Enumerating Subgraph Instances Using Map-Reduce

Foto N. Afrati #1, Dimitris Fotakis #2, Jeffrey D. Ullman *3

#*National Technical University of Athens, Greece*
1afrati@softlab.ece.ntua.gr
2fotakis@cs.ntua.gr

**Stanford University, California, US*
2ullman@gmail.com

***Abstract*— The theme of this paper is how to find all instances of a given "sample" graph in a larger "data graph," using a single round of map-reduce. For the simplest sample graph, the triangle, we improve upon the best known such algorithm. We then examine the general case, considering both the communication cost between mappers and reducers and the total computation cost at the reducers. To minimize communication cost, we exploit the techniques of [1] for computing multiway joins (evaluating conjunctive queries) in a single map-reduce round. Several methods are shown for translating sample graphs into a union of conjunctive queries with as few queries as possible. We also address the matter of optimizing computation cost. Many serial algorithms are shown to be "convertible," in the sense that it is possible to partition the data graph, explore each partition in a separate reducer, and have the total computation cost at the reducers be of the same order as the computation cost of the serial algorithm.**

I. Introduction

We address the problem of finding all instances of a given subgraph (the *sample graph*) in a very large graph (the *data graph*). The problem is computationally intensive, so we shall concentrate on algorithms that can be executed by a single round of map-reduce [2]. We investigate how to minimize two important measures of complexity. The first is the *communication cost*, i.e., how to hash the edges of the data graph to the reducers in order to minimize the total amount of data transferred from the mappers to the reducers. This problem is, in a sense, a special case of evaluating conjunctive queries or multiway joins on a single large relation, so our starting point is the algorithms for optimal evaluation developed in [1] and [3]. The second measure is the computation cost, which can be increased significantly when we move from serial algorithms to their parallel implementation. However, our techniques derive a parallel algorithm of the same complexity as the serial algorithm.

A. Applications

Finding occurrences of particular sample graphs in a social network is a tool for analyzing and understanding such networks. For instance, [4] shows how the stage of evolution of a community can be related to the frequency with which certain sample graphs appear. Our results apply directly to problems of these types. Similarly, [5] discusses how discovering instances of sample graphs supports work in Biomolecular networks.

Another example application concerns analysis of networks for discovering potential threats or discovering recommendations. In these applications, the edges of the network are labeled and possibly directed (e.g., "buys from" or "knows"), and the goal is to find sets of individuals with specific interconnections among them (see e.g., [6], [7]). For example, aiming to discover potential threats, one may want to answer questions like "find all instances of five people booked on the same flight each of whom has bought explosive materials in the past three months." Our methods can be extended to this sort of problems as well, although there are two relatively simple extensions needed.

1) The cited papers assume that the query (sample graph) specifies at least one node (individual) of the data graph. As a result, optimum algorithms for evaluation will surely start by searching from the fixed node or nodes. However, eventually, this search will lead to a neighborhood that is sufficiently large that sequential search no longer makes sense. At that point, our methods can take over with what remains of the sample graph after removing nodes that have been explored on the data graph.
2) We assume that edges are unlabeled. However, a graph with labeled edges can be represented by a collection of relations, one for each label. Search for instances of a sample graph can still be expressed as a conjunctive query, and the same techniques applied.

A discussion about the importance of using the map-reduce environment for finding subgraph instances in large data graphs can be found in [8].

B. Measures of Complexity

There are two ways to measure the performance of map-reduce algorithms.

1) *Communication cost* is the amount of data transmitted from the mappers to the reducers. In the algorithms discussed here, edges of the data graph are replicated; i.e., they are associated with many different keys and sent to many reducers.

2) *Computation cost* is the total time spent by all the mappers and reducers. In the algorithms to be discussed, the mappers do nothing but assign keys to the input (the edges of the data graph), so their computation cost is proportional to the communication cost. We shall therefore discuss only the computation cost at the reducers in this paper.

These measures and their relationship are discussed in [9].

Another measure we address is the "number of reducers" used by different algorithms. What we are actually measuring is the number of different keys, and this quantity is an upper bound on the number of Reduce tasks that could be used. Lowering the number of reducers is not necessarily a good thing, but the communication cost for all the algorithms discussed grows with the number of reducers. See Section II-D, where we show how algorithms that are parsimonious in their use of reducers can lead to lower communication cost when the number of reducers is fixed.

C. Outline of the Paper and Contributions

This paper is the first to offer algorithms for enumerating all instances of an arbitrary sample graph in a large data graph, using a single map-reduce round. We combine efficient mapping schemes to minimize communication cost with efficient serial algorithms to be used at the reducers. Throughout the paper:

1) The data graph is denoted G and has n nodes and m edges.
2) The sample graph is denoted S and has p nodes.

We first address the communication cost of algorithms for finding all instances of a sample graph in a data graph. Section II motivates the entire body of work. We apply the multiway join algorithm of [1] to the triangle-finding problem. Although multiways joins are frequently more expensive than a cascade of two-way joins, for the problem of finding triangles, or more generally instances of almost any sample graph, the multiway join in a single round of map-reduce is more efficient than two-way joins, each performed by its own round of map-reduce.

The balance of the paper deals with arbitrary sample graphs and is divided into two parts. First we look at communication cost beginning in Section III and then we address computation cost starting in Section VI.

In Section III we look at arbitrary sample graphs. We generate from a given sample graph a collection of conjunctive queries with arithmetic constraints that together produce each instance of the sample graph exactly once. We then use the automorphism group of the sample graph and the collection of edge orientations to simplify this collection, while still producing each instance only once.

Section IV covers the optimal evaluation of the conjunctive queries from Section III. We give a simple algorithm for minimizing the communication cost for a single conjunctive query, and then show how it can be modified to allow all the conjunctive queries to be evaluated in one map-reduce round. We show that combining the evaluation of all conjunctive queries into a single map-reduce job always beats their separate evaluation.

Then, Section V looks at the special case of enumerating cycles of fixed length. We give a method for generating a smaller set of conjunctive queries than is obtained by the general methods of Section III.

Section VI begins our examination of optimizing the computation cost of map-reduce algorithms. All the map-reduce algorithms we discuss involve partitioning the nodes and edges of the data graph into subgraphs and then looking for instances of the sample graph in parallel, in each subgraph. Thus, the key question to address is under what circumstances a serial algorithm for finding instances of a sample graph S will yield a map-reduce algorithm with the same order of magnitude of computation. We call such an algorithm *convertible* and give a (normally satisfied) condition under which an algorithm is convertible.

It turns out that all sample graphs have convertible algorithms. The real question is what is the most efficient convertible algorithm. The results of [10] give worst-case lower bounds for the running time of serial algorithms, and we shall show in Section VII that these lower bounds can be met by convertible algorithms (Theorem 7.2) . Moreover, when we limit the degree of nodes in the data graph, we can obtain more efficient, yet still convertible, algorithms (Theorem 7.3).

In this extended abstract, we present our results accompanied with as much technical justification as the space constraints permit. The interested reader may find any omitted technical details in [11].

D. Related Work

In the 1990's there was a considerable effort to find good algorithms to (a) detect the existence of cycles of a given length and/or (b) count the cycles of a given length [12]. A generalization to other sample graphs appears in [13]. These problems reduce to matrix multiplication. However, enumeration of all instances of a given subgraph cannot be so reduced. Probabilistic counting of triangles was discussed in [14]. More recently, there has been significant interest in probabilistic (approximate) counting of small sample graphs on large biological and social networks. To this end, Alon et al. [5] applied the color-coding technique, and obtained a randomized approximation algorithm for counting the occurrences of a bounded-treewidth sample graph with p nodes on a data graph with n nodes in $O(2^{O(p)}n)$ time. Subsequently, Zhao et al. [15] showed how the approach of [5] can be parallelized in a way that scales well with the number of processors.

Enumeration of triangles has received attention recently. It was the subject of the thesis by Schank [16]. Suri and Vassilvitskii [17] give one- and two-round map-reduce algorithms for finding triangles. In this paper, we begin with an improvement to their one-round algorithm, obtained by the use of multiway joins, in Section II, and then give the extensions needed for arbitrary subgraphs.

In [18] the triangle finding problem in map reduce is experimentally studied; actually, this paper implements a randomized counting algorithm for triangles. Finally, in a related problem, a few papers have investigated recently the question of counting the output of a multiway join [19] and finding a serial algorithm for computing optimally a multiway join [20]. The complexity of the algorithm is the same as the worst case maximum size of output of a multiway join in [19], where the exponent for the complexity of the algorithm can be obtained after solving a linear program. An open question would be whether this algorithm is convertible.

