Transitive Closure and Recursive Datalog Implemented on Clusters*

Foto N. Afrati
National Technical University of Athens,Greece
afrati@softlab.ntua.gr

Jeffrey D. Ullman
Stanford University, USA
ullman@infolab.stanford.edu

ABSTRACT

Implementing recursive algorithms on computing clusters presents a number of new challenges. In particular, we consider the *endgame* problem: later rounds of a recursion often transfer only small amounts of data, causing high overhead for interprocessor communication. One way to deal with the endgame problem is to use an algorithm that reduces the number of rounds of the recursion. Especially, in an application like transitive closure ("TC") there are several recursive-doubling algorithms that use a logarithmic, rather than linear, number of rounds. Unfortunately, recursive-doubling algorithms can deduce many more facts than the linear TC algorithms, which could negate the cost savings from the elimination of the overhead due to the proliferation of small files. We are thus led to consider TC algorithms that, like the linear algorithms, have the *unique decomposition* property that assures paths are discovered only once. We find that many such algorithms exist, and we show that they are incomparable, in that any of them could prove best on some data — even lower in cost than the linear algorithms in some cases. The recursive-doubling approach to TC extends to other recursions as well. However, it is not acceptable to reduce the number of rounds at the expense of a major increase in the number of facts that are deduced. In this paper, we prove it is possible to implement any Datalog program of right-linear chain rules with a logarithmic number of rounds and no order-of-magnitude increase in the number of facts deduced. On the other hand, there are linear recursions for which the two goals of reducing the number of rounds and maintaining the total number of deduced facts cannot be met simultaneously. We show that the reachability problem cannot be solved in logarithmic rounds without using a binary predicate, thus squaring the number of potential facts to be deduced. We also show that the same-generation recursion cannot be solved in logarithmic rounds without using a predicate of arity three.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*distributed databases, parallel databases, query processing*

General Terms

Theory, Algorithms, Performance

Keywords

Map-reduce, transitive closure, Datalog, recursion, polynomial fringe property

1. RECURSION ON CLUSTERS

Computing clusters have been the focus of a large body of research recently. A familiar tool for certain jobs in such a computing environment is map-reduce [17] and its open source equivalent Hadoop [7] built on top of a *distributed file system (DFS)* [19]. A synopsis of the technology behind map-reduce, including implementation of the core relational-algebra operations, such as join, in map-reduce can be found in [27]. Extensions of map-reduce called *workflow systems* allow arbitrary acyclic (nonrecursive) connections of functions and the parallel tasks that implement those functions. Workflow systems include Dryad [22] and its extension DryadLINQ [34] from Microsoft, Clustera [18] from the University of Wisconsin, Hyracks [9] from the U. C. Irvine, Boom [6] from the U.C. Berkeley, and Nephele/PACT [8] from T. U. Berlin.

A number of large-scale computations of interest today are essentially recursive. Examples include the computation of PageRank and a number of problems on social-network graphs, such as certain community-finding algorithms (e.g., [20]), and connectivity studies on the Web (e.g., [11]). A few systems have been developed recently that handle recursion. Haloop [12] essentially implements recursion as a sequence of iterations of Hadoop jobs with special attention paid to run each task in iteration i at a compute node where it can find its needed output from round $i-1$. A similar idea is used in Twister [1]. Pregel [26] implements real recursion using a message-passing model of computation.

The computation model for cluster computing presents several challenges that have heretofore not been studied or were examined only peripherally. In [3] a number of key points and positions were argued:

*This research has been partially co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS. Strengthening Interdisciplinary and Inter-institutional Research.

EDBT 2012, March 26–30, 2012, Berlin, Germany.

- Compute-nodes in a cluster process tasks in parallel, and data needs to be transferred among nodes. *Data-volume cost*, the sum over all tasks of the size of their input, is the right measure of algorithm efficiency for many applications, including those that use relational-algebra operations.
- When large numbers of compute-nodes and tasks are involved in a recursion such as transitive closure ("TC"), later rounds tend to discover few new facts and therefore send *many small files* among compute nodes.
- Because the overhead of file transmission in a computing cluster is high, it is desirable that a recursion finish in as *few rounds* as possible.
- For TC, the nonlinear, or *recursive-doubling* approach, where each round doubles the length of the paths that have been discovered, has advantages because it computes the TC of an n-node graph in $O(\log n)$ rounds rather than the $O(n)$ rounds taken by linear TC algorithms. Unfortunately, nonlinear TC algorithms can have a much higher data-volume cost than do the linear algorithms.

A technique called *Smart* TC was discovered many years ago to reduce the cost of nonlinear TC. We shall discuss Smart TC and related algorithms in Section 2, where we examine the relative efficiency, in terms of data-volume cost, for a number of TC algorithms.

There is another issue with which we must grapple when we examine the implementation of recursions on relations. If we rewrite the recursion to allow recursive doubling and thereby reduce the number of rounds, we may increase the volume of data beyond a realistic limit due to increase of the arity of the recursive predicates, which, in turn results in computing more facts than necessary. In the past, optimization techniques, like magic sets [28] where introduced to avoid computing non-relevant facts. Limitations of such techniques were also noticed [2,4,16]. In Section 4, we show limitations of such arity-reducing techniques not previously seen. In particular we show that there is a tradeoff between the arity of recursive predicates and the number of rounds. We show it by using pumping techniques developed in [4]. An example follows which will serve to introduce this problem as well as the Datalog notation we use.

1.1 The Arity of Recursive Predicates

Consider the *reachability problem*, which is defined by the Datalog program below, using the notation of [28].

```
reach(X) :- source(X)
reach(X) :- reach(Y) & arc(Y,X)
```

In the above rules, *source* and *arc* are EDB relations (extensional database relations, stored in the database), while *reach* is an IDB relation (intensional database relation, computed by recursive application of the rules). We assume *source* contains one source node, and tuple (a, b) is in the relation *arc* if and only if there is an arc in the graph from a to b.

