Maintaining Recursive Views 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

***Abstract*— The data management community has recently begun to consider declarative network routing and distributed acquisition: e.g., sensor networks that execute queries about contiguous regions, declarative networks that maintain shortest paths, and distributed and peer-to-peer stream systems that detect transitive relationships among data at the distributed sources. In each case, the fundamental operation is to maintain a *view* over dynamic network state. This view is typically distributed, recursive, and may contain aggregation, e.g., describing shortest paths or least costly paths.**

Surprisingly, solutions to computing such views are often domain-specific, expensive, and incomplete. We recast the problem as *incremental recursive view maintenance* given distributed streams of updates to tuples: new stream data becomes insert operations and tuple expirations become deletions. We develop techniques to maintain compact information about tuple derivability or *data provenance*. We complement this with techniques to reduce communication: aggregate selections to prune irrelevant aggregation tuples, provenance-aware operators that determine when tuples are no longer derivable and remove them from the view, and shipping operators that reduce the 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 communication overhead without sacrificing performance.

***Index Terms*— H.2.4.d. Distributed databases, H.2.4.h Query processing**

I. Introduction

As data management systems are handling increasingly distributed and dynamic data, the line between a network and a query processor is blurring. In a plethora of emerging applications, data originates at a variety of nodes and is frequently updated: routing tables in a peer-to-peer overlay network [1] or in a declarative networking system [2], [3], sensors embedded in an environment [4], [5], monitors within clusters at geographically distributed hosting sites [6], [7], data producers in large-scale distributed scientific data integration [8]. It is often natural to express distributed data acquisition, integration, and processing for these settings 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 quite different from the OLAP or OLTP queries that exemplify centralized DBMS query processing. We consider two main settings.

Declarative networking. In declarative networking [3], [9], an extended variant of datalog has been used to manage the state in routing tables — and thus to control how 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 [4], [5], primarily for aggregation queries. Outside the database community, a variety of macroprogramming languages [10], [11] have been proposed as alternatives, which include features like region and path computations. In the long run, we argue that the declarative query approach is superior because of data independence and optimization. However, the query languages and runtime systems must be extended to match the functionality of macroprogramming, particularly with respect to computing regions and paths.

Section II 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 properties that must be constantly refreshed). The majority of past work on recursive queries [12], [13] 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 [10], [11], [14], 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 [3] 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 computation* 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 [15], our approach emphasizes *minimizing the propagation of state* — both across the network (which is vital to reduce communication overhead) 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 propose a *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 heuristics to ensure that the absorption provenance structure, maintained in a Binary Decision Diagram (BDD), remains compact.
- We generalize *aggregate selection* to handle 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 simulated sensor data.

This paper extends [16] with a discussion and study of maintaining compact absorption provenance. Section II presents use cases for declarative recursive views. In Section III we discuss the distributed query processing settings we address. Sections IV through VII discuss our main contributions: absorption provenance, the *MinShip* operator, ensuring compact provenance, and our extended version of aggregate selection. Finally, we present experimental validation in Section VIII, describe related work in Section IX, and wrap up and discuss future work in Section X.

II. 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 [9]. 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 [1].

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.

```
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)
```

The techniques of this paper are agnostic as to the query language; we could express all queries in datalog, as in [9]. However, since SQL has a more familiar syntax, we present our examples using SQL-99's recursive query syntax[1]. 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, 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, in terms of the hop count (number of links) between the nodes:

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

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

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

This represents the composition of three views. The *path* recursive view is similar to the previous *reachable* query, with additional computation of the path length, as well as the path itself. The other (non-recursive) views *minHops* and *shortestPath* determine the length of the shortest path, and the set of paths with that length, respectively.

Network Highest-Bandwidth Path. We can similarly define the *highest bandwidth* path: instead of counting the number of links, we instead set a path's bandwidth to be the minimum bandwidth along any link; and then find, for any pair of endpoints, the path with maximum bandwidth.

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 (which we show and experimentally evaluate in Section VIII) starts with a series of reference nodes, and computes contiguous regions of triggered sensors near

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

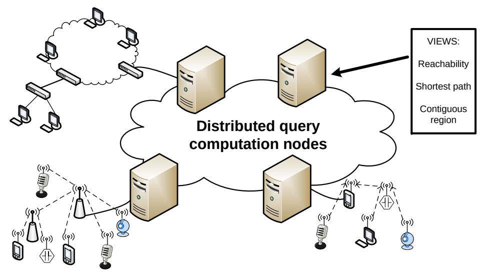


Fig. 1. Basic architecture: query processing nodes are placed in a number of sub-networks. Each collects state information about its sub-network, and the nodes share state to compute distributed recursive views such as shortest paths across the network.

these nodes. This is useful in sensor networks, e.g., in order to determine the average temperature of a fire.

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).

III. Execution Model and Approach

We consider techniques applicable to a 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 compute 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.

A. Query Execution Model

In networks, 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 insertions or deletions 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* [17] 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.

B. 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 direct correspondence between src value and location, although one could decouple each location from its physical encoding by using logical addresses (e.g., doing hash-based partitioning).

Now we define a materialized view $reachable(src, dst)$, which is also partitioned so tuples with source src are stored on node src. This query computes the transitive closure over the $link$ table, and was shown in the **Network Reachability** example of Section II. 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. Consider the execution plan 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

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

is shipped to the fixpoint via the *MinShip* (tuple shipping) operator, which in the simplest case simply sends data to a receiving node.

Computing the View Instance. 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.

The base-case contents of $reachable$ are computed directly from $link$, as specified in the first "branch" of the view definition (see **Network Reachability** query in Section II). 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 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 $A \rightarrow C$.

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. Observe that since we have a fully-connected network, the final resulting $reachable$ table at every node contains the set of all node pairs in the network with the first attribute matching the node's address.

