Grouping Distributed Stream Query Services by Operator Similarity and Network Locality

Sangeetha Seshadri*, Bhuvan Bamba*, Brian F. Cooper†, Vibhore Kumar*,
Ling Liu*, Karsten Schwan*, Gong Zhang*
*Georgia Institute of Technology †Yahoo! Research
{sangeeta,vibhore,gzhang3,lingliu,schwan}@cc.gatech.edu {cooperb}@yahoo-inc.com

Abstract

Distributed stream query services must simultaneously process a large number of complex, continuous queries with stringent performance requirements while utilizing distributed processing resources. In this paper we present the design and evaluation of a distributed stream query service that achieves massive scalability, a key design principle for such systems, by taking advantage of the opportunity to reuse the same distributed operator for multiple and different concurrent queries. We present concrete techniques that utilize the well-defined semantics of CQL-style queries to reduce the cost of query deployment and duplicate processing thereby increasing system throughput and scalability. Our system exhibits several unique features, including : (1) a 'reuse lattice' to encode both operator similarity and network locality using a uniform data structure; (2) techniques to generate an optimized query grouping plan in the form of 'relaxed operators' to capitalize on reuse opportunities while taking into account multiple run-time variations, such as network locality, data rates, and operator lifetime; and (3) techniques to modify operator semantics at runtime to facilitate reuse. Evaluation of our service-oriented design and techniques under realistic workloads shows that stream queries relaxed and grouped using our approach operate efficiently without a priori knowledge of workload, and offer an order of magnitude improvement in performance over existing approaches.

1 Introduction

Modern enterprise applications [16, 2, 3], scientific collaborations across wide area networks [4], and large-scale distributed sensor systems [19, 15] are placing growing demands on distributed streaming systems to provide capabilities beyond basic data transport such as wide area data storage [1] and continuous and opportunistic processing [3]. An increasing number of streaming services are applying 'in-network' and 'in-flight' data manipulation to data streaming systems designed for such applications. One challenge of 'in-network' processing [7, 17, 19] is how to best utilize these geographically distributed resources to carry out end user tasks and to reduce the bandwidth usage or delay [5], especially considering the dynamic and distributed nature of these applications and the variations in their underlying execution environments.

In this paper we address this challenge by exploiting reuse opportunities in large scale distributed stream processing systems, focusing on the class of stream data manipulations described as long-running continuous queries. It is observed that stream queries are typically processed by a selection of collaborative nodes and often share similar stream filters (such as stream selection or stream projection filters). The ability to reuse existing operators during query deployment, especially for long running queries, is critical to the performance and scalability of a distributed stream query processing service. Concretely, we argue that by taking advantage of opportunities to reuse the same distributed operators for multiple and different concurrent queries and intelligently consolidate operator computation across multiple queries, we can reduce the cost of query deployment and minimize duplicated in-network processing. The technical challenges of reuse in streaming systems include dealing with large and time-varying workloads (service requests), dynamically exploiting similarities between queries and the runtime application of network knowledge.

In exploiting reuse opportunities in stream query processing, one straightforward approach is to construct distributed query graphs. However it is known that distributed query graphs cannot be statically analyzed [5, 11, 13, 19] or optimized due to dynamic query arrivals and departures, and due to difficulty in obtaining accurate *a priori* knowledge of workload. Another naive approach is to devise a dynamic solution that considers re-planning of all queries in the system upon the arrival or departure of each single

query. This approach suffers from inordinately large computational overheads [6].

In this paper we present the design and evaluation of a reuse-conscious distributed stream query processing service, called STREAMREUSE. We develop a suite of reuse-conscious stream query grouping techniques that dynamically find cost-effective reuse opportunities based on multiple factors, such as network locality, data rates, and operator lifetime. First, STREAMREUSE not only groups queries with the same operators but also provides capabilities to take into account containments and overlaps between queries in order to utilize reuse opportunities in queries that are partially similar. Second, our system performs 'reuse refinement' by combining operator similarity and network locality information to enhance the effectiveness of in-network reuse. We aim at locating and evaluating different reuse opportunities possible at different network locations. A reuse lattice is devised to encode both operator similarity and network locality using a uniform data structure and to assist in fast identification of reuse opportunities from a large space of operators. With the reuse lattice and our cost model we can efficiently generate an optimized query grouping plan that capitalizes on those 'relaxed operators' satisfying both operator similarity and network locality requirements. Finally, we develop techniques to perform 'relaxations' at runtime and to allow modifications and seamless migration of existing queries to new plans.

