ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/301202747

Operator-aware Approach for Boosting Performance in Processing RDF streams

Article *in* Journal of Web Semantics · April 2016

DOI: 10.1016/j.websem.2016.04.001

CITATIONS
5

READS
165

1 author:

Danh Le Phuoc
Technische Universität Berlin
70 PUBLICATIONS **1,779** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

A Scalable and Elastic Platform for Near-Realtime Analytics on The Graph of Everything (SMARTER) View project

All content following this page was uploaded by Danh Le Phuoc on 01 May 2018.

The user has requested enhancement of the downloaded file.

Operator-aware Approach for Boosting Performance in RDF Stream Processing

Danh Le-Phuoc*

Insight Centre for Data Analytics, National University of Ireland, Galway

Abstract

To enable efficiency in stream processing, the evaluation of a query is usually performed over bounded parts of (potentially) unbounded streams, i.e., processing windows "slide" over the streams. To avoid inefficient re-evaluations of already evaluated parts of a stream in respect to a query, incremental evaluation strategies are applied, i.e., the query results are obtained incrementally from the result set of the preceding processing state without having to re-evaluate all input buffers. This method is highly efficient but it comes at the cost of having to maintain processing state, which is not trivial, and may defeat performance advantages of the incremental evaluation strategy. In the context of RDF streams the problem is further aggravated by the hard-to-predict evolution of the structure of RDF graphs over time and the application of sub-optimal implementation approaches, e.g., using relational technologies for storing data and processing states which incur significant performance drawbacks for graph-based query patterns. To address these performance problems, this paper proposes a set of novel operator-aware data structures coupled with incremental evaluation algorithms which outperform the counterparts of relational stream processing systems. This claim is demonstrated through extensive experimental results on both simulated and real datasets.

Keywords:
Continuous queries, Linked Stream Data, Linked Data, Semantic Web, stream processing

1. Introduction

There are billions of heterogeneous stream data sources that are continuously producing an enormous amount of information under diverse ownerships and controls. To overcome this issue, the RDF data model becomes a natural choice to provide an integrated view for querying the data without requiring an adherence to a specified schema. Hence, several RDF Stream Processing (RSP) engines [1] have been proving their advantages in tackling interoperability in several projects and products in Internet of Things and Smart Cities, etc. However, due to to the intrinsic nature of stream data processing in such targeted applications, the ability to process stream data at high throughput and low latency is a vital requirement that has not been throughly addressed in such engines.

A common strategy in implementing an RSP so far is delegating performance issues to underlying libraries or systems, e.g., RDF/SPARQL query processors, data stream management systems (DSMSs) or complex event processing engines (CEPs). This strategy greatly benefits from the available techniques, approaches and tools provided by such research communities. However we believe that it requires a lot of effort in investigation, tuning and re-engineering to build a performant RSP engine. In particular, the RDF store or SPARQL query processor is designed for heavily read-intensive contexts [2, 3, 4] whilst an RSP engine needs to deal with high writing throughput of unbounded incoming data and continuous computation corresponding to such updates. To answer this need, several RSP engines employ the work on evaluating sliding-window operators of DSMS or CEP. For instance, C-SPARQL [5] uses ESPER [1] together with Jena [2] and CQELS [6] implements operators of Aurora [7] using the underlying TDB libraries of Jena.

In such generic stream-processing engines, there are two evaluation strategies of executing sliding-window operators, i.e., re-evaluation and incremental evaluation [8]. In the re-evaluation strategy, the query is re-evaluated independent of the previous computation efforts. To avoid inefficient re-evaluations of already-evaluated parts of a stream in respect to a query, incremental-evaluation strategies are applied, i.e., the query results are obtained incrementally from the result set of the preceding processing state without having to re-evaluate all input buffers. This method has been proven highly efficient to some extents [9, 10, 11, 12, 13] but comes at the cost of having to maintain processing state, which is not trivial, and may defeat performance and scalability advantages of the incremental evaluation strategy [11, 8]. As shown in [8], several efforts in providing incremental evaluation algorithms for sliding windows have been conducted over the years but there are still challenging issues and a lot of room for improvement. In the context of RDF streams, the problem is further aggravated by the hard-to-predict evolution of the structure of RDF graphs over time and the application of sub-optimal implementation approaches, e.g., using relational technologies for storing data and processing states which incur significant performance drawbacks for graph-based query patterns.

To address the above shortcomings and challenging issues, this paper introduces a set of novel operator-aware data structures associated with efficient incremental evaluation algorithms

*E-mail address: danh@danhlephuoc.info

[1] http://www.espertech.com/esper/index.php
[2] https://jena.apache.org/

to deal with the specific properties of RDF stream data and common query patterns. These new data structures are designed to handle small data items and intermediate processing states very efficiently. The data structures include various low-maintenance-cost indexes to support high throughput in the probing operations that are used in various operator implementations. Based on such data structures, we propose several algorithms to enable incremental evaluation of basic operators such as join, aggregation, duplicate elimination and negation. These algorithms also overcome the typical problems of incremental evaluation of windowing operators. Our experimental results show that our approach performs better than relational approaches and re-evaluation approaches by orders of magnitude in reducing query execution delay.

The remainder of this article is structured as follows. In the next section we introduce some background concepts and techniques and present some analyses of current approaches to set the context of the contributions of the paper. Then, we describe our operator-aware data structures in Section 3, including storage design, access methods, usage, implementation variation, optimisation, and performance tuning After that we present incremental evaluation algorithms based on these data structures in Section 4. To evaluate the performance of the operators built on such data structures and algorithms, we present and discuss the experimental results using our new data structures and algorithms in Section 5. Besides, the related work is also discussed in the following Section 6. Finally, we finish the paper with our conclusions.

2. Background and Analysis

Before going into technical details of the article, this section introduces some background to review the technical shortcomings that motivate the design of our data structures and algorithms.

2.1. Continuous query operators on RDF Stream

An RDF stream is modelled by utilising the definitions of RDF nodes and RDF triples whereby the *stream elements* of an RDF stream are represented as RDF triples with temporal contexts. As the standardisation of RDF Streams is still the on-going work of the W3C RSP community[3], this paper will leave the formal definitions of an RDF stream generic enough for latter adoption. Instead, we focus on evaluation aspects of basic continuous query operators, e.g., join, aggregation and duplication elimination. Hence, we introduce an example of real RDF streams to investigate the issues of evaluating continuous queries inspired by the open dataset of taxi rides in New York [4]. Note that instead of following strictly to its stream format, we assume that there are three RDF streams, $\mathcal{S}_{pickup}$, $\mathcal{S}_{dropoff}$ and $\mathcal{S}_{fare}$ with the schema shown below. Two streams $\mathcal{S}_{pickup}$ and $\mathcal{S}_{dropoff}$ report the events of a taxi being picked up or dropped off respectively. The stream $\mathcal{S}_{fare}$ reports the payment for a taxi ride that was reported in the stream $\mathcal{S}_{pickup}$ at a pickup time given by the triple with the predicate :*pickupTime*. Note that in the example, an event is represented as an RDF graph, however, clearly other representations still can be processed by the operators to be discussed in this paper.

$$\mathcal{S}_{pickup} = \left\{ \begin{array}{l} :ride_1\ :taxi\ :89...CF4 \\ :ride_1\ :pickupTime\ \texttt{"2013-01-01 15:11:48"}. \end{array} \right\}.$$

$$\mathcal{S}_{dropoff} = \left\{ \begin{array}{l} :ride_1\ :dropoffTime\ \texttt{"2013-01-01 15:18:10"}. \\ :ride_1\ :triptime\ 382. \end{array} \right\}.$$

$$\mathcal{S}_{fare} = \left\{ \begin{array}{l} :trans_1\ :fare\ 7. \\ :trans_1\ :pickupTime\ \texttt{"2013-01-01 15:11:48"}. \end{array} \right\}.$$

The temporal context of each event recorded in each stream is specified by a triple with a temporal predicate, i.e., *pickupTime* or *dropoffTime*. Note that, the temporal context can be encoded differently in a certain implementation. For instance, the implementations of C-SPARQL [5] and CQELS [6] use the system timestamp at the time when an event (i.e. a triple) arrives to the system. This timestamp is encoded as the temporal context to determine the order of stream elements to be processed in the engine. In the scope of this paper, we assume the stream elements are totally ordered. This assumption might lead to a limitation that there cannot be two taxi rides with the same pickup time to guarantee a fare to be unambiguously connected with a ride. In practice, this limitation can be overcome by using suitable time-unit of the timestamps. Most such systems define the *sliding-window* operators which are used to extract a finite set of RDF triples as an RDF graph from an RDF stream based on a certain window. The idea of extracting an RDF graph from an unbounded RDF stream is to be able to apply SPARQL query operators, e.g., SPARQL 1.1, so that a continuous query language on RDF stream can inherit the SPARQL grammars. Along this line, the definitions of such window operators on RDF streams are adopted from window operators on relational streams of CQL [14]. Consequently, the semantics of a continuous query on RDF streams are defined as a composition of such query operators. For example, with the above data, a query based on currently agreed syntaxes of RSP community is illustrated as bellow. This query is used to continuously report *"hourly riding rate of **active taxies** of last 1000 payment transactions"* whereby **active taxis** are the taxis that reported picking-ups within 2 hours and dropping-offs within last 1 hour.

```
SELECT ?taxi (AVERAGE(?fare/(?tripTime/3600)) AS ?hourlyRate)
FROM  NAMED WINDOW :W1 ON  nyctaxi:fare [COUNT 1000]
FROM  NAMED WINDOW :W2 ON  nyctaxi:pickup
                           [RANGE PT2H @:pickupTime]
FROM  NAMED WINDOW :W3 ON  nyctaxi:dropoff
                           [RANGE PT1H @:dropoffTime]
WHERE {
  WINDOW :W1{ ?trans :fare ?fare. ?trans :pickupTime ?pTime}
  WINDOW :W2{ ?ride :taxi ?taxi. ?ride :pickupTime ?pTime}
  WINDOW :W3{ ?ride :triptime ?tripTime}
  }
GROUP BY  ?taxi
```

Example query on NYC taxi data SPARQL-like continuous query

To evaluate this query, a physical continuous query pipeline is translated as illustrated in Figure 1. At the root of this query pipeline, the aggregation operator (AVERAGE) consumes inputs from the join operator that joins three buffers storing solution mappings extracted from corresponding sliding windows W1, W2, W3 at the leaves of the query pipeline. The definition of a solution and mapping is borrowed from the SPARQL 1.1 specification [5]. For a shorter presentation, from this point forward,

[3] https://www.w3.org/community/rsp/

[4] http://chriswhong.com/open-data/foil_nyc_taxi/

[5] http://www.w3.org/TR/2013/REC-sparql11-query-20130321/#sparqlDefinition

we use the term "mapping" for "solution mapping". In effect, each of these operators consumes a set of bags of mappings and returns a bag of mappings which then can be used as intermediate mappings to be consumed by another operator. Basically, a query pipeline forms an acyclic graph of continuous query operators. Such continuous query operators are defined by extending the query operators of CQL [15] and SPARQL [16] similar to the ones defined in [5, 6]. In each query graph, the inputs at the leaves are bags of mappings stored in the window buffers of sliding window operators and the top, i.e. AVERAGE, returns a bag of mappings to the output stream.

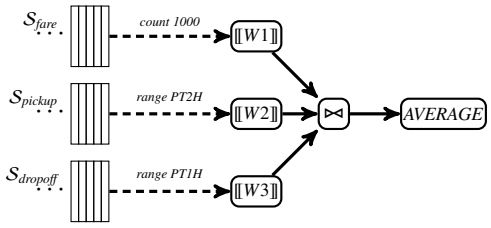


Figure 1: An example continuous query pipeline

By a simple modification of an RDF store, this continuous query can be processed by repeatedly executing SPARQL queries on the RDF Store. However, the data layout of a common RDF store is designed for heavily read-intensive contexts [2, 3, 4] while RDF stream data needs high writing throughput: each new stream element is an update, i.e., a write operation. Typically, DSMSs remedy this requirement by using sliding-window operators on in-memory storage. However, RDF-based data elements such as RDF triples and temporal RDF triples can be encoded as fixed-size integers. Hence, the processing states can be encoded as unusually large numbers of individual small data points compared to the amount incoming data in the raw form, i.e. RDF serialised format. Consequently, the row-based data structure used in relational DSMSs is inefficient because it requires the tuple header sizes that might dominate the total storage footprint [17]. In particular, the row-based data structure is typically designed for wider and shorter tables which might increase significantly the memory footprint for stream processing. Moreover, as shown in several works on the physical representation of RDF data [18, 19, 20, 2, 17, 3, 4], relational tables are not ideal for storing RDF data elements. This motivates us to design a tree-based data structure associated with indexing strategies to remedy this issue. More importantly, the second reason leading to this design is to enable a more efficient evaluation strategy than re-evaluation strategy in state-of-the-art literature discussed in Section 6, i.e, incremental evaluation. Before analysing the shortcomings of current approaches, we present an overview of the operational aspects in incrementally evaluating some basic sliding operators of relational DSMSs in the following section.