Incremental Deletion (Standard Approach). Now consider the case when $link(C, B)$ expires (hence is deleted). Commonly used schemes for maintaining non-recursive views, such as counting tuple derivations, do not apply to this recursive view. Instead, one might employ the standard algorithm for recursive view maintenance, DRed [15]. DRed works by first *over-deleting* tuples conservatively and then *re-deriving* tuples that may have alternative derivations. Figure 5 shows the DRed over-deletion phase (steps 0-4), followed by the rederivation phase (steps 5-8). In the over-deletion phase, it first deletes $reachable(C, B)$ 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: 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 $link(C, B)$ is deleted. One source of deletions in network settings is tuple expirations; a large-scale network tends to be highly dynamic, so tuples will need to expire frequently, thus triggering frequent re-computation and exacerbating the overhead. Perhaps surprisingly, our example illustrates the *common case* behavior for network state queries: most networks are well-connected with bi-directional connectivity along several redundant paths. DRed will over-delete such paths, and then re-derive data.

We have ignored a further issue that DRed must wait until all deletions have been processed before it can start rederiving. This requires distributed synchronization, which may be expensive.

C. Our Approach

We now propose a solution that eliminates the need for recomputation, and that also avoids global synchronization. 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 it is easy to determine whether a view tuple should be removed when a base tuple is removed. (Section IV.)
- Propagates *provenance information* from one node to another only when necessary to ensure correctness — thus reducing network and computation costs. (Section V.)
- Seeks to minimize the encoding of provenance through reordering. (Section VI.)
- Propagates *tuples* through distributed aggregate computations only when necessary for correctness — also reducing network and computation costs. (Section VII.)

We describe these features in the next four sections, with the query plan of Figure 4 as the central example. We then evaluate our methods (Section VIII).

IV. 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 over-delete and re-derive (as with DRed), we instead propose to keep around metadata about derivations, i.e., *provenance* [18], [**?**], also called *lineage* [19].

Provenance alternatives. Different proposed forms of provenance capture different amounts of information. Lineage in [19] 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 [18], which encodes *sets of source tuples* that produced the answer; and the semiring polynomial provenance representation of [8], [20], whose implementation we term *relative provenance* here. In physical form, the latter encodes

reachable(src,dst)
(derivation step 1)

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

reachable(src,dst)
(derivation step 2)

tuple	at $\rightarrow$ 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 $\rightarrow$ A	$p_1 \wedge p_2 \wedge p_3$
$(\mathbf{A}, \mathbf{B})$	B $\rightarrow$ A	$p_1 \wedge p_2 \wedge p_4$
$*(\mathbf{B}, \mathbf{B})$	C $\rightarrow$ B	$p_1 \wedge p_2 \wedge p_3$
$(\mathbf{B}, \mathbf{C})$	C $\rightarrow$ B	$p_2 \wedge p_4$
$(\mathbf{C}, \mathbf{A})$	B $\rightarrow$ C	$p_2 \wedge p_3 \wedge p_4$
$(\mathbf{C}, \mathbf{B})$	B $\rightarrow$ C	$p_2 \wedge p_4$
$(\mathbf{C}, \mathbf{C})$	A $\rightarrow$ C	$p_1 \wedge p_2 \wedge p_3$

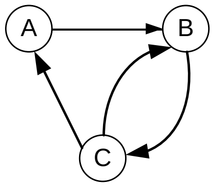


Fig. 3. Network represented in **link** relation

reachable(src,dst)
(derivation step 3)

tuple	at $\rightarrow$ 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 $\rightarrow$ A	$p_1 \wedge p_2 \wedge p_3$
$*(\mathbf{B}, \mathbf{C})$	C $\rightarrow$ B	$p_1 \wedge p_2 \wedge p_3$
$(\mathbf{C}, \mathbf{A})$	A $\rightarrow$ C	$p_1 \wedge p_2 \wedge p_3$
$*(\mathbf{C}, \mathbf{B})$	A $\rightarrow$ C	$p_1 \wedge p_2 \wedge p_4$

reachable(src,dst)
(derivation step 4)

tuple	at $\rightarrow$ 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	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$

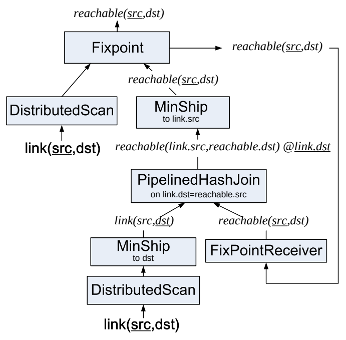


Fig. 4. Plan for $reachable$ query. Underlined attributes are the ones upon which data is partitioned.

Fig. 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 IV). A tuple marked "*" is an extra derivation only shipped in the absorption provenance model.

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
(C, C)	C
$-(\mathbf{C}, \mathbf{B})$	C

reachable(src,dst)
(step 1)

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

reachable(src,dst)
(step 2)

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

reachable(src,dst)
(step 3)

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

reachable(src,dst)
(step 4)

tuple	at $\rightarrow$ to

reachable(src,dst)
(step 5)

tuple	at $\rightarrow$ to
(A, B)	A
(B, C)	B
(C, A)	C

reachable(src,dst)
(step 6)

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

reachable(src,dst)
(step 7)

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

reachable(src,dst)
(step 8)

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

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

a derivation graph capturing which tuples are created as *immediate consequents* of others. The graph can be traversed after a deletion to determine whether a tuple is still derivable from base data [8]. 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 relies on the system of equations (encoded as edges in a graph) to resolve the problem of infinite derivations, which 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 the view tuples' derivations for 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 opera-

Algorithm 1 Fixpoint operator

```
$Fixpoint(B^{\Delta}, R^{\Delta})$
Inputs: Input base stream $B^{\Delta}$, recursive stream $R^{\Delta}$
Output: Output stream $U'^{\Delta}$
 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:     $B'^{\Delta} := AggSel(B^{\Delta}, \overline{uk}, n, agg_1, \cdots, agg_n)$
 5:     $B^{\Delta} := B'^{\Delta}$
 6:     $R'^{\Delta} := AggSel(R^{\Delta}, \overline{uk}, n, agg_1, \cdots, agg_n)$
 7:     $R^{\Delta} := R'^{\Delta}$
 8: end if
 9: while not $EndOfStream(B^{\Delta})$ and not $EndOfStream(R^{\Delta})$ do
10:     Read an update $u$ from $B^{\Delta}$ or $R^{\Delta}$
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 $B^{\Delta}$ then
28:         for each $t$ in $P$ do
29:             $oldPv := P[t]$
30:             $P[t] = restrict(P[t], \neg u.pv)$
31:             if $P[t]$ indicates no derivability then
32:                 Remove $t$ from $P$
33:                 Remove $t$ from the view
34:             end if
35:         end for
36:     end if
37: end while
```

$\sigma_\theta(R)$:	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 $\mathbf{P}(t_1) \vee \mathbf{P}(t_2) \vee \cdots \vee \mathbf{P}(t_n)$

Fig. 6. Relational algebra rules for composition of provenance expressions. Note that recursive fixpoint incorporates union.

tors 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 [20]).

