Efficient in-network evaluation of multiple queries

Vinayaka Pandit[1] and Hui-bo Ji[2]

[1] IBM India Research Laboratory, email:pvinayak@in.ibm.com
[2] Australian National University, email: hui-bo.ji@anu.edu.au

Abstract. Recently, applications in which relational data is generated in a distributed and streaming manner have emerged from diverse domains. Processing queries on such data has become very important. In-network evaluation of a query is a technique in which the query is evaluated in the network without transferring all the data to a central location. So far, algorithms for in-network of evaluation of a single query have been proposed. They are not designed to exploit common computations across multiple queries. There is a need to develop techniques for efficient in-network evaluation of multiple queries. We consider the problem of in-network evaluation of multiple queries on relational data generated on a distributed network of machines. We present a novel algorithm based on an algorithm for dynamic regrouping of queries.

1 Introduction

We are witnessing the emergence of a new class of applications in which the data is generated by data sources distributed over a large distributed network (often of the scale of WANs) [2]. Applications querying such data have become popular in diverse domains such as sensor networks, publish-subscribe systems, and financial services [16, 7, 1]. There are many applications in which the generated data is in relational form [11]. In this paper, we consider continuous versions of SQL queries over such distributed, relational, streaming data.

Platforms which expose a simple programming model to build applications which work on distributed, streaming, relational data are becoming popular [7, 14]. The user writes continuous version of an SQL query. The platform compiles the query into an equivalent execution tree. Each node of the execution tree represents a logical relational operator. In order to carry out computation, it can also communicate with its parents and children which could be executing at different locations of the network. Such an execution tree is called as *operator tree*. Note that placing the entire operator tree at one location is a valid way of evaluating the query which transfers the entire data to that location. *In-network evaluation* aims to efficiently evaluate the output of the operator tree by placing the nodes of the operator tree at appropriate locations in the network so as to minimize communication.

Note that, multiple queries can be represented by combining the individual operator trees corresponding to individual queries as one *global operator tree*.

Such a representation of multiple queries does not exploit the possibility of sharing common computations. In this paper, we consider the problem of evaluating multiple queries on a distributed network of machines by sharing common computations at appropriate locations in the network. We observe that, in order to share computation, the global operator tree has to be reorganized into an equivalent tree in which common sub-expressions are exploited. The main difference between our problem and the problem considered by recent papers on in-network evaluation of a single query is that they do not consider the possibility of reorganizing the global operator tree to achieve efficiency.

Ahmad and Cetintemel [1] consider the problem of in-network evaluation of a single operator tree on a distributed network. They consider the problem of minimizing the end-to-end delay. Furthermore, they assume that all the output tuples are delivered at a single location called the *proxy*. In comparison, we do not impose the restriction of accessing the output of all the queries from a single location. They propose a heuristic which traverses the operator tree in the post-order and at each step, places the currently visited node at one of the following locations: (i) one of its children's locations, (ii) the proxy location, and (iii) the node and the subtree below it are placed at a common location. The heuristic picks one of the three choices greedily. To focus on the communication aspect of the problem, they assume the processing cost of each node to be zero. Srivastava et.al [16] consider the same problem as Ahmad and Cetintemel in which the processing costs of the nodes can be non-zero. They consider the special case when the machines form a tree. They give an approximation algorithm for the problem based on dynamic programming. They view each operator as a filter and introduce the novelty of modeling network links as special filters. Each step of the dynamic programming involves solving a pipelined set-cover problem [9].

Pandit et.al [10] consider different efficiency measures for in-network evaluation of an operator tree. They formulate the end-to-end delay minimization as a a facility location problem and propose (and evaluate) a local search heuristic. They also consider the problem of computing load balanced placement in which the total communication across the network is minimized. They propose an algorithm based on minimum cost multi-way balanced partitioning of a graph.

Although in-network evaluation of multiple queries has not been studied before, multi-query optimization is a well studied problem [13, 12, 15] in database literature. Early work on multi-query optimization [12, 15] focused on finding the optimal query plan for a very small number of queries using exhaustive search techniques. These approaches are very expensive for large number of queries typically expected in the domain of applications highlighted before. Roy [13] proposes important heuristics to reduce the cost of the exhaustive search algorithms and applies these techniques for materialized view selection and maintenance. All these algorithms were designed for centralized databases. They do not deal with the trade-offs introduced by the network delays in the optimization. Furthermore, applications like financial services would require in-network evaluation of a large number of queries, thus resulting in a very large search space. So, there is a need to develop new techniques to solve the problem considered in this paper.

