DEBS Grand Challenge: RDF Stream Processing with CQELS Framework for Real-time Analysis

Danh Le Phuoc
Insight Centre for Data Analytics
National University of Ireland Galway
danh.lephuoc@nuigalway.ie

Minh Dao-Tran
Institute of Information Systems
Vienna University of Technology
dao@kr.tuwien.ac.at

Anh Le Tuan
Insight Centre for Data Analytics
National University of Ireland Galway
anh.le@insight-centre.org

Manh Nguyen Duc
Insight Centre for Data Analytics
National University of Ireland Galway
ducmanh.nguyen@insight-centre.org

Manfred Hauswirth
Institut für Telekommunikationssysteme
Technische Universität Berlin
Berlin, Germany
manfred.hauswirth@tu-berlin.de

ABSTRACT

This paper presents a solution to the Grand Challenge using CQELS (Continuous Query Evaluation over Linked Stream), a general execution framework to build RDF Stream Processing engines to answer continuous analytical queries. It provides an efficient execution architecture whereby incremental computing algorithms can be implemented to boost the performance.

Our experimental results show strong effects of the implemented approach as CQELS outperforms a base-line implementation which recomputes on every incoming input.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Dictionaries, Indexing methods*; D.1.0 [**Programming Techniques**]: General

General Terms

Algorithms, Experimentation, Performance

Keywords

RDF Stream Processing, Real-time Analysis

1. INTRODUCTION

The DEBS Grand Challenge 2015 [12] is based on an open data set on taxi trip reports from New York City. The goal of the challenge is to develop stream-processing systems to support real-time analytics over this high volume geospatial data stream. In particular, two queries to challenge the participants are: (Q_1) identify the most frequent routes during the last 30 minutes, and (Q_2) identify areas that are currently most profitable for taxi drivers.

Although implementing the problem from the scratch is not too complicated, there are many event data models and technologies that could serve as the basis for a solution: Complex Event Processing (CEP), Data Stream Processing, Rule-based systems, etc. In previous DEBS challenges, many of those models have been adopted to solve similar challenges. e.g. Rabl et al. [21] and Jerzak et al. [11] used custom solutions, Geesen and Grawunder [9], Fernandez et al. [8] used stream processing, Aders et al. [1] used both rules and stream processing, Koliousis and Sventek [13] introduced a new automata in a custom solution, and Perera et al. [17] used CEP.

To embrace the adoption of Semantic Web technology into event and stream processing, the RDF Stream Processing (RSP) Community Group[1] has been an emerging community which aims at defining a common model for producing, transmitting, and continuously querying RDF Streams. Members in the group have developed RSP engines to support continuous query processing using RDF data model such as C-SPARQL [4], EP-SPARQL [2], SPARQL$_{\text{Stream}}$ [6], and CQELS [19]. RDF-based event processing engines enable the interoperability for the environment which requires the integration of heterogeneous and distributed event sources and background databases. For example, by using an RSP engine, provided taxi data can be easily integrated with live and historical streams (live traffic speed data, bicycle parking, map of subway/ferry) of transport provided by New York city via NYC Open Data.[2] On top of that, if such data is represented in RDF, the open linked data sets such

DEBS'15, June 29 - July 3, 2015, Oslo, Norway
© ACM ISBN 978-1-4503-3286-6/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2675743.2772586.

[1]https://www.w3.org/community/rsp/
[2]https://nycopendata.socrata.com/

as Wikipedia, Open street map come as an "effort-less integration offer" for application builder.

Theoretically, queries in the Challenge can be modeled and processed by RSP engines; practically, gaining real-time performance under high throughput data is not trivial. The reason is that the RDF data presentation is verbose which introduces a lot overhead in processing it [19]. Moreover, the schema-less property of RDF, on the one hand, enables the flexibility of schema change; one the other hand, often (but not always) implies hard work in enabling strict low response and high throughput capability for an RSP processing. In particular, the hard-to-predict structure of RDF graphs has proved challenging for traditional DBMSs, and they have not been able to scale effectively to large quantities of RDF data [16]. This unpredictability also applies to RDF-based data streams, as a consequence, makes it difficult for query optimizers to handle.

Coping with these challenges, CQELS has been developed based on a native and adaptive approach, that is, to treat RDF as first class citizens and to dynamically switch between query plans at run time to adapt to the fluctuate and bursty nature of input streams. Furthermore, it employs sophisticated stream processing techniques to boost for performance, including, but not limited to, efficient mechanisms for indexing the input buffers, namely one-way index and ring index, and incremental computing algorithms on continuous query operators. CQELS has been proved to be the fastest engine in the RSP community [20], and is seeking its position in comparison to stream processing engines in other communities. The DEBS Grand Challenge is a great opportunity for CQELS to look outside its comfort zone.

This paper presents our solution for the Challenge using CQELS. It is organized as follows: Section 2 gives preliminaries on RDF, SPARQL, RSP, and provides the encoding of Q_1 and Q_2 using the CQELS query language. Section 3 describes the novel aspects in the design and implementation of CQELS engine that contribute to the efficiency of our solution. Instructions on how to deploy our solution and evaluation results are shown in Section 4. Finally, Section 5 concludes the paper.

2. RDF STREAM PROCESSING

RDF Stream Processing can be intuitively seen as extending querying RDF datasets[3] with SPARQL to querying RDF streams with "continuous SPARQL." We briefly review RDF, SPARQL, and the above extension.

