Optimizing Multiple Queries in Distributed Data Stream Systems

Sangeetha Seshadri, Vibhore Kumar, Brian F. Cooper
College of Computing, Georgia Institute of Technology
{sangeeta,vibhore,cooperb}@cc.gatech.edu

Abstract

We consider the problem of query optimization in distributed stream based systems where multiple continuous queries may be executing simultaneously. In such systems, distribution adds degrees of freedom to an already complex optimization problem. Thousands of network nodes may need to be considered for operator placements in order to support in-network processing - clearly overwhelming even from the perspective of distributed query optimization. Added to this complexity is the potential for significant savings by combining query plans in order to re-use the stream of intermediate results. These issues force us to develop new techniques for query optimization. We present a formal definition of the multi-query optimization problem in such systems and propose some initial directions.

1. Introduction

In recent years an important class of data-intensive applications has emerged. These applications require continuous queries deployed over multiple geographically distributed high-volume data-streams to be processed in real time. Examples include internet-enabled sourcing and procurement tools like TraderBot, financial applications like stock-tickers, applications focused on enterprise-wide inventory management using RFID tags, the distributed event-based operational information systems used by large corporations and remote collaborations where scientists share and analyze massive streams of data.

In such distributed data-stream systems, it is often too expensive to evaluate a continual query at a single central node for a number of reasons, especially the high communication cost of transporting raw streams to the central node and the resulting processing overload at the node. Instead, performing distributed processing of stream queries using techniques like in-network processing [2] and filtering [10] at the source minimizes the communication overhead on the system and helps spread processing load, significantly improving performance. However, in this paradigm we must revisit query optimization techniques to ensure the highest performance.

In this paper we address the problem of multi-query optimization in such a distributed data-stream management system. Traditional centralized databases consider permutations of join-orders in order to compute an optimal execution plan for a single query [9]. The same problem is addressed by networked-databases, while taking into consideration communication costs over a relatively small set of nodes [8]. Centralized stream based systems consider the problem of optimization for multiple queries, rather than just a single query, while trying to minimize processing costs at a single central node [3]. While the above problems are sufficiently challenging in themselves, and have generated a good deal of attention from researchers, the paradigm of in-network processing adds a whole new dimension to the problem - network placement of operators. All the above factors - *join-reordering*, *multi-query optimization* and *communication and processing overheads* now need to be addressed, while keeping in mind the actual placement of operators, sources and sinks over a network that could span thousands of nodes and whose characteristics could be constantly changing with time. In fact, all these considerations are interacting: for example, the best join order might be determined by communication costs and re-use of existing operators in the system.

2. An Illustrative Example

In this section, we use an example network and sample continuous queries to better illustrate the optimization opportunities that may be available in a distributed data stream system. Consider a distributed stream management system operating over a network `N` shown in Figure 1. Let A, B and C represent sources of data-streams and nodes `N1-N5` be available for in-network processing. Each line in the diagram represents a physical network link. Let us also assume that we have with us estimates of the expected data-rates of the stream sources and the selectivities of their various attributes, perhaps gathered from historical observations of the stream-data. Consider the following queries to be deployed on the system:

