Query Optimization in Distributed Graph Pattern Matching

Kartik Venkatraman
Yale University
kartik.venkatraman@yale.edu

ABSTRACT
Representations of graphs in everyday use have grown tremendously in recent years due to the proliferation of social-networking applications and the need to maintain as much information as possible about customers and product users in order to better monetize web- and smartphone-based applications. In order to perform efficient sub-graph pattern matching over graphs distributed across several machines, we consider a dynamic programming query plan generation algorithm which minimizes network communication and data transfer between disparate nodes. We found that our dynamic programming linear plan could yield significant improvement in query latency over a greedy query plan baseline for queries that consisted of edge patterns where a particular vertex was the head of multiple directed edges.

1. INTRODUCTION
The graph structure is one of the most studied structures in mathematics and computer science. However, with the rise of social-networking applications and the corresponding astronomical increase in amount of data represented in a graph format, traditional approaches and algorithms for efficient computation on graphs have become increasingly outdated. These large datasets make in-memory analysis of graphs impossible, and even other graphs cannot be saved on a single disk. Indeed, only allowing computation to occur on a single machine limits significantly the scope of problem that can be solved. In this paper, we will examine initial experimental results for our system for optimizing query plans for sub-graph pattern matching in distributed graphs.

For many companies today, the internal graph data that they maintain is essential to their businesses, and efficient sub-graph pattern matching allows them to provide better services to their users and generate more revenue. Consider the case of Amazon.com, which uses information about purchase history to recommend additional products to buy for users when they are browsing the site. Amazon collects data on the sequence of products users may browse, using this information to determine the relatedness of each product. The order in which products on Amazon are browsed can be represented by a directed graph, where each vertex is a product, and a directed edge representing either a product following or preceding another product in a browsing sequence (the actual direction is irrelevant). As more individuals browse through the same set of products in a particular order, Amazon can determine with greater accuracy which products are related, and links can be weighted appropriately to reflect this property.

In the context of this product graph, identifying structural patterns demonstrate consumer buying patterns. If a particular product precedes two other products with equal frequency, those latter two products could be substitutes. The same might be said about two products that precede and follow each other with equal frequency. Amazon’s recommendations for products can become better and better by identifying particular patterns within their product graph.

For Amazon, the advantage of recommending relevant products is readily apparent. Customers browsing through Amazon considering buying a particular item are more likely to purchase a second item that is related to that first item. There is a direct effect on Amazon’s bottom line through this process. Users are thus spending money on multiple products instead of just a single product. Additionally, in the case of certain products that require another purchase (e.g., a lamp and a bulb), Amazon can likely ensure that the user makes both purchases at the same time, but more importantly, through Amazon, rather than later purchasing the necessary component from another retailer, likely not Amazon in order to obtain the item more quickly so that they can use their newly received product. Indeed, the recommendations also provide an enhanced user experience that may spur the users to become repeat Amazon customers as well, thus further increasing the revenue the online retailer can generate.

With graphs as large as Amazon’s related products’ graph, the sheer size of the representation is too large to be stored on a single disk; the data is necessarily distributed across several nodes. Edge information can be stored on one of several machines, and analyzing this information to identifying graph patterns often requires significant amounts of data transfer. For the purposes of our system, we will consider datasets with tuples stored as three values—a head vertex, an edge label, and a tail vertex. Our data will be stored across several nodes with data hashed upon the head vertex; all tuples with edge originating at a vertex with label “A” will appear on the same machine. Our primary cost considerations will be network traffic, so as to limit the amount of data transfer and allow as much of the query plan to be performed locally, on existing relational database management systems, so as to achieve the lowest query latency.

2. BACKGROUND
Our system has theoretical underpinnings in both graph pattern matching as well as query optimization. We will explain further these concepts here.