II. Triangles and Multiway Joins

In this section we see that the problem of finding triangles in a large graph using a single round of map-reduce is a special case of computing a multiway join. We begin by discussing the "Partition Algorithm," which is a recent idea that almost-but-not-quite implements a multiway join. Then, we show how to apply the technique of [1] for optimal implementation of multiway joins by map-reduce. Finally, we combine these ideas with those of Partition to get a method that works better than either.

A. The Algorithm of Suri and Vassilvitskii

[17] gives the *Partition Algorithm* for enumerating triangles using a single round of map-reduce. This method has the property that the total computation done by the mappers and reducers is no more than proportional to the computation that would be done by a serial algorithm for the same problem. However, as we show in Section II-D, the communication cost of Partition is almost, but not quite as good as one can do.

Partition works as follows. Given a data graph of n nodes and m edges, partition the n nodes into b disjoint subsets $S_1, S_2, \ldots, S_b$ of equal size. For each triple of integers $1 \leq i < j < k \leq b$ create a reducer R_{ijk}; thus there are $\binom{b}{3} = b(b-1)(b-2)/6$ or approximately $b^3/6$ reducers. The mappers send to each R_{ijk} those edges both of whose nodes are in $S_i \cup S_j \cup S_k$. Thus, each reducer has a smaller graph to deal with; that graph has $3n/b$ nodes and an expected number of edges m/b^2. The paper [17] shows that assuming a random distribution of the edges, the total computation cost of the mappers and reducers is $O(m^{3/2})$, which is also the running time of the best serial algorithm [16].

The communication cost for Partition can be calculated as follows. An expected fraction $1/b$ of the edges will have both their ends in the same partition, say S_i. This edge must be sent by the mappers to $\binom{b-1}{2} = (b-1)(b-2)/2$ of the reducers – the reducers corresponding to all the subsets of three integers that includes i. The remaining fraction $(b-1)/b$ of the edges have their ends in two different partitions, say S_i and S_j. These edges are sent to only $b-2$ reducers, those corresponding to the subsets of the integers that include both i and j. The total communication per edge between the mappers and reducers is thus

$$\frac{1}{b}(b-1)(b-2)/2 + \frac{b-1}{b}(b-2) = \frac{3}{2}(b-1)(b-2)/b$$

For large b, the total communication cost for all the edges is approximately $3bm/2$.

As we shall see, the small problem with the partition algorithm is the fact that some edges need to be copied too many times. This problem also shows up in the details of the algorithm, where certain triangles – those with an edge both of whose ends are in the same partition – are counted more than once, and the algorithm as described in [17] needs to do extra computation to account for this anomaly. In the variant we propose in Section II-C, all edges are replicated the same number of times, and the communication cost is lowered from $3b/2$ to b per edge.

B. The Multway-Join Algorithm

In [1] the execution of multiway joins by a single round of map-reduce was examined, and it was shown how to optimize the communication cost. In fact, the case of finding triangles was considered in the guise of computing a simple cyclic join $R(X,Y) \bowtie S(Y,Z) \bowtie T(X,Z)$. In the case that the edges are unlabeled (the only case we consider here), the relations R, S, and T are the same; let us call it E. Then enumerating triangles can be expressed as evaluating the join $E(X,Y) \bowtie E(Y,Z) \bowtie E(X,Z)$.

There is an important issue that must be resolved, however: does an edge (a,b) appear as two tuples of E or as only one? If we use tuples $E(a,b)$ and $E(b,a)$, then in the join each triangle is produced six times. It is not hard to eliminate five of the copies; just produce (X,Y,Z) as an output if and only if $X < Y < Z$ according to a chosen ordering of the nodes. However, this approach is somewhat like counting cows in a field by counting the legs and dividing by 4. That's not too bad, but when we count subgraphs with larger numbers of nodes, we wind up counting centipedes or millipedes that way, and the idea cannot be sustained.

Thus, we shall assume an ordering $(<)$ of the nodes, and the tuple $E(a,b)$ will be in relation E if and only if (a,b) is an edge of the graph and $a < b$. In this case, each triangle is discovered exactly once.

Following the method of [1], to compute the join

$$E(X,Y) \bowtie E(Y,Z) \bowtie E(X,Z)$$

we must by symmetry hash each of the variables X, Y, and Z to the same number of buckets b. An ordered triple of buckets identifies a reducer. If we hash each variable to b buckets using hash function h, then there are b^3 reducers. If $E(u,v)$ is a tuple of E, this edge is sent by its mapper to $3b-2$ reducers in three groups:

1) Treated as an edge $E(X,Y)$, it is sent to the b different reducers whose triple is $[h(u), h(v), z]$ for any $z = 1, 2, \ldots, b$.
2) Treated as an edge $E(Y,Z)$, it is sent also to the b reducers $[x, h(u), h(v)]$ for any x.
3) Treated as an edge $E(X,Z)$, it is sent to the reducers $[h(u), y, h(v)]$ for any y.

However, regardless of whether or not $h(u) = h(v)$, exactly two of these reducers will be the same (see also [11, Section 2.2]). Thus the communication cost is $m(3b-2)$.

Each of the b^3 reducers computes the join for the tuples it is given. The triangle consisting of nodes u, v, and w, where $u < v < w$ is discovered only by the reducer $[h(u), h(v), h(w)]$. That is, we may substitute u for X, v for Y, and w for Z, and the tuples $E(u, v)$, $E(v, w)$, and $E(u, w)$ surely exist. However, if we make any other substitution of u, v, and w for X, Y, and Z, at least one pair of variables will be out of order and the corresponding tuple will not exist in E, although its reverse does.

C. Ordering Nodes by Bucket

We can improve the algorithm of Section II-B by exploiting the fact that the ordering of nodes is subject to our choice and thus can be related to the bucket numbers. Let h be a hash function that maps nodes to b buckets. When ordering nodes, think of node u as a pair consisting of $h(u)$ followed by u itself. That is, all nodes of bucket 1 precede all nodes of bucket 2, which precede nodes of bucket 3, and so on. Within a bucket, the name of the node breaks ties.

The advantage to this approach is that many of the lists of three buckets now correspond to reducers that get no triangles, and therefore we do not need reducers for these lists.[1] We only need a reducer for a list $[i, j, k]$ if $1 \le i \le j \le k \le b$. How many such lists are there? It is the same as the number of strings with $b-1$ 0's and three 1's, that is, $\binom{b+2}{3} = (b+2)(b+1)b/6$. In proof, we show that there is a 1-1 correspondence between the lists and the strings just described. Consider a string of 0's and 1's where the first 1 is in position p, the second 1 is in position q, and the third in position r. This string corresponds to $[i, j, k]$ where $i = p$, $j = q - 1$, and $k = r - 2$. Each list corresponds to a unique string, and each string corresponds to a unique list. Thus, like the Partition Algorithm, this method uses approximately $b^3/6$ reducers.

As with the algorithm of Section II-B, each reducer handles the portion of the data graph that it is given. A triangle is discovered by only one reducer – the reducer that corresponds to the buckets of its three nodes, in sorted order.

We claim that the communication cost is b per edge. Let (u, v) be an edge of the graph. This edge must be sent to all and only the reducers corresponding to the sorted list consisting of $h(u)$, $h(v)$, and any one of the buckets from 1 to b. Note that some of these lists have repeating bucket numbers, as must be the case since some triangles have two or three nodes that hash to the same bucket.

The argument that [17] used to show that the map-reduce implementation of Partition has the same order of computation time as the serial algorithm also works for this algorithm. The serial algorithm takes $O(m^{3/2})$ time on a graph of m nodes. If we hash nodes to b buckets, each of the m edges goes to b reducers. There are $O(b^3)$ reducers, so each reducer gets an expected $O(m/b^2)$ edges. The time this reducer will take to find triangles in its portion of the graph is $O\big((m/b^2)^{3/2}\big) = O(m^{3/2}/b^3)$. But since there are $O(b^3)$ reducers, the total computation cost of all the reducers is $O(m^{3/2})$, exactly as for the serial algorithm.

[1]Or, since it is likely that the number of reducers will be chosen first, our reasoning allows us to use less communication for a fixed number of reducers, as we shall see.