Q1: `SELECT * FROM A,B,C WHERE` $F_A \wedge F_B \wedge F_C$

Q2: `SELECT * FROM A,B WHERE` $F_A \wedge F_B$

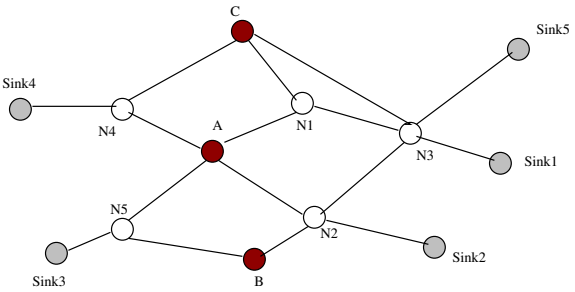

Figure 1. An example network N

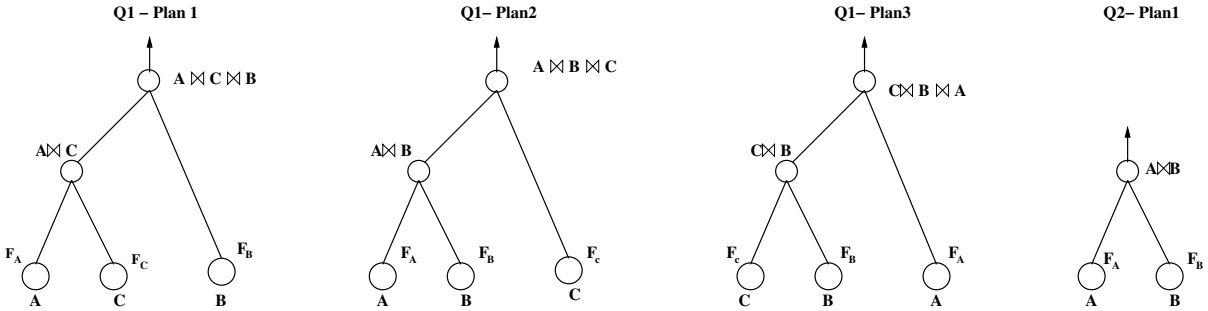

Figure 2. Execution Plans

where $\mathtt{F}_S$ represents a filter predicate on stream S. Assume that the joins A⋈C and A⋈B are data-decreasing while join A⋈B⋈C is data-increasing. The possible execution plans for the queries Q1 and Q2 are depicted in Figure 2. Assume that, with respect to the size of the intermediate results generated, Q1 - Plan1, shown in Figure 2(a), is the optimal execution plan for Q1 and Q2 - Plan1, shown in Figure 2(d), is the optimal execution plan for Q2. Consider the following scenarios:

1. Unique Operators: Assume that the sinks for Q1 are located at both `Sink1` and `Sink5` and that for query Q2 is located at `Sink3`. An optimal choice of execution plans, in this case, is likely to be one that minimizes the amount of data communicated across the network, by placing data-increasing operators close to the sink and data-decreasing operators close to the source. For example, a possible choice of plan for Q1 would be plan Q1-Plan1, with the data-decreasing operator A⋈C placed at node `N1` and and the data-increasing operator A⋈C⋈B placed at node `N3`. Similarly, a possible choice for Q2 may be plan Q2-Plan1 with the data-decreasing operator A⋈B placed at the node `N5`. Thus the choice of an optimal execution plan involves an appropriate selection of operator order(query plan) as well as operator placement. Generalizing, we may conclude that ***optimality of plans must take into account both the query plan and its actual network placement.***

2. Operator Re-use: Now suppose that the sink for Q2 is located at node `Sink2` instead. If we use plan Q1-Plan2 shown in Figure 2 it is possible to re-use operator A⋈B for both queries, thereby saving bandwidth and processing costs, since communication costs for transporting input data and processing costs for computing the join will be incurred only once. Thus, in spite of Q1-Plan2 being a sub-optimal plan for query Q1 in terms of the size of intermediate results, the combination of Q1-Plan2 and Q2-Plan1 might result in a lower cost deployment overall. Generalizing, we conclude, ***sub-optimal plans may have to be considered***.

3. Operator Duplication: Now consider a third query Q3 given by:

Q3: `SELECT * FROM A,C WHERE` $\mathtt{F}_A \wedge \mathtt{F}_C$

Suppose that the sinks for query Q1 are located at `Sink1` and `Sink5` and that for query Q3 is located at `Sink4`. In this scenario, although we may use Q1-Plan1 for query Q1, we may want to duplicate operator A⋈C placing one such operator at nodes `N1` (used by Q1) and `N4` (used by Q3) rather than re-use a single copy. This is because 're-use' may mean that large intermediate results need to be transported over long network distances between operators/sink. Generalizing, we conclude that, although the same operator may be used by multiple queries, ***network placement decides between duplication and re-use***.

4. Delayed Filtering: Say the sink for Q1 is located at `Sink1` and that of Q2 at `Sink2`. If the filter predicates on streams A and B in query Q1 and Q2 are *different* but not very selective in either case, it may be better to perform

delayed filtering, after the A▷⋈B operation, allowing us to re-use the A⋈B operator which would not be possible otherwise. Thus, depending upon the queries currently executing in the system and the network placement of sources and sinks it is ***possible that placing filters further downstream from the source minimizes cost***.