We observe that the problem of in-network evaluation of multiple queries has to deal with reorganizing the global operator tree in order to share computations as well as place the transformed operator tree efficiently on the network of machines. These two aspects of the problem are closely inter-related. We develop a framework motivated by an algorithm for dynamically regrouping queries proposed by Chen and Dewitt [3]. Our algorithm is based on a bottom-up traversal of the operator tree. At each level, it considers the possibility of exploiting common sub-expressions depending on whether the resulting tree can be placed efficiently on the network of machines.

2 Problem Formulation

2.1 Motivation

Consider two queries $Q1 = (A \bowtie B) \bowtie C$ and $Q2 = B \bowtie C$ which are represented as a global operator tree as shown in Figure 2(a). The inputs A, B, and C are the streaming relations, also called as *base relations*. Each box computes the join of its two input relations and its output is treated as a streaming relation. The output of a box which does not represent any of the queries, for example, the output of the box computing $A \bowtie B$, is called as *intermediate relation*. Suppose the cost of $A \bowtie (B \bowtie C)$ and $(A \bowtie B) \bowtie C$ are equal, then, the optimal way of evaluating the two queries on a single machine is to evaluate $Q2$ as $B \bowtie C$ and to evaluate $Q1$ as $A \bowtie (B \bowtie C)$ as shown in Figure 2(b). Multi-query optimization techniques presented in [13] are designed to exploit such common sub-expressions to optimize the performance of a centralized database.

In case of distributed query processing, there are other considerations which influence the optimization. Firstly, a given base relation may be *available* only at the location where it is produced. If it is involved in a computation at a different location, it has to be transferred there. Similarly, the results of a query has to be *attached* to the location of the end-user. The cost of transferring results to the attached location has to be considered as well. In general, we consider the base relations and the final queries as *tied* to specific locations in the network. As observed in the previous papers on in-network query processing [1], the communication cost dominates the cost of local processing. We consider the following example to illustrate the trade-offs introduced by the distributed setting which is absent in centralized setting.

Figure 1 shows two scenarios of evaluating the queries $Q1 = (A \bowtie B) \bowtie C)$ and $Q2 = B \bowtie C$ on a network of two machines, M_1, and M_2. In both the scenarios, the set of base relations and final queries tied to each machines is specified as shown. Consider Figure 1(a). Any attempt to share the computation of $Q2$ results in increased communication cost and it is beneficial not to reorganize the global operator tree. Whereas, in Figure 1(b), it is beneficial to reorganize the global operator tree to share the computation of $Q2$. Thus, the optimizations carried out are not only dependent on the common sub-expressions in the queries, but also depend on the way the relations and views are attached in addition to the network delays in transferring intermediate results.

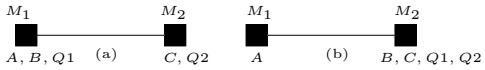

Fig. 1. Example Scenarios

2.2 Operator Tree and Topology Graph

The operator tree of a query is given by $Q = (J, R, I, O)$. R represents the set of base relations and O represents the different final views that the query computes. I represents the set of intermediate relations of Q. J is the set of operator nodes of the operator tree. Suppose $i_1 \in R \cup I$ and $i_2 \in R \cup I$ are the two inputs to a node $j \in J$, then the output of j is i_1 **Op** i_2 where **Op** is the relational operator corresponding to j. The incidence graph of the query is a tree. Suppose there are n user queries which have already been compiled into operator trees $Q_1, \ldots, Q_n$. Then, the main query Q is essentially the union of all the queries, $Q = \cup_{i=1}^{i=n} Q_i$. An equivalent query of Q is an operator tree $Q_F = (J_F, R, I_F, O)$ such that it computes the same set of final views as Q. The base relations in R are called as *producers* and the final views in O are called as *consumers*.