RDF stands for Resource Description Framework, a standardized model model for representing information in the web. Let I, B, and L be *RDF nodes* which are pair-wise disjoint infinite sets of Information Resource Identifiers (IRIs), blank nodes and literals respectively, and V is a set of variables. A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is an *RDF triple* where s is the subject, p the predicate, and o the object. An RDF graph (also referred as an RDF dataset, or simply dataset) is a set of RDF triples. A *triple pattern* is a triple $(sp, pp, op) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$. A *basic graph pattern* is a set of triple patterns.

Example 1 Take an input line of the Challenge's scenario:

```
07290D3599E7A0D62097A346EFCC1FB5,
E7750A37CAB07D0DFF0AF7E3573AC141,
2013-01-01 00:00:00, 2013-01-01 00:02:00,
120,0.44,-73.956528,40.716976,-73.962440,
40.715008,CSH,3.50,0.50,0.50,0.00,0.00,4.50
```

[3]`http://www.w3.org/RDF/`

Its data used for Q_1 and Q_2 can be represented as the following RDF graph:

$$g = \left\{ \begin{array}{l} :trip_1 \; :taxi \; \texttt{"07290D3599E7A0D62097A346EFCC1FB5"}. \\ :trip_1 \; :pickup\_datetime \; \texttt{"2013-01-01 00:00:00"}. \\ :trip_1 \; :dropoff\_datetime \; \texttt{"2013-01-01 00:02:00"}. \\ :trip_1 \; :pickLon \; \texttt{-73.956528}. \\ :trip_1 \; :pickLat \; \texttt{40.716976}. \\ :trip_1 \; :dropLon \; \texttt{-73.962440}. \\ :trip_1 \; :dropLat \; \texttt{40.715008}. \\ :trip_1 \; :fare \; \texttt{3.5}. \\ :trip_1 \; :tip \; \texttt{0.0}. \end{array} \right\}.$$

SPARQL is essentially a graph-matching query language for querying RDF graphs. A SPARQL query is of the form $H \leftarrow B$, where B, the body of the query, is a complex RDF graph pattern composed by combining basic graph patterns with different algebraic operators such as `UNION`, `OPTIONAL`, and `FILTER`; and H, the head of the query, is an expression that indicates how to construct the answer to the query [18].

Example 2 Assume that information regarding a snapshot of taxi trips in the last 30 minutes is collected in an RDF store identified by the IRI `<http://example/taxi.rdf>`, under the format shown in Example 1. The following SPARQL query computes the answer of Q_1 at a single time point.

```
 1 SELECT (ROUND((41.474937-?pLat)/0.005986)     AS ?pE)
 2        (ROUND((74.913585+?pLon)/0.004491556) AS ?pS)
 3        (ROUND((41.474937-?dLat)/0.005986)     AS ?dE)
 4        (ROUND((74.913585+?dLon)/0.004491556) AS ?dS)
 5        (COUNT(?trip) AS ?freq)
 6 FROM  <http://example/taxi.rdf>
 7 WHERE {
 8   ?trip :pickLon  ?pLon. ?trip :pickLat  ?pLat.
 9   ?trip :dropLon  ?dLon. ?trip :dropLat  ?dLat.
10 }
11 GROUP BY  ?pE ?pS ?dE ?dS
12 HAVING    (?pE>0 && ?pE<301 && ?pS>0 && ?pS<301 &&
13           ?dE>0 && ?dE<301 && ?dS>0 && ?dS<301)
14 ORDER BY  ?freq
15 LIMIT 10
```

One-shot Q_1 in SPARQL

Lines 1-4 compute the pick up and drop off cells from the coordinates as follows. Let *long* and *lat* be the longitude and latitude of a coordinate, the corresponding cell (i, j) is identified by:

$$i = \frac{41.474937 - lat}{0.005986} \text{ and } j = \frac{74.913585 + long}{0.004491556}.$$

The graph pattern in the `WHERE` clause (lines 8-9) extracts the pick up and drop off coordinates from the RDF graph loaded by the `FROM` clause. The trip frequency is calculated by the `COUNT` operator (line 5), grouped by cells (line 11). The conditions stated in the `HAVING` clause (lines 12-13) makes sure that we only consider trips in the 300×300 grid. Finally, `LIMIT 10` cuts the answers to the top 10, ordered by the trip frequency (line 14).

The semantics of SPARQL is defined via *mappings*. A mapping μ is a partial function from V to $I \cup B \cup L$. The result of a `SELECT` SPARQL query is a set of mappings that match the body of the query. For example, evaluating the

SPARQL query in Example 2 under the RDF graph in Example 1 returns a set of a single mapping:

$$\{\{\texttt{?pE}{\mapsto}127, \texttt{?pS} \mapsto 213, \texttt{?dE} \mapsto 127, \texttt{?dS} \mapsto 212, \texttt{?freq}{\mapsto}1\}\}\,.$$

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

$$\forall x \in dom(\mu_1) \cap dom(\mu_2)\colon \mu_1(x) = \mu_2(x).$$

However, one-shot queries by themselves are not able to answer queries under dynamic input as in the scenario of the Challenge. For this purpose, we need continuous queries over RDF streams and Instantaneous RDF datasets.