The purpose of the above example is to illustrate that, in a distributed stream management system where multiple queries may be executing simultaneously, optimal plan selection and operator placement are largely determined by the following factors - current workload of queries, network placement of operators, sources and sinks and individual operator selectivity. Plans that may be optimal for a single query may not result in an optimal deployment overall. Thus it is necessary to consider *combination of execution plans* and *actual placement of operators, sources and sinks*.

The remainder of this paper is organized as follows - a formal description of the query optimization problem and our cost-benefit model for such optimizations is presented in Section 3. In Section 4 we describe some initial solutions that have a strong analytical footing and conceptually blur the divide between networking and databases. A preliminary evaluation is presented in Section 5.

3. Problem Definition

We consider the query optimization problem for multiple continuous queries that may be executing simultaneously in a distributed stream management system. Our problem definition addresses the continual query equivalent of 'select-project-join' queries. We assume that potentially, *any* operator can be deployed at *any* node in the system. Given a query, there could possibly be multiple execution plans that the system could follow to produce results. We assume that all such plans are equivalent, i.e., all plans produce the same result tuples. Note that it is possible, depending upon the technique used for join operations, that a different ordering of operators may produce different results. However, as usually applicable in stream-based systems where we often rely on approximate results, such a working assumption is acceptable as long as the variation is small.

3.1. System Definition

We define the underlying infrastructure as follows: Let $N(V_n, E_n)$ represent a physical network where vertices V_n represent the set of actual physical nodes and E_n, the set of network connections between the nodes. We further associate each edge e_{ni} with an application-dependent cost function $\Phi_c(e_{ni})$ that represents the communication cost (per byte) associated with sending data along the corresponding network link. Also, we represent the processing cost associated with each vertex v_{ni} by $\Phi_p(v_{ni})$.

Let Q represent a single continuous query and let $P^Q = \{p_1^Q, \ldots, p_m^Q\}$ represent the set of all execution plans for query Q. Each plan p_j^Q can be represented as a directed acyclic graph $G(V_j^Q, E_j^Q)$ where

$$V_j^Q = V_{sources}^Q \bigcup V_{operators}^Q \bigcup V_{sink}^Q$$

$V_{sources}^Q$ is the set of stream sources for a particular dataflow graph and each source has a static association with a vertex in graph N. Each source vertex v_s^Q has an associated *data-flow rate* $\lambda(v_s^Q)$ at which it produces data. V_{sink}^Q is a singleton set that represents the sink for the results of the query Q and it also has a static association with a vertex in graph N. $V_{operators}^Q$ is the set of operators that can be dynamically associated with any vertex in graph N. Each operator vertex v_u^Q is characterized by a *selectivity-factor* ρ_u^Q, which is defined as the average ratio of the data-flow rate of the output stream to that of the product of input data-flow rates. The above definition of an operator treats the transformations within an operator as a black-box. This makes our model independent of the specific technique (such as windowed joins or symmetric hash joins) utilized at the operator. The data-flow rate $\lambda(e_{juw}^Q)$ associated with each edge e_{juw}^Q between operators v_{ju}^Q and v_{jw}^Q is the rate at which the source vertex v_{ju}^Q produces data. For edges between operator vertices, this equals the product of all incoming stream rates and the *selectivity-factor* of the operator deployed at the origin of the edge. Hence given data-flow rates at the *stream-sources*, the data-flow rate associated with any edge from/to an operator vertex can be calculated using:

$$\lambda(e_{juw}^Q) = \rho_u^Q \times \prod_{e_{jku}^Q \in E_j^Q} \lambda(e_{jku}^Q)$$

We define the *deployment* of a query plan p_j^Q over the network N to be a mapping $M(p_j^Q, N)$, which assigns each vertex $v_{jk}^Q \in V_j^Q$ to a network node $v_{nk} \in V_n$. Thus, M implies a corresponding mapping of edges in G to edges in N, such that each edge e_{juw}^Q between operators v_{ju}^Q and v_{jw}^Q is mapped to the network edges along the lowest cost path between the network nodes to which v_{ju}^Q and v_{jw}^Q are assigned. Thus, for each $v_{jk}^Q \in V_j^Q$ we have $M(v_{jk}^Q) \in V_n$ and similarly for each $e_{juw}^Q \in E_j^Q$ we have $M(e_{juw}^Q) \subseteq E_n$

3.2. Utility Model: Cost-Benefit Trade-Offs

In a distributed data-stream system where communication and processing costs are high and incurred continuously, a query execution plan should try to achieve an optimum over two conflicting objective functions - one that tries to minimize the costs incurred and another that tries to

minimize response time (e.g., source to sink data latency). The two objective functions may be conflicting, since it is possible that lower delay paths have higher costs associated with them. We deal with this conflict by combining both objectives into a single utility function, which we try to maximize. It is also possible to define a constraint on one objective while minimizing the other, but we leave this alternative for future work.

