Recursive Computation of Regions and Connectivity in Networks

Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, Boon Thau Loo

Computer and Information Science Department, University of Pennsylvania
Philadelphia, PA, U.S.A.
{mengmeng, netaylor, wenchaoz, zives, boonloo}@cis.upenn.edu

October 31, 2008

Abstract

In recent years, data management has begun to consider situations in which data access is closely tied to network routing and distributed acquisition: sensor networks, in which reachability and contiguous regions are of interest; declarative networking, in which shortest paths and reachability are key; distributed and peer-to-peer stream systems, in which we may monitor for associations among data at the distributed sources (e.g., transitive relationships). In each case, the fundamental operation is to maintain a *view* over dynamic network state; the view is frequently distributed, recursive and may contain aggregation, e.g., describing transitive connectivity, shortest paths, least costly paths, or region membership.

Surprisingly, solutions to this problem are often domain-specific, expensive to compute, and incomplete. In this paper, we recast the problem as one of *incremental recursive view maintenance* in the presence of distributed streams of updates to tuples: new stream data becomes insert operations and tuple expirations become deletions. We develop a set of techniques that maintain information about tuple derivability — a compact form of *data provenance*. We complement this with techniques to reduce communication: aggregate selections to prune irrelevant aggregation tuples, provenance-aware operators that can determine when tuples are no longer derivable and remove them from their state, and shipping operators that greatly reduce the tuple and provenance information being propagated while still maintaining correct answers. We validate our work in a distributed setting with sensor and network router queries, showing significant gains in bandwidth consumption without sacrificing performance.

1 Introduction

As data management has broadened its scope to take on increasingly distributed applications, we are increasingly seeing a blurring of the line between a network and a query processor. In a plethora of emerging applications, data originates at a variety of nodes and is being frequently updated: routing tables in a peer-to-peer overlay network [3] or in a declarative networking system [11, 22]; sensors embedded in an environment [13, 23]; monitors within various clusters at geographically distributed hosting sites [25, 18]; data producers in large-scale distributed scientific data integration [14]. In many of these situations, it is natural to express distributed data acquisition, integration, and processing using declarative queries — and in some cases to compute and incrementally maintain the results of these queries, e.g., in the form of a routing table, an activity log, or a status display.

The queries that are of interest in this domain are frequently quite different from the OLAP or OLTP queries that exemplify centralized DBMS query processing: they involve finding and continuously monitoring (1) contiguous regions with similar readings; (2) connectivity, reachability, or transitive associations; or (3) aggregates over the aforementioned regions or paths (e.g., smallest, shortest, cheapest). These three classes of queries have many overlapping characteristics.

Declarative networking. In declarative networking [22, 21], an extended variant of datalog has been used to manage the state in routing tables — and thus to control how network messages are forwarded through the network. Perhaps the central task in this work is to compute paths available through multi-hop connectivity, based on information in neighboring routers' tables. It has been shown that recursive path queries, used to determine reachability and cost, can express conventional and new network protocols in a declarative way.

Sensor networks. Declarative, database-style query systems have also been shown to be effective in the sensor realm [13, 23]. However, a variety of macroprogramming languages [29, 30] have been proposed as alternatives.

Macroprogramming languages have supported features like region and path computations, whereas database languages have been largely limited to aggregation. In the long run, we argue that the declarative query approach has data independence and optimization benefits. As sensors become increasingly powerful, it makes sense to develop more advanced query runtimes, and the primitive operators that can be used to support declarative queries over regions and paths.

Section 2 provides a number of detailed use cases and declarative queries for regions and paths in these two domains. The use cases are heavily reliant on *recursive* computations, which must be performed over distributed data that is being frequently updated in "stream" fashion (e.g., sensor state and router links are dynamic). The majority of past work on recursive queries [4, 5] has focused on recursion in the context of centralized deductive databases, and some aspects of that work have ultimately been incorporated into the SQL-99 standard and today's commercial databases. However, recursion is relatively uncommon in traditional database applications, and hence little work has been done to extend this work to a distributed setting. We argue that the advent of declarative querying over networks has made recursion of fundamental interest: it is at the core of the main query abstractions we need in a network, namely regions, reachability, shortest paths, and transitive associations.

To this point, only specializations of recursive queries have been studied in networks: in the sensor domain, algorithms have been proposed for computing regions and neighborhoods [19, 29, 30], but these are limited to situations in which data comes from physically contiguous devices, and computation is relatively simple. In the declarative networking domain, a semantics has been defined [22] that closely matches router behavior, but it is not formalized, and hence the solution does not generalize. Furthermore, little consideration has been given to the problem of *incremental recomputation* of results in response to data arrival, expiration, and deletion.

In this paper, we show how to compute and incrementally maintain recursive views over data streams, in support of networked applications. In contrast to previous maintenance strategies for recursive views [17], our approach emphasizes *minimizing the propagation of state* — both across the network (which is vital to reduce bandwidth consumption) and inside the query plan (which reduces computational cost). Our methods generalize to sensors, declarative networking, and data stream processing. We make the following contributions:

- We develop a novel, compact *absorption provenance*, which enables us to directly detect when view tuples are no longer derivable and should be removed.
- We develop a new *MinShip* operator that reduces the number of times that tuples annotated with provenance need to be propagated across the network and in the query.
- We develop a scheme for extending *aggregate selection* to streams of insertions and deletions, in order to reduce the propagation of tuples that do not contribute to the answer.
- We evaluate our schemes within a distributed query processor, and experimentally validate their performance in real distributed settings, with realistic Internet topologies and simualted sensor data.

Section 2 presents use cases for declarative recursive views. In Section 3.2 we discuss the distributed query processing settings we address. Sections 4 through 6 discuss our main contributions: absorption provenance, the *MinShip* operator, and our extended version of aggregate selection. We present experimental validation in Section 7, describe related work in Section 8, and wrap up and discuss future work in Section 9.

2 Distributed Recursive View Use Cases

We motivate our work with several examples that frame network monitoring functionalities as distributed recursive views. This is not intended to be an exhaustive coverage of the possibilities of our techniques, but rather an illustration of the ease with which distributed recursive queries can be used.

Throughout the paper, we assume a model in which logical relations describe state horizontally partitioned across many nodes, as in declarative networking [21]. In our examples, we shall assume the existence of a relation $link(src, dst)$, which represents all router link state in the network. Such state is partitioned according to some key attribute; unless otherwise specified, we adopt the convention that a relation is partitioned based on the value of its first attribute (src), which may (depending on the setting) directly specify an IP address at which the data is located, or a logical address like a DNS name or a key in a content-addressable network [3].

Network Reachability. The textbook example of a recursive query is graph transitive closure, which can be used to compute network reachability. Assume the query processor at node *X* has access to *X*'s routing table. Let a tuple *link(X,Y)* denote the presence of a link between node *X* and its neighbor *Y*. Then the following query computes all pairs of nodes that can reach each other.

Query 1 *Reachability:*

```
reachable(x,y) :- link(x,y).
reachable(x,y) :- link(x,z), reachable(z,y).

with recursive reachable(src,dst) as
 (  select src, dst
    from link
  union
    select link.src, reachable.dst
    from link, reachable
    where link.dst = reachable.src)
```

Datalog provides a concise representation of recursive queries, and has been used in recent work on declarative networks and sensor networks [11, 22]. However, it is by no means the only way to express recursive queries. SQL-99 has support for linear recursion, which comprises a bulk of network queries of interest. In our paper, we present our examples in both Datalog and SQL[1]. The techniques of this paper are agnostic as to the query language.

The SQL query (view) above takes base data from the *link* table, then recursively joins $link$ with its current contents to generate a transitive closure of links. Note that since all tables are originally partitioned based on the src attribute, computing the view requires a distributed join that sends $link$ tuples to nodes based on their dst attributes, who join with *reachable.src*.

There are many potential enhancements to this query, e.g., to compute reachable pairs within a radius, or to find cycles.

Network Shortest Path. We next consider how to compute the shortest path between each pair of nodes:

Query 2 *Shortest Path:*

```
path(x,y,p,c,l) :- link(x,y,c), p=concat([x,y],nil), l=1.
path(x,y,p,c,l) :- link(x,z,c0), path(z,y,p1,c1,l1), c=c0+c1, p=concat([x],p1), l=1+l1.
minCost(x,y,min<c>) :- path(x,y,p,c,l).
minHops(x,y,min<l>) :- path(x,y,p,c,l).
cheapestPath(x,y,p,c) :- path(x,y,p,c,l), minCost(x,y,c).
fewestHops(x,y,p,l) :- path(x,y,p,c,l), minHops(x,y,l).
shortestCheapestPath(x,y,p1,c,p2,l) :- cheapestPath(x,y,p1,c), fewestHops(x,y,p2,l).

with recursive path(src,dst,vec,cost,length) as
 ((select src, dst, src | '.' | dst, cost, 1
   from link
  ) union (
   select L.src, P.dst, L.src |'.'| P.vec, L.cost+P.cost, P.length+1
   from link L, path P
   where L.dst = P.src
 ))

create view minCost(src,dst,cost) as
  (select src, dst, min(cost) from path
   group by src, dst)

create view minHops(src,dst,length) as
  (select src, dst, min(length) from path
   group by src, dst)

create view cheapestPath(src,dst,vec,cost) as
  (select P.src, P.dst, P.vec, P.cost
   from path P, minCost C
   where P.src = C.src and P.dst = C.dst and P.cost = C.cost)
```

[1]We assume SQL UNIONs with set semantics, and that a query executes until it reaches fixpoint. Not all SQL implementations support these features.

```
create view fewestHops(src,dst,vec,length) as
  (select P.src, P.dst, P.vec, P.length
   from path P, minHops H
   where P.src = H.src and P.dst = H.dst and P.length = H.length)

create view shortestCheapestPath(src,dst,vec1,cost,vec2,length) as
  (select P.src, P.dst, P.vec, P.cost, H.vec, H.length
   from cheapestPath P, fewestHops H
   where P.src = H.src and P.dst = H.dst)
```

This represents the composition of serveral views. The *path* recursive view is similar to the previous *reachable* query, with additional computation of the path cost, path length, as well as the path itself. The expression *p = concat([x], p1)* is satisfied if *p* is the path produced by appending link *x* to the existing path *p1*.

Here *minCost* computes an aggregate over *path* that considers the path costs, *cheapestPath* determines the paths with minimum cost. Similarly, *minHops* and *fewestHops* determine the length of the fewest hop path, and the set of paths with that length respectively. Finally *shortestCheapestPath* combines the results from the *cheapestPath* with *fewestHops*.

As it stands, this query is inefficient since *path* enumerates all possible paths, and may not terminate. However, query optimization techniques such as *aggregate selections* [28] exist to improve performance by automated rewriting of the query. We will revisit this optimization with modifications for the distributed setting.

Sensing Contiguous Regions. In addition to querying the graph topology itself, distributed recursive queries can be used to detect regions of neighboring nodes that have correlated activity. One example is a *horizon* query, where a node computes a property of nodes within a bounded number of hops of itself. A second example as shown below starts with a series of reference nodes, and computes contiguous regions of triggered sensors near these nodes. Then it is able to find the largest region which contains the most sensors in it.

Query 3 *Largest Region:*

```
activeRegion(rid,x) :- sensor(x,posx), mainSensorInRegion(rid,x), isTriggered(x).
activeRegion(rid,y) :- sensor(x,posx), sensor(y,posy), isTriggered(x),
   activeRegion(rid,x), distance(posx,posy) < k.
regionSizes(rid, count<x>) :- activeRegion(rid, x).
largestRegion(max<size>) :- regionSizes(rid, size).
largestRegions(rid) :- regionSizes(rid,size), largestRegion(size).

with recursive activeRegion(regionid,sensorid) as
  ( select M.regionid, S.sensorid
    from sensor S, mainSensorInRegion M, isTriggered T
    where M.sensorid = S.sensorid and S.sensorid = T.sensorid
   union
    select A.regionid, S2.sensorid
    from sensor S1, sensor S2, activeRegion A, isTriggered T
    where distance(S1.coord, S2.coord) < k
          and S1.sensorid = A.sensorid and S1.sensorid = T.sensorid )

create view regionSizes(regionid,size) as
  (select regionid, count(sensorid)
   from activeRegion
   group by regionid)

create view largestRegion(size) as
   (select max(size) from regionSizes)

create view largestRegions(regionid) as
   (select R.regionid
    from regionSizes R, largestRegion L
    where R.size = L.size)
```

Other Example Queries. The *routing resilience* query counts the number of paths (alternate routes) between any two nodes. Another class of queries examines multicast or aggregation trees constructed within the network. A query could compute the height of each subtree and store this height at the subtree root. Alternatively, we might query for the *imbalance* in the tree – the difference in height between the lowest and highest leaf node. Finally, a query could identify all the nodes at each level of the tree (referred to as the “same generation” query in the datalog literature).

