A Datalog based Query Language for Supporting Recursive Query Processing over Data Streams

Guorui Xiao
University of California, Los Angeles
grxiao@ucla.edu

Jin Wang
Megagon Labs
jin@megagon.ai

Jiacheng Wu
University of Washington
jcwu22@cs.washington.edu

Carlo Zaniolo
University of California, Los Angeles
zaniolo@cs.ucla.edu

***Abstract*—While stream data processing has been a topic of much interest in the database community, efficient support for recursive queries from important real-world scenarios with high-level query languages remains a challenging problem for current data stream systems. In this paper, we argue that Datalog is a good solution to this problem and propose *Streamlog*, the first language for expressing recursive query over data streams in fully declarative ways. *Streamlog* has a syntax that is very similar with Datalog and is able to express concisely a large body of analytical queries by supporting aggregates in recursion. To efficiently evaluate the *Streamlog* programs, we extend the idea of semi-naive evaluation to the stream setting and design a light-weighted index to facilitate the incremental query evaluation across timestamps. We also describe the query planning techniques and verify the effectiveness of our designs by implementing a prototype system. Experimental results on several real world datasets illustrate the efficiency of our proposed techniques.**

I. Introduction

Query processing over data streams has been a popular topic in the database community, where continuous query languages with formal semantics were proposed as extensions of SQL or other query languages to streamline and speedup the development of data stream applications [1], [2], [3], [4]. However, while many popular distributed stream processing platforms, such as Spark Streaming [5], Apache Flink [6] and Kafka [7], can supports SQL-like query interfaces, they cannot directly support expressing recursive queries, which are important in many real world applications. For instance, in the recent "Smart City" world-wide initiatives, typical applications must find the minimum cost to travel from a place A to another place B considering different modes of transport, whose expenses can vary over time (e.g. flight prices, on-demand cab fares, etc). To capture the changes of expenses over time, the traffic network should be modeled as a streaming graph where edges with weights streamed in over time. This example of a shortest path query on streaming graph illustrates the need for languages supporting recursive queries with aggregates over data streams, which cannot be expressed by existing continuous query languages. Common application scenarios for recursive queries over data streams include online social network analysis, e-commerce, reasoning on data streams and several others.

Many recent efforts have focused on supporting complex queries over streaming graphs, which can be naturally expressed as recursive queries. However, current approaches either provide specialized algorithms written in procedure programming languages [8], [9] or support query interfaces with expressions using algebra [10], [11]. As a result, state-of-the-art solutions tend to be limited to a particular platform or a specific programming language and thus suffer with poor portability and other drawbacks. Indeed, users also need to gain a good understanding of both algorithm and the way to manage low-level features of the platforms on. Consequently, the usability becomes a concern.

In this paper, we claim that a simple and efficient solution to the problem of expressing recursive queries over data streams can be based on Datalog. Recently, a renaissance of interest has focused on Datalog because of its succinct and declarative expression of a wide spectrum of data-intensive applications, including machine learning [12], data mining [13], knowledge reasoning [14], declarative networking [15] and social networks [16] etc. The theoretical advances [17], [18] allow us to provide formal declarative semantics to the powerful recursive queries that use aggregates in recursion. Such outcomes outline the promising blueprints of expressing recursive queries on data stream with Datalog.

Following this route, we propose the *Streamlog* language that extends the standard Datalog to stream settings. *Streamlog* can express a broad spectrum of recursive queries, including both streaming graph algorithms and more general applications, such as those supporting stream reasoning [19] and data mining [20], in a fully declarative fashion. In fact, *Streamlog* provides rich expressive power by allowing aggregates in recursion with formal semantics. Furthermore, this paper also illustrates that all essential operators in *Streamlog* can be implemented in a non-blocking way, and discusses how this helps with the implementation of the query engine.

One hindrance in the evaluation of *Streamlog* is that under the sliding window semantics in stream settings, many tuples might be expired once they get time-out. In this case, the evaluation of classic Datalog programs needs to start from scratch on each timestamp when expiration happens. As a result, the overall performance will suffer greatly from the repeated computations across different timestamps. To address this issue, we borrow the idea of incremental computation from the semi-naive evaluation and apply it to the stream settings. Specifically, we allow the incremental computation between different snapshots to start from intermediate results in previous iterations rather than the very beginning. To this end, we propose the Queue based Index (QBI), a light-weigh index structure to support fast lookups and deletions. We then provide a theoretical analysis on the proposed evaluation algorithm and show that it is strictly no worse than the simple method that needs to handle expiration by starting over every time. In order to enable the system implementation of above evaluation method, we also devised effective query planning techniques which we demonstrated and refined in a prototype system. To the best of our knowledge, *Streamlog* is the first high-level language that can express advanced recursive queries over data streams in a fully declarative way.

To sum up, we make the following contributions:

- We propose *Streamlog*, a powerful high-level query language for expressing recursive queries over data stream.
- We define the syntax and semantics of *Streamlog*, and show that it can express a broad range of applications from real scenarios.
- We extend the idea of semi-naive evaluation for Datalog to the stream setting so as to define an efficient algorithm to evaluation *Streamlog* programs. Based on that, we further develop query planning techniques which can be generalized to different platforms.
- We implement a prototype to demonstrate the power of the proposed language and the effectiveness of the techniques used in its implementation. Experimental results on real world datasets demonstrate the superiority of the proposed query evaluation techniques.

The rest of this paper is organized as following: Section II introduces the necessary background knowledge. Section III defines the syntax and semantics of *Streamlog* as well as provides some examples of applications. Section IV proposes an efficient query evaluation method for *Streamlog* with theoretical guarantees. Section V designs the query processing techniques and implements the language into a prototype system. Section VI discusses some important issues related to the scope ot the paper and opportunities in the future. Section VII reports the results of evaluation. Section VIII surveys the related works. Finally Section IX concludes the whole paper.

II. Preliminary

A. Stream Data

First we introduce some basic terminologies about stream data so as to describe the data model. A data stream *tuple* consists of two parts: the attributes A and the *timestamp* τ. The set A consists of one or several attributes based on the pre-defined schema. And τ is the transaction (valid) timestamp of the tuple assigned by the data source. The notion of streams can now be defined as Definition 1.

Definition 1 (Stream): A Stream S is a constantly growing sequence of streaming tuples $S = \{t_1, t_2, ...\}$ where each tuple t_i is associated with a particular time τ_i.

In this paper, we assume tuples in the stream S are generated by a single source and arrive in the timestamp order. That is, for tuples t_i and t_j, we have $\tau_i \le \tau_j$ iff $i < j$. Following the previous studies in data stream processing, we can get the set of tuples at a given timestamps τ_i. We call it *snapshot* P_{τ_i} at the given timestamp. We will denote it as P_i when there is no ambiguity.

Following the previous studies about stream data processing, we also support the semantics of sliding window. There are two kinds of definition on windows, count based and time based ones [21]. The former decides the window size by the number of tuples while the latter decides it by the time interval between the timestamps of the beginning and end of a window. In this paper, we use the second definition which is described as Definition 2.

Definition 2 (Sliding Window): A sliding window W can be defined by the time interval $[\tau_B, \tau_E]$, where τ_B and τ_E is the beginning and end timestamp for the window, respectively. The slide interval of W is defined as a time-based window that progresses every s time units, where $s \le \tau_E - \tau_B$.

Note that throughout paper, when referring W as a numerical value, we mean its size, i.e. $\tau_E - \tau_B$. The content of a window is a set of tuples t_i s.t. $\tau_i \in [\tau_B, \tau_E]$. Given a window $W = [\tau_B, \tau_E]$ and a tuple $t = \langle \tau, A \rangle$, if $\tau < \tau_B$, we will say that the tuple t is expired. For a query with window semantics, we will exclude the tuples that are expired when constructing the snapshots.

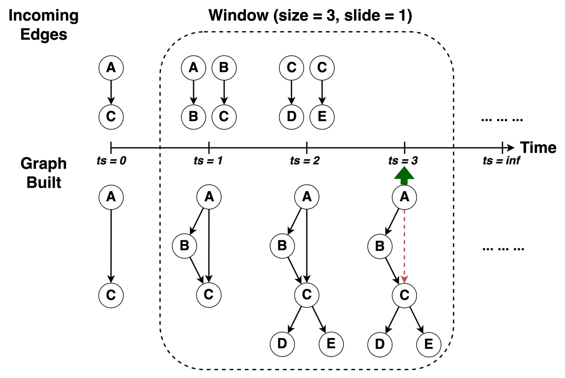

Fig. 1. Example of Streaming Data

Example 1: We present an example of streaming graph in Figure 1. There is a stream $S = \{(0, A, C), (1, A, B), ...\}$ arriving in the timestamp order. At timestamp $ts = 2$, the *Graph Built* below the axis shows a graph built by all the tuples in snapshot P_2. Suppose we have a window W of size 3 and a slide interval $s = 1$. When W slide forward by 1 time unit as $[1, 3]$ and tuples with $ts = 0$ will expire (*e.g.*, $(0, A, C)$ shown in red dotted line) At the same time, no new element arrives when $ts = 3$, thus P_3 equals removing tuple $(0, A, C)$ from P_2. This streaming graph can be queried with our proposed language shown later. Note that Figure 1 only serves as an example while our proposed method could also support recursive queries other than graph queries.