2.1 Sub-Graph Pattern Matching
While there are several variations of graph pattern matching [2], with the most prevalent being in terms of structural isomorphism [5], our primary consideration will be to accommodate semantic information. This information is usually represented in the form of types, labels, and attributes of the various vertices and edges; however, since types and labels can be easily converted to attributes, we will restrict ourselves to just attributed graphs. We will consider a data graph to be a directed graph with set of
vertices \( V \) and directed edges \( E \) where every edge in \( E \) connects two vertices contained in \( V \). There is a tuple of associated attributes for every vertex in \( V \), with attributes held constant for every vertex; there is a corresponding set of attributes for edges. A sub-graph is simply then a graph such that every vertex and edge is contained in the aforementioned sets \( A \) and \( E \). In identifying matches, we look for a one-to-one correspondence between a particular input pattern graph with specified vertex set, edge set, and both vertex and edge attribute values and any sub-graph of the working graph, with attributes considered equal over some specified equality operator.

2.2 Query Optimization

Query optimization is composed of three parts: Identification of the search space, an exploration of the search space to determine appropriate query plans, and finally cost estimations to determine the best plans.

2.2.1 Search Space

In general, given \( n \) relations, there are \(((2n-2))!/((n-1)!)\) different but logically equivalent join orders. To consider this in context, with 10 relations, the entire search space, only considering join orders, is comprised of 17.6 billion query plans [3]. The search space actually increases if additional properties, such as join implementation, are considered. Necessarily, thus, the search space must be restricted [1], with the possibility of skipping optimal plans. Even System R, the basis for many database systems today, explores only linear join orders, with only a small number, if any, “busy” or “balanced” plans considered [4]. For the purposes of this paper, we will also restrict ourselves to linear plans (explained further below).

2.2.2 Search Algorithms

Search algorithms can be deterministic or random, with the former consisting of brute force, greedy, or dynamic programming approaches and the latter including iterative improvement, simulated annealing [3] and genetic algorithms. For this paper, we will consider deterministic algorithms exclusively, implementing a greedy search as a baseline algorithm and a dynamic programming search as our optimized algorithm. Dynamic programming is discussed in System R, which assumes that an optimal plan of \( n \) joins is simply a single join added to an optimal plan of \((n-1)\) joins [4]. This greatly reduces the search space, from \( O(n!)\) to \( O(n^{2n})\) linear plans.

2.2.3 Cost Estimation

Once data is loaded into database systems, statistics are usually collected on the data stored itself. The type of information collected can include the number of tuples in a relation, the average size of each tuple, the variation in data (i.e., the set of distinct values) for a particular column or set of columns, and other potentially useful statistics. At the time the system receives a particular query plan, either partial or complete, the system can both predict statistics (such as those mentioned above) for the final output as well as the cost of the plan based on the statistics it maintains as well as constraining factors such as CPU time, disk access, memory use, as well as network communication.

For the purposes of our algorithms, the main cost consideration was network communication; we wished to limit the amount of data that was transferred between multiple machines.

3. SYSTEM IMPLEMENTATION

Our system was composed of two main parts: A query plan generator as well as a plan execution engine. Both parts of the system made assumptions about the types of joins available (explained below) as well as the fact that the main cost toward plan execution is network communication. We discuss these topics below.

3.1 System Architecture

Our system was implemented with a master-slave architecture. A single node took as input the sub-graph pattern as well as the number of nodes to find, and generated a query plan. After query plan generation, the plan executor would, following the steps of the query plan, execute the necessary queries and joins on every slave node. Each node had a PostgreSQL database instance on which to store its subset of the partitioned data. Queries were executed by invoking this instance when, and with the exception of the collocated join (explained below), data that needed to be transferred between machines to perform joins were transferred through system calls. Execution of queries was synchronous; only upon completion of a query on every node were data transferred between nodes. Hence, each type of join served as a “barrier.” The data were partitioned and hashed based on head vertex; for the rest of the paper we assume the data is partitioned as such.