A detailed experimental evaluation of the STREAMREUSE approach uses both simulations and a prototype. Results show that the STREAMREUSE approach outperforms existing approaches under different workloads by reducing network and computational resource usage, and offers an order of magnitude improvement in stream query processing throughput.

2 STREAMREUSE System Overview

This section presents some motivating examples and an overview of the STREAMREUSE system architecture. Multi-query optimization is important for a wide variety of systems and applications. Examples include long running queries in airline computer reservation systems, in enterprise operational information systems, queries that perform pre-caching over distributed data repositories or support scientific collaborations or carry out network monitoring. The specific motivating example used in our research is derived from the airline industry based on our collaboration with Delta Air Lines [16].

2.1 Operational Information Systems

Delta's operational information system (OIS) provides continuous support for the organization's daily operations

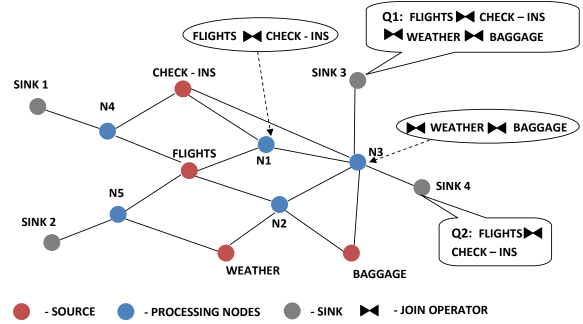

Figure 1. An example network N

and combines three different types of functionality: continuous data capture, for information such as flight and passenger status; continuous status updates, to a range of end-systems including overhead displays, gate agent PCs and large enterprise databases; and responses to client requests which arrive in the form of queries. In order to answer these queries, data streams from multiple sources need to be joined based on the flight, location or time attribute, perhaps using a technique like a symmetric hash join.

Let us assume Delta's OIS to be operating over the small network N shown in Figure 1. Let WEATHER, FLIGHTS, CHECK-INS and BAGGAGE represent sources of data-streams of the same name and nodes N1-N5 be available for in-network processing. Each line in the diagram represents a physical network link. Also assume that we can estimate the expected data-rates of the stream sources and the selectivities of their various attributes, perhaps gathered from historical observations of the stream-data.

Assume that the following CQL-like query Q1 is to be streamed to a terminal overhead display SINK3 and results are to be updated every 1 minute.

```
Q1: SELECT FLIGHTS.NUM, FLIGHTS.GATE,
    BAGGAGE.AREA,
    CHECK-INS.STATUS, WEATHER.FORECAST
    FROM FLIGHTS [RANGE 5 MIN], WEATHER [RANGE 5
    MIN], CHECK-INS [RANGE 1 MIN], BAGGAGE [RANGE
    1 MIN]
    WHERE FLIGHTS.DEST = WEATHER.CITY
    AND FLIGHTS.NUM = CHECK-INS.FLIGHT
    AND FLIGHTS.NUM = BAGGAGE.FLIGHT
    AND FLIGHTS.TERMINAL = ``TERMINAL A''
    AND FLIGHTS.CARRIER_CODE = ``DL'';
```

Q1 is deployed by applying the filter conditions and the project operators for the various attributes at the source. The join operator FLIGHTS⋈CHECK-INS is placed at node N1 and join with WEATHER and BAGGAGE at N3. All join operators are evaluated every minute.

Now assume that a new ad-hoc query Q2 is posed by an airline manager in order to determine whether any low-capacity flights can be canceled and customers shifted to a partner airline's flight. Let us assume that the results need to be refreshed every 5 minutes.

Q2:
```
SELECT FLIGHTS.NUM, CHECK-INS.STATUS,
CHECK-INS.VACANT_SEATS
FROM FLIGHTS [RANGE 5 MIN], CHECK-INS [RANGE
5 MIN]
WHERE FLIGHTS.NUM = CHECK-INS.FLIGHT
AND FLIGHTS.CARRIER_CODE IN (``DL'',``CO'');
```

Firstly, depending upon the sink for Q2, we may decide to reuse the existing join operator at node N1 or redeploy a new join operator. For example, if Q2 arrives at SINK4 it may be beneficial to reuse the operator but if it arrives at SINK1 we may prefer to deploy a new join operator. Secondly, in order to be able to reuse the join FLIGHTS⋈CHECK-INS, we would have to completely remove some filter conditions (on attribute TERMINAL) before the join, relax some conditions (on attribute CARRIER_CODE) and place the original conditions after the join. Thirdly, this would imply that we would have to project some additional columns (attribute TERMINAL and VACANT_SEATS in this case). Also, we must now expand the window size for the CHECK-INS stream at the FLIGHTS⋈CHECK-INS operator to 5 minutes, but only forward CHECK-INS data within a one minute window to query Q1. Additionally, we must filter updates based on timestamp such that results of query Q2 are streamed only every 5 minutes.