We model our system using an application-provided *utility-function* that is based on the notion that attributes such as response-time and user-priority have an associated *benefit*. The system then tries to find a mapping M_k for query Q that maximizes the *net-utility* $U_{net}(Q, M_k)$, which is defined as the difference between the benefit achieved ($U(Q, M_k)$) and the cost incurred in order to achieve that benefit ($TotalCost(M_k)$). The two components of net-utility, U_{net}, are expressed in terms of some unit of value (e.g. dollars). Next, we define the cost function used by our utility model.

3.2.1 Cost Model

Associated with each mapping is a cost per unit time specified by a cost function $TotalCost$. This function $TotalCost$, represents the total amount of processing and communication resources expended to execute the query using the specified mapping. Thus,

$$TotalCost(M) = Cost_C(M) + Cost_p(M)$$

We define the communication cost for mapping M, $Cost_c(M)$ as the sum of the cost of data-flow along the network edges mapped to the edges in the data flow graph:

$$Cost_C(M) = \sum_{e^Q_{juw} \in E^Q_j} \Phi_c(M(e^Q_{juw})) \times \lambda(e^Q_{juw})$$

For example, consider a cost function Φ that depends on the bandwidth consumed. If vertices v^Q_{ju} and v^Q_{jw} are the endpoints of edge e^Q_{juw}, and are assigned to vertices v_{nu} and v_{nw} respectively of the network graph N, then the cost corresponding to edge e^Q_{juw} is proportional to the bandwidth consumed along the shortest path between the vertices v_{nu} and v_{nw}.

We define the processing cost for mapping M, $Cost_p(M)$ as the sum of the processing costs at each of the physical nodes to which the vertices of the execution plan are mapped. Therefore,

$$Cost_p(M) = \sum_{v^Q_{ju} \in V^Q_j} \Phi_p(M(v^Q_{ju}))$$

If an operator v^Q_{ju} deployed on a physical node v_{ni}, or a data flow edge e^Q_{juv} between two nodes, is used by multiple queries, we only count the cost once.

3.3. Optimization Problems

We now define the query-optimization problem for queries to be deployed on a network N.

Single Query-Optimization Problem: Given a query Q and a network N on which the query is to be deployed, find a query plan p^Q_i and a deployment $M(p^Q_i, N)$ such that $U_{net}(Q, M(p^Q_i, N))$ is maximum over all possible plans for query Q and all possible deployments of such plans over N. ■

The incremental query optimization problem is to find an optimal deployment for a query, given a set of base stream sources and a set of query deployments already executing on the system, by taking into consideration operator re-use. This problem is similar to that of computing optimal query plans in database systems while utilizing existing materialized views [6].

Incremental Query-Optimization Problem: Given a query Q to be deployed over a network N, and a set of existing deployments $D = \{D_1, \ldots, D_n\}$, find a query plan p^Q_i and a deployment $M(p^Q_i, N)$ for Q such that $U_{net}(Q, M(p^Q_i, N))$ is maximum over all possible plans for query Q and all possible deployments of such plans over N. ■

The multi-query optimization problem is to find mappings for a set of queries at the same time.

Multi Query-Optimization Problem: Given a set of queries $Q = \{Q_1, \ldots, Q_i\}$, we need to find a set of plans $P = \{p^{Q1}_i, \ldots, p^{Qi}_k\}$ and a corresponding set of mappings $M = \{M(p^{Q1}_j), \ldots, M(p^{Qi}_n)\}$ for each $p^{Qj}_k \in P$ such that the *net-utility* of all the mappings $\sum_{M(p^{Qj}_k) \in M} U_{net}(Q_j, M(p^{Qj}_k))$ is maximum over all possible *sets of plans* and all possible *sets of mappings*. ■

4. Overview of the Approach

Traditional query optimizers use some form of exhaustive search over the plan space, using statistics-based cost estimates. Exhaustive search is infeasible for optimizing distributed stream queries in a large network. In the single and incremental case, the optimizer must consider all combinations of query plans and possible deployments. In the multi query case, the optimizer must also look at all possible combinations of queries. We have developed two approximate approaches for query optimization. Both use a hierarchical organization of the network nodes. We now briefly describe this organization, and both approaches. Due to space constraints, more details are presented in [11].