RDF Streams and Instantaneous RDF datasets. In continuous query processing over dynamic data, the temporal nature of the data is crucial and needs to be captured in the data representation. This applies to both Linked Stream Data and Linked Data, as updates in Linked Data collections are also possible. We define RDF streams to represent the former, and model the latter by generalizing the standard definition of RDF datasets to include the temporal aspect. Thereby:

1. An *RDF dataset at timestamp t*, denoted by $G(t)$, is a set of RDF triples valid at time t and called *instantaneous RDF dataset*. An *RDF dataset* is a sequence $G = [G(t)], t \in \mathbb{N}$, ordered by t.
2. An *RDF stream* $\mathcal{S}$ is a bag of elements $\langle g : [t] \rangle$, where g is an RDF graph and t is a timestamp.

Example 3 The input stream $\mathcal{S}_{Taxi}$ of the scenario used in the Challenge is a bag of elements $\langle g_i : [t_i] \rangle$, where g_i is of the form in Example 1 and t_i is the drop-off time in g_i.

Continuous Queries. Continuous queries in CQELS are inspired by the Continuous Query Language (CQL) [3], where a continuous query is composed from three classes of operators, namely stream-to-relation (S2R), relation-to-relation (R2R), and relation-to-stream (R2S) operators. S2R and R2R operators are relevant in the context of the Challenge. Intuitively, the former are captured by extending SPARQL 1.1 grammar[4] with a "stream graph" pattern, while the latter are taken care of by SPARQL's operators. The next two examples give the encoding of Q_1 and Q_2 in the CQELS query language. For more details on the query syntax, we refer the reader to [19].

Example 4 Q_1 can be encoded in CQELS as follows.

```
 1 SELECT (ROUND((41.474937-?pLat)/0.005986)     AS ?pE)
 2        (ROUND((74.913585+?pLon)/0.004491556)  AS ?pS)
 3        (ROUND((41.474937-?dLat)/0.005986)     AS ?dE)
 4        (ROUND((74.913585+?dLon)/0.004491556)  AS ?dS)
 5        (COUNT(?trip) AS ?freq)
 6 WHERE {
 7   STREAM <Taxi> [RANGE 30 minutes] {
 8     ?trip :pickLon  ?pLon. ?trip :pickLat  ?pLat.
 9     ?trip :dropLon  ?dLon. ?trip :dropLat  ?dLat. }
10 }
11 GROUP BY  ?pE ?pS ?dE ?dS
12 HAVING    (?pE>0 && ?pE<301 && ?pS>0 && ?pS<301 &&
13            ?dE>0 && ?dE<301 && ?dS>0 && ?dS<301)
14 ORDER BY  ?freq
15 LIMIT 10
```

Q_1 in CQELS

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

Compared to its one-shot counterpart in Example 2, we can see the change of the `FROM` clause (line 6, Example 2) to the `STREAM` graph pattern (line 7), which represents a window extracting the triples arriving at the engine within the last 30 minutes. Other parts of the two queries stay identical.

Example 5 Q_2 can be encoded by the following nested CQELS query.

```
 1 SELECT ?pE ?pS ?noEmptyTaxis
 2        ?prof (?prof/?noEmptyTaxis AS ?profitability)
 3 WHERE {
 4   {SELECT
 5     (ROUND((41.474937-?pLat1)*2/0.005986)    AS ?pE)
 6     (ROUND((74.913585+?pLon1)*2/0.004491556) AS ?pS)
 7     (MEDIAN(?fare+?tip)                     AS ?prof)
 8    WHERE {
 9      STREAM <Taxi> [RANGE 15 minutes] {
10        ?trip :fare    ?fare.
11        ?trip :tip     ?tip.
12        ?trip :pickLat ?pLat1.
13        ?trip :pickLon ?pLon1.
14      }
15    }
16    GROUP BY ?pE ?pS
17    HAVING (?pE>0 && ?pE<601 && ?pE>0 && ?pS<601)
18   }
19   {SELECT
20     (ROUND((41.474937-?pLat2)*2/0.005986)    AS ?pE)
21     (ROUND((74.913585+?pLon2)*2/0.004491556) AS ?pS)
22     (COUNT(?taxi)                   AS ?noEmptyTaxis)
23    WHERE {
24      STREAM <Taxi> [RANGE 30 minutes] {
25        ?trip2 :pickLat ?pLat2.
26        ?trip2 :pickLon ?pLon2.
27        ?trip2 :taxi  ?taxi.
28        ?trip2 :dropoffTime  ?dropoff.
29      }
30      FILTER NOT EXISTS {
31        STREAM <Taxi> [RANGE 30 minutes] {
32          ?trip3 :taxi ?taxi.
33          ?trip3 :pickupTime ?pickup.
34        }
35        FILTER (?pickup>?dropoff)
36      }
37   }
38   GROUP BY ?pE  ?pS
39   HAVING (?pE>0 && ?pE<601 && ?pS>0 && ?pS<601 &&
40           ?noEmptyTaxis>0)
41   }
42 }
43 ORDER BY ?profitability
44 LIMIT 10
```

Q_2 in CQELS

The first subquery (lines 4-18) calculates the profit of the areas. The stream graph pattern from lines 9 to 14 extracts information regarding trips reported within the last 15 minutes, including the trip's pickup coordinates, fare and tip. Then, the coordinates are converted into cell id stored in `?pE` and `?pS` using the same equations as in Example 2 (lines 5, 6). The outputs are grouped by these values, i.e., by cells (line 16). The median function (line 7) on the sum of fares and tips is carried out to calculate the profit on every cell with reported trips in the (600×600) grid (lines 16, 17).