2.2. *Incremental evaluation of sliding-window operators*

The complexity of an algorithm for incrementally evaluating a sliding-window operator is dictated by how to maintain the processing state for an execution of re-evaluation triggered. For stateless operators, re-evaluation can be done on-the-fly without having to maintain any processing state. For instance, Figure 2(a) shows how the selection operation over a stream S1 works [21]. The duplicate-preserving projection and union operators are also examples of stateless operators. In contrast to stateless operators, a stateful operator needs to probe[6] the previous processing states in every incremental computation step. Maintaining processing state is done differently for each type of operator. Next, we will discuss how to deal with basic stateful operators such as window join, aggregation, duplication elimination and negation.

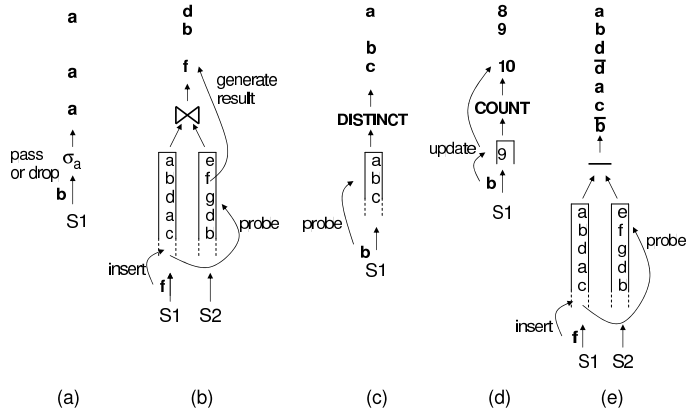


Figure 2: Operator implementations : selection (a), window join (b), duplication elimination (c), aggregation (d), and negation (e).

Window join operator

In a sliding window join, newly arrived tuples on one of the inputs trigger the probing the states of the other inputs. Additionally, expired tuples are removed from the processing state [9, 22, 23, 10, 12]. Expiration can be done periodically, provided that old tuples can be identified and skipped during the processing. Figure 2(b) is an example of a non-blocking pipeline join [24, 25, 26, 22, 27, 28]. 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 [26]. In the latter, new arrivals on the streams trigger the probing of the relation.

Duplicate elimination operator

Duplicate elimination operator is usually used to process the DISTINCT modifier of output streams. The duplicate elimination, as illustrated in Figure 2(c), maintains a list of distinct values already seen and filters out duplicates from the output stream. When a new tuple with value *b* arrives, the operator probes its output list, and drops the new tuple if a tuple with value *b* has already been seen 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

[6] term *probe* is used for operations of searching a processing state for needed data and the processing state can be an original data storage or auxiliary data structures like a hash table or a B-tree

present in the window [29]. Alternatively, duplicate elimination may produce a single result tuple with a particular value v and retain it on the output stream as long as there is at least one tuple with value v present in the window [15, 30]. In both cases, expirations must be handled eagerly so that the correct result is maintained at all times.

Aggregation operator

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 a value v do not need to be stored if there is another tuple in the window with a value greater than v and with a younger timestamp (see, e.g., [13, 31] for additional examples of reducing memory usage in the context of skyline queries and [32] in the context of top-k queries). Additionally, in order to enable incremental computation, the aggregation operator stores the current answer or frequency counters of the distinct values present in the window. 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. A non-blocking aggregation [33, 34, 35] 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.

Negation operator

As negation is a non-monotonic query pattern of SPARQL 1.1, we also consider how to incrementally evaluate it. Indeed, this operator can be evaluated incrementally 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 [29, 15]. Negation of two sliding windows, $S1 - S2$, may produce negative tuples (e.g., arrival of an $S2$-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 an $S1$-tuple with value v may need to be appended to the output stream [29]). An example is shown in Figure 2(e), where a tuple with value d was appended to the output because it is generated on the output stream upon subsequent arrival of an $S2$-tuple with value d.

2.3. Shortcomings

Next, we analyse the shortcomings of current approach in incremental evaluation of continuous query operators that set the context for this article's contributions. Some shortcomings in employing incremental evaluation methods have been identified in [29, 8]. In essence, such incremental evaluation methods are categorised by how to signal the expirations. There are two main techniques to signal expirations: *direct timestamp* [15, 36] and *negative tuple* [15, 36, 21]. While the direct-timestamp method needs extra timestamps, the negative tuple method doubles the number of tuples through the query pipelines. Hence, both approaches significantly add significant overheads to the processing load. In following, such shortcomings will be throughly analysed to propose our approach.

As a mapping is a partial function that maps a variable to an RDF Node (binding value), we employ the encoding strategy commonly used in RDF stores [17, 37] to encode binding values as integer identifiers to substantially improve the performance compared to storing RDF nodes in a lexical form. Therefore, a mapping contains integer identifiers representing URIs and literals. These identifiers are associated with mapping tables as dictionaries to translate them back into their lexical form. One could argue that maintaining an RDF dictionary would cause memory leak due to the expired RDF nodes that are accumulated after a period of time running the system. This issue can be easily addressed in a similar fashion of handling the processing buffer of a sliding window. The technical details will be discussed in the indexing structures presented in Section 3. The encoded form of a binding value enables most query operators to be executed on small fixed-size integers rather than large variable-length strings. This not only reduces memory footprint but also considerably improves cache efficiency and reduces I/O time in many cases [37].

A continuous execution query pipeline is continually applied to the input buffers storing bags of mappings. The incremental evaluation of a stateful operator needs to handle the events of arrivals of new stream elements and expirations of old stream elements [36]. Notably, processing states such as window buffers and operator inputs constantly change through "insert" and "evict" operations. Therefore, data structures and physical storages for mappings and bags of mappings have a significant impact on the performance of continuous query operators. As a mapping gives the access to encoded binding values as fixed-size integers, it would be inefficient to store it as a relation [18, 19, 20, 2, 17, 3, 4]. Moreover, the physical storages and indexing structures of bags of mappings have to be re-engineered to give a fast lookup speed while still coping with a high insert/evicting throughput. Therefore, new storages and indexes tailored to these requirements are proposed in Section 3.

In addition, the design of the new data structures also needs to be aware of the challenge of handling the insertion and expiration of processing states. Handling the expiration is often more complicated than the counterpart. An expired mapping may cause the removal of one or more items from the result (e.g., aggregation) or the addition of new mappings to the result (e.g., duplicate elimination used in the DISTINCT operator). There should be mechanisms to signal expiration events from the window buffers to the upper operators of a query graph to reflect what has been changed. For instance, a new payment made in the aforementioned example might cause an expiration of the oldest payment in the count window W1 that requires the query evaluator to check whether to trigger the incremental computation of the aggregation operator. To highlight the issues of the two aforementioned approaches of signalling the expirations, i.e. direct timestamp and negative tuple, we revisit them by considering the example shown in Figure 3. This example illustrates what happens in the join operator of the query

pipeline in Figure 1 with a simplistic version of data and window parameters. For instance, $\langle a_1, b_1 \rangle$ represents the mapping $\{?a \mapsto a_1, ?b \mapsto b_1\}$ whereby $?a$ is a variable and b_1 is a binding value and so forth. This figure shows how the data enters the windows, how the results are generated, and how the expired mappings should be invalidated.

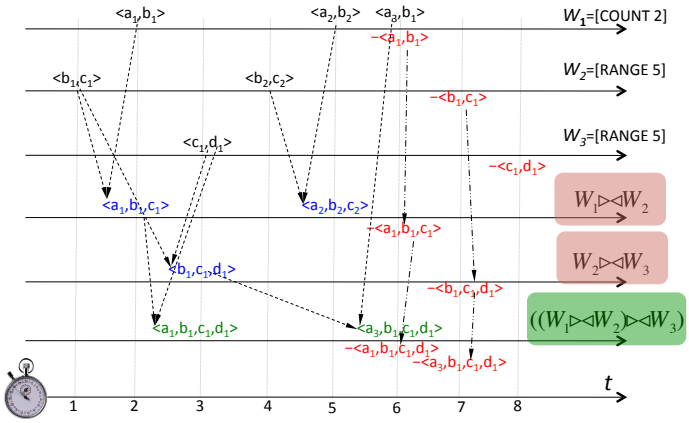


Figure 3: Invalidate expired mappings.

For the *direct-timestamp* approach, when each mapping arrives at a timestamp t, it will be assigned an expiration timestamp, exp, where $exp = t + windowlength$, and $windowlength$ is the length of the sliding window. For instance, $\langle b_1, c_1 \rangle.exp = 1 + 5 = 6$ and $\langle c_1, d_1 \rangle.exp = 3 + 5 = 8$. When we join these two mappings, we have the result mapping $\langle b_1, c_1, d_1 \rangle$ with an expiration timestamp equal to $min(\langle b_1, c_1 \rangle.exp, \langle c_1, d_1 \rangle.exp) = min(6, 8) = 6$. To invalidate the expired mappings at each time point, the operators simply check the expiration timestamps of the mappings in their processing states. However, this approach assumes that storage space for timestamps is much smaller that the space of each tuple, i.e, mapping in this case. However, in our example, a constituent mappings like $\langle b_1, c_1 \rangle$ or $\langle c_1, d_1 \rangle$ might only need the space for two integers for encoded binding values, i.e., 64 bits for each binding value which is the same storage size for a new timestamp created. This means that the intermediate timestamps created through the query pipeline would need significant storage size in comparison with original binding values.

Furthermore, this approach is not applicable to count-based windows like W_1. For instance, when the mapping $\langle a_1, b_1 \rangle$ arrives at W_1, its expiration time is not known until time point 6, where the mapping $\langle a_3, b_1 \rangle$ arrives. In this case, the negative tuple approach can be used to signal the expirations. For instance, at time point 6, a negative mapping $-\langle a_1, b_1 \rangle$ is sent as a signal for the invalidation. Therefore, the invalidation has to re-compute the join again with the negative mapping to find the expired mapping $\langle a_1, b_1, c_1, d_1 \rangle$ generated in the final results. Hence, [8] proposed a solution to reduce the overhead of re-processing the negative mappings because every mapping eventually expires from its window and generates the corresponding negative tuple. However, this requires extra memory since it adds expiration timestamps and piggy-back flags to intermediate results. Moreover, this approach can only be applied for time-sliding windows. A hybrid approach is also proposed in [38], but the processing and memory overhead still persist. These shortcomings will be then addressed systematically with the data structures presented in the next section and Section 4.

3. Operator-aware Data Structures

To address the issues highlighted in the previous section, this section presents the *operator-aware data structures* for mappings and bags of mappings that also provide efficient access methods for the incremental evaluation of windowing operators. These data structures are called "operator-aware" because their design is based on how the query operators manipulate the data items, i.e., binding values, mappings and bags of mappings. The design of storage and access methods for these data structures are presented in Section 3.1 and Section 3.2 respectively. Furthermore, implementation details and optimisation techniques are correspondingly discussed in Section 3.3 and Section 3.4.

3.1. Storage Design

By analysing the aforementioned issues of invalidating expired mappings in a query pipeline, we introduce a tree-based storage for storing mappings. In contrast to relation-based storage that stores mappings in rows of a table, the tree-based storage stores relevant mappings in a tree shape which is recursively formed from the query pipeline generating such mappings. The tree shape plays the role of a trail to check whether a mapping is expired (c.f Section 3.2) and to implement Algorithm 3 to invalidate expired mappings in Section 4.

