Distributed Algorithms for the Transitive Closure

Eric Gribkoff
University of California, Davis
eagribkoff@ucdavis.edu

ABSTRACT

Many database queries, such as reachability and regular path queries, can be reduced to finding the transitive closure of the underlying graph. For calculating the transitive closure of large graphs, a distributed computation framework is required to handle the large data volume (which can approach $O(|V|^2)$ space). Map Reduce was not originally designed for recursive computations, but recent work has sought to expand its capabilities in this regard. Distributed, recursive evaluation of the transitive closure faces two main challenges in the Map Reduce environment: a large number of rounds may be required to find all tuples in the transitive closure on large graphs, and the amount of duplicate tuples derived may incur a data volume cost far greater than the size of the transitive closure itself. Seminaive and smart are two algorithms for computing the transitive closure which make different choices in handling the tradeoffs between these two problems. Recent work suggests that smart may be superior to seminaive in the Map Reduce paradigm, based on an analysis of the algorithms but without an accompanying implementation. This view diverges from earlier work which found seminaive to be superior in the majority of cases, albeit in a non-distributed environment. This paper presents implementations of seminaive and smart transitive closure built upon the Hadoop framework for distributed computation and compares the performance of the algorithms across a broad class of data sets, including trees, acyclic, and cyclic graphs.

Categories and Subject Descriptors

H.4 [**Map Reduce**]: Hadoop; H.4 [**Transitive Closure**]: Distributed Evaluation

General Terms

Transitive Closure

1. INTRODUCTION

Computing the full transitive closure of a directed graph G involves finding all pairs of nodes (x, y) such that a directed path exists from x to y. For a graph with V vertices, the size of the transitive closure has an upper bound of $O(|V|^2)$. Although the transitive closure of typical graphs will not reach this upper bound, the transitive closure of graphs that exhibit a reasonable amount of connectivity among their nodes, like in social networks, becomes a challenge to process and store.

Seminaive and smart are two algorithms for computing the transitive closure of graphs. They have been extensively studied, and recent work has focused on optimizing seminaive in the Map Reduce environment [14]. However, previous comparisons of their relative performance are confined to the early 1990s, and primarily in a non-parallel environment [10].

While neither algorithm is optimal for all graphs, in [10], seminaive was found to outperform smart in the majority of cases. Despite these findings, [2] conjectures that smart may in fact be the superior algorithm in the cluster environment, as a consequence of different performance bottlenecks in the parallel environment.

This paper presents the results of parallel evaluation of seminaive and smart transitive closure on multiple data sets, including directed trees, directed acyclic graphs, and directed cyclic graphs. The performance results largely agree with the earlier findings in [10], suggesting that the distributed environment does not sufficiently alleviate the drawbacks inherent in the smart transitive closure algorithm.

The debate over the performance of seminaive versus smart cannot be settled conclusively, because there exist graphs where each algorithm's performance is optimal [2]. The results contained in this paper help demonstrate that seminaive outperforms smart, in many frequently arising cases. Further, the results are broken down by graph type and each experiment describes the parameters of the accompanying data set, allowing the results here to guide the choice of transitive closure algorithm for applications where the graph type is known beforehand.

As an aid to the further comparison of the performance of seminaive and smart, the implementations[1] return statistics on the number of derivations required during execution.

[1]Available from the author.

The number of derivations directly determines the overall data volume cost of the algorithms. Data volume cost, as a major barometer of performance, is discussed in [2]. The Map Reduce implementations of seminaive and smart rely on a common set of building blocks that provide join and set difference operations in the Hadoop framework. Thus, the performance difference between seminaive and smart can be directly attributed to the different amount of duplicated work each algorithm performs. As will be detailed in the experimental results, smart and seminaive vary quite markedly in the number of duplicate facts they derive during computation of the transitive closure.

The implementations here provide a direct and simple way to retrieve the total number of derivations required for each algorithm on a given data set. Therefore, aside from providing an early performance benchmark for future optimizations to improve upon, these implementations can also aid in exploring classes of graph data with the intent of discovering how seminaive and smart differ in their handling of the transitive closure.

1.1 Goals

I undertook this work to accomplish the following:

- Resolve the discrepancy between earlier research, which found that seminaive was generally superior to smart for evaluation of the transitive closure, and recent works, which propose that smart offers better performance than seminaive in the Map Reduce paradigm
- Build working Map Reduce implementations of the seminaive and smart algorithms, which allow comparison of runtimes and track relevant differences in the operations of the two algorithms (specifically, number of derivations and number of rounds)
- Benchmark these Map Reduce implementations of smart and seminaive on three types of data: trees, acyclic, and cyclic graphs
- Extend the theoretical discussion of the relative performance of smart and seminaive in the distributed setting to actual implementations
- Identify directions for future work on distributed algorithms for the transitive closure

1.2 Contributions

The relative performance of seminaive and smart transitive closure has been previously studied, but to my knowledge the only performance comparisons of their properties in a distributed computing environment are limited to the theoretical analysis on a fixed graph type presented in [2]. Most of the existing research into smart took place in or before the early 1990s, before the advent of Map Reduce frameworks like Hadoop.

[2] and [1] identify smart as a possibly valuable algorithm for general purpose computing that relies on the transitive closure in the Map Reduce framework, such as large-scale distributed Datalog evaluation. However, work specifically aimed at distributed Datalog evaluation focuses exclusively on seminaive evaluation [14].

In order to move forward with any computations requiring computation of the transitive closure, it is important to understand if smart does outperform seminaive, and if so, for what types of data smart exhibits superior performance.

To that end, this paper presents the performance results of these algorithms in the Map Reduce framework and compares these results to those of previous researchers working in the non-distributed setting.

Furthermore, this paper also presents a pedagogical aid for understanding the number of duplicate derivations derived by each algorithm for a given graph. The implementations accompanying this paper record the number of derivations at each step of the algorithms, and help empirical analysis of algorithms and graphs for future work.

2. BACKGROUND

2.1 Map Reduce

Map Reduce is a programming model and the associated architecture that supports implementations built on top of this model [11]. Fundamentally, Map Reduce divides problems into two phases: the Map phase and the Reduce phase. Mappers receive a list of (*key, value*) pairs, and for each pair, emit zero or more (*key', value'*) pairs. The output of the Map phase is sent to the Reducers, organized by the Map Reduce framework such that all pairs with the same key from the Map phase will be sent to a single Reducer. The Reducer then performs its processing, such as an aggregation, and outputs the final result(s).

Map Reduce, and the Hadoop framework used for this paper, does not directly support iterative algorithms or offer primitives for graph representations. This is regarded as a limitation of Hadoop and Map Reduce, but there is a great deal of research effort involved in optimizing Hadoop for large-scale iterative graph processing [14], [2], [3] as well as the building of new distributed systems designed for graph algorithms [12].

For more details, the reader is referred to [6], [11].