The second subquery (lines 19-41) counts the number of empty taxis. The stream graph pattern from lines 24 to 29 extracts information regarding trips reported within the last half an hour, including the trip's pickup coordinates, the taxi id, and the drop-off time. However, only trips having no next trips reported within the same window are kept. This is accomplished by using `FILTER NOT EXISTS` on the stream graph pattern from lines 31 to 34 together with the

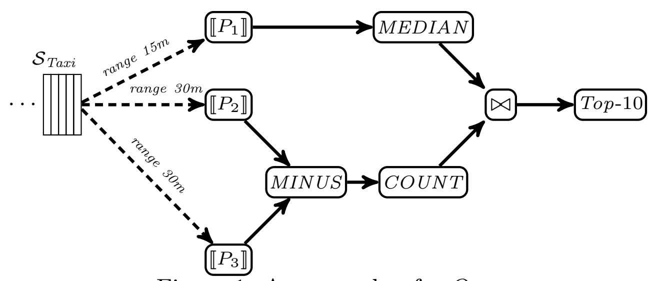


Figure 1: A query plan for Q_2

condition on line 35. The counting is done by lines 22 on cells (line 38) in the (600×600) grid (line 39). Only positive number of empty taxis are taken into account (line 40).

The results of the two subqueries are joined on the cells' ids, and the profitability is calculated on line 2. Furthermore, only top 10 results are reported due to the ordering and limiting on lines 43, 44.

3. CQELS SOLUTIONS FOR THE GRAND CHALLENGE

Our solution is based on CQELS, an *native* and *adaptive* execution framework for Linked Stream Data and Lined Data. The CQELS engines accepts RDF streams and RDF datasets as inputs and returns RDF streams or relational streams in the SPARQL Result format[5] as output. The output RDF streams can be fed into any CQELS engine, and the relational streams can be used by other relational stream processing systems.

Internally, a query is translated into different physical query plans consisting of operators that can process continuous input streams. A query plan composes of operators in a tree-shape where leaf nodes are pattern matching operators and intermediate nodes are relational operators. The former get as input RDF graphs from window buffers and produce bags of mappings that match the pattern. The latter consume a set of bags of mappings and returns a bag of mappings which can be used as input for other operators in the query plan. The root of the query plan might convert the final mappings into RDF graphs before sending the results to the receiver. A query plan of Q_2 in Example 5 is depicted in Figure 1. Here, $[\![P_i]\!]$ denotes the pattern matching operator with the pattern P_i, where P_1, P_2, P_3 are patterns taken from lines 9-13, 25-28, and 32-33 of Q_2, respectively. At run time, based on online statistics, the most efficient on-the-fly query plan is chosen for execution.

To gain efficiency, we pursue an incremental evaluation approach. That is, instead of recomputing on every new incoming input triple, only the difference between the new output and that of the previous step is evaluated. This strategy requires to maintain suitable data structures and algorithms to keep track of the changes in the structures, which introduces a trade-off with computing from scratch. This section highlights the novel aspects in our design and implementation to minimize the trade-off and boost the performance.

3.1 Tree-Based Data Structure

Based on tree-shape query plans, we introduce a tree-based data structure which contains *leaf mappings* and *intermediate mappings*. The former are similar to the row-based data structure in relational tables, and are generated by pattern matching operators. The latter are generated by relational operators during the execution of a query plan. The benefits of this data structure are:

• *Save memory space*: an intermediate mapping does not need to store the binding values but only one or two pointers to reference to the mapping that generated it. Furthermore, we only need to store timestamps of leaf mappings to be able to generate the timestamps of the final results and to detect expiration of intermediate results.

• *Allow for an efficient strategy to handle expiration of mappings*: when a leaf mapping μ expires (which can be triggered by source input or clock ticks), we just need to send a *negative* version of μ to the final operator in the query plan. Based on the pointers linking intermediate/final mappings and the ones used to generated them, we can travel backwards and recognize final/intermediate mappings that were created due to μ, and expire them.

3.2 Indexed Buffers

CQELS' operators store their input mappings in input buffers and continuously carry out operations such as probing, inserting, or evicting on them. Therefore, data structures and physical storages for input buffers have a significant impact on the performance of these operators.

3.2.1 Bags of mappings implementation

A bag of mappings is stored as a *list* of pointers referencing the mappings stored in the tree-based data structure. The list could be implemented as a linked list or an expandable array, depending on how predictable its size can be [14].

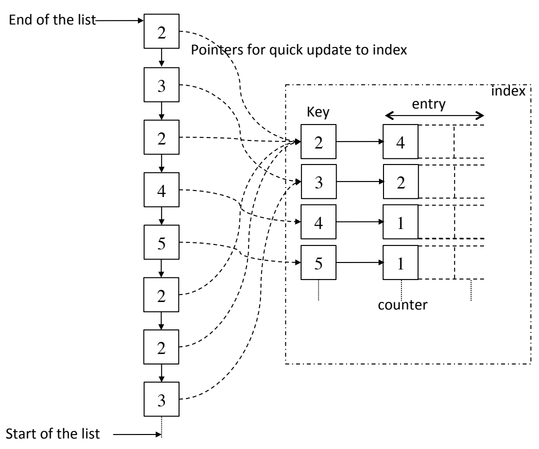
When the maximum size of the list is known, e.g., a count-based window with a fixed number of items, a circular array implementation is useful. When the maximum size is unknown, this option faces the overhead of reallocating data items when the size of the list exceeds the size of the array. In this case, a linked-list implementation is an alternative. In practice, if the size of the list does not change dramatically, the circular array-based implementation is faster than the one based on linked lists. In particular, for lists used in window buffers, they 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.