The tree-based storage categorizes mappings into two types: *leaf-mappings* and *intermediate-mappings*. The leaf-mappings are generated by the pattern matching operators. Hence, the binding values and timestamp(s) of a leaf-mapping can be stored in a row-based structure like an array, a hash map or a row in a relational table. The intermediate-mappings are mappings generated by the operators in the upper levels of a tree-based query pipeline. Figure 4 illustrates how relevant mappings are represented as a tree of the constituent mappings from the example in Figure 3. In this figure, the query processing pipeline $((W_1 \bowtie W_2) \bowtie W_3)$ generates different types of mappings. At the bottoms, we see three leaf-mappings in window W_1, two leaf-mappings in window W_2 and one leaf-mapping in window W_3. After that, at the first upper level of the query graph, $(W_1 \bowtie W_2)$ generates mappings $\bowtie_{11}, \bowtie_{12}, \bowtie_{13}$ and then at the second upper level, $((W_1 \bowtie W_2) \bowtie W_3)$ generates mappings $\bowtie_{21}$ and $\bowtie_{22}$. In

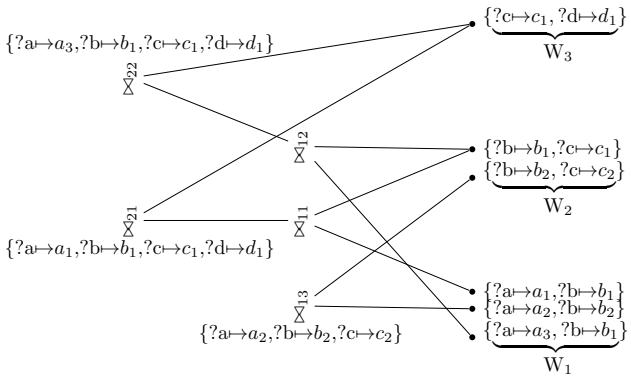


Figure 4: Trees of mappings.

this storage design, it requires considerably less memory to store intermediate mappings in comparison to the row-based storage. For instance, if we use the row-based storage, mappings $\bowtie_{11}, \bowtie_{12}$ and $\bowtie_{13}$ would need more than 3x64bits to store each one and also 4x64bits for each of mappings $\bowtie_{21}$ and $\bowtie_{22}$. On top of that, each pointer only needs less than 32 bits for a 32 bit CPU and 64 bits for a 64 bit-CPU since the operating systems or JVMs support pointer compression.

An operator in a query pipeline consumes or outputs mappings in a bag stored in a buffer. Hence, a bag of mappings can be stored in a *list* of pointers referencing the mappings stored in the tree-based storage. Similar to the concept of *synopsis* of STREAM [15], these lists are used to maintain and exchange processing states in a query pipeline. The mappings in the list might be linked in chronological order, for example, the newest leaf-mapping in the head and the oldest in the tail for window buffers. To support fast probing, a list needs indexing structures that have low-maintenance-cost for inserting and evicting its tree-based data items. The index should provide an interface to look up a key as a lookup condition over a set of binding values. The characteristics of operations such as inserting, evicting and probing dictate the choice of the indexing structures on their subsets of binding values. Due to different uses of the indices, the index entries and the list data structures are implemented as a one-way index or a ring index described in the following (Figure 5).

3.2. Access methods & Usage

With the tree-based storage, an intermediate mapping might not need to store binding values as they can be retrieved via its constituent mappings. It only needs one or two pointers to reference the mappings that generated it. However, in some operators like aggregation, some new binding values will be added to the generated mapping together with a mapping that gives access to other binding values. For instance, the AVERAGE function of the aggregation operator generates binding values for the new variable ?*hourlyRate* in the example query of Section 2. As the aggregated mapping accumulates values from several mappings that have a same "group by" value (cf. Section 4.2), it only has to reference to one mapping of the group to access to "group by" value, e.g. ?*taxi*.

Besides, with this tree-based structure, we only need to store timestamps on the leaf-nodes in order to generate timestamps for the final results and to detect the expiration of intermediate results. While saving memory space, this data structure needs more steps to reach the leaves to retrieve the binding values or to compute timestamps for intermediate mappings. However, this does not have a critical performance impact on the processing due to several reasons: First, retrieving binding values takes much less time in comparison to probing operations that dominate the processing time of the query processor. Another reason is due to the characteristics of modern hardware in which the time spent for traversing to the leaves is much smaller than the time spent for retrieving/copying binding values in the main memory. In particular, a single core of a modern CPU can execute up to four instructions per cycle [37] while the latency of accessing data in RAM can be over 200 processor cycles [39]. On top of that, a query usually has less than 100 graph patterns. Therefore, a tree of a mapping has only a few nodes to traverse for reaching the binding values.

To give fast access methods to a bag of mappings stored in a list associated with indices, we propose domain-based indices on keys constructed from a subset of binding values of the mappings stored in the list. The access methods provided by such indices are inspired by the domain indices proposed in [40, 41, 11]. The design of the indices are illustrated in Figure 5. In this design, the keyed binding values of the mappings stored in the list are used to construct the keys which are then used as the parameters to access to the corresponding index. In an index, each key is associated with an *entry* that stores the number of items, called *counter*. The key and entry pair are stored in data structures such as a B-tree or a hash table. The first index is a one-way index as illustrated in Figure 5a. The one-way index can only be used to efficiently check if there is an item with a certain key in the list. Therefore, it is useful for operators such as duplication elimination and aggregation. For operations that needs to retrieve the list of items that match a certain key, we extend the one-way index to the ring index [40, 41] as depicted in Figure 5b. The ring index links all mappings with the same index key in a ring. The index entry of this key contains the start-of-ring pointer to the first mapping of the ring. This first mapping is the newest mapping of the ring.

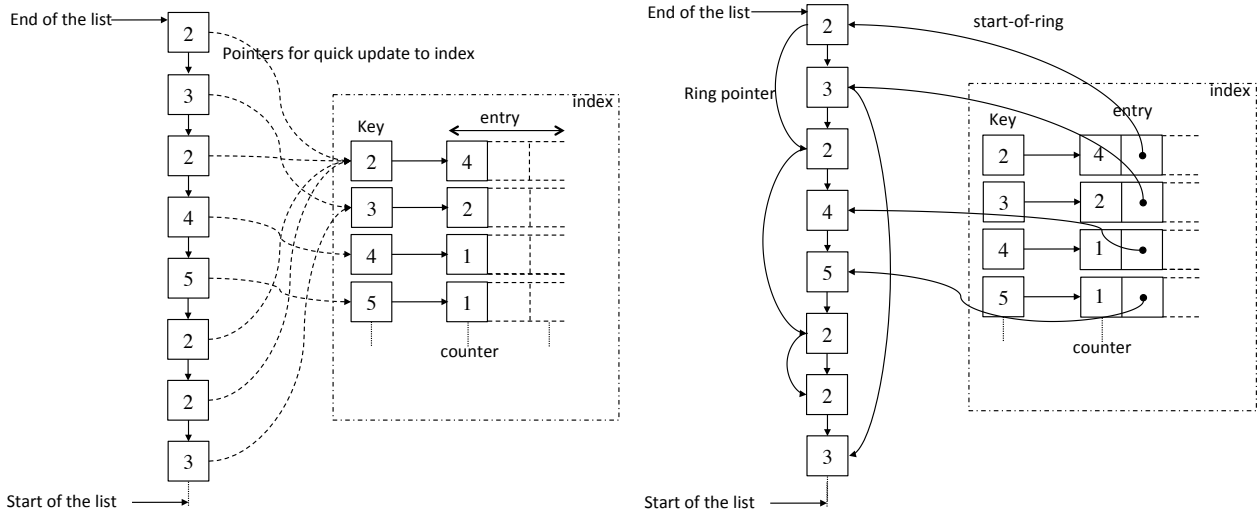
Let N denote the number of mappings in a list and n is the number of keys in an index. As n is considerably smaller than N, for instance, if we create a one-way index on the variable ?*taxi* of the example query in the running example, n can be only few thousand but N can be many times larger than n, e.g., 100-1000 times in our experiments in Section 5. Therefore, the complexity of any operation on our domain-based indices is much smaller than those of normal direct indices on mappings contained in a list. The implementation details of inserting and probing operations on these indexing structures are discussed in the following section.

3.3. Implementation variations

The important implementation aspect of the tree-based storage for a mapping is how to check if this mapping is expired. For leaf-mappings, checking expiration of a leaf-mapping at a clock-tick [7] is done easily by a window implementation. For instance, for time sliding windows, the expiration of a leaf-mapping can be detected by its timestamp. Consequently, an intermediate mapping is expired if at least one of its constituent leaf-mappings is expired. Figure 6 shows an example of how the expiration of some mappings is detected. In this example, if the mapping $\langle a_1, b_1 \rangle$ of window W_1 has expired, the expirations of the mappings $\bowtie_{21}$ and $\bowtie_{11}$ are detected by traversing through the doubled-tip arrows.

The list could be implemented as a linked list or an expandable array. The array-based implementation of the list has more chances of cache hits and it does not need an extra data structure or pointers for linking data elements in the list [37]. When the maximum size of the list is known, a circular array implementation of the list can be used, for instance, a count-based window with a fixed number of items. For the list whose maximum size is unknown, if the size of this list goes over the size of the array, another array has to be allocated. This introduces an overhead

[7] In the scope of this paper, we assume a clock-tick is a step to trigger a query execution



(a) One-way index

(b) Ring Index

Figure 5: Indices for bags of mappings.

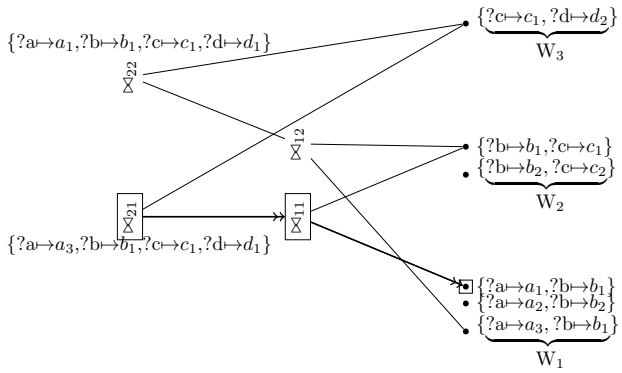


Figure 6: Tree-based Expiration checking.

of copying data elements to the new array. Therefore, the linked list is an alternative if the maximum size of the list is wildly unpredictable. In practice, if the size of the list does not change dramatically, the circular array-based implementation of the list is faster than the linked list. In particular, for a list used in a window buffer, it only needs to remove items from the tail and to insert into the head. Therefore, the circular array can be used to save pointers to the next items. In other cases, where data items might be removed randomly, the "next" pointer has its advantage over the circular array.

For the one-way index shown in Figure 5a, the implementation of the index entry can be extended to store aggregate values for each group (c.f. Section 4.2). It is also used to index hashed keys of RDF nodes to build encoding dictionaries as discussed in Section 2.3. In this case, the index entry can be extended to store the original value of an RDF node or the address of a big value, e.g., a long string, in the external disk (to reduce memory footprint). With this index, the incoming RDF can be maintained as a linked list in the same fashion of maintaining a bag of mappings for a sliding window. Therefore, an expired RDF node (e.g, its timestamp is older than any mapping of interest) can be released in the same way as releasing an expired mapping. For inserting a new mapping into the list, the index needs one lookup to find the entry corresponding to the key of the new mapping, then updates its counter. If the key has not been in the index, then a new entry will be created. The delete operation also needs a lookup to find the corresponding entry in the index to decrease its counter.

For the ring index shown in Figure 5b, to manipulate with mappings in a list, two methods, *insert* and *probe* need to be implemented as following. The method *insert* in Algorithm 1 is used to insert a new mapping to the end of the list with a ring index. If the key of the new mapping μ is already in the index, the algorithm re-assigns its ring pointer to the new mapping located in the beginning of the ring (line 4), and increases the counter of the ring index entry (line 5). Otherwise, a new ring index entry is created (line 8). The start-of-ring pointer then points to the inserting mapping μ (line 10).

Algorithm 1: *insert*(μ, $\mathcal{L}$)

Input: μ: mapping to be inserted, $\mathcal{L}$: the list with a ring index

1 $(K) \leftarrow$ key of μ for the ring index in $\mathcal{L}$;
2 $\mathcal{E} \leftarrow \mathcal{L}$.index.get($(K)$);
3 **if** $\mathcal{E}$ *is not null* **then**
4 $\quad \mu$.ring-pointer=$\mathcal{E}$.start-of-ring;
5 $\quad \mathcal{E}$.increaseCounter();
6 **end**
7 **else**
8 $\quad \mathcal{E} \leftarrow$ create a new ring index entry with counter=1;
9 **end**
10 $\mathcal{E}$.start-of-ring=μ;
11 $\mathcal{L}$.insert(μ);

To retrieve a list of mappings that have a particular key, the method *probe* is provided in Algorithm 2. It first gets the ring index entry corresponding to the key. If it is not null, an iterator

representing for $\mathcal{E}$ is created for iterating over all the mappings that have the same key by following the ring pointers. Note, this iterates over $\mathcal{E}$.count() mappings by following the ring pointers from the mapping pointed by $\mathcal{E}$.start-of-ring. Therefore, the counter in the index entry is used as the stop condition for the iterator.