2.2 Transitive Closure

2.2.1 Linear

In general, the linear transitive closure can be expressed with the following Datalog code:

```
tc(x, y) :- p(x, y).
tc(x, z) :- tc(x, y), p(y, z).
```

The linear transitive closure builds the transitive closure by building the `tc()` relation, starting with the path facts from the base graph. The closure is iteratively expanded by following additional edges, one at a time. In order to compute the transitive closure using a linear algorithm, the number of iterations required is equal to one less than the graph's diameter, which is the length of the longest shortest path between any pair of nodes.

As an example illustrating the number of rounds required by the linear transitive closure computation, consider Figure 1. Node a is connected to node e by a path of length 4. At initialization, the initial base path is all that is known,

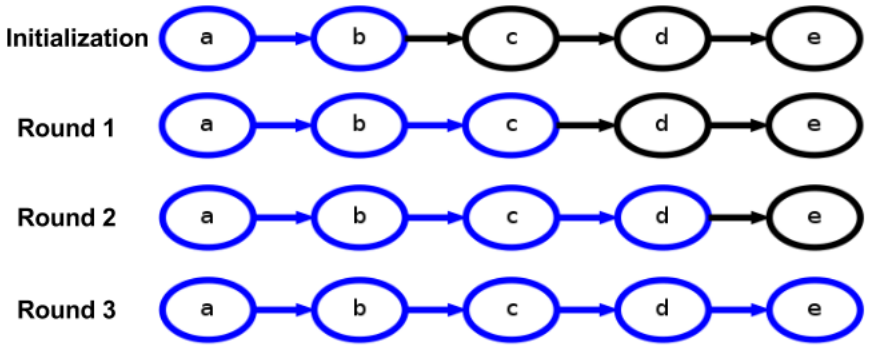

Figure 1: Linear Transitive Closure: Rounds Required to Derive (a, e)

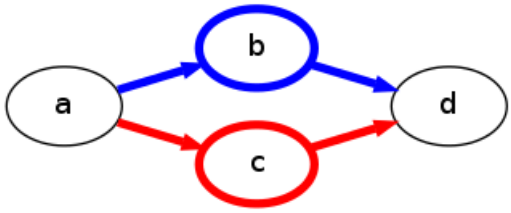

Figure 2: Two Derivations for (a, d)

(1) $T = R$
(2) $\Delta T = R$
(3) while $\Delta T \neq \emptyset$ do
(4) $\quad \Delta T = \Delta T \circ R - T$
(5) $\quad T = T \cup \Delta T$
(6) end

Figure 4: Seminaive TC Pseudocode

showing that a is directly connected to b. In Round 1, that fact is combined with the knowledge that b is connected to c and a path of length 2, $a \rightarrow c$, is derived. In Round n, paths of length n (derived in the previous round) are combined with paths of length 1 (edges from the original graph) to produce paths of length $n + 1$. Thus, in Round 3, the tuple (a, e) is produced, found by a path of length 4.

There are two important properties to note about the linear transition closure computation.

First, if a pair of nodes (x, y) is connected by multiple paths, the tuple (x, y) may be derived more than once over the course of computing the transitive closure. This is illustrated in Figure 2, where the transitive closure contains (a, d), which will be formed in Round 1 by combining (a, b) with (b, d), shown in blue, and in the same round by combining (a, c) with (c, d), shown in red.

Second, if there is a path of length n connecting (x, y), then (x, y) will be derived no later than round $n - 1$. This same tuple may be derived again in subsequent rounds, if there exist other paths of length greater than n between the two nodes. This is shown in Figure 3, which shows the derivation of (a, d) in Round 1.

2.2.1.1 *Seminaive.*

The evaluation of the linear transitive closure rule is accomplished using the seminaive algorithm. Pseudocode for the seminaive algorithm is presented in Figure 4.

R denotes the original graph. T is initialized to R, and will contain the transitive closure of R upon termination. ΔT contains the newly derived tuples from the previous round. At each iteration of the while loop, ΔT is updated by joining ΔT with the edges of the original graph, R.

On iteration n, ΔT contains all pairs of nodes such that a shortest path between them has length $n + 1$. These pairs could not have been discovered in a previous round (by virtue of the shortest path between them having length $n + 1$). If any pair produced by the join has already been found, then this implies there exists a shorter path between the two nodes which was found on a previous iteration. The set difference operation removes any such tuples. At the end of iteration n, every tuple in ΔT represents a newly discovered tuple in the transitive closure.

The reasoning behind the seminaive name is that it avoids the fully naive join, which would perform the join $T \circ R$ at each iteration. This is enabled by exploiting the knowledge that the only new tuples possible from a join of $T \circ R$ at iteration $n + 1$ come from elements of T that were newly derived at iteration n - these newly derived tuples are exactly the contents of ΔT in seminaive.

As seen in figure 2, it is possible that $\Delta T \circ R$ will contain duplicates. Without the set difference operation in line 4, $\Delta T \circ R$ may also contain tuples discovered via shorter paths in previous rounds. In the absence of cycles, the set difference operation of line 4 is not required for termination [10]. In the implementations discussed in this paper, the set difference operation (and its byproduct, duplicate elimination) is performed in the interest of speed. As will be seen, the strength of seminaive evaluation is that it avoids many, but not all, unnecessary derivations. Duplicate elimination is key to upholding that characteristic.

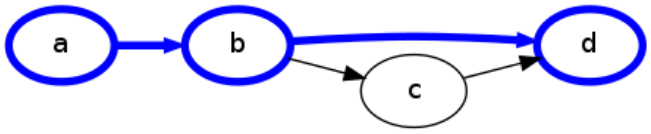

Figure 3: Derivation of (a, d)

2.2.2 Non-Linear

In contrast to the linear transitive closure, consider the standard non-linear transitive closure algorithm, expressed in the following Datalog:

```
tc(x, y) :- p(x, y).
tc(x, z) :- tc(x, y), tc(y, z).
```

By joining the `tc()` relation to itself, this algorithm is able to find all paths of up to length 2 on the first iteration, up to length 4 on the second iteration, up to length 8 on the third, up to length 16 on the fourth, and so on. This will compute the transitive closure of a graph in a number of rounds logarithmic in the graph's diameter.

In a distributed computation environment, and in particular using the Map Reduce framework, there is significant overhead and network traffic involved in initiating a single round of computation. This is especially true with graph computations that require passing the entire current state of the computation through the cluster on each iteration [12]. Finding the transitive closure in fewer rounds can help to avoid that fixed overhead cost as much as possible. Figure 5 provides an example of this, showing that the longest path length which can be found in 3 rounds of the nonlinear transitive closure computation is 8, compared to 4 in the same number of rounds of the linear transitive closure computation, as shown in figure 1.

In general, in n rounds, the non-linear transitive closure computation produces tuples containing all pairs of nodes connected by paths of up to length 2^n. Thus, in a logarithmic number of rounds, the entire transitive closure will be found.

However, there is a significant trade off involved in using the non-linear transitive closure algorithm instead of its linear counterpart. This trade-off takes the form of a dramatically larger amount of potentially duplicated work.