Our key innovation with respect to provenance is to develop a physical representation in which we can exploit *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*. 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 benefit of a compact provenance annotation is reduced network traffic. Even better, we can use absorption provenance to help maintain a view after a base tuple has been deleted: we assign the value **false** to the provenance variable for each deleted base tuple, then substitute this value into 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 annotated p_1, p_2, p_3, and p_4; we see that an additional 4 tuples (beyond the previous set-oriented execution model) are 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 that cannot be absorbed, 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. (In the worst case it is still possible that deletions may need to be propagated to all nodes in the network.)

A. Implementing Absorption Provenance

There are multiple alternatives when attempting to encode an absorption provenance 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* [21] (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 [22]. Such libraries 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 VII.

B. 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 instead use pipelined semi-naïve evaluation [9], where tuples are handled in the order in which they arrive via the network (assuming a FIFO channel), and are only combined with tuples that arrived 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 now contains three fields: $type$, which indicates whether it is an INS or DEL tuple; $tuple$, which records its raw tuple values; and pv, which stores its annotated 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 VII. Now, upon receipt of an insertion operation u (Lines 11–26), the fixpoint operator first determines whether the tuple has already been encountered (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 directly caused by deletions on the base stream. Hence, we only need to focus on deletion tuples generated from the base (B^{Δ}) stream. When we receive 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" [22] 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.

C. Join Operator

The $PipelinedHashJoin$ must not only maintain two hash tables for its input relations (as is the norm), but also a hash table from each tuple to its current absorption provenance. It maintains this provenance state in a manner similar to the $Fixpoint$; due to space constraints we refer the reader to the extended technical report [23] for pseudocode. As insertions are received, provenance is updated for the associated tuple. The *difference* between the tuple's existing and new provenance is computed; then the tuple is added to the appropriate hash table (if it does not already exist), and probed against the opposite relation. Deletion happens similarly, except that a tuple is removed from the join hash table *only if* its provenance becomes **false** (i.e., it is no longer derivable).

V. 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. Even 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. Our goal is to reduce the number of *derivations* (provenance annotations) 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 that deletion operation. 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 a 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.)

$MinShip$'s internal state management again resembles that of the $Fixpoint$ operator. Pseudocode is given in [23].

VI. Producing Compact Provenance BDDs

As described previously, our approach to encoding and maintaining absorption provenance relies on their compact representation in ordered BDDs. In fact, the compactness of a BDD depends heavily on the order of its construction. Each BDD is a DAG with two terminals, representing 0 and 1. Every internal node in the BDD represents a variable, and every variable appears at a certain level in the DAG, according to a pre-defined ordering of variables. Every internal node has two outgoing edges: one representing the associated variable being assigned **true**, and the other representing **false**. Isomorphic subgraphs in the DAG are merged. All paths leading to the "1" terminal node represent possible truth assignments for the Boolean expression. Some variable orderings lead to different shared subgraphs or to elimination of certain nodes.

Recall that in our setting, the provenance tokens associated with tuples are converted into BDD variables. We begin by reviewing the BDD variable ordering problem. Then we consider heuristics for ordering the variables as tuple updates arrive during stream processing.

A. BDD Variable Ordering Problem

Variable ordering has been heavily studied in the BDD literature. Unfortunately, even determining the optimal order of a single BDD is an NP-hard problem [24]. Thus, one must rely on heuristics. Two common heuristics used in practice are *variable-swap* and *sifting* [24]. Variable-swap, as its name implies, seeks to minimize BDD size by trying different swaps of adjacent variables. It is inexpensive but gets trapped in local minima. An extension called sifting searches for a good position for each variable in the order. This is significantly more expensive, but finds better solutions.

Many other techniques have been proposed, including using simulated annealing [25], genetic algorithms [26], or machine learning [27], [28] to guide the search. However, all of these approaches assume a setting in which the set of variables and the set of Boolean expressions is known apriori. In probabilistic databases, Olteanu and Huang recently considered the BDD ordering problem for a certain subclass of queries in [29], but their class is very different from our transitive closure queries.

Our problem does not fall under the standard setting: we are given the task of incrementally computing and maintaining a set of BDD annotations to a set of tuples in a streaming transitive closure computation. We are limited in our knowledge of the Boolean expressions to be merged, as the expressions are formed through evaluating transitive closure queries. Our problem is to incrementally re-order BDD variables (corresponding to provenance tokens) every time we receive and process changes to network link data.

B. Motivation for Depth-first Traversal Heuristic

Our approach will be to order BDD variables according to depth-first traversal order of the network. To explain the intuition for why, Figure 7 shows an example network with six nodes and eight links (which are unidirectional for simplicity). If we form the BDD for the connections between start and end, $reachable(A, F)$, following the rules of Figure 6, then we get a provenance expression: $p_0(p_1+p_2p_3)+(p_4+p_6p_7)p_5$, where for every $0 \le i \le 7$, p_i is the provenance token for e_i.

A depth-first traversal of the graph might visit the nodes in the order $e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7$. Figure 8(a) shows that this BDD is quite compact, with 9 nodes. A breadth-first traversal might be $e_0, e_4, e_6, e_1, e_2, e_5, e_7, e_3$. Here, we get an 18-node BDD, shown in Figure 8(b). Note this has twice as many nodes as the previous example.