Algorithm 2: *probe*($\mathcal{K}$)

Input: *key*: key to be probed, $\mathcal{L}$: the list with a ring index

```
1 E ← L.index.get(K));
2 if E is not null then
3 |   return E; // E is accessed via an
  |      iterator over L following the
  |      ring pointers corresponding to
  |      key K)
4 end
5 else
6 |   return empty iterator;
7 end
```

3.4. Optimisation & Performance tuning

For deleting an item in the list with indices, we just need to search the corresponding keys in the indices and decrease the frequency count of that key. Consequently, when the frequency count of a key in an index reaches zero, this means that the key is no longer referred to any mappings. However, deleting such keys in the index might be inefficient. For instance, if a self-balancing tree is used for the index, a lazy-deletion approach could avoid a number of re-balancing operations [41]. To enable lazy-deletion, we delay the deletion by still maintaining the keys with zero frequency count. A simple condition to check if a frequency count > 0 must be added in the search function of the index implementation. We maintain a list of keys with zero frequency count, then when the size of the list reaches a certain threshold, a batch deletion operation will be carried out. There might be the case that by the time the batch deletion is started, some of the keys in the list have a frequency count greater zero. By just ignoring these keys after delaying the delete operation, we can save unnecessary insert/delete operations on them. In practice, the overhead of lazy deleting operation might be negligible, then, the complexity of maintaining an index for a deleting operation of a mapping is O(S(n)) where S(n) is the complexity of the a search operation on the index and n is the number of indexing keys. For example, if we use a binary tree for the index, we have S(n)=$log_2(n)$, thus, O($log_2(n)$) is the complexity of maintaining index on deleting operation. If we compare with a regular relational storage approach with a binary index, the complexity might be O(N^2) for reorganising this index. Note that N is the number of mapping in the list and highly like much greater than n. If n is small, e.g., few thousand or less, which is quite common for several RDF predicates, it is possible to implement the index with O(1) complexity using an array.

Moreover, for each delete operation in a list, one of its indices needs a lookup to find the corresponding entry in the index to decrease its counter. Such lookups can be avoided by adding a pointer to every mapping in the list for each index. The mappings would have a pointer to point back to its index entry. When a mapping is deleted, we just need to follow the pointer to the entry to decrease its counter. For one-way index, if the number of keys is large and the deletion rate is high, adding one more pointer to each mapping might improve the throughput, provided that extra memory consumption is not critical (see Figure 5a). On the other hand, if the number of keys is small and there is a large number of items in the buffer, not having such pointers might be a better solution. Hence, the instantiation of the mappings for the one-way index can be dynamically switched at runtime based on the size of the list and the number of keys.

We can use balanced trees or hash tables for indexing keys for both one-way and ring indices. The hash tables can only be used for the equality predicates whilst the balanced trees also support range scans. The balanced trees have fixed boundaries of search time defined by their heights. For instance, an AVL tree's height is $1.44log_2(n+2)-1$ and a Red-black [42] tree's height is $2log_2(n+1)$, where n is the number of indexed keys. In principle, the hash table can have search time of O(n) in the worst case. However, the average search time of a hash table is O($n/k+1$), where k is the number of buckets [42]. We favour hash tables for equality predicates because the adaptive executor can trade memory for better speed by increasing the number of buckets to have lower search time and higher inserting throughput. For instance, in our experiments in Section 5 the inserting throughputs of the ring index on the hash one ranges from 400,000 to 800,000 mappings per second for the window size up to 10 million mappings and its average probing time on a 1-million-mapping window takes around 600-800 nano seconds.

4. Algorithms for Incremental Evaluation

Based on the new data structures introduced in previous section, an expired mapping can be identified by following the process which created it. For instance, recall the example in Section 2.3, we can trace back that $\langle a_1, b_1\rangle$ created $\langle a_1, b_1, c_1, d_1\rangle$, then, when $\langle a_1, b_1\rangle$ is expired, we will know $\langle a_1, b_1, c_1, d_1\rangle$ is also expired. Therefore, we enable the invalidation of an expired mapping by maintaining a tree structure according to the operator tree that created this mapping. To signal an invalidation to the following operators, the window containing the expired leaf-mapping sends a negative leaf-mapping to the final operator in the query pipeline to trigger the invalidation. The negative leaf-mapping of a leaf-mapping is an expired version of it. It can be created by simply changing the timestamp to its negative value. For instance, at time point 6, the third mapping $\langle a_3, b_1\rangle$ arrives at the window, and thus, the first mapping $\langle a_1, b_1\rangle$ expires. The negative leaf-mapping $-\langle a_1, b_1\rangle$ is sent to the final operator to check whether a processing state was generated from this expired mapping. In this example, the final operator is the aggregation operator and the expired processing state is $\langle a_1, b_1, c_1, d_1\rangle$ needs to be invalidated to trigger the incremental computation of the average value of the corresponding taxi. Note that, this expiration signalling mechanism does not need to recompute the join operator to find which joined mapping(s) were generated from this expired mapping(c.f Section 4.1). This is highly efficient for operators placed after complicated joins of big windows. The later evaluation results confirm this observation.

In the output buffer of an intermediate operator, the expired mappings triggered by a negative leaf-mapping might not be

stored in a consecutive order. For instance, a join operator can generate output mappings in a random order. The output mappings can be stored in the input buffer of the next operator, e.g. aggregation. In the worst case, the invalidation has to go over all mappings in the buffer to check if any of them has expired. To avoid unnecessary checks, we use an auxiliary buffer with a one-way index (see one-way index in Section 3) to remember how many mappings generated from a leaf-mapping were inserted into the buffer. As a result, Algorithm 3 shows how this invalidation strategy is done. Line 2 assigns the number of mappings that were generated from the mapping μ to the variable *toBeInvalidated*. This number is retrieved from the index counter of the keyed entry corresponding to μ^-. This variable then is used to check if the all expired mappings derived from μ^- are already invalidated as the stop condition at lines 10-12. Lines 6-9 check if each μ^* in R is generated from μ^-, i.e., μ^* is also expired, to add it to $\mathcal{EXP}$ and remove it from R, and then increase variable *invalidated*. This new invalidation mechanism together with above data structures allow us to design algorithms for join, aggregation, duplicate elimination and negation operators as follows.

Algorithm 3: Invalidate expired mappings

Input: μ^-: negative leaf-mapping, R: an input buffer, $\mathcal{LM}$: a buffer for book-keeping the number of mappings of every leaf-mappings that generate all mapping in R

```
1  invalidated ← 0
2  toBeInvalidated ← LM.get(μ⁻).count()
3  EXP ← ∅
4  for μ* ∈ R // iterate over R in the inserting order
5  do
6      if μ* is generated from μ⁻ then
7          EXP.insert(μ*)
8          Remove μ* from R
9          invalidated + +
10     if invalidated==toBeInvalidated then
11         Remove μ⁻ from LM
12         return EXP;
```

4.1. Multiway Join

A multiway join query can be evaluated by trees of partially blocking, and pipelined binary-join operators. However, this approach is not sufficient for processing streaming inputs that need to dynamically reorganize the evaluation tree in response to the changes of stream data [6]. Therefore, similar to MJoin [26], we introduce a single multiway join that works over more than two input buffers. This multiway join generates and propagates results in a single step without having to pass these results through a multi-stage binary execution pipeline.

We extend the incremental equations [43, 8] to represent incremental semantics for the multiway join operator. The incremental evaluation of this operator is based on two equations for handling the events of new mapping arrival and mapping expiration. For a new mapping μ arriving in the input buffer R, the operator $\uplus$ is used to indicate this event and the evaluation

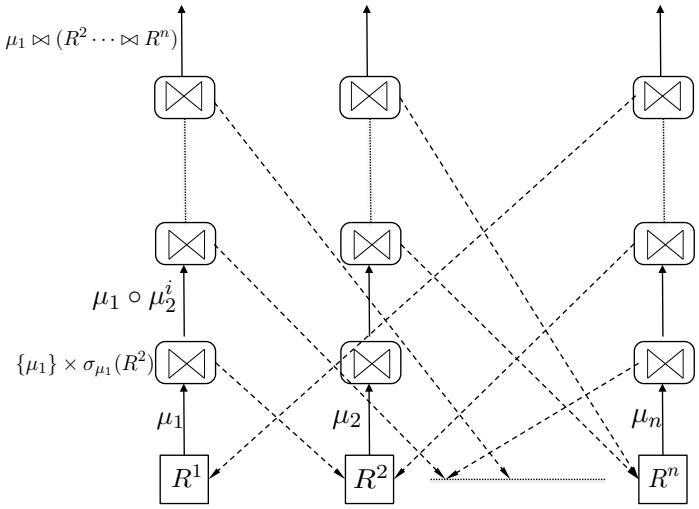


Figure 7: Multiway join process.

is denoted as $R \uplus \mu$. The evaluation for the event of expiring mapping μ in the input buffer R is represented as $R - \mu$. Because the multiway join is symmetric, without loss of generality, we present the incremental equations of n-way join as shown in Equations 1 and 2 where the inserting and expiring events happen in the input buffer R^1. In order to employ our ring-index for window buffers, we represent the incremental evaluation of these equations in Equation 3 using the select operator $\sigma_{\mu_1}(R^2)$. The operator $\sigma_{\mu_1}(R^2)$ returns all the mappings stored in the input buffer R^2 which are compatible with μ_1 [6]. This operator is supported by the high throughput *probe* method on the data structure that has a ring-index on the variables to check the compatibility with μ_1 .

$$(R^1 \uplus \mu_1) \bowtie (R^2 \cdots \bowtie R^n) = (R^1 \bowtie R^2 \cdots \bowtie R^n) \cup (\mu_1 \bowtie (R^2 \cdots \bowtie R^n)) \quad (1)$$

$$(R^1 - \mu_1) \bowtie (R^2 \cdots \bowtie R^n) = (R^1 \bowtie R^2 \cdots \bowtie R^n) \setminus (\mu_1 \bowtie (R^2 \cdots \bowtie R^n)) \quad (2)$$

$$\mu_1 \bowtie (R^2 \cdots \bowtie R^n) = (\{\mu_1\} \times \sigma_{\mu_1}(R^2)) \bowtie (R^3 \cdots \bowtie R^n) \quad (3)$$

The evaluation of a new mapping μ_1 inserted into the input buffer R^1 is illustrated in Figure 7. When a mapping μ_1 is inserted into the input buffer R^1, it will be used to probe one of the other input buffers $R^2 \cdots R^n$. Let us assume that R^2 is the next input buffer in the probing sequence [26]. For each mapping μ_2^i in R^2 that is compatible with μ_1, an intermediate joined mapping in the form $\mu_1 \circ \mu_2^i$ is generated. Subsequently, $\mu_1 \circ \mu_2^i$ is recursively used to probe the other input buffers to generate the final mappings. When a buffer that does not return any compatible mapping is found, the probing sequence stops. Algorithm 4 shows our incremental evaluation algorithm for a multiway join with n input buffers. Lines 2-4 handle new mappings and line 6 is for forwarding the negative mappings to the upper operator. Line 4 calls the recursive sub-routine *probingPropagate* given in Algorithm 5.

Algorithm 5 dynamically choose to the next buffer in each step for subsequent probing sequences to generate outputs from that stage (defined by μ, k sliding windows W[i$_1$],..,W[i$_k$]). The algorithm chooses the next buffer that generates minimal cost for the later propagating step from line 2 to line 8. To have a light-weight cost model, we heuristically estimate the cost of following probing steps from a chosen buffer by counting number

Algorithm 4: Multi-Way Join

```
   Input: n input buffers W1,..,Wn
1  if a new mapping μ arrives at window W[i] then
2      remove expired tuples from all windows
3      W[i].insert(μ)
4      probingPropagate(μ,{W[1],..,W[n]}\{W[i]})
5  else
6      propagate negative mapping to upper operator
```

Algorithm 5: Probing propagation *probingPropagate*

```
   Input: μ, k sliding windows {W[i1],..,W[ik]}
1  if k>0 then
2      i_next ← i1
3      minNextProb← ∞
4      for j ∈ [1..k] do
5          if μ shares joining variable with window W[i_j] then
6              if minNextProb<estimateNextProb(i_j) then
7                  minNextProb ← estimateNextProb(i_j)
8                  i_next ← i_j
9      for μ* ∈ W[i_next].probe(μ) do
10         probingPropagate(μ ∘ μ*, {W1,..,Wk}\{W[i_next]})
11 else
12     dispatch μ
```

of intermediate mappings to be generated in line 6. This can be easily done by using the counter of the indexes we introduced. The more sophisticated cost models such as [44, 28, 45, 46] can be extended from this point of the algorithm. Line 10 recursively calls *propbingPropagate* function to execute subsequent probing processes derived by a joined mapping $\mu \circ \mu^*$ which is generated from μ^* retrieved in Line 9.