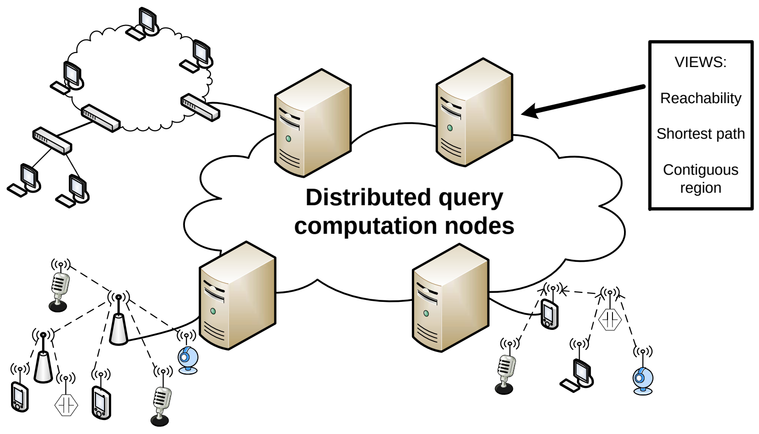


Figure 1: Basic *partially distributed* architecture: query processing nodes are placed in a number of sub-networks. Each collects state information about its sub-network, and the nodes share state to compute distributed recursive views such as shortest paths across the network.

3 Execution Model and Approach

We consider techniques applicable to a broad variety of networked environments, and we make few assumptions about our execution environment. We assume that our networked query processor executes across a number of distributed nodes in a network; in addition, we allow for the possibility that there exist other legacy nodes that may not run the query processor (as indicated in Figure 1). In this flexible architecture, the query processing nodes will serve as proxy nodes storing state information (connectivity, sensor status, etc) about devices on their sub-networks: IP routers, overlay nodes, sensors, devices, etc.

Individual sub-networks may have a variety of types of link-layers (wired IP, wireless IP with a single base station, multi-hop wireless/mesh, or tree-structured sensor networks). They may even represent different autonomous systems on the Internet backbone, or different locations within a multi-site organization. Through polling, notifications, or snooping, our distributed query processing nodes can acquire detailed information about these sub-networks. The query processing nodes each maintain a *horizontal partition* of one or more views about the overall network state: cross-sub-network shortest paths, regions that may span physically neighboring sub-networks (e.g., a fire in a multi-story building), etc. During operation, the nodes may exchange state with one another, either (1) to partition state across the nodes according to keys or ranges, or (2) to perform computation of joins or recursive queries.

Importantly, in a volatile environment such as a network, both sensed state and connectivity will frequently change. Hence a major task will be to *maintain* the state of the views, as base data (sensor readings, individual links) are added or deleted, as distributed state ages beyond a *time-to-live* and gets expired, and as the effects of deletions or expirations get propagated to derived data.

3.1 Query Execution Model

Query execution is a distributed, continuous stream computation, over a set of horizontally partitioned base relations that are updated constantly. We assume that all communication among nodes is carried out using a reliable in-order delivery mechanism. We also assume that our goal is to compute and update *set* relations, not bag relations: we stop computing recursive results when we reach a fixpoint.

In our model, inputs to a query are streams of deletions or insertions over the base data. Hence, we process more general *update streams* rather than tuple streams. *Sliding windows*, commonly used in stream processing, can be used to process *soft-state* [26] data, where the time-based window size essentially specifies the useful lifetime of base tuples. Thus, a base tuple that results from an insertion may receive an associated timeout, after which the tuple gets deleted. When this happens, the derived tuples that depend on the base tuples have to be deleted as well. Due to the needs of network state management, we consider timeouts or windows to be specified over base data only, not derived tuples.

reachable(src,dst)
(derivation step 1)

tuple	at → to	pv
(A, B)	A	p_1
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	p_4
$(\mathbf{A}, \mathbf{C})$	B → A	$p_1 \wedge p_2$
$(\mathbf{B}, \mathbf{A})$	C → B	$p_2 \wedge p_3$
$(\mathbf{B}, \mathbf{B})$	C → B	$p_2 \wedge p_4$
$(\mathbf{C}, \mathbf{B})$	A → C	$p_1 \wedge p_3$
$(\mathbf{C}, \mathbf{C})$	B → C	$p_2 \wedge p_4$

reachable(src,dst)
(derivation step 2)

tuple	at → to	pv
(A, B)	A	p_1
(A, C)	A	$p_1 \wedge p_2$
(B, A)	B	$p_2 \wedge p_3$
(B, B)	B	$p_2 \wedge p_4$
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	$p_4 \vee (p_1 \wedge p_3)$
(C, C)	C	$p_2 \wedge p_4$
$(\mathbf{A}, \mathbf{A})$	B → A	$p_1 \wedge p_2 \wedge p_3$
$(\mathbf{A}, \mathbf{B})$	B → A	$p_1 \wedge p_2 \wedge p_4$
$*(\mathbf{B}, \mathbf{B})$	C → B	$p_1 \wedge p_2 \wedge p_3$
$(\mathbf{B}, \mathbf{C})$	C → B	$p_2 \wedge p_4$
$(\mathbf{C}, \mathbf{A})$	B → C	$p_2 \wedge p_3 \wedge p_4$
$(\mathbf{C}, \mathbf{B})$	B → C	$p_2 \wedge p_4$
$(\mathbf{C}, \mathbf{C})$	A → C	$p_1 \wedge p_2 \wedge p_3$

reachable(src,dst)
(derivation step 3)

tuple	at → to	pv
(A, A)	A	$p_1 \wedge p_2 \wedge p_3$
(A, B)	A	p_1
(A, C)	A	$p_1 \wedge p_2$
(B, A)	B	$p_2 \wedge p_3$
(B, B)	B	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	$p_4 \vee (p_1 \wedge p_3)$
(C, C)	C	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$
$*(\mathbf{A}, \mathbf{B})$	B → A	$p_1 \wedge p_2 \wedge p_3$
$*(\mathbf{B}, \mathbf{C})$	C → B	$p_1 \wedge p_2 \wedge p_3$
$(\mathbf{C}, \mathbf{A})$	A → C	$p_1 \wedge p_2 \wedge p_3$
$*(\mathbf{C}, \mathbf{B})$	A → C	$p_1 \wedge p_2 \wedge p_4$

reachable(src,dst)
(derivation step 4)

tuple	at → to	pv
(A, A)	A	$p_1 \wedge p_2 \wedge p_3$
(A, B)	A	p_1
(A, C)	A	$p_1 \wedge p_2$
(B, A)	B	$p_2 \wedge p_3$
(B, B)	B	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	$p_4 \vee (p_1 \wedge p_3)$
(C, C)	C	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$

Figure 2: Recursive derivation of $reachable$ in recursive steps (bold indicates new derivations). The "at" column shows where the data is produced. The "to" column shows where it is shipped after production (if omitted, the derivation remains at the same node. The "pv" column contains the *absorption provenance* of each tuple (Section 4). A tuple marked "*" is an extra derivation only shipped in the absorption provenance model.

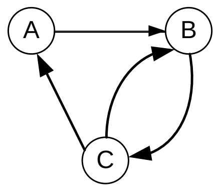


Figure 3: Network represented in **link** relation

3.2 Motivation for New Distributed Recursive Techniques

To illustrate the need for our approach, we consider an example. Assume our goal is to maintain, at every node, the set of all nodes reachable from this node. Refer to Figure 2, which shows a network consisting of three nodes and four links (visualized in Figure 3). Each node "knows" its direct neighbors: we represent these in the $link$ table, consisting of four entries $link(A, B)$, $link(B, C)$, $link(C, A)$, and $link(C, B)$. As in our previous examples, the $link$ table is partitioned such that all values with source src are stored on node src. In our simple example, there is a one-to-one mapping between attributes and physical storage, although one can decouple each location from its physical attribute by using logical addresses (e.g., doing a hash-based partitioning).

Now we define a materialized view $reachable(src, dst)$ that is again partitioned such that all values with source src are stored on node src. This query computes the transitive closure over the $link$ table, and is the query shown in the **Network Reachability** example of Section 2. Unlike in traditional recursive query execution (e.g., for datalog), here computing the transitive closure requires a good deal of communications traffic: $link$ data must be shipped to the node corresponding to its dst attribute in order to join with $reachable$ tuples[2]; and the output of this join may need to be shipped to a new location depending on what its src is. Figure 2 steps through the execution of $reachable$, showing state after each computation step in semi-naïve evaluation (equivalent to steps in stratified execution), as well as communication (the "$at \rightarrow to$" columns). (We defer discussion of the column marked pv.) Given the input $link$ tuples, the result of executing this query is the set of $reachable$ tuples stored at the src node: essentially at fixpoint, all nodes compute their reachability set. In this case, since we have a fully-connected network, the final resulting $reachable$ table at every node trivially contains the set of all nodes in the network.

Computing the View Instance. The base-case contents of $reachable$ are computed directly from $link$, as specified

[2]Or vice-versa, depending on the query plan.

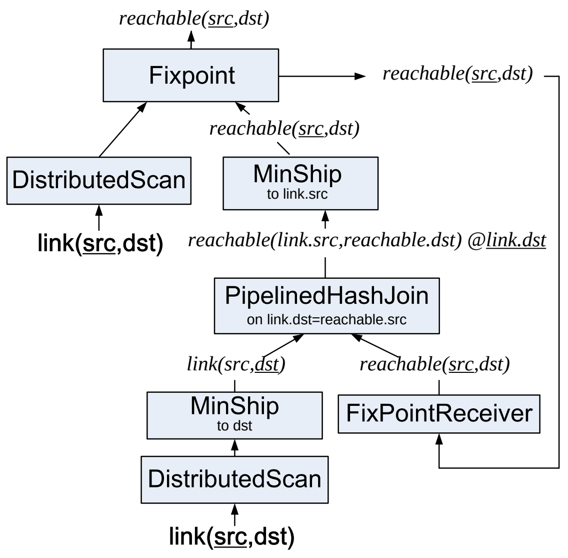


Figure 4: Plan for $reachable$ query. Underlined attributes are the ones upon which data is partitioned.

in the first "branch" of the view definition (Section 2). The recursive query block joins all $link$ tuples with those currently in $reachable$. Since the tables are distributed by their first attribute, all $link$ tuples must first be shipped to nodes corresponding to their dst attribute, where they are joined with $reachable$ tuples with matching srcs. Finally, the resulting $reachable$ tuples must be shipped to the nodes corresponding to their src attributes. For instance, in step 1, $reachable(C, B)$ is computed by joining $link(C, A)$ and $reachable(A, B)$ as computed from step 0. That requires first shipping $link(C, A)$ to node A, performing the join with $reachable(A, B)$ to generate $reachable(C, B)$, and sending the resulting tuple to node C. In our figure, we indicate the communication for the resulting $reachable$ table in the third column as $A \to C$.

Since we are following set-semantics execution, duplicate removal will eliminate tuples with identical values; but this only occurs *after* they are created and sent to the appropriate node. For instance, consider $reachable(C, C)$, which is first computed in step 1 and sent to node C. During step 2, node A re-derives this same tuple; however, it must send this result to node C before the duplication can be detected, and the tuple eliminated. In total, 16 tuples (4 initial $link$ tuples, and 12 $reachable$ tuples) are shipped during the recursive computation. In the final step, a fixpoint is reached when no new tuples are derived.

Incremental Deletion (Standard Approach). Now consider the case when $link(C, B)$ expires (hence is deleted). Commonly used schemes such as counting tuple derivations — used in maintaining non-recursive views — cannot be used here. Instead, one must rely on a standard algorithm for recursive view maintenance, in particular DRed [17]. DRed works by first *over-deleting* tuples conservatively and then *re-deriving* tuples that may have alternative derivations. Figure 5 shows the DRed deletion phase (steps 0-4), followed by the rederivation phase (steps 5-8). In the deletion phase, we first delete $reachable(C, B)$ based on the initial deletion of $link(C, B)$. This in turns leads to the deletion of all $reachable$ tuples with $src = C$ (step 1), then those with $src = B$ (step 2) and $src = A$ (step 3). The $reachable$ table is empty in step 4. DRed will ultimately re-derive *every* $reachable$ tuple, as shown in steps 5-8. Overall, DRed requires shipping a total of 16 tuples, equivalent to computing the entire $reachable$ view from scratch, despite having just a single deletion.