B. Datalog

A Datalog program P consists of a finite set of rules operating on sets of facts which can be evaluated with logical implication. A Datalog rule r is in the format of $h \leftarrow r_1, r_2, ..., r_n$, where h is the head of rule, $r_1, r_2, ..., r_n$ is the body and the comma separating tuples in the body is logical conjunction (AND). The rule head h and each r_i are tuples having form $p(x_1, x_2, ..., x_k)$, where p is a predicate and $x_1, x_2, ..., x_k$ are terms which can be variables or constants. There are two kinds of tables: (i) the base tables are defined by tables in the *EDB* (extensional database) and (ii) the derived tables are defined by the heads of rules and form the *IDB* (intentional database).

Query 1 - Transitive Closure (TC)

```
# Input Stream Schema
{arc(X:Integer, Y: Integer) }

tc(X,Y) ← arc(X,Y).
tc(X,Y) ← tc(X, Z), arc(Z,Y).
query tc(X, Y).
```

An example of recursive Datalog program is shown above in the Transitive Closure program in Query 1. It derives the IDB *tc* from the EDB arc representing the edges of a graph. Since the predicate *tc* is contained in both the head and the body of the first rule, *tc* is a recursive predicate and the second rule is a *recursive rule*. Meanwhile, the first rule is non-recursive and therefore provides the *base rule* in the fixpoint definition.

Algorithm 1: Semi-naive Evaluation of Query 1

```
1  begin
2      $\delta tc$ = $arc(X, Y)$;
3      $tc$ = $\delta tc$;
4      do
5          $\delta tc' = \Pi_{X,Y}(\delta\ tc(X, Z) \bowtie arc(Z, Y)) - tc$;
6          $tc = tc \cup \delta tc'$;
7          $\delta tc$ = $\delta tc'$
8      while $\delta tc \neq \emptyset$ ;
9      return $tc$;
10 end
```

The state-of-the-art method for evaluating a Datalog program is the *semi-naive* (SN) evaluation [22]. SN evaluation performs the differential fixpoint computation of Datalog programs in a bottom-up manner. It starts with the base rule and then iteratively applies recursive delta rules until a *fixpoint* is reached. The core idea of the SN evaluation is that, instead of using the original tables, the evaluation can use delta tables that are based on the facts which were generated in the previous iteration step. We provide the example of evaluating Query 1 with SN evaluation in Algorithm 1.

III. The Streamlog Language

In this section, we propose the *Streamlog* language to express complex recursive queries over stream data. We first introduce the syntax and basic building blocks of the language in Section III-A. We then provide several examples of *Streamlog* to illustrate its potential application scenarios in Section III-B. Finally we discuss the semantics in Section III-C.

A. Syntax

The syntax of *Streamlog* is very similar with that of classic Datalog. The Data Definition Language (DDL) specifies the schema of the table, which includes the table name, the list of attribute names and data types. As standard practice in DSMS [1], there are two kinds of tables: static *relation* and dynamic *stream*.

We add the keyword RELATION or STREAM in front of the table name to specify the table type [1]. For stream table, there is a timestamp associated with each tuple; while for relation table, there is no timestamp. The Data Manipulate Language (DML) describes the *Streamlog* program by a set of rules as Datalog does. It also specifies the table for output via keyword *query* followed by the table name and the set of attribute values. To make it clear, we require the first attribute of a stream table to be the timestamp. Moreover, attributes computed via aggregates occupy the last position in the head of our rules. Furthermore, the sliding window semantics in the query is supported by the keyword WINDOW(x,y), where argument x is the size of the window and y is the size of slide (optional, 1 by default).

Example 2: Query 2 - Employee Count

```
# Input Stream Schema
{dptEmpl(Ts: Timestamp,Depart:String, Employer:String)}

empCnt(Ts, Emp, mcount<(Emp,1)>) ← dptEmpl(Ts, _, Emp).
empCnt(Ts, Dpt, mcount<(Emp,S)>) ← empCnt(Ts1, Emp, S), dptEmpl(Ts2, Dpt, Emp), larger(Ts, Ts1, Ts2).
query empCnt(_, Dpt, C), WINDOW(1 year, 3 month).
```

In Query 2 we have a *Streamlog* program to express the application of counting the number of employees in each department. Here we first write the DML to specify a stream table *dptEmpl* which has two attributes *Department* and *Employer*, both in string format. Then the DDL consists of two rules to recursively compute the number of employees with a *monotonic count* operator. Finally, we use the *query* keyword to specify the table and attributes to output. We also specify a sliding window with length one year and slide size 3 month.

In *Streamlog* queries timestamps must be handled explicitly, and the comparison of timestamp in different tables must explicitly specified in the rules. In this way, it could also support both transaction and valid time [21]. This can be easily done by simple procedures, as illustrated by the following two procedures which will be directly used in the programs for

[1]In this paper we assume a table is a stream by default and thus omit the STREAM keyword in the following definitions

comparing the timestamp: *larger(A, n_1, n_2)* assigns the larger one between n_1 and n_2 to A; while *largest(A, n_1, ..., n_k)* assign the largest value $n_i \in \{n_1, ..., n_k\}$ to A. It is rather easy to express them in Datalog rules. For example, in Query 2, the second rule specifies that the timestamp value of the head should be the larger one between those of *dptEmpl* and *empCnt* tuples with the *larger* procedure.

B. Applications

Next we provide a list of example programs for *Streamlog* and show the corresponding application scenarios. Note that the recursive queries expressed by *Streamlog* can be either stream reasoning ones, where the datasets are small and available offline, or queries over unbounded stream such like those handled by distributed stream platforms. From the aspect of system implementation, they are quite different. But from the aspect of expression and query evaluation, both of them can be handled in a similar way. In this paper, we focus more on the language side and thus do not distinguish them when introducing the language and query evaluation techniques.

Query 3 - Sub-organization

```
# Input Stream Schema
{subOrg(Ts: Timestamp, OrgId: Integer, SubOrgId: Integer)}

isSubOrg(Ts, X, Y) ← subOrg(Ts, X, Y).
isSubOrg(Ts, X, Z) ← subOrg(Ts1, X, Y), isSubOrg(Ts2, Y, Z), largest(Ts, Ts1, Ts2).
query isSubOrg(_,X, Z).
```

Example 3: Query 3 shows the program to find all pairs of organizations (A, B) in which B is a sub-organization of A, which is equivalent to the computation of Transitive Closure. This is a typical application in stream reasoning [23]. The schema of input stream *so* consists of two attributes *OrgId* and *SubOrgId*, which denotes the id of an organization and the organization that is a sub-organization of it, respectively.

Query 4 - CheapPath

```
# Input Stream Schema
{ride(Ts: Timestamp, Pickup: String, Drop: String, Fare: Float)}

bestPrice(Ts, P, D, mmin<F>) ← ride(Ts, P, D, F).
bestPrice(Ts, P, D, mmin<F>) ← ride(Ts1, P, P2, F1), bestPrice(Ts2, P2, D, F2), F = F1 + F2, largest(Ts, Ts1, Ts2).
query bestPrice(_, P, D, C), WINDOW(60 minutes).
```

Example 4: Query 4 shows the program to find the lowest cost in transportation between any two reachable locations in a road network. Here the road network can be modeled as a graph where nodes are locations and edges road between two locations. It is crucial to support such a query on streaming setting because in real scenarios, the cost on each edge could change along with the time. The `min` aggregate is used to select the cheapest path. The output will be the locations for pick-up and drop, as well as the cost between them within a sliding 60-minute time window.

Query 5 - Earliest Organizer

```
# Input Stream Schema
{relateTo(Ts: Timestamp, X:Integer, Y:Integer)}

earliestOrg(Ts, X, mmin<X>) ← relateTo(Ts, X, _).
earliestOrg(Ts, Y, mmin<V>) ← earliestOrg(Ts1, X, V), relateTo(Ts2, X, Y), larger(Ts, Ts1, Ts2).
query earliestOrg(_, Y, T).
```

Example 5: Query 5 is a typical application in social network analysis, which is equivalent to the Connected Component query. The input stream *relateTo* has two attributes, denoting user X is related to Y. If two users are related, then they are regarded as belonging to the same user group. The program aims at finding the earliest user(in ascending order of id) in each group a user belongs to.

