R(t)$ will contains a bag of RFID readings from previous 60 seconds.

2. *Tuple-based windows.* A tuple-based sliding window on a stream S takes a positive integer N as a parameter and is specified by following the reference to S in the query with [Rows N]. At any given point in time, the window contains the last N tuples of S. More formally, let s_1, s_2, . . . , denote the tuples of S in increasing order of their timestamps, breaking ties arbitrarily. The output relation R of "S [Rows N]" is defined as:

   $$R(t) \; = \; \{s_i \mid max\{1, n(t) - N + 1\} \leq i \leq n(t)\} \qquad (2)$$

   where $n(t)$ denotes the size of S at time t, i.e., the number of elements of S with timestamps $\leq t$.

   Example 2. Similar to example 1, "RFIDstream [ROWS 1]" returns the last RFID reading from the stream at any time instant.

3. *Partitioned windows.* A partitioned sliding window is applied to a stream S with two parameters: a positive number N for number of rows and a subset of attributes of S, {A_1,...,A_k}. The CQL syntax for partitioned windows is [Partition S By A_1,...,A_k Rows N]. Similar to SQL Group By, this window operator logically partitions stream S into sub-streams based on equality of attributes A_1,...,A_k. The parameter N is used to compute the tuple-based windows from those sub-streams.

   Example 3. "RFIDstream [Partition By *tagid* ROWS 1]" partitions the RFIDstream into a collection of sub-streams based on *tagid* and gets the latest readings from each sub-stream. This query can be used to find where the last locations of all the RFID tags were detected.

The windows might have a *slide* parameter for specifying the granularity at which window slides. The formal definition can be found in [8]. Additionally, fixed windows and value-based windows were proposed in [108] and [100], respectively.

Relation-to-Relation Operator. The relation-to-relation operators are introduced to employ relational operators. Therefore, they have the same semantics as the counterparts. However, CQL introduces the *instantaneous relations* that are relations computable at a specific time instant t, e.g. outputs from stream-to-relation operators.

Example 4. From relations output from the sliding window in example 1, the projection(SELECT) and duplicate elimination (*Distinct*) operators can be applied as showed in the query below :

```
SELECT Distinct tagid
FROM RFIDstream [RANGE 60 seconds]
```

Relation-to-Stream Operator. A Relation-to-stream operator produces a stream from a relation. A relation-to-stream operator takes a relation R as input and produces a stream S as output with the same schema as R. For instance, CQL introduced three relation-to-stream operators: *Istream*, *Dstream*, and *Rstream.*

1. *Istream* (for "insert stream") applied to a relation R contains a stream element $\langle s, t\rangle$ whenever the tuple s is in R(t)−R($t-1$). Assuming R(-1) =$\emptyset$ for notational simplicity, it is defined as follow:

$$Istream(R) = \bigcup_{t \geq 0} (R(t) - R(t-1)) \times \{t\}) \tag{3}$$

   Example 5. Consider the following CQL query for creating a new stream by filtering another stream:

   ```
   SELECT Istream(*)
   FROM RFIDstream [RANGE Unbounded]
   WHERE signalstrength>=85
   ```

   This query continuously applies the Unbounded window to the RFIDstream, then filter all the RFID readings that have signal strength values equal or greater than 85.

2. *Dstream* (for "delete stream") applied to relation R contains a stream element $\langle s, t\rangle$ whenever the tuple s is in R(t-1) − R(t). Formally:

$$Dstream(R) = \bigcup_{t > 0} (R(t-1) - R(t)) \times \{t\}) \tag{4}$$

Example 6. Bellow is a query to detect when a person leaves the building/-conference by tracking the RFID tag of that person. The sliding windows keep all the readings in last 60 seconds, and the Dstream operator will report the tagid that was not detected in last 60 seconds but had been detected before.

```
SELECT Dstream(tagid)
FROM RFIDstream [60 seconds]
```

3. *Rstream* (for "relation stream") applied to relation R contains a stream element $\langle s, t\rangle$ whenever the tuple s is in R at time t . Formally:

$$Rstream(R) = \bigcup_{t\geq 0}(R(t) \times \{t\} \tag{5}$$

Example 7. The query in Example 5 can be written with Rstream as following:

```
SELECT Rstream(*)
FROM RFIDstream [NOW]
WHERE signalstrength>=85
```

2.3 Time Management

The described semantics for continuous queries in a data stream system typically assumes timestamps on data stream elements, thus, a consistent semantics for multiple streams and updatable relations relies on timestamps. To achieve semantic correctness, the DSMS query processor usually needs to process tuples in increasing timestamp order. That is, the query processor should never receive a stream element with a lower timestamp than any previously received ones. According to [106], there are two common type of timestamps: *system timestamp* and *application timestamp.*

The system timestamp is issued to stream elements when entering to the DSMS using the DSMS's system time. The application timestamp is given by the data sources before sending the stream elements to the DSMS. As an example of application timestamps, consider monitoring sensor readings to correlate changes in temperature and pressure. Each tuple consists of a sensor reading and an application timestamp affixed by the sensor, denoting the time at which that reading was taken. In general there may not be any relationship between the time at which the reading is taken (the application timestamp) and the time at which the corresponding stream tuple reaches the DSMS (the system timestamp).

The recommended architecture for time management is shown in Figure 1 [106]. Since stream tuples may not arrive at the DSMS in increasing timestamp order, there is an *input manager* that buffers tuples until they can be moved to

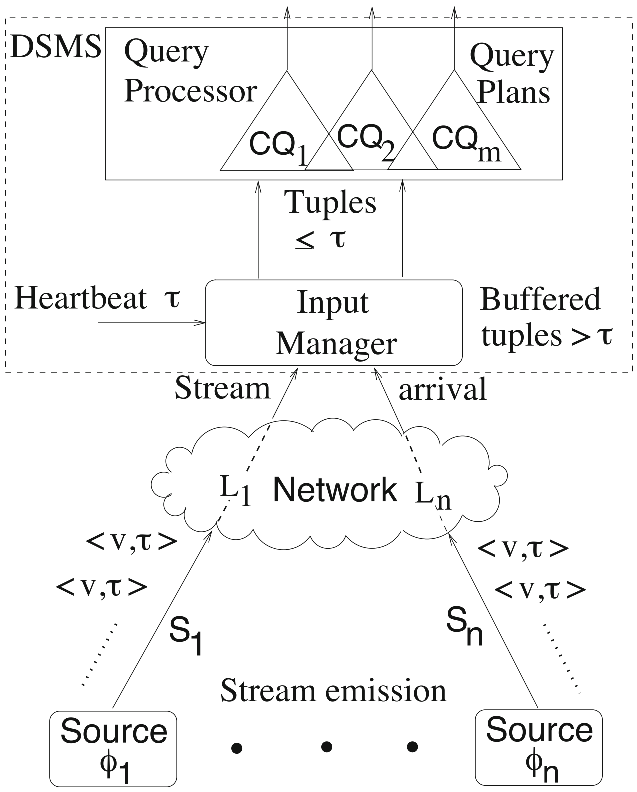


Fig. 1. Recommended architecture for time management

the query processor in a proper order. The decision when a tuple can be moved to the query processor is based on *heartbeats.* A heartbeat for a set of streams S_1, S_2, . . . , S_n at wall-clock time c is defined as the maximum application timestamp t such that all tuples arriving on S_1, S_2, . . . , S_n after time c must have timestamp $> t$.

Along with the solution for generating heartbeats of [106], there are also other solutions to deal with time management in some other data stream management projects like Aurora [2], Niagara [34], TelegraphCQ [30], and Gigascope [35]. The operators of Aurora have a slack parameter to deal with out-of-order streams. Essentially, the slack parameter instructs its operator to wait a certain period of time before closing each window. In Niagara, the proposed solution is based on punctuations [114]. Punctuations define arbitrary predicates over streams. Thus, heartbeats can be thought of special types of punctuations. A more detail comparison of heartbeat solution with others can be found in [106].

2.4 Implementation of Operators over Streams

For continuously executing operators over streams, there are two execution strategies: *eager re-evaluation* and *periodic re-evaluation* [52]. The eager re-evaluation generates new results every time new stream element arrives. However, it might be infeasible in situations where streams have high arrival rate. The periodic evaluation is to execute the query periodically [9,32]. In this case, sliding windows may be advanced and queries re-evaluated periodically with a specified frequency [2,34,76,79,83,104,30,131]. A disadvantage of periodic query evaluation is that results may be stale if the frequency of re-executions is lower than the frequency of the update. One way to stream new results after each new item arrives is to bound the error caused by delayed expiration of tuples in the oldest sub-window. However, long delays might be unacceptable in streaming applications that must react quickly to unusual patterns in data.

The continuous query evaluation needs to handle two types of events: arrivals of new stream elements and expirations of old stream elements [49]. The actions taken upon arrival and expiration vary across operators [60,121]. A new stream element may generate new results (e.g., join) or remove previously generated results (e.g., negation). Furthermore, an expired stream element may cause a removal of one or more items from the result (e.g., aggregation) or an addition of new items to the result (e.g., duplicate elimination and negation). Moreover, operators that must explicitly react to expired elements (by producing new results or invalidating existing results) have to perform state purging eagerly (e.g., duplicate elimination, aggregation, and negation), whereas others may do so eagerly or lazily (e.g., join).

The new stream element arrivals are obviously triggered by stream sources. However, there should be mechanisms to signal the events of expirations. There two techniques to signal the expirations, *negative tuple* [8,49,54] and *direct timestamp* [8,49]. In the negative tuple technique, every window in the query is equipped with an operator to explicitly generate a negative tuple for every expiration on the arrivals of new stream elements. For queries without negation operations, the direct expiration timestamps on each tuple can be used to initiate the expirations.

The re-evaluation of the stateless operators is straightforward because the new stream elements can be processed on-the-fly. For instance, Figure 2(a) shows how the selection operation over stream S1 works [54]. The duplicate-preserving projection and union operators are also examples of stateless operators. On contrary to stateless operators, a stateful operator needs to probe the previous processing states in every re-evaluation. Maintaining processing states is done differently on each operators. In following, we will discuss how to deal with stateful operators such as window join, aggregation, duplication elimination and non-motonic operators.

Window Join Operators. In a sliding window join, newly arrived tuples on one of the inputs probe the state of the other inputs. Additionally, expired tuples are removed from the state [52,61,62,71,124]. Expiration can be done periodically,

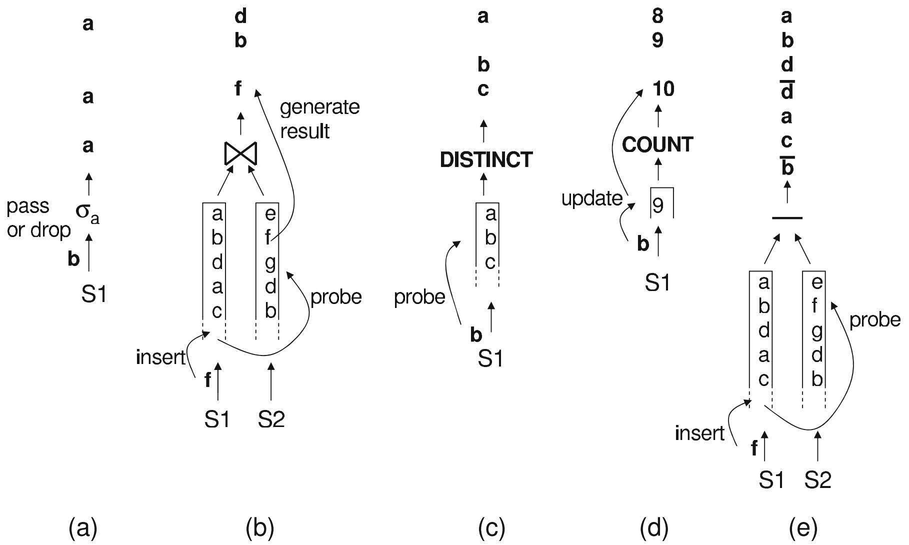


Fig. 2. Operator implementations : selection (a), window join (b), duplication elimination (c), aggregation (d), and negation (e)

provided that old tuples can be identified and skipped during processing. Figure 2(b) is an example of a non-blocking pipeline join [127,59,41,85,120,61,89,19,111]. It stores the input streams (S1 and S2), possibly in the form of hash tables, and for each arrival on one of the inputs, the state of the other input is probed to generate new results. Joins of more than two streams and joins of streams with static relations are straightforward extensions. In the former, for each arrival on one input, the states of the other inputs are probed [120]. In the latter, new arrivals on the stream trigger the probing of the relation.