Intuitively, the BDD is most effective at merging Boolean terms that share initial variables (i.e., segments close to the start node), and a depth-first traversal computes paths in a way that maximizes sharing of these initial variables (segments).

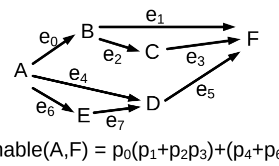
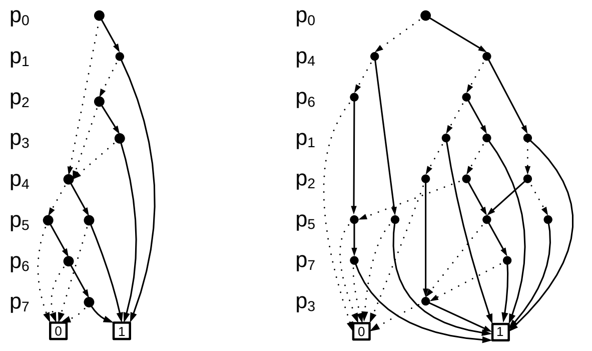


Fig. 7. An example network between nodes A and F



(a) Depth-first traversal BDD (b) Breadth-first traversal BDD

Fig. 8. Different variable orders representing the provenance for $reachable(A, F)$ in Figure 7

C. Incremental depth-first labeling algorithm

The previous section gave a rationale for our basic heuristic of extending a BDD in depth-first traversal order. We now describe how to incrementally maintain, as the network graph is being traversed, a global ordering on all variables, such that each BDD will be generated in a fashion that follows a depth-first ordering on the variables. In our initial implementation, this variable ordering process requires global coordination, either through a single central server or through state replication on all nodes. We assume that as new base tuples ($link$ for the $reachable$ query) are incrementally received by the system, they are fed into a variable reordering algorithm (Algorithm 2 shows the insertion portion; due to space constraints we omit the deletion processing steps but sketch them below). This algorithm incrementally maintains an $edges$ vector that establishes an ordering on the network edges ($link$ tuples) that conforms to a depth-first traversal.

Given the current state of this vector, we can assign each edge $edges[i]$ in the vector to the variable v_i in the BDD, located at depth i. Now, when a union sub-operation (within a fixpoint or aggregate) or join operation occurs, the BDDs associated with the input tuples will conform to the same variable ordering and will combine in (ideally) a compact fashion. As the $edges$ vector gets updated, we may need to do fairly inexpensive variable swapping within each BDD (a functionality already supported).

The intuition of the algorithm is that we maintain the $edges$ vector in a way that exactly describes a depth-first traversal of the graph (this includes all cyclic edges). Suppose we have two nodes n and n', which are siblings in terms of the DFS traversal. By definition, we will traverse all edges reachable from node n before those reachable from n'. Hence, if the first edge originating from n is recorded at position $startInx[n]$ in the $edges$ vector, then *all* of the edges reachable from n will

Algorithm 2 Incremental depth-first search algorithm with interval labeling

```
$IncrementalDepthFirst(e, startInx, endInx, edges)$
Input: Incoming edge $e$, map from variable to start edge position $startInx$, map from variable to end edge position $endInx$, edge vector $edges$.
Output: Updated $startInx, endInx$, and $edges$.
1: $x := e.start; y := e.end$;
2: if $startInx[x] < 0$ then {no outgoing edge from $x$}
3:     if $endInx[x] < 0$ then {no incoming edge to $x$}
4:         if $startInx[y] < 0$ then {no outgoing edge from $y$}
5:             Insert $e$ to the end of $edges$;
6:             Update $startInx[x], endInx[x], endInx[y]$;
7:         else {there exists an outgoing edge from $y$}
8:             if the $startInx[y] - 1$ edge of $edges$ ends in $y$ then {there exists an incoming edge to $y$}
9:                 Insert $e$ to the end of $edges$;
10:                Update $startInx[x], endInx[x]$;
11:            else {no incoming edge to $y$}
12:                Insert $e$ before position $startInx[y]$ of $edges$;
13:                Update labels for $x$, $y$ and all labels with value larger than $startInx[y]$;
14:            end if
15:        end if
16:    else {there exists an incoming edge to $x$}
17:        Insert $e$ after position $endInx[x]$ of $edges$;
18:        Update labels for $x$, $y$ and all labels with value larger than $endInx[x]$
19:        if $endInx[x] < startInx[y]$ then {$x$'s interval appears before $y$'s interval and do not overlap}
20:            Move sub-vector $edges[startInx[y]..endInx[y]]$ forward to position $endInx[x] + 1$ of $edges$ and shift other elements;
21:            Update all labels according to the new positions in $edges$;
22:        end if
23:    end if
24: else {there exists an outgoing edge from $x$}
25:    Same procedure as lines 17-22 but do not modify $startInx[x]$;
26: end if
```

appear in $edges$ before the index position $startInx[n']$. We record the vector position of the *last* edge reachable from n as $endInx[n]$. At initialization, we set all elements of $startInx$ and $endInx$ to the null indicator -1.

Now, to incrementally maintain $edges$ and the index positions, we must consider the different scenarios for how some new edge (x, y) may relate to existing paths in the graph. Figure 9 illustrates these cases. In case (a) (lines 5-6), x is a new node and y has no outgoing edges. We append (x, y) to vector $edges$. For case (b) (lines 8-14), x is a new node and y has outgoing edge(s). If there exists an incoming edge to y, we append (x, y) to the end of $edges$; otherwise, we insert (x, y) into $edges$ before position $startInx[y]$. Case (c) (Lines 17-22) is where x has incoming edges and y is now traversed earlier. We insert (x, y) into $edges$ after position $endInx[x]$ and set this index to $startInx[x]$. If $endInx[x] < startInx[y]$, then we move edges in the range $[startInx[y], endInx[y]]$ directly after $endInx[x]$. Finally, case (d) (line 25) is when x already has outgoing edge(s). This is similar to case (c), except we do not modify $startInx[x]$. Note that we choose to insert the new edge to the end of vector $edges$ in case (a) and (b), since it does not affect the previous ordering and is the most cost-efficient way to maintain the vector. Incremental insertion requires $O(|edges|)$ operations.