Query 6 - Lloyd Clustering

```
# Input Stream Schema
{point(Ts: Timestamp, Pno:String, Dim:Int, Val:Float), init(Ts: Timestamp, Cno:String, Dim:Int, Val:Float)}

center(Ts, J, Cno, Dim, Val) ← init(Ts, Cno, Dim, Val), J = 0.
dist(Ts, J, Pno, Cno, sum<SqDis>) ← point(Ts1, Pno, Dim, Val), center(Ts2, J, Cno, Dim, CVal), SqDis = (Val-Cval) * (Val -CVal), larger(Ts, Ts1, Ts2).
mdist(Ts, J, Pno, min<DCno>) ← dist(Ts, J, Pno, Cno, DSum), encd(DSum,Cno,DCno).
center(Ts, J1, Cno, Dim, avg<Val>) ← point(Ts1, Pno, Dim, Val), mdist(Ts2, J, Pno, DmCno), decd(DmCno, _, Cno), J1 = J + 1, larger(Ts, Ts1, Ts2).
query center(_, 30, Cno, Dim, Val), WINDOW(1 hour).
```

Example 6: Query 6 shows the clustering Lloyd over a set of points in stream data, which is a typical application in data mining. The non-stream version of this program is proposed in [24]. And tables *encd* and *decd* are used to reconstruct the original number. The input stream table consists of data points with K dimensions in vertical representation. The goal is to compute closest center for each point and we want to output the results in the 30-th iteration within a time range of 1 hour.

Query 7 - PaperRank

```
# Input Stream Schema
{pub(Ts: Timestamp, PaperId: String, PaperRef: String)}

ref(Ts, P, count<R>) ← pub(Ts, P, R).
prank(Ts, P, V) ← ref(Ts, P, R), V = 1.0.
prank(Ts, P, sum<V>) ← prank(Ts1, C, W), pub(Ts2, C, P), ref(Ts3, C, R), V = W/R, largest(Ts, Ts1, Ts2, Ts3).
query prank(_, P, S), WINDOW(365 days).
```

Example 7: Query 7 describes the program to find publications with the highest impact from the high-energy physics citation network [25]. The impact of a publication within the research community can be gauged by computing its pagerank as the network grows over time. We apply the sliding window to consider the impact only within one year, which is also common in real scenarios.

C. Semantics

Finally we discuss the semantics of *Streamlog*. As illustrated in previous studies [26], [27], the operators in stream query languages should be non-blocking. Therefore, we will illustrate that the essential operators required in *Streamlog* are non-blocking. The definition of non-blocking operator is stated as: one that produces all the tuples of the output before it has detected the end of the input [28]. To recognize whether an operator is blocking or not, we can deduce the conclusion show in Theorem 1 based on Prepositions 3.5, 3.9, 4.1 from [29]. Following its notion, we use G to denote an operator and $S^j = \{t_1, t_2, \cdots, t_j\}$ to denote the subset stream consisting of first j elements of stream S.

Theorem 1: An operator G on a sequence S is a non-blocking operator, iff G is monotonic with respect to the set containment ordering $\subseteq$, i.e., for $i < j$, $G(S^i) \subseteq G(S^j)$.

With the notion of blocking operators thus defined, we can now revisit the semantics of negation. From the original semantics of Datalog, i.e., Closed-World Assumptions [30], the negation of a predicate can be evaluated only when all the tuples for the predicate are known. However, under this semantics, the implementation of negation would be blocking in *Streamlog*. To resolve this problem, we have the **P**rogressive **C**losing **W**orld **A**ssumption developed in [31] for data streams as shown in Definition 3, where negation is applied considering tuples only seen till now.

Definition 3 (Progressive Closing World Assumption (PCWA)): Consider a timestamped-ordered stream and database facts, Once a fact $t_i = \langle \tau_i, A_i \rangle$ is observed from the stream S, **PCWA** asserts that $\neg t_j$ for which $\tau_j < \tau_i$ and $t_j = \langle \tau_j, A_j \rangle$ is not derived from the stream tuples whose timestamp $\leq \tau_i$.

Next we illustrate that all essential operators in *Streamlog* are non-blocking. Based on the previous studies [32], [33], [34], the required operators in Datalog includes selection, projection, join, aggregation and recursion. It has been recognized in many previous studies that selection, projection and some ways of implementation of join operators are non-blocking. Therefore we only need to look at the recursion and aggregation operators.

The recursion operator is applied to recursive tables and is responsible for controlling the process of evaluating the recursive rules. When an iteration is done, it needs to check whether the fixpoint is reached. If not, it will proceed to the next iteration; otherwise the recursion terminates. We then show in Theorem 2 that the recursion operator is non-blocking.

Theorem 2: The recursion operator G without aggregation (which follows the fixpoint semantics) is a non-blocking operator.

Proof: Given a stream S, where the recursion operator G does not contain aggregates, we want to show G is monotonic w.r.t. the set containment ordering $\subseteq$. Before step i, the operator G already seen the tuples in S^{i-1} and the recursion must reach the fixpoint which output $G^{i-1}(S^{i-1})$. Similarity, after the step i, the operator comes the output $G^i(S^i)$. Thus, we could obtain that $G^{i-1}(S^{i-1}) \subseteq G^i(S^i)$ since $S^{i-1} \subseteq S^i$.

Based on the fixpoint semantics, we could assert that $G^{i-1}(S^{i-1}) = G(S^{i-1})$ and $G^i(S^i) = G(S^i)$ since the fixpoint only depends on the input rather than the calculation orders. Then, we have $G(S^{i-1}) \subseteq G(S^i)$. Thus, $G(S^i) \subseteq G(S^j)$ is established by mathematics induction. Based on the theorem 1, we know the recursion operator is non-blocking.

Next we consider the aggregations. There are many previous studies [35], [36], [17], [37], [24] about how to use aggregates in recursive Datalog rules. Here we assume that all *Streamlog* programs with aggregates in recursion satisfy the PREM property defined in [17].

In fact, extrema aggregates which satisfy the PREM conditions, can be pushed into recursion for certain queries resulting in an efficient and robust computation. Meanwhile, we can see that aggregations that satisfy the PREM property, also satisfy the requirements of continuous aggregates [26], which are non-blocking aggregation operators used in data stream. Thus we can make the conclusion shown in Theorem 3.

Theorem 3: The recursion operator G with aggregation satisfying the PreM properties is also a non-blocking operator.

Proof Sketch: The core idea of proof is to transform the inner aggregation to the monotonic aggregation out of the recursion, then won't influence the non-blocking properties.

IV. SEMI-NAIVE EVALUATION OVER STREAMING DATA

In this section, we discuss how to extend the semi-naive evaluation technique to streaming queries. To this end, we introduce a light-weighted index structure and propose the streaming based semi-naive evaluation algorithm in Section IV-A. Then we make an extensive set of theoretical analysis on the proposed algorithm in Section IV-B.

A. Our Solution

After defining the syntax and semantics of *Streamlog*, the next issue to be resolved is the evaluation of programs. We cannot directly apply the semi-naive (SN) evaluation because the prerequisite to use SN is that the EDB keeps unchanged in the whole process of evaluation, which no longer holds under the stream settings. Besides, when the window semantic is applied, older tuples will be deleted from the window after expiration. In this case, SN evaluation can be applied only within each snapshot. When it comes to a new snapshot with modifications on the EDB, the evaluation needs to start over. Then the straightforward method denoted as Simple needs to compute SN evaluation from scratch every time once expiration happens.

To address this problem, we need to not only apply the idea of SN evaluation to tuples from different iterations within one snapshot, but also among those from different snapshots. To this end, we need to trace the tuples across different snapshots. The basic idea is that given an tuple a in the EDB as well as derived IDBs, we need to keep track of the whole process of deriving it in all snapshots within the window. This is realized by identifying the *derivation path* for each tuple: given an tuple a, the derivation path is a set of tuples that lead to its derivation. For a tuple in EDB, the derivation path only contains itself; for that in IDB, the derivation path starts from an tuple in EDB and includes those in IDB from previous iterations that generate it.

For example, suppose we evaluate the *CheapPath* program for each window W in Figure 1 and each hop will be seen as a Fare of 1. Then within the snapshot P_2, we can see that there are two ways for A to reach C, namely $A \rightarrow C$ and $A \rightarrow B \rightarrow C$. Then the derivation path for $A \rightarrow C$ is the set $\{ride(0, A, C)\}$ while the derivation path for $A \rightarrow B \rightarrow C$ is the set $\{ride(1, A, B)$, $ride(1, B, C)$, $bestPrice(1, A, B, 1)$, $bestPrice(1, B, C, 1)\}$.

With the help of derivation path, we can easily handle the expiration by locating to the starting position of expired tuples in EDB and IDB. However, it brings much extra space overhead to save the whole derivation path for all tuples. We further observe retrieving the starting position of expired tuples only requires keeping the earliest timestamp of a tuple. We call such a timestamp *Derivation Path Timestamp* (DPT) and formally define it in Definition 4.

Definition 4 (DPT): Given a tuple a in the EDB or IDB, the DPT of a is minimum timestamp value of all tuples in its derivation path.