Aggregation Operators. Aggregation over a sliding window updates its result when new tuples arrive and when old tuples expire. In many cases, the entire window needs to be stored in order to account for expired tuples, though selected tuples may sometimes be removed early if their expiration is guaranteed not to influence the result. For example, when computing MAX, tuples with value v need not be stored if there is another tuple in the window with value greater than v and a younger timestamp (see, e.g., [82,110] for additional examples of reducing memory usage in the context of skyline queries and [90] in the context of top-k queries). Additionally, in order to enable incremental computation, the aggregation operator stores the current answer (for distributive and algebraic aggregates) or frequency counters of the distinct values present in the window (for holistic aggregates). For instance, computing COUNT entails storing the current count, incrementing it when a new tuple arrives, and decrementing it when a tuple expires. Note that, in contrast to the join operator, expirations must be dealt with immediately so that an up-to-date aggregate value can be returned right away. Non-blocking aggregation [64,122,77] is shown in Figure 2(d). When a new tuple arrives, a new result is appended to the output stream if the aggregate value has changed. The new result is understood to replace previously reported results. GROUP BY may be thought of as a general case of aggregation, where a newly arrived tuple may produce new output if the aggregate value for its group has changed.

The time and space requirements of the aggregation operator depend upon the type of function being computed [57]. An aggregate f is distributive if, for two disjoint multi-sets X and Y, $f(X \cup Y) = f(X) \cup f(Y)$. Distributive aggregates, such as COUNT, SUM, MAX and MIN, may be computed incrementally using constant space and time (per tuple). For instance, SUM is evaluated by storing the current sum and continually adding to it the values of new tuples as they arrive. Moreover, f is algebraic if it can be computed using the values of two or more distributive aggregates using constant space and time (e.g., AVG is algebraic because $AVG = SUM/COUNT$). Algebraic aggregates are also incrementally computable using constant space and time. On the other hand, f is holistic if, for two multi-sets X and Y , computing $f(X \cup Y)$ requires space proportional to the size of $X \cup Y$. Examples of holistic aggregates include TOP-k, QUANTILE, and COUNT DISTINCT. For instance, multiplicities of each distinct value seen so far may have to be maintained in order to identify the k most frequent item types at any point in time. This requires $\Omega(n)$ space, where n is the number of stream tuples seen so far—consider a stream with $n - 1$ unique values and one of the values occurring twice.

Duplicate Elimination Operators. Duplicate elimination, illustrated in Figure 2(c), maintains a list of distinct values already seen and filters out duplicates from the output stream. As shown, when a new tuple with value b arrives, the operator probes its output list, and drops the new tuple because a tuple with value b has already been seen before and appended to the output stream.

Duplicate elimination over a sliding window may also produce new output when an input tuple expires. This occurs if a tuple with value v was produced on the output stream and later expires from its window, yet there are other tuples with value v still present in the window [60]. Alternatively, duplicate elimination may produce a single result tuple with a particular value v and retain it on the output stream so long as there is at least one tuple with value v present in the window [8,53]. In both cases, expirations must be handled eagerly so that the correct result is maintained at all times.

Non-monotonic Operators. As non-monotonic query patterns like negation are parts of SPARQL 1.1, the non-monotonic operators over streams are desirable. Indeed, these operators are possible if previously reported results can be removed when they no longer satisfy the query. This can be done by appending corresponding negative tuples to the output stream [60,8]. Negation of two sliding windows, $S1 - S2$, may produce negative tuples (e.g., arrival of a $S2$-tuple with value v causes the deletion of a previously reported result with value v), but may also produce new results upon expiration of tuples from $S2$ (e.g., if a tuple with value v expires from $S2$, then a $S1$-tuple with value v may need to be appended to the output stream [60]). An example is shown in Figure 2(e), where a tuple with value d was appended to the output because generated on the output stream upon subsequent arrival of an $S2$-tuple with value d.

2.5 Handling Memory Overflow

To handle memory overflow, the secondary storage must be used in the query operators. The XJoin operator [116] was introduced to address the memory overflow in binary window joins by spilling some partitions of inputs to disk. XJoin extends the Symmetric Hash Join (SHJ) [66,128] to use less memory by allowing parts of hash tables to be moved to a secondary storage. The MJoin operator [120] generalised the XJoin operator to deal with multiway stream joins. MJoin maximises the output rate of the multi-join operator by efficiently coordinating the spilling processes instead of spilling the inputs to disk randomly without considering the values in their join attributes.

If the second storage is used for storing the sliding window, then an index might be used to improve the performance. However, the index introduces the cost of maintenance especially in the context of frequent updates. In order to reduce the index maintenance costs, it is desirable to avoid bringing the entire window into memory during every update. This can be done by partitioning the data to localise updates (i.e., insertions of newly arrived data and deletion of tuples that have expired from the window) to a small number of disk pages. For example, if an index over a sliding window is partitioned chronologically [45,104], then only the youngest partition incurs insertions, while only the oldest partition needs to be checked for expirations (the remaining partitions in the "middle" are not accessed). A similar idea of grouping objects by expiration time appears in [85] in the context of clustering large file systems, where every file has an associated lifetime. However, the disadvantage of chronological clustering is that records with the same search key may be scattered across a very large number of disk pages, causing index probes to incur prohibitively many disk I/Os. One way to reduce index access costs is to store a reduced (summarised) version of the data that fit on fewer disk pages [31], but this does not necessarily improve index update times. In order to balance the access and update times, a wave index has been proposed that chronologically divides a sliding window into n equal partitions, each of which is separately indexed and clustered by search key for efficient data retrieval [104]. However, the access time of this approach is slower because multiple sub-indices are probed to obtained the answer. To accelerate the access time [55] proposed the doubly partitioned indices to simultaneously partition the index on insertion an expiration times.

2.6 Optimisation

The continuous query is usually issued in a declarative language like CQL, then it is translated to a logical query plan. In some DSMSs like Aurora, the logical query plan can be composed by the user. The query optimisation might be applied at *logical level* by rewriting the plan to improve efficiency, called *algebraic optimisation.* The common rewriting rules such as reordering selection before joins and evaluating inexpensive predicates before complex ones were used in [15,51]. Particularly for continuous queries, [8] proposed rules on window-based operators such as commutative rules on time-based and count-based windows.

The logical query plan needs to be scheduled to be executed in the execution engine with a physical plan composed of concrete physical operators. As the data arrives to the engine continuously, the DSMS scheduler can use different equivalent physical plans to execute a logical query plan during the life time of the query. Traditional DBMSs use selectivity information and available indices to choose efficient physical plans (e.g., those which require the fewest disk accesses). However, this cost metric does not apply to (possibly approximate) continuous queries, where processing cost per-unit-time is more appropriate [71]. Alternatively, if the stream arrival rates and output rates of query operators are known, then it may be possible to optimise for the highest output rate or to find a plan that takes the least time to output a given number of tuples [117,119,111]. Finally, quality-of-service metrics such as response time may also be used in DSMS query optimisation [2,18,97,98].

Optimisation by rescheduling physical query plans are similar to those used in relational databases, e.g., re-ordering a sequence of binary joins in order to minimise a particular cost metric. There has been some work in join ordering for data streams in the context of the rate-based model [119,120]. Furthermore, adaptive re-ordering of pipelined stream filters is studied in [13] and adaptive materialisation of intermediate join results is considered in [14]. Note the prevalence of the notion of adaptivity in query rescheduling; operators may need to be re-ordered on-the-fly in response to changes in system conditions. In particular, the cost of a query plan may change for three reasons: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream [10].

Initial efforts on adaptive query plans include mid-query re-optimisation [70] and query scrambling, where the objective was to pre-empt any operators that become blocked and schedule other operators instead [5,118]. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddies approach [10] performs scheduling of each tuple separately by routing it through the operators that make up the query plan. Thereby, the operators of the query plan are dynamically re-ordered to adapt to the current system conditions. This is driven by tuple routing policies that attempt to find which operators are fast and selective, and those operators are executed first. This approach was applied to continuous queries in [32,87] and was evaluated in [38]. The extended version for multi-way joins can be found in [113,95]. On top of that, it was also extended to consider semantics information such as attribute correlations during routing [21]). For distributed settings [112], the queue length is considered as a third factor for tuple routing strategies.

To achieve the adaptivity, the processing engine has to deal with some overheads. First overhead is having to reroute each tuple separately. The next overhead is migrating internal states stored in some operators from current query plan to new query plan that has new arrangement of operators. The issue of state migration across query plans were studied in [39,134]. More details on adaptive query processing may be found in [56,12,40].

When there are multiple continuous queries registered, memory and computing can be shared to optimise the overall processing. For selection queries, a possible multi-query optimisation is to index the query predicates and store auxiliary information in each tuple that identify which queries it satisfies [28,130,32,76,81]. When a new tuple arrives for processing, its attribute values are extracted and matched against the query index to see which queries are satisfied by this tuple. Data and queries may be thought of as duals, in some cases reducing query processing to a multi-way join of the query predicate index and the data tables [49, 168]. Indexing range predicates is discussed in [130,81], whereas a predicate index on multiple attributes is presented in [78,81].

Besides, memory usage may be reduced by sharing internal data structures that store operators' states [37,42,132]. Additionally, in the context of complex queries containing stateful operators such as joins, computation may be shared by building a common query plan [34]. For example, queries belonging to the same group may share a plan, which produces the union of the results needed by the individual queries. A final selection is then applied to the shared result set and new answers are routed to the appropriate queries. An interesting tradeoff appears between doing similar work multiple times and doing too much unnecessary work; techniques that balance this tradeoff are presented in [33,75,123]. For example, suppose that the workload includes several queries referencing a join of the same windows, but having a different selection predicate. If a shared query plan performs the join first and then routes the output to the appropriate queries, then too much work is being done because some of the joined tuples may not satisfy any selection predicate (unnecessary tuples are being generated). On the other hand, if each query performs its selection first and then joins the surviving tuples, then the join operator cannot be shared and the same tuples will be probed many times. Finally, sharing a single join operator among queries referencing different window sizes is discussed in [62].

2.7 Scheduling

After the query optimiser chooses a physical query plan, the query engine starts to execute it. Different from pull-based operator of DBMS, DSMS operators consume data pushed into the plan by the sources. At any point during an execution, there may be many tuples in the input and inter-operator queues. Queues allow sources to push data into the query plan and operators to retrieve data as needed [2,10,8,87,86]; see [67] for a discussion on calculating queue sizes of streaming relational operators using classical queueing theory.

Each operator consumes data from its input queue(s) to return outputs to upper queues. The DSMS scheduler must determine which data item in which queue to process next. A round-robin strategy can be used to execute each operator in round-robin until it has processed all the data items in its queue(s). Another simple technique, first-in-first-out, is to process one data item at a time in order of arrival, such that each item is processed to completion by all the operators in the plan. This execution strategy ensures good respond time, however, scheduling one tuple at a time may incur too much overhead.

Another scheduling strategy is to allocate a time slice to each operator, during which the operator extracts tuples from its input queue(s), processes them in timestamp order, and deposits output tuples into the next operator's input queue. The time slice may be fixed or dynamically calculated based upon the size of an operator's input queue and/or processing speed. A possible improvement could be to schedule one or more tuples to be processed by multiple operators at once. In general, there are several possible conflicting criteria involved in choosing a scheduling strategy, among them queue sizes in the presence of bursty stream arrival patterns [11], average or maximum latency of output tuples [29,68,92] and average or maximum delay in reporting the answer relative to the arrival of new data [102]. Additionally, [119,29,101] proposed strategies for scheduling operators to achieve low latency by producing highest output rates.

3 Linked Stream Data Processing

3.1 Linked Stream Data

The success of Linked Data in terms of flexibility and data interoperability has uncountable efforts in both transforming existing data and generating new one, following the Linked Data principles [22], in many different areas. The field of ubiquitous computing was not an exception: with so many heterogeneous sensor data sources, data integration is currently a difficult and labor-intensive tasks, and because of that applications involving sensor sources are still limited to specific domains. Applying the Linked Data principles here would enable a vast range of new, real-time applications in the areas of smart cities, green IT, e-health, to name a few.

There is one aspect common to this research area which is not covered in the original concept of Linked Data, which is data usually is output in the form of streams. With the increasing demand for real-time applications, stream data is also becoming popular in sources other than sensors. In the Web for instance, services delivering real-time information, like Facebook or Twitter, are increasingly popular.

Linked Stream Data [99] was introduced in order to bridge this gap between stream and Linked Data, and to facilitate data integration among stream sources and also between streams and other static sources. It follows the standards of Linked Data, and it is usually represented as an extension of RDF—the most popular standard for Linked Data representation. Assigning URIs to RDF streams not only allows to access the RDF streams as materialised data but also enables the query processor to treat the RDF streams as RDF nodes, such that other SPARQL query patterns can be directly applied.