In the above example, DRed is prohibitively expensive — and we have not even considered the requirement that DRed waits until all deletions have been processed before it can rederive. (This requires distributed synchronization, which may be expensive.) Perhaps surprisingly, our example illustrates the *common case*: most networks are well-connected with bi-directional connectivity along several redundant paths. DRed will over-delete such paths, and then re-derive data. In our example, deleting a single link resulted in the deletions of all *reachable* tuples; yet, it is clear that nodes A, B, and C are still connected after the link is deleted. One source of deletions is tuple expirations; given the fact that large-scale network tends to be highly dynamic, tuples will need to expire frequently, thus triggering frequent

reachable(src,dst)
(step 0)

tuple	at
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
(C, A)	C
(C, B)	C
$-(\mathbf{C}, \mathbf{B})$	C
(C, C)	C

reachable(src,dst)
(step 1)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
(C, A)	C
(C, C)	C
$-(\mathbf{C}, \mathbf{A})$	B → C
$-(\mathbf{C}, \mathbf{B})$	B → C
$-(\mathbf{C}, \mathbf{C})$	B → C

reachable(src,dst)
(step 2)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
$-(\mathbf{B}, \mathbf{A})$	C → B
$-(\mathbf{B}, \mathbf{B})$	C → B
$-(\mathbf{B}, \mathbf{C})$	C → B

reachable(src,dst)
(step 3)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
$-(\mathbf{A}, \mathbf{A})$	B → A
$-(\mathbf{A}, \mathbf{B})$	B → A
$-(\mathbf{A}, \mathbf{C})$	B → A

reachable(src,dst)
(step 4)

tuple	at → to

reachable(src,dst)
(step 5)

tuple	at → to
(A, B)	A
(B, C)	B
(C, A)	C

reachable(src,dst)
(step 6)

tuple	at → to
(A, B)	A
(B, C)	B
(C, A)	C
$(\mathbf{A}, \mathbf{C})$	B → A
$(\mathbf{B}, \mathbf{A})$	C → B
$(\mathbf{C}, \mathbf{B})$	A → C

reachable(src,dst)
(step 7)

tuple	at → to
(A, B)	A
(A, C)	A
(B, A)	B
(B, C)	B
(C, A)	C
(C, B)	C
$(\mathbf{A}, \mathbf{A})$	B → A
$(\mathbf{B}, \mathbf{B})$	C → B
$(\mathbf{C}, \mathbf{C})$	A → C

reachable(src,dst)
(step 8)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
(C, A)	C
(C, B)	C
(C, C)	C
$(\mathbf{A}, \mathbf{B})$	B → A
$(\mathbf{B}, \mathbf{C})$	C → B
$(\mathbf{C}, \mathbf{A})$	A → C

Figure 5: DRed algorithm: over-delete and re-derive steps after deletion of **link(C,B)**.

recomputation and exacerbating the overhead.

3.3 Our Approach

The major challenge with distributed incremental view maintenance lies in handling deletions of tuples. In general, we must either buffer base tuples, then recompute the majority of the query (as in our example); or we must maintain *state* at intermediate nodes, which enables them to propagate the appropriate updates when a base tuple is removed. We adopt the latter approach, developing a scheme that:

- Maintains a concise form of *data provenance* — bookkeeping about the derivations and derivability of tuples — such that the existence of a view tuple can be directly determined when a base tuple is removed. (Section 4.)
- Propagates as little provenance or tuple derivation information from one node to another as possible, in order to minimize network and computation costs, but preserves sufficient state to support deletions. (Section 5.)
- Distributes aggregate computations, such that only minimal state is propagated by the aggregate operations, but enough state is preserved to support deletions. (Section 6.)

To help frame the discussion in the next three sections, consider an execution plan for the $reachable$ query, shown in Figure 4. This plan is disseminated to all nodes, from which it continuously generates and updates partitions of the reachability relation. The left *DistributedScan* represents the table scan required for the base case, which fetches the contents of $link$ and sends them to the *Fixpoint* operator. In the recursive case, the *Fixpoint* invokes the right subtree of the query plan: it sends its current contents to a *FixPointReceiver*, where they are joined via a *PipelinedHashJoin* with a copy of $link$ — whose contents have been re-partitioned and shipped to the nodes corresponding to the dst attribute. The output is shipped to the fixpoint via the *MinShip* operator, which in the simplest case simply ships data.

4 Provenance for Efficient Deletions

In order to support view maintenance when a base tuple is deleted, we must be able to test whether a derived tuple is *still derivable*. Rather than delete and re-compute (as with DRed), we instead propose to keep around metadata about derivations, i.e., *provenance* [8], also called *lineage* [12].

Provenance alternatives. Different proposed forms of provenance capture different amounts of information. Lineage in [12] encodes the set of tuples from which a view tuple was derived — but this is not sufficiently expressive to distinguish what happens if a base tuple is removed. Alternatives include why-provenance [8], which encodes the

Algorithm 1 Fixpoint operator

```
$Fixpoint(B^{\triangle}, R^{\triangle})$
Inputs: Input base stream $B^{\triangle}$, recursive stream $R^{\triangle}$
Output: Output stream $U'^{\triangle}$
 1: Init hash map $P$: $U(\bar{x}) \rightarrow$ provenance expressions over $U(\bar{x})$
 2: if there is a aggregate selection option then
 3:     Get the grouping key $\overline{uk}$, number of aggregate functions $n$ and aggregate functions $agg_1, \cdots, agg_n$
 4:     $B'^{\triangle} := AggSel(B^{\triangle}, \overline{uk}, n, agg_1, \cdots, agg_n)$
 5:     $B^{\triangle} := B'^{\triangle}$
 6:     $R'^{\triangle} := AggSel(R^{\triangle}, \overline{uk}, n, agg_1, \cdots, agg_n)$
 7:     $R^{\triangle} := R'^{\triangle}$
 8: end if
 9: while not $EndOfStream(B^{\triangle})$ and not $EndOfStream(R^{\triangle})$ do
10:     Read an update $u$ from $B^{\triangle}$ or $R^{\triangle}$
11:     if $u.type$ = INS then
12:         if $P$ does not contain $u.tuple$ then
13:             $P[u.tuple] := u.pv$
14:             Add $u.tuple$ to the view
15:             Output $u$ to the next operator
16:         else
17:             $oldPv := P[u.tuple]$
18:             $P[u.tuple] = P[u.tuple] \vee u.pv$
19:             $deltaPv := P[u.tuple] \wedge \neg oldPv$
20:             if $oldPv \neq P[u.tuple]$ then
21:                 $u'.tuple := u.tuple$
22:                 $u'.type$ := INS
23:                 $u'.pv := deltaPv$
24:                 Output $u'$ to the next operator
25:             end if
26:         end if
27:     else if $u$ is from $B^{\triangle}$ then
28:         for each $t$ in $P$ do
29:             $oldPv := P[t]$
30:             $P[t] = restrict(oldPv, \neg u.pv)$
31:             if $P[t] = false$ then
32:                 Remove $t$ from $P$
33:                 Remove $t$ from the view
34:             end if
35:         end for
36:     end if
37: end while
```

set of *sets of source tuples* that produced the answer; and the general provenance representation of [14, 15], which we term *relative provenance* here. In physical form, the latter encodes a derivation graph capturing which tuples are created as an *immediate consequent* of others. The graph can be traversed after a deletion to determine whether a tuple is still derivable from base data [14]. Either of these latter two forms of provenance will allow us to detect whether a view tuple remains derivable after a deletion of a base tuple. However, to our knowledge, why-provenance is always created "on demand" and has no stored representation; and relative provenance encodes all derivations, which we later show can be expensive in a distributed setting.

Moreover, we note that the tuple derivability problem has several properties for which we can optimize. In particular, base (EDB) tuples may each participate in *many* different derivations — yet the deletion of that base tuple "invalidates" all of these derivations. View maintenance requires testing each view tuple for derivability once base tuples have been removed — which can be determined by testing all of view tuples' derivations for their dependencies on the deleted base tuples.

Our compact representation. We define a simplified provenance model, *absorption provenance*, which starts with the following intuition. We annotate every tuple in a view with a Boolean expression: the tuple is in the view iff the expression evaluates to **true**. Let the provenance annotation of a tuple t be denoted $\mathbf{P}(t)$. For base relations, we set $\mathbf{P}(t)$ to a variable whose value is **true** when the tuple is inserted, and reset to **false** when the tuple gets deleted. The relational algebra operators return provenance annotations on their results according to the laws of Figure 6 (this matches the Boolean specialization of provenance described in the theoretical paper [15]).

Our key innovation with respect to provenance is to develop a physical representation in which we can exploit

$\sigma_\theta(R)$:	If tuple t in R satisfies θ, annotate t with $\mathbf{P}(t)$
$R_1 \bowtie R_2$:	For each tuple t_1 in R_1 and tuple t_2 in R_2, annotate the output tuple $t_1 \bowtie t_2$ with $\mathbf{P}(t_1) \wedge \mathbf{P}(t_2)$.
$R_1 \cup R_2$:	For each tuple t output by $R_1 \cup R_2$, annotate t with $\mathbf{P}(t_1) \vee \mathbf{P}(t_2)$, where $\mathbf{P}(t_1)$ is **false** iff t does not exist in R_1; similarly for $\mathbf{P}(t_2)$, R_2
$\Pi_A(R)$:	Given tuples $t_1, t_2, \ldots, t_n$ that project to the same tuple t', annotate t' with $\mathbf{P}(t_1) \vee \mathbf{P}(t_2) \vee \cdots \vee \mathbf{P}(t_n)$

Figure 6: Relational algebra rules for composition of provenance expressions.

Boolean absorption to *minimize the provenance expressions*: absorption is based on the law $a \wedge (a \vee b) \equiv a \vee (a \wedge b) \equiv a$, and it eliminates terms and variables from a Boolean expression that are not necessary to preserve equivalence. We term this model *absorption provenance*, and it describes in a minimal way exactly which tuples, in which combinations of join and union, are essential to the existence of a tuple in the view. The more compact provenance annotations reduce the network traffic. Even better, we can update a view after a base tuple has been deleted as follows: for each base tuple, we substitute value **false** for its provenance variable, within all provenance annotations of tuples in the view. If applying absorption to the tuple's provenance results in the value **false**, we remove the tuple. Otherwise, it remains derivable.

Absorption provenance in the example of Figure 2. Absorption provenance adds a bit of overhead to normal query computation: the fixpoint operator must propagate a tuple through to the recursive step whenever it receives *a new derivation* (even of an existing tuple), not simply when it receives a new tuple. Refer back to the $reachable$ query example of Figure 2. The pv column shows the absorption provenance for every tuple during the initial view computation, with respect to the input $link$ tuples labeled p_1, p_2, p_3, and p_4; we see that an additional 4 tuples (beyond the previous set-oriented execution model) is shipped during query evaluation, as a result of computing absorption provenance. For instance, $reachable(B, B)$ is derived in both strata 1 and 2. They have different provenance, hence we must track both derivations.

Absorption provenance shows its value in handling deletions. When $link(C, B)$ is deleted, the only step required with absorption provenance is to zero out p_4 in the provenance expressions of all $reachable$ tuples. In this example, zeroing out this derivation only requires two message transmissions, and it does not result in the removal of any tuples from the view. (It is still possible that deletions may need to be propagated to all nodes in the network, in the worst case.)

4.1 Implementing Absorption Provenance

There are multiple alternatives when attempting to encode an absorption provenance expression and to develop operators over this expression. Each expression can, of course, be normalized to a sum-of-products expression, since in the end there are possibly multiple derivations of the same tuple, and each derivation is formed by a conjunctive rule (or a conjunction of tuples that resulted from conjunctive rules). From there we could implement absorption logic that is invoked every time the provenance expression changes. We choose an alternative — and often more compact — encoding for absorption provenance: the *binary decision diagram* [7] (BDD), a compact encoding of a Boolean expression in a DAG. A BDD (specifically, a *reduced ordered BDD*) represents each Boolean expression in a canonical way, which automatically eliminates redundancy by merging isomorphic subgraphs and removing isomorphic children: this process automatically applies absorption. Since BDDs are frequently used in circuit synthesis applications and formal verification, many highly optimized libraries are available [20]. Such libraries, e.g., [20], provide abstract BDD types as well as Boolean operators to perform on them: pairs of BDDs can be ANDed or ORed; individual BDDs can be negated; and variables within BDDs can be set or cleared. We exploit such capabilities in our provenance-aware stateful query operators.