Chen et.al [4] showed that an effective heuristic for optimizing large number of continuous queries with similar join operator is to pull up the select operators over the join operators. As a first step towards efficient multi-query in-network evaluation we consider the problem of efficient evaluation of multiple queries consisting of only join operators [3], i.e, all the operator nodes in J are joins. It is a common practice in database literature to consider the operator tree to be *left-deep*. In our context, a tree is left-deep when at least one of the inputs of every node is a base relation. So, the inputs of a node at level i are a base relation, and the output of a node at level $(i-1)$. In our formulation, we assume $Q_1, \ldots Q_n$ to be left-deep while the main query Q itself may not be left-deep.

Suppose we represent a base relation by a unique symbol. Let us associate a string of symbols with the output of a join node as follows. If s_1 and s_2 represent the strings corresponding to the inputs of a join node $j \in J$, the output of j is denoted by $concat(s_1, s_2)$. Note that, in case of a left-deep tree, s_2 is a symbol corresponding to a base relation. So, a string of symbols can be used to unambiguously specify a left-deep tree. We shall use this way of conceptualizing an intermediate or a final view wherever it simplifies the exposition.

The distributed network of machines is given by $G = (V, E)$ where V is the set of machines and E is the set of communication links. Each link $e \in E$ has an attributed called *cost* which is an estimate of the delay across it. The cost of an edge between two nodes u, v is denoted by $c_{u,v}$ and by c_e when the edge in question is unambiguous. The producers and consumers are tied to the machines where they are generated and consumed respectively.

2.3 Cost Model

In traditional databases, statistics of the static tables are used for estimating cost. In our setting, we use the rate of arrival and rate of output as the main cost estimates. Suppose a node has two inputs with rates r_1, r_2 and a selectivity factor of s, then, the *processing cost* of the node is $r_1 \cdot r_2$ and its output rate is equal to $D(s \cdot r_1 \cdot r_2)$ where D is a function dependent on the scheduling policy at the node. Suppose a node $j_1 \in J$ with output rate r_1 is assigned to machine M_1 in G. Suppose the output of j_1 is input to a node $j_2 \in J$ which is assigned to machine M_2 in G. The *communication cost* of the data transfer between j_1 and j_2 is given by $r_1 \cdot l(M_1, M_2)$ where $l(M_1, M_2)$ denotes the length of the shortest path from M_1 to M_2. Given an assignment function $A : (J \in Q) \rightarrow (V \in G)$, the communication cost between nodes j_1 and j_2 is denoted by $CS_A(j_1, j_2)$.

Estimating the output rate of nodes is essential in order to compute costs. When the joins are *correlated*, i.e, the probability of two tuples joining at a node depends on the path they have taken, estimating the output rate is a hard problem [16]. Instead, we work with a simpler but, limited model. For every intermediate view in the global tree, we assume that its join selectivity with each of the base relations is given. The input has to specify $|R| \cdot |I|$ explicit selectivity factors corresponding to the I intermediate views. This *augmented selectivity information* is sufficient for the purpose of our algorithm.

2.4 Objective Function

In-network query evaluation is specified by the global operator tree $Q = (J, I, R, O)$ and the topology graph $G = (V, E)$. The goal is to compute a global operator tree $Q_F = (J_F, I, R_F, O)$ equivalent to Q and an assignment function $A : J_F \rightarrow V$. The cost of a solution (Q_F, A) is given by

$$C(Q_F, A) = \sum_{s \in R, t \in O} \sum_{(j_1, j_2) \in SP(s,t)} CS_A(j_1, j_2).$$

$SP(s, t)$ denotes the shortest path from s to t. If a pair $(s \in R, t \in O)$ is not connected, then the communication cost between them is assumed to be zero. We call this problem as *distributed, continuous, multi-query optimization* (DCMQOPT). The problems considered in [16, 1, 10] differ from our formulation as they assume that Q_F will be same as Q. In other words, they do not consider reorganizing the operator tree to save processing and communication costs.

3 Algorithm

Previous algorithms for in-network query evaluation did not consider the trade-offs discussed in Section 2.1. Chen and Dewitt [3] considered the problem of sharing computations across multiple dynamic queries. They showed the efficacy of their approach when the number of queries is large. As our algorithm is motivated by their ideas, we briefly summarize their work.