The extensions made to the standard RDF account for handling the temporal aspects of stream data. For that, new data models and query language have been proposed, which we will discuss next. Designing and implementing a Linked Stream Data processor has many challenges, some close to the challenges in data stream processing highlighted in Section 2, and some particultar to Linked Stream Data. We provide an extensive and comparative analysis of the current state of the art in Linked Stream Data processing, their different design choices and solutions to address the different issues. We also present a discussion about the remaining open problems.

3.2 Formalisation

This section will show how to formalise the data model for RDF streams and RDF datasets in continuous context. From the formal data model, the semantics of the query operators will be defined.

Data Model. The Linked Stream Data is modelled by extending the definitions of RDF nodes and RDF triples [93]. Let I, B, and L be *RDF nodes* which are pair-wise disjoint infinite sets of Information Resource Identifiers (IRIs), blank nodes and literals, and $IL = I \cup L$, $IB = I \cup B$ and $IBL = I \cup B \cup L$ be the respective unions. A triple $(s, p, o) \in IB \times I \times IBL$ is an *RDF triple.*

Stream elements of Linked Stream Data are represented as RDF triples with temporal annotations. A temporal annotation of an RDF triple can be an interval-based [84] or point-based [58] label. An interval-based label is a pair of timestamps which commonly are natural numbers representing for logical time. The pair of timestamps, $[start, end]$, is used to specify the interval that the RDF triple is valid. For instance, $\langle$*:John :at :office,[7,9]*$\rangle$ represents that John was at the office from 7 to 9. The point-based label is a single natural number representing the time point that the triple was recorded or received. In the previous example, the triple$\langle$*:John :at :office*$\rangle$ might be continuously recorded by a tracking system, so three temporal triples are generated $\langle$*:John :at :office,7*$\rangle$, $\langle$*:John :at :office,8*$\rangle$, $\langle$*:John :at :office,9*$\rangle$.

The point-based label looks redundant and less efficient in comparison to interval-based one. Furthermore, the interval-based label is more expressive than point-based one because the later is a special case of the former, i.e., when *start* = *end*. Streaming SPARQL[23] uses interval-based labels for representing its physical data stream items and EP-SPARQL[6] uses them for representing triple-based events. However, point-based label is more practical for streaming data sources because triples are generated unexpectedly and instantaneously. For example, a tracking system detecting people at an office can easily generate a triple with a timestamp whenever it receives the reading from the sensors.

Otherwise, the system has to buffer the readings and do some further processing in order to generate the interval that the triple is valid. Moreover, the instantaneity is vital for some applications that need to process the data as soon as it arrives in the system. For instance, an application that notifies where John is should be triggered at the time point 7 other than waiting until time 9 to report that he was in the office from 7 to 9. Point-based label is supported in C-SPARQL[17], SPARQL$_{stream}$[27] and CQELS [94] . Without lost of generality, we will use point-based label for defining stream elements for Linked Stream Data, called *RDF stream.*

An *RDF stream* S is a bag of elements $\langle (s,p,o) : [t] \rangle$, where (s,p,o) is an *RDF triple* and t is a timestamp. $S^{\leq t}$ denotes the bag of elements in S with timestamps $\leq t$, i.e.,

$$S^{\leq t} = \{\langle (s,p,o) : [t'] \rangle \in S \mid t' \leq t\} \tag{6}$$

To enable the integration of stream data with non-stream data, the concept of an RDF dataset has to be included in the data model. As applications on stream data can be run for days, months or years, the changes of RDF dataset during the query lifetime need to be modelled. Similar to *instantaneous relation* of CQL, we extend the definition of a static RDF dataset to *instantaneous RDF dataset* as following.

An *RDF dataset at timestamp* t, denoted by $G(t)$, is a set of *RDF triples* valid at time t, called *instantaneous RDF dataset.* An *RDF dataset* is a sequence $G = [G(t)], t \in \mathbb{N}$, ordered by t. When it holds that $G(t) = G(t+1)$ for all $t \geq 0$, we call G a *static RDF dataset* and denote $G^s = G(t)$.

Query Operators. The primitive operation on RDF stream and instantaneous RDF dataset is *pattern matching* which is extended from the *triple pattern* of SPARQL semantics [93]. Each output of a pattern matching operator consists of a mapping which is defined as partial functions. Let V be an infinite set of variables disjoint from IBL, a partial function μ from V to IBL denoted as

$$\mu : V \longmapsto IBL. \tag{7}$$

The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are compatible, denoted as $\mu_1 \cong \mu_2$ if :

$$\mu_1 \cong \mu_2 \iff \forall x \in dom(\mu_1) \cap dom(\mu_2) \Rightarrow \mu_1(x) = \mu_2(x) \tag{8}$$

For a given triple pattern τ, the triple obtained by replacing variables within τ according to μ is denoted as $\mu(\tau)$.

Three primitive operators on RDF dataset and RDF stream, namely, *triple matching pattern operator*, *window matching operator* and *sequential operator*, are introduced in current Linked Stream Data processing systems. Similar to SPARQL, the triple matching pattern operator on an instantaneous RDF dataset at timestamp t is defined as

$$[\![P,t]\!]_G = \{\mu \mid dom(\mu) = var(P) \land \mu(P) \in G(t)\} \tag{9}$$

where $P \in (I \cup V) \times (I \cup V) \times (IL \cup V)$.

A window matching operator $[\![P,t]\!]_S^\omega$ over an RDF stream S is then defined by extending the operator above as follows:

$$[\![P,t]\!]_S^\omega = \{\mu \mid dom(\mu) = var(P) \wedge \langle \mu(P) : [t'] \rangle \in S \wedge t' \in \omega(t)\} \tag{10}$$

where $\omega(t) \colon \mathbb{N} \to 2^{\mathbb{N}}$ is a function mapping a timestamp to a (possibly infinite) set of timestamps. This gives the flexibility to choose between the different window modes introduced in Section 2.2. For example, a time-based sliding window of size T defined in Equation 1 can be expressed as $\omega_{RANGE}(t) = \{t' \mid t' \leq t \wedge t' \geq \max(0, t-T)\}$, and a window that extracts only events happening at the current time corresponds to $\omega_{NOW}(t) = \{t\}$.

A triple-based event matching pattern like the *sequential operator* SEQ of EP-SPARQL, denoted as $\Rightarrow^t$, can be defined by using above operator notations as following :

$$\begin{aligned}[\![P_1 \Rightarrow^t P_2]\!]_S^\omega = \{\mu_1 \cup \mu_2 \mid \mu_1 \in [\![P_1,t]\!]_S^\omega \wedge \mu_2 \in [\![P_2,t]\!]_S^\omega \wedge \mu_1 \cong \mu_2 \\ \wedge \langle \mu_1(P) : [t_1'] \rangle \in S \wedge \langle \mu_2(P) : [t_2'] \rangle \in S \wedge t_1' \leq t_2'\}\end{aligned} \tag{11}$$

Other temporal relations introduced in [129,3,7,6] can be formalised similarly to the sequential operator.

As an output of primitive operators are a mapping set. The join, union, difference and left outer-join operators over mapping sets Ω_1 and Ω_2 are defined as following:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \cong \mu_2\} \tag{12}$$

$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu_1 \in \Omega_1 \vee \mu_2 \in \Omega_2\} \tag{13}$$

$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \neg \exists \mu' \in \Omega_2, \mu' \cong \mu\} \tag{14}$$

$$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2) \tag{15}$$

3.3 Query Languages

To define a descriptive query language, firstly, the basic query patterns need to be introduced to express the primitive operators, i.e, triple matching, window matching, and sequential operators. Then the composition of basic query patterns can be expressed by AND, OPT, UNION and FILTER patterns of SPARQL. These patterns are corresponding to operators in the Equations (12)-(15).

In [17], an aggregation pattern is denoted as $A(v_a, f_a, p_a, G_a)$, where v_a is the name of the new variable, f_a is the name of the aggregation function to be evaluated, p_a is the parameter of f_a, and G_a is the set of the grouping variables. The evaluation of [[A]] is defined by a mapping $\mu_a : V \mapsto IBL$, where

$dom(\mu_a) = v_a \cup G_a$; also $||\mu_a|| = ||G_a|| + ||v_a|| = ||G_a|| + 1$ where $||\mu||$ is the cardinality of $dom(\mu)$. This extension fully conforms to the notion of compatibility between mappings. Indeed, $\mu_a \notin dom(P)$ and, therefore, calling μ_p the mapping that evaluate [[P]] and $\mu_p \cong \mu_a$.

The result of the evaluation produces a table of bindings, having one column for each variable $v \in dom(\mu)$. $\mu_{(i)}$ can be referred as a specific row in this table, and to a specified column is given by $\mu[v]$. The $i-th$ binding of v is therefore $\mu_{(i)}[v]$.

The values to be bound to a variable v_a are computed as $\forall i \in [1, ||\mu||], \mu_{(i))}[v_a] = f_a(p_a, \mu[G_a])$, where $f(p_a, [G_a])$ is the evaluation of the function $f_a \in (SUM, COUNT, AVG, MAX, MIN)$ with parameters p_a over the groups of values in $\mu[G_a]$. The set of groups of values in $\mu[G_a]$ is made of all the distinct tuples $\mu_{(i)}[G_a]$ i.e., the subset of the mapping $\mu[G_a]$ without duplicate rows.

From above query patterns, let P_1, P_2 and P be basic query patterns or composite ones. A declarative query can be composed recursively using the following rules:

1. $[[P_1 \text{ AND } P_2]] = [[P_1]] \bowtie [[P_2]]$
2. $[[P_1 \text{ OPT } P_2]] = [[P_1]] \mathbin{⟕} [[P_2]]$
3. $[[P_1 \text{ UNION } P_2]] = [[P_1]] \cup [[P_2]]$
4. $[[P \text{ FILTER } R]] = \{\mu \in [[P]] | \mu \vDash R\}$, where $\mu \vDash R$ if μ satisfies condition R.
5. $[[P \text{ AGG } A]] = [[P]] \mathbin{⟕} [[A]]$

The above query pattern construction enable the extension of the SPARQL grammar for continuous query. Streaming SPARQL extended SPARQL 1.0 grammar[3] by adding the *DatastreamClause* and a clause for window as showed in following EBNF grammar rules :

SelectQuery	::=	'SELECT' ('DISTINCT' | 'REDUCED'')?'(*Var* | '*')(*DatasetClause** | **DatastreamClause***) *WhereClause SolutionModifier*
DatastreamClause	::=	'FROM' (*DefaultStreamClause* | *NamedStreamClause*)
DefaultStreamClause	::=	'STREAM' *SourceSelector Window*
NamedStreamClause	::=	'NAMED' 'STREAM' *SourceSelector Window*
GroupGraphPattern	::=	{ *TriplesBlock*? ((*GraphPatternNotTriples* | *Filter*)'.'? *TriplesBlock*?)*(**Window**)?)}
Window	::=	(*SlidingDeltaWindow* | *SlidingTupleWindow* | *FixedWindow*)
SlidingDeltaWindow	:=	'WINDOW' 'RANGE' *ValSpec* 'SLIDE' *ValSpec*?
FixedWindow	:=	'WINDOW' 'RANGE' *ValSpec* 'FIXED'
SlidingTupleWindow	::=	'WINDOW' 'ELEMS' INTEGER
ValSpec	::=	INTEGER | *Timeunit*?
Timeunit	:=	('MS' | 'S' | 'MINUTE' | 'HOUR' | 'DAY' | 'WEEK')

Similarly, the C-SPARQL language is extended from SPARQL 1.1's grammar[4] by adding FromStrClause and a clause for windows as following:

[3] http://www.w3.org/TR/rdf-sparql-query/#grammar
[4] http://www.w3.org/TR/sparql11-query/#grammar

$$
\begin{aligned}
\mathit{FromStrClause} &\rightarrow \text{`FROM' [`NAMED'] `STREAM'} \ \mathit{StreamIRI} \ \text{`[RANGE'} \ \mathit{Window} \ \text{`]'} \\
\mathit{Window} &\rightarrow \mathit{LogicalWindow} \mid \mathit{PhysicalWindow} \\
\mathit{LogicalWindow} &\rightarrow \mathit{Number} \ \mathit{TimeUnit} \ \mathit{WindowOverlap} \\
\mathit{TimeUnit} &\rightarrow \text{`d'} \mid \text{`h'} \mid \text{`m'} \mid \text{`s'} \mid \text{`ms'} \\
\mathit{WindowOverlap} &\rightarrow \text{`}\mathit{STEP}' \ \mathit{Number} \ \mathit{TimeUnit} \mid \text{`TUMBLING'}
\end{aligned}
$$