Now we describe in detail the implementation of absorption provenance within the Fixpoint operator. We defer a discussion of how aggregation state management works to Section 6.

4.2 Fixpoint Operator

The key operator for supporting recursion is the Fixpoint operator, which first calls a **base case** query to produce results, then repeatedly invokes a **recursive case** query. It repeatedly unions together the results of the base case and each recursive step, and terminates when no new results have been derived. We define the fixpoint in a recursive query as follows: we reach a fixpoint when we can no longer derive any new results that affect the absorption provenance of any tuple in the result.

Unlike traditional semi-naïve evaluation, our fixpoint operator does not block or require computations in synchronous rounds (or iterations), a prohibitively expensive operation in distributed settings. We achieve this with the use of pipelined semi-naïve evaluation [21], where tuples are arrived in the order in which they arrive via the network (assuming a FIFO channel), and are only processed with tuples that arrive previously.

Pseudocode for this operator is shown in Algorithm 1. The fixpoint operator receives insertions from either the base (B^{Δ}) or recursive (R^{Δ}) streams. It maintains a hash table P containing the absorption provenance of each tuple that it has received, which remains derivable. Note that in our algorithms, each tuple contains three attributes, $type$ which indicates whether it is an INS or DEL tuple, $tuple$ which records its raw tuple values, and pv which stores its provenance.

Initially (Lines 2–8), we apply any portions of an aggregation operation that might have been "pushed into" the fixpoint — this uses a technique called *aggregate selection* discussed in Section 6. Now, upon receipt of an insertion operation u (Lines 11–26), the fixpoint operator first determines whether the tuple has already been encountered, and perhaps with a different provenance. If u is new, it is simply stored in $P[u.tuple]$ as the first possible derivation; otherwise we merge it with the existing absorption provenance in $P[u.tuple]$. We save the resulting *difference* in $deltaPv$. If the provenance has indeed changed despite absorption, u gets propagated to the next operator, annotated with provenance $deltaPv$.

Deletions are handled in a straightforward fashion (Lines 27–35), given our implementation of absorption provenance. In our scheme deletions on the recursive stream are essentially caused by the deletions on the base stream. Hence, we only need to focus on deletion tuples generated from the base (B^{Δ}) stream. When we meet a deletion operation u, for each tuple t in the table P, we zero out the associated provenance of tuple u ($u.pv$) from the provenance expression of each t ($P[t]$), computed by bdd operation "restrict" [20] shown in Line 30. If the result is a provenance expression returning **false** (zero), a deletion operation on t is propagated to the next operator after removing its entry from P.

4.3 Join Operator

The PipelinedHashJoin operator is modified from a regular pipelined hash join to incorporate the use of absorption provenance. The join operator needs to maintain two pairs of hashtables: the h_R and h_S maintains the tuples indexed on the join keys $\overline{Rk}$ and $\overline{Sk}$ of each R and S tuple respectively, and p_R and p_S maintains the provenance indexed on all attributes of each R and S tuple. The hashtables are similar to those used in the earlier Fixpoint operator, except we need to maintain two hashtables, one for each input table.

We will describe in terms of processing a new update tuple u from R^{Δ}. Processing of updates from S^{Δ} is symmetrical. We consider two cases, when u is a deletion or an insertion. Replacements are treated as a deletion followed by an insertion.

4.3.1 Deletions

We first consider the case where u is a delete tuple. We focus on describing the *HalfPipeDel* function in Algorithm 2 invoked by the main *PipeHashJoin* function for each delete tuple u. Two sets of updates need to be performed. First, the internal state (i.e. hashtables of join tuples and respective provenance) maintained by the join operator is updated. Second, new tuples are output from the join, and propagated as deletions up to the next operator according to the query plan.

Operator state update (Lines 1– 8 in *HalfPipeDel*): The provenance state $p_u[u.tuple]$ for tuple u is retrieved, and u's provenance $u.pv$, is deleted from $p_u[u.tuple]$ (In BDD operations, $x - y \equiv x \wedge \neg y$). This essentially clears the absorption provenance of the derivation of u. If $p_u[u.tuple]$ is **false** (zero), which means that all possible derivations of u have been deleted, then we need to remove u from subsequent joins by purging its entry from both p_u and h_u hashtable.

Delete propagation (Lines 9– 16 in *HalfPipeDel*): The deletion of u may cascade the deletion of other tuples when $p_u[u.tuple]$ has been changed. New tuples u' are formed by joining u with matching tuples t in $h_j[u.tuple[\overline{u}k]]$ retrieved using the join key $\overline{u}k$ of u. The absorption provenance of u' is set to the join of $u.pv$ and $p_j[t]$(computed by the BDD as $u.pv \wedge p_j[t]$).The resulting u' tuple with the new absorption provenance is then propagated up the query plan.

4.3.2 Insertions

If u is an insert tuple, a similar set of updates are performed. We focus on the *HalfPipeIns* function in Algorithm 2, which is similarly invoked from *PipeHashJoin* for each insert tuple u.

Operator state update (Lines 1– 7 in *HalfPipeIns*): First, the provenance $p_u[u.tuple]$ is retrieved and updated based on the $u.pv$ of the new tuple u. This is required since the new u tuple may have a different derivation from any u that have been derived previously. In addition, if we haven't met u before, we add u to the hashtable h_u indexed by the join key $\overline{u}k$ of u. Since there may be multiple tuples with a common join key, we maintain a set of them.

Insert propagation (Lines 8– 15 in *HalfPipeIns*): Similar to deletions, the join operator may output new tuples when $p_u[u.tuple]$ has been changed. The h_j hashtable is probed using the join key of u, and each resulting u' tuple from the join is propagated up the query plan. For each matching t in $h_j[u.tuple[\overline{u}k]]$ used in the join, the absorption provenance of u' is set to the join of $u.pv$ and $p_j[t]$ (computed by the BDD as $u.pv \wedge p_j[t]$).

4.3.3 Tuple Expirations

To this point, we have discussed the stream join in terms of processing updates *without* considering window semantics. In our algorithm, we encapsulate the window-checking logic in functions W_R and W_S. Each takes a new update, plus the provenance hash table describing the current contents of the relation. As we have previously mentioned, we only support windowing on base relations — for non-base relations, W_R or W_S simply perform no operations and return the empty set. For base relations, the window function (1) updates any relevant internal windowing relations based on the provenance from the new tuple (e.g., advancing the timestamp, or, if the update was an insertion, adding the new tuple's identity as the last-received), and (2) returns the set of tuples that have expired, with the specific provenance terms that have expired.

Calling the window functions (Lines 8–12 in *PipeHashJoin*): . After processing the update, we now pass it along to the window function, so that it may update its internal state. The window function may then return a set of tuples that have expired, with provenance. We delete each tuple from the join.

5 Minimizing Propagation of Tuple Provenance

With provenance, each time a given operator receives a new *derivation* of a tuple, it must typically propagate that tuple and derivation, in much the same fashion as it would a completely new tuple. If a tuple is derivable in many ways, it will be processed many times, just as a tuple might be propagated multiple times in a bag relation (versus a set). This increases the amount of work done in query processing, as well as the amount of state shipped across the network can increase correspondingly. Worse, in the general case, a recursive query may produce an infinite number of possible derivations.

Fortunately, absorption helps in the last case. If a new tuple derivation is received whose provenance is completely absorbed, we do not need to propagate any information forward. We will reach a fixpoint when we can no longer derive any new results that affect the absorption provenance of any tuple in the result.

However, we must take additional steps to reduce the amount of state shipped by our distributed query processor nodes. In order to minimize the number of *tuple derivations with provenance expressions*, we propagate through the query plan and the network, while still maintaining the ability to handle deletions. Here we define a special stateful *MinShip* operator. *MinShip* replaces a conventional *Ship* operator, but maintains provenance information about the tuples produced by incoming updates. It always propagates the *first* derivation of every tuple it receives, but simply buffers all subsequent derivations of the same tuple — merely updating their absorption provenance. By absorption, the stored provenance expression absorbs multiple derivations into a simpler expression.

Now if the original tuple derivation is deleted, *MinShip* responds by propagating forward any alternate derivations it has buffered — then it propagates the deletion of the first derivation. Additionally, depending on our preferences about state propagation, we can require the *MinShip* operator to propagate all of its buffered state periodically, e.g., when the buffer exceeds a capacity or time threshold. By changing the batching interval or conditions, we can adjust how many alternate derivations are propagated through the query plan — a smaller interval will propagate more state, and a larger interval will propagate less state. In the extreme case, we can set the interval to infinity, resulting in what we term *lazy provenance propagation*. In the lazy case, alternate derivations of a tuple will only be propagated when they affect downstream results; this significantly reduces the cost of insertions. (In some cases it may slightly increase the cost of deletion propagation.)

If we go back to our example in Figure 2, basically our MinShip algorithm would buffer all the newly derived $reachable$ tuples computed at the same node, given that a certain $reachable$ tuple from that node has already been shipped. For example, $reachable(B, B)$ derived at node C twice will only be shipped once in the first stratum. The second derivation is buffered at node C. Similarly, $reachable(B, C)$ derived in strata 2 and 3 at node C needs to be shipped once, and the second derivation is buffered. The buffering mechanism eliminates a good deal of excess traffic, but is limited in scope to a single node: $reachable(C, B)$ is generated at different nodes, and hence no buffering can be done. Overall, buffering potentially reduces the number of messages communicated if the delayed communication result in the absorption of provenances of similar derivations.

If adopting *lazy propagation*, where *only one* derivation is shipped, and the other are buffered until the previously shipped derivation has expired. In this case, the second derivation of $reachable(B, C)$ will be stored at node C until the previous derivation has been deleted. With lazy propagation, we note that the number of messages communicated to evaluate the query is 16, which is the same as not storing any provenance. Meanwhile, deletions become cheaper than DRed, and flood propagation of deletions are avoided.

The pseudocode for the new ship operator is shown in Algorithm 3. The algorithm takes as input S, which is the outgoing socket connection to the recipient node, a current window size W that is used for batching purposes and an input stream U^{Δ}. The operator maintains three hashtables that maps from tuples to their respective absorption provenance: P_{ins} which stores batched insert tuple provenances seen so far; P_{del}, which consists only of delete tuple provenances that are batched for delete propagation; and B_{sent}, which indicates the tuple provenances that have been sent. We describe the algorithm in terms of the following operations:

Base case and insertions (Lines 10–28): In the base case (Lines 11–13), the ship operator sees the first instance of u. Given no prior u, this tuple is shipped right away (via the $sendMessage$ command in line 12) since its insertion would directly affect the final output result. Whenever $sendMessage$ command is issued, the u is added to the hashtable B_{sent}. On the other hand, if u already exists (Lines 15–27), there are two circumstances: If u is an insert tuple, then it first checks if its provenance can be absorbed into the provenances already being sent, if not then it adds its provenance $u.pv$ to P_{ins}; If u is a delete tuple, then it needs to wipe out the provenances in P_{ins} of this derivation of u, and then adds its provenance $u.pv$ to P_{del}.

Batched shipping (Lines 29–31): Periodically, the absorption provenance needs to be batched and sent from both P_{ins} and P_{del}. This is triggered either using a counter-based approach by flushing after every W tuples (an alternative using periodic timers can easily be incorporated). At each batched shipment, there are two modes: eager shipping and lazy shipping(shown in two sub-functions $BatchShipEager$ and $BatchShipLazy$ respectively).

- Eager shipping (Lines 1–14 in $BatchShipEager$): In the eager shipping mode, it basically ships every tuple with their absorbed provenance in P_{ins} and P_{del} respectively.

- Lazy shipping (Lines 1–14 in $BatchShipLazy$): In the lazy shipping mode, it does not eagerly propagate every derivation in P_{ins}. Whenever there is a deletion tuple t in P_{del}, it propagates its corresponding provenance in $P_{ins}[t]$. In this way, it avoids over-delete in the end.

End-of-stream (Lines 33): When an end-of-stream signal is received by the ship operator, no more new update tuples will be received. This triggers the final shipment of any outstanding tuples in a similar fashion as a single invocation of batched shipping.