A query (or an intermediate view) x is said to be a *sub-query* of another query y if x is defined over a subset of producers over which y is defined. For example, BC is a sub-query of ABC. In other words, y *contains* x. In an operator tree Q, if the output of a join node j is input to a join node p, then, the output of the node p contains the output of j. The node j is called a *child* of the node p. The children of the same parent are called as *siblings*.

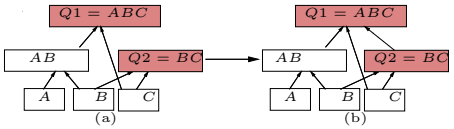

Fig. 2. Capturing sub-query relationships

Algorithm 1 briefly summarizes their approach. It is designed to exploit common sub-expressions in a top-down manner. The main component is the heuristic **Minimize_Graph**. It retains a minimum subset of views at level i which is sufficient to compute the set of views retained at level $i+1$.

Algorithm 1 Chen and Dewitt Algorithm

Pass 1

1: **for** $i = 2$ to num_level **do**
2: Reflect all the sub-query relations between the nodes at level i and level $i+1$ as shown by the dotted edge from $Q2$ to $Q1$ in Figure 2.
3: **end for**

Pass 2

1: **for** $i = num\_level - 1$ to 2 **do**
2: Use a heuristic called *Minimize_Graph* to retain a minimum subset of relations, say, I_i at level i such that every relation at level $i+1$ has a sub-query in I_i.
3: **end for**

DCMQOPT poses a complex trade-off involving the processing cost and the communication cost. So, we consider an algorithmic approach which systematically decouples these two costs. Specifically, we are interested in traversing the operator tree in a specific order to identify common sub-expressions to save processing cost. At each stage, we share the identified common sub-expressions if only they can be computed in a communication-efficient manner. We first point out the difficulty in extending their algorithm in such a way.

Consider invoking a procedure to place the modified operator tree after step 2 in each iteration of Pass 2. The modified tree is accepted if the placement cost is lesser than before. But, when the set of views to be retained at level i is decided (so that all views at level $i+1$ can be computed), the set of views to

be retained at level $i-1$ is not yet decided. So, there is uncertainty about the inputs (and hence, input rates) to the transforms at level i. So, estimating the processing cost and communication cost (See Section 2.3) of the transforms at level i is difficult. Thus, during a top-down traversal, it is difficult to estimate the impact of the decisions made at each step.

Alternatively, let us consider a bottom-up traversal of the operator tree. When a decision is made on the set of views to be retained at level i is made, the same decision has already been made regarding views at all the levels below i. Now, it is easy to observe that the input rates of the transforms at all the levels can be computed using the augmented selectivity information (See Section 2.3). Thus, we can traverse the tree in such a way that the impact of the decisions made at each step (regarding the shared sub-expressions) on the overall cost can be reasonably estimated. The details are presented in Algorithm 2.

Algorithm 2 Bottom-up Regrouping Placement(BRP)

1: $Q_C = Q$
2: **for** $i = 2$ to $num\_level - 1$ **do**
3: $Cost_1$ = Place_Single_Query(Q_C)
4: $Q_N = Q_C$
5: Add dotted edges between nodes in levels i and $i+1$ of Q_N which capture sub-query relations (as shown in Figure 2).
6: $F_i(Q_N) = F_i(Q_C)$
{$F_i(Q)$ denotes the set of final views of Q at level i.}
7: Let I_i^N be minimum cost subset of $I_i(Q_C)$ such that every node of Q_N at level $i+1$ has a sub-query in $R_i(Q_N) = I_i^N \cup F_i(Q_N)$.
{$R_i(Q)$ and $I_i(Q)$ are defined similar to $F_i(Q)$. In effect, the previous two steps are identifying common sub-expressions.}
8: For every node of Q_N at level $i+1$ add an edge to its cheapest sub-query in $R_i(Q_N)$.
9: $Cost_2$ = Place_Single_Query(Q_N)
10: **if** $Cost_2 < Cost_1$ **then**
11: $Q_C = Q_N$
12: **end if**
13: **end for**

In step 5, we exploit the existing sub-query relations by creating new dotted edges which represent shareable sub-expressions. After this step, there may be many alternative ways of computing a view at level $i+1$. However, the cost of taking each alternative can be estimated using the augmented selectivity information described in Section 2.3. In steps 7 through 9, we identify a set of beneficially shareable sub-expressions at level i.