Deletion follows similar principles, but is somewhat more complex. If an edge is removed, this may "orphan" a portion of the original DFS traversal subgraph. We must now scan forward in the $edges$ vector to find the *first* edge that references any node n in this orphaned subgraph. Immediately after this edge connecting to n, we insert the edges (in DFS order) reachable from n. We repeat the process for any remaining nodes from the orphan subgraph, and drop any edges that are

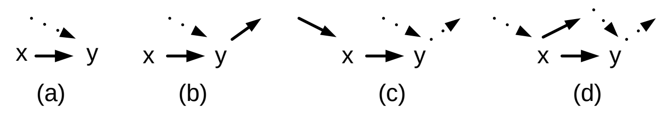


Fig. 9. Four different cases when edge(x,y) is added. A solid arc represents at least one edge and a dashed arc represents zero or more edges

no longer connected. If we use hash sets to match nodes in the orphaned subgraph, deletion can be done in $O(|edges|)$ operations.

VII. Minimizing Propagation of State

Our third challenge is to minimize the amount of state (in terms of unique tuples, not just alternate derivations of the same tuple) that gets propagated from one node to the next. Given that aggregation is commonplace in network-based queries (as in most queries of Section II), we need a way to also suppress tuples that have no bearing on the output aggregate values. We adapt a technique called *aggregate selection* [30] to a streaming model, with a *windowed* aggregation (group-by) operation [31]. We consider MIN, MAX, COUNT, and SUM functions[3]. In essence, the aggregate computation is split between a partial-aggregate operation that is used internally by stateful operators like the $Fixpoint$ and $MinShip$ to prune irrelevant state, and a final aggregation computation is done at the end over the partial aggregates' outputs. 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.

Our aggregate selection ($AggSel$ for short) module (Algorithm 3) can be embedded within any operator that creates and ships state. (In our system, both $Fixpoint$ and $MinShip$ have calls to this module.) The module takes as input a stream U^{Δ}, 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 hash table 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 hash table P maps from each tuple to their absorption provenance. Another hash table 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'^{Δ} of the update tuples.

Each time $AggSel$ receives a stream insertion (Lines 6–25), it inserts this tuple into the internal map H from group-by key $\overline{uk}$ to source tuple set. (If a tuple with the same value already exists in the set, then it simply updates the provenance P for the tuple.) Next, if the insertion affects the result of any aggregate attribute associated with $\overline{uk}$ — it changes the MIN or MAX value, or it revises the COUNT or SUM — the aggregation selection module will then propagate a *deletion* operation on the old aggregate value. After checking all the aggregate functions, if at least one of the aggregate values is affected, then it propagates this input insertion tuple as an *insertion*; if none of them is affected, it propagates nothing (see the loop starting at Line 12). Meanwhile, the module applies the change to its internal state.

Upon encountering a stream deletion or an expiration (Lines 25–49), $AggSel$ checks whether the deletion has any

[3]AVERAGE can be derived from SUM and COUNT, as in [32].

Algorithm 3 Aggregate selection sub-module

```
$AggSel(U^{\Delta}, \overline{uk}, n, agg_1, agg_2, \cdots, agg_n)$
Inputs: Input stream $U^{\Delta}$, grouping keys $\overline{uk}$, number of aggregate functions $n$, aggregate function $agg_1, agg_2, \cdots, agg_n$.
Output: Stream $U'^{\Delta}$.
1: Init hash map $H$: $U(\bar{x})[\overline{uk}] \to \{U(\bar{x})\}$
2: Init hash map $P$: $U(\bar{x}) \to$ provenance expressions over $U(\bar{x})$
3: Init hash map $B$: $U(\bar{x})[\overline{uk}] \to [1..n] * \{U(\bar{x})\}$
4: while not $EndOfStream(U^{\Delta})$ do
5:     Read an update $u$ from $U^{\Delta}$
6:     if $u.type$ = INS then
7:         if $H$ does not contain $u.tuple$ then
8:             $H[u.tuple[\overline{uk}]] := u.tuple$
9:         end if
10:        $P[u.tuple] := u.pv$
11:        if $oldPv \neq P[u.tuple]$ then
12:            for $i = 1$ to $n$ do
13:                if $B$ does not contain $u.tuple[\overline{uk}]$ then
14:                    $B[u.tuple[\overline{uk}]].i := u.tuple$
15:                else if $u.tuple$ is better than $B[u.tuple[\overline{uk}]].i$ for $agg_i$ then
16:                    $u'.tuple := B[u.tuple[\overline{uk}]].i$
17:                    $u'.type$ := DEL
18:                    $u'.pv = P[B[u.tuple[\overline{uk}]].i]$
19:                    Output $u'$
20:                    $B[u.tuple[\overline{uk}]].i := u.tuple$
21:                end if
22:            end for
23:            if $B[u.tuple[\overline{uk}]]$ is updated then Output $u$
24:        end if
25:    else if $H$ contains $u.tuple$ then
26:        $oldPv := P[u.tuple]$
27:        Remove $u.pv$ from $P[u.tuple]$
28:        if $P[u.tuple]$ indicates no derivability then
29:            Remove $u.tuple$ from $P$
30:            Remove $u.tuple[\overline{uk}]$ from $H$
31:        end if
32:        if $oldPv \neq P[u.tuple]$ then
33:            for $i = 1$ to $n$ do
34:                if $B[u.tuple[\overline{uk}]].i = u.tuple$ then
35:                    Remove $u.tuple$ from $B[u.tuple[\overline{uk}]].i$
36:                    for each tuple $t$ in $H[u.tuple[\overline{uk}]]$ do
37:                        if $B[u.tuple[\overline{uk}]].i = null$ or $t$ is better than $B[u.tuple[\overline{uk}]].i$ for $agg_i$ then
38:                            $B[u.tuple[\overline{uk}]].i := t$
39:                        end if
40:                    end for
41:                    $u'.tuple := B[u.tuple[\overline{uk}]].i$
42:                    $u'.type$ = INS
43:                    $u'.pv = P[B[u.tuple[\overline{uk}]].i]$
44:                    Output $u'$
45:                end if
46:            end for
47:            if $B[u.tuple[\overline{uk}]]$ is updated then Output $u$
48:        end if
49:    end if
50: end while
```

affect on the derivability of the deleted tuple (Lines 26–28), and then whether any aggregate value associated with the group-by key $\overline{uk}$ is affected. If an aggregate value is modified (i.e., this deletion tuple at least partly determines the aggregate value), then $AggSel$ traverses through the current version of buffered tuple table, computes the updated aggregate value, and propagates an *insertion* of the tuple with the new aggregate value. If any of the aggregate values is affected, then it propagates a *deletion*. Meanwhile, the module applies the change to its internal state.