3.2 Types of Joins

For the purposes of the experimental prototype we built, we considered four types of joins: Collocated Joins, Directed Joins, Broadcast Joins, and Hash Joins.

The Collocated Join is the easiest to reason about. A collocated join is a join that can be performed all on a single node. Consider a query that searches for heads that have multiple tails (A\(\rightarrow\)B, A\(\rightarrow\)C). If the data is hashed on the head vertex, then all edges for any particular head are stored on the same machine. Hence, a collocated join will allow for identification of all such vertices without any network communication. The actual implementation of a collocated join in our system was pushed to each individual machine’s database system, where a self-join was performed. This resulted in a highly optimized join.

A directed join is a join that requires output of one of its two sub-queries to be sent to other nodes. Each output tuple from that sub-query is first hashed and then directed to the corresponding node. Consider a query that searches for vertices that are both the tail of one edge connection and the head of another (A\(\rightarrow\)B, B\(\rightarrow\)C). In this case, two sub-queries can be performed to find, on every machine, every tuple that represents a relationship on the appropriate edge type. The output from the sub-query that finds tuples of type A\(\rightarrow\)B can be hashed on their tail vertex and sent to the appropriate machine; the result will be that every machine will have all the data that it needs to determine which vertices are both heads and tails for the edge type in question.

A broadcast join requires all output from one sub-query to be sent to all other nodes. The difference between a directed join and a broadcast join is that the intermediate output of the sub-query is not hashed and sent only to the one machine that a tuple is hashed on, but rather sent to all machines. Consider a query that aims to find all vertices that are the tail of two different heads on some edge type (A\(\rightarrow\)B, C\(\rightarrow\)B). This query can be satisfied with a broadcast join, with one of the sub-queries having all it’s data sent to every other node. As a result, all nodes will have all the
relevant data to find, for every tail, whether there exists multiple
heads that connect to it.

The final join to consider is a hash join. The hash join, like the
directed join, requires hashing of sub-query intermediate output
and movement to only one machine for every tuple. However,
unlike the directed join or broadcast join, intermediate output that
is the result of both sub-queries will be moved as necessary. An
example query that could require a hash join is the same query as
given as an example for the broadcast join. Note that in that query,
instead of sending all output from one sub-query to every
machine, the tuples from both sub-queries’ output can instead be
hashed on the tail vertex; thus all tuples with a particular tail will
be sent to the same machine, and thus each machine will have
enough data through which to find the tuples that match the
pattern.

In terms of relative cost, collocated joins are certainly the least
cost since they require no communication between machines.
Next is a directed join, as each tuple from a single sub-query from
every machine is sent to just a single other machine. Depending
on the number of machines and types of queries, hash joins are
less costly than broadcast joins. While hash joins send output
from two queries, each tuple from every sub-query is, like a
directed join, hashed and sent to just a single other machine.
Broadcast joins, on the other hand, require no hashing and only
movement of the output of one sub-query, but intermediate is sent
to all machines rather than just a single one. Note that in the
implementation of all these joins, the intermediate output from
sub-queries is written to disk, sent between machines (if
necessary), and once the necessary data are compiled, the data are
re-imported into the database system instance where the
database’s natural join is invoked.

3.3 Query Plan Generation

The main part of the system was the implementation of a dynamic
programming algorithm to generate performant query plans. In
order to evaluate the performance of these plans, we implemented
a greedy algorithm for query plan generation as a baseline to test
against. Both plans considered the amount of data to transfer as
the primary cost. Note that as a result, it was important to consider
the number of machines that data was partitioned over, as the
number of partitions influences the amount of communication
necessary in performing the joins. Pseudocode for the greedy
algorithm appears below:

function GreedyPlan(Q):
  e = edge in Q with the fewest matches
  plan = “run e;”
  while (there are still edges in Q):
    for each edge and join method:
      find edge e’ and join j with the min cost
      plan += “run e’; join using j;”
      remove e’ from Q

The greedy plan, as expected, considers each edge one at a time,
determines as long as there are still other edges to consider,
which join method to use between the two.