To avoid the double computation issue of the negative tuple approach in an event of expiration, the outputs of a multiway join are stored in a buffer which is invalidated through the *invalidate* method of Algorithm 3. When an expired mapping arrives as a negative mapping, it is used as a parameter to call the *invalidate* method to find the expired outputs. This invalidating operation is usually done by the upper operator of the query pipeline, which consumes the multiway join output as its input buffer (e.g., aggregate and duplicate elimination operators). For example, Figure 8 illustrates a process of the incremental evaluation of our example query in Section 2.1. In this figure, we have the mappings arriving to 5 buffers according the timeline on the left. The first three are the buffers of 3 windows W_1,W_2 and W_3, the next one from left to right is output buffer of the join $W_1 \bowtie W_2 \bowtie W_3$. The join output buffer is then used compute the aggregation operator AVERAGE on the last buffer on the right. The schema of the mappings in each buffer are shown in the bottom according the variables of the corresponding patterns, e.g $\langle ?trans, ?fare, ?pTime\rangle$ and $\langle ?ride, ?pTime, ?taxi\rangle$. The red items with the minus signs are negative mappings of the counterparts without the minus sign. e.g $-\langle 1,4\rangle$ and $-\langle 2,4\rangle$ in the window buffer W_3 are the negative mappings of $\langle 1,4\rangle$ and $\langle 2,4\rangle$ generated at the time points 5 an 7 respectively. Let have a closer look at the time point 11 when the mapping $\langle 1,4\rangle$ in window W_3 is expired, we need to invalidate all mappings that were created by $\langle 1,4\rangle$ in the buffer $W_1 \bowtie W_2 \bowtie W_3$. In this example, we have

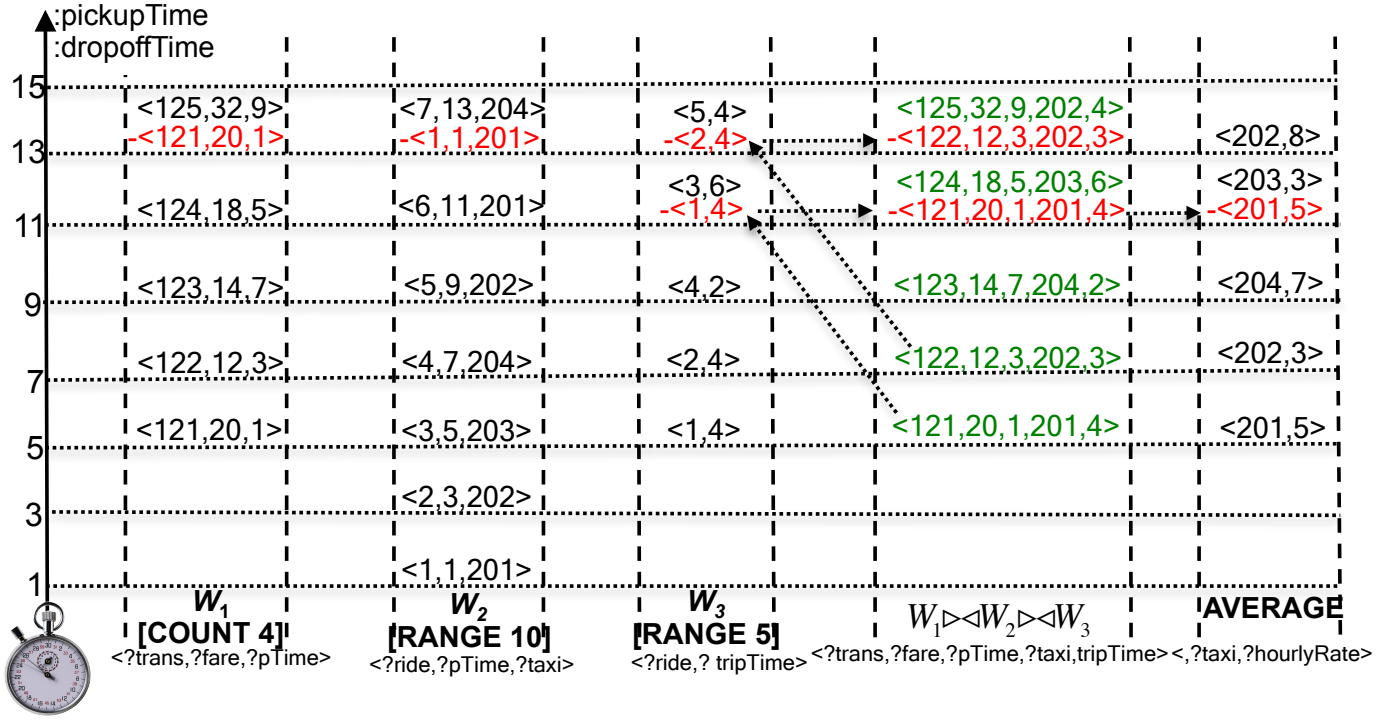


Figure 8: An example on a running process of join and aggregation

mapping $\langle 121, 20, 1, 201, 4\rangle$ created at the time point 5. In the negative tuple approach, we have to generate a negative tuple from $\langle 1, 4\rangle$ then join again to find mapping $\langle 121, 20, 1, 201, 4\rangle$ to signal as an expired tuple to AVERAGE to update the results. In our approach, we just keep the output of the join in the buffer $W_1 \bowtie W_2 \bowtie W_3$, then we signal the negative mapping $-\langle 1, 4\rangle$ to the join, it will forward to AVERAGE and then it will call the *invalidate* method of Algorithm 3 to find mapping $\langle 121, 20, 1, 201, 4\rangle$ via the parameter $-\langle 1, 4\rangle$.

4.2. Aggregation

An aggregation operator $\mathcal{AGG}_{f_1(A_1),f_2(A_2),...,f_k(A_k)}$ maps each input mapping to a group G and produces one output mapping for each non-empty group G. The output has the form $<G, Val_1, .., Val_k>$, where G is the group identifier and Val_i is the group's aggregate value of the function $f_i(A_i)$. The value Val_i is updated whenever the set of mappings in G changes in the case of new and expired mappings. From the example in Figure 8 we have AGG as AVERAGE operator with the function f_1 is computed from the two variables $?fare$ and $?tripleTime$. For sake of the brevity, we use the timescale of the timeline so that $?hourRate$ could be computed from average of $f_1 = ?fare/?tripleTime$ as shown in the mappings computed in the buffer of AVERAGE of Figure 8. In this example, we group the aggregation by the variable $?taxi$. Both new mappings and expired mappings can result in an update to the aggregate value of a group and the aggregation operator needs to report the new aggregate value for that group[8]. For example, at time point 13, we have two updates for the group on the taxi 202 corresponding to the invalidation of $-\langle 122, 12, 3, 202, 3\rangle$ and the arrival of $\langle 124, 32, 9, 202, 4\rangle$. The algorithm uses a one-way index for the input buffer with an extension to store aggregate values. The composite key generated from binding values of the "group by" variables is used as the identification of a group, e.g., $?taxi$.

The operator consumes the input mappings orderly, and multiple input mappings can contribute to the aggregate value of a group at a certain point in time, i.e., at a defined clock tick. For example, the aggregation operator consuming input mappings from a multiway join might have to update the aggregated value of a group several times before delivering the final update to the upper operator. Therefore, to signal that all input mappings from the lower operator at a particular clock tick have already been delivered, we use a special mapping, called *null mapping*. When an upper operator receives the null mapping, it can deliver its aggregated evaluation results for that clock tick. For example, at the time point 13 of Figure 8, the function aggregation is called twice to update the aggregation values for the taxi 202 corresponding to the invalidation of $-\langle 122, 12, 3, 202, 3\rangle$ and the arrival of $\langle 125, 32, 9, 202, 4\rangle$, the null mapping will signal the operator to put the final result $-\langle 202, 8\rangle$ for the clock tick 13.

In the Algorithm 6, we use a buffer to store all updates of aggregate values before they are dispatched to the output. Lines 1-3 handle the dispatching through the null mapping. Lines 5-16 handle the arrival of new mapping, e.g., the mappings in green in the buffer $W_1 \bowtie W_2 \bowtie W_3$. The algorithm uses a one-way index

Algorithm 6: Aggregation

Input: μ : dispatched mapping, $\mathcal{AGG}_{f_1(A_1),f_2(A_2),...,f_k(A_k)}$: aggregation functions , O: output buffer to be dispatched, $\mathcal{EXP}$: buffer of input mappings waiting to be invalidated

```
1  if μ is null mapping then
2      dispatch output buffer O
3      O ← ∅
4  else
5      if μ is a new mapping then
6          gKey ← generate the composite key on group
           bindings of μ
7          E ← EXP.index.get(gKey)
8          if E is not null then
9              E.increaseCount()
10         else
11             E ← create new one-way index entry for a new
               group
12         for i ∈ [1..k] do
13             A ← Recompute f_i(A_i) for adding new
               mapping μ
14             Update new aggregation value A to the
               aggregation entry E
15         O.insert(E)
16         EXP.insert(μ)
17     else
18         for μ⁻ ∈ EXP.invalidate(μ) do
19             gKey ← generate the composite key on group
               bindings of μ⁻
20             E ← EXP.index.get(gKey)
21             if E.count() > 0 then
22                 for i ∈ [1..k] do
23                     A ← Recompute f_i(A_i) for removing
                       expired mapping μ⁻
24                     Update new aggregation value A to the
                       aggregation entry E
25                 O.insert(E)
26             else
27                 EXP.index.remove(gKey)
```

for the input buffer with an extension to store aggregate values. The composite key is used as the identification of a group. Lines 6-11 find a group to update its current aggregate values. The actual update is done in lines 13-14. Lines 15 and 16 store the updates and add the new input mapping to the input buffer, respectively. As described in the beginning of Section 4 and the end of Section 4.1, the expired mappings are signalled by negative leaf mappings which are then used as parameters to invalidate the expired mappings as shown in line 18. Similar to updating aggregate values for new mappings, the updates for expired mappings are done in lines 19-27. The incremental evaluation of the process is demonstrated in step by step with the example data in the Figure 8.

[8]Note that we do not consider round-off or approximation approaches in updating aggregation values in the scope of this paper.

4.3. Duplicate elimination

The duplicate elimination, i.e., the DISTINCT operator, eliminates duplicates in the input buffer to return the distinct mappings to the output. Similar to the multiway join, the incremental evaluation Equations 4 and 5 describe how to incrementally evaluate the DISTINCT operator in the event of inserting a new mapping or removing an expired mapping. Equation 4 states that a new mapping μ only produces a new output if there is no duplicate of μ in the operator's input buffer. The opposite applies to expired mappings: the operator only outputs the expired mappings if μ is already in the input buffer. Hence, handling new and expired mappings can be done similarly to the aggregation operator.

$$\epsilon(R \uplus \mu) = \begin{cases} \epsilon(R) & \text{if } \mu \in \text{R} \\ \epsilon(R) \cup \{\mu\} & \text{Otherwise} \end{cases} \quad (4)$$

$$\epsilon(R - \mu) = \begin{cases} \epsilon(R) & \text{if } \mu \in \text{R} \\ \epsilon(R) \setminus \{\mu\} & \text{Otherwise} \end{cases} \quad (5)$$

Algorithm 7 shows the DISTINCT operator. As defined in Equation 4, handling the arrival of a mapping is similar to the case of an aggregate operator (lines 5-14). Lines 17-22 show how to handle expirations corresponding to Equation 5.

Algorithm 7: DISTINCT

```
Input: μ : dispatched mapping, O: output buffer to be
       dispatched, EXP: buffer of input mappings waiting
       to be invalidated
1  if μ is null mapping then
2      dispatch output buffer O
3      O ← ∅
4  else
5      if μ is a new mapping then
6          gKey ← generate the composite key for μ
7          E ← EXP.index.get(gKey)
8          if E is null then
9              E ← create a new entry
10             gIndex.insert(gKey,E)
11             O.insert(μ)
12         else
13             E.increaseCount()
14         EXP.insert(μ)
15     else
16         for μ⁻ ∈ EXP.invalidate(μ) do
17             gKey ← generate the composite key for μ⁻
18             E ← EXP.index.get(gKey)
19             if E.count() > 1 then
20                 E.decreaseCount()
21             else
22                 EXP.index.remove(gKey)
```

4.4. Negation

A negation operator between two input buffers, R^1 and R^2, produces mappings that are in R^1 and are not compatible with any mappings in R^2. The negation operator ($R^1 \setminus R^2$) is asymmetric because handling new or expired mappings depends on whether the mapping is from R^1 or R^2. In case of a new mapping arriving at R^1, incremental evaluation Equation 6 says that if μ is compatible with a mapping of R^2 (denoted as $\mu \cong R^2$) then the output is unchanged, otherwise, μ will be added as a new mapping in the output. On the other hand, if the new mapping μ arrives at R^2, all μ' in R^1 that is compatible with μ will the be removed from the current output as specified in the incremental evaluation Equation 7. Similarly, we have incremental evaluation Equations 8 and 9 that specify the how to incrementally evaluate an expired mapping μ signed at R^1 and R^2 correspondingly.

$$(R^1 \uplus \mu) \setminus R^2 = \begin{cases} (R^1 \setminus R^2) & \text{if } \mu \cong R^2 \\ (R^1 \setminus R^2) \cup \mu & \text{Otherwise} \end{cases} \quad (6)$$

$$R^1 \setminus (R^2 \uplus \mu) = (R^1 \setminus R^2) \setminus \{\mu' \mid \mu' \in R^1 \wedge \mu \cong \mu'\} \quad (7)$$

$$(R^1 - \mu) \setminus R^2 = \begin{cases} (R^1 \setminus R^2) \cup \mu & \text{if } \mu \cong R^2 \\ (R^1 \setminus R^2) & \text{Otherwise} \end{cases} \quad (8)$$

$$R^1 \setminus (R^2 - \mu) = (R^1 \setminus R^2) \cup \{\mu' \mid \mu' \in R^1 \wedge \mu \cong \mu'\} \quad (9)$$

Algorithm 8 presents our implementation for the negation operator. This algorithm is also similar to the aggregation operator in handling the input mappings. For a new mapping, line 6 checks if it comes from the left or right side. Depending on the result, the mapping is then evaluated using either Equation 6 (lines 7-13) or Equation 7 (lines 15-17). Similarly, lines 21-22 and 25-29 implement Equation 8 and Equation 9 respectively. This algorithm uses one-way indices for the input and the output buffers. The indexing keys are generated as composite keys from shared binding values between the two input buffers. These keys are used to check the existence of compatible mappings from each input buffer.