Then for each tuple in EDB, we only need to keep the DPT rather than the whole derivation path. Once a is expired, it is safe to remove all tuples whose timestamp is earlier than or equal to the DPT of a. For example, at snapshot P_2 from Figure 1, the DPT for $A \rightarrow C$ is 0 because derivation path for $A \rightarrow C$ is $\{ride(0, A, C)\}$. On the other hand, the DPT for $A \rightarrow B \rightarrow C$ is 1 because 1 is the minimum timestamp in its derivation path set $\{ride(1, A, B)$, $ride(1, B, C)$, $bestPrice(1, A, B, 1)$, $bestPrice(1, B, C, 1)\}$. This implies that if tuples with timestamp 0 expire for a sliding window, then any derived tuple with DPT $= 0$ should also expire. More specifically, though in Figure 1 at P_2 we have $A \rightarrow C$ and $A \rightarrow B \rightarrow C$ and currently the shortest path between A and B is $bestPrice(A, C, 1)$. When a window slides forward by 1 and at P_3, $A \rightarrow C$ can be safely removed as its DPT equals to 0. Meanwhile, $A \rightarrow B \rightarrow C$ remains because its DPT is equal to 1. As a result, at P_3, we output the result $bestPrice(A, C, 2)$ instead.

Based on above discussion, we come up with a Queue Based Index (QBI) to store the intermediate results based on the DPT. Specifically, we need to build the QBI on all EDBs and IDBs that might be involved in the recursive evaluation. The structure of QBI is rather simple as shown in Figure 2: We insert flags denoting DPTs into the QBI and tuples based on their DPT values. Further more, *within* each set of IDB tuples indexed by the same DPT, we further optimize both the look up time and the space by utilizing the PREM property [17]. With the help of such flags, QBI can facilitate the incremental computations across snapshots when evaluating *Streamlog* queries: During the evaluation process, we can quickly locate to the influenced tuples and start the recursive evaluation from their states in the last snapshot they appear. When insertion and deletion happens to the EDB, we can locate to the tuples needed to be changed in EDB and IDBs based on above flags in constant time and update such relations in linear time.

Algorithm 2: Streamlog Evaluation with QBI

```
1  begin
2      // Suppose a slideing window is applied, repeat for each timestamp
3      Obtain the current timestamp $\tau_{cur}$;
4      if Find expired tuples then
5          Locate to the start point of expired tuples in EDB via QBI;
6          Identify tuples in IDBs associated with above tuples;
7          Remove the expired tuples;
8      Insert new tuples into the EDB and update DPTs;
9      $C \leftarrow$ Set of new tuples at timestamp $\tau_{cur}$;
10     Initialize $C' = \emptyset$;
11     foreach tuple $t \in C$ do
12         $t' \leftarrow$Lookup tuples related to $t$ via QBI;
13         If $t'$ exists, add $t'$ into $C'$;
14         Add $t$ into $C'$;
15     Perform SN evaluation starting from $C'$ and update QBI accordingly;
16 end
```

Algorithm 2 illustrates the process of evaluating *Streamlog* program with the help of QBI when a sliding window is applied. Since the stream is unbounded, we will update the EDB as well as the index for a batch of new tuples with the same timestamp. After obtaining the current timestamp τ_{cur} (line 3), we first check whether there are expired tuples in the window. If so, we need to delete the expired tuples from both EDBs and IDBs. With the help of QBI, we can find the starting point of expired tuples in constant time (line 5 to 7). Otherwise, we only need to insert the new EDBs and update the DPT (line 8). Then we start to perform SN evaluation with the set of newly inserted tuples C (line 9). In order to make use of intermediate results that is already computed in previous snapshots, we search for such tuples in C via QBI (line 12). For a tuple t, if we find a related tuple t' from previous snapshot, we can start from the t' generated in a later iteration rather than from the very beginning (line 13). We also need to add t into C' to avoid missing any potential results (line 14). Next we conduct the SN evaluation based on the results collected from QBI, and during the evaluation, we keep looking up tuples from QBI that are related to newly produced IDBs to add to C'(line 15). To avoid missing any join result, for the *first* join operation in a new snapshot, we need to first join the tuples in EDB having the latest timestamps with the related tuples in IDB. In this process, we also keep updating the QBI with newly produced tuples and their DPTs.

Example 8: Figure 2 shows the process of updating QBI over the streams in Figure 1 with Query 4 *CheapPath* using Algorithm 2. Here we will use the notion of QBI to denote Algorithm 2 if there is no ambiguity. The green tuples denote that they are newly arrived or produced at a particular snapshot. The red tuples denote expired ones w.r.t. the sliding window.

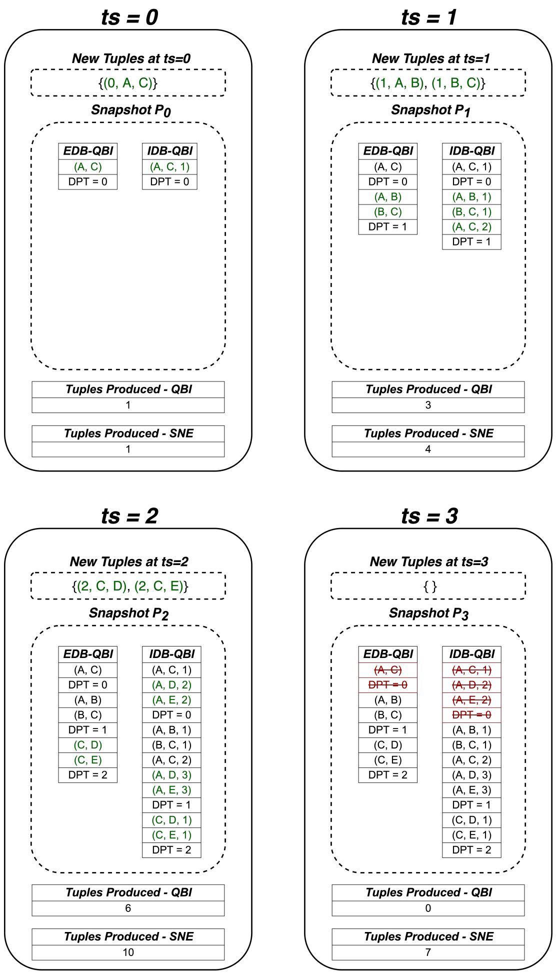

Fig. 2. The Queue-based Index in Streamlog Evaluation

We show the transition from P_1 to P_3 in this example.

At snapshot P_2, we find there is no expired tuple after checking the index. Then the content of C is: $\{(C, D, 1), (C, E, 1)\}$. Now for each tuple in C we try to find their related tuples t' in QBI: based on Query 4, all EDBs are related and IDBs s.t. $(\_, C, \_)$ are related. Thus we fetch these tuples into C'. Note that there are two IDBs s.t. $(A, C, \_)$ and we keep both because they have different DPTs. Finally we perform SN evaluation on top of these existing tuples and derive a new set of tuples $\{(A, D, 2), (A, D, 3), (A, E, 2), (A, E, 3)\}$ and update our QBI accordingly. During this evaluation, we counted total number of tuples produced by both QBI and Simple, which is 6 and 10, respectively.

Next at P_3, all tuples indexed by DPT equals to 0 are expired, and no new tuples arrive. While QBI does not need to do any computation, Simple need to compute and re-produce 7 purely redundant tuples due to the lack of knowledge from the previous snapshots. In the whole process, Simple produces as 2.3 times more tuples than QBI. For real applications with larger window size, the advantage of QBI will definitely be more obvious.

B. Theoretical Analysis

Next we make a theoretical analysis on above proposed algorithm (denoted as QBI) by comparing it against Simple. In order to measure the computational complexity of an algorithm, we look at the number of generated intermediate tuples in the whole process of evaluation. Then we can evaluate the cost of an algorithm following Definition 5.

Definition 5: Given an algorithm A and input N, the cost of this algorithm $A(N)$ is defined or measured as the number of intermediate tuples generated by algorithm A on these N tuples.

Before considering the situation of an unbounded stream, we first consider the behaviors of QBI and Simple in a sliding window with N elements arrived on average when sliding forward. Within a sliding window, N can be seen as the cardinality of EDBs upon which we compute the fixpoint. Then, we will describe the *basic cost* for the two algorithms and use symbols to refer to the costs of them in Definition 6.

Definition 6 (Basic Costs): $F(N)$ denotes the cost by computing the fixpoint from scratch with Simple. Here, computing from scratch means that for all tuples $a \in IDBs$, it is *NOT* fetched from previous fixpoint computation in earlier windows. On the other hand, $C(N)$ denotes the cost by computing the fixpoint from some intermediate results, or partial fixpoint, i.e., there exists at least one tuple $a \in IDBs$ that is directly fetched from previous windows.

Here, we do not need to further describe the detailed composition of the cost and instead, we only need to compare them. Specifically, we can conclude that QBI is always cheaper than Simple in Lemma 1.

Lemma 1: At any given timestamp, $C(N)$ will always be less or equal to $F(N)$.