3.2.2 Indexing on bags of mappings

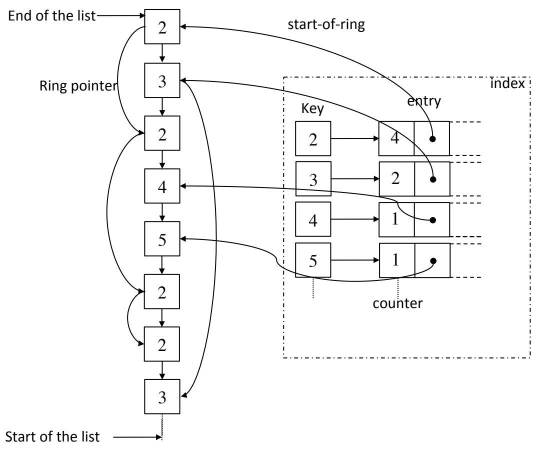
Inspired by [5, 10, 7], we propose indices on keys constructed from a subset of value bindings of mappings stored in a list. The index provides an interface to look up a key for a lookup condition over a set of value bindings. Each key is associated with an *entry* that stores the number of items (called *counter*) in the list with the corresponding key. The key and entry pair are stored in data structures such as B-trees or hash tables. 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.

One-way Index. Figure 2a illustrates the idea of one-way index. This index can 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 aggrega-

[5]`http://www.w3.org/TR/rdf-sparql-XMLres/`



(a) One-way index



(b) Ring Index

Figure 2: Indices for bags of mappings.

tion. 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 is counter. Such lookups can be avoided by adding a pointer to every mapping in the list for each index. The mappings would have pointers to point back to their index entries. When a mapping is deleted, we just need to follow the pointer to the entry to decrease its counter. 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. 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 is dynamically switched at runtime based on the size of the list and the number of keys.

Ring Index. For operations that need to retrieve the list of items that match a certain key, we extend one-way index to ring index [5, 10] as depicted in Figure 2b. 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 (newest) mapping of the ring. When inserting a new mapping to the end of the list with a ring index, if the key of the new mapping is already in the index, the ring pointer is reassigned to the new first mapping of the ring, and the counter of the corresponding ring index entry is increased. Otherwise, a new ring index entry is created. The start-of-ring pointer then points to the inserted mapping.

To retrieve a list of mappings that have a particular key, we first get the ring index entry corresponding to the key. If it is not null, an iterator is created for iterating over all the mappings that have the key by following the ring pointers. Note that the counter in the index entry is used as the stop condition for the iterator.

When the frequency count of a key in an index reaches zero, this means the key is not being referred to by any mapping anymore. However, immediate deleting of such keys in the index might not be efficient. For instance, if a self-balancing tree is used for the index, a lazy-deletion approach could avoid a number of re-balancing operations [10]. 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. A list of keys with zero frequency count is maintained until its size reaches a certain threshold; then, 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.

We can use balanced trees or hash tables for indexing keys for both one-way and ring indices. Hash tables can only be used for the equality predicates while balanced trees also support range scans.

3.3 Incremental Algorithms

The above indexing techniques allow efficient implementation of incremental algorithms for CQELS' operators. This section gives an overview on the algorithms of four operators that are heavily used in evaluating the queries of the Challenge, namely Aggregation, Minus, Join, and Top-K. In stead of going into technical details [14], we illustrate the algorithms on the queries of the Challenge.

Aggregation. An aggregate operator is of the form

$$G_1, \ldots, G_m \mathcal{AGG}_{f_1(X_1),\ldots,f_k(X_k)},$$

where $G_1, \ldots, G_m$ are variables, $X_1, \ldots X_k$ are vector of variables, and $f_1, \ldots f_k$ are aggregate functions. For example, lines 7 and 12 in Q_2 form the operator

$$\texttt{?pE}, \texttt{?pS}\mathcal{AGG}_{MEDIAN(\texttt{?fare+?tip})}$$

which computes the profit of trips that originates from cell (`?pE`, `?pS`). Given a set of mappings, this operator divides the mappings into groups of mappings having common values on (`?pE`, `?pS`). Each group is associated with an output of the form $\langle g, m\rangle$, where g is the group identifier, computed as the composite key of the value bindings on `?pE`, `?pS`, and m is the median of the sum of `?fare` and `?tip` provided by the mappings belonging to g. This value is updated whenever the set of mappings identified by g changes in the case of new or expired mappings.

Our aggregate algorithm maintains a one-way indexed buffer $\mathcal{EXP}$ to store the input mappings that contribute to the aggregate value. Here, the group identifiers g are the keys of $\mathcal{EXP}$. When a mapping

$$\mu = \{\texttt{?pE} \mapsto pe, \texttt{?pS} \mapsto ps, \texttt{?fare} \mapsto \mathit{fare}, \texttt{?tip} \mapsto \mathit{tip}\}$$

is processed, pe and ps are used to identify the group g that μ belongs to. We then find in $\mathcal{EXP}$ an index entry $\mathcal{E}$ with g and create such an entry if g has not yet existed (in case of new mappings). Then, *fare* and *tip* are used to update the median value. Depending on whether μ is a new or expired mapping, $\mathcal{E}$ is updated accordingly.