Also extending SPARQL 1.1's grammar, CQELS language is built by adding a query pattern to apply window operators on RDF Streams into the *GraphPatternNotTriples* clause.

$$
\begin{aligned}
\mathit{GraphPatternNotTriples} ::= \ & \mathit{GroupOrUnionGraphPattern} \mid \mathit{OptionalGraphPattern} \\
& \mid \mathit{MinusGraphPattern} \mid \mathit{GraphGraphPattern} \\
& \mid \textbf{StreamGraphPattern} \mid \mathit{ServiceGraphPattern} \mid \mathit{Filter} \mid \mathit{Bind}
\end{aligned}
$$

Assuming that each stream is identified by an IRI as identification, the **StreamGraphPattern** clause is then defined as follows.

$$
\begin{aligned}
\textbf{StreamGraphPattern} &::= \text{`STREAM' `['} \ \mathit{Window} \ \text{`]'} \ \mathit{VarOrIRIref} \ \text{`\{'} \mathit{TriplesTemplate} \text{`\}'} \\
\mathit{Window} &::= \mathit{Range} \mid \mathit{Triple} \mid \text{`NOW'} \mid \text{`ALL'} \\
\mathit{Range} &::= \text{`RANGE'} \ \mathit{Duration} \ (\text{`SLIDE'} \ \mathit{Duration} \mid \text{`TUMBLING'})? \\
\mathit{Triple} &::= \text{`TRIPLES' INTEGER} \\
\mathit{Duration} &::= (\text{INTEGER `d'} \mid \text{`h'} \mid \text{`m'} \mid \text{`s'} \mid \text{`ms'} \mid \text{`ns'})^{+}
\end{aligned}
$$

where *VarOrIRIRef* and *TripleTemplate* are clauses for the *variable/IRI* and *triple template* of SPARQL 1.1, respectively. *Range* corresponds to a time-based window while *Triple* corresponds to a triple-based window. The keyword *SLIDE* is used for specifying the sliding parameter of a time-based window, whose time interval is specified by *Duration.* In special cases, the [NOW] window is used to indicate that only the triples at the current timestamp are kept and [ALL] window is used to indicate that all the triples will be kept in the window.

The following example describes 5 queries involving linked stream data, then, we show how to express these queries in above query languages.

Example 8. To enhance the conference experience in our running scenario, each participant would have access to the following services, which can all be modelled as continuous queries:

(Q1) Inform a participant about the name and description of the location he just entered.
(Q2) Notify two people when they can reach each other from two different and directly connected (from now on called *nearby*) locations.
(Q3) Notify an author of his co-authors who have been in his current location during the last 5 seconds.
(Q4) Notify an author of the editors of a paper of his and that have been in a nearby location in the last 15 seconds.
(Q5) Count the number of co-authors appearing in nearby locations in the last 30 seconds, grouped by location.

The grammars of Streaming SPARQL and C-SPARQL are similar, the URI of the stream is defined after keywords "FROM STREAM" and the triple patterns are placed in the WHERE clause. Consider the query Q1, and assume that URI of the RFID stream is `http://deri.org/streams/rfid`. An RDF dataset has to specified to integrate the metadata of the building. For example, the named graph `http://deri.org/floorplan/` can be used as the RDF dataset to correlate with the RFID stream. Because, in Streaming SPARQL [23], using RDF dataset is not clearly described. In C-SPARQL, query Q1 is expressed as followings.[5]

```
SELECT ?locName ?locDesc
FROM STREAM <http://deri.org/streams/rfid>  [NOW]
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  ?person lv:detectedat ?loc. ?loc lv:name ?locName. ?loc lv:desc ?locDesc
  ?person foaf:name ‘‘$Name$’’.
}
```

Query Q1:C-SPARQL

In CQELS, the streams are specified after the keywords “STREAM” Streaming to declare the STREAM clause inside a WHERE clause. Thereby, the query Q1 is expressed in CQELS as followings:

```
SELECT ?locName ?locDesc
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [NOW] {?person lv:detectedat ?loc}
  GRAPH  <http://deri.org/floorplan/>
     {?loc lv:name ?locName. ?loc lv:desc ?locDesc}
  ?person foaf:name ‘‘$Name$’’.
}
```

Query Q1-CQELS

```
CONSTRUCT {?person1 lv:reachable ?person2}
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [NOW] {?person1 lv:detectedat ?loc1}
  STREAM <http://deri.org/streams//rfid>  [RANGE 3s]
                      {?person2 lv:detectedat ?loc2}
  GRAPH  <http://deri.org/floorplan/>   {?loc1    lv:connected  ?loc2}
}
```

Query Q2-CQELS

Queries Q2, Q3, Q4, and Q5 need to declare two different windows on the RFID stream, then join these windows with other data. However, grammars of Streaming SPARQL and C-SPARQL only allow to specify one window on one stream URI, therefore, these four queries can not be expressed directly in single queries in Streaming SPARQL and C-SPARQL languages. In C-SPARQL, it is possible to get around this issue by creating two separate logical streams from the same stream. These new streams will then be used to apply two windows needed in those 4 queries. On the other hand, the STREAM clause of CQELS allows expressing Q2-Q4 as single queries as below.

[5] For the sake of space we omit the PREFIX declarations of lv, dc, foaf, dcterms and swrc.

```
SELECT ?coAuthName
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [TRIPLES 1]
    {?auth   lv:detectedat ?loc}
  STREAM <http://deri.org/streams/rfid> [RANGE 5s]
    {?coAuth lv:detectedat ?loc}
  { ?paper dc:creator ?auth.     ?paper  dc:creator ?coAuth.
    ?auth  foaf:name ‘‘$Name$’’. ?coAuth foaf:name  ?coAuthorName}
  FILTER (?auth != ?coAuth)
}
```

Query Q3-CQELS

```
SELECT ?editorName
WHERE {
  STREAM <http://deri.org/streams/rfid> [TRIPLES 1]
    {?auth   lv:detectedat ?loc1}
  STREAM <http://deri.org/streams/rfid> [RANGE 15s]
    {?editor lv:detectedat ?loc2}
  GRAPH  <http://deri.org/floorplan/> {?loc1   lv:connected  ?loc2}
  ?paper      dc:creator  ?auth.   ?paper  dcterms:partOf ?proceeding.
  ?proceeding swrc:editor ?editor. ?editor foaf:name      ?editorName.
  ?auth       foaf:name  ‘‘$Name$’’
}
```

Query Q4-CQELS

```
SELECT ?loc2 ?locName count(distinct ?coAuth) as ?noCoAuths
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [TRIPLES 1]
    {?auth   lv:detectedat ?loc1}
  STREAM <http://deri.org/streams/rfid> [RANGE 30s]
    {?coAuth lv:detectedat ?loc2}
  GRAPH  <http://deri.org/floorplan/>
     {?loc2 lv:name ?locName. loc2 lv:connected ?loc1}
  {
?paper dc:creator ?auth. ?paper dc:creator ?coAuth.?auth foaf:name ‘‘$Name$’’
  }
  FILTER (?auth != ?coAuth)
}
GROUP BY ?loc2 ?locName
```

Query Q5-CQELS

To support runtime discovery for multiple streams that share the same triple patterns, CQELS supports expressing stream URIs as variable. For instance, the triples that match the pattern ⟨*?person lv:detectedat ?loc*⟩ can be found in different streams generated by RFID readers, Wifi-based tracking systems, GPSs, etc. For instance, to query all the streams nearby a location (filtered by triple pattern ⟨*?streamURI lv:nearby* : *DERI_Building*⟩ in metadata) that can give such triples, the following query expressed in CQELS can be used.

```
SELECT ?name ?locName
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM ?streamURI [NOW] {?person lv:detectedat ?loc}
  GRAPH  <http://deri.org/floorplan/>
  { ?streamURI lv:nearby :DERI_Building. ?loc lv:name ?locName.
   ?person foaf:name ?name. }
```

CQELS query with variable on stream's URI

In some queries, the built-in functions on timestamps on stream elements are needed. Among the aforementioned query language, C-SPARQL [17] and EP-SPARQL [6] enable expressing queries with functions to manipulate the timestamps. The timestamp of a stream element can be retrieved and bound to a variable using a built-in timestamp function. The timestamp function has two arguments. The first is the name of a variable, introduced in the WHERE clause and bound by pattern matching with an RDF triple of that stream. The second (optional) is the URI of a stream, that can be obtained through SPARQL GRAPH clause. The function returns the timestamp of the RDF stream element producing the binding. If the variable is not bound, the function is undefined, and any comparison involving its evaluation has a non-determined behaviour. If the variable gets bound multiple times, the function returns the most recent timestamp value relative to the query evaluation time. Following is a example of EP-SPARQL using function *getDurration*() to filter triples that have timestamps in the duration of 30 minutes.

```
CONSTRUCT {?person2 lv:comesAfter ?person1}
{
SELECT ?person1 ?person2
WHERE {
 {?person1 lv:detectedat ?loc}
 SEQ
  {?person2 lv:detectedat ?loc}
  }
 FILTER (getDURATION()<"P30m"^^xsd:duration)
```

EP-SPARQL query with built-in time function

3.4 System Design and Architecture

The architecture design of current available systems that support continuous query processing over RDF data streams and RDF data can be classified into two categories. The first one is a "whitebox" architecture as depicted in Figure 3. From the semantics formalised in Section 3.2, this architecture needs to implement the physical operators such as sliding windows, join, and triple pattern matching. Such operators can be implemented using techniques and algorithms described in Section 2.4. To consume data, these operators use access methods which may employ data structures such as B-Tree+, hashtable, or triple-based indexes for fast random data access to RDF datasets or RDF streams. To execute a declarative query in SPARQL-like language, the optimiser has to translate the it to a logical query plan, then find an optimal execution plan based on the corresponding operator implementations. The execution plan is then executed by the Executor. As the continuous query is executed continuously, the Optimiser can re-optimise it to find a new execution plan to adapt to the changes in the data and computing environment.

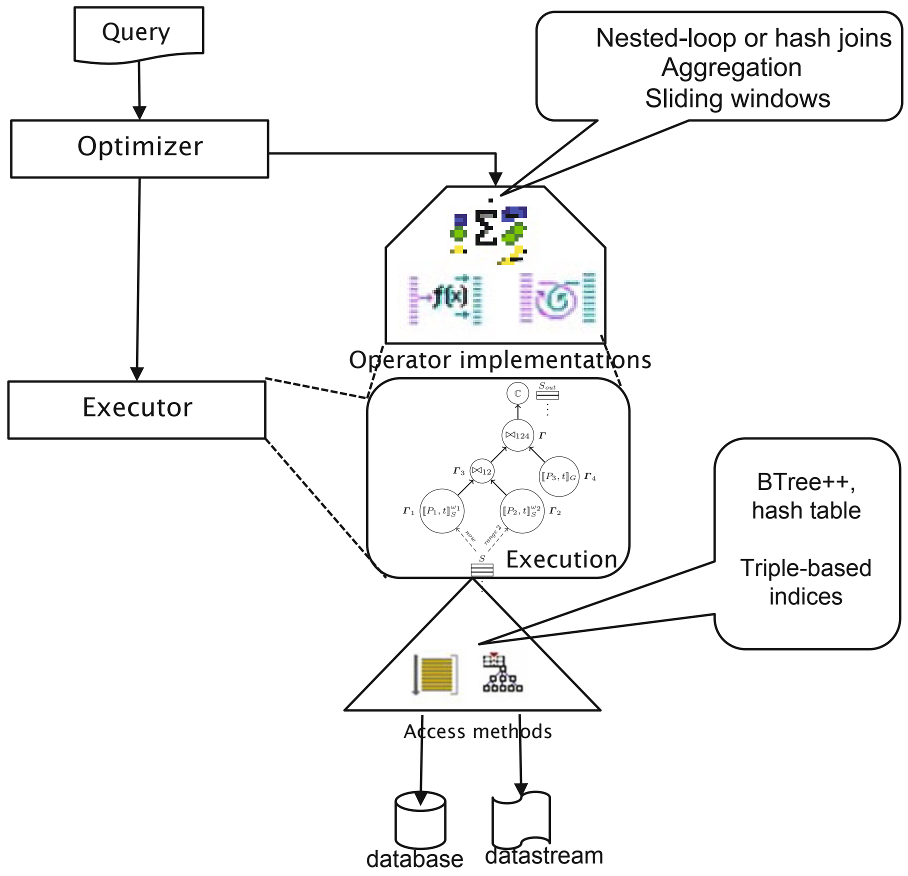


Fig. 3. The whitebox architecture

In the whitebox architecture, the query engine needs to implement all the components of a DSMS. To avoiding implementing most of those components, an alternative is the "blackbox" architecture shown in Figure 4. The blackbox architecture recommends to use other systems as sub-components. Such sub-components are used to delegate the sub-processings needed. Commonly, the chosen systems for sub-components have a whitebox architecture but they can be interfaced as blackboxes with different query languages and different input data formats. Hence, the systems designed using blackbox approach need a Query Rewriter, an Orchestrator and a Data Transformer. The Query rewriter is used

Table 1. Current systems that implement the blackbox architecture

	Data transformation	Query rewriter	Underlying systems
C-SPARQL	RDF↔Relation	C-SPARQL→SPARQL&CQL	SPARQL engine& STREAM or ESPER
SPARQL$_{stream}$	Relation→RDF	SPARQL$_{stream}$ →SNEEql	SNEE engine
EP-SPARQL	RDF→logic facts	EP-SPARQL→logic rules	Prolog engines

to rewrite SPARQL-like query languages to a query and sub-queries that underlying systems can understand. The Orchestrator is used to orchestrate the execution process by externalising the processing to sub-systems with the rewritten sub-queries. In some cases, the Orchestrator also includes some components for correlating and aggregating partial results returned from blackbox systems if they support. The Data Transformer is responsible for converting input data to the compatible formats of the Access methods used in sub-components. The Data Transformer also has to transform the output from the underlying systems to the format that the Orchestrator needs. Table 1 gives a summary of some properties of the systems using the blackbox architecture.