Figure 3 shows the example considered in Section 2. For $Q = Q1 \cup Q2$, the two trees placed in steps 3 and step 9 are shown in the figure. If ($Cost_2 < Cost_1$), then the tree is modified to share the computation for $Q2$. Otherwise, the operator tree is not modified.

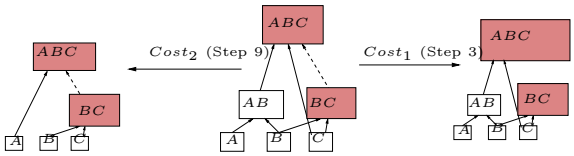

Fig. 3. Illustration of BRP Algorithm

Time Complexity. During an iteration, if there are x nodes at level i and y nodes at level $i+1$, then, Step 5 takes $O(xy)$ time. Thus, over all levels, Step 5 takes $O(n^3)$ time where n is the number of nodes in Q. At Step 7, we select a subset of non-final nodes I_i at level i so that all the nodes at level $i+1$ have a sub-query in $F_i \cup I_i$. The main trade-offs involved are (i) retaining a non-final node which is a sub-query of many nodes at $i+1$ and (ii) retaining non-final nodes whose output rates are not too high. This can be formulated as a *set cover* problem which is NP-Complete [5]. The natural greedy heuristic is known to give the best approximation and we use it to compute I_i. Over all levels it runs in $O(n^2 \log n)$ time. At Steps 3 and 9, we use the placement algorithm proposed in [1]. Note that our framework is not tied to any particular placement and any of the algorithms proposed in [10, 16] can also be used. Our algorithm runs in time $O(n^3 + n \cdot P(t))$ where $P(t)$ denotes the time taken by the placement module.

4 Experimental Evaluation

We now carry out empirical evaluation of our algorithm on a simulation framework developed by us. The framework can be used to generate operator trees which contain common sub-expressions so that they can be reorganized to share computation. It can be used to generate topology graphs with properties of large, distributed networks. We can tie streams and views to specific machines of the topology graph. Our algorithm is integrated with the framework. It also computes a normalized cost which is indicative of the average end-to-end delay.

On programming platforms like [7, 14], it is possible to obtain real instances of queries written by different users which contain enough computational redundancy. One of the disadvantages of our framework is that it is currently not integrated with such a platform. So, it has to generate instances of global operator trees in which it is possible to reorganize the tree to share computation across multiple final views. We briefly discuss how to generate such instances.

An operator tree can be thought of as a set of paths from the streams to final views. Each edge in these paths satisfies the sub-query relationship between its end-points. Intuitively, common sub-expressions are those intermediate or final views through which many paths can pass through. Consider an operator tree $Q = (J, R, I, O)$. Let T_O be the tree defined by the data flow from the streams in R to the final views in O via intermediate views I. Let $n_O = |I|$. Let T_A be the augmented graph on the nodes in $R \cup I \cup O$ reflecting the data flow of

the query and containing additional edges to reflect all the sub-query relations (refer to Section 3). Let $\mathcal{P}$ be a set of paths in T_A from the nodes in R to the set of final views in O such that they pass through as few intermediate nodes as possible. Let n_A be the number of intermediate nodes that paths in $\mathcal{P}$ pass through. An instance Q with lot of shareable computations satisfies the property that n_A is much smaller than n_O. We modify the algorithm by Melançon and Philippe [8] to generate random operator trees which contain sufficient computational redundancy.

We also generate random topologies for the network of machines on which the computed operator tree Q_F is placed. As mentioned in the introduction, the data sources are distributed over very large networks like WANs and the Internet. So, we are interested in generating random graphs which satisfies properties exhibited by such networks. We use an iterative algorithm similar to the Markov Chain simulation based generators proposed by Gkantsidis et. al [6] to generate the topology graphs with realistic edge costs for our experiments. It is possible to programmatically tie the streams and views of the operator tree to specific machines of the topology graph.