D. Comparison of Triangle-Finding Algorithms

The three algorithms are rather similar in critical measures, but the one given in Section II-C is best for communication cost by a small amount. First, let us assume that there are k reducers, and k is large enough that b plus or minus a constant can be approximated by b. Then Fig. 1 gives the communication cost for each of the algorithms. Using the same number of reducers, the algorithm of Section II-C beats the Partition Algorithm for communication cost by a factor of $3/2$ and beats the algorithm of Section II-B by a factor of $3/\sqrt[3]{6} = 1.65$. The computation costs for the three algorithms are similar, but only the Partition Algorithm finds some triangles more than once, requiring extra time to compensate for that effect.[2]

Algorithm	Buckets b	Communication Cost
Partition	$\sqrt[3]{6k}$	$3m\sqrt[3]{6k}/2$
Section II-B	$\sqrt[3]{k}$	$3m\sqrt[3]{k}$
Section II-C	$\sqrt[3]{6k}$	$m\sqrt[3]{6k}$

Fig. 1. Asymptotic performance of three triangle-finding algorithms

For a comparison using specific values of b, note that $216 = 6^3$ and $220 = \binom{12}{3}$. Figure 2 makes the comparison, using 216 reducers for the algorithm of Section II-B and (almost the same number of) 220 reducers for the other two algorithms. We see that the asymptotic comparison holds up for these reasonable numbers of reducers.

Algorithm	Buckets b	Reducers	Communic. Cost
Partition	12	220	$13.75m$
Section II-B	6	216	$16m$
Section II-C	10	220	$10m$

Fig. 2. Comparison of the three algorithms. For Partition, the 12 "buckets" is the number of sets into which the nodes are partitioned. The constant 13.75 in the communication cost is $\frac{3}{2}(b-1)(b-2)/b$ for $b = 12$. The constant 16 for the algorithm of Section II-B is $3b - 2$ for $b = 6$.

III. Sample Graphs and Conjunctive Queries

When we consider finding instances of sample graphs more complex than triangles, the multiway-join approach continues to apply and to be superior to a cascade of two-way joins. However, the joins must be constrained by arithmetic comparisons among nodes, to enforce certain node orders. Those constraints in turn are needed to avoid producing an instance

[2]While it is true that the difference in communication cost is only a constant factor, note that there is no "big-oh" involved. For each of the algorithms, the key-value pairs are the same; they consist of a list of three bucket numbers, an edge, and an indication of between which of the three pairs of nodes the edge lies.

of a sample graph more than once. A natural notation for such joins-plus-selections is conjunctive queries (abbreviated CQ) with arithmetic comparisons ([21] [22]), and we shall use this notation in what follows.

In this section we present our technique which, given a sample graph, constructs a number of conjunctive queries with arithmetic comparisons. These queries will be used to define the hashing scheme according to which edges are distributed to reducers in a way that minimizes the communication cost. The minimization of the communication cost will be achieved using the algorithm in [1].

Suppose we are searching for instances of a sample graph S. A CQ will have a variable corresponding to each node of S, and it will have a relational subgoal $E(X, Y)$ whenever S has an edge between nodes X and Y. Relation $E(X, Y)$ contains each edge of the data graph exactly once, and does so in the order $X < Y$; i.e., the node in the first argument precedes the node in the second argument according to some given order of the nodes. For the case of a general sample graph S, we construct CQ's for S by a three-step process:

1) Typically, the sample graph S will have a nontrivial automorphism group.[3] Thus, some orders of the nodes of S are automorphic to others, and the node orders fall into equivalence classes. Select one representative from each equivalence class.
2) For each chosen order of the nodes of S, write a CQ that uses subgoals E with arguments chosen to respect that ordering. The arithmetic condition for the CQ enforces the ordering.[4]
3) For most sample graphs S, there will be several selected CQ's that have the same orientation of all the edges of S; i.e., the relational subgoals of the CQ's will be identical, although the arithmetic conditions will differ. Combine CQ's with identical edge orientations by taking the logical OR of the arithmetic conditions.

REMARK. Regarding step (1) and the size of the automorphism group of S, we note that using such methods to mine data, e.g., for discovering potential threats or for providing recommendations, mostly involves answering questions with a lot of symmetry in them (see e.g. that mentioned in Section I-A). Hence, we expect that S will typically have a relatively large automorphism group, which is exploited by our approach. On the other hand, almost all very large random graphs are asymmetric, i.e., they have no nontrivial automorphisms. However, the sample graphs are typically very small, and most small graphs have nontrivial automorphism groups. For example, all graphs with 5 nodes have nontrivial automorphisms, and there are only 4 asymmetric graphs with 6 nodes (see e.g., [23]).

[3]An *automorphism* is a 1-1 mapping from nodes to nodes that preserves the presence of an edge.

[4]That is, we believe that E will contain only edges oriented in the direction given by the ordering $<$, but we do not rely on that assumption when expressing CQ's.

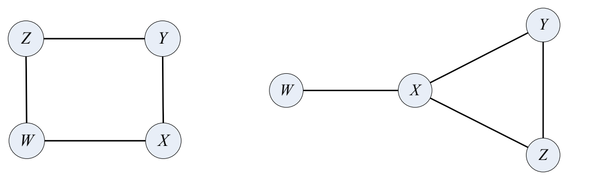

Fig. 3. The square and the lollipop.

A. Generating CQ's from Orderings

We begin with step (2), the simplest point. Given an ordering $X_1, X_2, \ldots, X_p$ for the nodes of the sample graph S, the CQ has:

1) A relational subgoal $E(X_i, X_j)$, if S has an edge (X_i, X_j) and $i < j$.
2) An arithmetic subgoal $X_i < X_{i+1}$, for $i = 1, \ldots, p-1$.

Example 3.1: Let us consider the square as our sample graph, with nodes labeled by variables as in Fig. 3. There are 24 orders for the four variables W, X, Y, and Z that label the nodes. However, as we shall see, three CQ's suffice to get all squares. Consider the order $W < X < Y < Z$. The body of the CQ corresponding to this order is

$$E(W, X) \ \& \ E(X, Y) \ \& \ E(Y, Z) \ \& \ E(W, Z) \ \& \\ W < X \ \& \ X < Y \ \& \ Y < Z$$

Notice that each of the four edges is represented by a subgoal E with the two arguments in the required order.

B. Exploiting Automorphisms

Suppose we wish to find all instances of a p-node sample graph S in data graphs. Then for each of the $p!$ orders of the nodes of S there is a CQ. This CQ has an E subgoal for each edge, with its two arguments in the order required. The arithmetic condition is that the variables are in the given order. In Example 3.1, we gave one such CQ when S is the square. There are 23 others.

However, often S will have a nontrivial automorphism group. If so, it is not necessary to use a CQ for each permutation. Rather, one CQ per element of the quotient group suffices. In fact, we can discover exactly once every instance of S in a data graph G by applying to G one CQ for each member of the group that is the quotient of the symmetric group of p elements (permutations of p things) with the automorphism group of S.

Example 3.2: The square has an automorphism group of size eight and a symmetric group of size 24. The automorphisms of the square are described by allowing a rotation of the square to any of four positions. Additionally, we can choose to "flip" the square (turn the paper on which it is written over) or not. For example, using the node names of Fig. 3 the automorphisms of the order $WXYZ$ are the identity, $XYZW$ (rotate 90 degrees clockwise), $YZWX$ (rotate 180 degrees), $ZWXY$ (rotate 270 degrees), and the four flips of these rotations: $WZYX$, $ZYXW$, $YXWZ$, and $XWZY$. Intuitively, these orders are those in which the four nodes form an increasing sequence in one direction around the square with any node as the starting point.

Since $24/8 = 3$, we expect there are two other sets of orders that are automorphic. One of these is the orders in which two opposite corners are each higher than the other two opposite corners. These orders are $WYXZ$, $YWXZ$, $WYZX$, $YWZX$, $XZWY$, $ZXWY$, $XZYW$, and $ZXYW$. The other group covers the cases where two opposite corners are the extreme values (low and high), and the other two nodes are in the middle. These are $WXZY$, $WZXY$, $YXZW$, $YZXW$, $XWYZ$, $XYWZ$, $ZWYX$, and $ZYWX$.