VIII. Experimental Evaluation

We have developed a Java-based distributed query processor (see [**?**]) that implements all operators as described in Sections IV-VII. Our implementation utilizes the FreePastry 2.0_03 [33] DHT for data distribution, and JavaBDD v1.0b2 [22] as the BDD library for absorption provenance maintenance. 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.

A. Experimental Setup

We studied two query workloads taken from our use cases:

Workload 1: Declarative networks. Our query workloads consist of the *reachable query* and the *shortest-path* query (Section II). As input to these queries, we use simulated Internet topologies generated by GT-ITM [34], a package that is widely used for this purpose. 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 our case). 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, 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 "activeRegion" 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. Based on that, we can compute the the largest such active region.

```
with recursive activeRegion(regionid,sensorid) as
  ( select M.regionid, S.sensorid
    from sensor S, coordSensor 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)
```

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 processed by each distributed node for executing a distributed query to completion.
- **Per-node state within operators (MB):** the total state overhead maintained inside operators on each distributed node.
- **Convergence time (s):** the time taken for a distributed query to finish execution on all distributed nodes.

B. 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* [8] where each tuple is annotated with information describing derivation "edges" from 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 10 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 11 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 [8]), 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 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 computed over a different topology from the $reachable$ case, and it exhibits slightly different update characteristics. Still, as we see in Figure 12, 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 are within closer proximity.) Under deletion workloads, the trends shown by the $region$ query also closely mirror that of the reachable query and those graphs are shown in [23]. Since the queries exhibit similar performance, we focus on the *reachable* query for our remaining experiments.

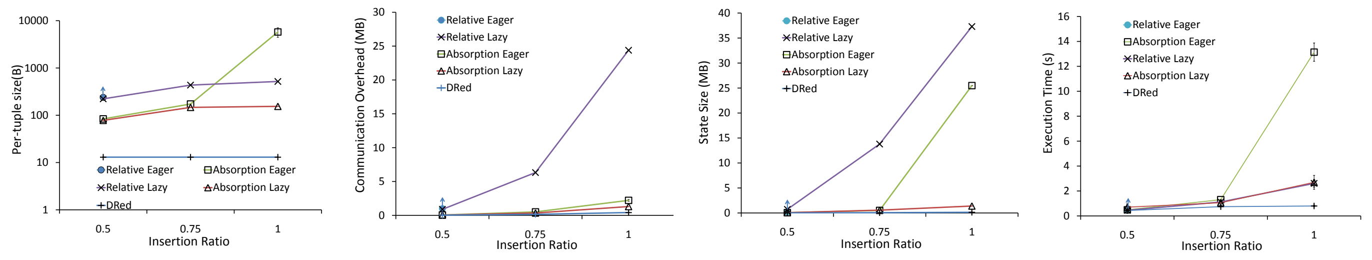
C. Scalability

Next we consider how our absorption provenance schemes scale, 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) on average, whereas the *sparse* setting has two. Figure 13 shows the insertion-only workload.[4] The dense network has far more derivations than the sparse network: here, *Eager Dense* did not complete after 5 minutes on a 800-link network, whereas *Lazy Dense* finished in under 5 seconds.

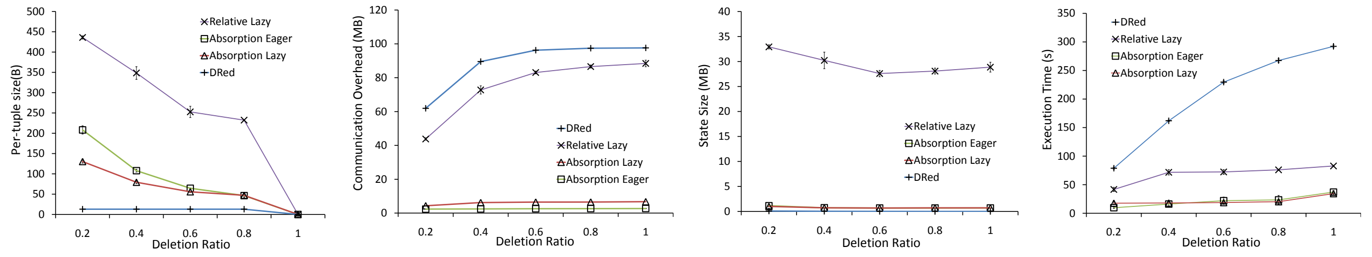
Increasing Query Processing Nodes. Next, we increase the number of query processing nodes, while keeping the input dataset constant. Figure 14 shows the results. Per-tuple provenance overhead increases, then eventually levels off, as the number of nodes increases: each node now processes fewer tuples, and the opportunities to absorb or buffer are reduced. More 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 lower-bandwidth connection between our

[4]We further experimented with deleting an additional 20% of the links. Observations were similar and we omit graphs due to space constraints.