5. Experimental evaluations

To evaluate the performance and efficiency of our data structures and algorithms we conducted two types of experiments: *Performance of indices*, *Throughput of stateful operators*, textit-Performance in real-life scenarios and *Memory footprint*. The first two are to evaluate indices and specific operators with randomized data. The latter is to evaluate how these operators perform in a real engine with realistic scenarios. Finally, we also compare the memory footprint of our approach with the rival approaches. The experiments are executed on a standard workstation with Debian Squeeze AMD64 2x E5606 Intel Quad-Core Xeon 2.13GHz with 16GB RAM, running the Java SE Runtime Environment (build 1.7.0_51-b13) Java HotSpot 64-Bit Server VM.

5.1. Performance of indices

To evaluate the performance of our indices on different data structures, we experimented with the implementations of an AVL tree, a red-black tree and a hash table. In the first experiment shown in Figure 9, we measured the throughputs of the insert operation on different numbers of keys and window sizes for each implementation of the ring index. In another experiment shown in Figure 10, we measured the average probing time of

Algorithm 8: Negation

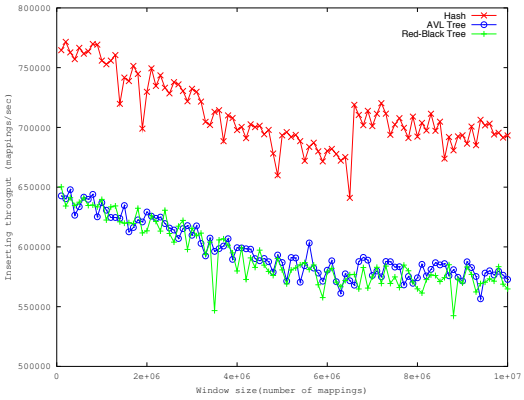
Input: μ : dispatched mapping, O: output buffer to be dispatched, $\mathcal{EXP}^L$: left buffer of input mappings waiting to be invalidated, $\mathcal{EXP}^R$: right buffer of input mappings waiting to be invalidated

```
1  if μ is null mapping then
2      dispatch output buffer O
3      O ← ∅
4  else
5      if μ is a new mapping then
6          if μ is from the left input then
7              gKey ← generate the composite key for μ
8              E ← EXP^R.index.get(gKey)
9              if E is null then
10                 O.insert(μ)
11             else
12                 E.increaseCount()
13             EXP^L.insert(μ)
14         else
15             gKey ← generate the composite key for μ
16             O.purgeByKey(gKey)
17             EXP^R.insert(μ)
18     else
19         if μ is from the left input then
20             for μ⁻ ∈ EXP^L.invalidate(μ) do
21                 gKey ← generate the composite key for μ⁻
22                 O.purgeByKey(gKey);
23         else
24             for μ⁻ ∈ EXP^R.invalidate(μ) do
25                 gKey ← generate the composite key for μ⁻
26                 E ← EXP^R.index.get(gKey)
27                 if E is null then
28                     for μ⁺ ∈ EXP^L.probe(gKey) do
29                         O.insert(μ⁺)
```

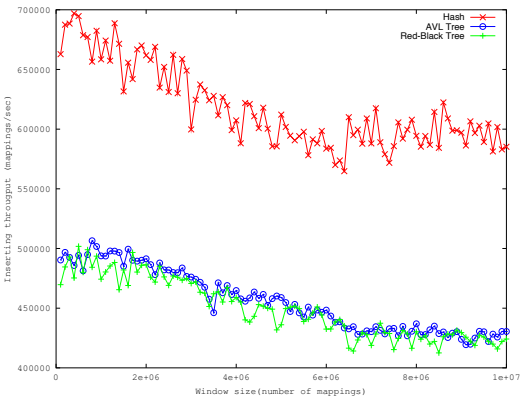
these three implementations of the ring index on window buffers with 1 million mappings with different numbers of distinct keys.

In these experiments, we set the load factor at 75% which is default load factor of most hash table implementations. Figure 9 shows that the hash table is the winner over the AVL tree and the red-black tree for the inserting operation. With the throughputs ranging from 400,000 to 800,000 mappings/second for the window size up to 10 million mappings, the ring index meets our requirement of a low maintenance cost and high throughput indexing mechanism. Figure 10 shows that the hash table also delivers a faster probing time for the ring index implementation. The average probing time on a 1-million-mapping window is consistently around 600-800 nano seconds. Intuitively, a nano second is the cycle time of a 1GHz processor. This fast and stable search time is the performance ground for the incremental evaluation algorithms introduced in Section 4.

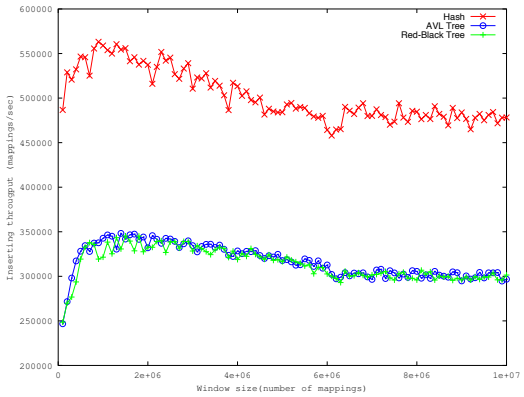
Next, we conducted experiments to analyse the behaviour



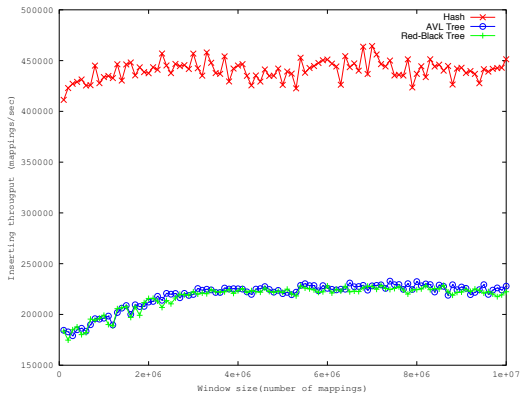
(a) Insert throughput on 1000 keys



(b) Insert throughput on 10000 keys



(c) Insert throughput on 100000 keys



(d) Insert throughput on 1000000 keys

Figure 9: Insert throughput with different number of keys.

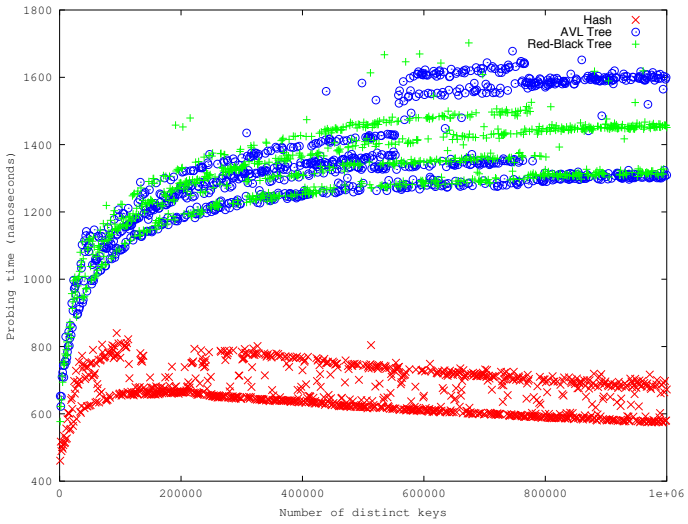


Figure 10: Probing time of an AVL/Red-black Tree/hash index on a 1 million mapping window.

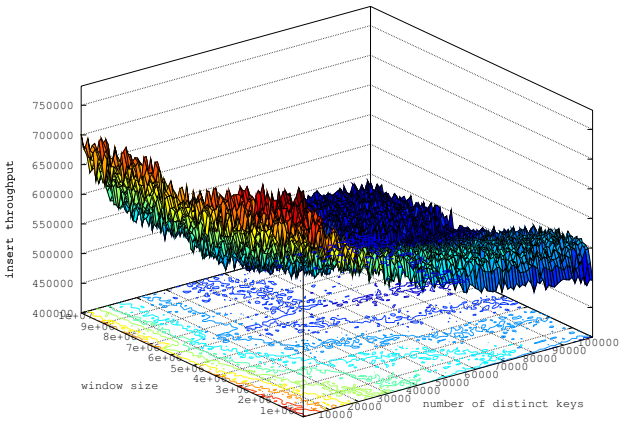


Figure 11: Insert throughput on windows with hash index.

of the inserting and probing operations on the window buffers which use a hash table for the ring index. We varied the number of distinct keys in the windows as well as the window size then we recorded the throughput of inserting and probing operations. The number of keys and window sizes range from 10,000 to 1 million. Figure 11 and Figure 12 show the throughput of inserting and probing operations. Both figures use the contour maps to represent the clusters of throughputs. They show that the throughputs are gradually clustered into groups that have the window sizes in a certain range of number of keys. The convergence of throughputs is clearer when the ratio of the number of keys to the window size is smaller. This might be used by an algorithm to estimate the throughputs of operations by monitoring the number of distinct keys in the windows. The throughput estimation is used in rate-based optimisation algorithms [44] to maximise the output rate in a fluctuating stream rate setting.

5.2. Throughput of stateful operators

To evaluate the operator-aware data structures accompanying the algorithms, we compare with their counterpart which uses the re-evaluation method and relational tables in CQELS in