4.1. Scalability Using Network Partitions

We partition the nodes in the network into a hierarchy, with nodes clustered based on a parameter such as *inter-*

node traversal cost. Each cluster has a maximum cluster-size, max_{size} that is pre-defined. These nodes are organized into a hierarchy as follows: Beginning at the lowest level, nodes are organized into clusters, whose size is limited by the parameter max_{size}. Each node within a cluster is aware of the inter-node cost between every pair of nodes in the cluster. A single node from each cluster is then selected as the *coordinator* node for that cluster and promoted to the next level. This process of clustering and partitioning continues until *Level N* where we have just a single cluster. Such an organization provides us with both locality information and network information at different granularities.

4.2. The Bottom-Up Approach

The *Bottom-Up* approach, given a query and a set of sources, finds the optimal query plan and its deployment in an integrated manner. We observe that every deployed operator and sink is a new stream source for the data computed by its underlying query. We refer to these streams as *derived streams* and the basic stream sources as *base streams*. Each coordinator maintains a list of all base streams and derived streams available in its underlying cluster. Queries are registered at their sink. When a new query `Q` arrives at a sink, the coordinator is informed. The coordinator then decomposes the query into subqueries that can be answered by locally and remotely available streams. An exhaustive search is used to plan the execution of the local subquery. The remote subquery is passed up to the next level coordinator, who similarly decomposes it. This process continues up the hierarchy until all the necessary streams have been found. By limiting exhaustive searches to only sub-queries that can be composed in a single partition, the bottom-up approach is able to bound the search-space. By considering plans that are composed using both derived and base sources, the coordinator takes into account re-usability of existing operators when deploying queries incrementally. Coordinators handle multi-query optimization by materializing consolidated optimal plans for intersecting portions of multiple queries.

4.3. The Top-Down Approach

The *Top-Down* approach bounds sub-optimality by making deployment decisions using bounded approximations of the underlying network; specifically, each coordinator's estimate of the distance between its cluster and other clusters. The approach works as follows: The query Q is submitted to the top-level (level t) coordinator. The coordinator then chooses an optimal deployment at that level through an exhaustive search on its bounded underlying cluster while taking into consideration operator re-use. An assignment of operators to nodes essentially partitions the query plan into a number of views, each allocated to a single node at level t-1 which then repeats this process using the sub-query assigned to it. This process continues until level 1 at which all operators are assigned to actual physical nodes. Multi-query optimization is performed by applying the top-down approach to a single consolidated query plan. The hierarchical structure, the progressive partitioning of operators and the limited size of partitions ensure that there are few plans and nodes at each level, minimizing the expense of the exhaustive mapping.

5. Experiments

In this section, we present experiments that corroborate our claim that significant cost savings can be achieved in distributed data stream systems through optimization techniques that exploit both network placement of sources, sinks and operators and re-use of existing operators, besides considering the size of intermediate results.

Our experiments are based on simulations over 128-node and 64-node transit stub topologies generated using the GT-ITM internetwork topology generator [16]. Links inside a stub domain are 100Mbps and other links are 622Mbps resembling OC-12 lines. Our synthetic workload of 100 queries was generated using a random workload generator. Each query contained 2-5 join operators and sink placements were also generated by the workload generator. A simplified utility calculation model was used in the experiments - it assumed equal benefit for all the deployed queries and therefore minimizing the deployed cost was our aim.

5.1. Cost of Network-Aware Query Plans

In our first experiment, we studied the effect of network-aware query plans on the total cost of the system using a 64-node network and 5 stream sources. For each query, we calculated the total cost of deployments generated by a phased approach that first chooses an execution plan with the least size of intermediate results and then computes an optimal deployment of the operators. We compared this total cost with the total cost of deployments generated using the bottom-up approach. Figure 3 shows the total cost (cumulative) per unit time (since the queries are long-running continuous queries) over 100 queries. It is evident that an integrated approach to deployment decreases the total deployment cost of the system by more than 50%. This can be attributed to the ability of such plans to exploit network knowledge, and re-use existing operators.