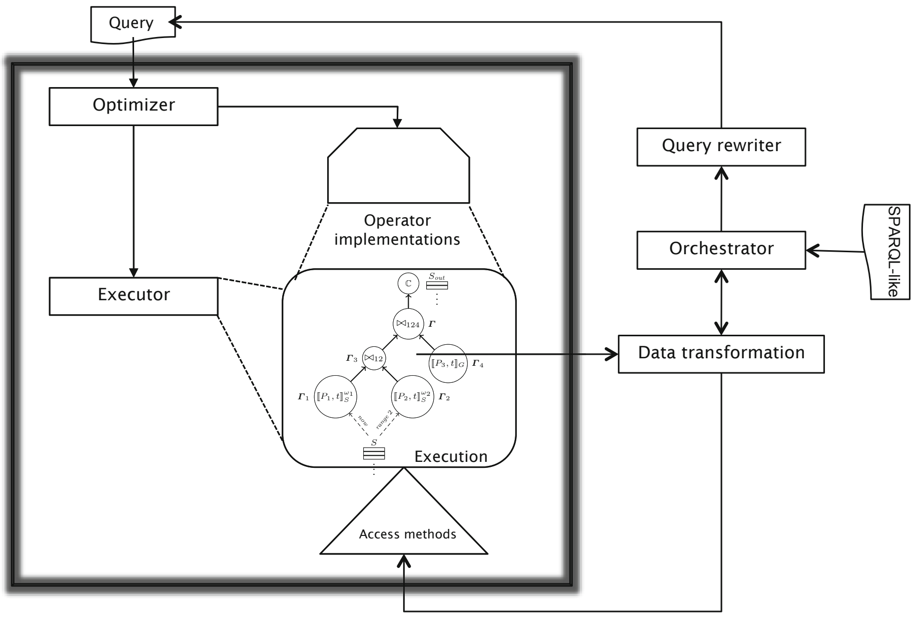


Fig. 4. Components of blackbox systems

By delegating the processing to available systems, building the system with the blackbox architecture takes less effort than using the whitebox one. However, such systems do not have full-control over the sub-components and suffers the

Table 2. System design choices

	Architecture	Access methods	Execution
Streaming SPARQL	whitebox	Physical RDF streams	Stream-to-Stream operators
C-SPARQL	blackbox	Relations&Views	CQL&SPARQL queries
EP-SPARQL	blackbox	Logic predicates	Logic programmes
SPARQL$_{stream}$	blackbox	Web services	SNEE queries
CQELS	whitebox	Native data structures	Adaptive physical operators

overhead of data transformation. To summarise, Table 2 gives the design choices of current available systems. In the next section, we describe each of these system in more details.

3.5 State-of-the-art in Linked Stream Data Processing

There are a few systems that currently support Linked Stream Data Processing. In this section, we provide an extensive and comparative analysis of these systems, their different design choice and solutions to address the different issues.

Streaming SPARQL. To show that query processing over RDF Streams is possible, Streaming SPARQL [23] proposed to extend the ARQ[6] query compiler to transform the continuous query to a logical algebra plan. This logical query plan is then compiled to a physical execution plan composed by physical query operators. Streaming SPARQL extends SPARQL 1.1 for representing continuous queries on RDF Streams. For implementing physical operators, Streaming SPARQL uses the approaches proposed by [74]. However, to the best of our knowledge, the implementation of the system is not publicly available. The Streaming SPARQL suggests to apply *algebraic optimisation* after translating a declarative query to a logical query plan.

C-SPARQL. C-SPARQL realises the blackbox architecture with the orchestration design as shown in Figure 5. In this architecture, the C-SPARQL engine uses a SPARQL plugin to connect to a SPARQL query engine to evaluate the static part of the query, i.e, the sub-queries involving the RDF datasets. For evaluating parts of the query relevant to streams and aggregates, the engine delegates the processing to an existing relational data stream management system. One limitation of this architecture is that aggregations can only be performed by the DSMS. A parser parses the C-SPARQL query and hands it over to the orchestrator. The orchestrator is the central component of C-SPARQL engine and translates the query in a static and dynamic part. The static queries are used to extract the static data from the SPARQL engine, while the dynamic queries is registered in the DSMS. The query initialisation is executed only once

[6] `http://incubator.apache.org/jena/documentation/query/`

when a C-SPARQL query is registered as the continuous evaluation is handled subsequently by the DSMS. Therefore, C-SPARQL does not allow updates in the non-stream data.

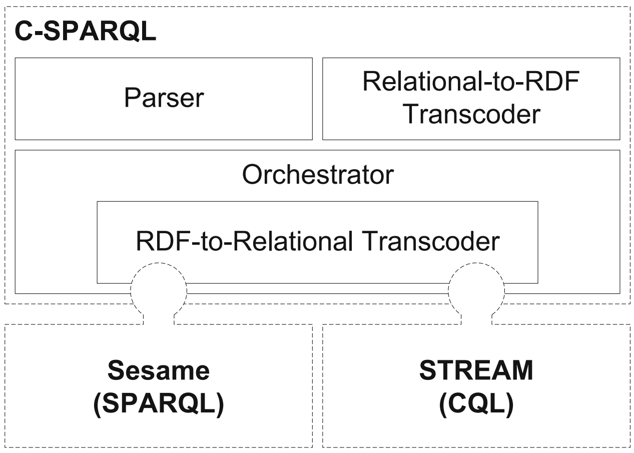


Fig. 5. Architecture of C-SPARQL

The evaluation process of the C-SPARQL engine is illustrated in Figure 6. The results returned from the SPARQL engine for the static part of the query are loaded into materialised relations as inputs for the DSMS. These relations together with RDF streams will be computed via cascading views created in CQL queries driven by the C-SPARQL query. The first views are sliding window views over RDF Streams. Then they are used to correlated with the static relations via join views. As C-SPARQL employs algebraic optimisation, it tries to filter the data as earlier as possible. Hence, it pushes filters, projections and aggregations by rewriting rules [105]. The order of the evaluation process illustrates how the views are created on top of others. The current version of C-SPARQL based on Jena and ESPER can be downloaded at `http://streamreasoning.org/download`

EP-SPARQL. EP-SPARQL uses the blackbox approach backed by a logic engine. It translates the processing into logic programs. The execution mechanism of EP-SPARQL is based on event-driven backward chaining (EDBC) rules, introduced in [7]. EP-SPARQL queries are compiled into EDBC rules, which enable timely, event-driven, and incremental detection of complex events (i.e., answers to EP-SPARQL queries). EDBC rules are logic rules, and hence can be mixed with other background knowledge (domain knowledge that is used for Stream Reasoning). Therefore, it provides a unified execution mechanism for Event Processing and Stream Reasoning which is grounded in Logic Programming (LP).

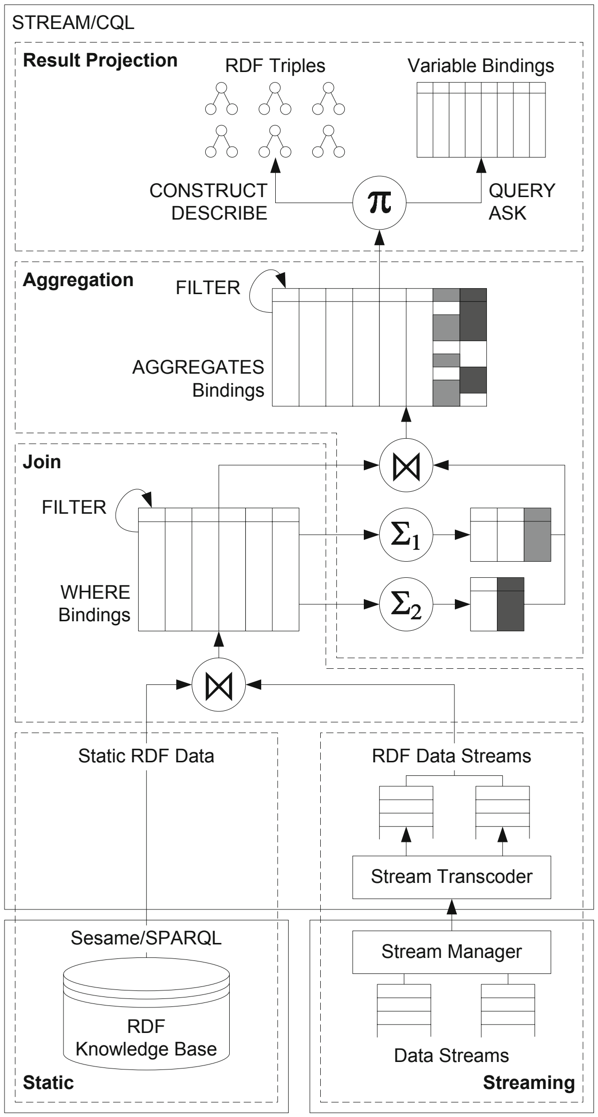


Fig. 6. Evaluation process of C-SPARQL

For encoding, EP-SPARQL uses a simple correspondence between RDF triples of the form $\langle s, p, o\rangle$ and Prolog predicates of the form $triple(s', p', o')$ so that s', p', and o' correspond to the RDF symbols s, p, and o, respectively. This means that whenever a triple $\langle s, p, o\rangle$ is satisfied, the corresponding predicate $triple(s', p', o')$ is satisfied too, and vice versa. Consequently, a time-stamped RDF triple $\langle\langle s, p, o\rangle, t_\alpha, t_\omega\rangle$ corresponds to a predicate $triple(s', p', o', T^{'}_\alpha, T^{'}_\omega)$ where $T^{'}_\alpha$ and $T^{'}_\omega$ denote timestamps. Timestamps are assigned to triples either by a triple source (e.g., a sensor or an application that generates triple updates) or by an EP-SPARQL engine. They facilitate time-related processing, and do not necessarily need to be kept once the stream has been processed (e.g., the pure RDF part could be persisted in a RDF triple store without timestamps). From RDF triples and time-stamped RDF encoded as logic predicates, the query operators like $SeqJoin$, $EqJoin$ and $Filters$ for EP-SPARQL query language are rewritten as rule patterns.

To enable the detection of more complex events, EP-SPARQL combines streams with background knowledge. This knowledge describes the context (domain) in which complex events are detected. As such, it enables the detection of real-time situations that are identified based on explicit data (e.g., events) as well as on implicit knowledge (derived from the background knowledge). The background knowledge may be specified as a Prolog knowledge base or as an RDFS ontology. This enables EP-SPARQL's execution model to have all relevant parts expressible in a unified (logic rule) formalism, and ultimately to reason over a unified space. The implementation of EP-SPARQL can be found at `http://code.google.com/p/etalis/`

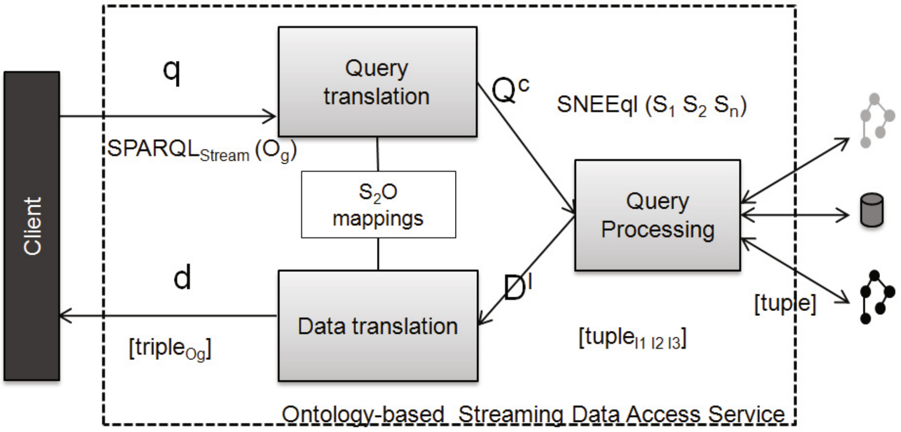


Fig. 7. Architecture of SPARQL$_{stream}$

SPARQL$_{stream}$. The SPARQL$_{stream}$ is designed to support the integration of heterogenous relational stream data sources. The SPARQL$_{stream}$ engine is a wrapping system that rewrites SPARQL$_{stream}$ query language to a relational continuous query language, e.g., SNEEql [46]. Its architecture is shown in Figure 7. In order to transform the SPARQL$_{stream}$ query, expressed in terms of the ontology, into queries in terms of the data sources, a set of mappings must be specified. These mappings are expressed in S_2O, an extension of the R_2O mapping language, which supports streaming queries and data, most notably window and stream operators.