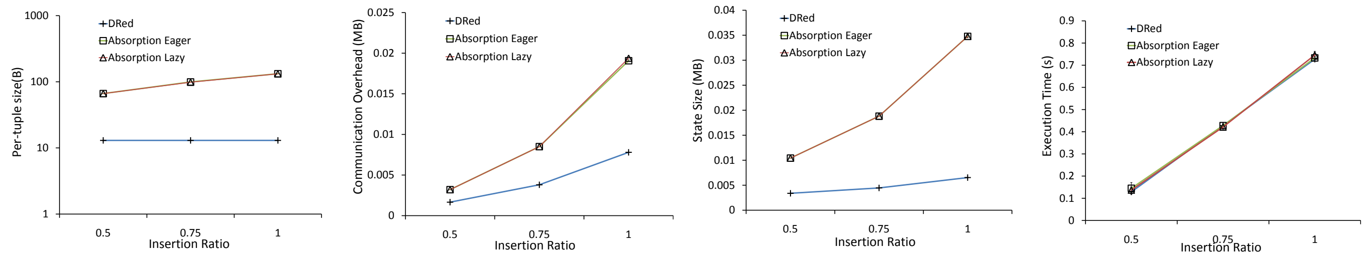
(a) Per-tuple Prov. Overhead (B) (b) Comm. Overhead (MB) (c) State within operators (MB) (d) Convergence time (s)

Fig. 10. *reachable* query computation as **insertions** are performed



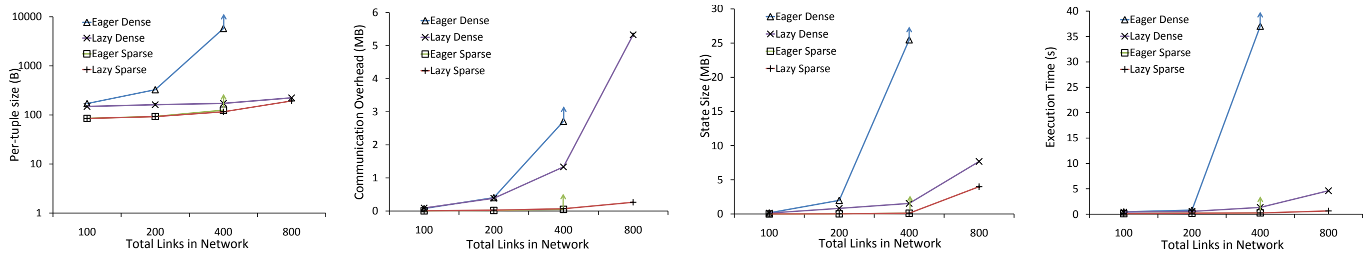
(a) Per-tuple Prov. Overhead (B) (b) Comm. Overhead (MB) (c) State within operators (MB) (d) Convergence time (s)

Fig. 11. *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)

Fig. 12. *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)

Fig. 13. Increasing the number of links (and nodes) for the *reachable* query over inserts

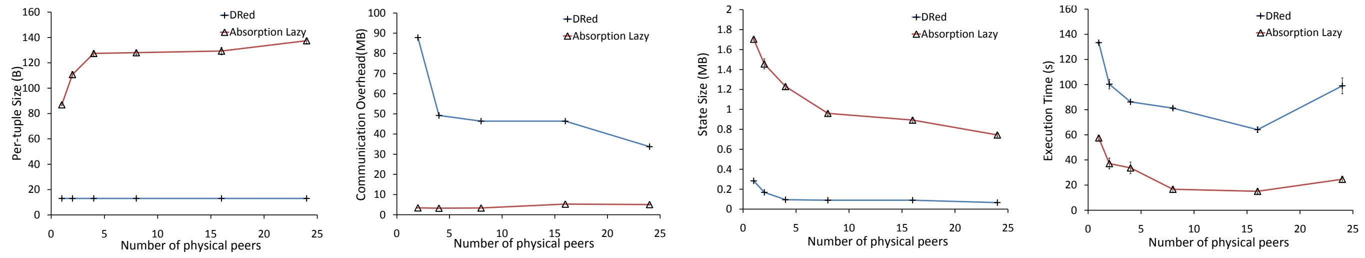
two subnets. In all cases, DRed incurs higher communication overhead and takes longer to complete than our approach.

D. Provenance BDD Ordering Heuristic

In Figure 15, we compare the performance using our depth-first traversal heuristic, versus naive merging and ordering of BDDs based on the order of tuple arrival. To better study the performance, we randomize edge arrivals for this experiment. From the figure, the depth-first traversal heuristic saves up to 50% of the provenance overhead. This also results in lower communication overhead and memory footprint. Additionally, execution time remains essentially the same, because the variable reordering process is relatively lightweight and does not affect query processing dataflow. The results show that the execution overhead of applying this heuristics can be offset by the performance gain in memory and communication.

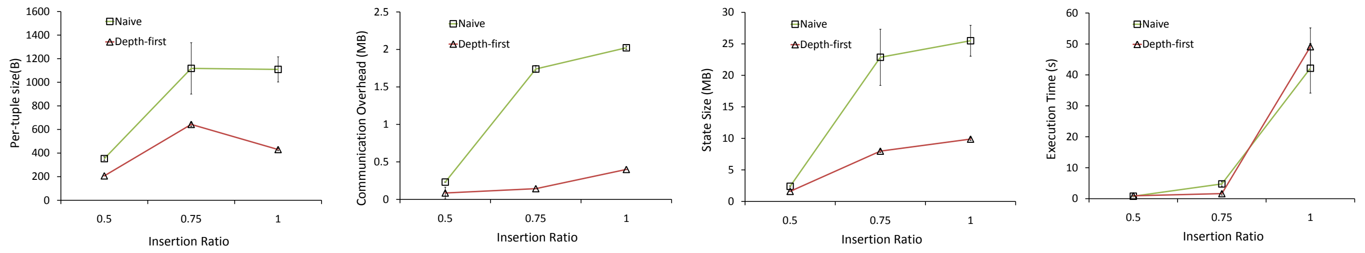
E. Multi-aggregate Selection

Figure 16 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 II: *Multi AggSel* computes two aggregates (one for shortest path and the other for cheapest cost path); *Single AggSel* minimizes only based on the cheapest cost path. 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