Suppose that the database describes the link structure of a significant portion of the Web, say a billion nodes. If we implement the program above in the obvious way, the number of rounds needed by the recursion is the distance of the node in the graph that is furthest from the source node. In the Web, most nodes are fairly close to one another, but all it takes is one long path to force there to be a very large number of rounds. For example, most people writing a manual with 50 chapters will create a directory page that links to all 50 chapters. But somebody will decide to link Chapter i only from Chapter $i-1$, thereby creating a very long path. In fact, the Google WebIQ project has explored reachablity in the Web from many different source sets and finds that, while almost everything reachable is reached within 15 hops, some searches must use "hundreds of rounds" before terminating.[1]

We might decide to reduce the number of rounds, and thereby reduce the overhead of moving many small files, by using nonlinear TC. The Datalog program might look like:

```
path(X,Y) :- arc(X,Y)
path(X,Y) :- path(X,Z) & path(Z,Y)
reach(X)  :- source(Y) & path(Y,X)
```

The first two rules compute the nonlinear TC for the entire graph, and the third rule uses the path information to discover those nodes reachable from the source node.

However, such a program could never be implemented on a graph of a billion nodes, because there would be 10^{18} possible path facts.[2] The reason for the difference is that while the original reachability program has only IDB predicates (*reach*) of arity 1, the revised program above has an IDB predicate (*path*) of arity 2. If the *domain size* (number of values an argument of a predicate can take) is m, then the number of potential facts that the program can derive is m raised to the maximum arity of any IDB predicate. Thus, the revised program winds up computing many facts about paths that do not originate at one of the source nodes, even though it will do all this work in a small number of rounds, say 10 rounds if the maximum-length shortest path in the graph is 1000.

1.2 Summary of Paper

We are thus led to consider the question that will be addressed in this paper:

- When is it possible to replace a *linear recursion* (recursive Datalog program with at most one IDB subgoal in the body of any rule) by an equivalent program that:
  a) Requires for its evaluation a number of recursive rounds that is only logarithmic in the domain size for the program, yet
  b) Uses IDB predicates of no greater arity than that of the IDB predicates in the original program?

In Section 2 we consider nonlinear TC algorithms. We introduce the *unique-decomposition property*, which is the property possessed by certain TC algorithms such as the

[1] T. Vassilakis, private communication, March, 2011.

[2] One might wonder, therefore, how a Web analysis such as [11], which appears to have computed the transitive closure of a large Web graph, could be accomplished. In practice, it is possible to find strongly connected components quickly, and thus reduce significantly the number of nodes on which the full TC must be computed. For example, a search for short cycles will enable us to collapse many large groups of nodes to single nodes.

linear algorithms, that assure each path is discovered only once. One such nonlinear algorithm, called "smart TC," has been known for some time. We show that there are others, and that depending on the data, any of these algorithms might outperform the others.

Then, in Section 3, we examine a generalization of TC, called right-linear chain programs. These recursions have the desirable property that they can be converted to nonlinear recursions that use logarithmic rounds only, and yet the arity of the recursive predicates remains the same as in the original linear recursion.

Finally, in Section 4 we consider two examples of linear recursion for which we can prove it is impossible to convert to a recursion that both completes in a logarithmic number of rounds and has recursive predicates with the same arity as the original program. One of these is the reachability problem discussed in Section 1.1. The other is the "same-generation" problem, which has a linear recursion with predicates of arity two, yet requires arity three at least, if it is to be computed in a logarithmic number of rounds.

2. TC ALGORITHMS DISCOVERING EACH PATH ONLY ONCE

Algorithms such as the simple version of nonlinear TC not only discover the same path fact $path(a, b)$ several times if there are several different paths from a to b, but they will discover this fact using the same path several times. For example, if there is a path $a \to x \to y \to b$, then the fact $path(a, b)$ will be discovered once by combining $path(a, x)$ with $path(x, b)$ and again by combining $path(a, y)$ with $path(y, b)$. There are, however, several known algorithms (and many not previously considered) that have the property that a single path is never discovered more than once.

Most obvious of these algorithms are the left- and right-linear versions of TC. That is, if we are constrained to combine an edge (path of length one) with a following path, then the only way to discover $path(a, b)$ in the example above is to combine $edge(a, x)$ with $path(x, b)$. Similarly, if we are constrained to combine a path with a following edge, then the only way to discover $path(a, b)$ is to combine $path(a, y)$ with $edge(y, b)$. More interesting is the algorithm called *Smart TC* [21, 25, 31], which combines two paths only if the first has a length a power of 2 and the second has a length no greater than the length of the first. This algorithm has several desirable features:

1. It discovers paths only once.
2. It requires only $O(\log n)$ rounds to discover all path facts about a graph of n nodes.
3. It has a very clean and succinct iterative implementation.

We call property (1) above the *unique-decomposition* property. For unique-decomposition algorithms, the number of disjoint paths joining two nodes equals number of derivations for the fact in the transitive closure that is the pair of these two nodes.

However, there is an infinite variety of algorithms with the unique-decomposition property. We can characterize each such algorithm by a function from integers to pairs of integers that tells how paths of a certain length are to be discovered.