After the continuous query has been generated, the query processing phase starts, and the evaluator uses distributed query processing techniques [72] to extract the relevant data from the sources and perform the required query processing, e.g. selection, projection, and joins. Note that query execution in sources such as sensor networks may include in-network query processing, pull or push based delivery of data between sources, and other data source specific settings. The result of the query processing is a set of tuples that the data translation process transforms into ontology instances. The current version of SPARQL$_{stream}$ can be found at `http://code.google.com/p/semanticstreams/wiki/SPARQL Stream`

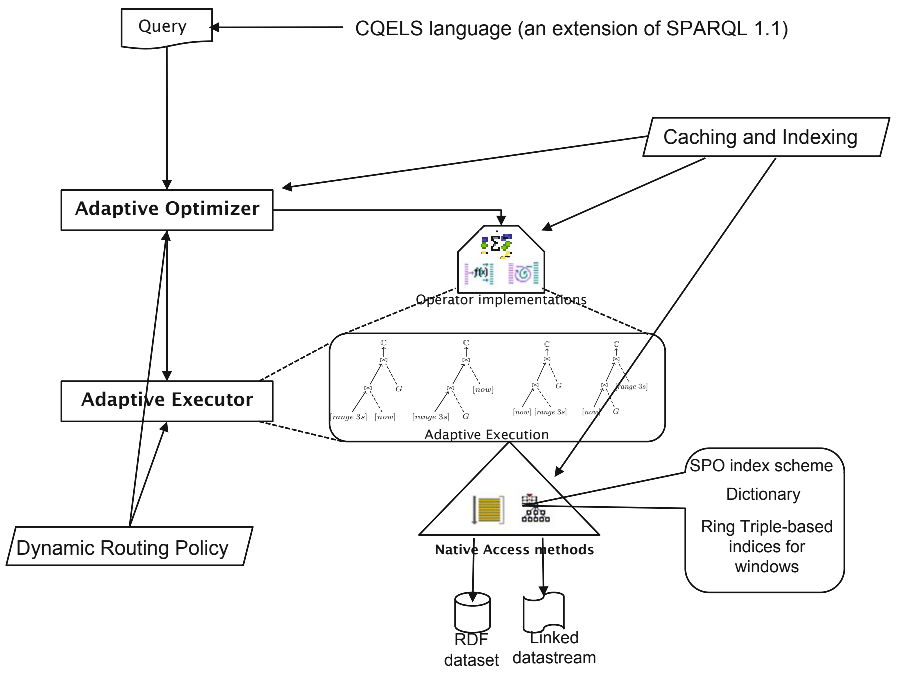


Fig. 8. Adaptive architecture of CQELS

CQELS. CQELS engine realised the whitebox approach with the adaptive execution mechanism as shown in Figure 8. The query executor is able to switch between equivalent physical query plans during the life time of the query. The adaptability of the query executor enables the query optimiser to re-schedule the next execution to achieve a better performance according to the changes of data and computing environment. As considering triples as the first-class data elements, CQELS engine employs both efficient data structures for sliding windows and triple storages to provide high-throughput native access methods on RDF dataset and RDF data streams. By adapting the implementations of the

Table 3. System comparison by features

	Input	Query language	Extras
Streaming SPARQL	RDF stream		
C-SPARQL	RDF Stream & RDF	TF	
EP-SPARQL	RDF Stream & RDF	EVENT,TF	Event operators
SPARQL$_{stream}$	Relational stream	NEST	Ontology-based mapping
CQELS	RDF Stream & RDF	VoS,NEST	Disk spilling

physical operators introduced in Section 2.4, CQELS is equipped with an adaptive caching mechanism [14] and indexing schema [63,125,50,130] to accelerate the processing in physical operators and access methods.

Similar to other systems, CQELS engine extended SPARQL 1.1 for continuous query, called CQELS language. However, CQELS language supports updates in RDF datasets as well as variables for stream identifiers. By expressing variables in stream identifiers, users can ask queries that continuously discover streams that can provided matched properties of interest as shown in section 3.3. Unlike other systems that only support in-memory processing, CQELS supports disk-based processing when the main memory is not enough to accommodate all data. By applying the techniques for memory overflow handling from Section 2.5, the size of the RDF data involved in the CQELS queries is not restricted to the capacity of the main memory. Details about CQELS's implementation can be found at `http://code.google.com/p/cqels/`

System Comparisons. Table 3 gives a comparison of all the systems according to features supported. Unlike the other systems, SPARQL$_{stream}$ takes relational stream as input other than RDF streams. C-SPARQL, EP-SPARQL and CQELS support correlation of RDF streams and RDF datasets. All systems extend SPARQL for stream processing, but each of their languages support some special patterns as shown in the column "query language". TF stands for support built-in time functions in the query patterns. EVENT corresponds to event-based patterns. NEST means that the engine support nested queries. VoS indicates that the query language allow using variable for the Stream's URI. The "Extras" column shows the extra features supported by the corresponding engines.

To compare the systems' execution mechanisms, Table 4 categorises the systems by architecture, re-execution strategy, how the engine schedules the execution and what type of optimisation is supported. Since Streaming SPARQL and C-SPARQL schedule the execution at logical level, the optimisation can only be done at algebraic level and statically. On the contrary, CQELS is able to choose the alternative execution plans composed from available physical implementations of operators, thus, the optimiser can adaptively optimise the execution at physical level. EP-SPARQL and SPARQL$_{stream}$ schedule the execution via a

Table 4. Comparisons by execution mechanism

	Architecture	Re-execution	Scheduling	Optimisation
Streaming SPARQL	whitebox	periodical	Logical plan	Algebraic & Static
C-SPARQL	blackbox	periodical	Logical plan	Algebraic & Static
EP-SPARQL	blackbox	eager	Logic program	Externalised
SPARQL$_{stream}$	blackbox	periodical	External call	Externalised
CQELS	whitebox	eager	Adaptive physical plans	Physical & Adaptive

Table 5. Average query execution time for single queries (in milliseconds)

	Q_1	Q_2	Q_3	Q_4	Q_5
CQELS	0.47	3.90	0.51	0.53	21.83
C-SPARQL	332.46	99.84	331.68	395.18	322.64
ETALIS	0.06	27.47	79.95	469.23	160.83

declarative query or a logic program, so, it completely externalises the optimisation to other systems.

The work in [94] provides the only experimental evaluation comparison available so far. It compares the average query execution time among C-SPARQL, ETALIS/EP-SPARQL and CQELS on five queries Q1,Q2,Q3,Q4 and Q5 introduced in section 3.3. For the RFID stream data, the evaluation uses the RFID-based tracking data streams provided by the Open Beacon community.[7] The data is generated from active RFID tags, the same hardware used in the Live Social Semantics deployment [4]. For user profiles, the simulated DBLP datasets generated from SP^{2}Bench [96] are used.

The table 5 shows the evaluation for a single query instance. The figure 9 reports the experiments on varying the size of static RDF dataset. The evaluation for multiple queries instances is showed in figure 10. In overall EP-SPARQL performed best for a single query pattern and small datasets and CQELS consistently outperformed to others for other cases.

4 Challenges in Linked Stream Data Processing

There are still a number of open challenge regarding Linked Stream Data processing. One of them is related to query optimisation. The continuous queries need to be optimised adaptively to cope with arbitrary changes in the stream characteristics and system conditions. The systems using the blackbox architecture only support the algebraic optimisation which is done in the query compiling phase. However, the adaptive optimisation can be applied separately at physical levels in each underlying sub-systems. This leads to the sub-optimal issue

[7] `http://www.openbeacon.org/`

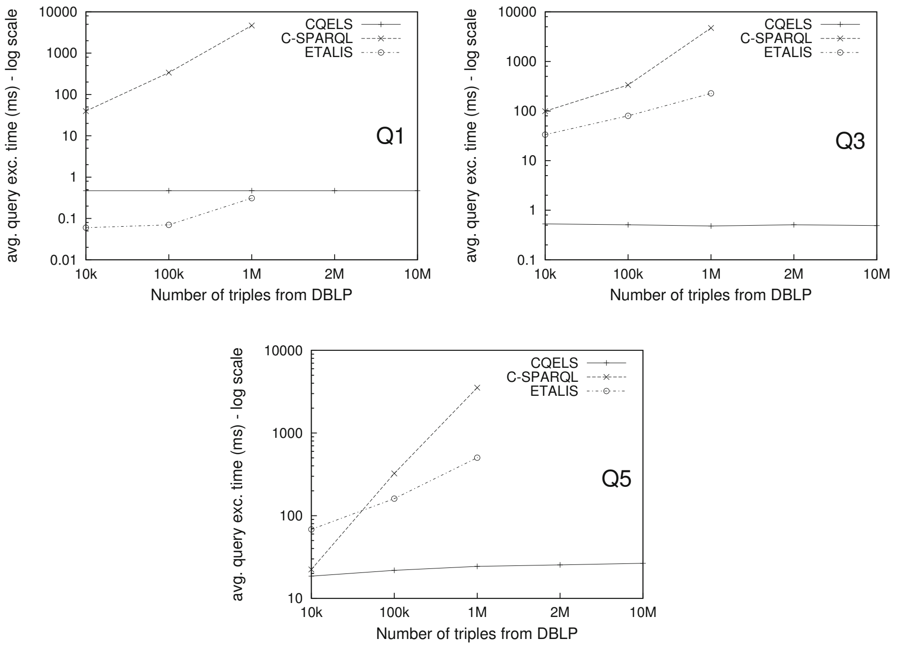
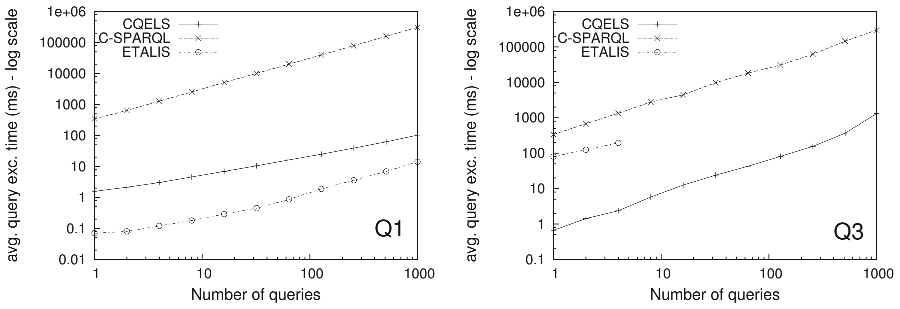


Fig. 9. Average query execution time for varying sizes of simulated DBLP dataset

in optimising the continuous query in blackbox systems. CQELS is currently the only system applying the optimisation techniques [12,40] of the adaptive query processing. In this case, the challenge in continuous query optimisation is similar to the challenge of optimisation in adaptive query processing. However, despite the several works in adaptive query processing, this area is still in an early stage [12]. Most of the proposed solutions are heuristic and rather ad-hoc. It is not straightforward to employ any such general optimisation technique for continuous queries over the triple-based data model. Furthermore, the continuous queries with the SPARQL-like patterns usually involve several self-joins of physical relational data structures for RDF triples. This might be too costly to re-optimise the query plan at running time in terms of search space and statistics maintenance. Besides, the joining properties of continuous queries over RDF streams might be highly correlated. In this case, estimating cardinalities of join queries is more difficult and complicate, therefore, maintaining the statistics of query plans at run-time might cost much more computing resources than the optimisation process can gain. Furthermore, Linked Stream Data makes the semantic of data understandable for the query processing engine, thus, the query optimiser can use the meaning of the data for optimising the query execution. For example, a person cannot be at two different times at two different places and hence the optimiser can use this knowledge to narrow down the stream sources and execute a query more efficiently. Nevertheless, to the best of our



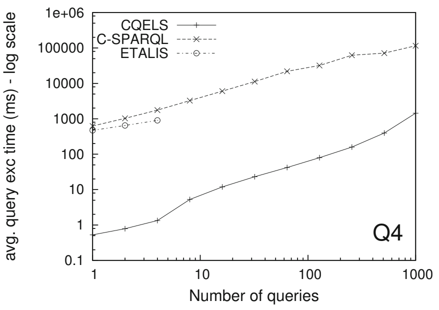


Fig. 10. Average query execution time when running multiple query instances

knowledge, there has not been any work on semantic-based optimisation for continuous query processing.