Several attributes of the example presented in this section are important to our research. When queries are not known *a priori*, reuse opportunities that exploit cross-query optimization need to be identified and deployed at runtime. Also, the benefit from reuse depends on network locality.

2.2 STREAMREUSE System Architecture

The STREAMREUSE sub-system is implemented over IFLOW [14], a distributed data stream processing system. The IFLOW system utilizes a scalable virtual hierarchical network partition structure to collect and maintain information about the nodes and operators in current use [14]. Briefly, this virtual hierarchy is a structure composed of a network of nodes grouped into regions based on the notion of network locality. In each region, one node is designated as the coordinator or **planner node** for the region and manages all nodes and links within its region.

The key contribution of this paper is the query planning process at the **planner node**. The reuse lattice, the semantic analyzer, and the cost model components of the planner node provide the functionality required for identifying and evaluating reuse opportunities. The semantic analyzer utilizes operator semantics to identify existing operators that can be reused in the computation of the new query request (see Section 3). The reuse lattice congregates information from the operator repository and the network hierarchy into a single structure to allow efficient search and indexing (see Section 4). Finally, the cost model combines information from all components to compute a cost-measure for each candidate reuse opportunity/deployment.

We use the metric of 'network usage' [17] to compute costs of query deployments. The network usage metric computes the total amount of data in-transit in the network at a given instant. We define a cost function $\mathcal{C}(G, \Theta(t))$ that estimates the total network usage per unit time for deploying operators $\Theta(t)$ over the system G. Note that the set of operators $\Theta(t)$ remains constant as long as no queries join or leave the system and may change only at the instant of deployment or departure of a query. When new queries are deployed, $\Theta(t)$ may change due to the addition of new operators and the modification of existing operators. Then the total cost of the system is given by $\sum_t^\infty \mathcal{C}(G, \Theta(t))$. We consider the minimization of the overall system cost while taking the long-running nature of the queries into consideration, $min(\sum_t^\infty \mathcal{C}(G, \Theta(t)))$ as an objective function.

3 Identifying Reuse Candidates

The aim of the semantic analyzer is to identify two kinds of reuse opportunities: (1) Containment or Exact matches and (2) Overlaps, where even existing operators that cannot be directly reused to compute a query can be modified in order to induce reuse. Throughout the paper, continuous queries are specified using the SQL semantics. Each operator θ is specified using its definition, network location and lifetime. An operator serves as a source for the stream computed by its underlying query.

In order to reuse θ_i in the evaluation of Q, the following **base conditions** should be satisfied: both should refer to the same set of stream sources, specify identical join conditions and the group-by conditions (for aggregation) used by Q should be a subset of those used by θ_i.

3.1 Relaxation and Compensation

Our focus is primarily on relaxing join operators. We consider four kinds of relaxations of existing operators in the system: (1) relaxation of selection predicates, (2) relaxation of project operators (3) relaxation of operator lifetime, and (4) relaxation of window specification. Depending upon the query and the existing operator, one or more of these relaxations may be applied. During relaxation, an existing operator is modified into a 'relaxed' operator and compensation operators. Compensation operators are introduced to ensure the consistency of results of existing queries and are also used to rewrite the new and existing queries in terms of the relaxed operator.

Relaxation and Compensation: Let θ represent an already deployed operator and Q represent a new query. Then, $\overline{\theta}$ is called a 'relaxation' of θ under Q, if both Q and θ can be computed from $\overline{\theta}$ by applying only simple selection, projection and temporal filter operators additionally over $\overline{\theta}$. These additional operators are referred to as **compensation**

operators. A relaxation $\overline{\theta}$ of θ under Q is called a **minimal relaxation** if $\forall$ relaxations θ_i of θ under Q, $\overline{\theta} \subseteq \theta_i$.

Given that an operator θ satisfies the base conditions with a query Q, a minimal relaxation of the operator is computed using the following steps. These steps are then demonstrated using an example.