DEFINITION 2.1. *A* length partition *is a function from integers $\ell \geq 2$ to pairs of positive integers (ℓ_1, ℓ_2) such that $\ell_1 + \ell_2 = \ell$. The TC algorithm induced by a length partition P infers the fact $path(x, y)$ if and only if there exists a node z such that:*

1. *The shortest path from x to z is known to be of length ℓ_1.*
2. *The shortest path from z to y is known to be of length ℓ_2.*
3. *$P(\ell_1 + \ell_2) = (\ell_1, \ell_2)$. Less formally, the shortest path from x to y has length ℓ, and the only way that P allows the inference of paths of length ℓ is by combining a path of length ℓ_1 with a following path of length ℓ_2.*

EXAMPLE 2.2. *The right-linear algorithm constructs paths of length ℓ by combining paths of length 1 with a following path of length $\ell - 1$. That is, the right-linear length partition is $\ell \to (1, \ell - 1)$. Similarly, the left-linear length partition is $\ell \to (\ell - 1, 1)$. The Smart length partition is $\ell \to (\ell_1, \ell_2)$, where ℓ_1 is the largest power of 2 that is strictly less than ℓ and $\ell_2 = \ell - \ell_1$.*

We shall also consider here two other algorithms in the same class, in order to demonstrate several points about the space of options available.

1. The algorithm *Balance* is defined by the length partition $\ell \to (\lceil \ell/2 \rceil, \lfloor \ell/2 \rfloor)$. That is, Balance divides a path into two paths as nearly equal in length as possible.
2. The algorithm *Thirds* uses the length partition $\ell \to (\ell_1, \ell_2)$, where
   (a) ℓ_2 is the larger of 1 and $\lfloor \ell/3 \rfloor$.
   (b) $\ell_1 = \ell - \ell_2$.

   That is, the second path is about 1/3 of the length, and the first path is about 2/3 the length. For example, paths of lengths 2 through 9 are constructed from paths with the following pairs of lengths: $(1, 1)$, $(2, 1)$, $(3, 1)$, $(4, 1)$, $(4, 2)$, $(5, 2)$, $(6, 2)$, and $(6, 3)$.

2.1 Computation in Rounds

Consider any algorithm driven by a length partition P. We can associate with any $path(a, b)$ fact discovered the length of the shortest path from a to b. We can then compute all path facts and their associated lengths in rounds as follows.

Basis: On the first round, all the facts $edge(a, b)$ become $path(a, b)$ facts, with a length of 1.

Induction: Suppose that after round i, we have discovered all path facts with shortest paths of length up to m_i. On round $i+1$, we discover all paths of length ℓ provided $P(\ell)$ is a pair of integers, each no greater than m_i. Since $P(m_i + 1)$ is surely a pair of integers less than or equal to m_i, we know that $m_{i+1} > m_i$, and thus eventually all paths will be discovered.

It is desirable that the number of rounds needed to discover all paths is small. Some length partitions require few rounds, while others require many. The following example analyzes the five algorithms mentioned previously.

EXAMPLE 2.3. *The left- and right-linear TC algorithms each have the property that $m_i = i$. That is, we add only one to the length of discovered paths at each round.*

The Smart algorithm discovers paths in as few rounds as possible. That is, $m_1 = 1$ and $m_2 = 2$, as is the case for all algorithms. But $m_3 = 4$, since on the third round we can discover any path composed of a path of length 2 and a path of length up to 2. Likewise, $m_4 = 8$, since on the fourth round we discover paths composed of a path of length 4 and a path of length up to 4. It is easy to conclude that $m_i = 2^{i-1}$, and thus that Smart requires $1 + \log_2 n$ rounds on a graph of n nodes.

Next, consider the algorithm Balance. This algorithm is defined by the length partition that constructs paths of lengths $2, 3, \ldots$ by the sequence of pairs $(1,1)$, $(2,1)$, $(2,2)$, $(3,2)$, $(3,3)$, $(4,3)$, $(4,4)$, and so on. We again observe that $m_i = 2^{i-1}$ for Balance, which thus uses the same number of rounds as Smart.

Finally, consider the algorithm Thirds. As mentioned above, the sequence of pairs for this algorithm begins $(1,1)$, $(2,1)$, $(3,1)$, $(4,1)$, $(4,2)$, $(5,2)$, $(6,2)$, $(6,3), \ldots$. The discovery of new paths starts slowly, with $m_1 = 1$, $m_2 = 2$, $m_3 = 3$, and $m_4 = 4$. But then $m_5 = 6$, $m_6 = 9$, $m_7 = 13$, and thereafter, the values of m_i grow by a factor of close to $3/2$ each time i grows by one. We conclude that the number of rounds required for Thirds on a graph of n nodes is $O(\log n)$, although the constant of proportionality is not as small as for Smart or Balance. There is some evidence, however, that Thirds in practice produces fewer redundant path facts than these two algorithms; i.e., the number of times Thirds will discover a path fact that was already known is lower (see Section 2.3).

2.2 Ladder Graphs and the Incomparability of Algorithms

While we can compare the algorithms based on different length partitions by the number of rounds that require, we cannot order them strictly by the total work required for their execution. The computation model we shall use is *data-volume cost* from [3]. In this model, we assume TC algorithms are to be executed on a computing cluster. The usage of the cluster is the sum over all compute-nodes used of the amount of time that node is used. This sum is, for simple algorithms like those implementing TC, proportional to the sum over all tasks of the amount of input to that task. Also from [3] is the observation that for TC, the data-volume cost is proportional to the sum over all triples x, y, z of nodes such that the path from x to y and the path from y to z are combined to infer a path from x to z during the execution of the algorithm. These combining events are called *derivations*.

With these observations in mind, we shall define a family of graphs such that the number of derivations executed by different algorithms varies widely. A *ladder graph* is defined as follows:

1. The nodes are organized in *rungs*, numbered $0, 1, \ldots$.
2. The ith rung can have any number $m_i \geq 1$ of nodes.
3. There is a directed edge from every node of rung i to every node of rung $i + 1$.