5.2. Evaluation of the Deployment Approaches

In this section we experimentally demonstrate that the sub-optimality resulting from the approximations of the

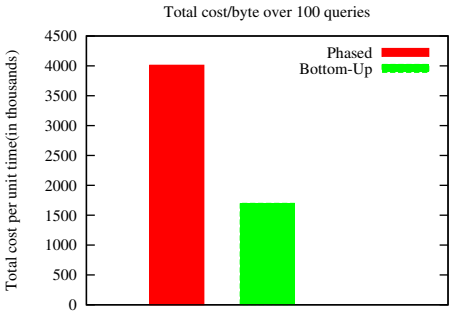

Figure 3. Cost of deploying database plans versus network-aware plans

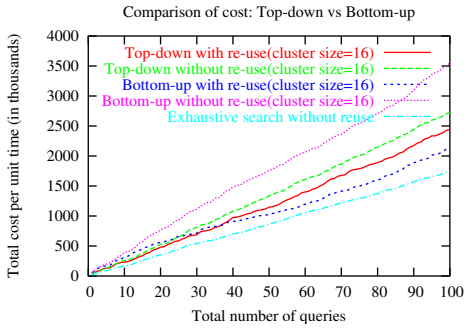

Figure 4. Cost comparison of Top-Down and Bottom-Up approaches with exhaustive search

Bottom-Up and Top-Down approaches is not huge. The experiment was conducted on a 128 node topology with 10 source streams and the query workload was generated as described earlier. Figure 4 shows the cumulative cost of deployment using the bottom-up and top-down approaches, both with and without operator reuse. The figure also shows the cost of deployment using the exhaustive search without reuse. While the exhaustive search required an average of 3-4 minutes per query, our approaches required only 0.08-0.15 minutes. As is evident from the results, the two suggested approaches for network-aware multi-query deployment perform well even against the optimal deployment.

6. Related Work & Conclusion

There have been a number of efforts focusing on distributed processing of stream queries in general [1, 5, 12] and optimization in particular [2, 4, 15, 14]. Systems like Eddies [4, 7] have also used a tuple-by-tuple routing approach to adaptively decide the execution plan of a query. Shneidman et al [13] also identify the need to combine query optimization and operator deployment, focussing on join ordering. We consider a wider range of range of query planning options; in particular, we examine multi-query optimization, which presents other optimization opportunities (including operator reuse, delayed filtering and so on). Moreover, our bottom-up and the top-down planning are a different approach than the cost-space heuristics proposed in [13].

We described the query-optimization problem in distributed data-stream systems and illustrated how traditional database paradigms may not apply to such systems. We presented approximation-based approaches - the *Top-Down* approach and the *Bottom-Up* approach. We presented some initial simulation results that show that both approaches offer costs that are comparable to that provided by an exhaustive search while exploring fewer plans. We intend to perform further evaluations of the two approaches, using both the existing simulations and with real prototypes. In ongoing work we are exploring more sophisticated utility formulations and further optimization opportunities achievable through delayed filtering and query containment.

References

[1] inTransit. *http://www.cc.gatech.edu/~vibhore/inTransit/*.

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

[3] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM TODS*, 29(1), 2004.

[4] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD '00*, 2000.

[5] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Load management and high availability in the MEDUSA distributed stream processing system. In *SIGMOD '04*, 2004.

[6] R. Chirkova and C. Li. Materializing views with minimal size to answer queries. In *PODS*, 2003.

[7] A. Deshpande and J. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[8] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD '78*.

[9] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[10] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03*, pages 563–574.

[11] S. Seshadri, V. Kumar, and B. F. Cooper. Optimizing multiple queries in distributed data stream systems. (extended version). *http://www.cc.gatech.edu/~sangeeta/QueryOpt.pdf*.

[12] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Technical Report CS-02-1205, U.C. Berkeley*, 2002.

[13] J. Shneidman, P. Pietzuch, M. Welsh, M. Seltzer, and M. Roussopoulos. A cost-space approach to distributed query optimization in stream based overlays. In *NetDB '05*.

[14] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS '05*.

[15] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD '02*.

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