Minus. A minus operator $(R_1 \setminus R_2)$ between two input buffers R_1 and R_2 produces mappings in R_1 which are not compatible with any mapping in R_2. This operator is asymmetric because handling new or expired mappings depends on whether the mapping is from R_1 or R_2. Its incremental evaluation is divided into four cases, corresponding to four incremental Equations (1)-(4). Here, $\mu \cong R$ iff there exists some $\mu' \in R$ such that $\mu \cong \mu'$, where the notion of compatibility between mappings was introduced in Section 2.

$$(R_1 \uplus \mu) \setminus R_2 = \begin{cases} (R_1 - R_2) \cup \mu & \text{if } \mu \cong R_2 \\ (R_1 - R_2) & \text{otherwise} \end{cases} \quad (1)$$

$$(R_1 - \mu) \setminus R_2 = \begin{cases} (R_1 \setminus R_2) \cup \mu & \text{if } \mu \cong R_2 \\ (R_1 \setminus R_2) & \text{otherwise} \end{cases} \quad (2)$$

$$R_1 \setminus (R_2 \uplus \mu) = (R_1 \setminus R_2) \setminus \{\mu' \mid \mu' \in R_1 \wedge \mu \cong \mu'\} \quad (3)$$

$$R_1 \setminus (R_2 - \mu) = (R_1 \setminus R_2) \cup \{\mu' \mid \mu' \in R_1 \wedge \mu \cong \mu'\} \quad (4)$$

Consider the subquery from line 24 to line 36 of Q_2 (Example 5) which finds empty taxis. The `FILTER NOT EXISTS` keyword issues a `MINUS` operator on two input buffers R_1 and R_2 of the following forms:

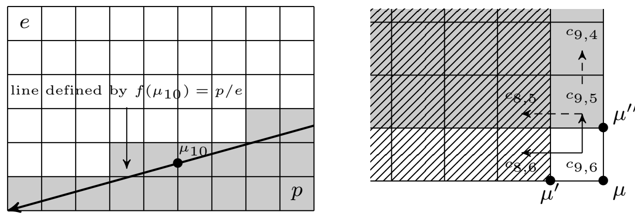
$$R_1 = \left\{ \begin{array}{ll} \texttt{?trip2} \mapsto \mathit{trip}_2, & \texttt{?taxi} \mapsto \mathit{tx}, \\ \texttt{?pLat2} \mapsto \mathit{plat}_2, & \texttt{?pLon2} \mapsto \mathit{plon}_2, \\ \texttt{?dropoff} \mapsto \mathit{dropoff} & \end{array} \right\},$$

$$R_2 = \{\texttt{?trip3} \mapsto \mathit{trip}_3, \texttt{?taxi} \mapsto \mathit{tx}, \texttt{?pickup} \mapsto \mathit{pu}\}.$$

Our implementation of the minus operator is tree-based. For a new mapping μ, we first check if it comes from the left (R_1) or the right side (R_2). Depending on the case, the mapping is then evaluated using either Equation (1) or (3). Similarly, expired mappings are handled case-wise by either Equation (2) or (4). This algorithm uses one-way indices for the input and output buffers. The indexing keys are generated as composite keys from shared binding values between the two input buffers, which are values of `?taxi` in this case. These keys are used to check the existence of compatible mappings from each input buffer.

Join. The join operator works on two input buffers and exploits ring indices to retrieve compatible mappings. Consider Q_2 from Example 5 and the join between the results of the two subqueries: the first one from line 4 to 18 and the second one from line 19 to 41. The two input buffers for this join operator are of the form:

$$R_1 = \{\{\texttt{?pE} \mapsto pe_1, \texttt{?pS} \mapsto ps_1, \texttt{?prof} \mapsto p_1\}, \ldots\}$$
$$R_2 = \{\{\texttt{?pE} \mapsto pe_2, \texttt{?pS} \mapsto ps_2, \texttt{?noEmptyTaxis} \mapsto e_2\}, \ldots\}$$



(a) Index grid (b) Cell visiting order

Figure 3: A top-k computation example

Our join algorithm maintains two ring-indices $\mathcal{B}_1$ and $\mathcal{B}_2$ for R_1 and R_2, respectively. When R_1 gets a new mapping $\mu_1 = \{\texttt{?pE} \mapsto pe, \texttt{?pS} \mapsto ps, \texttt{?prof} \mapsto p\}$, the two values pe and ps will be used to compute a key to probe all mappings in R_2 which are compatible with μ_1, that is, mappings of the form $\mu_2 = \{\texttt{?pE} \mapsto pe, \texttt{?pS} \mapsto ps, \texttt{?noEmptyTaxi} \mapsto e\}\}$. From these inputs, results of the following form are created: μ=$\{\texttt{?pE} \mapsto pe, \texttt{?pS} \mapsto ps, \texttt{?prof} \mapsto p, \texttt{?noEmptyTaxi} \mapsto e\}\}$.