6 Minimizing Propagation of State

Our third challenge is to minimize the amount of state (not just the number of alternate derivations) that gets propagated from one node to the next. Given that aggregation is commonplace in network-based queries (as in most queries of Section 2), we need a way to also suppress tuples that have no bearing on the output aggregate values. We adapt a technique called *aggregate selection* [28] to a streaming model, with a *windowed* aggregation (group-by) operation [24]. We consider MIN, MAX, COUNT, and SUM functions[3]). In essence, the aggregate computation is split between a sub-component that is used internally by stateful operators like the $Fixpoint$ and $MinShip$, and a final aggregation computation is done at the end. Our main contributions are to support revision (particularly deletion) of results within a windowed aggregation model, and to combine aggregate selection with minimal provenance shipping.

[3]AVERAGE can be derived from SUM and COUNT, as in [10].

Our aggregate selection ($AggSel$ for short) module (Algorithm 4) can be embedded within any operator that ships state(In our system, both $Fixpoint$ and $MinShip$ have calls to this module). The module takes as input a stream $U^{\triangle}$, a grouping key $\overline{uk}$, the number of aggregate functions n, and a set of aggregate functions $agg_1, agg_2, \cdots, agg_n$. The module maintains a hashtable H indexed on the grouping key $\overline{uk}$, which records all the buffered tuples met so far based on its grouping key values — this is necessary to support tuple deletion. A corresponding hashtable P maps from each tuple to their absorption provenance. Another hashtable B is maintained to record the value associated with each aggregate attribute agg_i, for the grouping key $\overline{uk}$. $AggSel$ finally outputs a stream $U'^{\triangle}$ of the update tuples.

Here is a detailed description of the module.

Insertions (Lines 6–29): If the input tuple is an insertion operation, it first updates its state in H and P (Lines 7–12): if it hasn't met u before, it adds u to the hashtable H indexed by the join key $\overline{u}k$ of u; if it met u before, it updates its provenance in table P. Then, if the provenance of $u.tuple$, $P[u.tuple]$, has been changed, it does the following operations (Lines 14–28). For each aggregate function in this module, it checks whether this insertion tuple affects the aggregate value associated with this tuple's grouping key uk. More specifically, for a certain aggregate function agg_i, if there is no aggregate value associated with u's grouping value $u.tuple[\overline{uk}]$, which means u is the first tuple received so far on this grouping value, then the aggregate value is updated; if tuple $u.tuple$ is better than the current aggregate value $B[u.tuple[\overline{uk}]]$, then a deletion operation on the old aggregate value should be propagated. After checking all the aggregate functions, if at least one of the aggregate values is changed, then u should be propagated as output; if none of them is affected, it just ignores u and propagates nothing.

Deletions (Lines 30–56): If the input tuple is a deletion operation, in similar to insertions, it first updates its state in H and P (Lines 31–36). Note that if H doesn't contain this tuple (Line 30), which means it meets deletions before insertions, in our assumption this is not allowed, so it just ignores the input. It updates u's provenance in table P by deleting $u.pv$ from $P[u.tuple]$ (In BDD operations, $x - y \equiv x \wedge \neg y$). If $P[u.tuple]$ is **false** (zero), which means that all possible derivations of u have been deleted, then we need to remove $u.tuple$ from both P and H hashtable. Next, if the provenance of $u.tuple$, $P[u.tuple]$, has been changed, then it does the following operations(Lines 37–56). For each aggregate function in this module agg_i, if the aggregate tuple is just the same as this tuple $u.tuple$, then it traverses through the current version of buffered tuple table B, computes the next aggregate value if there's any, and propagates an *insertion* of the new aggregate value. After checking all the aggregate functions, if at least one of the aggregate values is affected, then this deletion tuple u should be propagated; if none of them is affected, it just ignores u and propagate nothing.

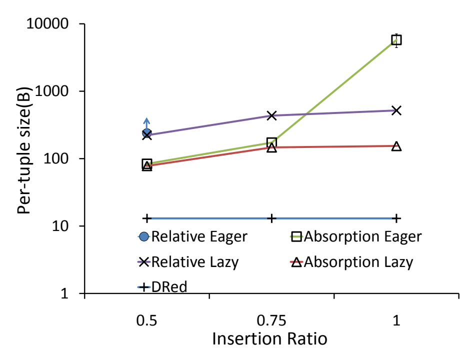
7 Experimental Evaluation

We have developed a Java-based distributed query processor that implements all operators as described in Sections 4-6. Our implementation utilizes the FreePastry 2.0_03 [27] DHT for data distribution, and JavaBDD v1.0b2 [20] as the BDD library for absorption provenance maintenance. Our experiments are carried out on two clusters: a 16-node cluster consisting of quad-core Intel Xeon 2.4GHz PCs with 4GB RAM running Linux 2.6.23, and an 8-node cluster consisting of dual-core Pentium D 2.8GHz PCs with 2GB RAM running Linux 2.6.20. The machines are internally connected within each cluster via a high-speed Gigabit network, and the clusters are interconnected via a 100Mbps network shared with the rest of campus traffic. Our default setting involves 12 nodes from the first cluster; when we scale up, we first use all 16 nodes from this cluster, then add 8 more nodes from the second cluster to reach 24 nodes. All experimental results are averaged across 10 runs with 95% confidence intervals included.

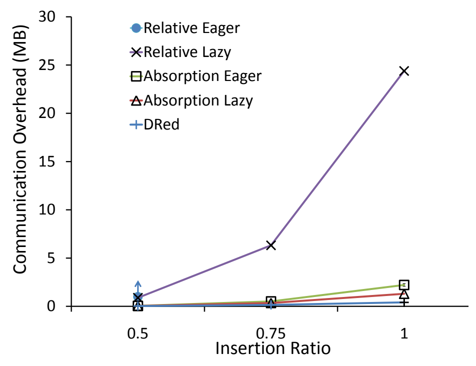
7.1 Experimental Setup

We utilize two sets of query workloads that are representative of our use cases:

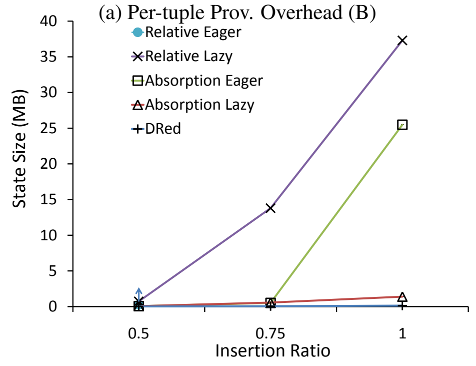
Workload 1: Declarative networks. Our query workloads consist of the *reachable query* (Query 1 in Section 2) and the *shortest-path* query (Query 2 in Section 2). As input to these queries, we use simulated Internet topologies generated with the GT-ITM [16], a package that is widely used to model Internet topologies. By default we use GT-ITM to create "transit-stub" topologies consisting of eight nodes per stub, three stubs per transit node, and four nodes per transit domain. In this setup, there are 100 nodes in the network, and approximately 200 bidirectional links (hence 400 $link$ tuples) in the network. Each input $link$ tuple contains src and dst attributes, as well as an additional $latency$ cost attribute. Latencies between transit nodes are set to 50 ms, the latency between a transit and a stub node is 10 ms,



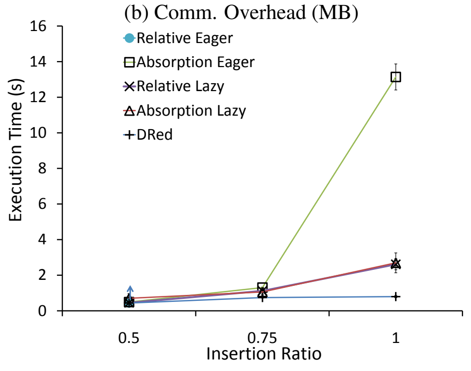
(a) Per-tuple Prov. Overhead (B)



(b) Comm. Overhead (MB)



(c) State within operators (MB)



(d) Convergence time (s)

Figure 7: $reachable$ query computation as **insertions** are performed

and the latency between any two nodes in the same stub is 2 ms. To emulate network connectivity changes, we add and delete $link$ tuples during query execution.

Workload 2: Sensor networks. Our second workload consists of region-based sensor queries executed over a simulated 100m by 100m grid of sensors, where the sensors report data to their local query processing node. We include 5 "seed" groups, each initialized to contain a single device. Our recursive view (below) finds contiguous (within k meters, where by default k=20) triggered nodes and adds them to the group — or removes them if they are no longer triggered. The query is as Query 3 in Section 2.

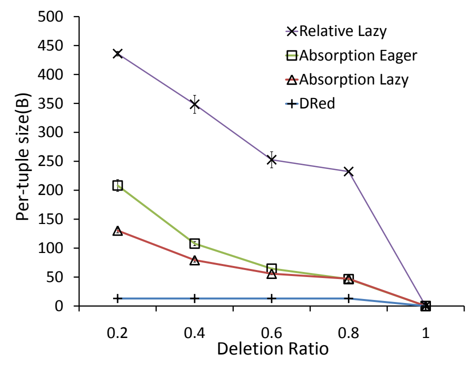
Initially all the seed sensors are triggered. Also we trigger half of the sensors in the network to study the effects of insertions, and then randomly remove them to study the effects of deletions. Note that while the input topology simulates a grid-based sensor topology, the queries are executed over our real distributed query processor implementation.

Our evaluation metrics are as follows:

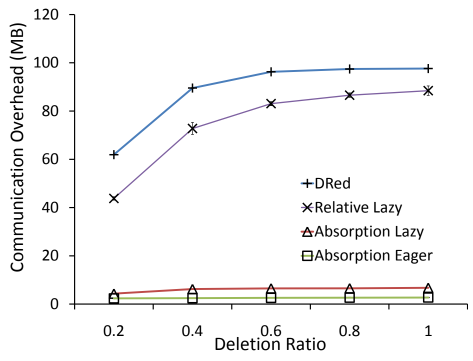
- **Per-tuple provenance overhead(B):** the space taken by the provenance annotations on a per-tuple basis.
- **Communication overhead(MB):** the total size of communication messages on each distributed node for executing a distributed query to completion.
- **Per-node states of operators(MB):** the total overhead of states maintained inside operators on each distributed node.
- **Convergence time(s):** the time taken for a distributed query to finish execution on all distributed nodes.

7.2 Incremental View Maintenance with Provenance

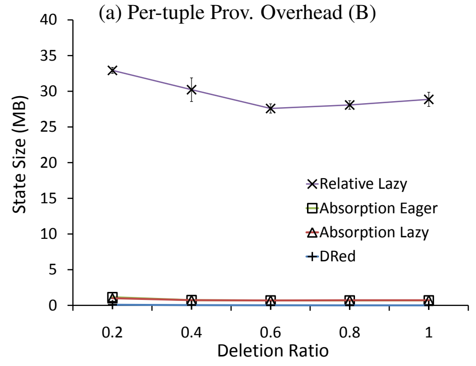
Our first set of experiments focuses on measuring the overhead of incremental view maintenance. Using the $reachable$ query as a starting point, we compare three different schemes: the traditional *DRed* recursive view maintenance strategy, *relative provenance* [14] where each tuple is annotated with information describing derivation "edges" from



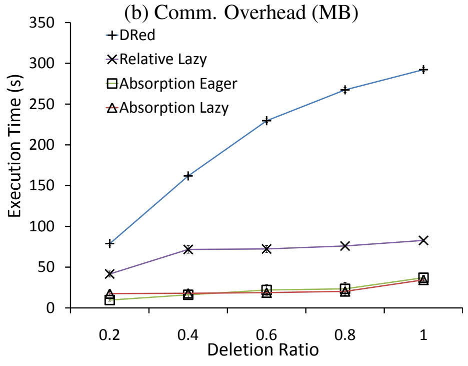
(a) Per-tuple Prov. Overhead (B)



(b) Comm. Overhead (MB)

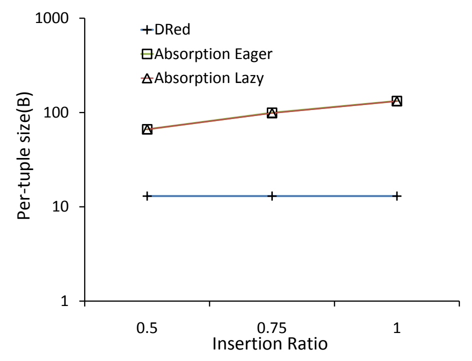


(c) State within operators (MB)

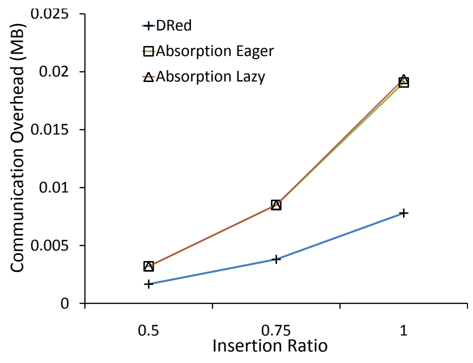


(d) Convergence time (s)