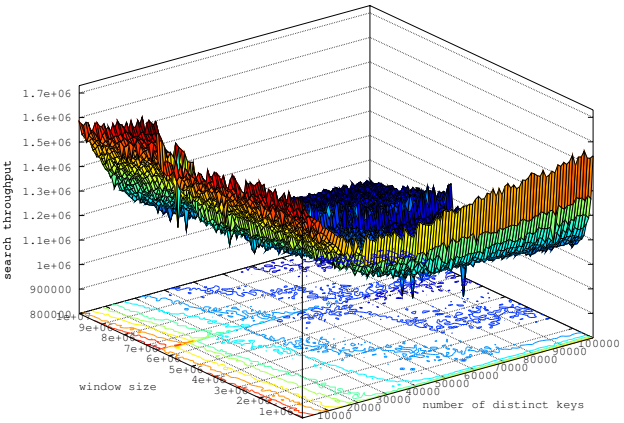


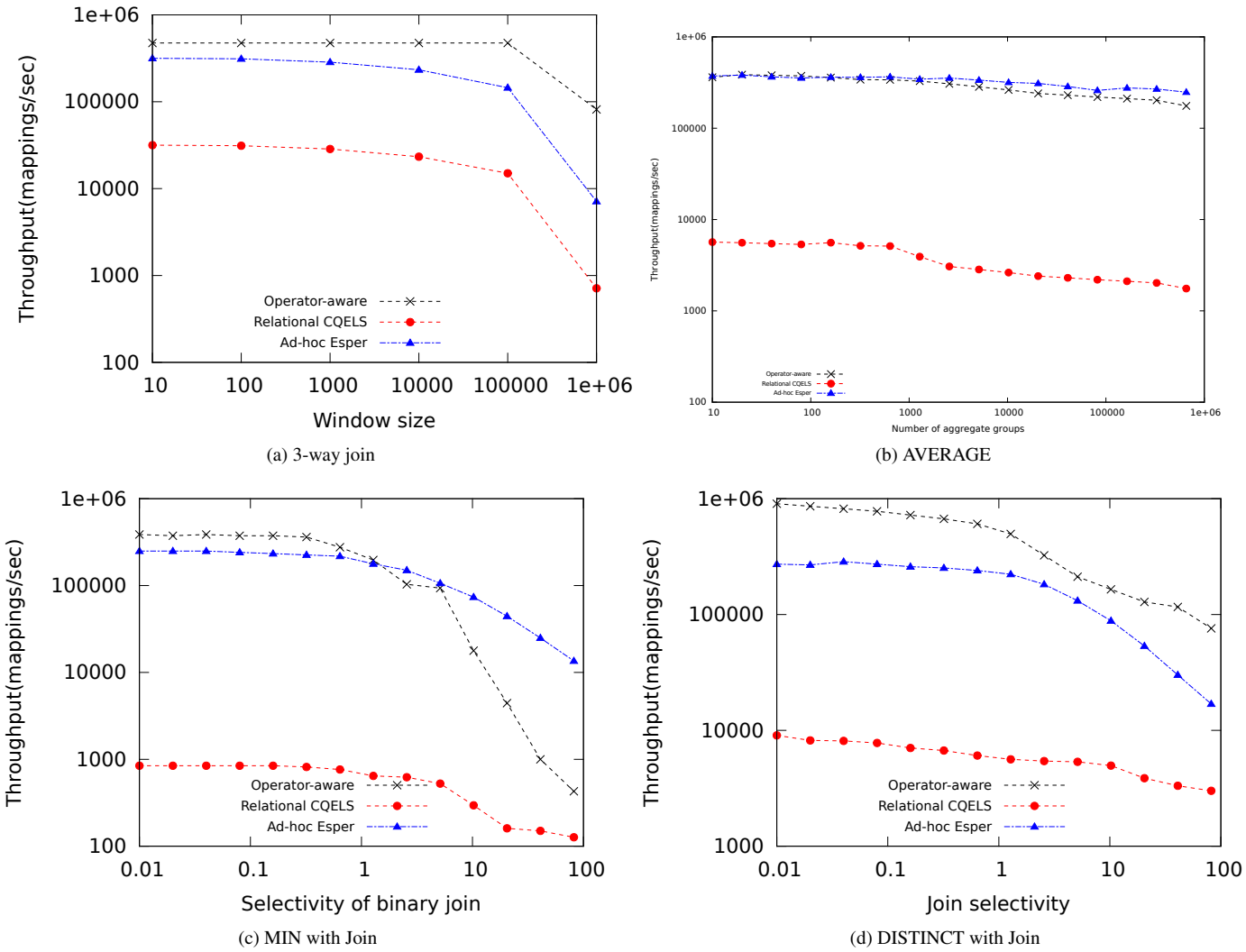
Figure 12: Probing throughput on windows with hash index.

terms of throughput, called Relational CQELS (R-CQELS). The throughput in this experiment is defined as the average number of input elements that a system can process in a unit of time. We also compare them against corresponding algorithms implemented in ESPER with ad-hoc data structures as ESPER is the underlying stream processing engine of C-SPARQL. These ad-hoc data structures are manually programmed as Java classes with attributes for storing encoded versions of binding values of a mapping. Therefore, they are more compact than our data structures and faster to access their binding values. Nevertheless, the experiments below clearly show that our generally applicable data structures and algorithms are significantly faster in most of the cases.

In the first test, we simulate the input data in the encoded form (fixed-size integer of a binding value) which is randomly generated. For the multi-way join algorithm, we tested a 3-way join on window buffers with different window sizes which have 10,000 distinct values in the join predicates. The results reported in Figure 13a show, with different window sizes, our algorithm consistently has the best throughputs in comparison to the relation-based algorithms used in CQELS and ad-hoc algorithms implemented in ESPER.[9] The new data structures and the multiway algorithm improve the join performance by orders of magnitude, in comparison with the relation-based implementation.

Next, we measured the aggregation operator. For the simple case, we tested AVERAGE function placed right after a window of 1 million of mapping with different numbers of aggregate group. For the complicated aggregation case, we choose the MIN function consuming the results of a binary join which joins two windows with one million mappings each. Similar to the above two experiments, the input data is also randomly generated but varying the join selectivity between 0.01 and 100. The join selectivity is the ratio of the number of outputs to the number of inputs of a join operator. Figure 13c reports the throughputs according to the selectivity. The results in Figure 13b and Figure 13c show that our algorithm performs better than the relation-based algorithm by orders of magnitude in both cases. On the simple case, our algorithm is slightly slower than the ad-hoc one. On the complicated case, when the selectivity is

[9] Please note that all charts use logarithmic scales.



(a) 3-way join

(b) AVERAGE

(c) MIN with Join

(d) DISTINCT with Join

Figure 13: Throughputs of the stateful operators

≤ 1, our method delivers the best throughput. However, the ad-hoc algorithm outperforms our algorithm if the selectivity is > 1. The reason for this is that, as introduced earlier, the manually programmed ad-hoc data structures are more compact (each attribute stores an encoded version of a binding value of a mapping) than our tree-based data structures. This leads to faster access to its binding value. Furthermore, our algorithm needs to traverse a tree to check the expirations. Additionally, a greater number of input mappings due to greater selectivity requires more scan operations for finding the expired mappings in the input buffer of the aggregate operator.

In the same manner of the aggregate operator, we evaluated the duplication elimination algorithm when it consumes the output of a binary join. The results are shown in Figure 13d. Our algorithm delivers the best throughput while the relation-based and re-evaluation implementation give much poorer performance. This explains the drawback of re-evaluation in the distinct operator, because the majority of such re-computations can be ignored if fast indices for the processing state are available.

5.3. Performance in real-life scenarios

To evaluate how our data structures and algorithms perform in a practical stream engine, we integrated them into the CQELS execution framework [47] which was used to develop the relational CQELS engine [6] with the re-evaluation method and relational data structures (R-CQELS). The CQELS engine with operator-aware data structures and associated algorithms is simply called CQELS [10]. In this experiment, we compare the performance of CQELS against R-CQELS on two popular benchmarking systems SRBench [48] and LSBench [49].

SRBench uses a real-world sensor data set, i.e., LinkedSensorData[11] linked with GeoNames[12] and DBpedia[13] associated with with a set queries to capture a wide range of use cases. In below experiment, we evaluate a subset of those queries that above targeted engines supported, i.e., Q1, Q4, Q5, Q8 and Q10. On the other hand, LSBench simulates a realistic stream data for a social network. Its data generator generates stream data according to the graph-based stream data schema and sliding window parameters. This benchmark includes a set of continuous queries with the increasing complexities. Similar to SRBench, we pick Q1, Q2, Q3, Q4, Q5, Q6 and Q10 which are 7 queries all supported

[10] The latest open source release of CQELS can be found at `http://cqels.org/`

[11] `http://wiki.knoesis.org/index.php/LinkedSensorData`

[12] `http://www.geonames.org/ontology/`

[13] `http://wiki.dbpedia.org/`

	SRBench (triples/sec)					LSBench (triples/sec)						
	Q_1	Q_4	Q_5	Q_8	Q_{10}	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_{10}
R-CQELS	1214	820	47	1774	3343	24122	8462	9828	1304	7459	3491	2326
CQELS	25147	20161	13966	22278	29463	118924	96789	88647	60467	52890	44391	103698

Table 1: Execution throughput on realistic scenarios

by mentioned engines to conduct the performance experiment.

To get the correct evaluation results, we modified the stream elements of benchmarking queries of SRBench and LSBench from time sliding windows to triple-based/count-based windows. With SRBench, we chose the largest stream dataset to stream to CQELS and R-CQELS, i.e. the Ike storm and measured queries they support. Before conducting the test, we computed the statistics for the basic triple patterns of these queries on the testing stream. Statistics show that the triples matching the patterns with the constant *weather:SnowfallObservation* rarely appear. Therefore, to get a clearer picture of the performance result whereas not changing the semantic of the test, we changed this constant to a different one to have significantly higher load on the queries with such patterns. For Q8 and Q10 which involve the static data set, we chose the random coordinates that appear in the corresponding static dataset. With LSBench, we adjusted the parameters of the LSBench data generator to guarantee that all operators in the testing query pipelines are triggered and have fair shares in the query load.

The results of SRBench and LSBench in Table 1 show that the CQELS engine outperforms R-CQELS by several orders of magnitude. The maximum execution throughput of CQELS is over 20-200 times higher than the throughput of R-CQELS. Especially, for the queries which have complicated join patterns such as Q4 and Q5, CQELS shows its advantage for join operators. For LSBench, the reported result indicates that CQELS performs more than 10 times faster than R-CQELS queries tested. Hence, both benchmarks confirm that operator-aware approach can accelerate the relation-based and re-evaluation-based implementation by an order of magnitude.

5.4. Memory Footprint

To show the impact of our operator-aware approach on memory footprint, we compared the memory consumption of our multiway-join algorithm with the counter-parts with a similar setup for the experiment shown in Figure 13a. We increased the number of windows in each test from 2 to 8 corresponding from 2-way join to 8-way join. Each window is a count-based window with the size of 10000. We also recorded the memory footprint of CQELS and R-CQELS on each test of the experiments on SRBench and LSBench. The results in Table 2 show that ESPER consumes less memory than CQELS on 2-way and 3-way joins. However, CQELS is a clear winner among EPSER and R-CQELS for N-way joins with N≥4. Furthermore, the bigger N is, the more memory CQELS could save in comparison to EPSER and R-CQELS. In all tests reported, CQELS consumes more than two-fold less memory to that of R-CQELS on the queries that have more than 4 query patterns, e.g. all tested queries on SRBench. In the other hand, CQELS and R-CQELS consume roughly the same amount memory on the simple queries like Q1, Q2, Q3 and Q10. In overall, the more complicated queries are, the larger ratio of memory saving CQELS can gain over R-CQELS. This effect is aligned with the fact that our storage design for mapping only saves space on intermediate mappings. Additionally, the more complicated the query pipeline is, the more intermediate mappings are generated.

6. Related Work

For continuously executing operators over streams, there are two main execution strategies: *eager execution* and *periodic execution* [9]. The eager execution strategy generates new results every time a new stream element arrives. However, this might be infeasible in the situations where streams have high arrival rates. The periodic execution executes the query periodically [14, 50]. In this case, sliding windows may be advanced and queries are evaluated periodically with a specified frequency [7, 51, 52, 53, 54, 55, 56, 57]. 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 updates. 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. Hence, in the scope of this paper, CQELS uses the eager execution strategy which is different from some other RSP engines like C-SPARQL and SPARQL$_{stream}$ [58] which use the periodic execution strategy .

A continuous query contains sliding windows to deal with the unbounded nature of data streams. When the window slides, the query is continuously computed to reflect both new tuples entering the windows and old tuples expiring from the window. Two methods for continuous computation are query *re-evaluation* and *incremental evaluation*. In the query re-evaluation method, two consecutive evaluations of a query are carried out in two independent query pipelines. For example, Aurora [7] and Borealis [59] use the re-evaluation method. On the other hand, the query pipeline of the query incremental evaluation method only processes changes in the window to produce new answer when the window slides. The incremental operators are used in the pipeline to process both new and expired tuples to produce incremental changes to the output stream of a query [8]. The incremental evaluation method is used in STREAM [60] and Nile [61]. The query re-evaluation method is easier to implement in comparison to the query incremental evaluation because it can reuse the operator implementations of relational databases. For instance, this method is used to implement the first version of the CQELS [6] engine which suffers significantly poorer performance as shown in Section 5, i.e., R-CQELS. On the other hand, the query incremental evaluation method is widely adopted in stream processing systems because it can avoid redundant computation to deliver better performance.

	Multiway Join (MB)					SRBench (MB)					LSBench (MB)						
	2	3	4	6	8	Q_1	Q_4	Q_5	Q_8	Q_{10}	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_{10}
R-CQELS	25.61	28.67	38.95	49.24	54.61	457	745	834	620	488	385	404	420	490	502	560	420
CQELS	12.36	13.41	14.74	16.95	18.69	206	327	370	248	218	374	370	380	398	389	402	370
ESPER	8.93	12.04	15.13	21.26	27.44												

Table 2: Memory footprint

The query incremental evaluation needs to handle two types of events: arrival of new stream elements and expiration of old stream elements [36]. The actions taken upon an arrival and an expiration vary across operators [29, 62]. 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 lazily (e.g., join).