Pick representatives, say $WXYZ$, $WYXZ$, and $WXZY$, for each of the three groups. Then the three CQ's that together find each square exactly once are:

$$E(W,X) \;\&\; E(X,Y) \;\&\; E(Y,Z) \;\&\; E(W,Z) \;\&\; W < X \;\&\; X < Y \;\&\; Y < Z$$
$$E(W,X) \;\&\; E(Y,X) \;\&\; E(Y,Z) \;\&\; E(W,Z) \;\&\; W < Y \;\&\; Y < X \;\&\; X < Z$$
$$E(W,X) \;\&\; E(X,Y) \;\&\; E(Z,Y) \;\&\; E(W,Z) \;\&\; W < X \;\&\; X < Z \;\&\; Z < Y$$

Notice that all three have the subgoals $E(W,X)$ and $E(W,Z)$, but differ in the orders of the arguments of the second and third subgoals. Also, in the second and third CQ's, the arithmetic condition enforces a total order that is stronger than what the order of arguments of E implies.

C. Exploiting Edge Orientations

For some sample graphs S, the CQ's generated by the method of Section III-B will repeat some edge orientations. If so, these CQ's can be combined if we replace the arithmetic conditions from each by the OR of those conditions.[5] In Example 3.2 there were only three CQ's for the square, each with a different edge orientation.

However, for other sample graphs, such as the "lollipop," shown with names/variables for its nodes in Fig. 3, edge orientation allows significant simplification. Since the lollipop has four nodes, there are 24 orders, but its automorphism group has only two members: the identity and the mapping that swaps Y with Z. Thus, the quotient group has twelve members. We can break the symmetry of the automorphisms by requiring $Y < Z$. That inequality restricts an edge, so we would expect that there are eight orientations of the edges. However, two of these orientations are impossible, since we cannot have both $Z < X$ and $X < Y$, or there would be a contradiction with $Y < Z$. Thus, only six CQ's suffice.

Example 3.3: In Fig. 4 we see the twelve CQ's that come from the twelve orders with $Y < Z$. The three arithmetic subgoals that enforce the order for each CQ are omitted. Observe that all twelve have subgoal $E(Y,Z)$, as they must. However, the twelve divide into six groups with identical relational subgoals. These groups are summarized in Fig. 5. The orientations are represented by listing the low end of each edge first. For instance the first group corresponds to the edge orientation where $W < X$, $X < Y$, and $X < Z$.

[5]In some cases, the OR of arithmetic conditions cannot be expressed in the form needed for a conjunctive query. However, since we implement each CQ as a multiway join followed by a selection, and any selection condition, whether or not it is the AND of simple comparisons, can be implemented at the end of the Reduce function, we need not worry about the nature of the selection condition in what follows.

	Order	Conjuct. Query (Relational Subgoals Only)
1.	$W < X < Y < Z$	$E(W,X) \,\&\, E(X,Y) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
2.	$W < Y < X < Z$	$E(W,X) \,\&\, E(Y,X) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
3.	$W < Y < Z < X$	$E(W,X) \,\&\, E(Y,X) \,\&\, E(Z,X) \,\&\, E(Y,Z)$
4.	$X < W < Y < Z$	$E(X,W) \,\&\, E(X,Y) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
5.	$Y < W < X < Z$	$E(W,X) \,\&\, E(Y,X) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
6.	$Y < W < Z < X$	$E(W,X) \,\&\, E(Y,X) \,\&\, E(Z,X) \,\&\, E(Y,Z)$
7.	$X < Y < W < Z$	$E(X,W) \,\&\, E(X,Y) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
8.	$Y < X < W < Z$	$E(X,W) \,\&\, E(Y,X) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
9.	$Y < Z < W < X$	$E(W,X) \,\&\, E(Y,X) \,\&\, E(Z,X) \,\&\, E(Y,Z)$
10.	$X < Y < Z < W$	$E(X,W) \,\&\, E(X,Y) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
11.	$Y < X < Z < W$	$E(X,W) \,\&\, E(Y,X) \,\&\, E(X,Z) \,\&\, E(Y,Z)$
12.	$Y < Z < X < W$	$E(X,W) \,\&\, E(Y,X) \,\&\, E(Z,X) \,\&\, E(Y,Z)$

Fig. 4. Twelve CQ's for the lollipop

Orientation	CQ's
WX, XY, XZ	1
WX, YX, XZ	2, 5
WX, YX, ZY	3, 6, 9
XW, XY, XZ	4, 7, 10
XW, YX, XZ	8, 11
XW, YX, ZY	12

Fig. 5. Grouping CQ's for the lollipop by edge orientation

The first and last groups have only one CQ, so the first and twelfth CQ's are carried over intact. The second group consists of CQ's (2) and (5). Notice that their arithmetic conditions differ only in that (2) has $W < Y$ while (5) has $Y < W$. The logical OR of the conditions thus replaces these two inequalities by $W \neq Y$. Similarly , the fifth group $\{8, 11\}$ has conditions that differ only in the order of W and Z, so we replace the two inequalities on W and Z by $W \neq Z$ to obtain the logical OR.

Now consider the third group $\{3, 6, 9\}$. The three orders of variables for these CQ's all have $Y < Z < X$ and $W < X$. However, W can appear anywhere in relation to Y and Z. The OR of the three conditions is thus $Y < Z$, $Z < X$, $W < X$, $W \neq Y$, and $W \neq Z$. The fourth group, $\{4, 7, 10\}$ is handled similarly, except in this group X is lowest rather than the highest of the variables. Figure 6 shows the six resulting CQ's for the lollipop sample graph.

$$E(W,X) \;\&\; E(X,Y) \;\&\; E(X,Z) \;\&\; E(Y,Z) \;\&\; W < X \;\&\; X < Y \;\&\; Y < Z$$
$$E(W,X) \;\&\; E(Y,X) \;\&\; E(X,Z) \;\&\; E(Y,Z) \;\&\; W \neq Y \;\&\; Y < X \;\&\; X < Z$$
$$E(W,X) \;\&\; E(Y,X) \;\&\; E(Z,X) \;\&\; E(Y,Z) \;\&\; W < X \;\&\; Y < Z \;\&\; Z < X \;\&\; W \neq Y \;\&\; W \neq Z$$
$$E(X,W) \;\&\; E(X,Y) \;\&\; E(X,Z) \;\&\; E(Y,Z) \;\&\; X < W \;\&\; X < Y \;\&\; Y < Z \;\&\; W \neq Y \;\&\; W \neq Z$$
$$E(X,W) \;\&\; E(Y,X) \;\&\; E(X,Z) \;\&\; E(Y,Z) \;\&\; Y < X \;\&\; X < W \;\&\; W \neq Z$$
$$E(X,W) \;\&\; E(Y,X) \;\&\; E(Z,X) \;\&\; E(Y,Z) \;\&\; Y < Z \;\&\; Z < X \;\&\; X < W$$

Fig. 6. Six CQ's for the lollipop after combining CQ's with the same orientation

IV. Evaluation of CQ's with Optimal Communication Cost

We can apply the method of [1] to evaluate each of the CQ's for a sample graph S optimally as regards the communication cost. There are three broad approaches to doing so:

1) *CQ-Oriented Processing*. Perform a separate join for each CQ. This approach never dominates the others, but we shall begin our discussion of evaluation by focusing on a single CQ in Section IV-A.

2) *Variable-Oriented Processing*. Treat all the CQ's as if they were a single join of the relations for the edges of S. More precisely, if the subgoal $E(X, Y)$ appears in each CQ for S, then the relation for the edge (X, Y) is E. However, if both $E(X, Y)$ and $E(Y, X)$ appear among different CQ's for S, then the relation for the edge (X, Y) is two copies of E, one with the attributes in the order (X, Y) and the other in the order (Y, X). In that case, the relation for the edge (X, Y) is twice as large as it would be if the edge appeared in only one orientation among all the CQ's. Note that the reducers still evaluate each of the CQ's separately, although there might be some common subexpressions that can simplify the work.
3) *Bucket-Oriented Processing*. Here, we use the same number of buckets for each of the variables in each CQ. For each nondecreasing sequence of bucket numbers, evaluate each CQ using the edges whose ends are in buckets that appear in the sequence.

One might suppose that there is a fourth approach, where we treat E as a relation of undirected edges and include both $E(a, b)$ and $E(b, a)$. We then take a single multiway join and eliminate duplicate copies of instances of S by enforcing some constraints on the order of nodes in the instance. However, it is easy to see that this approach is never superior to the variable-oriented method and can be worse.