1. Relaxing selection predicates : An operator can be relaxed by modifying the selection predicates, there by imposing a less restrictive filter condition. We relax the operator θ such that the new relaxed operator $\overline{\theta}$ is a minimal cover of θ and Q. Since selection operators are idempotent, a simple way to compose compensation operators σ^Q (or σ^θ) is to include all predicates that appear in Q (or θ).

2. Relaxing projections : Relaxing projection conditions involves expanding the list of projected columns to include those required by both Q and θ. The compensation projection operators Π^Q and Π^θ are simply those columns in the output list of Q and θ respectively.

3. Relaxing operator lifetimes : The lifetime of the relaxed operator is set to the maximum of the lifetimes of all the queries using it. The lifetime of compensation operators are set to those of the original operator or query respectively.

4. Relaxing windows : Our relaxation techniques are primarily aimed at sliding-window join operators where windows are specified using a *range* i.e., size and *slide* i.e., frequency of computation. Windows are relaxed by using the larger of the range specifications of θ and Q and by using a boolean OR condition over the two slide specifications.

5. Cascading relaxations : If relaxing an operator involves the relaxation of conditions that are not local (i.e. not at the current operator itself) but instead are embedded into the input streams by some upstream operator, then we may need to perform cascading relaxations (and the associated compensations) for those upstream operators as well.

3.2 Example

The following example explains the relaxation and compensation process for the queries $Q1$ and $Q2$ described in Section 2.1. We explain how the FLIGHTS⋈CHECK-INS join operator θ_j deployed for query $Q1$ should be relaxed to be reused in the evaluation of query $Q2$. Originally,

θ_j :
```
SELECT FLIGHTS.NUM, FLIGHTS.GATE,
  FLIGHTS.DEST, CHECK-INS.STATUS FROM FLIGHTS
  [RANGE 5 MIN], CHECK-INS [RANGE 1 MIN]
  WHERE FLIGHTS.NUM = CHECK-INS.FLIGHT
  AND FLIGHTS.CARRIER_CODE = ``DL''
  AND FLIGHTS.TERMINAL = ``TERMINAL A'';
```

Since the base conditions are satisfied by θ_j under $Q2$, the operator can be reused with the query. However, relaxation is required. We briefly outline the steps to compute the minimally relaxed operator $\overline{\theta_j}$ next.

1. **Relaxing selection predicates :** The selection conditions C of $\overline{\theta_j}$ is given by:

C :
```
FLIGHTS.NUM = CHECK-INS.FLIGHT
  AND FLIGHTS.CARRIER_CODE IN (``DL'',``CO'')
```

In this case, compensation selection operators are required only for the existing query, and the conditions C^θ in the compensation operator σ^θ are the selection predicates specified in the original operator θ_j.

2. **Relaxing project operators :** The project column list L in $\overline{\theta_j}$ is given by:

L :
```
FLIGHTS.NUM, FLIGHTS.GATE, FLIGHTS.DEST,
  FLIGHTS.TERMINAL,FLIGHTS.CARRIER_CODE,
  CHECK-INS.STATUS, CHECK-INS.VACANT_SEATS
```

The compensation projection operators are simply those columns specified by θ_j and $Q2$.

3. **Relaxing lifetimes :** The lifetime of the new operator $\overline{\theta_j}$ will be the maximum of the lifetimes of $Q2$ and $Q1$.

4. **Relaxing windows :** The window range for the CHECK-INS stream in $\overline{\theta_j}$ is set to 5 minutes (maximum of range for Q1 (i.e. 1 minute) and Q2 (i.e. 5 minute)). Similarly, the slide is specified by the following boolean condition: $((t \; mod \; 1 \equiv 0) \vee (t \; mod \; 5 \equiv 0))$, which is simplified to just $(t \; mod \; 1 \equiv 0)$. Range compensation operator τ^θ filters out tuples whose data items corresponding to the CHECK-INS stream fall beyond a 1 minute window. Similarly, slide compensation operator Γ^{Q2} only forwards results that appear at a 5 minute interval.

5. **Cascading relaxations :** Since the selection conditions on the FLIGHTS source are actually performed at the source, the conditions in the selection operator at the source will have to be replaced with C. The same applies to the project operators at the sources FLIGHTS and CHECK-INS.

4 Searching using Reuse Lattice

The 'Reuse Lattice' data structure combines information from the operator repository and the network hierarchy into a single structure that allows efficient search and identification of reuse opportunities.