Top-K. A top-k operator gets as input a buffer of mappings. As new mappings are fed into the buffer or old mappings get expired, the operator continuously reports a set of k mappings that have highest scores according to a preference function f. The top-10 operator in Q_2 consumes the output of the Join operator (Fig. 1) which are mappings of the form

$$\mu = \{\texttt{?pE} \mapsto pe, \texttt{?pS} \mapsto ps, \texttt{?prof} \mapsto p, \texttt{?noEmptyTaxi} \mapsto e\}$$

and computes 10 mappings with highest scores computed by the preference function

$$f(\texttt{?prof}, \texttt{?noEmptyTaxi}) = \texttt{?prof}/\texttt{?noEmptyTaxi} = p/e.$$

The continuous top-k operator in CQELS is implemented according to the Top-k Monitoring Algorithm (TMA) [15]. Follow this approach, for Q_2, we use a regular grid to index the valid mappings. The extent of each cell on each dimension is δ so that cell $c_{i,j}$ contains all mappings with $p \in [i \cdot \delta, (i+1) \cdot \delta)$ and $e \in [j \cdot \delta, (j+1) \cdot \delta)$. Conversely, given a mapping μ of the above form, its covering cell can be determined as $c_{i,j}$ where i=$\lfloor p/\delta \rfloor$ and j=$\lfloor e/\delta \rfloor$. A cell stores a linked list of pointers to its mappings to support fast inserting and evicting of mappings in a first-in-first-out manner.

Let μ_i=$\{\texttt{?pE}{\mapsto}pe_i, \texttt{?pS}{\mapsto}ps_i, \texttt{?prof}{\mapsto}p_i, \texttt{?noEmptyTaxi}{\mapsto}e_i\}$ be the mapping with i-th highest score, denoted by $\mathit{score}(\mu_i)$, on function f. The *influence region* of Q_2 is the set of cells containing some values p, e such that $p/e > p_{10}/e_{10}$. In Figure 3, the line defined by $p/e = f(\mu_{10})$ divides the grid into two parts, the lower part contains points (p', e') having $p'/e' > f(\mu_{10})$ while the upper part contains points (p'', e'') having $p''/e'' < f(\mu_{10})$. Note that $(0, 0)$ is excluded from this line. The influence region contains the shaded cells.

Observe that f is increasing monotone on p and decreasing monotone on e, meaning that

$$\forall (p', e), (p'', e) \colon p' \geq p'' \Rightarrow f(p', e) \geq f(p'', e)$$
$$\forall (p,' e), (p, e'') \colon e' \geq e'' \Rightarrow f(p, e') \leq f(p, e'').$$

This property guarantees that the bottom right corner of each cell has the highest score on f compared to other points in the cell. We say this point holds the *maxscore* of the cell.

To find the initial top k mappings, we start with an empty *toplist* and a *barscore* initialized to $-\infty$. This value is used to store the score of the score of the k-th mapping in *toplist*.

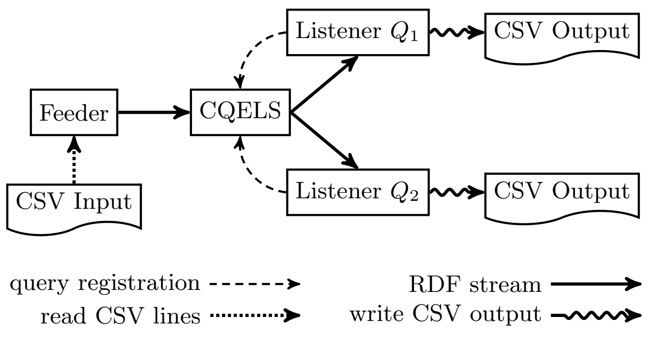


Figure 4: Evaluation Deployment

Then, we travel the grid in a descending order on the cell's *maxscore*, starting from the bottom right cell like $c_{9,6}$ in Figure 3b. To determine the second cell to visit, we just need to compare the *maxscore* of the two neighboring cells $c_{9,5}$ and $c_{8,6}$, that is, $score(\mu')$ and $score(\mu'')$. The former dominates the score of the stripped area while the latter dominates that of the shaded area. Suppose that $score(\mu'') > score(\mu')$, then $c_{9,5}$ is visited next. For the third cell, following the same strategy, we pick the one with highest *maxscore* between $c_{8,6}$, $c_{8,5}$, and $c_{9,4}$.

In each cell, all mappings belong to the cell are examined whether they give score higher than *barscore*. If yes, *toplist* is updated and *barscore* might be updated, depending on the current length of *toplist*. The travel only visits cells with *maxscore* $>$ *barscore*, thus only visits cells in the influence region of the k-th mapping.

To deal with updates of input buffer, the operator check if the new mapping falls into the influence region of the k-th mapping. If yes, *toplist* and *barscore* are updated; otherwise, no computation is needed. Similarly, an expired mapping only triggers recomputation if it belongs to the influence region of the k-th mapping. Besides, the mapping is removed from the cell it belongs to.

4. DEPLOYMENT AND TEST

Figure 4 describes our setup to evaluate Q_1 and Q_2 using CQELS. Each query is registered with CQELS via a Listener. A Feeder reads CSV input lines from the input data file, converts the lines into RDF triples, and feeds them into CQELS. CQELS outputs RDF streams to respective listeners. The Listeners then convert these outputs to the CSV format defined by the Challenge and write them to respective output files. Under this setup, the delay and execution times in our solution are as follows:

- The *delay time* for each output with respect to a query is calculated as the duration between right after reading the input (by Feeder) that triggers the output and the time right before writing the output to the respective output file (by the respective Listener).

- The *execution time* is calculated as duration between the time when the first input line is read by Feeder until the time when the final output is streamed out. This means the time used to write output to files is also included.