A. Optimization of Single CQ's

To review [1] and [3], the way to optimize the communication cost of the map-reduce evaluation of a CQ is by finding a hashing scheme as follows:

- For each variable of the CQ X, there is a *share* x that is the number of buckets into which values of X are hashed.
- Each reducer is identified by a list of bucket numbers, one for each variable of the CQ, in a fixed order.
- The communication cost for evaluating the CQ is a sum of terms, one for each relational subgoal of the CQ. This term is the product of the size of the relation for that subgoal and all the shares of the variables that *do not* appear in that subgoal.
- The minimum value of this sum occurs when the sums of certain subsets are all equal. There is one subset for each share, and that subset consists of all the terms in which that share appears.

Example 4.1: The first of the six CQ's in Fig. 6 is

$$E(W, X) \ \& \ E(X, Y) \ \& \ E(X, Z) \ \& \ E(Y, Z)$$

We have dropped the arithmetic comparisons. They will be implemented by a selection after performing the join, at the same reducers that produce the join, so they have no effect on the communication cost. There are four shares w, x, y, and z, corresponding to the four variables of the CQ. However, a theorem of [1] says that when one variable is *dominated* by another (the first variable appears only in terms where the second appears, then the share of the first may be taken as 1; i.e., the dominated variable may be ignored when determining the reducers to which a tuple is sent. As W appears only where X appears, we shall assume $w = 1$ and drop share w from formulas.

All four terms have the same relation E, so we shall use e as the size of E. Thus, all terms in the expression for communication cost will have e as a factor. The terms for our example CQ are thus:

$$eyz + ez + ey + ex$$

In explanation, the first term eyz comes from subgoal $E(W, X)$. It consists of the size e times the shares of the variables Y and Z that do not appear in the subgoal. The second term ez comes from the subgoal $E(X, Y)$. The missing variables are W and Z, but recall that the dominated W has $w = 1$ so there is no factor w needed. The last two terms are derived from the last two subgoals in a similar manner.

Now, we must derive the subsets of terms that are required to be equal at the optimum point. The share x is present only in the last term, so that term is one of the sums. Share y is present in the first and third terms, so another subset is $eyz + ey$. Share z appears in the first two terms, so the last subset is $eyz + ez$. The minimum communication cost thus occurs when

$$ex = eyz + ey = eyz + ez$$

From the above equalities, we deduce $z = y$ and $x = y^2 + y$. Since the number of reducers is the product of all the shares, we now can pick a value of y and know completely how to replicate edges to evaluate the CQ. For instance, pick $y = 5$. Then $x = 30$, $z = 5$, and there are $xyz = 750$ reducers. Each edge is replicated as a tuple for the first subgoal $E(W, X)$ to $yz = 25$ reducers. It is replicated 5 times as a tuple for the each of the second and third subgoals, and it is replicated 30 times as a tuple for the last subgoal, a total of 65 times.

B. Variable-Oriented Processing

In this approach, we treat all the CQ's as if they were one. That is, there is a reducer for each list of buckets, one for each variable. The number of buckets for different variables may differ.

Each CQ for a sample graph S has one subgoal for each undirected edge. It may be that for some of these edges, the orientation is the same in each CQ. For these subgoals, each edge is communicated from mappers to reducers in only one orientation, while for the other subgoals, each edge must be communicated in both orientations. The effect is that if e is the number of undirected edges, the relation size for a subgoal whose edge appears in both orientations among the CQ's is $2e$ rather than e.

Example 4.2: Consider the three CQ's

$$\begin{array}{c} E(W, X) \ \& \ E(X, Y) \ \& \ E(Y, Z) \ \& \ E(W, Z) \\ E(W, X) \ \& \ E(Y, X) \ \& \ E(Y, Z) \ \& \ E(W, Z) \\ E(W, X) \ \& \ E(X, Y) \ \& \ E(Z, Y) \ \& \ E(W, Z) \end{array}$$

derived for the square in Example 3.2 (the arithmetic subgoals are omitted). In these CQ's, the edges (W, X) and (W, Z) appear in only one orientation each, while the other two edges appear in both orientations. The expression for the communication cost is thus $eyz + 2ewz + 2ewx + exy$, where we conventionally use the corresponding lower-case letter to represent the share for a variable of the CQ. To solve for the shares, we must satisfy the equalities

$$2ewz + 2ewx = 2ewx + exy = eyz + exy = eyz + 2ewz$$

Interestingly, these equations do not provide unique values for the shares, even under the constraint that $wxyz = k$, where k is the desired number of reducers. However, we can derive the simple equalities $x = z$ and $y = 2w$. We are free to select values for the shares within these constraints; any choice will provide the same, optimum communication cost, even though the shares themselves differ. For instance, a simple choice would be $x = z = 1$, $w = \sqrt{k/2}$, and $y = \sqrt{2k}$. With that choice (or any other choice that satisfies $x = z$, $y = 2w$, and $wxyz = k$), the communication cost per edge is $4\sqrt{2k}$.

C. Bucket-Oriented Processing

While the method of Section IV-B determines the optimal number of buckets for each variable, this approach uses the same number of buckets, b, for each of the variables in each CQ. We thus lose the opportunity to optimize this number of buckets, but the compensating advantage is that each edge is distributed among the reducers in only one orientation. Which method is better depends on how far from optimal the choice of equal numbers of buckets is, and on how many subgoals among the set of CQ's have their arguments in both directions.

The bucket-oriented approach does the following:

1) Create a reducer for each nondecreasing sequence of p bucket numbers in the range 1 to b.
2) For each edge (u, v), hash u and v to buckets, using the hash function h. To determine which reducers get this edge, form a multiset of integers, starting with $h(u)$ and $h(v)$. Then, add $p - 2$ integers in the range 1 to b. These integers may duplicate $h(u)$ and $h(v)$ as well as each other. Sort the multiset to get a nondecreasing list, which corresponds to exactly one reducer. These are the reducers that receive a copy of the edge (u, v).
3) Let each reducer evaluate each of the CQ's for the given sample graph, using the edges it is given. Since every CQ has a total order of the variables, each solution for the CQ will be discovered by exactly one of the reducers.

While we cannot directly compare the bucket-oriented and variable-oriented methods, we can at least claim that the bucket-oriented method beats the generalization of Partition for arbitrary sample graphs. The advantage, however, decreases as p increases.

To this end, we first work as in Section II-C, and count the number of useful reducers by comparing them to certain binary strings. Thus, we can show that the number of reducers used by an application of the bucket-oriented method, using b buckets for a sample graph of p nodes is $\binom{b+p-1}{p}$. Similarly, we can count the number of reducers that receive each edge, which is equal to $\binom{b+p-3}{p-2}$. For large b, this count is approximately $b^{p-2}/(p-2)!$. The details of counting the number of reducers can be found in [11, Appendix B].

Let us compare this number with what we get by generalizing the Partition Algorithm. If instead we partition the nodes into b buckets and use reducers that correspond to sets of p groups (the obvious generalization of the Partition Algorithm), then edges going between nodes in two different groups are sent to $\binom{b-2}{p-2}$ reducers. However, an edge going between nodes of the same group are sent to $\binom{b-1}{p-1}$ reducers. Since $1/b$th of the edges are of the latter kind, the average number of reducers receiving an edge is, for large b, approximately

$$b^{p-2}\Big(\frac{1}{(p-2)!} + \frac{1}{(p-1)!}\Big)$$

Thus, the ratio of the communication cost per edge for generalized Partition, compared with that of the bucket-oriented algorithm is, for large b, equal to $1 + \frac{1}{p-1}$. This ratio is always greater than 1, although it approaches 1 as p, the number of nodes in the sample graph, gets large.

V. Conjunctive Queries for Cycles

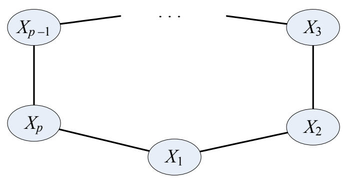

Fig. 7. A cycle of p nodes

In this section we consider an algorithm for finding all occurrences of the cycle C_p of length p. The strategy is based on the orientation of the edges. Intuitively, when we start with all orders of the nodes, use the automorphisms to reduce the number, and then further reduce the number of CQ's by clustering according to edge orientations, we do not avoid many of the possible orientations. If we start with the orientations only, then we can use the automorphisms effectively to cut down the number of orientations that actually need CQ's. However, the effect of automorphisms on edge orientations is more complex than their effect on node orders, so it is only in special cases such as cycles that we can get general rules for selecting CQ's by starting with orientations.