The reuse lattice uses the operator definition to encode containment. Since operator definitions are similar to view definitions in traditional databases, this allows us to leverage the large body of existing work in rewriting queries using materialized views. Particularly, we adapt the filter tree index structure described in [12] to efficiently maintain containment relationships between operators. Section 4.1 describes the adaptation of the structure to the context of continuous queries to create the reuse lattice. Section 4.2 describes techniques that extend the structure to incorporate network locality.

4.1 Encoding Operator Containment

The reuse lattice adapts a restricted filter tree structure [12] to the context of a distributed stream processing

system. Given a query, the filter tree structure can be used to quickly narrow down the list of candidate operators in the system that will give rise to valid rewrites. The filter tree is a multiway search tree where all leaves are at the same level. A single node in this structure represents a collection of operators. Different partitioning conditions are applied to the nodes at each level to further partition the operators into multiple smaller disjoint nodes at the next level. For example, at the top-most level, operators are partitioned into disjoint subsets based on source streams (specified in the FROM clause of the operator definition). Each disjoint subset is represented at that level by a single node in the filter tree. A different partitioning condition is applied at each subsequent level. For example, we partition nodes into disjoint subsets based on join predicates at the second level and group-by predicates at the next level. At this point, all the base conditions have been accounted for. We further partition each node based on each of the relaxable conditions, viz. selection predicates, project column list and window specifications. The last three levels in the filter tree are only used when searching for reuse opportunities that do not require modifications. The key at each level is determined by the partitioning condition. For example, if the partitioning condition is the set of source streams, then the list of sources specified in the FROM clause of the definitions serves as the key. Each node in the filter tree is a collection of <key,pointer> pairs that may further be organized into an internal index structure based on containment of keys (determined by the partitioning condition) to further speed-up search within a node. At the lowest level in the lattice, the internal nodes contain pointers to actual operator definitions.

4.2 Encoding Network Location

In order to allow search based on different granularities of network locality, network nodes are organized into 'regions' based on the notion of "nearness in the network". The organization of network nodes into regions can be based on a clustering algorithm like K-Means that uses inter-node delay as a clustering parameter or a static grouping if the distribution of nodes in the infrastructure is known before hand. Each region is identified by a unique bit-vector of length n, where n is the number of regions. We refer to this bit-vector as a 'Region ID' (RID). The RID for the ith region has the ith bit set to 1 and all other bits set to 0.

The network location indicator (NID) of an operator, is a bit-vector that represents the region(s) to which the operator belongs. The ith bit of the NID is set to 1 only if the node belongs to the ith region. At the lowest level, each internal node contains pointers to all operators with the same key at each level of the lattice. Each internal node in the lattice is again associated with an NID which is the bitwise OR of all

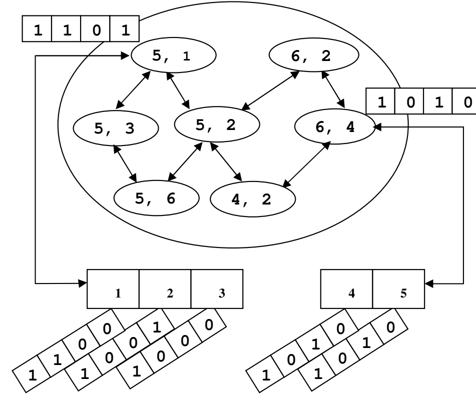

Figure 2. A single leaf lattice node.

the associated operator NIDs. Note that the same operator may appear at multiple network locations causing the operator NID to have more than one bit set to 1. Search by network location is supported by providing the lattice search algorithm with a bit-vector that specifies network locality restrictions. If relaxations are allowed, the search algorithm selects all operators that satisfy the base conditions.

Figure 2 shows an example lattice node. The figure shows a single leaf level lattice node where the partitioning condition is the window specification. As the figure shows, the lattice node contains a collection of keys, (RANGE, SLIDE) specifications in this case, such as (5,1), (6,2) etc., organized into a containment (see Section 3.1) based structure. Since this is a leaf node, each internal node contains a pointer to a set of operators. Each operator is associated with a NID corresponding to the regions where the operator resides. For example, the NID of key (5,1) indicates that such an operator is available in regions 1, 2 and 4.

5 Experimental Evaluation

Experimental evaluation of the STREAMREUSE approach studies the performance of our techniques with respect to a number of metrics such as resource usage, latency, and planning time. Experiments were performed on both simulations and a prototype and were conducted using two very different workloads: an enterprise workload obtained from Delta Air Lines and a synthetically generated RFID workload. Our results show that (1) our techniques can reduce network usage by as much as 96% when compared to the state-of-art approaches and (2) by our dynamic grouping approach, computation costs can be reduced by more than an order of magnitude while the increase in latency and time-to-deployment is negligible.