Our dynamic programming algorithm also generated a linear plan
for query execution, but looked more than just a single edge ahead
at a time. Pseudocode for the algorithm is included below:

function LinearPlan(Q):
  if (Q.triples != null):
    return;
  if (Q has only one unique head h):
    Q.triples = {(h, “run Q locally”)}
    return
  B = ∅
  A = LinearDecomposition(Q)
  for each decomposition (q1, q2) in A:
    FindLinearPlan(q1)
    FindLinearPlan(q2)
    B = B U ComputeLinearCosts(q1, q2)
    A’ + GenerateLayeredDAGs(Q)
  Q.triples = EliminateNonMinCosts(B)

function LinearDecomposition(D = (V, E)):
  A=∅
  Group edges in E on the head into subgraphs
  G1, G2, ..., Gn
  for each subgraph gi:
    A = A U {{g, E - g}}
  return A

function ComputeLinearCosts(q1, q2):
  A=∅
  for each triple (v1, c1, p1) in q1:
    for each triple (v2, c2, p2) in q2:
      for each common vertex in q1 and q2:
        if (v1, c1, p1) is in hash vertex and cost of j
        for each join method j in with (v1, c1, p2)
          P = “run p1; run p2; join them using j”
          A = A U {{v, C, P}}
  return A

The dynamic programming algorithm attempts to group sub-
graphs of the input pattern such that it can consider join orders
that the greedy plan does not. The way it performing this grouping
is by considering the head vertex of edges. That is, while the
greedy algorithm considers just the next edges, the dynamic
programming algorithm will find all edges that share the same
head. The intuition behind this decision is predicated on the fact
that our system stores data with data hashed on the head vertex.
By considering edges that share the same head vertex together, the
likelihood that data will need to be transferred at intermediate
steps becomes lower and lower. The system can push these self-
join queries to the single-machine database system, where queries
are well-optimized today. Additionally, just as single-machine
database systems try to limit the amount of tuples that are output
at any one step, so too here does the Dynamic Programming
algorithm. By performing self-joins early on in the query plan, not
only are the number of highly optimized queries that are
performed increased, but the intermediate output after the
completion of these queries has the potential to be much smaller
than intermediate output might be, depending upon the dataset.
Indeed, collocated joins can be relatively selective queries, and
hence data transfer can be limited by performing them earlier in
the query plan.

Note that the dynamic programming plan generates a linear plan;
there is only a single instance of two single-machine database
queries being joined together. No intermediate join will itself join
the result of two joins other joins. All intermediate joins will be
performed between the result of a join and a single database
query.
4. EXPERIMENTAL RESULTS

4.1 Experiment Environment
Experiments were conducted on a cluster of 10 machines, where each machine has a single 2.40 GHz Intel Core 2 Duo processor running 64-bit Red Hat Enterprise Linux 5 (Kernel Version 6.18) with 4 GB of RAM and two 250 GB SATA-I hard disks. According to htparm, the hard disks deliver 74MB/sec for buffered reads. All machines are on the same rack, connected via 1 Gbps network to a Cisco Catalyst 3750E-48TD switch.

4.2 Datasets
The dataset considered was of Amazon related product data, using Amazon product numbers. The dataset contained 548,552 vertices, with 1,788,725 edges.