Imagine cycle C_p with nodes $X_1, X_2, \ldots, X_p$ arranged in a circle, with X_i counterclockwise of X_{i-1}, as suggested in Fig. 7. If $X_{i-1} < X_i$ we shall say the edge (X_{i-1}, X_i) is an *up edge* (designated by u) and otherwise it is a *down edge* (designated d). Likewise, if $X_p > X_1$ then the edge (X_1, X_p) is an up edge, and otherwise it is a down edge.

Every cycle can be oriented so that X_1 is lower than its neighbors. E.g., we could pick X_1 to be the lowest node on the cycle, but often there are other choices for X_1 as well. In general, an orientation of the edges counterclockwise around the cycle can be described by the runs of up and down edges. This sequence must begin with a run of up edges and end with a run of down edges, because of our assumption about X_1.

The sum of the run lengths is n, and there must be an even number of runs, because they begin with u and end with d.

Example 5.1: Consider the pentagon C_5. The possible sequences of run lengths are 14, 23, 32, 14, 1112, 1121, 1211, and 2111. These are all the sequences of positive integers that sum to five and have even length. The sequence 14 corresponds to the orientation of edges where, starting with X_1 and proceeding counterclockwise, we have orientations $udddd$. That is, $X_1 < X_2$, but $X_2 > X_3 > X_4 > X_5 > X_1$. Similarly, the other seven sequences of runs correspond to orientations $uuddd$, $uuudd$, $uuuud$, $ududd$, $uduud$, $uddud$, and $uudud$.

A. Automorphisms and Run Sequences

The cycle C_p has an automorphism group of size $2p$. This group is the product of the group of cyclic shifts (p elements) and the group of two elements "flip" and "don't flip" (the identity). Some run sequences are transformed into other sequences by these automorphisms. Because we insist that X_1 be lower than its neighbors, not every cyclic shift corresponds to another run sequence. However, if we rotate the run sequence by two (which may correspond to rotating the cycle by more than two positions), we get another node as X_1, and that node will also be less than its neighbors. To avoid double-counting of cycles, we want to eliminate a run sequence if it is a cyclic shift of another run sequence by an even number of positions.

Example 5.2: Thus, the orientations $ududd$ and $uddud$, in Example 5.1, are equivalent; each is a cyclic shift by two runs of the other. Thus, the CQ for either produces exactly the same instances of C_5 that the other does. Likewise, $uduud$ and $uudud$ are equivalent, and we can use either one. Let us therefore eliminate $uddud$ and $uudud$.

The automorphism in which we flip the cycle allows us to eliminate some of the run sequences. Flipping reverses the sequence of run lengths. Its effect on a sequence of u's and d's is twofold:

1) The sequence of u's and d's is reversed.
2) Then each u is replaced by d and vice-versa.

Example 5.3: The six orientations that remain after using the cyclic shifts of Example 5.2 are $udddd$, $uuddd$, $uuudd$, $uuuud$, $ududd$, and $uduud$. If we flip $udddd$, we reverse it to get $ddddu$ and then exchange u's and d's to get $uuuud$. That is, we can eliminate $uuuud$ in favor of $udddd$. Similarly, the flip of $uuddd$ is $uuudd$, so we can eliminate the latter. Finally, the flip of $ududd$ is $uudud$. The latter was already found to produce the same cycles as $uduud$, so we know that $uduud$ provides no cycles that $ududd$ does not provide. There are thus only three CQ's needed to find all pentagons, those corresponding to orientations $udddd$, $uuddd$, and $uduud$. These CQ's are, respectively:

$E(X_1,X_2)$ & $E(X_3,X_2)$ & $E(X_4,X_3)$ & $E(X_5,X_4)$ & $E(X_1,X_5)$ & $X_1 < X_2$ & $X_3 < X_2$ & $X_4 < X_3$ & $X_5 < X_4$ & $X_1 < X_5$
$E(X_1,X_2)$ & $E(X_2,X_3)$ & $E(X_4,X_3)$ & $E(X_5,X_4)$ & $E(X_1,X_5)$ & $X_1 < X_2$ & $X_2 < X_3$ & $X_4 < X_3$ & $X_5 < X_4$ & $X_1 < X_5$
$E(X_1,X_2)$ & $E(X_3,X_2)$ & $E(X_3,X_4)$ & $E(X_4,X_5)$ & $E(X_1,X_5)$ & $X_1 < X_2$ & $X_3 < X_2$ & $X_3 < X_4$ & $X_4 < X_5$ & $X_1 < X_5$

Notice that if we use the methods of Section III we wind up with seven CQ's rather than the three above. That is, there are 120 orders of five nodes. The automorphism group of the pentagon has size 10, so we start with 12 CQ's. However, if we choose these CQ's to all satisfy the constraints that X_1 is smallest and $X_2 < X_5$, then the CQ's group into seven orientations of the remaining edges (X_2, X_3), (X_3, X_4), and (X_4, X_5). Note that one of the eight orientations is impossible, because we cannot have $X_2 < X_5 < X_4 < X_3 < X_2$.

While we always can eliminate a run sequence that produces the same instances as some other run sequence, there are some run sequences that are automorphic to themselves. We cannot eliminate a sequence in favor of itself, so we are forced to find some other way to eliminate duplication. The number of times each cycle will be discovered by the CQ for a sequence is the number of flips and cyclic shifts (including the identity) that leave the sequence unchanged. We can avoid discovering a cycle more than once by adding the inequalities that make X_1 the smallest of all nodes, and also the inequality $X_2 < X_p$ to prevent a cycle and its flip from both being recognized. The problem and its solution can be seen by examining the hexagon, C_6.

Example 5.4: For the hexagon, there are five run sequences of length 2: 15, 24, 33, 42, and 51. The last two are obviously reversals of the first two, so we can eliminate them. But 33, which corresponds to the orientation $uuuddd$, will produce each hexagon that it produces twice, as any matching hexagon can be flipped.

There is only one run sequence of length 6: 111111, or $ududud$. This matches each matching hexagon six times, corresponding to zero, one, or two rotations of 120 degrees and/or flipping.

There are ten sequences of four runs. We can have one run of 3 and three runs of 1. Of these, 1113 is the reverse of 3111 and 1311 is the reverse of 1131, so only 1113 and 1131 need be considered. Other sequences of four runs have two 1's and two 2's. There are six such sequences, but when we eliminate reversals and rotations by two positions, we are left with only 1122, 1212, and 1221. There are thus seven sequences for which we must write CQ's: the three just mentioned plus 15, 24, 33, and 111111. All but the last two are straightforward. For 33, we need to force $X_2 < X_6$ to prevent flipping. For 111111, we need to force $X_1 < X_3$, and $X_1 < X_5$ to prevent rotation by 120 or 240 degrees, and we need to force $X_2 < X_6$ to prevent flipping. The resulting CQ is

$E(X_1,X_2)$ & $E(X_3,X_2)$ & $E(X_3,X_4)$ &
$E(X_5,X_4)$ & $E(X_5X_6)$ & $E(X_1,X_6)$ &
$X_1 < X_2$ & $X_3 < X_2$ & $X_3 < X_4$ & $X_5 < X_4$ & $X_5 < X_6$ &
$X_1 < X_6$ & $X_1 < X_3$ & $X_1 < X_5$ & $X_2 < X_6$

B. Algorithm for Finding Cycles Using Runs and Orientation

The algorithm for finding uniquely cycles of length p that formalizes the above examples is the following:

1) Find all bags (i.e., multisets) of an even number of positive integers that sum to p.

2) For each bag find all permutations of its elements and form set S_1 of permutations. From S_1 delete permutations to form its subset S, so that in S no permutation is a nontrivial cyclic shift, with optional flip, of another.
3) For each permutation in S (generated in the previous step) create the corresponding pattern of u's and d's; i.e., as you read the permutation, replace every integer by that number of u's or d's,, starting with u's and alternating u's and d's. This sequence of u's and d's tells us the order of arguments for the relational subgoals of the CQ for this permutation. The arithmetic subgoals enforce the relationship between adjacent nodes of the cycle, as are implied by the u's and d's. We then modify these CQ's as follows.
4) For each CQ created in the previous step do:
    a) If this CQ is not a palindrome and has no nontrivial periodicity, do nothing.
    b) If this CQ is a palindrome, add $X_2 < X_p$.
    c) If this CQ has nontrivial periodicity, add a number of inequalities (equal to p divided by the period, i.e., the length of the smallest repeated string) that say that x_1 is less than any of the other positions that are also less than both neighbors.