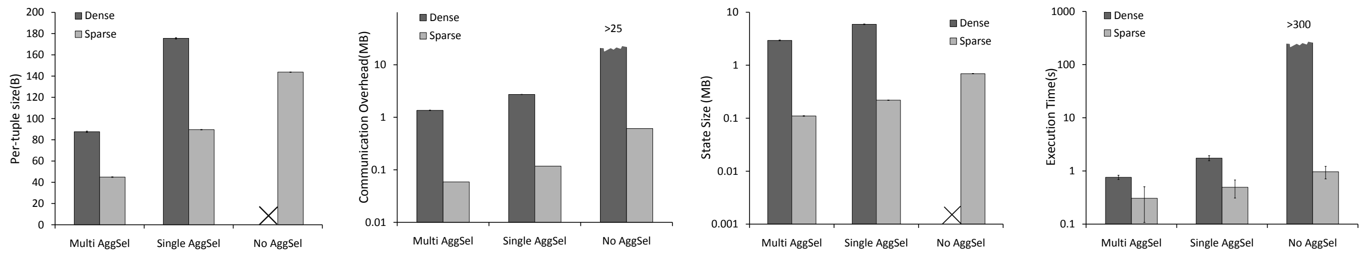
(a) Per-tuple Prov. Overhead (B) (b) Comm. Overhead (MB) (c) State within operators (MB) (d) Convergence time (s)

Fig. 14. Varying the number of *physical query processing nodes* in computing $reachable$ query



(a) Per-tuple Prov. Overhead (B) (b) Comm. Overhead (MB) (c) State within operators (MB) (d) Convergence time (s)

Fig. 15. Depth-first search order vs. naïve random order on $reachable$ query



(a) Per-tuple Prov. Overhead (B) (b) Comm. Overhead (MB) (c) State within operators (MB) (d) Convergence time (s)

Fig. 16. Aggregate selections performance on $shortestPath$ and $cheapestCostPath$ query

queries are prohibitively expensive, and do not complete within 5 minutes for dense topologies.

F. Summary of Results

We summarize our experimental results with reference to the contributions of this paper as outlined in Section III-C.

- *Absorption provenance* (Section IV) incurs some overhead during insertions and consumes increased memory, vs. traditional schemes such as DRed. That increase is offset by huge improvements in communication overhead and execution times when deletions are part of the workload. Moreover, our concise representation of data provenance is far more efficient than an encoding of relative provenance. Most network applications include time-based expiration for state, and hence require frequent deletion processing.
- *Lazy propagation* of derivations (Section V) reduces traffic when there are multiple possible derivations. Lazy propagation results in significant communication cost savings. Given a dense network topology, lazy propagation sped computation by more than an order of magnitude.
- Our heuristic of reordering variables according to a depth-first traversal (Section VI) results in up to 50% space and communications savings, with minimal impact on query performance.
- Multiple *aggregate selections* significantly reduce the propagation of tuples during query evaluation (Section VII). This is especially true in a dense network with alternative routes, resulting in at least an order of magnitude reduction in communication cost 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.

IX. Related Work

Stream query processing has been popular in the recent database literature, encompassing sensor network query systems [4], [5] as well as Internet-based distributed stream management systems [35], [36], [37]. 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 networks [10], [11].

Provenance (also called lineage) has often been studied to help “explain” why a tuple exists [18] or to assign a ranking or score [8], [38]. Lineage was studied in [19] as a means of maintaining data warehouse data. Our absorption provenance model is a compact encoding of the PosBool provenance semiring in [20] (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 [15] which does not support recursion. We have experimentally demonstrated benefits versus DRed [15] and maintenance based on relative provenance [8] (both of which were developed for non-distributed query settings).

The problem of BDD minimization has been well-studied and we discuss the related literature in Section VI-A.

X. Conclusions and Future Work

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 [15] are not well-suited for the distributed environment. We then showed how absorption provenance could be used to encode tuple derivability in a compact fashion, then incorporated into 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 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 in part by NSF grants CNS-0721845, CNS-0721541, IIS-0812270, IIS-0447972, and IIS-0713267, and DARPA HRO1107-1-0029.

References

[1] 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.

[2] 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.

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

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

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

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

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

[8] 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.

[9] 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.

[10] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *NSDI*, March 2004.

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

[12] I. Balbin and K. Ramamohanarao, "A generalization of the differential approach to recursive query evaluation," *J. Log. Program.*, vol. 4, no. 3, 1987.

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

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

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *SIGMOD*, 1993.

[16] M. Liu, W. Zhao, N. Taylor, Z. Ives, and B. T. Loo, "Maintaining recursive stream views with provenance," in *ICDE*, 2009.

[17] 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*, 1999, pp. 15–25.

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

[19] Y. Cui, "Lineage tracing in data warehouses," Ph.D. dissertation, Stanford University, 2001.

[20] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *PODS*, 2007.

[21] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[22] John Whaley, "JavaBDD library," http://javabdd.sourceforge.net.

[23] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo, "Recursive computation of regions and connectivity in networks," University of Pennsylvania, Tech. Rep. MS-CIS-08-32, 2008.

[24] T. T. Christoph Meinel, "Algorithms and data structures in vlsi design," *Springer-Verlag*, pp. 1–267, August 1998.

[25] B. Bollig, M. Lobbing, and I. Wegener, "Simulated annealing to improve variable orderings for OBDDs," in *International Workshop on Logic Synthesis*, 1995.

[26] R. Drechsler and N. G. amd Bernd Becker, "Learning heuristics for OBDD minimization by evolutionary algorithms," *Parallel Problem Solving from Nature – PPSN IV*, 1996.

[27] O. Grumberg, S. Livne, and S. Markovitch, "Learning to order BDD variables in verification," *Journal of AI Research*, vol. 18, 2003.

[28] M. Carbin, "Learning effective BDD variable orders for BDD-based program analysis," Stanford University, Tech. Rep., May 2006, honors thesis.

[29] D. Olteanu and J. Huang, "Using OBDDs for efficient query evaluation on probabilistic databases," in *Scalable Uncertainty Management*, 2008.

[30] S. Sudarshan and R. Ramakrishnan, "Aggregation and Relevance in Deductive Databases," in *VLDB*, 1991.

[31] 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.

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

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

[34] GT-ITM, "Modelling topology of large networks," http://www.cc.gatech.edu/projects/gtitm/.

[35] 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.*, vol. 12(2), August 2003.

[36] 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.

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

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