5.1 Workloads

Our techniques are primarily aimed at workloads with a high number of simultaneously executing continuous

queries where there is significant opportunity for operator reuse. It should be noted that while our system can handle the other workloads, if the number of queries is low, the benefit to be realized from enabling operator reuse is also low. We evaluate our techniques using two representative workloads. Enterprise information systems (Section 2.1) are representative of workloads with a high number of queries and a low number of sources. Similarly, applications like the ones used for inventory tracking using RFID tags can be categorized as a workload with a high number of sources and a high number of queries.

EW: Enterprise-Workload The enterprise workload is a real-world workload consisting of gate-agent, terminal and monitoring queries posed as part of the day-to-day operations of Delta Air Lines' enterprise operational information system. The query workload is based on the 5 query sources - Flights, Check-ins, Baggage, Weather and Sales. Gate agent queries constitute 80% of the workload and the SLIDE for these queries is set to 1 minute. The queries use the following template.

```
Q3: SELECT FL.GATE, BG.STATUS, CI.STATUS
    FROM FLIGHTS FL [RANGE 5 MIN], BAGGAGE BG
    [RANGE 1 MIN], CHECK-INS CI [RANGE 1 MIN]
    WHERE FLIGHTS.NUM = CHECK-INS.FLIGHT
    AND FLIGHTS.NUM = BAGGAGE.FLIGHT
    AND FLIGHTS.NUM = ?;
```

Each gate agent query originates at the gate of departure of a flight and lasts for 2 hours prior to departure of the flight. Terminal queries, which represent 15% of the workload, are longer running queries (12 hours lifetime) and follow the template of query Q1 in Section 2.1 and are evaluated every minute. Finally, the last 5% of the workload represent long-running (6 hours) ad-hoc monitoring queries over any combination of the 5 sources. For these queries, we use window ranges and slides that are uniformly distributed between [1-5] minutes for all streams. With nearly 1500 flights a day, we assume that queries arrive with a Poisson distribution with $\mu = 60\ queries/hour$. Each update record was assumed to be of the same size (100 bytes).

RW: RFID-Application Workload The synthetic RFID workload models the quadrant representing systems with a large number of queries over a large numbers of sources resulting in smaller overlaps between queries. This workload consisted of 20 sources with varying query size (3-5 joins), Poisson arrival rate ($\mu = 30 queries/hr$) and query lifetimes similar to that of the enterprise workload.

5.2 Experimental Setup

Our prototype was built over a distributed stream processing system [14] and used a testbed of 128 Emulab nodes (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9), organized into a topology that was generated using the standard tool, the GT-ITM internetwork topology generator [20]. Links were 100Mbps and the inter-node delays were set between 1msec and 6msec. The simulation experiments were conducted over transit-stub topology networks generated using GT-ITM. Experiments used a 128 node network, with a standard Internet-style topology: 1 transit (e.g., "backbone") domain of 4 nodes, and 4 "stub" domains (each of 8 nodes) connected to each transit domain node. Link costs (per byte transferred) were assigned such that the links in the stub domains had lower costs than those in the transit domain, corresponding to transmission within an intranet being far cheaper than long-haul links. We used a uniformly random selection of nodes for sink placements.

The STREAMREUSE approach is compared with two other state-of-the-art techniques: (1) NO REWRITING, reuses existing operators only if their definitions exactly match the requirements and does not perform any rewritings [17, 18] and (2) NO REUSE [8] does not take into consideration any existing operators while deploying new queries.

In order to compare the resource usage of our techniques with other existing approaches, the following two concrete metrics are used: the **instantaneous network resource usage** and the **number of operators** in the system which is indicative of the processing resource usage. The effect of dynamic operator grouping on response time of individual deployments is measured using the **end-to-end latency** of deployments. Finally, the **time-to-deployment** is used to evaluate the overhead imposed on the planning process.

5.3 Efficiency of Deployments

Figure 3 shows the total network usage per instant of time for a 5 hour duration of system deployment with the EW workload. Gate, terminal and monitoring queries last for 2, 12 and 6 hours respectively. After initial ramp-up, approximately 120 queries execute concurrently.