We claim that when the CQ's created by the algorithm are applied to a data graph, each cycle is discovered once. Moreover we claim that the group of CQ's found by our algorithm is minimum. The full proof of these claims can be found in the full version of the paper.

VI. Map-Reduce Computation Cost

We now turn to the second important measure of the quality of a map-reduce algorithm – the total computation cost at the mappers and the reducers. In each of the algorithms discussed, the computation at the mappers is proportional to the communication cost, so we shall ignore the mappers and focus on the computation at the reducers. This cost is polynomial in the size of the data graph, but the degree of the polynomial can be large, and the critical issue is how low can we make the degree of the polynomial.

A. Convertible Algorithms

Each of the methods we have described depends on a serial algorithm for finding instances of the same sample graph. This algorithm is applied to a smaller graph at each reducer. However, the relationship between the number of edges and the number of nodes in the graphs at each reducer generally differs from the node/edge relationship for the entire graph.

Definition 6.1: A *mapping scheme* is a function from input elements to sets of key-value pairs.

Definition 6.2: We call a serial algorithm $\mathcal{A}$ *convertible* with respect to a given mapping scheme if, given random input, when the algorithm is run at each reducer, the total computation cost at the reducers is, with high probability, proportional to the running time of $\mathcal{A}$ on a single machine. The constant of proportionality may depend upon the algorithm's characteristics but not on the number of reducers.

For algorithms that enumerate instances of a sample graph S, we consider only one mapping scheme. Assume that we hash nodes of S into b buckets, and the reducers correspond to lists of bucket numbers, one for each node of S. Thus, we have $O(b^p)$ reducers, where p is the number of nodes of S. Edges are sent from the mappers to those reducers whose lists include both nodes of the edge. Then the probability that a node appears in the subgraph of a given reducer is $O(1/b)$; the constant of proportionality is approximately the number of nodes in the sample graph. The probability that an edge appears in the sample graph at a given reducer is $O(1/b^2)$, since both its nodes must be hashed to buckets in the list for the reducer. Since data is random, skew is limited, and with high probability the reducers all get within a constant factor of the average numbers of nodes and edges.

If a serial algorithm for finding all instances of S on a graph of n nodes and m edges has running time $O(n^\alpha m^\beta)$, for some constants α and β, the computation performed by any reducer on a graph of $O(n/b)$ nodes and $O(m/b^2)$ edges is $O\big((n/b)^\alpha (m/b^2)^\beta\big)$. Thus, the total computation is $O\big(b^p (n/b)^\alpha (m/b^2)^\beta\big)$, or simplifying $O(b^{p-\alpha-2\beta} n^\alpha m^\beta)$. That is, the computation at the reducers is on the order of the computation of the serial algorithm on the original graph times $b^{p-\alpha-2\beta}$. If this exponent is positive, the total computation exceeds the computation of the serial algorithm. However, when the exponent is nonpositive, we have:

Theorem 6.1: If the best serial algorithm for finding all instances of a sample graph S with p nodes runs in time $O(n^\alpha m^\beta)$ on a data graph of n nodes and m edges, and $p \leq \alpha + 2\beta$, then there is a convertible algorithm for finding all instances of S.

For triangles, $p = 3$, $\alpha = 0$, and $\beta = 3/2$, so Theorem 6.1 applies. We pointed out this observation of [17] in Sec. II-A.

B. Decomposition of Sample Graphs

Let us call a serial algorithm with running time $O(n^\alpha m^\beta)$, where $\alpha \geq 0$ and $\beta \geq 0$, an (α, β)-algorithm. Given that the number p of nodes of the sample graph is a constant, any computation that depends only on the size of the sample graph can be ignored when discussing (α, β)-algorithms. An important consequence of this observation is that it is possible to decompose sample graphs, and the algorithms for discovering instances of the subgraphs can then be combined in a way that preserves convertibility.

In what follows, we shall assume that the data graph is preprocessed so that there is an index on pairs of nodes that lets us determine in $O(1)$ time whether there is an edge between any two given nodes. This index can be constructed in time $O(m)$, where m is the number of edges of the data graph, and surely any algorithm for finding instances of a sample graph will at least look at each edge of the data graph. Thus, the existence of this index will be assumed and the time to construct it can be ignored.

Now, suppose that the sample graph S has p nodes, and is partitioned into two sets of p_1 and p_2 nodes. We let S_1 and S_2 be the subgraphs of S induced by these two sets of nodes, and assume that S_i has an (α_i, β_i)-algorithm for $i = 1, 2$. Based on these, we present an $(\alpha_1 + \alpha_2, \beta_1 + \beta_2)$-algorithm for S.

We first use the two algorithms to enumerate all instances of S_1 and S_2 in the data graph. For each pair of instances:

1) Check that the nodes of the two instances are disjoint.
2) Check that for each edge in S that connects a node of S_1 with a node of S_2, the edge between the corresponding nodes of the two instances exists in the data graph.
3) Check that each instance of S is generated only once.

The total computation of the above steps for any pair of instances depends only on p. Step (1) clearly takes time $O(p)$. The index allows Step (2) to be carried out in $O(1)$ time per edge of S.

Step (3) is a little trickier. Whenever we generate an instance, we need to check that it is lexicographically first among all the ways that this instance can be generated from instances of S_1 and S_2. To this end, we assign an order to the nodes of the data graph G. Once we have identified an instance H of S in G, order the nodes of that instance according to the order for G. Form a string of 1's and 2's, where the ith position of the string is 1, if the ith node of H came from the instance of S_1, and 2 otherwise. Now, consider all other possible assignments of the nodes of H to the nodes of S, and construct their strings in the same way. Only if this construction of H is the lexicographically first among all these strings do we now generate instance H. Otherwise, H will be generated when we consider some other pair of instances of S_1 and S_2.

Thus, the total computation is proportional to the number of pairs of instances plus the time taken by the algorithms that enumerate the instances of S_1 and S_2. Each of the latter algorithms takes time $O(n^{\alpha_i} m^{\beta_i})$, and enumerates at most so many instances of S_i, $i = 1, 2$. Therefore, the entire algorithm for finding all instances of S takes time $O(n^{\alpha_1+\alpha_2} m^{\beta_1+\beta_2})$.

Now, we assume that each subgraph S_i has a convertible (α_i, β_i)-algorithm, and that $p_i \leq \alpha_i + 2\beta_i$, $i = 1, 2$. Then, $p = p_1 + p_2 \leq (\alpha_1 + \alpha_2) + 2(\beta_1 + \beta_2)$, and by Theorem 6.1, the entire algorithm for finding all instances of S is convertible. Thus, we conclude:

Theorem 6.2: Let the sample graph S be partitioned into subgraphs S_1 and S_2 with p_1 and p_2 nodes, respectively. If S_1 and S_2 have convertible algorithms with running times $O(n^{\alpha_i} m^{\beta_i})$ and $p_i \leq \alpha_i + 2\beta_i$, for $i = 1, 2$, then S has a convertible algorithm with running time $O(n^{\alpha_1+\alpha_2} m^{\beta_1+\beta_2})$.

Example 6.1: [10] shows that if a sample graph S can be decomposed into (i) pairs of nodes connected by an edge, and (ii) odd-length cycles (possibly with some additional edges), then in the worst case, the data graph has $\theta(m^{p/2})$ instances of S. Recursive applications of Theorem 6.2 let us show that there is a matching serial algorithm for every such sample graph, and moreover, the serial algorithm can be converted to a map-reduce algorithm with the same computation time. We have only to exhibit serial algorithms for the basis cases (edges and odd-length cycles). A pair of nodes with an edge is very easy; just enumerate the edges in the data graph in time $O(m)$. Thus, there is a $(0, 1)$-algorithm, and by Theorem 6.1 this algorithm is convertible. The case of odd cycles is trickier. We give the proof of the existence of an $O(m^{p/2})$ algorithm when S is a cycle of odd length p in Theorem 7.1.

VII. Optimal Convertible Algorithms for General Sample Graphs

Next, we show that the bounds of [10] can be met with concrete serial algorithms. These algorithms are all convertible, so they can be used in map-reduce implementations with minimal computation cost. The difficult part is the case of odd-length cycles, so we handle that first. In Section VII-B, we consider the restriction of the problem to the case where there is a degree limit for the data graph. We show that if no node of a data graph with m edges has degree higher than $\sqrt{m}$, then every connected sample graph of p nodes has a convertible serial algorithm with running time $O(m^{p/2})$.