4.3 Results
The results can be seen in Table 1.

Table 1: Execution Runtime Results

<table>
<thead>
<tr>
<th>Input Graph Pattern</th>
<th>Runtime (Greedy)</th>
<th>Runtime (DP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A→B, B→A, C→B, D→B</td>
<td>45.64s</td>
<td>46.24s</td>
</tr>
<tr>
<td>A→C, A→D, B→C, B→D</td>
<td>220.10s</td>
<td>35.46s</td>
</tr>
<tr>
<td>A→B, C→A, C→B, D→B, D→E, E→D</td>
<td>265.64s</td>
<td>182.47s</td>
</tr>
<tr>
<td>A→B, B→A, B→C, B→D, C→B, C→D, C→A</td>
<td>118.08s</td>
<td>90.27s</td>
</tr>
</tbody>
</table>

4.4 Runtime Analysis
As can be gleaned from looking through the runtime results for the above dataset and different query plan generation algorithms, the dynamic programming approach was a major improvement in two of the four sub-graphs input, while yielding only a small improvement on the final query without any improvement at all on the first query. We will explore the execution time for each of these input patterns and their corresponding query plans in detail below.

The first input graph is identifying patterns where three nodes point to a particular other node, which itself points back to (at least) one of those three. Table 1 shows the query plans for both the dynamic programming and greedy algorithms, with selected intermediate runtime information. For each of the tables displayed, “Execute X” refers to a particular SQL execution of the pattern-matching query X on each node’s PostgreSQL instance. For broadcast joins and directed joins, the wording following the hash vertex indicates which query output was sent to other nodes in the cluster. Note that runtimes for joins include not just the time taken to move data between machines but also to import data into the database instance and run the appropriate SQL query to handle the join specified with all data received.

For these two queries, the difference in runtime was not materially different, although the query plans did have some important differences. In the greedy plan, the hash join moved output from step (3) and step (4) to every machine. This seems like it would be significantly more work than the directed join in step (5) of the dynamic programming linear plan. However, an important observation is that the size of the output of (3), the result of the directed join, in the greedy plan is likely relatively small, as the cycle identification query is relatively selective. As a result, even though less data was transferred between each computer in the dynamic programming algorithm, the difference in amount was still relatively small; hence the dynamic programming algorithm was in actuality immaterially (1.3%) slower.

The second input graph created much more widely varying query plans, with the dynamic programming linear plan performing dramatically better (83.8%).

Table 2: Plan Comparison for Input Pattern 1

<table>
<thead>
<tr>
<th>Greedy Plan</th>
<th>Dynamic Programming Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Execute A→B</td>
<td>1. Execute C→B</td>
</tr>
<tr>
<td>2. Execute B→A</td>
<td>2. Execute B→A</td>
</tr>
<tr>
<td>3. Directed Join on A from (2) to (1) [8.69 s]</td>
<td>3. Directed Join on B from (1) to (2) [10.50s]</td>
</tr>
<tr>
<td>4. Execute C→B</td>
<td>4. Execute A→B</td>
</tr>
<tr>
<td>7. Directed Join on B from (6) to (5) [23.69 s]</td>
<td>7. Directed Join on B from (6) to (5) [21.46s]</td>
</tr>
</tbody>
</table>

Table 3: Plan Comparison for Input Pattern 2

<table>
<thead>
<tr>
<th>Greedy Plan</th>
<th>Dynamic Programming Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Execute A→C [0.59s]</td>
<td>1. Execute B→C; B→D [1.91s]</td>
</tr>
<tr>
<td>2. Execute A→D [1.28s]</td>
<td>2. Execute A→C; A→D [1.70s]</td>
</tr>
<tr>
<td>4. Execute B→C</td>
<td></td>
</tr>
<tr>
<td>5. Hash Join on C [41.35s]</td>
<td></td>
</tr>
<tr>
<td>6. Execute B→D</td>
<td></td>
</tr>
<tr>
<td>7. Broadcast Join on D from (6) to (7) [171.47s]</td>
<td></td>
</tr>
</tbody>
</table>

Note that step (2) in the dynamic programming plan is the same as steps (1)–(3) in the greedy plan. However, due to our implementation that requires writing the intermediate output of queries, even a collocated join has significant overhead that prevents it from reaching the level of performance of an optimized database system query. The real advantage to the dynamic programming plan is that it performs a collocated join for vertex B (Step (1)) as well as for vertex A. As a result, a single hash join can move the intermediate output from each query to find the sub-graph matching the pattern. Note that the number of tuples moved...
in step (3) of the linear plan cannot be greater than the number of tuples moved in step (5) of the greedy plan as the output of step (1) in the linear plan is necessarily smaller than the output of step (4) in the greedy plan. This assertion is borne out in the runtimes, with the hash join in the dynamic programming plan running in about 75% of the time. The only network communication for the dynamic programming plan is of step (3), which is significantly less than the greedy plan.