Our experimental set-up is as follows. We generate a random topology of machines and an operator tree which is known to be an instance that can be reorganized to share computations. The streams and the final views of the operator tree are tied randomly to different machines of the topology graph. We then compute placement by three algorithms: Random Placement(RP), Greedy Placement(GP) and our Bottom-up Regrouping Placement(BRP). In RP, every node of the operator tree is randomly assigned to one of the machines while the streams and final views are tied to their respective locations. In GP, the greedy placement algorithm proposed in [1] is used to compute the placement. BRP is our algorithm described in Section 3. A normalized integer score indicative of the end-to-end delay is computed for each placement. We present both experimental results and qualitative results. As the greedy heuristic was originally proposed for a single query with all the final views accessed from a single proxy, we discuss how we implement it for our purposes.

The main point to be emphasized about the algorithm in [1] is that it does not reorganize the operator tree. We modify the input so that it meets the input specification of their algorithm. In the operator tree, from every final view, we create a dummy operation whose output is marked as a final view. The original final views are now intermediate views tied to their respective machines. We add a new proxy machine into the topology graph. For an original final view v, let $m(v)$ denote the machine it is tied to. For every original final view v, we add an edge of cost zero from $m(v)$ to the proxy machine. This modified input meets the input specification of their algorithm.

Figure 4 shows a comparison of the three algorithms. The first column shows the details of the different operator trees considered for optimization. For each query Q, the number of nodes in the query, the number of base relations in the query, and the number of shareable common sub-expressions(CSE)s are mentioned. For each tree, the experimental results for topology graphs of different

sizes are given in the second column. Specifically, the cost of the objective function under the assignment computed by the three algorithms are given.

Observations: The experimental results in Figure 4 indicate the efficacy of our approach. Let us emphasize some important observations. On a single machine, sharing computations is always beneficial. The results on Q_3, Q_4, Q_5 show that on a single machine, we do get improved performance. On queries Q_3, Q_4 which contain high degree of redundant computations, our algorithm computes very efficient placements. It shows that in realistic scenarios of evaluating large number of queries with highly redundant computations, our algorithm can improve the performance significantly. Currently, we manually count the number of maximum shareable sub-expressions. Although, we have tested our algorithm on larger instances, we have presented results on only those cases for which we could compute the maximum number of shareable sub-expressions.

Operator Tree			
Tree	#Nodes in Q	#BaseRels	#CSEs
Q_1	11	5	1

Experimental Results			
#Nodes in G	RP	GP [1]	BRP
5	50	43	35
10	109	68	26
20	103	98	54

Operator Tree			
Tree	#Nodes in Q	#BaseRels	#CSEs
Q_2	16	6	3

Experimental Results			
#Nodes in G	RP	GP [1]	BRP
5	102	68	64
10	200	115	59
20	226	197	138

Operator Tree			
Tree	#Nodes in Q	#BaseRels	#CSEs
Q_3	26	7	3

Experimental Results			
#Nodes in G	RP	GP [1]	BRP
1	40	40	28
2	40	40	28
5	280	213	78
10	303	186	97
20	745	439	259

Operator Tree			
Tree	#Nodes in Q	#BaseRels	#CSEs
Q_4	31	10	8

Experimental Results			
#Nodes in G	RP	GP [1]	BRP
1	40	40	36
2	40	40	36
5	283	164	109
10	228	140	40
20	995	521	278

Operator Tree			
Tree	#Nodes in Q	#BaseRels	#CSEs
Q_5	41	16	4

Experimental Results			
#Nodes in G	RP	GP [1]	BRP
1	82	82	75
5	723	430	430
10	1204	547	488
20	4810	2803	2352

Fig. 4. Experimental evaluation of different algorithms.

In Section 2.1 we argued that, an optimal evaluation of multiple queries may chose not to share a sub-expression even though it can be shared. Qualitatively, it is important to verify that our algorithm does make such decisions on non-trivial examples. Figure 5(a) shows an example. The boxes represent transforms which compute views represented by the strings inside them. Views are indexed by the integers adjacent to the boxes. As shown in Figure 5(b), computation of the view $ABCDE$ can be reused to reduce the number of transforms by one. Figure 5(c) shows a network of six machines with edge costs and a specific attachment of streams and views of the example operator tree. In this case, any assignment which shares the computation of the view $ABCDE$ incurs a higher cost than an assignment which does not do so. Indeed, our algorithm chooses to keep the operator tree as it is. In this example, in addition to the augmented selectivity information, we input the selectivity of the join between AB and CD. On larger examples, our algorithm decides to share smaller number of sub-expressions than what is possible.