In this section, we assume that before the described manipulations, the data graph has been processed to create two indexes. One is the index discussed in Section VI-B that lets us test whether an edge exists in $O(1)$ time. This index takes $O(m)$ time to create. The other index, also creatable in $O(m)$ time, lets us find, for each node, the set of adjacent nodes in time proportional to the degree of that node. Since all algorithms we discuss run in time at least $O(m)$, we shall neglect the cost of index creation and use.

A. Odd-Length Hamilton Cycles

For sample graphs which are cycles of odd length the following theorem holds.

Theorem 7.1: Let S be a p-node sample graph, with p odd, that contains a Hamilton cycle. Then, S has a convertible $(0, p/2)$-algorithm.

B. Optimal Convertible Algorithms

For general sample graphs, we combine Theorem 6.2 and Theorem 7.1 with the observation that an isolated node has a $(1, 0)$-algorithm. Since q plus twice $(p-q)/2$ is exactly p, this algorithm is convertible. Also, since it always pays to trade n^2 for m in the running time, we seek for a decomposition of S that minimizes the number of isolated nodes.

Theorem 7.2: Let S be a sample graph with p nodes. If S can be decomposed into node-disjoint subgraphs consisting of q isolated nodes, pairs of nodes connected by an edge, and graphs with an odd-length Hamilton cycle, plus possible edges, then S has a convertible $\big(q, (p-q)/2\big)$-algorithm.

[10] implies that the running time of Theorem 7.2 is essentially best possible for general data graphs. However, if S is connected and the data graph G has maximum degree Δ, we can obtain a stronger upper bound of $O(m\Delta^{p-2})$ on the running time. The idea is to partition S into two subgraphs, similarly to Section VI-B, and to apply a more careful analysis based on the maximum degree Δ. Specifically, we partition S in a subgraph S_1 consisting of a single node u that is not an articulation point of S, and a connected subgraph S_2 induced by the remaining nodes of S. Assuming inductively that S_2 has a $O(m\Delta^{p-3})$-time algorithm, we can check all possible

choices of u in G that may allow us to complete an instance of S in time $O(\Delta)$ per instance of S_2 (see also [11, Section 7.3] for the details). Thus, we have:

Theorem 7.3: Suppose data graphs are restricted to have maximum degree of at most Δ (which may be a function of the number of edges m), and let S be a connected sample graph with $p \geq 2$ nodes. Then S has an enumeration algorithm with running time of the form $O(m\Delta^{p-2})$.

In Theorem 7.3, the maximum degree Δ of the data graph can be a constant or any function of m and n. Therefore, for data graphs of constant maximum degree, all sample graphs have an $O(m)$-time enumeration algorithm, and for data graphs of maximum degree $O(\sqrt{m})$, all sample graphs have a $(0, p/2)$-algorithm. However, such an algorithm is convertible only if the maximum degree Δ of the data graph is large enough compared with the number b of buckets into which we hash its nodes (e.g., if $\Delta/b = \Omega(\log n)$). Then, as in Theorem 6.1, skew is limited, and each reducer processes a subgraph with $O(m/b^2)$ edges and a maximum degree of $O(\Delta/b)$. Since the number of reducers is b^p, the total computation is $O(b^p m/b^2(\Delta/b)^{p-2}) = O(m\Delta^{p-2})$.

Joins of binary relations of different sizes. For single binary relations, the bounds that we have given here are tight, in that for any size of the relation, the running time of our serial algorithm that enumerates all sample graphs meets, to within a constant factor, the lower bound (given in [10]). Hence it is both optimal as a serial algorithm and is also a convertible algorithm. However, when we have a multiway join over binary relations of different sizes, this is not the case as pointed out in [20]. It is an open question whether we can refine the bounds for this case to be more precise. In [11], we give a complete analysis for the case where the sample graph is a cycle of size 5 and binary relations are of different sizes. There we show the following: For the join $R_1(A,B)\&R_2(B,C)\&R_3(C,D)\&R_4(D,E)\&R_5(E,A)$, where each relation $R_i, i = 1,\ldots,5$ has n_i tuples the following are optimal bounds: i) $\sqrt{n_1 n_2 \cdots n_5}$, if $n_1 n_5 n_3 \geq n_2 n_4$ for all cyclic automorphisms of the cycle that defines the 5-way join and ii) $n_1 n_5 n_3$ if one of the conditions in (i) is not satisfied; wlog we assume $n_1 n_5 n_3 < n_2 n_4$. E.g., if $n_1 = 1, n_2 = n, n_3 = 1, n_4 = n, n_5 = 1$ then the upper and lower bound is equal to n.

VIII. Conclusions and Open Problems

The problem of enumerating instances of a sample graph in a huge data graph has many applications, including social networks, threat detection, and Biomolecular networks. We have, in this paper given algorithms that use a single round of map-reduce and are able to detect all instances of a given sample graph. These algorithms are efficient both in communication cost between mappers and reducers and in the computation cost at the mappers and reducers.

We have not addressed the case where nodes and/or edges have labels. Neither have we addressed the case of directed graphs. Many of the same techniques carry over in a straightforward way. For instance, we can still express the instances of a labeled, directed sample graph as a union of CQ's. The automorphism groups tend to be smaller, so the number of CQ's is greater, but the same methods for evaluating CQ's by a multiway join will work. We expect that there are provably convertible algorithms in all or almost every case, but the mapping schemes may require some thought.

Acknowledgements. This work was supported by the project Handling Uncertainty in Data Intensive Applications, co-financed by the European Union (European Social Fund) and Greek national funds, through the Operational Program "Education and Lifelong Learning", under the program THALES.

References

[1] F. Afrati and J. Ullman, "Optimizing multiway joins in a map-reduce environment," *IEEE Transaction of Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1282–1298, 2011.
[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[3] F. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. Ullman, "Cluster computing, recursion and datalog," in *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK. Revised Selected Papers*, LNCS 6702, 2011, pp. 120–144.
[4] S. Kairam, D. J. Wang, and J. Leskovec, "The life and death of online groups: Predicting group growth and longevity," in *WSDM*, 2012.
[5] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. 241–249, 2008.
[6] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian, "Cosi: Cloud oriented subgraph identification in massive social networks," in *ASONAM*, 2010, pp. 248–255.
[7] ——, "A budget-based algorithm for efficient subgraph matching on huge networks," in *ICDE Workshops*, 2011, pp. 94–99.
[8] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 29–41, 2009.
[9] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman, "Map-reduce extensions and recursive queries," in *EDBT*, 2011, pp. 1–8.
[10] N. Alon, "On the number of subgraphs of prescribed type of graphs with a given number of edges," *Israel Journal of Mathematics*, vol. 38, no. 1-2, pp. 116–130, 1981.
[11] F. Afrati, D. Fotakis, and J. Ullman, "Enumerating subgraph instances using map-reduce," *Stanford InfoLab*, vol. TR-1020, 2011.
[12] N. Alon, R. Yuster, and U. Zwick, "Finding and Counting Given Length Cycles," *Algorithmica*, vol. 17, pp. 209–223, 1997.
[13] M. Kowaluk, A. Lingas, and E.-M. Lundell, "Counting and detecting small subgraphs via equations and matrix multiplication," in *SODA*, 2011, pp. 1468–1476.
[14] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: counting triangles in massive graphs with a coin," in *KDD*, 2009, pp. 837–846.
[15] Z. Zhao, M. Khan, V. A. Kumar, and M. Marathe, "Subgraph enumeration in large social contact networks using parallel color coding and streaming," in *ICPP*, 2010, pp. 594–603.
[16] T. Schank, *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD Thesis, Universität Karlsruhe (TH), 2007.
[17] S. Suri and S. Vassilvitskii, "Counting Triangles and the Curse of the Last Reducer," in *WWW*, 2011, pp. 607–614.
[18] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *CoRR*, vol. abs/1103.6073, 2011.
[19] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," in *FOCS*, 2008, pp. 739–748.
[20] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms: [extended abstract]," in *PODS*, 2012, pp. 37–48.
[21] X. Zhang and Z. M. Özsoyoglu, "Some results on the containment and minimization of (in) equality queries," *Inf. Process. Lett.*, vol. 50, no. 5, pp. 259–267, 1994.
[22] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom, "Constraint checking with partial information," in *PODS*, 1994, pp. 45–55.
[23] P. Erdos and A. Rényi, "Asymmetric graphs," *Acta Mathematica Hungarica*, vol. 14, no. 3, pp. 295–315, 1963.