Distributing the stream processing across a cluster of machines to deal with high-speed data streams, big datasets and large amount of registered queries. There are two broad issues in distributed stream processing over Linked Stream Data and stream data in general: parallelising and distributing the system itself, and shifting some computation to the data sources [54]. One way to distribute the processing is to split the query plan across multiple processing nodes [1]. Another way is to partition the stream and let each node process a subset of the data to completion [69]. Systems like Borealis [1], StreamBase[8] and InfoSpheres Streams[9] support distributed processing. So far, there has not been any mature distributed systems for Linked Stream Data processing. However, such distributed DSMSs can be employed to build Linked Stream Data Processing engine using the blackbox architecture. Besides, there are distributed real-time data processing platforms such as S4[10], Storm[11] that can be used to scale Linked Data Stream processing on networked computing environments. For instance, [65] attempts to use S4 for scaling the stream reasoning. To overcome the challenge of

[8] `http://www.streambase.com/`
[9] `http://www-01.ibm.com/software/data/infosphere/streams/`
[10] `http://incubator.apache.org/s4/`
[11] `https://github.com/nathanmarz/storm`

scaling Linked Data Stream Processing on distributed computing setting, there will be interesting efforts and innovations in terms of experimenting, engineering and optimising to bring distributed Linked Data Stream engine in production.

5 Conclusion

This tutorial gave an overview about Linked Stream Data and how to build the processing engines on this data model. It described the basic requirements for the processing, highlighting the challenges that are faced, such as managing the temporal aspects and memory overflow. It presented the different architectures for Linked Stream Data processing engines, their advantages and disadvantages. The tutorial also reviewed the state of the art Linked Stream Data processing systems, and provided a comparison among them regarding the design choices and overall performance. It also discussed current challenges in Linked Stream Data processing that might be interesting in terms of researching and engineering in years to come.

Acknowledgments. This research has been supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-II), by the Irish Research Council for Science, Engineering and Technology (IRCSET), and by the European Commission under contract number FP720117287661 (GAMBAS).

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, pp. 277–289 (2005)
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal 12(2), 120–139 (2003)
3. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 147–160. ACM, New York (2008)
4. Alani, H., Szomszor, M., Cattuto, C., Van den Broeck, W., Correndo, G., Barrat, A.: Live Social Semantics. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 698–714. Springer, Heidelberg (2009)
5. Amsaleg, L., Franklin, M.J., Tomasic, A., Urhan, T.: Scrambling query plans to cope with unexpected delays. In: Proceedings of the Fourth International Conference on on Parallel and Distributed Information Systems, DIS 1996, pp. 208–219. IEEE Computer Society, Washington, DC (1996)

6. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: Ep-sparql: a unified language for event processing and stream reasoning. In: Proceedings of the 20th International Conference on World Wide Web, WWW 2011, pp. 635–644. ACM, New York (2011)
7. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: A Rule-Based Language for Complex Event Processing and Reasoning. In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 42–57. Springer, Heidelberg (2010)
8. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–142 (2006)
9. Arasu, A., Widom, J.: Resource sharing in continuous sliding-window aggregates. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 336–347. VLDB Endowment (2004)
10. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. SIGMOD Rec. 29(2), 261–272 (2000)
11. Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator scheduling in data stream systems. The VLDB Journal 13(4), 333–353 (2004)
12. Babu, S.: Adaptive query processing in the looking glass. In: Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, pp. 238–249 (2005)
13. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD 2004, pp. 407–418. ACM, New York (2004)
14. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive caching for continuous queries. In: Proceedings of the 21st International Conference on Data Engineering (ICDE 2005), pp. 118–129. IEEE Computer Society, Washington, DC (2005)
15. Babu, S., Srivastava, U., Widom, J.: Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. ACM Trans. Database Syst. 29(3), 545–580 (2004)
16. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002, pp. 623–632. Society for Industrial and Applied Mathematics, Philadelphia (2002)
17. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An execution environment for c-sparql queries. In: Proceedings of the 13th International Conference on Extending Database Technology, EDBT 2010, pp. 441–452. ACM, New York (2010)
18. Berthold, H., Schmidt, S., Lehner, W., Hamann, C.J.: Integrated resource management for data stream systems. In: Proceedings of the 2005 ACM Symposium on Applied Computing, SAC 2005, pp. 555–562. ACM, New York (2005)
19. Bertino, E., Catania, B., Wang, W.Q.: Xjoin index: Indexing xml data for efficient handling of branching path expressions. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004. IEEE Computer Society Press, Washington, DC (2004)
20. Bifet, A., Holmes, G., Pfahringer, B., Gavaldà, R.: Mining frequent closed graphs on evolving data streams. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2011, pp. 591–599. ACM, New York (2011)
21. Bizarro, P., Babu, S., DeWitt, D., Widom, J.: Content-based routing: different plans for different data. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 757–768. VLDB Endowment (2005)

22. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
23. Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL - Extending SPARQL to Process Data Streams. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 448–462. Springer, Heidelberg (2008)
24. Bonnet, P., Gehrke, J., Seshadri, P.: Towards Sensor Database Systems. In: Tan, K.-L., Franklin, M.J., Lui, J.C.-S. (eds.) MDM 2001. LNCS, vol. 1987, pp. 3–14. Springer, Heidelberg (2000)
25. Bouillet, E., Feblowitz, M., Liu, Z., Ranganathan, A., Riabov, A., Ye, F.: A Semantics-Based Middleware for Utilizing Heterogeneous Sensor Networks. In: Aspnes, J., Scheideler, C., Arora, A., Madden, S. (eds.) DCOSS 2007. LNCS, vol. 4549, pp. 174–188. Springer, Heidelberg (2007)
26. Bry, F., Eckert, M.: Rules for making sense of events: Design issues for high-level event query and reasoning languages. In: AI Meets Business Rules and Process Management, Proceedings of AAAI 2008 Spring Symposium, Stanford University/Palo Alto, California, USA, March 26. AAAI (2008)
27. Calbimonte, J.-P., Corcho, O., Gray, A.J.G.: Enabling Ontology-Based Access to Streaming Data Sources. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 96–111. Springer, Heidelberg (2010)
28. Carnes, C., Park, J.B., Vernon, A.: Scalable trigger processing. In: Proceedings of the 15th International Conference on Data Engineering, p. 266. IEEE Computer Society, Washington, DC (1999)
29. Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., Stonebraker, M.: Operator scheduling in a data stream manager. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 838–849. VLDB Endowment (2003)
30. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: Telegraphcq: Continuous dataflow processing for an uncertain world. In: First Biennial Conference on Innovative Data Systems Research, CIDR 2003 (2003)
31. Chandrasekaran, S., Franklin, M.: Remembrance of streams past: overload-sensitive management of archived streams. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 348–359. VLDB Endowment (2004)
32. Chandrasekaran, S., Franklin, M.J.: Psoup: a system for streaming queries over streaming data. The VLDB Journal 12(2), 140–156 (2003)
33. Chen, J., DeWitt, D.J., Naughton, J.F.: Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In: Proceedings of the 18th International Conference on Data Engineering, ICDE 2002, Washington, DC, USA (2002)
34. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for Internet databases. SIGMOD Rec. 29(2), 379–390 (2000)
35. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD 2003, pp. 647–651. ACM, New York (2003)

36. Das Sarma, A., Gollapudi, S., Panigrahy, R.: Estimating pagerank on graph streams. In: Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, pp. 69–78. ACM, New York (2008)
37. Denny, M., Franklin, M.J.: Predicate result range caching for continuous queries. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 646–657. ACM, New York (2005)
38. Deshpande, A.: An initial study of overheads of eddies. SIGMOD Rec. 33(1), 44–49 (2004)
39. Deshpande, A., Hellerstein, J.M.: Lifting the burden of history from adaptive query processing. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 948–959. VLDB Endowment (2004)
40. Deshpande, A., Ives, Z., Raman, V.: Adaptive query processing. Found. Trends Databases 1 (January 2007)
41. Dittrich, J.P., Seeger, B., Taylor, D.S., Widmayer, P.: Progressive merge join: a generic and non-blocking sort-based join algorithm. In: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB 2002, pp. 299–310. VLDB Endowment (2002)
42. Dobra, A., Garofalakis, M.N., Gehrke, J., Rastogi, R.: Sketch-Based Multi-query Processing over Data Streams. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 551–568. Springer, Heidelberg (2004)
43. Eckert, M., Bry, F., Brodt, S., Poppe, O., Hausmann, S.: A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed. In: Helmer, S., Poulovassilis, A., Xhafa, F. (eds.) Reasoning in Event-Based Distributed Systems. SCI, vol. 347, pp. 47–70. Springer, Heidelberg (2011)
44. Eckert, M., Bry, F., Brodt, S., Poppe, O., Hausmann, S.: Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra (CERA) and Its Application to XChange^{EQ}. In: Helmer, S., Poulovassilis, A., Xhafa, F. (eds.) Reasoning in Event-Based Distributed Systems. SCI, vol. 347, pp. 71–97. Springer, Heidelberg (2011)
45. Folkert, N., Gupta, A., Witkowski, A., Subramanian, S., Bellamkonda, S., Shankar, S., Bozkaya, T., Sheng, L.: Optimizing refresh of a set of materialized views. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 1043–1054. VLDB Endowment (2005)
46. Galpin, I., Brenninkmeijer, C.Y.A., Jabeen, F., Fernandes, A.A.A., Paton, N.W.: An architecture for query optimization in sensor networks. In: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, ICDE 2008, pp. 1439–1441. IEEE Computer Society, Washington, DC (2008)
47. Ganguly, S., Saha, B.: On Estimating Path Aggregates over Streaming Graphs. In: Asano, T. (ed.) ISAAC 2006. LNCS, vol. 4288, pp. 163–172. Springer, Heidelberg (2006)
48. Ghanem, T.M., Elmagarmid, A.K., Larson, P.A., Aref, W.G.: Supporting views in data stream management systems. ACM Trans. Database Syst. 35(1), 1:1–1:47 (2008)
49. Golab, L.: Sliding Window Query Processing over Data Streams. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, Canada (2006), `http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-27.pdf`

50. Golab, L., Garg, S., Özsu, M.T.: On Indexing Sliding Windows over Online Data Streams. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 712–729. Springer, Heidelberg (2004)
51. Golab, L., Johnson, T., Spatscheck, O.: Prefilter: predicate pushdown at streaming speeds. In: Proceedings of the 2nd International Workshop on Scalable Stream Processing System, SSPS 2008, pp. 29–37. ACM, New York (2008)
52. Golab, L., Özsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 500–511. VLDB Endowment (2003)
53. Golab, L., Özsu, M.T.: Update-pattern-aware modeling and processing of continuous queries. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 658–669. ACM, New York (2005)
54. Golab, L., Özsu, M.T.: Data stream management. Synthesis Lectures on Data Management, 1–73 (2010)
55. Golab, L., Prahladka, P., Ozsu, M.T.: Indexing time-evolving data with variable lifetimes. In: Proceedings of the 18th International Conference on Scientific and Statistical Database Management, SSDBM 2006, pp. 265–274. IEEE Computer Society, Washington, DC (2006)
56. Gounaris, A., Paton, N.W., Fernandes, A.A.A., Sakellariou, R.: Adaptive Query Processing: A Survey. In: Eaglestone, B., North, S.C., Poulovassilis, A. (eds.) BNCOD 2002. LNCS, vol. 2405, pp. 11–25. Springer, Heidelberg (2002)
57. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Min. Knowl. Discov. 1(1), 29–53 (1997)
58. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing Time into RDF. IEEE Transactions on Knowledge and Data Engineering 19, 207–218 (2007)
59. Haas, P.J., Hellerstein, J.M.: Ripple joins for online aggregation. SIGMOD Rec. 28, 287–298 (1999)
60. Hammad, M., Aref, W.G., Franklin, M.J., Mokbel, M.F., Elmagarmid, A.K.: Efficient execution of sliding-window queries over data streams (2003)
61. Hammad, M.A., Aref, W.G., Elmagarmid, A.K.: Stream window join: tracking moving objects in sensor-network databases. In: Proceedings of the 15th International Conference on Scientific and Statistical Database Management, SSDBM 2003, pp. 75–84. IEEE Computer Society, Washington, DC (2003)
62. Hammad, M.A., Aref, W.G., Elmagarmid, A.K.: Optimizing in-order execution of continuous queries over streamed sensor data. In: Proceedings of the 17th International Conference on Scientific and Statistical Database Management, SSDBM 2005, pp. 143–146. Lawrence Berkeley Laboratory, Berkeley (2005)
63. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 211–224. Springer, Heidelberg (2007)
64. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. SIGMOD Rec. 26(2), 171–182 (1997)
65. Hoeksema, J., Kotoulas, S.: High-performance distributed stream reasoning using s4. In: Ordring Workshop at ISWC (2011), `http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Workshops/OrdRing/paper_8.pdf`