As noted above, hash joins and broadcast joins are particularly expensive types of joins, and as noted above the hash join in the greedy plan is a more expensive hash join than that of the dynamic programming plan. The broadcast join is also fairly expensive because the output being moved (in step (6)) is as large of output that can be moved in this set-up—the entire edges table stored on every machine. Additionally, the runtime included for the broadcast join include the cost of join within the database system of these large tables, which themselves are expensive.

The third input graph also produced outputs that were significantly different in runtime, with the linear dynamic programming query plan being approximately 31.3% faster. The plans are as follows:

<table>
<thead>
<tr>
<th>Table 4: Plan Comparison for Input Pattern 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy Plan</td>
</tr>
<tr>
<td>1. Execute A→B [0.54s]</td>
</tr>
<tr>
<td>2. Execute C→A [0.58s]</td>
</tr>
<tr>
<td>3. Directed Join on A from (2) to (1) [9.81s]</td>
</tr>
<tr>
<td>4. Execute C→B [0.53s]</td>
</tr>
<tr>
<td>5. Directed Join on C from (3) to (4) [14.29s]</td>
</tr>
<tr>
<td>8. Execute D→E</td>
</tr>
<tr>
<td>9. Broadcast Join on D from (8) to (7) [88.15s]</td>
</tr>
<tr>
<td>10. Execute E→D</td>
</tr>
<tr>
<td>11. Broadcast Join on D from (10) to (9) [129.15s]</td>
</tr>
</tbody>
</table>

The main difference between the dynamic programming plan and the greedy plan for this input sub-graph was the fact that the number of tuples being transferred was significantly less in the dynamic programming plan, and the trade-off in terms of computation was not significantly higher. Through step (3), the greedy plan is actually ahead of the linear plan, because it has less computation to perform for step (2), and the directed join in (3) for the dynamic programming plan must handle a larger table on the back-end of the join implementation when it joins the output of (1) and (2). Then, step (4) of the greedy plan will also be computationally less expensive than the corresponding step of the linear plan, as again the output table of (4) in the dynamic programming plan will be larger than the corresponding output of the greedy plan, and again that cost appears in not just the data transfer but the PostgreSQL join between larger tables. However, after the completion of step (5), which is still faster in the greedy plan because we are comparing a small directed join and a large hash join, the greedy plan must still go through another hash join, which is still smaller than the hash join in (5) of the dynamic programming plan, as well as an extremely expensive broadcast join (again moving an entire table, the output of (8) in the greedy plan, to all machines). After the hash join in (5) for the linear plan, however, the only remaining communication step is a directed join on B, which will be a large directed join but still not nearly as expensive as two broadcast joins that must be performed with the greedy plan.

The final query plan did not yield as significant differences as some of the other queries, but provides important insight into our cost minimization algorithm. The dynamic programming plan ran approximately 23.6% faster with many fewer steps within the query plan and much less network communication.