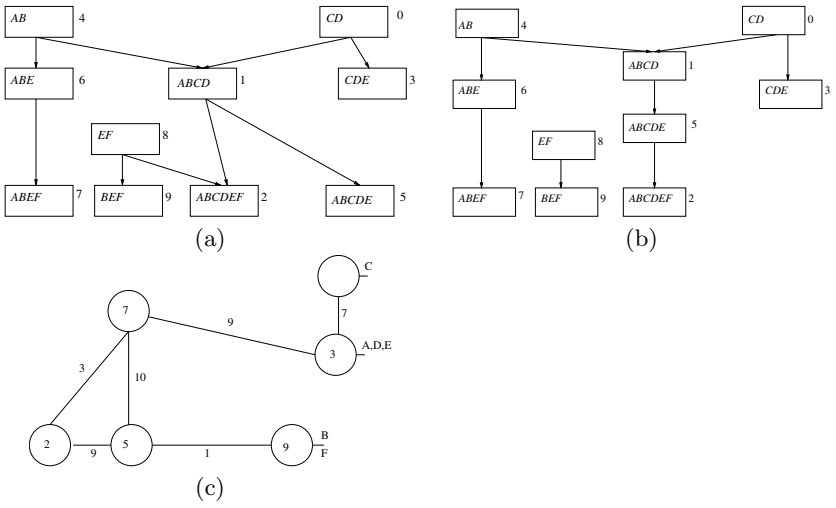

Fig. 5. An example where the algorithm chooses not to share computation based on the topology graph and how the streams and views are attached.

5 Future Work

We show that in-network evaluation of multiple queries involves dealing with issues which are not addressed by previous work on single query evaluation. We present a novel algorithm for in-network evaluation of multiple queries and provide empirical evidence of its efficacy. Our work can be extended in several

directions. Our algorithm needs to be extended when other relational operators are present and the selectivities of operator nodes are correlated. It also has to be extended for the case when individual query trees can be bushy instead of being left-deep. In future, we intend to demonstrate the efficacy of our algorithm in practice by integrating it with distributed publish-subscribe systems like [7].

References

1. Y. Ahmad and U. Cetintemel. Network aware query processing for stream based applications. In *Proceedings of Very Large Data Bases (VLDB)*, 2004.
2. G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content based publish-subscribe systems. In *Proceedings of the international conference on distributed computing systems*, 1999.
3. J. Chen and D. Dewitt. Dynamic regrouping of continuous queries. Technical report, University of Wisconsin-Madison, 2002.
4. J. Chen, D. DeWitt, and J. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.
5. U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.
6. C. Gkantsidis, M. Mihail, and E. Zegura. The markov chain simulation method for generating connected power law random graphs. In *ALENEX*, 2003.
7. Y. Jin and R. Strom. Relational subscription middleware for internet-scale publish-subscribe. In *DEBS*, 2003.
8. G. Melançon and F. Philippe. Generating connected acyclic digraphs uniformly at random. *Information Processing Letters*, 90(4):209–213, 2004.
9. K. Munagala, S. Basu, R. Motwani, and J. Widom. The pipelined set-cover problem. In *Proceedings of the International Conference on Database Theory*, 2005.
10. V. Pandit, R. Strom, G. Buttner, and R. Ginis. Performance modeling and placement of transforms for distributed stream processing. IBM Research Report, 2006.
11. B. Plale and K. Schwan. Dynamic querying of streaming data with the dquob system. *IEEE Transactions on Parallel and Distributed Databases*, 14(4), 2003.
12. A. Rosenthal and U. Chakravarthy. Anatomy of a modular multiple query optimizer. In *VLDB*, pages 230–239, 1988.
13. P. Roy. *Multiquery optimization and Applications*. PhD thesis, IIT Bombay, 2000.
14. R.Strom. Extending a content based publish-subscribe system with relational subscriptions. Technical report, IBM Research, 2003.
15. T. Sellis. Multiple query optimization. *ACM Transactions on database systems*, 10(3), 1986.
16. U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, 2005.