Given the basic cost defined above for Simple and QBI algorithms, we then could compute the cost for them within a sliding window, respectively. To this end, we first describe the *sequential rule*s in *Streamlog* programs. A rule is said to be sequential when it satisfies the following three conditions: (i) the timestamp of its head is equal to the that of some positive goal, (ii) the timestamp of its head is no smaller than that of the remaining goals, and (iii) its negated goals are strictly sequential or have a timestamp that is smaller than the timestamp of the head. Formally, given a sequential linear *Streamlog* program P with time-based window W and slide s, and we suppose that when the window moves forward N elements arrive. We have the cost of Simple and QBI within such window as Equation (1):

$$\begin{aligned} Simple(W) &= F(N \cdot \frac{W}{s}) && (1) \\ QBI(W) &= C\big(N \cdot (\frac{W}{s} - 1)\big) + F(N) && (2) \end{aligned}$$

Based on such definition of cost, we can borrow the idea of amortized analysis from previous work [38] to compare them by computing their *Max/Max Ratio*. We provide the formal definition of it as Definition 7.

Definition 7 (Max/Max Ratio): Let A be an online algorithm and let $A(S)$ be the cost of A on an input stream S. The amortized cost of A is defined as $M(A) = \limsup_{L\to\infty}(\max_{|S|=L} \frac{A(S)}{|S|})$. Thus, the Max/Max Ratio of A, denoted $wM(A)$ is

$$wM(A) = \frac{M(A)}{M(OPT)} = \limsup_{L\to\infty} \frac{\max_{|S|=L} A(S)}{\max_{|S|=L} OPT(S)} \quad (3)$$

where OPT is an optimal off-line algorithm.

Since both Simple and QBI are designed for unbounded stream data, they can be regarded as *online algorithms*. From above definition, we can see that the Max/Max Ratio has the common denominator $M(OPT)$. Therefore, we could directly compare two online-algorithms A and B using $\frac{M(A)}{M(B)}$, without knowing the optimal off-line algorithm. Specifically, we can draw the following conclusion:

Theorem 4: Given a sequential linear *Streamlog* program P with window size w and slide s, the Max/Max Ratio between QBI and Simple is always ≤ 1, i.e. $\frac{M(QBI)}{M(\mathsf{Simple})} \leq 1$.

Proof Sketch: For each window W, we could simply draw that $\mathsf{QBI}(W) \leq \mathsf{Simple}(W)$ for any input based on Lemma 1 and Equation 1. Then, we apply mathematical induction to prove that $\mathsf{QBI}(S) \leq \mathsf{Simple}(S)$ established for any stream S. Finally, based on the definition, we deduce that $\frac{M(QBI)}{M(SNE)} \leq 1$.

Based on Theorem 4, we can conclude that the performance of QBI is no worse than Simple. The worst case for QBI is the situation when a window slides forward, *all* intermediate results are expired. In such a case, the cost for computing fixpoint of such a window would be $F(N)$ for both QBI and Simple. Thus in such a worst case which is rather rare, QBI is equivalent to Simple. To be more specific, QBI can avoid computation and production of duplicated IDBs which are rather common in the process of recursive evaluation.

V. The Prototype System

In this section, we design and implement a prototype to verify above proposed evaluation techniques for *Streamlog*. We first introduce the overall architecture of the prototype in Section V-A. Then we describe the query planning techniques in detail in Section V-B.

A. Overview

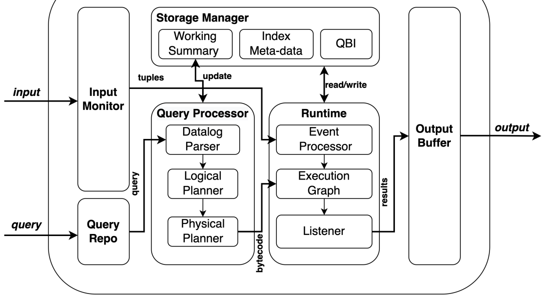

Fig. 3. The Overall Architecture

The overall architecture of the prototype is shown in Figure 3. Here we follows the design paradigm of traditional DSMS where users register queries offline and a plan, represented by an emphexecution graph, is generated. Then during the online stage data entries will stream through the query plan and are evaluated. Finally, the query results are fetched from the output buffer of the query plan. The major components in the prototype are illustrated as following:

The Input Monitor is responsible to handle and arrange the incoming tuples, which will be sent to the execution engine to evaluate the queries later.

The Query Processor takes the registered *Streamlog* query as input. Then it performs query compilation and planning to generate the logical and physical plan accordingly. We will introduce the details later in Section V-B.

The Storage Manager is responsible to allocate the memory by maintaining necessary statistics and meta data. Besides, as the QBI is crucial to support efficient incremental computation, the management of such index structures is also the duty of this module.

The Execution Engine takes the tuples that are streamed in during the online query processing step and guide them through the execution graph generated by the query planner. In this way, the registered queries are executed over the input stream of tuples.

The Output Buffer saves the generated tuples from the execution graph and output them based on the system configurations.

B. Query Planning

Next we provide a more detailed description of the Query Processor component. When a *Streamlog* program is registered to the system, Query Processor is responsible to generate the execution plan for the program. It consists of three steps: compilation, logical planning and physical planning. The logical/physical plan of our prototype is very similar with that in traditional DSMSes [39]. Based on the given *Streamlog* program, the output buffer is assigned to the operator corresponding to the table specified by the *query* keyword. And the input will be the stream table(s) from which the tuples are streaming in. When a tuple enters the execution graph, it will first be put in the buffer. And when the buffer is full, the batch of tuples will be pushed into the associated operator for evaluation, and the results will "flow" into the next buffer in a similar way. This process is repeated until the evaluation terminates and the results are pushed into the output buffer.

1) Compilation: The input processed by the query compiler includes the DDL to specify the database schema and the *Streamlog* program for expressing particular applications. The output of the compilation step will be the logical plan of the input program. One important task for the compiler of *Streamlog* is to resolve the recursions. When a recursive tables is recognized, the compiler switches from the task of building the operator tree for non-recursive queries to the specialized task of handling recursive references. After that, the compiler produces the Predicate Connection Graph (PCG) [40] to identify the dependency of relations within the program which is essential in the generation of logical plans.

2) Logical Plan: The logical plan maps the PCG to a tree containing standard relational operators and recursion operators. Specifically, the *recursion operator* is used in the logical and physical plan to process the recursive query. When a set of tuples reaches the recursion operator, it will decide whether to send them to the output buffer (when the fixpoint is reached) or continue the recursion (otherwise). Following previous studies [33] for Datalog query planning, the plan consists of two parts: (i) The *exit plan* specifies the base case of the recursion which starts the iterations; and (ii) The *recursive plan* defines behaviors within each iteration. In this process, aggregates and group-by columns are automatically identified and some heuristic optimizations are applied.

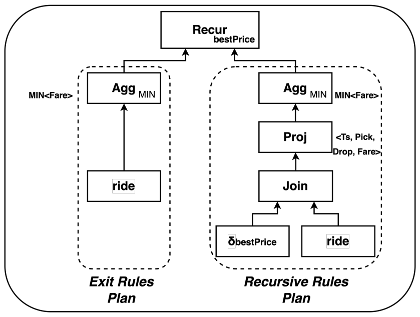

Fig. 4. Logical Plan of CheapPath

Example 9: Figure 4 shows an example logical plan generated for Query 4. The left side of the plan is the *exit plan* that will initialize the *bestPrice* IDBs with EDBs whereas the right side of the plan, namely the *recursive plan*, will join the newly produced *bestPrice* IDBs with *ride* EDBs and perform projection and aggregation. Both aggregation operator will perform MIN over the Fare field of the tuples. On top of both plans is the recursion operator that is responsible for deciding whether a fixpoint has been reached.

3) Physical Plan: The physical plan is generated by analyzing the logical plan with some pre-defined rules. For example, we would like to use the QBI indexes to accelerate the query processing by replacing the full table with the intermediate results obtained from the previous snapshot. Moreover, we can also build index on the recursive tables following previous studies [32], [41] to reduce the time of join processing as well as de-duplication in the process of SN evaluation.

Example 10: Figure 5 shows an example physical plan for Query 4. The green arrow denotes the input source from the outside world where as the blue arrow denotes the output sink for the program. The stream source operator will be responsible for loading input and sending it to the Sliding Window Operator (SW(ride)). Note that this operator is also connected to a QBI via red dotted line to store and update new EDBs.

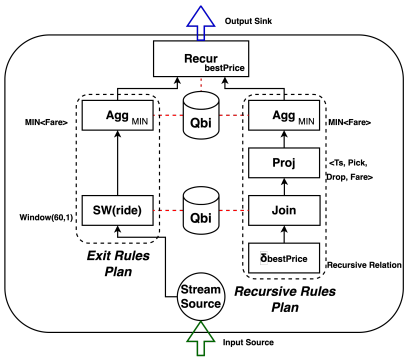

Fig. 5. Physical Plan of CheapPath

VI. Discussion

First of all, we are aware that a fully functioned stream processing system also needs to consider many other issues, such as allowing multiple input streams, supporting operations between static relations and streams, multi-query optimization, handling out-of-order tuples, dynamic query optimization etc. But we would like to highlight that the main contributions of this project consist of a novel high-level query language for expressing powerful recursive queries over data streams provided with formal semantics, as well as feasible query evaluation techniques for our language. The development of the prototype described in Section V allows us to refine and validate techniques that will enable us to develop a very efficient full stream system.