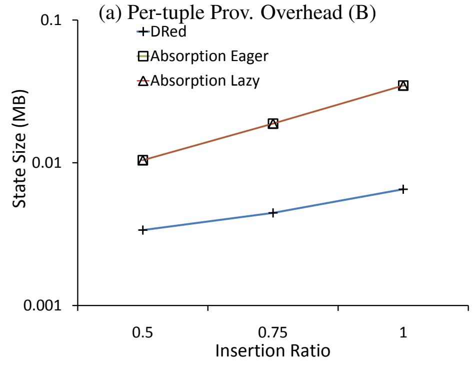
Figure 8: $reachable$ query computation as **deletions** are performed



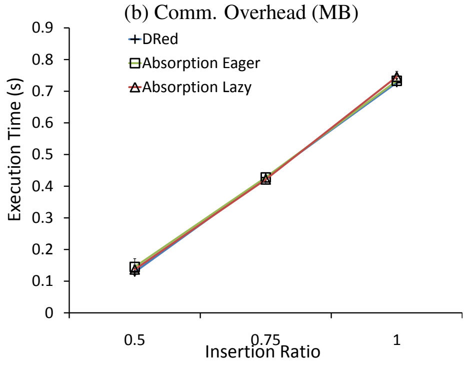
(a) Per-tuple Prov. Overhead (B)



(b) Comm. Overhead (MB)

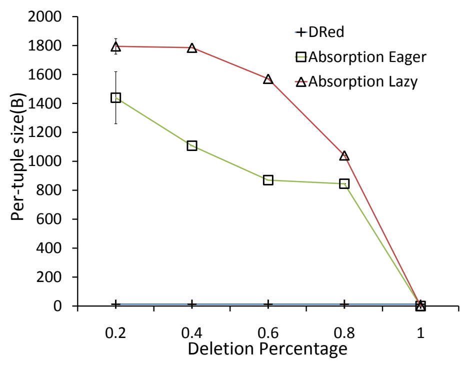


(c) State within operators (MB)

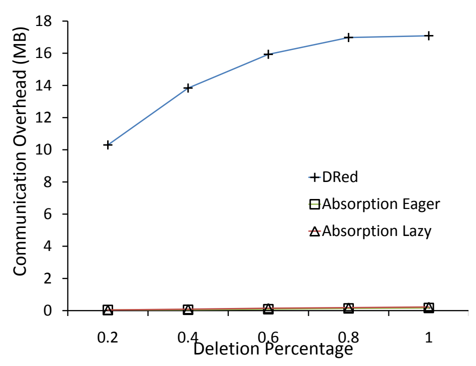


(d) Convergence time (s)

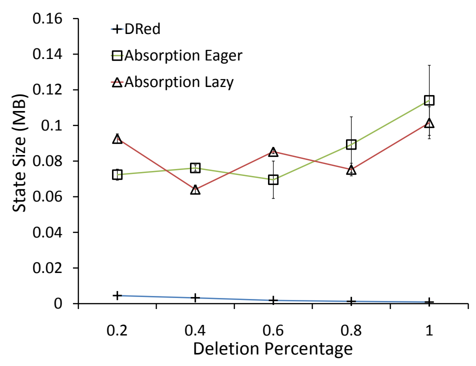
Figure 9: $region$ query computation as **insertions** are performed



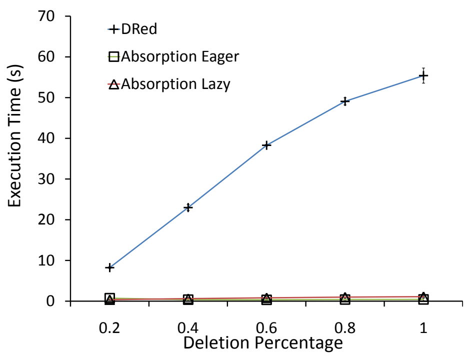
(a) Per-tuple Prov. Overhead (B)



(b) Comm. Overhead (MB)



(c) State within operators (MB)



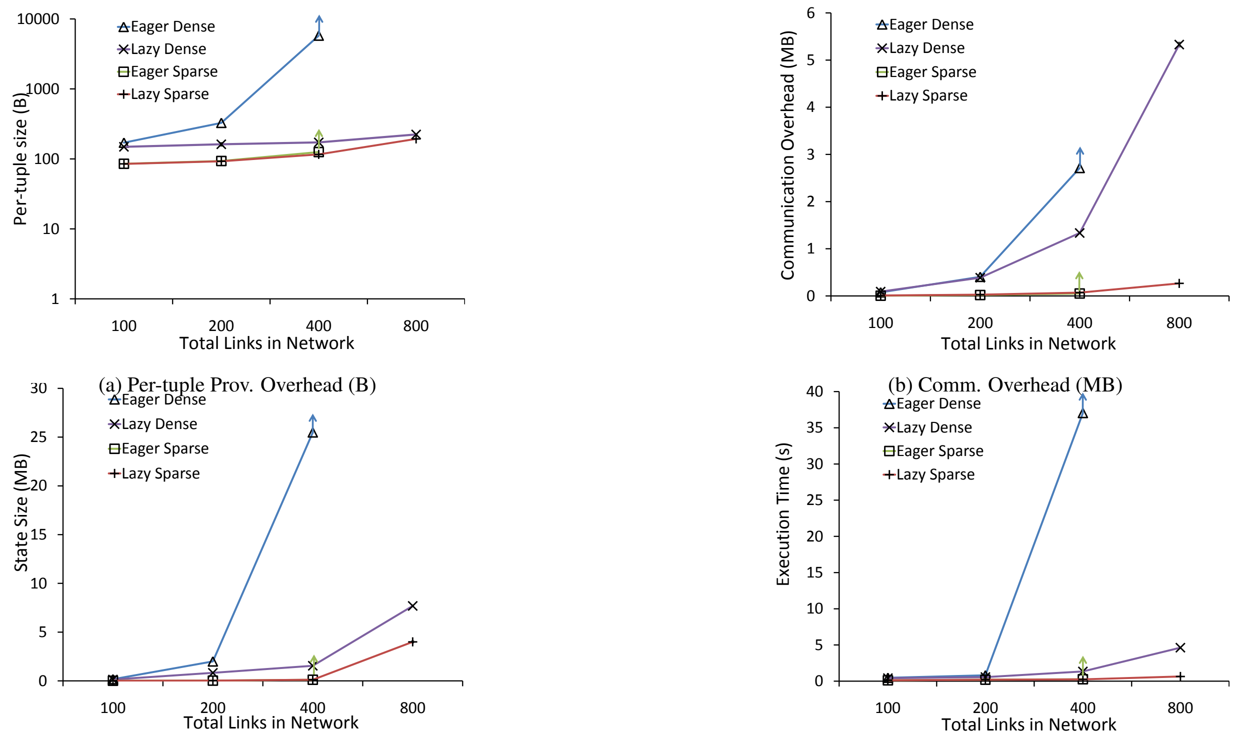
(d) Convergence time (s)

Figure 10: $region$ query computation as **deletions** are performed

other tuples, and our proposed *absorption provenance*. We also consider two schemes for propagating provenance: an *eager* strategy (propagate state from $MinShip$ once a second) and a *lazy* one (propagate state only when necessary).

Insertions-only workload: We first measure the overhead of maintaining provenance, versus normal set-oriented execution. Figure 7 shows the performance of the $reachable$ query, where the Y-axis shows our four evaluation metrics, and the X-axis shows the fraction of links inserted, in an incremental fashion, up to the maximum of 400 link tuples required to create the 100-node GT-ITM topology. Given an insertion-only workload, *DRed* has the best overall performance, since no provenance needs to be computed or maintained. Relative provenance encodes more information than absorption provenance, resulting in larger tuple annotations, more communication, and more operator state. Relative provenance with eager propagation (*Relative Eager*) did not converge within 5 minutes for insertion ratios of 0.75 or higher; hence, we only show lazy propagation (*Relative Lazy*) for the remaining graphs. Eager propagation with absorption provenance (*Absorption Eager*) also is costly due to the overhead of sending every new derivation of a tuple. Lazy propagation of absorption provenance (*Absorption Lazy*) is clearly the most efficient of the provenance schemes.

Insertions-followed-by-deletions workload: Our next set of experiments separately measures the overhead of deletions: here provenance becomes useful, whereas in the insertion case it was merely an overhead. (One can estimate the performance over a mixed workload by considering the relative distribution of insertions vs. deletions and looking at the overheads on each component.) Given the same 100-node topology, after inserting all the $link$ tuples as above, we then delete $link$ tuples in sequence. Each deletion occurs in isolation and we measure the time the query results take to converge after every deletion is injected. Figure 8 shows that *DRed* is prohibitively expensive for deletions when compared to our absorption provenance schemes: it is an order of magnitude more expensive in both communication overhead and execution time. Relative provenance wins versus *DRed* in communication cost and convergence time because it does not over-delete and re-derive. However, its performance is far worse than absorption provenance, and it also incurs more per-tuple overhead and operator state. Relative provenance relies on graph traversal operations to determine derivability from base tuples (see [14]), and thus is expensive in a distributed setting. In contrast, absorption provenance directly encodes whether a derived tuple is dependent on a base tuple. Overall, absorption provenance is



(a) Per-tuple Prov. Overhead (B) (b) Comm. Overhead (MB)

(c) State within operators (MB) (d) Convergence time (s)

Figure 11: Increasing the number of links (and nodes) for the $reachability$ query over inserts

the most efficient method in deletion handling, and consequently ships fewer tuples than the other methods. Taking both insertions and deletions into account, *Absorption Lazy* has the best mix of performance.

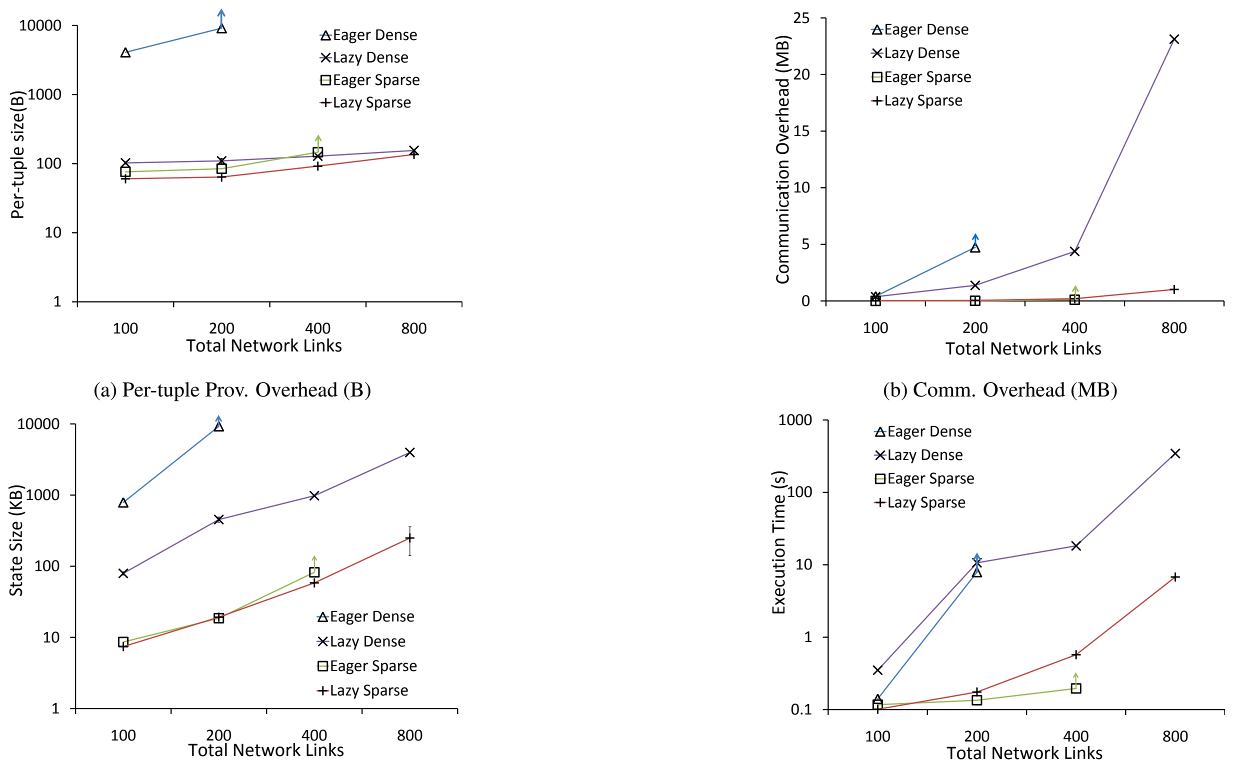
Region-based sensor query: The $region$ query is over a different topology from $reachable$, and it exhibits slightly different update characteristics. Still, as we see in Figure 9, which measures performance with the insertion workload described earlier in the experimental setup, performance follows similar patterns. (The overhead is lower across each of the four metrics, since the network is smaller here and neighbors must be within close proximity.) Under deletion loads, the trends shown by the $region$ query shown in in Figure 10 also closely mirror that of the reachable query. Since the queries exhibit similar performance, we focus on *reachable* in the rest of our experiments.