Each row of figure 6 represents a valid pair of two path facts which, using non-linear transitive closure, will be combined to derive the (a, i) path fact in the 9-node line graph's transitive closure. Whereas linear transitive closure would derive this fact exactly once (combining (a, h) with (h, i), as shown in figure 7), non-linear transitive closure derives the same fact in 8 different ways. While a line graph is perhaps the easiest means of displaying the additional work that can be done by the non-linear algorithm, it does not show the true magnitude of the number of duplicate derivations that may be produced. In general, the number of derivations of transitive closure facts in a graph with n nodes is given by the formula:

$$\Sigma_{i=1}^{n} \texttt{from}(i)\texttt{to}(i)$$

To(i) denotes the number of nodes which can reach node i, and from(i) denotes the number of nodes which node i can reach. This sum is never less than the number of derivations produced by a linear transitive closure algorithm, and is often much greater [1].

The experimental results presented in this paper give data on the number of derivations produced by transitive closure algorithms executed on the Gnutella 08 Peer-to-Peer network graph, which contains 20,777 edges in the base graph and 13,148,244 edges in the complete transitive closure. Non-linear transitive closure, executed on this graph, would produce approximately 27 billion derivations, nearly all of them redundant. This makes the direct implementation of non-linear transitive closure impractical on real data sets.

While the fewer number of rounds required by the non-linear transitive closure has benefits in a distributed setting, the huge number of duplicate path facts more than offsets those benefits. While the overhead of additional rounds is non-trivial, the overhead of processing billions of additional tuples incurs a far more significant performance overhead, as those facts must be shuffled and distributed across the network by the Map Reduce framework before they can be recognized as duplicates and discarded.

2.2.2.1 Smart.

The smart transitive closure algorithm was developed in an attempt to harness the benefits of the non-linear algorithm's logarithmic number of rounds, while at the same time avoiding a dramatic blow-up in the number of duplicate derivations.

Smart shares with the linear algorithm the desirable property of discovering each shortest path only once [2]. On each iteration, it joins together paths whose length is a power of 2 with paths of strictly lesser length. In this way, it is able to compute the transitive closure in a logarithmic number of rounds while deriving far less duplicate facts than the standard non-linear algorithm.

Smart iteratively builds two relations. At iteration i, Q holds all pairs of nodes between which the length of the shortest path is exactly equal to 2^i. P holds all pairs of nodes between which the shortest path has length less than 2^i.

The algorithm is presented in pseudocode in figure 8.

The relation Q is built similarly to the example derivations in figure 5. In the first round of the computation, it is simply

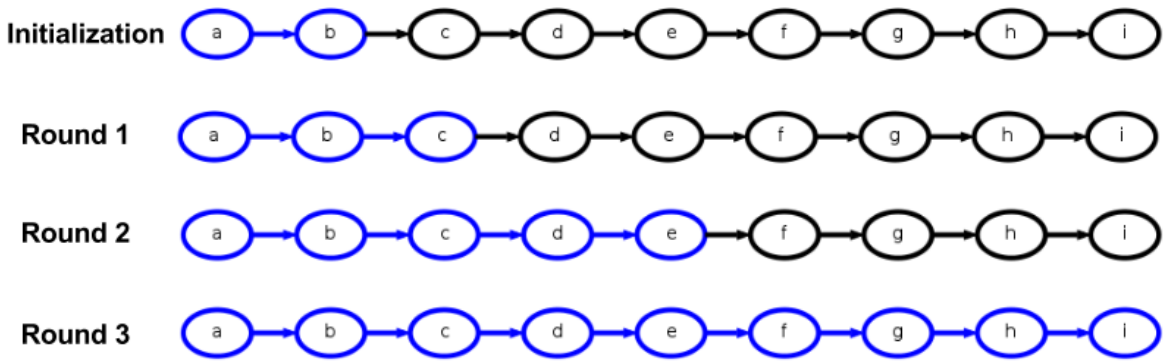

Figure 5: Non-linear Transitive Closure

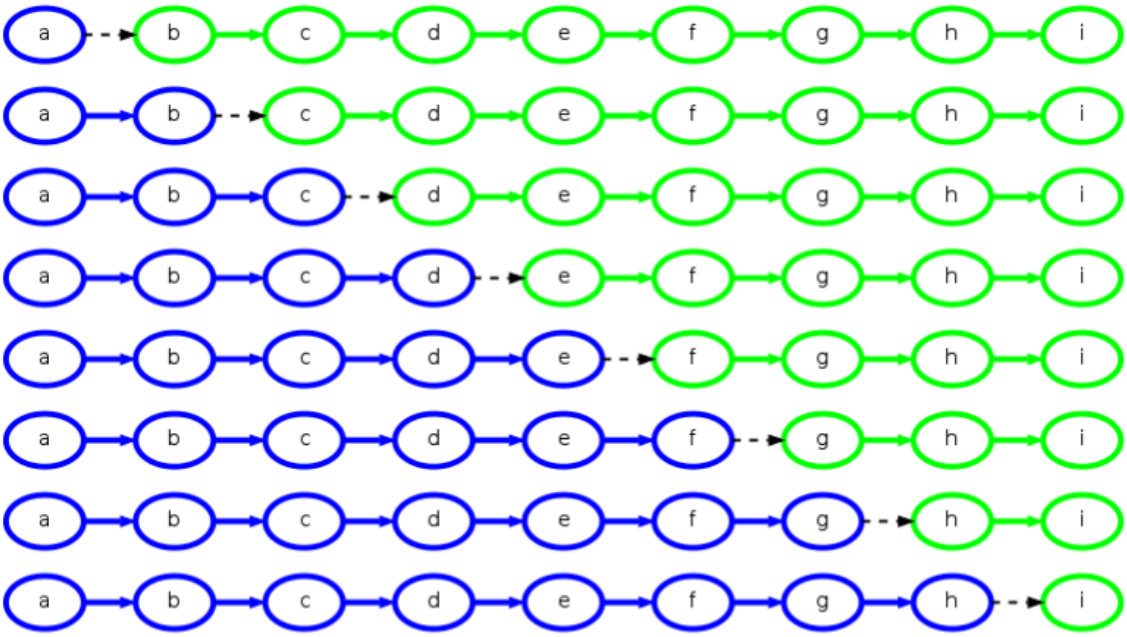

Figure 6: Non-Linear Derivations of (a, i)

Figure 7: Linear Derivation of (a, i)

(1) $Q = Edges$
(2) $P = \emptyset$
(3) while $Q \neq \emptyset$ do
(4) $\quad \Delta P = Q \circ P$
(5) $\quad P = Q \cup P \cup \Delta P$
(6) $\quad Q = Q \circ Q - P$
(7) end

Figure 8: Smart TC Pseudocode

a self-join of the original edges; in fact, all the transitive closure algorithms discussed here begin with this first step. The result of this join contains all pairs of nodes connected by a path of length 2, but there may be nodes in this result that are connected by a shorter path (a direct edge) so the set difference operation in line 6 is required to remove any pairs of nodes where there exists a path between them of length less than 2. In the next iteration, Q is again be joined with itself, producing (after the set difference) all pairs of nodes connected by a shortest path of exactly length 4.

In round i, P is formed from the join of Q and P. This combines paths from Q (of length exactly 2^{i-1}) with paths from P (of length strictly less than 2^{i-1}). This will find all shortest paths of any length between 2^{i-1} and $2^i - 1$. To also incorporate previously discovered paths of lesser length, line 5 takes the union of $Q \circ P$ with the tuples of Q and P. Therefore, at the end of round i, P contains all paths of length less than 2^i.