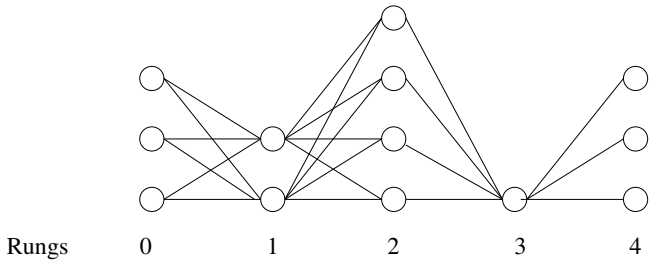

Figure 1: Example of a ladder graph

EXAMPLE 2.4. *Figure 1 is an example of a ladder graph with five rungs numbered 0 through 4. The "ladder" is on its side, with rungs vertical. m_0 through m_4 are 3, 2, 4, 1, and 3, respectively. The edges are directed from left to right, although we do not show the arrows at the ends of the edges.*

An important property of ladder graphs that makes easier our estimate of the work performed by various algorithms is:

- All paths are shortest paths.

If the TC algorithm is defined by a length partition P, there is a simple formula for the number of derivations executed by the algorithm. This number is the sum over all triples of rungs (i, j, k), where $i < j < k$ and $P(k - i) = (j - i, k - j)$, of $m_i m_j m_k$.

To see why, suppose there is a derivation that combines the facts $path(x, y)$ with the fact $path(y, z)$, where nodes x, y, and z are on rungs i, j, and k, respectively. Then the distance from x to y is $\ell_1 = j - i$, the distance from y to z is $\ell_2 = k - j$, and the distance from x to z is $\ell = k - i$. Since the algorithm combines these two paths, it must be that $P(\ell) = (\ell_1, \ell_2)$; i.e., $P(k - i) = (j - i, k - j)$.

Next, consider ladder graphs with n rungs, in which the ith rung has either one node or m nodes, where m is very large compared with n^3. Then the number of derivations executed by the algorithm derived from length partition P is dominated by the number of triples (i, j, k) such that:

1. $m_i = m_j = m_k = m$, and
2. $P(k - i) = (j - i, k - j)$.

EXAMPLE 2.5. *Consider a ladder graph G with $2^r - 1$ rungs. Rungs whose numbers are of the form $2^r - 2^s$, where $2 \geq s \geq r$ have m nodes, and other rungs have one node. For instance, if $r = 4$, then the rungs are numbered $0, 1, \ldots, 14$, and the rungs with m nodes are 0, 8, 12, and 14.*

If we apply Smart to graph G, there are three triples of integers such that involve combining paths between rungs all of which have m nodes; these are $(0, 8, 12)$, $(0, 8, 14)$, and $(8, 12, 14)$. Thus, the data-volume cost of Smart on G will be $\Omega(m^3)$.

On the other hand, the algorithms Balance, Left-linear and Right-linear each combine no triples of rungs all of which have m nodes. Thus, these algorithms have a data-volume cost that is $O(n^3 m^2)$ at worst. Since we assumed m is large compared with n^3, Smart is the worst performing.

But the analysis could just as easily have gone the other way. For example, a ladder graph with rungs $0, 3, 6, 9, \ldots$ having m nodes and the other rungs having 1 node demonstrates that Balance can be the worst of all as well.

2.3 Comparison on a Fixed Graph

In this section, we shall try to estimate typical behavior of the various algorithms by examining a particular graph, shown in Fig. 2, that represents the explosion of paths between two nodes that occurs in dense random graphs. This graph does, however, share with the ladder graphs the property that all paths are shortest paths, which simplifies the counting.

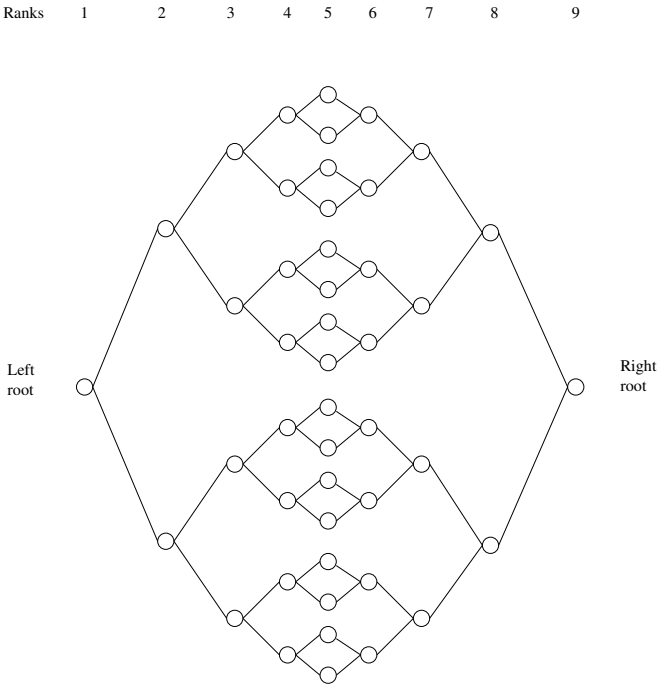

Figure 2: Graph with many paths

The graph of Fig. 2 is two complete binary trees with leaves shared. All arcs go from left to right, although we have omitted the arrows on the edges. There are 279 pairs of nodes (a, b) such that a reaches b along a path of length one or more. Of these paths, 60 are the arcs, and the remaining 219 pairs are connected by paths of length 2 or more. It is only the paths of length ≥ 2 that need to be constructed recursively. So, we are going to count only derivations using the recursive rule, ignoring the 60 derivations for the basis rule that is present in all algorithms.

Now, consider the right-linear algorithm, where paths of length 1 are combined with paths of any length in the recursive rule. Although there are only 219 path facts, some facts are discovered several times. For instance, the path from a node at rank 4 to the right root is discovered by two separate derivations, corresponding to the two nodes of rank 5 that are reached by arcs from each node at rank 4. The total number of derivations using the recursive rule is 268. This number should be compared with 219, the theoretically smallest number of derivations any recursive rule could have, because 219 is the number of path facts that must be derived recursively by any TC algorithm.