Nevertheless, we also want to claim that our proposed approaches, including the QBI index based SN evaluation and the query planning techniques, can be easily implemented in existing distributed stream processing platforms. Actually many such platforms already provide query engine supporting SQL-like languages. We can implement *Streamlog* in these platforms by leveraging the existing components.

For example, we can utilize the KSQL [3] system on top of Kafka to implement the *Streamlog* query engine in the following way: First we can implement a parser of *Streamlog* either from open-sourced ANTLR [2] or the one from our prototype. Then we could use KStream/KTable in Kafka to represent the streams in *Streamlog*. We can then use KSQL engine's components for planning steps and follow the previous studies of distributed Datalog engines [42], [33], [43] to implement the recursion operator. Finally, we could leverage "topics" in the Kafka streams DSL for the query execution.

Due to the space limitation, we will leave the implementation of *Streamlog* on existing systems as a future work.

Besides, it is very easy to provide SQL-like queries that have equivalent semantics to the *Streamlog* programs. This can be realized by following the idea of RaSQL [44], which supported aggregates in recursion by introducing a simple extension in the syntax of the SQL:2003 SQL standards. For example, an equivalent SQL query with the *Streamlog* program in Query 4 can be written as following:

```
Create Stream: ride(Ts: Timestamp Pickup: String, Drop:
    String, Fare: Float)
WITH recursive bestPrice (Ts, Pickup, Drop, min() AS
    Fare) AS
(SELECT Ts, Pickup, Drop, Fare FROM edge) UNION
```

[2]https://www.antlr.org/

```
(SELECT larger(bestPrice.Ts, ride.Ts),
    bestPrice.Pickup, ride.Drop, bestPrice.Fare +
    ride.Fare FROM bestPrice, ride WHERE
    bestPrice.Drop = ride.Pickup)
SELECT Pickup, Drop, Fare FROM WINDOW(bestPrice, 60)
```

VII. Experiments

A. Experiment Setup

1) Datasets and Workloads: We evaluate the proposed techniques using the four *Streamlog* programs introduced in Section III-B: Q_3 (short for Query 3, same for other queries), Q_4, Q_5 and Q_7. Following the applications in real scenarios, we also apply a sliding window on each of them. To provide more insights of the algorithms, we further come up with a version of Q_3 without sliding window and denoted it as Q_8.

We select the dataset for each program based on the real application scenario of them, respectively. Specifically, we ran the query Q_7 over HighEnergy dataset [25]; Q_3 over LUBM [45] and Q_8, which is without window, over a smaller set of LUBM; Q_4 over ChiTaxi2019 [46]; and Q_5 over the largest dataset Twitter [47]. The statistics of each dataset is shown in Table I.

TABLE I
Statistics of Datasets

Name	Duration (Time Span)	Cardinality	Size (MB)
Twitter	73 B	1468 M	41000
LUBM	156 M	31 M	514
LUBM(small)	0.04 M	0.01 M	0.13
ChiTaxi2019	0.5 M	10.4 M	210
HighEnergy	6513	0.35 M	7.4

2) Baseline Methods: To the best of our knowledge, our work is the first one to support recursive query over data stream with a high-level query language. Existing work on query processing over streaming data, such as DSMSes and distributed stream processing systems, cannot support the recursive queries with SQL-like query interface, and extending them to our application scenarios would be far from trivial. On the other hand, the existing Datalog engines that can handle recursive queries are designed for the scenario where data is persistent and queries are transient. Any fair comparison will also have to exclude special purpose direct implementations of specific algorithms that do not use high-level query languages.

Therefore, the proposed techniques will be evaluated with two methods Simple and QBI introduced in Section IV. For each method, we implement two versions: One runs on the Apache Flink platform with procedure programming language so as to show the superiority of our proposed query evaluation techniques. The other is implemented in our prototype to verify the query planning techniques introduced in Section V and to show that the QBI method can be seamlessly integrated into the query planning module of a streaming system.

Indeed the main objective of these experiments is not to explore the superiority of our prototype over existing systems, but to illustrate the power of our high-level query language for expressing recursive queries over data streams, and the effectiveness of the evaluation and query planning techniques proposed for it. The efforts of building a real query engine for *Streamlog* will be left as future work.

3) Environment: As stated above, we have two implementation versions. For the version on Apache Flink, we implemented both methods with Java using the APIs provided by Flink. For our prototype, we implement the system using C++ and the compiler is GCC 9.0 with O3 flag. We run the experiments of all the systems on a server with four AMD Opteron 6376 CPUs (8 physical cores per CPU, 2 hyper-thread per core), 256GB memory (configured into eight NUMA regions) and 1 TB hard disk. The operating system is Ubuntu Linux 14.04 LTS.

B. Main Results

Following the previous studies [10], [11], we use the throughput and tail latency as the main evaluation metrics. Here throughput is the average number of incoming tuples that the system can process per second; while tail latency reflects the 99-th percentile latency of processing a window slide and produce the corresponding resulting tuples. When reporting the main results, the window and slide sizes for each query is fixed as following where the bold ones are corresponding to the results in Figure 6 and 7: We ran the query Q_7 with a window size $\{40, 60, 80, \mathbf{100}\}$ and slide length $\{\mathbf{10}, 20, 30\}$; Q_3 with a window size $\{50000, 100000, 150000, \mathbf{200000}\}$ and slide length $\{\mathbf{10000}, 15000, 20000\}$; For Q_8, we output the computed result whenever a slide-length time has passed, and we run the query with slide length $\{\mathbf{1}, 100, 1000\}$; Q_4 with a window size $\{4000, 6000, 8000, \mathbf{10000}\}$ and slide length $\{\mathbf{1000}, 1500, 2000\}$; and finally Q_5 with a window size $\{1M, 2M, 3M, \mathbf{4M}\}$ and slide length $\{\mathbf{0.1M}, 0.3M, 0.5M\}$.

We first look at the results on Apache Flink shown in Figure 6. Note that since Flink enforces users to specify a window, we did not run Q_8 on this platform (a limitation that underscores the need for having a more expressive language). As we can see, the QBI based method can outperform Simple by 2.6 to 14.9 times in throughput. This is because with the help of QBI, we avoid the duplicate computation caused by the expiration of tuples. Furthermore, we can see that the tail latency of Q_5 is 1,960 and 22,046 seconds for QBI and Simple, respectively.

We can now observe from Figure 7 the results produced by our prototype system. Such results show the similar trends with those on Apache Flink. Actually, the performance gain of QBI on our prototype is even better than that on Apache Flink. For example, on the same query Q_5 with QBI, it has a throughput of 39.9 on the prototype system but 30.9 on Flink. These results can be explained by the fact that the system overhead of Flink is larger than that of our prototype. Moreover, the current API provided by Flink makes the specification of recursive queries difficult, inasmuch that recursive operators must be encoded in the window function program, which the Flink engine then handles in a generic way. It further illustrates the necessity to develop a query engine for *Streamlog* over Flink

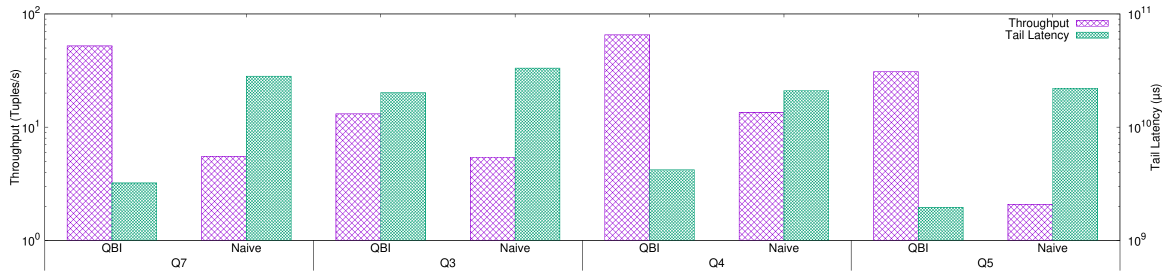

Fig. 6. Main Result on Flink

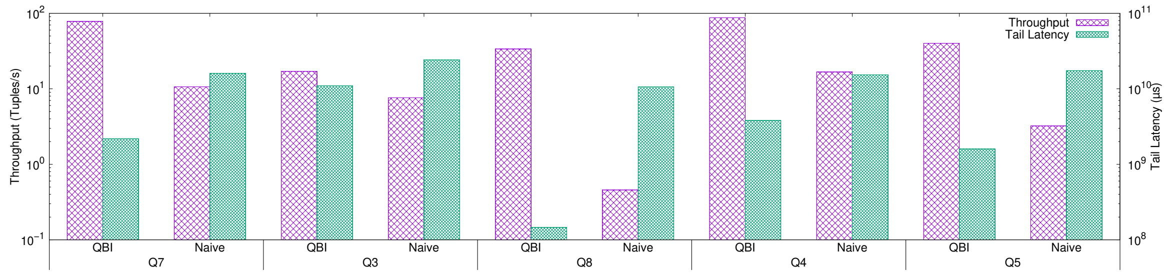

Fig. 7. Main Result on Our Prototype

instead of programming with the current APIs for procedure language, which is left as a promising future direction.