Not shown in the pseudocode is a duplicate elimination step after line 5, to remove redundant tuples produced by the join and union operations that form P. Duplicate elimination is required on the Q relation for termination detection in the presence of cycles, and is implicit in the set difference operation of line 6. However, duplicate elimination on P is not required to detect termination, but it is necessary for efficient performance on graphs where smart will produce duplicate derivations [10].

Smart discovers each shortest path only once and shares with the nonlinear algorithm the ability to compute the transitive closure in a logarithmic number of rounds. This combination allows smart to compute the transitive closure in fewer rounds than the linear algorithm while deriving fewer duplicate tuples than the nonlinear algorithm. However, the number of derivations of smart will typically be greater than the derivations required by seminaive. In some cases, these additional derivations outweigh the advantage smart gains from terminating in fewer rounds.

2.2.3 Other Approaches

Linear transitive closure algorithms have the beneficial property that they discover each shortest path only once [1]. This seems to be the best that can be practically done. The linear algorithms can still derive the same fact in different ways if multiple paths exist between two nodes. More complicated algorithms which produce even fewer derivations are unlikely to produce any practical benefits due to the larger number of set operations required to ensure this fewer number of derivations. For a discussion of this, see [4] and its discussion of the "Minimal" transitive closure algorithm. Such alternatives do not appear to be considered feasible for actual implementations, and are not discussed in [1], and likewise excluded from further consideration in this paper.

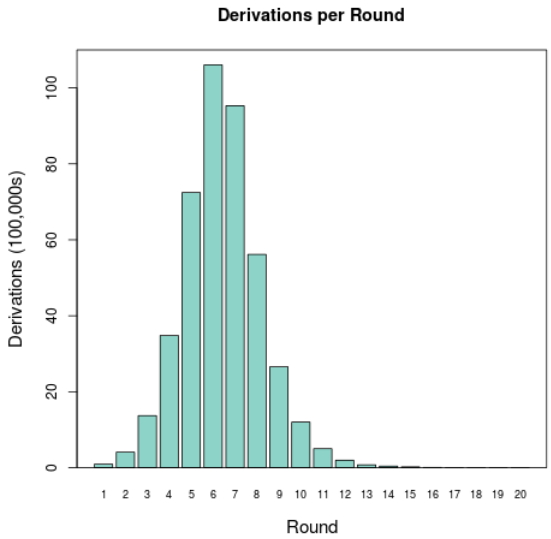

Figure 9: Derivations per Round of Seminaive on Gnutella 08

2.3 Challenges Faced in Distributed Transitive Closure

2.3.1 The Long Tail

Seminaive evaluation of the transitive closure provides a good example of a phenomenon known as the long tail, exhibited by a wide spectrum of recursive programs: as the program enters its final iterations, it begins to derive fewer and fewer facts. Eventually, the number of new facts equals 0 and the computation ends, but usually this is proceeded by many iterations where only a few new facts are derived. In the Map Reduce environment, these long tails have a fixed cost per round, while the cluster distributes the data and begins the computation for each round, with very little payoff when few facts are derived. This phenomenon is demonstrated in figure 9, displaying the number of new facts derived by each round of the seminaive transitive closure algorithm.

By limiting the number of rounds, algorithms like smart aim to avoid the problem of the long tail. However, in so doing, they encounter a potentially more serious problem: duplicate tuples.

2.3.2 Data Volume

Data volume, or the amount of data passed through the cluster during the entire computation, is proposed by [1] as an appropriate cost model for distributed Map Reduce computations. With any parallel computation that is not significantly CPU-bound, such as seminaive and smart transitive closure, the running times of individual mappers and reducers during a Map Reduce computation will be proportional to the size of their input. Therefore, measuring the total data size that must be processed by the system, summed

across all rounds of the computation, gives a method of assessing costs that correlates well with actual wall-clock time.

The smart algorithm works by combining the work of multiple rounds of seminaive into a single round of computation. This means that the capability of seminaive to proactively eliminate duplicate tuples is much greater than that of smart. If the nth round of smart derives all pairs of nodes whose shortest path between them is of length 2^{n-1} to 2^n, then by the time smart is able to perform duplicate elimination on these tuples, seminaive would have performed duplicate elimination 2^{n-1} times. As seen in the experiments in this paper, on some graphs this matters very little, but on other graphs, this delay in performing duplicate elimination leads to multiplicative effects in the number of derivations produced by smart within a single round [10].

2.3.2.1 Analysis of Seminaive and Smart on Ladder Graphs.

Following the definition and example of ladder graphs given in [2], consider a directed acyclic graph with 5 "levels". Every node in level 1 has a directed edge connecting it to every node in level 2, every node in level 2 has a directed edge connecting it to every node in level 3, and so on. Suppose that level 1 has m nodes, level 2 has 1 node, level 3 again has m nodes, level 4 has 1 node, and level 5 has m nodes.

The transitive closure of this ladder graph will contain $3m^2 + 6m + 1$ tuples. As presented in [2], seminaive and smart can be analyzed in terms of the length pairs which the algorithms will combine together to form paths. For this ladder graph, the valid length pairs for seminaive are $(1,1),(2,1)$, and $(3,1)$. For smart, the valid length pairs are $(1,1),(2,1)$, and $(2,2)$. The only difference in the two algorithms is how they derive paths of length 4.

In figure 10, which shows a ladder graph with $m = 3$, seminaive will combine the tuples $(a,h),(b,h)$, and (c,h) with $(h,i),(h,j)$, and (h,k) to derive all 9 tuples from level 1 to level 5. This requires $m^2 = 9$ derivations, the minimum possible, as this equals the number of pairs of nodes connected by a shortest path with length 4 in the transitive closure.

In contrast, smart will combine the 9 paths from level 1 to level 3 with the 9 paths from level 3 to level 5, requiring $m^3 = 27$ derivations.

On a ladder graph of this type, the total number of derivations produced by seminaive will be $2m(2m+1) = O(m^2)$. The number of derivations produced by smart will be $m(m+1)(m+2) = O(m^3)$. Thus the performance of smart can be made arbitrarily bad when compared to that of seminaive by increasing m.

This graph, and others like it, highlight the difficulty of comparing seminaive and smart over arbitrary graphs. It is possible to construct similar examples where seminaive does far less work than smart. However, as seen in the experiments conducted for this paper, there are also classes of graphs where smart's fewer rounds translates into far better performance than the seminaive algorithm.

3. IMPLEMENTATIONS

In this paper, I describe implementations and provide performance benchmarks for the following transitive closure algorithms:

- Seminaive TC
- Smart TC

According to previous research, the relative performance of these algorithms can be highly data dependent [2] [4]. However, for problems which require computation of the transitive closure, they must calculate it using some algorithm. As such, the intent of the presented research is to provide some guidelines for choosing the algorithm with the best chance of performing well over a broad variety of graph types.