7.3 Scalability

Next we consider the scale-up of our methods, with respect to inputs and to query processing nodes.

Scaling Data. We increase the number of input link tuples, by increasing the average number of transit nodes in the GT-ITM generated topology. We considered two network topologies: each node in the *dense* topology has four links (as in our default setting), whereas the *sparse* setting has half the number of links for a given network size. Figure 11 shows the insertion-only workload. We observe that the dense network is more costly to evaluate than the sparse network: there are more derivations. Here, lazy propagation is essential: *Eager Dense* did not complete after 5 minutes on a 800-link network, whereas *Lazy Dense* finished in under 5 seconds. We further experimented with deleting an additional 20% of the links shown in Figure 12. Observations are similar as the insertion case.

Increasing Query Processing Nodes. Next, we increase the number of query processing nodes to up to 24 machines, while keeping the input dataset constant. Figure 13 shows the results. Per-tuple provenance overhead increases, then eventually levels off, as the number of nodes increases: each node will now process fewer tuples, and the opportunities of absorption and buffering are reduced. Higher numbers of query processors leads to a reduction in query execution latency, per-node communication overhead, and per-node operator state. The increase of latency between 16 and 24 nodes is due to the low-bandwidth connection between our two subnets. In all cases, DRed incurs higher communication overhead and takes longer to complete than our approach. A more detailed analysis is as follows:



(a) Per-tuple Prov. Overhead (B)

(b) Comm. Overhead (MB)

(c) State within operators (MB)

(d) Convergence time (s)

Figure 12: Increasing the number of links (and nodes) for the *reachability* query over deletions

- Per-tuple provenance size slightly increases because more derivations of insertions are triggered by deletions in MinShip(deletions at every peer would trigger its insertions).
- Communication cost slightly increases because deletion tuples are broadcast to more nodes, also there are insertion tuples triggered by deletions in MinShip(same reason as before). However, in *DRed*, it decreases since we actually compute per-node communication cost here.
- States of operators decreases since less tuple states are maintained on each peer.
- Execution time decreases since each peer has less tuples to process, but 24-cluster has a jump because of slower network connections between 16-node and 8-node subnetwork.

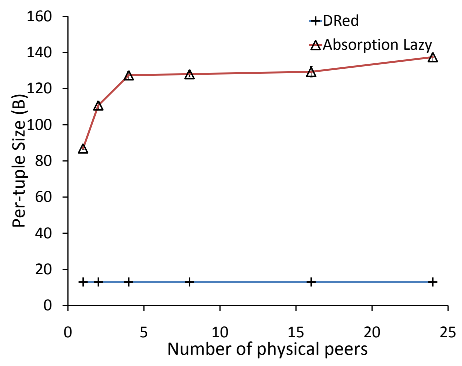
7.4 Multi-aggregate Selection

Figure 14 shows the effectiveness of aggregate selections over the dense and sparse topology of 100 nodes. We experiment with two extensions of the *shortest path* query presented in Section 2: *Multi AggSel* computes two aggregates (one for lowest path cost and the other for shortest hop count); *Single AggSel* minimizes only based on the path cost metric. We observe that aggregate selections are most effective in dense topologies, and *Multi AggSel* costs only half as much as *Single AggSel* due to aggressive pruning of the two aggregates simultaneously. Without the use of aggregate selections, all queries are prohibitively expensive, and do not complete within 5 minutes for dense topologies.

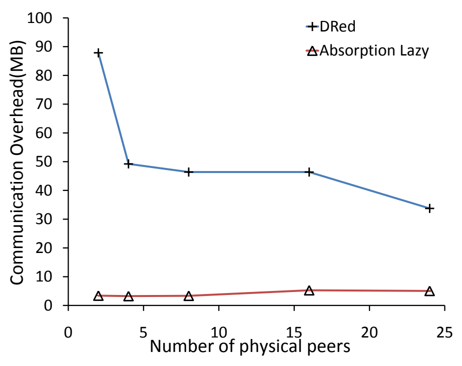
7.5 Summary of Results

We summarize our results with reference to the contributions of this paper as outlined in Section 3.3.

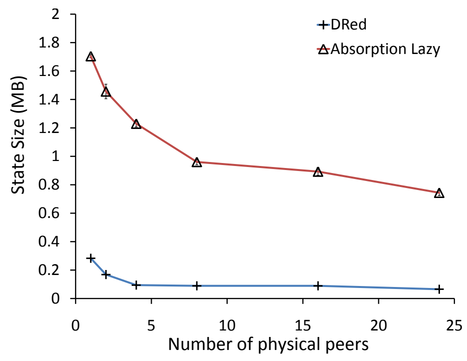
- *Absorption provenance* (Section 4) results in an order-of-magnitude reduction compared to traditional schemes such as DRed, in communication overhead and execution times for view maintenance with insertions and dele-



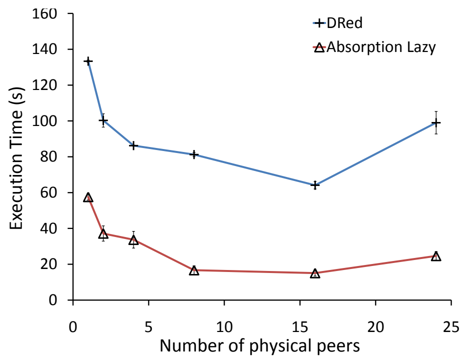
(a) Per-tuple Prov. Overhead (B)



(b) Comm. Overhead (MB)



(c) State within operators (MB)



(d) Convergence time (s)

Figure 13: Varying the number of *physical query processing nodes* in computing $reachable$

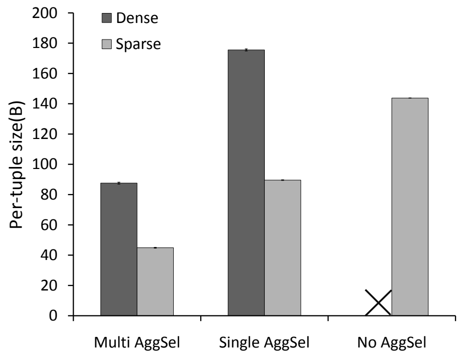
tions. Moreover, our concise representation of data provenance is far more efficient compared to relative provenance. Computing provenance incurs some overhead during insertions, and increased memory consumption, but the increase is offset by huge improvements in bandwidth and execution times when deletions are part of the workload. Most applications (both for declarative networking and sensor monitoring) include time-based expiration for state, and hence include many deletions.

- Our second technique, *lazy propagation* of derivations (Section 5) using the *MinShip* operator, reduces traffic when there are multiple possible derivations. Lazy propagation results in significant bandwidth savings. Given the dense network topology with 800 links and many alternative routes, lazy propagation resulted in 5-second running times, versus 5 minutes for eager propagation in the same network.

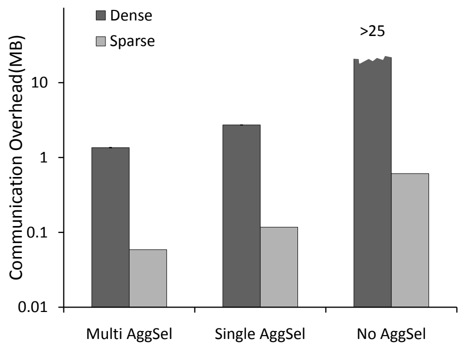
- Our third technique of multiple *aggregate selections* results in minimal propagation of tuples during query evaluation (Section 6). A dense network produces several alternative routes, and aggregate selections are especially effective in this setting, resulting in at least an order of magnitude reduction in bandwidth and execution times. While the benefits of aggregate selections have been explored previously in centralized settings, our main contribution here was the extension to a stream model, including support for deletions, and validating that similar benefits are observed in a distributed recursive stream query processor.

8 Related Work

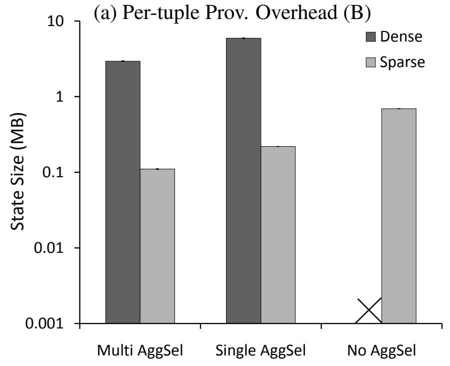
Stream query processing has been popular in the recent database literature, encompassing sensor network query systems [23, 13] as well as Internet-based distributed stream management systems [1, 9, 2]. To the best of our knowledge, none of these systems support recursive queries. Distributed recursive queries have been proposed as a mechanism for managing state in declarative networks. Our work formalizes aspects of soft-state management and significantly improves the ability to maintain recursive views. Our distributed recursive view maintenance techniques are applicable to other networked environments, particularly programming abstractions for region-based computations in sensor



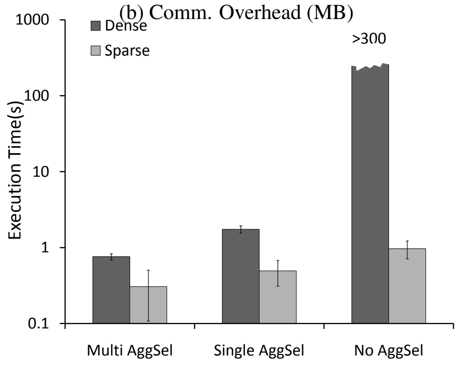
(a) Per-tuple Prov. Overhead (B)



(b) Comm. Overhead (MB)



(c) State within operators (MB)



(d) Convergence time (s)

Figure 14: Aggregate selections performance on $shortestPath$ and $cheapestCostPath$ query

networks [30, 29].

Provenance (also called lineage) has often been studied to help "explain" why a tuple exists [8] or to assign a ranking or score [6, 14]. Lineage was studied in [12] as a means of maintaining data warehouse data. Our absorption provenance model is a compact encoding of the PosBool provenance semiring in [15] (which provides a theoretical provenance framework, but does not consider implementability). We specialized it for maintenance of derived data in recursive settings. Our approach improves over the counting algorithm [17] which does not support recursion. We have experimentally demonstrated benefits versus DRed [17] and maintenance based on relative provenance [14].

9 Conclusions and Future Work

In this paper, we have proposed novel techniques for distributed recursive stream view maintenance. Our work is driven by emerging applications in declarative networking and sensor monitoring, where distributed recursive queries are increasingly important. We demonstrated that existing recursive query processing techniques such as DRed are not well-suited for the distributed environment. We then presented techniques that combine the use of absorption provenance to encode tuple derivability in a compact fashion, plus provenance-aware operators that are bandwidth efficient and avoid propagating unnecessary information, while maintaining correct answers.

Our work is proceeding along several fronts. Since our experimental results have demonstrated the effectiveness of techniques, we are working towards deploying our system in both the declarative networking and sensor network domains. We intend not only to support efficient distributed view maintenance, but also to utilize the provenance information to enforce decentralized trust policies, and perform real-time network diagnostics and forensic analysis. We also hope to explore opportunities for adaptive cost-based optimizations based on the query workload, network density, network connectivity, rate of network change, etc.

Acknowledgments

This work was funded by NSF grants CNS-0721845, CNS-0721541, IIS-0812270, IIS-0447972, and IIS-0713267, and a DARPA Computer Science Study Group grant. We thank the anonymous reviewers for their suggestions.

References

[1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2), August 2003.

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

[3] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2*, Feb. 2003.

[4] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 4(3), 1987.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, 1986.

[6] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[7] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[8] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[9] 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.

[10] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, 1994.

[11] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, New York, NY, USA, 2007.