To guarantee correctness, we also implemented a base-line system which simply recomputes the queries every time a new input arrives. We have checked that CQELS and this system agree on the outputs. Furthermore, we compare their performance to see the effect of the optimization techniques implemented in CQELS.

	Avg. delay time (ms)				Exe. time (s)		
	Indv.		Mixed		Indv.		Mixed
	Q_1	Q_2	Q_1	Q_2	Q_1	Q_2	Q_1 & Q_2
C	0.022	0.068	0.023	0.075	65.77	69.02	76.64
B	5	194	174	220	5000	22000	28000

Table 1: Compare CQELS and the base-line system

Our evaluation was done on a host system using dual-core Intel(R) Xeon(R) CPU 2.50GHz processor with 8GB, running Ubuntu Linux 3.8.0-29-generic.

Table 1 reports the average delay time and execution time when running Q_1 and Q_2 individually (Indv. mode) and simultaneously (Mixed mode), on CQELS (denoted as C) the base-line system (denoted as B). The test was conducted on the data file comprising the first 20 days of data. Based on execution time, we observe that with respect to Q_1, CQELS is about 76 times faster than simple recomputation, while for the more complicated Q_2 (resp. the mixed case), the improvement factor is about 320 (resp., 360) times.

The queries of the Grand Challenge are designed on windows of 30 minutes on the input streams, and a window of 15 minutes for computing the profit (Q_2). In our experiment, we tried another step by varying the window sizes to 60, 120, and 240 minutes to see how CQELS handles heavy loads of input buffers.

We were a little surprised that the delay time stays almost identical when the window size varies. This can be explained as follows: with real taxi trips data, the different window sizes of 30, 60, 120, or 240 minutes do not introduce drastic change in the set of trips grouped by pickup (and drop-off) cells. Therefore, a lookup in the indexed buffer does not take longer with bigger window size. Thus, it makes almost no difference in processing one new incoming input event with such varying window sizes. In other words, the window size of 30 minutes is ideal to analyze the scenario on real-life data on taxi trips.

Our solution installed in a VirtualBox image[6] is available at

`http://graphofthings.org/debs2015/cqels.zip`.

5. CONCLUSIONS

This paper describes our solution for the DEBS 2015 Grand Challenge using CQELS, the leading engine in the RDF Stream Processing community. We presented novel techniques implemented in CQELS that are relevant for gaining performance in evaluating the queries of the Challenge. Our experimental results confirm the effect of the techniques by a comparison to a base-line system built with recomputation approach. Furthermore, our experiment on varying the window sizes shows an interesting property of real-life taxi trip data, which suggests that the window size of 30 minutes is perfect to analyze the scenario.

[6]The instruction how to run the evaluation is given in the README file at current folder when the image starts.

Acknowledgement

This work has emanated from research supported in part by research grants from Irish Research Council under Grants No. GOIPD/2013/104 and No. GOIPG/2014/917, European Commission under Grant No. FP7-ICT-608662 (VITAL), and by the Austrian Science Fund (FWF) project P26471.

6. REFERENCES

[1] L. Aders, R. Buffat, Z. Chothia, M. Wetter, C. Balkesen, P. M. Fischer, and N. Tatbu. DEBS'11 Grand Challenge: Streams, Rules, or a Custom Solution? Technical report, ETH, Department of Computer Science, 2011.

[2] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, pages 635–644, 2011.

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

[4] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: a continuous query language for rdf data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.

[5] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *VLDB*, pages 11–20, 2000.

[6] J.-P. Calbimonte, Ó. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC (1)*, pages 96–111, 2010.

[7] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107, 2004.

[8] R. C. Fernandez, M. Weidlich, P. Pietzuch, and A. Gal. Scalable stateful stream processing for smart grids. In *DEBS*, pages 276–281, 2014.

[9] D. Geesen and M. Grawunder. Odysseus as platform to solve grand challenges: DEBS grand challenge. In *DEBS*, pages 359–364, 2012.

[10] L. Golab, S. Garg, and M. T. Özsu. On indexing sliding windows over online data streams. In *EDBT*, pages 712–729, 2004.

[11] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The DEBS 2012 grand challenge. In *DEBS*, pages 393–398, 2012.

[12] Z. Jerzak and H. Ziekow. The DEBS 2015 Grand Challenge. In *DEBS*, June 2015.

[13] A. Koliousis and J. S. Sventek. Glasgow automata illustrated: DEBS grand challenge. In *DEBS*, pages 353–358, 2014.

[14] D. Le-Phuoc. *A Native and Adaptive Approach for Linked Stream Data Processing*. PhD thesis, Digital Enterprise Research Institute, National University of Ireland, Galway, 2013.

[15] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[16] A. Owens. *An Investigation Into Improving RDF Store Performance*. PhD thesis, University of Southampton, April 2011.

[17] S. Perera, S. Suhothayan, M. Vivekanandalingam, P. Fremantle, and S. Weerawarana. Solving the grand challenge using an opensource CEP engine. In *DEBS*, pages 288–293, 2014.

[18] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34:16:1–16:45, September 2009.

[19] D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC (1)*, pages 370–388, 2011.

[20] D. L. Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *ISWC - ET*, pages 300–312, 2012.

[21] T. Rabl, K. Zhang, M. Sadoghi, N. K. Pandey, A. Nigam, C. Wang, and H. Jacobsen. Solving manufacturing equipment monitoring through efficient complex event processing: DEBS grand challenge. In *DEBS*, pages 335–340, 2012.