3.1 Common Building Blocks

Each of the transitive closure algorithms uses a sequence of Map Reduce jobs. These Map Reduce jobs consist of a sequence of join and set difference tasks. The particular sequence of tasks is different for seminaive and smart, but the tasks themselves are implemented identically for both algorithms. Before discussing how the tasks are combined together to form each algorithm, the following pseudocode describes the join and set difference tasks used. The join and set difference algorithms are adapted from those presented in [1] for computation of the non-linear transitive closure.

3.1.1 Join

Figure 11 describes the operation of the mapper and reducer tasks for the join phase of the transitive closure algorithm. $h(x)$ is a hashing function, where the number of hash buckets is equal to the number of reducers. In the implementations described in this paper, $h(x) = x \,\%\, r$, where r is the number of reducers.

Given the join $R \circ S$, which joins tuples (x,y) in R to tuples (y,z) in S, the mapper sends all tuples (x,y) from R to the reducer corresponding to $h(y)$, while all tuples (p,q) from S are sent to the reducer corresponding to $h(p)$. If $y = p$, the tuples will be sent to the same reducer.

Because multiple nodes in the original relations will hash to the same key, the join reducer keeps a store of all nodes x from tuples (x,y) in R which can reach y, and a similar store for all nodes z from tuples (y,z) which can be reached from y in S. Then, when a new tuple (x,y) arrives from R, the reducer emits all pairs of nodes (x,v) where v is a node reachable from y in S. Likewise, if a tuple (y,z) arrives from S, the reducer emits all pairs of nodes (u,z) where u is a node which can reach y in R.

3.1.2 Set Difference

As presented in figure 12, the set difference map reduce task is simpler than the join. The map is the identity: it receives and outputs a tuple (x,y) along with its source tag, which indicates whether the tuple came from the left or right side of the set difference operation. In the reducer, if the right relation is not present in the list of sources, the tuple (x,y) is emitted. Otherwise, the reducer outputs nothing, as the

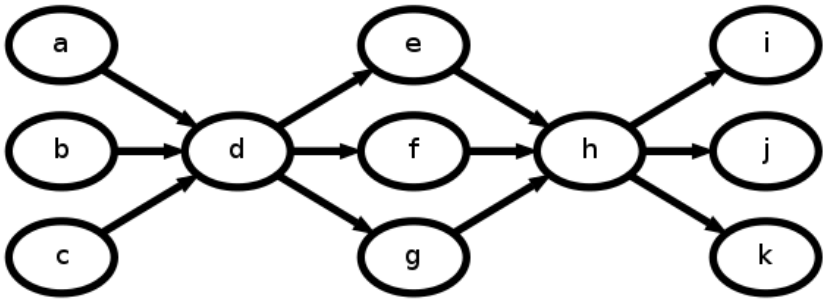

Figure 10: Ladder Graph with $m = 3$

```
function JOIN_MAP(<(x, y), source>)
    if source = left_relation then
        emit <h(y), (x, y, source)>
    else
        emit <h(x), (x, y, source)>
    end if
end function

function JOIN_REDUCE(<key, [(x_1, y_1, source_1), ..., (x_n, y_n, source_n)]>)
    if source = left_relation then
        add x to canReach(y)
        for all v in reachableFrom(y) do
            emit <x, v>
        end for
    else
        add y to reachableFrom(x)
        for all u in canReach(x) do
            emit <u, y>
        end for
    end if
end function
```
Figure 11: Join Map and Reduce

```
function SETDIFF_MAP(<(x, y), source>)
    emit <(x, y), source>
end function

function SETDIFF_REDUCE(<(x, y), [source_1, ..., source_n]>)
    if right_relation not in [source_1, ..., source_n] then
        emit <x, y>
    end if
end function
```
Figure 12: Set Difference Map and Reduce

tuple was present in the relation on the right-hand side of the set difference operation.

As specified here, it should be noted the set difference job also performs duplicate elimination.

3.2 Seminaive

The Hadoop implementation of seminaive requires one join task and one set difference task per iteration.

The union, specified in the pseudocode in figure 4, is a non-operation in the Hadoop framework. Instead of writing code to perform the union, the relations in the union are simply fed as separate inputs to the next step of the algorithm. Hadoop is optimized to work on a small number of large files instead of many small files. The Hadoop API allows specifying a CombineFileInputFormat() to aggregate multiple input files into a single file before processing, and this allows the union step to be left out of the seminaive implementation without incurring the performance overhead of reading from numerous small files.

The split size used to specify the amount of input sent to each mapper is set at 5MB for the experiments below, which was found to be the best performing split size for seminaive on the Gnutella 08 peer-to-peer network graph.

3.3 Smart

Each iteration of the Hadoop implementation of smart requires two join tasks, one to form P and another to form Q. The set difference operation in line 6 of figure 8 is expressed using a single set difference task. Additionally, a duplicate elimination operation is performed after the join to produce P. The duplicate elimination Map Reduce task is essentially the same as the procedure presented in figure 12, with the source parameter omitted. As reported by [10], and confirmed by initial tests of the algorithms, duplicate elimination is essential for efficient execution.

The split size used to specify the amount of input sent to each mapper is set at 128MB for the experiments below, which was found to be the best performing split size for smart on the Gnutella 08 graph.

4. EXPERIMENTS

4.1 Graph Parameters

The runtime of transitive closure algorithms depends heavily on the structure of the input graph. The algorithms can ex-

hibit wildly varying runtimes for graphs with approximately the same number of nodes and edges.

For example, the Gnutella 08 peer-to-peer network from the Stanford Network Analysis Project [13] contains 6,301 nodes and 2,077 edges. The edge list text file is only 211KB. However, the transitive closure of this graph contains 13,148,244 tuples and occupies 121MB of space. In comparison, a binary tree with a depth of 20 has 2,097,150 edges and its transitive closure contains 39,845,890 tuples. The transitive closure is only 3 times larger than that of Gnutella 08, even though the original graph containss approximately 1000 times as many tuples. Furthermore, smart far outperforms seminaive on the binary tree data, while the reverse is true for the Gnutella 08 graph.

Drawing on the definitions of [10], this paper uses the following graph properties to parametrize the experimental evaluation of transitive closure algorithms:

- **R**, the size of the original graph in tuples (number of edges)
- **T**, the size of the transitive closure in tuples
- **d**, the diameter of the graph (the length of the longest shortest directed path).

4.2 Trees

[9] introduced the smart algorithm for computing the transitive closure and presented experimental results comparing the performance of smart and seminaive on lists and tree data. Smart was found to be superior. In subsequent work, [10] performed comprehensive benchmarking of the algorithms in a non-parallel environment. They found that smart outperformed seminaive on trees and graphs that were very "tree-like", with a large depth and a very limited number of redundant paths.

4.2.1 Binary Trees

Binary trees are easy to generate, and their regular growth makes a suitable framework for measuring algorithm scalability as the input size grows [12]. Adding nodes to graphs with a less regular structure can have unforeseen impacts on the computation of the full transitive closure. Adding an additional level of depth to a binary tree simply increases, in a predictable fashion, the parameters, **R**, **T** and **d**.