<table>
<thead>
<tr>
<th>Table 5: Plan Comparison for Input Pattern 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy Plan</td>
</tr>
<tr>
<td>1. Execute A→B [0.80s]</td>
</tr>
<tr>
<td>2. Execute B→A [0.76s]</td>
</tr>
<tr>
<td>3. Directed Join on A from (2) to (1) [12.21s]</td>
</tr>
<tr>
<td>4. Execute B→C [0.78s]</td>
</tr>
<tr>
<td>5. Directed Join on B from (3) to (4) [11.48s]</td>
</tr>
<tr>
<td>6. Execute B→D [0.81s]</td>
</tr>
<tr>
<td>7. Collocated Join on B [5.02s]</td>
</tr>
<tr>
<td>8. Execute C→B [0.79s]</td>
</tr>
<tr>
<td>9. Directed Join on B from (8) to (7) [15.96s]</td>
</tr>
<tr>
<td>10. Execute C→A [0.70s]</td>
</tr>
<tr>
<td>11. Broadcast Join on A from (10) to (9) [32.95s]</td>
</tr>
<tr>
<td>12. Execute C→D [1.03s]</td>
</tr>
<tr>
<td>13. Broadcast Join on D from (12) to (11) [34.71s]</td>
</tr>
</tbody>
</table>

Even though the linear plan produced a plan with many fewer steps, it was still not significantly faster, certainly a surprising result. Looking through the runtimes, it may be surprising to realize that step (11) of the greedy plan finished less than 2 seconds after step (3) of the dynamic programming plan. However, a directed join dominated the cost of the last two steps of the dynamic programming plan, which was smaller than the cost of the broadcast join the greedy plan has to perform. Let us consider each of these query plans holistically. The dynamic programming plan was able to significantly reduce the network communication cost; however, it paid a price in terms of the cost of joining large tables in step (3). Suppose a vertex has five neighbors (is the head of five edge relationships). Thus that vertex will contribute five tuples to a query of the form X→Y. For a query of the form X→W; X→Y; X→Z, a vertex with five neighbors will contribute 5 * 4 * 3 = 60 tuples to the final output.
So depending upon the connectedness of the graph, the output size of these queries can become extremely large (if the vast majority of vertices had fewer than three neighbors, the output of this query would be extremely small). In steps (1) and (2) of the dynamic programming plan, we see just how expensive this self-join query is combined with the cost of writing the output to disk, with steps taking nearly ten times longer than the corresponding steps of the greedy plan. This plan demonstrates that although our primary cost consideration was network communication and data transfer, there are substantial costs associated with the joining of large tables, thus finding a balance between the two could yield even better results.

5. CONCLUSION
We found that our dynamic programming algorithm for query plan generation to minimize network communication could outperform the baseline query plan generated by a greedy algorithm which considered only a single edge at a time. Because our data were partitioned and hashed based on head vertex, the dynamic programming algorithm searched for vertices that served as the head of multiple edge relations to perform these queries in the form of a database system self-join with no network communication necessary as early in the query plan as possible.

The major costs to greedy algorithm-generated query plans were broadcast joins late in the execution. These joins would move large amounts of data to every machine, and then pay a penalty in terms of performing a natural join within the database system with a large tables. Because the greedy algorithm did not look to order the edges considered, a late broadcast join was unavoidable in order to obtain the correct output. The dynamic programming algorithm was able to reorder the relations in such a way that late broadcast joins were more unlikely; instead directed or hash joins became more common.

Our cost considerations however did miss a few important factors, and these were the main costs of the query plans generated by way of the linear dynamic programming plan. Depending upon the connectedness of the graph, the output of the early queries that were the result of database system self-joins can become extremely large. Although no network traffic is necessary to compute them, later joins executed within the database system become slower due simply to their size. Thus further using statistics to determine the relative size of the output will be important to improve upon these query plans.

6. FUTURE WORK
In the future we intend to further use more information about our dataset in order to obtain even better query plans. Additionally, we will examine not just linear but “bushy” plans as well, which will construct query plan trees that contain as intermediate nodes a join between the output of two joins, rather than simply a join between the output of a join and a new query. This could result in much less data transfer between machines as well as the potential for greater parallelizability.

7. ACKNOWLEDGMENTS
My thanks to Jiewen Huang for his efforts related to query plan generation, Daniel Abadi, Stanley Eisenstat, and the entire computer science department.

8. REFERENCES