Finally, we also report the results of memory usage on our prototype system in Table II. We can see that the QBI is a light-weighted index structure and the memory overhead is very trivial. For example, even on the largest dataset Twitter with the Q_5 with largest window and smallest slide, thus largest amount of information stored, on average it will cost additional 5.81MB per-window.

TABLE II
AVERAGE MEMORY OVERHEAD PER WINDOW (MB)

Query/Window Size	**Slide Size**		
	10	20	30
Q_7 (W = 40)	0.15	0.15	0.14
Q_7 (W = 60)	0.28	10.25	0.2
Q_7 (W = 80)	0.27	0.26	0.25
Q_7 (W = 100)	0.32	0.32	0.31
	10000	15000	20000
Q_3 (W = 50000)	0.72	0.7	0.68
Q_3 (W = 100000)	2.19	2.15	2.08
Q_3 (W = 150000)	4.48	4.4	4.31
Q_3 (W = 200000)	7.67	7.25	6.82
	1000	1500	2000
Q_4 (W = 4000)	0.38	0.36	0.34
Q_4 (W = 6000)	0.45	0.41	0.39
Q_4 (W = 8000)	0.53	0.49	0.44
Q_4 (W = 10000)	0.61	0.55	0.49
	0.1M	0.3M	0.5M
Q_5 (W = 1M)	1.41	1.31	1.22
Q_5 (W = 2M)	2.62	2.58	2.42
Q_5 (W = 3M)	3.88	3.76	3.63
Q_5 (W = 4M)	5.81	5.66	5.58

C. Detailed Analysis

We then conduct more experiments to show the effect of some important parameters on the overall performance. Specifically, we focus on two parameters in this work: the window size W and slide size s. Since the cardinality of datasets varies greatly for different queries, we try different window and slide sizes for different queries, respectively. The results of QBI are summarized in Table III and Table IV. We can see that QBI scales well along with the increasing window and slide size.

TABLE III
THROUGHPUT (TUPLES/S) WITH DIFFERENT PARAMETERS

Query/Window Size	**Slide Size**		
	10	20	30
Q_7 (W = 40)	128.96	143.24	169.77
Q_7 (W = 60)	90.05	91.87	104.38
Q_7 (W = 80)	82.82	88.69	90.17
Q_7 (W = 100)	78.02	81.13	86.39
	10000	15000	20000
Q_3 (W = 50000)	729.59	763.012	850.89
Q_3 (W = 100000)	179.75	180.34	202.85
Q_3 (W = 150000)	64.90	68.55	76.27
Q_3 (W = 200000)	17.01	23.15	29.33
	1000	1500	2000
Q_4 (W = 4000)	103.15	115.62	124.6
Q_4 (W = 6000)	95.05	108.9	122.33
Q_4 (W = 8000)	94.01	100.94	120.66
Q_4 (W = 10000)	87.39	95.96	118.85
	0.1M	0.3M	0.5M
Q_5 (W = 1M)	192	193.63	219.72
Q_5 (W = 2M)	90.12	94.24	104.11
Q_5 (W = 3M)	66.25	69.08	70.49
Q_5 (W = 4M)	39.99	40.06	42.24

For instance of throughput, Q_5 has the highest throughput with smaller window and large slide because generally less computation is needed in this case. With the increase of window size, more tuples are included and thus the throughput becomes less. When the slide size gets smaller, QBI will help to avoid redundant computation and the throughput decreases due to fact that it need to look up the QBI multiple times. However, Simple would perform much worse in such situations due to the redundant information and thus much less throughput. This is the same case for all other queries.

TABLE IV
TAIL LATENCY WITH DIFFERENT PARAMETERS

Query/Window Size	Slide Size		
	10	20	30
Q_7 (W = 40)	949.86	1009.16	1011.265
Q_7 (W = 60)	1390.05	1491.87	1635.82
Q_7 (W = 80)	1572.82	1688.69	1899.35
Q_7 (W = 100)	2190.18	2281.13	2594.03
	10000	15000	20000
Q_3 (W = 50000)	7203.99	7683.012	8008.02
Q_3 (W = 100000)	8815.95	9014.35	9629.39
Q_3 (W = 150000)	9095.26	10095.23	11340.24
Q_3 (W = 200000)	10993.34	12475.56	13464.34
	1000	1500	2000
Q_4 (W = 4000)	2678.20	2912.96	3411.54
Q_4 (W = 6000)	2739.94	3070.51	3564.66
Q_4 (W = 8000)	2813.63	3253.34	3614.99
Q_4 (W = 10000)	2898.66	3312.45	3812.05
	0.1M	0.3M	0.5M
Q_5 (W = 1M)	282.49	287.017	374.96
Q_5 (W = 2M)	582.59	600.53	611.18
Q_5 (W = 3M)	880.01	885.93	920.17
Q_5 (W = 4M)	1599.05	1599.12	1600.63

VIII. RELATED WORK

A. Stream Query Language

There is a long stream of studies on the query language over data streams in the database community. A comprehensive survey is made in [48]. Dobra et al. [26] introduced how to use aggregates in stream query languages. The Continuous Query Language (CQL) [1] defined an SQL-based declarative language for continuous queries against relational data streams and was implemented in many famous DSMSes. Law et al. [27], [29] proposed to use user-defined aggregates (UDAs) to overcome the problem of loss of expressive powers in continuous query languages. Bai et al. [2] further developed the Expressive Stream Language based on this idea. Jain et al. [21] addressed the problem of unifying the timestamps by coming up with the notion of stream groups and a new operator to manipulate the timestamps. Some recent studies aimed at developing query language for distributed streaming platforms. Begoli et al. [4] illustrated that high-level query language like SQL is an essential query interface for modern distributed streaming platforms, such as KSQL [3] for Apache Kafka and Flink SQL for Apache Flink.

However, none of above studies can express recursive queries illustrated here. There are some languages focusing on stream reasoning over RDF data, such as EP-SPARQL [49] and C-SPARQL [50], but they have not been extended to generalized recursive queries over data stream. There are also initial proposals about recursive query processing [31], [51]. Meanwhile, we provide more use cases and richer semantics, come up with the query evaluation techniques as well as implement a prototype to verify the proposed techniques.

B. Datalog

Many previous works [52], [53], [54], [36] focused on providing formal declarative semantics for the usage of aggregates in recursion. Recently Pre-mappability(PREM) [17] is introduced so that programs under this property using min and max in recursion are equivalent to those aggregate-stratified programs. To efficiently evaluate *Datalog* programs, many Datalog engines are developed, such as LogicBlox [55], DeALS [32] and SociaLite [16]. There are also studies in parallel and scalable Datalog systems in both shared-memory [41], [56] and shared-nothing [33], [42], [43] architectures. Nevertheless, none of these systems supports DSMS style computation where queries are persistent and the data transient. Although there are some studies about incremental view maintenance [57] and computation [58], [59] of Datalog, they have a different problem settings: the main challenges we resolve include those related to unbounded data streams, non-blocking operators, and sliding window semantics with expiration which are not considered in above studies.

C. Data Stream Systems

Data Stream Management System (DSMS) has been a hot topic in database community for a long time. Some earlier works in this field included STREAM [39], Aurora [60] TelegraphCQ [61], StreamBase [62] and Esper [63]. Recent studies focused on distributed stream processing based on the distributed dataflow model of computation. Examples include Twitter Storm [64], Microsoft Trill [65], Drizzle [66], Spark Streaming [5] and Apache Flink [6]. However, none of them can support recursive query processing in high-level query language. Some studies queried streaming graphs by programs written in procedure programming languages [67], [8], [9], regular expression [10] or relational algebra [11]. To the best of our knowledge, our work is the first one to propose a high-level query language for recursive query processing over stream data.

IX. CONCLUSION AND FUTURE WORK

In this paper, we studied the problem of recursive query processing over stream data and proposed *Streamlog*, the first database query language to express such queries. We formally describe the syntax and semantics of *Streamlog* and illustrate its application scope with several examples. Various optimization techniques were introduced to support efficient query evaluation, including an index based method that improves Datalog semi-naive evaluation for stream window queries. We demonstrated the effectiveness of our proposed query planning and execution techniques by implementing the *Streamlog* prototype. Experimental results on several real and synthetic streaming workloads illustrate the feasibility and efficiency of our proposed techniques.

For the future work, we plan to further develop a SQL-style query language for recursive query processing over stream data based on the efforts and results discussed in this paper. We plan to implement a query engine for this language on top of popular distributed streaming platforms such as Apache Kafka to further show the effectivenes of recursive queries over data stream in most real-world scenarios.