Runtimes for binary trees with depths from 5 to 20 levels are recorded in table 1. These tests were performed on a 4-node cluster on Amazon's EC2 service, using Cluster Compute Quadruple Extra Large instances with 7GB of memory and 8 virtual CPUs each. EC2's cluster compute instances are optimized for high-speed data transfer between instances. Smart's execution on a binary tree of depth 20 requires just under 12 minutes on 4 cluster compute nodes.

This is significantly faster than initial tests on EC2 M1 Large instances, with 7GB of memory and 2 virtual CPUs per machine but without the high-speed network capabilities. A binary tree of depth 20 took 21 minutes on a 12-node M1 Large cluster, and nearly 15 minutes even on a 48-node M1 Large cluster.

For trees, smart outperforms seminaive for all but very small datasets. The difference becomes larger as the data size increases. With a binary tree of depth 20, the computation of its 37,748,740 tuple transitive closure takes the seminaive algorithm 23 minutes, compared to under 12 minutes for smart.

The reasons for smart's superior performance on these data sets is shown in table 1. The number of derivations required by each algorithm is identical. In fact, neither algorithm derives any duplicate, or redundant, tuples. This is a general property of seminaive and smart transitive closure algorithms when executed on trees. In both algorithms, each path is discovered exactly once [1]. As each pair of connected nodes in a tree is connected by exactly one path, each element of the transitive closure is computed once, and neither seminaive nor smart does any duplicate work.

The total amount of work done by each algorithm is the same. This implies that smart does not suffer from its one potential drawback in relation to seminaive, which is the production of more unnecessary tuples. Instead, the only difference in their performance is smart's logarithmic number of rounds, enabling it to terminate the computation sooner.

When smart was first introduced in [9], the algorithm's performance was analyzed solely on list and tree data. With these inputs smart outperforms seminaive by a significant margin, as verified by the experiments performed here. This result does not necessarily generalize to non-tree data sets, as was noted in [4] and [10], and as confirmed by the following experiments.

As a performance-related note the runtime of each algorithm can be cut roughly in half by removing the duplicate elimination and set difference steps, since neither algorithm produces duplicate tuples on tree inputs. The duplicate elimination and set difference tasks consume approximately half of the execution time of each algorithm. Because trees do not contain duplicate paths, this effort is unnecessary. However, for testing purposes, duplicate elimination was still performed. This does not impact the relative performance of the algorithms and, in many general-purpose use cases, the input may not be known beforehand to form a tree.

4.3 Acyclic Graphs

Any cyclic graph can be condensed into an acyclic graph in linear time [15], by condensing all strongly connected components into single nodes. As such, [1] suggests studying seminaive and smart on acyclic graphs only. However, [10] reported no significant performance difference between cyclic and acyclic graphs. Furthermore, there is a time cost involved in condensing cyclic graphs into acyclic graphs. To determine whether or not the presence of cycles has an effect beyond that of the shrinking of the graph inherent in the condensation process, the implementations are tested on acyclic and cyclic graphs.

4.3.1 Random DAGs

Random directed acyclic graphs were generated for testing the algorithms. The procedure for generating the DAGs requires specifying the number of levels and the number of nodes per level, along with the probability of an edge exist-

Algorithm	d (Tree Depth)	R	T	Rounds	Derivations	Runtime (sec)
Seminaive	5	62	258	5	196	157
Smart	5	62	258	3	196	206
Seminaive	10	2046	18434	10	16388	325
Smart	10	2045	18434	4	16388	275
Seminaive	15	65534	917506	15	851972	561
Smart	15	65534	917506	4	851972	294
Seminaive	20	2097150	39845890	20	37748740	1403
Smart	20	2097150	39845890	5	37748740	718

Table 1: Binary Tree Results

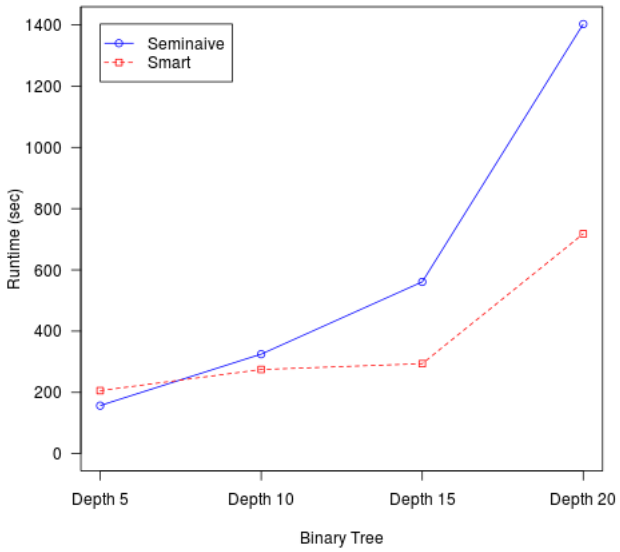

Figure 13: Seminaive and Smart as **d** increases

ing from any node in level i to any node in level j, $i > j$. Therefore, even though the DAGs contain "rungs" as in a ladder graph, there may be an edge from a node in one rung directly to any node in any lower rung. As such, duplicate paths can and do exist in the generated DAGs, which provides a useful contrast to the experiments on trees in the previous section, wherein both smart and seminaive were guaranteed to perform the same amount of total work.

The properties of the five randomly generated DAGs used in the testing are displayed in table 2. Performance results for seminaive and smart are presented in table 3. As with the binary tree experiments, these tests were performed on a 4-node cluster on Amazon's EC2 service, using Cluster Compute Quadruple Extra Large instances with 7GB of memory and 8 virtual CPUs each.

For graph A, the performance of seminaive and smart is virtually identical. With a graph of this small size (50 levels, each with 50 nodes, but only 3009 total edges), the produced DAG lacks many redundant paths. Seminaive and smart both produce only 21 duplicate derivations. While seminaive requires more rounds, the diameter of the graph (only 7) is not enough to substantially differentiate between the performance of seminaive and smart.

Graph D, with the same 0.001 probability of an edge existing as in graph A but with 100 nodes in each of 100 levels, has a diameter of 18. Here, seminaive produces about 3 million less duplicate tuples than smart, but that advantage is offset by requiring 13 more rounds than the smart implementation. As a result, graph D sees much better performance for the smart algorithm.

Likewise, in graph B, smart produces approximately 10 times as many derivations as seminaive, but still performs better, although the difference is less pronounced.

Graphs C and E possess a very high number of edges and hence more redundant paths. As a result, seminaive performs significantly better than smart. When the number of duplicate derivations is large for seminaive, it tends to be very large for smart due to the multiplicative effects of smart's less-frequent duplicate elimination. This phenomenon is noted in [10]. For a fixed-size cluster, the amount of data that can be quickly processed has an upper limit. The absolute number of tuples generated by smart explains its poorer performance on these two graphs.

4.4 Cyclic Graphs

4.4.1 Gnutella Peer-to-Peer Network Traffic

The Gnutella 08 network graph contains a snapshot of the Gnutella peer-to-peer file sharing network from August 2002 [13]. It represents a small example of a directed social network graph, and is well-suited as a test case for computing the transitive closure. The base graph has only 6,301 nodes and 20,777 edges, but the full transitive closure contains 13,148,244 tuples.