As highlighted in Section 2.3, there are several shortcomings in two main techniques to signal expirations: *direct timestamp* [15, 36], and *negative tuple* [15, 36, 21]. Next, we review some other issues and relevant aspects of using such techniques. 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 this window. The negative tuples are used to signal expirations by propagating them through the query pipeline to trigger the operators to invalidate expired tuples in their state. The basic idea of this technique is attractive but may not be practical due to the fact that generating a negative tuple for every expired input doubles the number of tuples through the query pipeline. Therefore, the overhead of processing tuples through various operators such as join is doubled [8]. For queries without negation operations, base tuples or intermediate results are associated with extra timestamps to explicitly specify when the tuples are expired, called expiration timestamps. Therefore, the expiration of each tuple can be checked directly based on the expiration timestamp. This technique does not incur the overhead of negative tuples and does not have to store the base windows referenced in the query. However, as shown in [29, 8], this technique might cause some errors in the output results and it might be slower than the negative-tuple technique. Such shortcomings and issues also apply to RDF Stream Processing, hence, it motivated us to systematically analyse and address them in Section 2, Section 3 and Section 4.

7. Conclusions

In this paper we proposed novel operator-aware data structures associated with efficient incremental-evaluation algorithms to deal with realistic RDF stream data and query patterns. Our data structures are designed to handle small data items and intermediate mappings contained in the processing state flexibly and very efficiently. The data structures include various indices to support high throughput in the probing operations that are used in the different operator implementations at very low maintenance costs. Based on these new data structures, we propose several algorithms to enable incremental evaluation of basic operators such as join, aggregation, duplicate elimination, and negation. The algorithms are superior to other popular approaches such as direct-timestamp and negative tuple in terms of memory footprint, performance and scalability, especially with a processing state containing exceptionally long lists of very small data items. Our experiments show a performance gain of orders of magnitude to re-evaluation and relation-based approaches in all operator implementations with our data structures.

Acknowledgements

This publication has emanated from research supported in part by Irish Research Council under Grant No. GOIPD/2013/104 and Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289. I would like to thank Prof. Manfred Hauswirth (TU Berlin), Dr. Josiane Xavier Parreira (Siemens AG.) and Dr. Minh Dao Tran (TU Vienna) for valuable comments and recommendations on the several parts of this article.

References

[1] D. Le-Phuoc, J. Xavier Parreira, M. Hauswirth, Linked stream data processing, in: T. Eiter, T. Krennwallner (Eds.), Reasoning Web. Semantic Technologies for Advanced Query Answering, Vol. 7487 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 245–289.

[2] A. Harth, J. Umbrich, A. Hogan, S. Decker, Yars2: a federated repository for querying graph structured data from the web, in: Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference, ISWC'07/ASWC'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 211–224.

[3] C. Weiss, P. Karras, A. Bernstein, Hexastore: sextuple indexing for semantic web data management, Proc. VLDB Endow. 1 (1) (2008) 1008–1019.

[4] T. Neumann, G. Weikum, The rdf-3x engine for scalable management of rdf data, The VLDB Journal 19 (2010) 91–113.

[5] D. F. Barbieri, D. Braga, S. Ceri, M. Grossniklaus, An execution environment for c-sparql queries, in: Proceedings of the 13th International Conference on Extending Database Technology, EDBT '10, ACM, New York, NY, USA, 2010, pp. 441–452.

[6] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: Proceedings of 10th International Semantic Web Conference, 2011, pp. 370–388.

[7] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: a new model and architecture for data stream management, The VLDB Journal 12 (2) (2003) 120–139.

[8] T. Ghanem, M. Hammad, M. Mokbel, W. Aref, A. Elmagarmid, Incremental evaluation of sliding-window queries over data streams, Knowledge and Data Engineering, IEEE Transactions on 19 (1) (2007) 57 –72.

[9] L. Golab, M. T. Özsu, Processing sliding window multi-joins in continuous queries over data streams, in: Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '03, VLDB Endowment, 2003, pp. 500–511.

[10] J. Kang, J. F. Naughton, S. Viglas, Evaluating window joins over unbounded streams, in: Proceedings of the 19th International Conference on Data Engineering, ICDE'03, 2003, pp. 341–352.

[11] L. Ding, E. A. Rundensteiner, Evaluating window joins over punctuated streams, in: Proceedings of the thirteenth ACM international conference on Information and knowledge management, CIKM '04, ACM, New York, NY, USA, 2004, pp. 98–107.

[12] W. Wang, J. Li, D. Zhang, L. Guo, Processing sliding window join aggregate in continuous queries over data streams, in: Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, 2004, pp. 348–363.

[13] X. Lin, Y. Yuan, W. Wang, H. Lu, Stabbing the sky: Efficient skyline computation over sliding windows, in: Proceedings of the 21st International Conference on Data Engineering, ICDE '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 502–513.

[14] A. Arasu, J. Widom, Resource sharing in continuous sliding-window aggregates, in: Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, VLDB '04, VLDB Endowment, 2004, pp. 336–347.

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

[16] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Trans. Database Syst. 34 (3) (2009) 1–45.

[17] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable semantic web data management using vertical partitioning, in: Proceedings of the 33rd international conference on Very large data bases, VLDB '07, VLDB Endowment, 2007, pp. 411–422.

[18] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: A generic architecture for storing and querying rdf and rdf schema, in: International Semantic Web Conference, 2002, pp. 54–68.

[19] K. Wilkinson, C. Sayers, H. A. Kuno, D. Reynolds, Efficient rdf storage and retrieval in jena2, in: SWDB, 2003, pp. 131–150.

[20] E. I. Chong, S. Das, G. Eadon, J. Srinivasan, An efficient sql-based rdf querying scheme, in: Proceedings of the 31st international conference on Very large data bases, VLDB '05, VLDB Endowment, 2005, pp. 1216–1227.

[21] L. Golab, M. T. Özsu, Data stream management, Synthesis Lectures on Data Managemen (2010) 1–73.

[22] M. A. Hammad, W. G. Aref, A. K. Elmagarmid, Stream window join: tracking moving objects in sensor-network databases, in: Proceedings of the 15th International Conference on Scientific and Statistical Database Management, SSDBM '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 75–84.

[23] M. A. Hammad, W. G. Aref, A. K. Elmagarmid, 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, Lawrence Berkeley Laboratory, Berkeley, CA, US, 2005, pp. 143–146.

[24] J.-P. Dittrich, B. Seeger, D. S. Taylor, P. Widmayer, 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 '02, VLDB Endowment, 2002, pp. 299–310.

[25] G. Luo, J. Naughton, C. Ellmann, A non-blocking parallel spatial join algorithm, in: Proceedings. 18th International Conference on Data Engineering, ICDE '02, 2002, pp. 697 –705.

[26] S. D. Viglas, J. F. Naughton, J. Burger, 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 - Volume 29, VLDB '03, VLDB Endowment, 2003, pp. 285–296.

[27] M. F. Mokbel, M. Lu, W. G. Aref, 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 '04, IEEE Computer Society, Washington, DC, USA, 2004.

[28] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, N. Mamoulis, 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 '05, ACM, New York, NY, USA, 2005, pp. 371–382.

[29] M. Hammad, W. G. Aref, M. J. Franklin, M. F. Mokbel, A. K. Elmagarmid, Efficient execution of sliding-window queries over data streams (2003).

[30] L. Golab, M. T. Özsu, Update-pattern-aware modeling and processing of continuous queries, in: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 658–669.

[31] Y. Tao, D. Papadias, Maintaining sliding window skylines on data streams, IEEE Trans. on Knowl. and Data Eng. 18 (3) (2006) 377–391.

[32] K. Mouratidis, S. Bakiras, D. Papadias, Continuous monitoring of top-k queries over sliding windows, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SIGMOD '06, ACM, New York, NY, USA, 2006, pp. 635–646.

[33] J. M. Hellerstein, P. J. Haas, H. J. Wang, Online aggregation, SIGMOD Rec. 26 (2) (1997) 171–182.

[34] H. Wang, C. Zaniolo, C. R. Luo, 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 - Volume 29, VLDB '03, VLDB Endowment, 2003, pp. 1113–1116.

[35] Y.-N. Law, H. Wang, C. Zaniolo, Query languages and data models for database sequences and data streams, in: Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, VLDB '04, VLDB Endowment, 2004, pp. 492–503.

[36] L. Golab, Sliding window query processing over data streams, Ph.D. thesis, University of Waterloo, Waterloo,Ontario,Canada (2006).
URL `http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-27.pdf`

[37] A. Owens, An investigation into improving rdf store performance, Ph.D. thesis, University of Southampton (April 2011).
URL `http://eprints.soton.ac.uk/272232/`

[38] J. Krämer, B. Seeger, A temporal foundation for continuous queries over data streams, in: Proceedings of the Eleventh International Conference on Management of Data (COMAD 2005), 2005, pp. 70–82.

[39] P. A. Boncz, M. Zukowski, N. Nes, Monetdb/x100: Hyper-pipelining query execution, in: CIDR, 2005, pp. 225–237.

[40] C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, Picodmbs: Scaling down database techniques for the smartcard, in: Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, pp. 11–20.

[41] L. Golab, S. Garg, M. T. Özsu, On indexing sliding windows over online data streams, in: EDBT, 2004, pp. 712–729.

[42] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, 2nd Edition, McGraw-Hill Higher Education, 2001.

[43] T. Griffin, L. Libkin, Incremental maintenance of views with duplicates, in: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, SIGMOD '95, ACM, New York, NY, USA, 1995, pp. 328–339.

[44] S. D. Viglas, J. F. Naughton, Rate-based query optimization for streaming information sources, in: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, SIGMOD '02, ACM, New York, NY, USA, 2002, pp. 37–48.

[45] M. Cammert, J. Krämer, B. Seeger, S. Vaupel, A cost-based approach to adaptive resource management in data stream systems, IEEE Trans. Knowl. Data Eng. 20 (2) (2008) 230–245.

[46] Y. Mei, S. Madden, Zstream: a cost-based query processor for adaptively detecting composite events, in: Proceedings of the 35th SIGMOD international conference on Management of data, SIGMOD '09, ACM, New York, NY, USA, 2009, pp. 193–206.

[47] D. Le Phuoc, A native and adaptive approach for linked stream processing, Ph.D. thesis, National University of Ireland, Galway (2013).
URL `http://hdl.handle.net/10379/3589`

[48] Y. Zhang, P. M. Duc, O. Corcho, J.-P. Calbimonte, Srbench: A streaming rdf/sparql benchmark, in: Proceedings of the 11th International Conference on The Semantic Web - Volume Part I, ISWC'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 641–657.

[49] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, M. Fink, Linked stream data processing: Facts and figures, in: Proceedings of 11th International Semantic Web Conference, 2012.

[50] S. Chandrasekaran, M. J. Franklin, Psoup: a system for streaming queries over streaming data, The VLDB Journal 12 (2) (2003) 140–156.

[51] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for Internet databases, SIGMOD Rec. 29 (2) (2000) 379–390.

[52] S. Krishnamurthy, C. Wu, M. Franklin, On-the-fly sharing for streamed aggregation, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SIGMOD '06, ACM, New York, NY, USA, 2006, pp. 623–634.

[53] J. Li, D. Maier, K. Tufte, V. Papadimos, P. A. Tucker, Semantics and evaluation techniques for window aggregates in data streams, in: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 311–322.

[54] L. Liu, C. Pu, W. Tang, Continual queries for internet scale event-driven information delivery, IEEE Trans. on Knowl. and Data Eng. 11 (4) (1999)

610–628.

[55] N. Shivakumar, H. García-Molina, Wave-indices: indexing evolving databases, in: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, SIGMOD '97, ACM, New York, NY, USA, 1997, pp. 381–392.

[56] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. A. Shah, Telegraphcq: Continuous dataflow processing for an uncertain world, in: First Biennial Conference on Innovative Data Systems Research (CIDR'03), 2003.

[57] D. Zhang, J. Li, Z. Zhang, W. Wang, L. Guo, Dynamic adjustment of sliding windows over data streams, in: Advances in Web-Age Information Management: 5th International Conference, WAIM 2004, 2004, pp. 24–33.

[58] J.-P. Calbimonte, O. Corcho, A. J. G. Gray, Enabling ontology-based access to streaming data sources, in: Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I, ISWC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 96–111.

[59] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. B. Zdonik, The design of the borealis stream processing engine, in: Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), 2005, pp. 277–289.

[60] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, J. Widom, Stream: the stanford stream data manager (demonstration description), in: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, SIGMOD '03, ACM, New York, NY, USA, 2003, pp. 665–665.

[61] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, X. Xiong, Nile: A query processing engine for data streams, in: Proceedings of the 20th International Conference on Data Engineering, ICDE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 851–.

[62] h. Vossough, J. R. Getta, Processing of continuous queries over unlimited data streams, in: Proceedings of the 13th International Conference on Database and Expert Systems Applications, DEXA '02, Springer-Verlag, London, UK, UK, 2002, pp. 799–809.

View publication stats