Under the EW workload each gate query specifies a unique flight number. Similarly, all terminal queries are unique. All selection predicates are placed earlier in the query deployment, as close to the source as possible. In the presence of all unique predicates, the NO REWRITING technique degrades to a NO REUSE technique. As the graph shows, even in the presence of unique selection predicates, our approach of runtime relaxation and rewriting can reduce network usage by nearly 96% as compared to a NO REUSE/ NO REWRITING approach. This large win can be attributed to the fact that fewer join operators to which inputs need to be streamed are deployed in the system. In the presence of highly selective joins, the small increase in input size to few join operators is negligible compared to streaming inputs to a large number of join operators.

Figure 4 shows the total network usage with the RFID workload (RW) over 5 hours. This workload has more

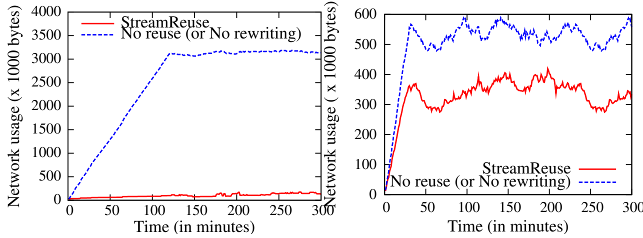

Figure 3. EW: Cost

Figure 4. RW: Cost

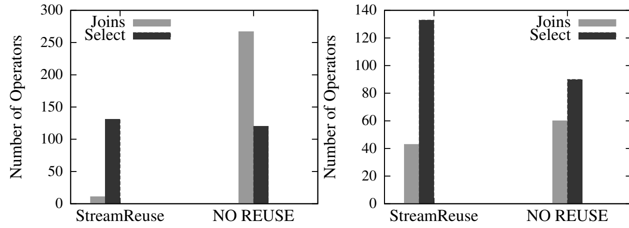

Figure 5. EW: Deployed operators

Figure 6. RW: Deployed operators

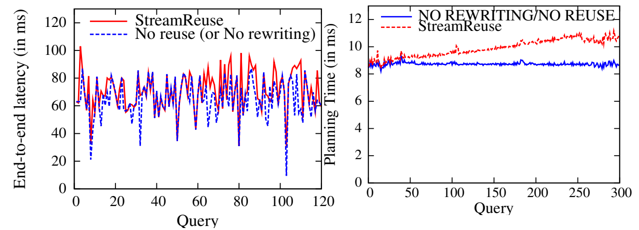

Figure 7. EW: Latency

Figure 8. Total Planning Time

sources than the enterprise workload, and sources specified in queries are chosen from a large range of combinations. In fact, with the RW workload, within any set of concurrently executing queries, only 5 queries had any common joins with other queries. Consequently the number of rewrite/reuse opportunities available are fewer. Again, as in the case of the enterprise workload, since negligible number of queries specify the exact same selection predicates, the NO REWRITING approach degenerates into a NO REUSE approach. The figure shows that a dynamic grouping based approach results in a 28% reduction in network usage even with this workload.

5.4 Evaluation of Computation Costs

Computation costs are evaluated using the average number of operators deployed at any instant of time. Recall that, since all selection predicates are unique in EW, the NO REWRITING approach degenerates into a NO REUSE approach. Figure 5 shows the average number of join, and select operators that are concurrently executing at any given instant of time with STREAMREUSE and the NO REUSE approaches. The figure shows that the STREAMREUSE approach effects a massive decrease in the number of join operators with only a slight increase in the number of selection operators. The increase in selection operators can be attributed to the introduction of additional compensation operators while reusing existing joins. This figure shows that even in the presence of unique queries, by effectively sharing operators between queries through dynamic grouping, the number of join operators can be reduced by an order of magnitude. It is a well known fact that joins are expensive operators and can reduce throughput. By reducing the number of such expensive joins, we expect the throughput to increase significantly. Figure 6 shows the number of deployed operators with the RW workload. Briefly, in spite of the low overlap between queries, our techniques resulted in a 28% reduction in join operators over the NO REUSE approach.

5.5 Effect of Grouping on Latency

Figure 7 shows the end-to-end latency of 120 queries of the EW workload with STREAMREUSE and with NO REUSE/NO REWRITING. The figure shows that latencies experienced by the queries with STREAMREUSE is comparable to that with NO REUSE or NO REWRITING. On an average, latency increases by 13 msec with STREAMREUSE. Under the RW workload the average increase in latency with STREAMREUSE is only 3.675 msec. Since most applications can tolerate this slight increase, this is a small price to pay for the large savings in network and computational costs.