The runtimes of seminaive and smart on the Gnutella 08 graph are presented in figure 14.

The number of derivations produced by seminaive at each round were presented in figure 9. All together, seminaive produces 43,062,382 derivations. In contrast, smart produces 3,351,668,837. This increase in the number of derivations by a factor of approximately 77 more than offsets the difference in the number of rounds required: 20 for seminaive and 5 for smart.

For seminaive on the Gnutella 08 graph, performance peaks with a 6-node cluster and actually decreases slightly when moved to an 8-node cluster. Smart, on the other hand, continues to improve as the cluster size grows from 4 to 6 to 8 nodes. This may indicate that, with a certain amount of resources that would represent a vast surplus to the seminaive algorithm, smart may be able to process its larger data

Graph Designation	Levels	Nodes per Level	Probability of Edge	R	T
A	50	50	0.001	3009	7590
B	50	50	0.01	30524	1751972
C	50	50	0.1	306544	2877593
D	100	100	0.001	49381	4624135
E	100	100	0.01	494818	42713082

Table 2: Random DAG Properties

Algorithm	Graph	Rounds	Derivations	Runtime (sec)
Seminaive	A	7	4,602	226
Smart	A	3	4,602	204
Seminaive	B	11	10,795,412	445
Smart	B	4	101,241,793	398
Seminaive	C	4	223,828,338	436
Smart	C	3	1,661,412,216	1728
Seminaive	D	18	7,033,301	808
Smart	D	5	10,343,476	391
Seminaive	E	11	1,294,864,315	2462
Smart	E	Did not finish	>17,000,000,000	>6000

Table 3: Random DAG Results

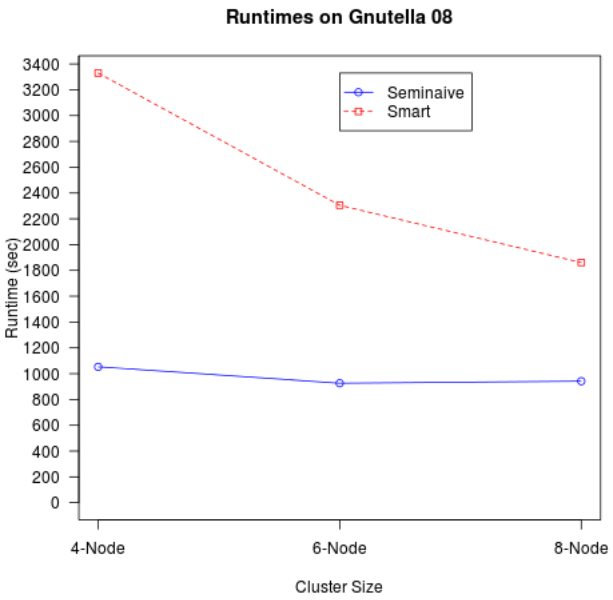

Figure 14: Performance of seminaive and smart as cluster size increases

volume quickly enough to outperform the seminaive implementation. The finite amount of available funding for EC2 resources precluded exploring this possibility.

4.4.1.1 Duplicate Elimination Costs.

For transitive closure computations on general graphs, including directed acyclic and cyclic graphs, the dominant factor in determining runtime is how quickly duplicate tuples can be removed [10]. Figure 15 shows the runtime for each round of seminaive transitive closure on the Gnutella 08 graph, broken down by time spent on the join and set difference operations of each round.

A similar presentation, albeit on a log scale, is provided in figure 16 for the execution of smart on the Gnutella 08 data. As in the seminaive algorithm, the runtime of smart is dominated by the cost of the duplicate elimination and set difference operations.

5. RELATED AND FUTURE WORK

5.1 Regular Path Queries

The graphs used in these experiments are small enough to fit in the memory of a single computer. With data of this size, the Hadoop Map Reduce framework cannot compete with the performance of libraries like NetworkX [8], a highly optimized non-distributed graph library. However, the Map Reduce algorithms scale to larger data sets in a manner difficult for single-computer implementations.

Map Reduce's ability to scale to vastly larger data sets provided the impetus for this work, which arose from difficulties encountered in evaluating Regular Path Queries, or RPQs [7], on large relations. RPQ/4 results can be viewed as providing complete, piecewise provenance for each tuple in the transitive closure: if (x, y) is in the transitive closure, the RPQ/4 result will include $\{(x, y, x, z_1), (x, y, z_1, z_2), ..., (x, y, z_n, y)\}$, describing all of the tuples along *any* path between x and y. While the current Map Reduce implemen-

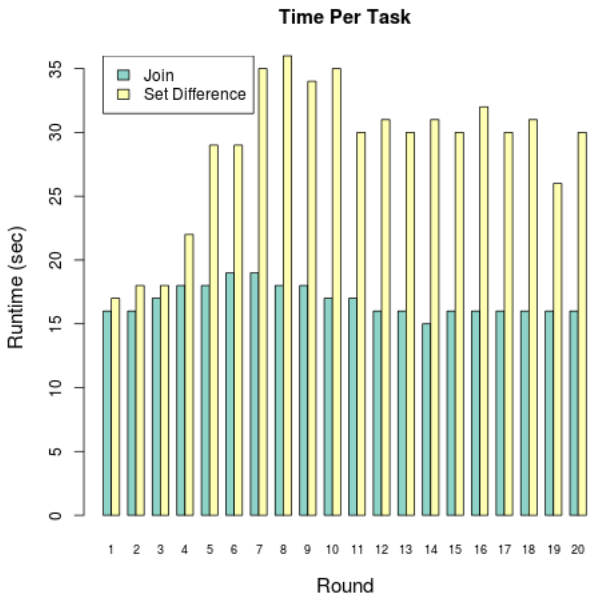

Figure 15: Seminaive task runtimes: Gnutella 08, 6-node cluster

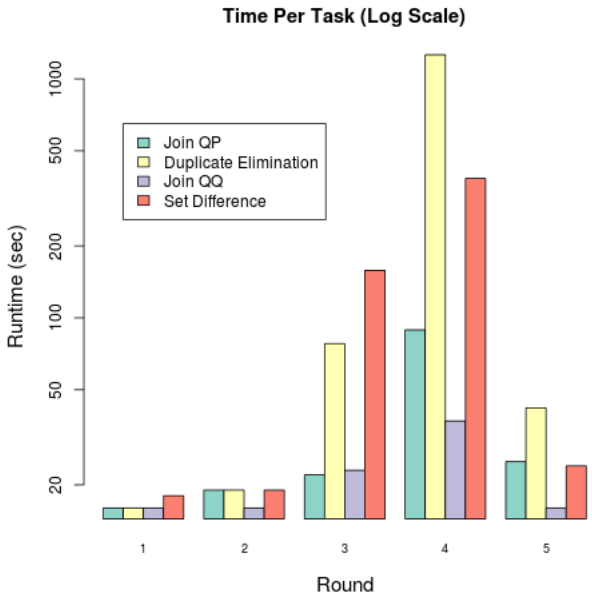

Figure 16: Smart task runtimes: Gnutella 08, 6-node cluster

tations described in this paper do not fully record all of this data (primarily because of the even worse space complexity required by RPQ/4), this provenance data is relatively easy to capture from the operations of the given implementations and remains an area of future work.