For the algorithms Smart, Balance and Thirds, it is easier to count the number of derivations by looking at every length pair that is relevant to the graph of Fig. 2. For Balance, the length pairs that apply are $(1,1)$, $(2,1)$, $(2,2)$, $(3,2)$, $(3,3)$, $(4,3)$, and $(4,4)$. The total number of derivations is 316, which somewhat larger than the count for the linear algorithms. Of course the benefit of algorithms such as Balance is that the number of rounds required is $O(\log n)$ rather than $O(n)$ as it is for the linear algorithms.

Next, consider the Smart TC algorithm. The length pairs that apply are $(1,1)$, $(2,1)$, $(2,2)$, $(4,1)$, $(4,2)$, $(4,3)$, and $(4,4)$. Except for the fourth and fifth in this sequence, the pairs are the same as for Balance. The total number of derivations is 312.

Last, consider the Thirds TC algorithm, which for the graph of Fig. 2 is characterized by the sequence of length pairs $(1,1)$, $(2,1)$, $(3,1)$, $(4,1)$, $(4,2)$, $(5,2$, and $(6,2)$. The total number of derivations is 288. It should not be surprising that this sum is between what we get from the linear algorithms and the Smart or Balance algorithms, since the number of rounds needed by Thirds also lies between these two pairs of algorithms.

2.3.1 Calculations for the General Case

Here we calculate the number of path facts (pairs of nodes connected by one or more paths) and the number of distinct paths in the general case of the paired trees graph with two trees of height H. Notice that when a TC algorithm has the unique-decomposition property, the number of distinct paths equals the number of derivations and therefore is the appropriate measure of performance on a computing cluster. It turns out that the number of path facts and paths do not differ much (e.g., for data of 1TB, their ratio is approximately 10) giving formal evidence that on this graph the various algorithms in our class have similar performance.

We have rank from 1 through to $H+1$ (the leaves) to $2H+1$ (the right root). (H is the number of edges from the root to a leaf. In the figure $H = 4$.)

The total number of path facts is

$$2^H - 1 + (H+1) \times 2^{H+1} - 2^{H+2} + 2 + (H+1) \times 2^{H+1} - 2^{H+2} + 2 + (H-1)2^H - 2^H + 2 + (H-1)2^H - 2^H + 2$$

The total number of distinct paths is

$$2^{H+1} + H(H-1)2^H + H(H-1)2^H + (H-1)2^H - 2^H + 2 + (H-1)2^H - 2^H + 2$$

The ratio of the two numbers computed above (number of paths and number of path facts) is less than $H/3 + 3$ and larger than $H/3$. Thus if $n = 3 \times 2^H = 1TB$, then we have that $H \approx 10$. The number of derivations of any of the four algorithms will be between these two numbers. Thus we expect the four algorithms not to have large deviations in performance, at least not larger than a factor of $\log n$ on the size n of the data. In the example the ratios were much smaller than that and indicated a constant ratio (of the order of 2). This can be explained by observing that the large term in counting the number of path facts was contributed by the asymmetrical facts. When computing such facts, however the four algorithms need many fewer derivations than the number of different paths that lead from the first node of the computed fact to its second node. The two linear algorithms need two derivations for each fact. The other two algorithms need a larger number of derivations but only for a small fraction of the facts. Thus this does not worsen them very much. In particular, for the asymmetrical facts, the number of derivations is only large when the two

nodes of a fact are not "very asymmetrical," i.e., when their distances from the leaves are very close to each other. If not, then the number of derivations of this fact is much smaller than the number of distinct paths between its nodes.

3. RECURSIVE DOUBLING FOR LINEAR CHAIN PROGRAMS

Many years ago, [23] showed how to convert any linear Datalog program into a transitive closure plus small pieces for initialization and for extracting the result. The motivation they expressed was that thus an efficient implementation of TC could serve as an implementation of all linear recursions. Today, there is another motivation for studying this question in the context of cluster computing. As we discussed in Section 1.1, we would like to implement all recursions, or at least all linear recursions, in a way that uses a logarithmic number of rounds, and yet does not increase significantly the number of facts we must derive.

In Section 4.1 we shall show that the reachability query requires either binary IDB predicates or $\Omega(n)$ rounds on databases of size n. In Section 4.2 we shall show that the same-generation query requires either ternary IDB predicates or $\Omega(n)$ rounds on a databases of size n. Here, we explore a class of linear Datalog programs that can be implemented without increasing the domain size significantly, and yet require only a logarithmic number of passes to complete.

3.1 Some Examples of the Rounds-Arity Tradeoff

By using the nonlinear version of TC, that is,

```
path(X,Y) :- arc(X,Y)
path(X,Y) :- path(X,Z) & path(Z,Y)
```

we can evaluate *path* in $O(\log n)$ rounds on an n-node graph, and yet only have to construct *path* facts with two arguments, just as we would if we used a linear recursion such as

```
path(X,Y) :- arc(X,Y)
path(X,Y) :- arcX,Z) & path(Z,Y)
```

It appears not to be the case for all linear recursions. An example is the "same-generation" query for finding cousins:

```
sg(X,Y) :- par(X,Z) & par(Y,Z)
sg(X,Y) :- par(X,Xp) & par(Y.Yp) & sg(Xp,Yp)
```

In the above query, $par(A, B)$ is an EDB relation meaning that B is a parent of A. The IDB predicate $sg(A, B)$ means that A and B have a common ancestor the same number of generations back (i.e., they are cousins, or siblings if the number of generations is one).