[12] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[13] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The Cougar project: a work-in-progress report. *SIGMOD Record*, 32(3), 2003.

[14] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[15] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[16] GT-ITM. Modelling topology of large internetworks. http://www.cc.gatech.edu/projects/gtitm/.

[17] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[18] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Quering the Internet with PIER. In *VLDB*, 2003.

[19] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

[20] John Whaley. Javabdd library. http://javabdd.sourceforge.net.

[21] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *Proc. SIGMOD*, June 2006.

[22] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, 2005.

[23] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[24] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[25] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with Seaweed. In *VLDB*, 2006.

[26] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, pages 15–25, 1999.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, Nov. 2001.

[28] S. Sudarshan and R. Ramakrishnan. Aggregation and Relevance in Deductive Databases. In *Proceedings of VLDB Conference*, 1991.

[29] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, March 2004.

[30] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MASN*, 2004.

Algorithm 2 Pipelined hash join operator

$HalfPipeIns(u, h_u, p_u, h_j, p_j, \overline{uk})$

Inputs: Update u, build hash table h_u, build provenance table p_u, probe hash table h_j, probe tuple provenance table p_j, join keys $\overline{uk}$.
Output: Update stream u'.

```
 1: $oldPv := p_u[u.tuple]$
 2: **if** $h_u$ does not contain $u.tuple$ **then**
 3:     $h_u[u.tuple[\overline{uk}]] := u.tuple$
 4:     $p_u[u.tuple] := u.pv$
 5: **else**
 6:     $p_u[u.tuple] := p_u[u.tuple] \vee u.pv$
 7: **end if**
 8: **if** $oldPv \neq p_u[u.tuple]$ **then**
 9:     **for** each $t$ in $h_j[u.tuple[\overline{uk}]]$ **do**
10:         $u'.tuple := u.tuple \bowtie t$
11:         $u'.type$ := INS
12:         $u'.pv := u.pv \wedge p_j[t]$
13:         Output $u'$
14:     **end for**
15: **end if**
```

$HalfPipeDel(u, h_u, p_u, h_j, p_j, \overline{uk})$

Inputs: Update u, build hash table h_u, build provenance table p_u, probe hash table h_j, probe tuple provenance table p_j, join keys $\overline{uk}$.
Output: Update stream u'.

```
 1: $oldPv := p_u[u.tuple]$
 2: **if** $h_u$ contains $u.tuple$ **then**
 3:     $p_u[u.tuple] := p_u[u.tuple] \wedge \neg u.pv$
 4:     **if** $p_u[u.tuple] = 0$ **then**
 5:         Remove $u.tuple$ from $p_u$
 6:         Remove $u.tuple[\overline{uk}]$ from $h_u$
 7:     **end if**
 8: **end if**
 9: **if** $oldPv \neq p_u[u.tuple]$ **then**
10:     **for** each $t$ in $h_j[u.tuple[\overline{uk}]]$ **do**
11:         $u'.tuple := u.tuple \bowtie t$
12:         $u'.type$ := DEL
13:         $u'.pv := u.pv \wedge p_j[t]$;
14:         Output $u'$
15:     **end for**
16: **end if**
```

$Process(u)$

Inputs: Update u.

```
 1: **if** $u.type$ = DEL **then**
 2:     **if** $u$ is from $R^{\triangle}$ **then**
 3:         $HalfPipeDel(u, h_R, p_R, h_S, p_S, \overline{Rk})$
 4:     **else**
 5:         $HalfPipeDel(u, h_S, p_S, h_R, p_R, \overline{Sk})$
 6:     **end if**
 7: **else**
 8:     **if** $u$ is from $R^{\triangle}$ **then**
 9:         $HalfPipeIns(u, h_R, p_R, h_S, p_S, \overline{Rk})$
10:     **else**
11:         $HalfPipeIns(u, h_S, p_S, h_R, p_R, \overline{Sk})$
12:     **end if**
13: **end if**
```

$PipeHashJoin(R^{\triangle}, S^{\triangle}, \overline{Rk}, \overline{Sk}, W_R, W_S)$

Inputs: Update streams $R^{\triangle}, S^{\triangle}$, join keys $\overline{Rk}, \overline{Sk}$, window evaluation functions W_R, W_S.
Output: Update stream $RS^{\triangle}$.

```
 1: Init hash map $h_R : R(\bar{x})[\overline{Rk}] \rightarrow \{R(\bar{x})\}$
 2: Init hash map $h_S : S(\bar{y})[\overline{Sk}] \rightarrow \{S(\bar{y})\}$
 3: Init hash map $p_R : R(\bar{x}) \rightarrow \mathbf{P}(R(\bar{x}))$
 4: Init hash map $p_S : S(\bar{y}) \rightarrow \mathbf{P}(S(\bar{y}))$
 5: **while** not $EndOfStream(R^{\triangle})$ and not $EndOfStream(S^{\triangle})$ **do**
 6:     Read an update $u$ from $R^{\triangle}$ or $S^{\triangle}$
 7:     $Process(u)$
 8:     Let $expired_R$ := the results of calling $W_R(u, p_R)$
 9:     **for** $t$ in $expired_R$ **do**
10:         $t.type$ := DEL
11:         $Process(t)$
12:     **end for**
13: **end while**
```

Algorithm 3 MinShip operator

```
$BatchShipEager(P_{ins}, P_{del})$
Inputs: Provenance table: $P_{ins}$ and $P_{del}$
Ouput: Output stream $U'^{\Delta}$ sent on $S$.
 1: for each $t$ in $P_{ins}$ do
 2:     $u'.tuple := t$
 3:     $u'.type$ := INS
 4:     $u'.pv := P_{ins}[t]$
 5:     $sendMessage(S, u')$
 6: end for
 7: Reset $P_{ins}$
 8: for each $t$ in $P_{del}$ do
 9:     $u'.tuple := t$
10:     $u'.type$ := DEL
11:     $u'.pv := P_{del}[t]$
12:     $sendMessage(S, u')$
13: end for
14: Reset $P_{del}$

$BatchshipLazy(P_{ins}, P_{del})$
Inputs: Provenance table: $P_{ins}$ and $P_{del}$
Ouput: Output stream $U'^{\Delta}$ sent on $S$.
 1: for each $t$ in $P_{del}$ do
 2:     $u'.tuple := t$
 3:     $u'.type$ := DEL
 4:     $u'.pv := P_{del}[t]$
 5:     $sendMessage(S, u')$
 6:     if $t$ is in $P_{ins}$ then
 7:         $u'.tuple := t$
 8:         $u'.type$ := INS
 9:         $u'.pv := P_{ins}[t]$
10:         $sendMessage(S, u')$
11:         Remove $t$ from $P_{ins}$
12:     end if
13: end for
14: Reset $P_{del}$

$MinShip(U^{\Delta}, W, S)$
Inputs: Input stream $U^{\Delta}$, batched size $W$, socket $S$.
Output: Output stream $U'^{\Delta}$ sent on $S$.
 1: Init hash map $P_{ins} : U(\bar{x}) \rightarrow$ provenance expressions over $U(\bar{x})$
 2: Init hash map $P_{del} : U(\bar{x}) \rightarrow$ provenance expressions over $U(\bar{x})$
 3: Init hash map $B_{sent} : U(\bar{x}) \rightarrow$ provenance expressions over $U(\bar{x})$
 4: if there is a aggregate selection option then
 5:     Get the grouping key $\overline{uk}$, number of aggregate functions $n$ and aggregate functions $agg_1, \cdots, agg_n$
 6:     $U'^{\Delta} := AggSel(U^{\Delta}, \overline{uk}, n, agg_1, \cdots, agg_n)$
 7:     $U^{\Delta} := U'^{\Delta}$
 8: end if
 9: while not $EndOfStream(U^{\Delta})$ do
10:     Read an update $u$ from $U^{\Delta}$
11:     if $B_{sent}$ does not contain $u.tuple$ then
12:         $B_{sent}[u.tuple] := u.pv$
13:         $sendMessage(S, u)$
14:     else
15:         if $u.type$ = INS then
16:             if $B_{sent}[u.tuple] \vee u.pv \neq B_{sent}[u.tuple]$ then
17:                 $P_{ins}[u.tuple] := P_{ins}[u.tuple] \vee u.pv$
18:             end if
19:         else
20:             for each $t$ in $P_{ins}$ do
21:                 $P_{ins}[t] := restrict(P_{ins}[t], \neg u.pv)$
22:                 if $P_{ins}[t] = 0$ then
23:                     Remove $t$ from $P_{ins}$
24:                 end if
25:             end for
26:             $P_{del}[u.tuple] := P_{del}[u.tuple] \vee u.pv$
27:         end if
28:     end if
29:     if size of $P_{ins}$ + size of $P_{del}$ >= $W$ then
30:         Call $BatchShipEager(P_{ins}, P_{del})$ or
            $BatchShipLazy(P_{ins}, P_{del})$ based on ship mode
31:     end if
32: end while
33: Call $BatchShipEager(P_{ins}, P_{del})$ or $BatchShipLazy(P_{ins}, P_{del})$ based on ship mode
```

Algorithm 4 Aggregate selection sub-module

$AggSel(U^{\triangle}, \overline{uk}, n, agg_1, agg_2, \cdots, agg_n)$

Inputs: Input stream $U^{\triangle}$, grouping keys $\overline{uk}$, number of aggregate functions n, aggregate function $agg_1, agg_2, \cdots, agg_n$.
Output: Stream $U'^{\triangle}$.

```
 1: Init hash map $H$: $U(\bar{x})[\overline{uk}] \rightarrow \{U(\bar{x})\}$
 2: Init hash map $P$: $U(\bar{x}) \rightarrow$ provenance expressions over $U(\bar{x})$
 3: Init hash map $B$: $U(\bar{x})[\overline{uk}] \rightarrow [1..n] * \{U(\bar{x})\}$
 4: while not $EndOfStream(U^{\triangle})$ do
 5:     Read an update $u$ from $U^{\triangle}$
 6:     if $u.type$ = INS then
 7:         if $H$ does not contain $u.tuple$ then
 8:             $H[u.tuple[\overline{uk}]] := u.tuple$
 9:             Set $P[u.tuple]$ to the provenance of $u$
10:         else
11:             Add the provenance of $u$ to $P[u.tuple]$
12:         end if
13:         if $oldPv \neq P[u.tuple]$ then
14:             $changed := false$
15:             for $i = 1$ to $n$ do
16:                 if $B$ does not contain $u.tuple[\overline{uk}]$ then
17:                     $B[u.tuple[\overline{uk}]] := u.tuple$
18:                     $changed := true$
19:                 else if $u.tuple$ is better than $B[u.tuple[\overline{uk}]].i$ for $agg_i$ then
20:                     $u'.tuple := B[u.tuple[\overline{uk}]].i$
21:                     $u'.type$ := DEL
22:                     Set provenance of $u'$ to $P[B[u.tuple[\overline{uk}]].i]$
23:                     Output $u'$
24:                     $B[u.tuple[\overline{uk}]].i := u.tuple$
25:                     $changed := true$
26:                 end if
27:             end for
28:             if changed then Output $u$
29:         end if
30:     else if $H$ contains $u.tuple$ then
31:         $oldPv := P[u.tuple]$
32:         Remove the provenance of $u$ from $P[u.tuple]$
33:         if $P[u.tuple]$ indicates no derivability then
34:             Remove $u.tuple$ from $P$
35:             Remove $u.tuple[\overline{uk}]$ from $H$
36:         end if
37:         if $oldPv \neq P[u.tuple]$ then
38:             $changed := false$
39:             for $i = 1$ to $n$ do
40:                 if $B[u.tuple[\overline{uk}]].i = u.tuple$ then
41:                     $changed := true$
42:                     Remove $u.tuple$ from $B[u.tuple[\overline{uk}]].i$
43:                     for each tuple $t$ in $H[u.tuple[\overline{uk}]]$ do
44:                         if $B[u.tuple[\overline{uk}]].i = null$ or $t$ is better than $B[u.tuple[\overline{uk}]].i$ for $agg_i$ then
45:                             $B[u.tuple[\overline{uk}]].i := t$
46:                         end if
47:                     end for
48:                     $u'.tuple := B[u.tuple[\overline{uk}]].i$
49:                     $u'.type$ = INS
50:                     Set provenance of $u'$ to $P[B[u.tuple[\overline{uk}]].i]$
51:                     Output $u'$
52:                 end if
53:             end for
54:             if changed then Output $u$
55:         end if
56:     end if
57: end while
```