66. Hong, W., Stonebraker, M.: Optimization of parallel query execution plans in xprs. Distrib. Parallel Databases 1(1), 9–32 (1993)
67. Jiang, Q., Chakravarthy, S.: Queueing analysis of relational operators for continuous data streams. In: Proceedings of the Twelfth International Conference on Information and Knowledge Management, CIKM 2003, pp. 271–278. ACM, New York (2003)
68. Jiang, Q., Chakravarthy, S., Williams, H., MacKinnon, L.: Scheduling Strategies for Processing Continuous Queries over Streams, pp. 16–30. Springer, Heidelberg (2004)
69. Johnson, T., Muthukrishnan, M.S., Shkapenyuk, V., Spatscheck, O.: Query-aware partitioning for monitoring massive network data streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1135–1146. ACM, New York (2008)
70. Kabra, N., DeWitt, D.J.: Efficient mid-query re-optimization of sub-optimal query execution plans. SIGMOD Rec. 27(2), 106–117 (1998)
71. Kang, J., Naughton, J.F., Viglas, S.: Evaluating window joins over unbounded streams. In: Proceedings of the 19th International Conference on Data Engineering, ICDE 2003, pp. 341–352 (2003)
72. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. 32(4), 422–469 (2000)
73. Krämer, J., Seeger, B.: A temporal foundation for continuous queries over data streams. In: Proceedings of the Eleventh International Conference on Management of Data, COMAD 2005, pp. 70–82 (2005)
74. Krämer, J., Seeger, B.: Semantics and implementation of continuous sliding window queries over data streams. ACM Trans. Database Syst. 34(1), 4:1–4:49 (2009)
75. Krishnamurthy, S., Franklin, M.J., Hellerstein, J.M., Jacobson, G.: The case for precision sharing. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 972–984. VLDB Endowment (2004)
76. Krishnamurthy, S., Wu, C., Franklin, M.: On-the-fly sharing for streamed aggregation. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, pp. 623–634. ACM, New York (2006)
77. Law, Y.N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 492–503. VLDB Endowment (2004)
78. Lee, K.C.K., Leong, H.V., Si, A.: Quay: A data stream processing system using chunking. In: Proceedings of the International Database Engineering and Applications Symposium, IDEAS 2004, pp. 17–26. IEEE Computer Society, Washington, DC (2004)
79. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: Semantics and evaluation techniques for window aggregates in data streams. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 311–322. ACM, New York (2005)
80. Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., Maier, D.: Out-of-order processing: a new architecture for high-performance stream systems. Proc. VLDB Endow. 1(1), 274–288 (2008)
81. Lim, H.S., Lee, J.G., Lee, M.J., Whang, K.Y., Song, I.Y.: Continuous query processing in data streams using duality of data and queries. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, New York, NY, USA, pp. 313–324 (2006)

82. Lin, X., Yuan, Y., Wang, W., Lu, H.: Stabbing the sky: Efficient skyline computation over sliding windows. In: Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, pp. 502–513. IEEE Computer Society, Washington, DC (2005)
83. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Trans. on Knowl. and Data Eng. 11(4), 610–628 (1999)
84. Lopes, N., Polleres, A., Straccia, U., Zimmermann, A.: AnQL: SPARQLing Up Annotated RDFS. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 518–533. Springer, Heidelberg (2010)
85. Luo, G., Naughton, J., Ellmann, C.: A non-blocking parallel spatial join algorithm. In: Proceedings of the 18th International Conference on Data Engineering, ICDE 2002, pp. 697–705 (2002)
86. Madden, S., Franklin, M.J.: Fjording the stream: An architecture for queries over streaming sensor data. In: Proceedings of the 18th International Conference on Data Engineering, ICDE 2002, pp. 555–566 (2002)
87. Madden, S., Shah, M., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: 2002 ACM SIGMOD International Conference on Management of Data, pp. 49–60 (2002)
88. Mei, Y., Madden, S.: Zstream: a cost-based query processor for adaptively detecting composite events. In: Proceedings of the 35th SIGMOD International Conference on Management of Data, SIGMOD 2009, pp. 193–206. ACM, New York (2009)
89. Mokbel, M.F., Lu, M., Aref, W.G.: Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004. IEEE Computer Society, Washington, DC (2004)
90. Mouratidis, K., Bakiras, S., Papadias, D.: Continuous monitoring of top-k queries over sliding windows. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, pp. 635–646. ACM, New York (2006)
91. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. The VLDB Journal 19(1), 91–113 (2010)
92. Ou, Z., Yu, G., Yu, Y., Wu, S., Yang, X., Deng, Q.: Tick Scheduling: A Deadline Based Optimal Task Scheduling Approach for Real-Time Data Stream Systems. In: Fan, W., Wu, Z., Yang, J. (eds.) WAIM 2005. LNCS, vol. 3739, pp. 725–730. Springer, Heidelberg (2005)
93. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. ACM Trans. Database Syst. 34, 16:1–16:45 (2009)
94. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
95. Raman, V., Deshpande, A., Hellerstein, J.M.: Using state modules for adaptive query processing. In: Proceedings of the 19th International Conference on Data Engineering, ICDE 2003, pp. 353–364 (2003)
96. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: Sp2bench: A sparql performance benchmark. In: Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, pp. 222–233 (2009)

97. Schmidt, S., Berthold, H., Lehner, W.: Qstream: deterministic querying of data streams. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 1365–1368. VLDB Endowment (2004)
98. Schmidt, S., Legler, T., Schär, S., Lehner, W.: Robust real-time query processing with qstream. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 1299–1301. VLDB Endowment (2005)
99. Sequeda, J.F., Corcho, O.: Linked stream data: A position paper. In: SSN 2009 (2009)
100. Seshadri, P., Livny, M., Ramakrishnan, R.: Seq: A model for sequence databases. In: Proceedings of the Eleventh International Conference on Data Engineering, ICDE 1995, Washington, DC, USA, pp. 232–239 (1995)
101. Sharaf, M.A., Chrysanthis, P.K., Labrinidis, A., Pruhs, K.: Algorithms and metrics for processing multiple heterogeneous continuous queries. ACM Trans. Database System 33(1), 5:1–5:44 (2008)
102. Sharaf, M.A., Labrinidis, A., Chrysanthis, P.K., Pruhs, K.: Freshness-aware scheduling of continuous queries in the dynamic web. In: WebDB, pp. 73–78 (2005)
103. Sheth, A.P., Henson, C.A., Sahoo, S.S.: Semantic Sensor Web. IEEE Internet Computing 12(4), 78–83 (2008)
104. Shivakumar, N., García-Molina, H.: Wave-indices: indexing evolving databases. In: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, SIGMOD 1997, pp. 381–392. ACM, New York (1997)
105. Smith, J.M., Chang, P.Y.T.: Optimizing the performance of a relational algebra database interface. Commun. ACM 18(10), 568–579 (1975)
106. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2004, pp. 263–274. ACM, New York (2004)
107. Stuckenschmidt, H., Vdovjak, R., Houben, G.J., Broekstra, J.: Index structures and algorithms for querying distributed rdf repositories. In: WWW, pp. 631–639 (2004)
108. Sullivan, M., Heybey, A.: Tribeca: a system for managing large databases of network traffic. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC 1998, USENIX Association, Berkeley (1998)
109. Szomszor, M., Cattuto, C., Van den Broeck, W., Barrat, A., Alani, H.: Semantics, Sensors, and the Social Web: The Live Social Semantics Experiments. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6089, pp. 196–210. Springer, Heidelberg (2010)
110. Tao, Y., Papadias, D.: Maintaining sliding window skylines on data streams. IEEE Trans. on Knowl. and Data Eng. 18(3), 377–391 (2006)
111. Tao, Y., Yiu, M.L., Papadias, D., Hadjieleftheriou, M., Mamoulis, N.: Rpj: producing fast join results on streams through rate-based optimization. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 371–382. ACM, New York (2005)
112. Tian, F., DeWitt, D.J.: Tuple routing strategies for distributed eddies. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 333–344. VLDB Endowment (2003)
113. Tok, W.H., Bressan, S.: Efficient and Adaptive Processing of Multiple Continuous Queries. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 215–232. Springer, Heidelberg (2002), `http://dl.acm.org/citation.cfm?id=645340.650211`

114. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Transactions on Knowledge and Data Engineering 15, 555–568 (2003)
115. Umbrich, J., Karnstedt, M., Land, S.: Towards understanding the changing web: Mining the dynamics of linked-data sources and entities. In: KDML, Workshop (2010)
116. Urhan, T., Franklin, M.J.: Xjoin: A reactively-scheduled pipelined join operator. IEEE Data Eng. Bull. 23(2), 27–33 (2000)
117. Urhan, T., Franklin, M.J.: Dynamic pipeline scheduling for improving interactive query performance. In: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB 2001, pp. 501–510. Morgan Kaufmann Publishers Inc., San Francisco (2001)
118. Urhan, T., Franklin, M.J., Amsaleg, L.: Cost-based query scrambling for initial delays. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD 1998, pp. 130–141. ACM, New York (1998)
119. Viglas, S.D., Naughton, J.F.: Rate-based query optimization for streaming information sources. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD 2002, pp. 37–48. ACM, New York (2002)
120. Viglas, S.D., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 285–296. VLDB Endowment (2003)
121. Vossough, E., Getta, J.R.: Processing of Continuous Queries over Unlimited Data Streams. In: Hameurlain, A., Cicchetti, R., Traunmüller, R. (eds.) DEXA 2002. LNCS, vol. 2453, pp. 799–809. Springer, Heidelberg (2002)
122. Wang, H., Zaniolo, C., Luo, C.R.: Atlas: a small but complete sql extension for data mining and data streams. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 1113–1116. VLDB Endowment (2003)
123. Wang, S., Rundensteiner, E., Ganguly, S., Bhatnagar, S.: State-slice: new paradigm of multi-query optimization of window-based stream queries. In: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB 2006, pp. 619–630. VLDB Endowment (2006)
124. Wang, W., Li, J., Zhang, D., Guo, L.: Processing Sliding Window Join Aggregate in Continuous Queries over Data Streams. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 348–363. Springer, Heidelberg (2004)
125. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. 1(1), 1008–1019 (2008)
126. Whitehouse, K., Zhao, F., Liu, J.: Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. In: Römer, K., Karl, H., Mattern, F. (eds.) EWSN 2006. LNCS, vol. 3868, pp. 5–20. Springer, Heidelberg (2006)
127. Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. In: Proceedings of the First International Conference on Parallel and Distributed Information Systems, PDIS 1991, pp. 68–77. IEEE Computer Society Press, Los Alamitos (1991)
128. Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. Distrib. Parallel Databases 1(1), 103–128 (1993)

129. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, pp. 407–418. ACM, New York (2006)
130. Wu, K.L., Chen, S.K., Yu, P.S.: Interval query indexing for efficient stream processing. In: Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management, CIKM 2004, pp. 88–97. ACM, New York (2004)
131. Zhang, D., Li, J., Zhang, Z., Wang, W., Guo, L.: Dynamic Adjustment of Sliding Windows over Data Streams. In: Li, Q., Wang, G., Feng, L. (eds.) WAIM 2004. LNCS, vol. 3129, pp. 24–33. Springer, Heidelberg (2004)
132. Zhang, R., Koudas, N., Ooi, B.C., Srivastava, D.: Multiple aggregations over data streams. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 299–310. ACM, New York (2005)
133. Zhao, P., Aggarwal, C.C., Wang, M.: gsketch: on query estimation in graph streams. Proc. VLDB Endow. 5(3), 193–204 (2011)
134. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic plan migration for continuous queries over data streams. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD 2004, pp. 431–442. ACM, New York (2004)