While we can compute the above recursion discovering only facts that involve two arguments (i.e., pairs of individuals), it appears that if we want to use a nonlinear recursion that gets the same result in only $O(\log n)$ rounds on a database involving n individuals, then we need to compute an IDB predicate of arity 4. A program such as:

```
implies(W,X,Y,Z) :- par(W,Y) & par(X,Z)
implies(W,X,Y,Z) :- implies(W,X,A,B) &
                        implies(A,B,Y,Z)
sg(X,Y) :- implies(X,Y,Z,Z)
```

will serve. Here, $implies(W, X, Y, Z)$ should be interpreted as "if Y and Z are either cousins or the same individual, then W and X are cousins.

3.2 Right-Linear Chain Rules

DEFINITION 3.1. *A Datalog program is* linear *if no rule body has more than one IDB subgoal. A* right-linear chain program *consists of rules of the following forms:*

$$p(X,Y) \text{ :- } e(X,Z) \ \& \ q(Z,Y)$$
$$p(X,Y) \text{ :- } e(X,Y)$$

where in each rule, X, Y, and Z are distinct variables, e is an EDB predicate, and p and q are IDB predicates.

There are several extensions to the above forms. The variables could be sequences of distinct variables, and the order of the arguments in the various subgoals need not be as shown. Further, the subgoal with predicate e could be a sequence of EDB subgoals, and these subgoals could involve variables not shown, as long as Y is not among them. However, what is essential is that the head predicate and the IDB predicate in rules of the first form share the attribute Y (which could represent several attributes). We shall not consider these straightforward extensions here.

3.3 Expansions and Derivation Trees

In the discussion that follows, we assume that the reader is familiar with Datalog and conjunctive queries (CQ), using the notation of [28]. In particular, you should understand that a Datalog rule or CQ consists of a head and a body with one or more subgoals. Variables that appear in the head are "distinguished," while variables that appear only in the body are "nondistinguished." The following definitions will be used in proofs that follow.

DEFINITION 3.2. *A* canonical database *of a conjunctive query is the database that is formed by the subgoals in the body of the query if we* freeze *the variables (replace each variable by a unique constant).*

Let Π be a Datalog program that computes a predicate p. Let Q be a conjunctive query with head p and a body that results from unfolding a rule with head p several times, until there are no IDB predicates in the body. By "unfold" we mean that an IDB subgoal $q(\cdots)$ in the body is replaced by the body of a rule with head q. During this replacement, we first unify the head of the rule with the subgoal, and use new variables for all the nondistinguished variables of the body. Such a query Q is called an expansion *of Π.*

The process of expanding a goal (predicate with variables for arguments) can be shown as a derivation tree. *This tree has the goal at the root. Each interior node represents an IDB subgoal at some stage of the expansion, and its children are the subgoals with which it was replaced. The leaves are EDB subgoals.*

Each derivation tree of a Datalog program Π has an expansion of Π that corresponds to it and is constructed by following the unfolding that is depicted by the derivation tree. Thus, in the proofs we will talk about a derivation tree and its *corresponding* expansion. The relation between expansions and derivation trees was noticed in [13] (where derivation trees are called "expansion trees") and in [2, 4] (where derivation trees are called "skeleton trees").

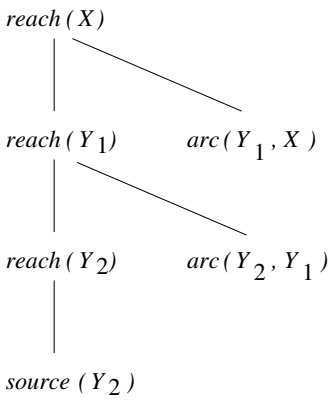

Figure 3: A derivation tree

EXAMPLE 3.3. *Figure 3 is a derivation tree for the reach program from Section 1.1:*

```
reach(X) :- source(X)
reach(X) :- reach(Y) & arc(Y,X)
```

The goal at the root is $reach(X)$, and this goal was expanded using the recursive rule. We chose Y_1 as the nondistinguished variable in place of Y, to avoid accidently using the same variable twice. The subgoal $reach(Y_1)$ was also expanded using the recursive rule. Variable X was unified with Y_1, and nondistinguished variable Y replaced by Y_2. Finally, the subgoal $reach(Y_2)$ was expanded using the basis rule.

$$reach(X) \text{ :- } source(Y_2) \ \& \ arc(Y_2, Y_1) \ \& \ arc(Y_1, X)$$

is the expansion corresponding to this derivation tree.

3.4 Recursive Doubling for Right-Linear Chain Programs

THEOREM 3.4. *For every right-linear chain program, there is an equivalent program with the following properties:*

1. *All IDB predicates are binary.*
2. *The program can be evaluated in $O(\log n)$ rounds on a database of size n.*

The proof of Theorem 3.4 involves a construction and a sequence of lemmas about this construction. Given a right-linear chain program Π, we construct a program Π' meeting the conditions of the theorem. The IDB predicates of Π' are the IDB predicates of Π plus:

$$\{t_{pq}(U, V) \mid p \text{ and } q \text{ are IDB predicates of } \Pi\}$$

The intent of t_{pq} is that $t_{pq}(a, b)$ is true if and only if for all c, $q(b, c)$ implies $p(a, c)$. The rules of Π' are as follows:

1. For all IDB predicates p, q, and r,

$$t_{pq}(X, Y) \text{ :- } t_{pr}(X, Z) \ \& \ t_{rq}(Z, Y)$$

2. If $p(X, Y)$:- $e(X, Z)$ & $q(Z, Y)$ is a rule of Π, then Π' has the rule

$$t_{pq}(X, Z) \text{ :- } e(X, Z)$$

3. Each basis rule $p(X, Y)$:- $e(X, Y)$ of Π is also a rule of Π'.

4. If p and q are IDB predicates of Π, then Π' has the rule

$$p(X, Y) \text{ :- } t_{pq}(X, Z) \ \& \ q(Z, Y)$$

EXAMPLE 3.5. *Consider the following Datalog program Π:*

```
p(X,Y) :- r(X,Z) & q(Z,Y)
q(X,Y) :- b(X,Z) & p(Z,Y)
p(X,Y) :- r(X,Y)
```

Intuitively, r and b are EDB predicates representing "red" and "blue" arcs, respectively. IDB predicate p represents paths that alternate red and blue arcs, but both begin and end with a red arc. IDB predicate q also represents paths of alternating color, but its paths begin with blue and end with red.