5.2 Within the Map Reduce Paradigm

The seminaive implementation presented in this paper could see a large-performance speed up from the Map Reduce optimizations presented in [3] and [14]. HaLoop, which is a fork of Apache Hadoop 0.20, automatically incorporates performance enhancements into the Hadoop environment, including quicker fixpoint detection and caching of loop-invariant data. The creators of HaLoop reported significant speedup for iterative jobs when compared to the original Hadoop framework.

The work by [14] on optimizing seminaive Datalog evaluation in Hadoop reports that, among other optimizations, they were able to use Hadoop's caching API to reduce the amount of data transmitted throughout the entire cluster at every iteration. This led to a reduction in running time from over 50 hours to 5 hours, for a single-source reachability query on a 2 billion node graph. The techniques they describe could be applied to the seminaive implementation presented in this paper.

5.3 Single Source Transitive Closure

Single source transitive closure is the problem of finding all nodes in a graph reachable from a single, designated source node. Rather than compute the full transitive closure, an algorithm can compute the subset of the transitive closure such that the designated node is the first element of each tuple in the returned set. This was studied in [10], which noted that seminaive can be modified in the following fashion to only compute the portion of the transitive closure required for answering the single-source query:

$T = \sigma_{x=source}(R)$
$\Delta T = \sigma_{x=source}(R)$
while $\Delta T \neq \emptyset$ do
    $\Delta T = \Delta T \circ R - T$
    $T = T \cup \Delta T$
end

Figure 17: Seminaive Single-Source TC Pseudocode

In other words, all that is required is to initially populate T and ΔT with tuples from the original relation R that match the selection $x = source$, for a designated *source* node.

[10] notes that smart cannot as easily compute solely the subset of the full transitive closure necessary for answering the single source query. Instead, smart requires computing all paths of length 2^i, then applying the same selection as in the single-source seminaive psuedocode. While [10] includes runtimes for experiments with single-source seminaive, they excluded smart from the single-source tests, regarding it as unlikely to compete with seminaive.

Beyond the Map Reduce paradigm, there are Pregel implementations capable of answering the single-source transitive

closure query on large graphs, as reported in [12]. An interesting avenue of future work would be to explore possible implementation or adaptation of the single-source seminaive algorithm to a Pregel-like graph computing environment.

5.4 Characterization of DAGs

[5] proposes a set of theorems that characterize the "height" and "width" of a DAG, where the calculations of these parameters are designed to capture information about the diameter of the DAG and number of redundant paths it contains. The calculation of height and width, according to their definitions, can be completed in a single depth-first or breadth-first traversal of the graph. They hoped that their proposed metrics would provide a strong indicator of expected performance for a range of transitive closure algorithms.

However, their work neglects to study trees, and their explanation of width corresponding intuitively to the number of duplicate derivations falls short when faced with binary trees. They define the height of a DAG as equal to the average height of all the nodes, where the height of a node equals the length of the longest path leaving that node. According to their definition, binary trees of depths from 5 to 20 have a height of less than one because the majority of the nodes are leaves. They define the width of a DAG as the number of edges divided by the height. Thus, the calculation of width becomes approximately the number of edges divided by 1. The width of a binary tree graph is approximately the same as the number of edges it contains. These width values are much higher than for the randomly generated DAGs, yet the binary trees in fact have no duplicate derivations. This suggests a shortcoming in [5]'s analysis of the intuitive meaning of their defined DAG height and width properties.

[10] makes the observation, repeated in the results of this paper, that smart usually only outperforms seminaive on input graphs that are very "tree-like". When the input graph is a tree, it is easy to see exactly how "tree-like" it is. When the input graph is a DAG with some number of duplicate paths, it is not so obvious how to determine the graph's "tree-ness".

In [2], the authors provide an analysis of a structured DAG formed by fusing together a binary tree with an inverted binary tree. The result is not a tree, but arguably, very "tree-like", in [10]'s terminology. Indeed, the analysis presented in [2] shows that the number of extra derivations required by smart over seminaive will never be very large, making smart and its logarithmic number of rounds the best choice for graphs of this form. Yet, knowledge of performance on this one class of graph does not generalize to other DAGs or imply a method of deriving expected performance on a broad class of graphs.

Future work in this space could examine random DAGs with a large number of varying properties, such as average outdegree and overall connectivity, to search for any correlation of these properties and the runtime of seminaive and smart transitive closure.

6. CONCLUSION

[2] describes two main costs faced in distributed computation of the transitive closure. The first cost is data volume: how much data is shuffled through the system throughout the entire computation. The second cost is the number of rounds required and the long tail phenomenon exhibited by recursive computations. As shown previously, the new facts derived in the later rounds of a transitive closure computation may be quite few.

In an early comparison of seminaive and smart, [10] found that seminaive was the best choice for most graph types. However, smart requires fewer rounds to compute the transitive closure. This, combined with the high cost per round in the Map Reduce environment, led [2] to propose that smart's implementation of non-linear transitive closure may be cost-effective to alleviate the long tail problem.

Despite [2]'s analysis, the experimental results presented in this paper demonstrate that the data volume cost tends to dominate the cost of additional rounds for non-tree graphs. While this reinforces [2]'s proposal of data volume as a model of cost for Map Reduce environments, it points to a strong limitation in the use of the smart algorithm for computing the transitive closure.

The experimental results suggest that seminaive is the best choice for calculating the transitive closure of large graphs in the Map Reduce paradigm. For all tested non-tree graphs, the performance of seminaive either far exceeds or is at least comparable to that of smart.

7. REFERENCES

[1] F. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. Ullman. Map-reduce extensions and recursive queries. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 1–8. ACM, 2011.

[2] F. N. Afrati and J. D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 132–143. ACM, 2012.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The haloop approach to large-scale iterative data analysis. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(2):169–190, 2012.

[4] F. Cacace, S. Ceri, and M. A. Houtsma. An overview of parallel strategies for transitive closure on algebraic machines. In *Parallel Database Systems*, pages 44–62. Springer, 1991.

[5] S. Dar and R. Ramakrishnan. A performance study of transitive closure algorithms. In *ACM SIGMOD Record*, volume 23, pages 454–465. ACM, 1994.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] S. Dey, V. Cuevas-Vicenttín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher. On implementing provenance-aware regular path queries with relational query engines. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 214–223. ACM, 2013.

[8] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), 2008.

[9] Y. E. Ioannidis. *On the computation of the transitive closure of relational operators*. Electronics Research Laboratory, College of Engineering, University of California, 1986.

[10] R. Kabler, Y. E. Ioannidis, and M. J. Carey. Performance evaluation of algorithms for transitive closure. *Information Systems*, 17(5):415–441, 1992.

[11] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[13] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *arXiv preprint cs/0209028*, 2002.

[14] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing large-scale semi-naive datalog evaluation in hadoop. In *Datalog in Academia and Industry*, pages 165–176. Springer, 2012.

[15] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 230–242. ACM, 1990.