5.6 Prototype Experiments

The scalability of the STREAMREUSE approach was studied by examining the overhead imposed by the planning process (including searching the lattice and computing relaxations) on the total time to deployment. Figure 8 shows the total planning time for the deployment of 300 queries from the EW workload each with an average of 8 operators (including selections and projections). The figure demonstrates: (1)that the increase in planning time is negligible (an average increase of 1 ms) and (2) the increase in planning time with the size of the lattice is near linear. Given the large gains in network and computational costs, this leads us to conclude that this one time overhead at the time of deployment is completely justifiable.

6 Related Work

A number of data-stream systems such as STREAM [10], Borealis [5], NiagaraCQ [11] and System S [9] have been developed to process queries over continuous streams of data. We present a summary of related work pertaining to techniques for operator reuse.

Optimistic-deployment (Static): Such systems use rules such as 'pull-up selections' to improve reusability [11] or assume that the workload is already known [19, 13]. The approach may be effective when the workload is known *a priori*, but is unable to handle a dynamic workload since the system must pay the price of increased intermediate data size even for operators that are never reused.

Runtime recomputation (Dynamic): In this approach, with each arrival or departure of a single query, a portion of the query network [5] is replanned. However, operator groupings are performed by the system administrator and are not dynamic.

System level functionalities that allow runtime modification of operator parameters have been implemented in systems such as Borealis [5]. Our techniques utilize such system-level functionality, along with semantic knowledge, to perform dynamic grouping of queries and runtime migration of query plans. Our work also builds on query rewriting techniques that have been widely studied in the context of materialized views [12]. While operator definitions in our context are similar to view definitions, our problem is complicated by the need to consider network locality, operator similarity, windows and runtime modifications in addition to containment.

7 Conclusion and Future Work

High performance and massive scalability are some of the most important goals in the design of a distributed stream query processing service. In such systems, where the service is often specified as a data flow graph consisting of well-defined user-written or system generated operators, the ability to decompose the request into smaller sub-graphs and effectively utilize reuse opportunities may be the key to achieving the desired levels of scalability and performance.

We have described STREAMREUSE, a reuse-conscious distributed stream query processing system. It uses a dynamic query grouping approach to identify and encode reuse opportunities at runtime based on operator semantics and network locality awareness. We introduce the notion of 'relaxations' and the 'reuse lattice' data structure to enable transformations of existing operators that exhibit operator similarity and network locality similarity to be reused. We evaluate the STREAMREUSE approach by conducting experiments over a range of different workloads. The experimental results show that our reuse techniques can reduce resource usage and computational costs by more than an order of magnitude compared to existing approaches. Our research on STREAMREUSE continues along a number of dimensions. Currently, we are investigating issues pertaining to the migration of stateful operators and scalable techniques for distribution of the planning process and the reuse lattice.

References

[1] Akamai. *http://akamai.com/*.

[2] Amazon ec2. amazon elastic computing cloud. *aws.amazon.com/ec2*.

[3] Ibm websphere. *http://www-306.ibm.com/software/websphere/*.

[4] Terascale supernova initiative. *http://www.phy.ornl.gov/tsi/, 2005.*

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[6] 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 Journal*, 2003.

[7] Z. Abrams and J. Liu. Greedy is good: On service tree placement for in-network stream processing. In *ICDCS*, 2006.

[8] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.

[9] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS*, 2006.

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

[11] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, 2002.

[12] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *SIGMOD*, 2001.

[13] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica. Sharing aggregate computation for distributed queries. In *SIGMOD*, 2007.

[14] V. Kumar et al. Implementing diverse messaging models with self-managing properties using IFLOW. In *ICAC*, 2006.

[15] L. Luo, Q. Cao, C. Huang, T. Abdelzaher, J. A. Stankovic, and M. Ward. Enviromic: Towards cooperative storage and retrieval in audio sensor networks. In *ICDCS*, 2007.

[16] V. Oleson et al. Operational information systems - an example from the airline industry. In *First Workshop on Industrial Experiences with Systems Software, WIESS 2000.*

[17] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[18] S. Seshadri, V. Kumar, B. F. Cooper, and L. Liu. Optimizing multiple distributed stream queries using hierarchical network partitions. In *IPDPS*, 2007.

[19] S. Xiang, H. B. Lim, K.-L. Tan, and Y. Zhou. Two-tier multiple query optimization for sensor networks. In *ICDCS*, 2007.

[20] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Infocom*, 1996.