Part 1 of the construction of Π' gives us the following eight rules:

$$\begin{array}{l} t_{pp}(X, Y) \text{ :- } t_{pp}(X, Z) \ \& \ t_{pp}(Z, Y) \\ t_{pp}(X, Y) \text{ :- } t_{pq}(X, Z) \ \& \ t_{qp}(Z, Y) \\ t_{pq}(X, Y) \text{ :- } t_{pp}(X, Z) \ \& \ t_{pq}(Z, Y) \\ t_{pq}(X, Y) \text{ :- } t_{pq}(X, Z) \ \& \ t_{qq}(Z, Y) \\ t_{qp}(X, Y) \text{ :- } t_{qp}(X, Z) \ \& \ t_{pp}(Z, Y) \\ t_{qp}(X, Y) \text{ :- } t_{qq}(X, Z) \ \& \ t_{qp}(Z, Y) \\ t_{qq}(X, Y) \text{ :- } t_{qp}(X, Z) \ \& \ t_{pq}(Z, Y) \\ t_{qq}(X, Y) \text{ :- } t_{qq}(X, Z) \ \& \ t_{qq}(Z, Y) \end{array}$$

Part 2 gives us the rules:

$$\begin{array}{l} t_{pq}(X, Z) \text{ :- } r(X, Z) \\ t_{qp}(X, Z) \text{ :- } b(X, Z) \end{array}$$

From Part 3 we get the basis rule

$$p(X, Y) \text{ :- } r(X, Y)$$

and from Part 4 we get

$$\begin{array}{l} p(X, Y) \text{ :- } t_{pp}(X, Z) \ \& \ p(Z, Y) \\ p(X, Y) \text{ :- } t_{pq}(X, Z) \ \& \ q(Z, Y) \\ q(X, Y) \text{ :- } t_{qp}(X, Z) \ \& \ p(Z, Y) \\ q(X, Y) \text{ :- } t_{qq}(X, Z) \ \& \ q(Z, Y) \end{array}$$

We now turn to the proof of Theorem 3.4. First, any derivation tree for a right-linear chain program has a spine of IDB subgoals on the right. Moreover, each of these subgoals has the same variable in the second argument. Figure 4 suggests what these derivation trees look like. Suppose we apply the program to a database with n different constants, and the program has k different IDB predicates.

When we instantiate the variables of the derivation tree with constants, to make all EDB subgoals true, then there are at most kn different instantiated IDB subgoals that can appear along the spine. If the spine has more than kn IDB predicates, then two are the same, and we do not have the smallest derivation tree for the fact at the root. We have thus proved:

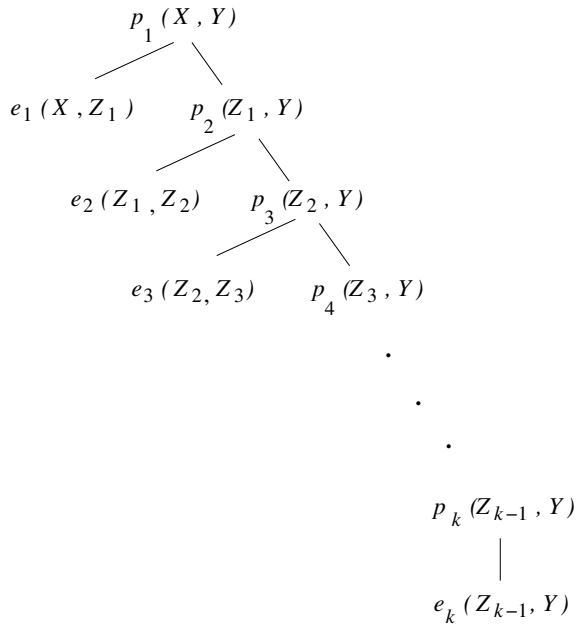

Figure 4: A derivation tree for a right-linear chain program

LEMMA 3.6. *If a right-linear chain program* Π *with* k *IDB predicates is applied to a database* D *with* n *different constants appearing among its tuples, then every fact derived from* D *by* Π *has a derivation tree of height at most* kn.

Next, let us examine the rules of the first two groups in the construction of Π'; these are the rules that define the IDB predicates t_{pq}. A *partial* derivation tree is like a derivation tree, except some of the leaves may be IDB subgoals. For a right-linear chain program, a partial derivation tree has a subgoal $p(X, A)$ at the root, an IDB subgoal $q(Y, A)$ as the rightmost leaf (the bottom of the "spine"), and appropriate EDB leaves coming off the spine to the left. Note that if the EDB predicates can be made true, then this tree can be interpreted as saying "for all A, if $q(Y, A)$ then $p(X, A)$," and is thus a justification for $t_{pq}(X, Y)$.