REFERENCES

[1] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006.

[2] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo, "A data stream language and system designed for power and extensibility," in *CIKM*, 2006, pp. 337–346.

[3] H. Jafarpour and R. Desai, "KSQL: streaming SQL engine for apache kafka," in *EDBT*, 2019, pp. 524–533.

[4] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. L. Knowles, "One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables," in *SIGMOD*, 2019, pp. 1757–1772.

[5] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative API for real-time applications in apache spark," in *SIGMOD*, 2018, pp. 601–613.

[6] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: Consistent stateful distributed stream processing," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1718–1729, 2017.

[7] "Apache kafka," https://kafka.apache.org/.

[8] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *FAST*, 2019, pp. 249–263.

[9] K. Kim, I. Seo, W. Han, J. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *SIGMOD*, 2018, pp. 411–426.

[10] A. Pacaci, A. Bonifati, and M. T. Özsu, "Regular path query evaluation on streaming graphs," in *SIGMOD*, 2020, pp. 1415–1430.

[11] ——, "Evaluating complex queries on streaming graphs," in *ICDE*, 2022.

[12] J. Wang, J. Wu, M. Li, J. Gu, A. Das, and C. Zaniolo, "Formal semantics and high performance in declarative machine learning using datalog," *VLDB J.*, vol. 30, no. 5, pp. 859–881, 2021.

[13] Y. Li, J. Wang, M. Li, A. Das, J. Gu, and C. Zaniolo, "Kddlog: Performance and scalability in knowledge discovery by declarative queries with aggregates," in *ICDE*, 2021, pp. 1260–1271.

[14] L. Bellomarini, E. Sallinger, and G. Gottlob, "The vadalog system: Datalog-based reasoning for knowledge graphs," *PVLDB*, vol. 11, no. 9, pp. 975–987, 2018.

[15] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *SOSP*, 2005, pp. 75–90.

[16] J. Seo, S. Guo, and M. S. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *ICDE*, 2013, pp. 278–289.

[17] C. Zaniolo, M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi, "Fixpoint semantics and optimization of recursive datalog programs with aggregates," *TPLP*, vol. 17, no. 5-6, pp. 1048–1065, 2017.

[18] C. Zaniolo, M. Yang, M. Interlandi, A. Das, A. Shkapsky, and T. Condie, "Declarative bigdata algorithms via aggregates and relational database dependencies," in *AWM*, 2018.

[19] P. A. Walega, M. Kaminski, and B. C. Grau, "Reasoning over streaming data in metric temporal datalog," in *AAAI*, 2019, pp. 3092–3099.

[20] H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and C. Zaniolo, "SMM: A data stream management system for knowledge discovery," in *ICDE*, 2011, pp. 757–768.

[21] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik, "Towards a streaming SQL standard," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1379–1390, 2008.

[22] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[23] X. Ren, O. Curé, H. Naacke, and G. Xiao, "Bigsr: real-time expressive RDF stream reasoning on modern big data platforms," in *IEEE BigData*, 2018, pp. 811–820.

[24] C. Zaniolo, A. Das, J. Gu, Y. Li, M. Li, and J. Wang, "Monotonic properties of completed aggregates in recursive queries," *CoRR*, vol. abs/1910.08888, 2019.

[25] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *ACM SIGKDD*, 2005, pp. 177–187.

[26] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams," in *SIGMOD*, 2002, pp. 61–72.

[27] Y. Law, H. Wang, and C. Zaniolo, "Query languages and data models for database sequences and data streams," in *VLDB*, 2004, pp. 492–503.

[28] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *PODS*, 2002, pp. 1–16.

[29] Y. Law, H. Wang, and C. Zaniolo, "Relational languages and data models for continuous queries on sequences and data streams," *ACM Trans. Database Syst.*, vol. 36, no. 2, pp. 8:1–8:32, 2011.

[30] R. Reiter, "On closed world data bases," in *Logic and Data Bases*, 1977, pp. 55–76.

[31] C. Zaniolo, "Logical foundations of continuous query languages for data streams," in *Datalog in Academia and Industry - Second International Workshop*, 2012, pp. 177–189.

[32] A. Shkapsky, M. Yang, and C. Zaniolo, "Optimizing recursive queries with monotonic aggregates in deals," in *ICDE*, 2015, pp. 867–878.

[33] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *SIGMOD*, 2016, pp. 1135–1149.

[34] Q. Wang, Y. Zhang, H. Wang, L. Geng, R. Lee, X. Zhang, and G. Yu, "Automating incremental and asynchronous evaluation for recursive aggregate data processing," in *SIGMOD*, 2020, pp. 2439–2454.

[35] K. A. Ross and Y. Sagiv, "Monotonic aggregation in deductive databases," in *PODS*, 1992, pp. 114–126.

[36] M. Mazuran, E. Serra, and C. Zaniolo, "Extending the power of datalog recursion," *VLDB J.*, vol. 22, no. 4, pp. 471–493, 2013.

[37] Y. R. Wang, M. A. Khamis, H. Q. Ngo, R. Pichler, and D. Suciu, "Optimizing recursive queries with progam synthesis," in *SIGMOD*, 2022, pp. 79–93.

[38] S. Ben-David and A. Borodin, "A new measure for the study of on-line algorithms," *Algorithmica*, vol. 11, pp. 73–91, 2005.

[39] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: the stanford stream data manager," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 19–26, 2003.

[40] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo, "The deductive database system LDL++," *TPLP*, vol. 3, no. 1, pp. 61–94, 2003.

[41] J. Wu, J. Wang, and C. Zaniolo, "Optimizing parallel recursive datalog evaluation on multicore machines," in *SIGMOD*, 2022, pp. 1433–1446.

[42] J. Wang, M. Balazinska, and D. Halperin, "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines," *PVLDB*, vol. 8, no. 12, pp. 1542–1553, 2015.

[43] Q. Zhang, A. Acharya, H. Chen, S. Arora, A. Chen, V. Liu, and B. T. Loo, "Optimizing declarative graph queries at large scale," in *SIGMOD*, 2019, pp. 1411–1428.

[44] J. Gu, Y. Watanabe, W. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo, "Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark," in *SIGMOD*, 2019, pp. 467–484.

[45] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semant.*, vol. 3, no. 2–3, p. 158–182, oct 2005. [Online]. Available: https://doi.org/10.1016/j.websem.2005.06.005

[46] "City of chicago taxi trips - 2019," https://data.cityofchicago.org/Transportation/Taxi-Trips-2019/h4cq-z3dy.

[47] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[48] M. Hirzel, G. Baudart, A. Bonifati, E. D. Valle, S. Sakr, and A. Vlachou, "Stream processing languages in the big data era," *SIGMOD Rec.*, vol. 47, no. 2, pp. 29–40, 2018.

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

[50] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, "An execution environment for C-SPARQL queries," in *EDBT*, 2010, pp. 441–452.

[51] A. Das, S. M. Gandhi, and C. Zaniolo, "ASTRO: A datalog system for advanced stream reasoning," in *CIKM*, 2018, pp. 1863–1866.

[52] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The magic of duplicates and aggregates," in *VLDB*, 1990, pp. 264–277.

[53] S. Sudarshan and R. Ramakrishnan, "Aggregation and relevance in deductive databases," in *VLDB*, 1991, pp. 501–511.

[54] S. Ganguly, S. Greco, and C. Zaniolo, "Extrema predicates in deductive databases," *J. Comput. Syst. Sci.*, vol. 51, no. 2, pp. 244–259, 1995.

[55] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *SIGMOD*, 2015, pp. 1371–1382.

[56] M. Yang, A. Shkapsky, and C. Zaniolo, "Scaling up the performance of more powerful datalog systems on multicore machines," *VLDB J.*, vol. 26, no. 2, pp. 229–248, 2017.

[57] S. Singh, S. Madaminov, M. A. Bender, M. Ferdman, R. Johnson, B. Moseley, H. Q. Ngo, D. Nguyen, S. Olesen, K. Stirewalt, and G. Washburn, "A scheduling approach to incremental maintenance of datalog programs," in *IPDPS*, 2020, pp. 864–873.

[58] M. Imran, G. E. Gévay, J. Quiané-Ruiz, and V. Markl, "Fast datalog evaluation for batch and stream graph processing," *World Wide Web*, vol. 25, no. 2, pp. 971–1003, 2022.

[59] M. Imran, G. E. Gévay, and V. Markl, "Distributed graph analytics with datalog queries in flink," in *LSGDA@ VLDB*, 2020, pp. 70–83.

[60] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[61] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *CIDR*, 2003.

[62] "Tibco streambase," https://www.tibco.com/resources/datasheet/tibco-streambase.

[63] "Esper," https://www.espertech.com/esper/.

[64] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," in *SIGMOD*, 2014, pp. 147–156.

[65] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A high-performance incremental query processor for diverse analytics," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 401–412, 2014.

[66] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *SOSP*, 2017, pp. 374–389.

[67] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *EDBT*, 2015, pp. 157–168.