LEMMA 3.7. *After* i *rounds of application of the rules of* Π', $t_{pq}(x, y)$ *will be derived if and only if there is a partial derivation tree with* $p(X, A)$ *at the root,* $q(Y, A)$ *as the rightmost leaf, height at most* 2^{i-1}, *and a homomorphism from variables of the tree to constants that makes all of the EDB leaves true in the database to which* Π' *is applied.*

PROOF. We proceed by induction on i. For the basis, $i = 1$, the inference must be from a rule of Group 2. These rules let us infer $t_{pq}(x, y)$ from the database fact $e(x, y)$. But then, there must be a rule of Π (possibly with different variables):

$$p(X, A) \text{ :- } e(X, Y) \ \& \ q(Y, A)$$

We see immediately that a partial derivation tree with root $p(X, A)$ and two children $e(X, Y)$ and $q(Y, A)$ exists. We also know that if we substitute constant x for variable X and constant y for variable Y, then the EDB subgoal $e(x, y)$ is true. Thus, if the fact is inferred at the first round of Π', the tree exists. Conversely, if the tree exists and the substitution makes the EDB subgoal true, then there is a Group 2 rule that allows us to make the inference at the first round of Π'.

For the inductive part, we use the Group 1 rules of Π' that combine two t predicates. Suppose that on round i we derive $t_{pq}(x, y)$ from $t_{pr}(x, z)$ and $t_{rq}(z, y)$. From the latter two facts, we know there are two partial derivation trees. The first has root $p(X, A)$ and rightmost leaf $r(Z, A)$ for some variable A. The second has root $r(Z, A)$ and rightmost leaf $q(Y, A)$.[3] Both trees are of height at most 2^{i-2}, since they correspond to facts derived at round $i - 1$ or before. We may splice the two trees together, by identifying the root of the second with the rightmost leaf of the first. We also know from the inductive hypothesis that both trees have substitutions that make all their EDB leaves true, and both of these substitutions replace variable Z by constant z. Since there are no variables but Z that are shared by the two trees, we can construct a substitution for the combined tree that makes all of its EDB leaves true and also maps X to x and Y to y. These observations prove that if $t_{pq}(x, y)$ is derived on round i, then there is a partial derivation tree as described in the lemma.

Conversely, suppose there is a partial derivation tree T of height at most 2^i in Π. Let the tree have root $p(X, Y)$ and rightmost leaf $q(Z, Y)$. Let $r(W, Y)$ be the IDB subgoal that is most nearly in the middle of the spine of T. If we break T into two at the node $r(W, Y)$, we have two partial derivation trees T_1 and T_2, each of height at most 2^{i-1}. Any homomorphism that instantiates T to have true EDB subgoals also instantiates T_1 and T_2 and makes their EDB subgoals true. Let this instantiation map X, Y, Z, and W to a, b, c, and d, respectively. Then by the inductive hypothesis, on or before round $i-1$ we infer $t_{pr}(a, d)$ because of the instantiated T_1 and $t_{rq}(d, c)$ because of the instantiated T_2. Applying a Group 1 rule to these two facts lets us infer $t_{pq}(a, c)$ by round i. □

Now, we can prove Theorem 3.4.

PROOF. Suppose we derive $p(a, b)$ in Π. Let the derivation tree for this derivation be T. By Lemma 3.6, T can have height at most n, the number of constants in the database. Let this height be h. If $h = 1$, then there must be a rule $p(X, Y)$:- $e(X, Y)$ of Π. If so, this rule is also a rule of Π' in the third group. Moreover, $e(a, b)$ must be true in the database. Thus, $p(a, b)$ is derived by Π' on the first round.

Suppose $h > 1$. Like all derivation trees of a linear-chain program, T has a spine of IDB subgoals going down the right, and each interior node has one EDB child as well. We can instantiate the variables of T so that the root is $p(a, b)$, the rightmost IDB subgoal is $q(c, b)$, and the rightmost EDB subgoal is $e(c, b)$ for some predicates q and e and some constant c. This instantiation makes all the EDB subgoals true.

If we remove the rightmost EDB subgoal $e(X, Y)$, we get a partial derivation tree of height $h - 1$, By Lemma 3.7 we know that we can infer $t_{pq}(a, c)$ by round $\log_2 h - 1$ of Π'. Moreover, we know $q(X, Y)$:- $e(X, Y)$ is a rule of Π and Π'. Since we know $e(c, b)$ is true, we can apply the rule of Group 4

$$p(X, Y) \text{ :- } t_{pq}(X, Z) \ \& \ q(Z, Y)$$

[3]technically, the second tree might not have Z and A as the variables at the root, but we can always substitute for whatever variables actually appear there.

to infer $p(a,b)$ in Π' by round $\log_2 h$. As $h \leq n$, we see that all facts inferred by Π on any round are inferred by Π' by round $\log_2 n$.

To complete the proof, we must observe that any fact involving an IDB predicate of Π that is derived in Π' is also derived in Π. Then we can conclude that Π and Π' are equivalent. This part of the proof turns on the interpretation of the t predicates that we suggested. In particular, we claim that if in Π' we can infer $t_{pq}(a,b)$, then for all constants c, $q(b,c)$ implies $p(a,c)$ in Π. This statement is a simple induction on the number of times we apply rules of Groups 1 and 2 in Π'.

Then, we can show by induction on the number of times we apply any rule of Π' that any derivation in Π' of a fact $p(a,b)$, where p is an IDB predicate of Π implies that $p(a,b)$ is also true in Π. Such a derivation must either end with a rule of Group 3 or 4. If of Group 3, it is obvious that $p(a,b)$ is also derived in Π. If of Group 4, then we know that for some constant c, $t_{pq}(a.c)$ is derived in Π' and so is $q(c,b)$, by the inductive hypothesis. The first tells us (changing names of constants to fit the situation) that for all b, $q(c,b)$ implies $p(a,b)$ in Π, while the second tells us $q(c,b)$ is true in Π for the particular b in question. Thus, $p(a,b)$ is true in Π. □

THEOREM 3.8. *For every linear program Π , there is an equivalent program Π' with the following properties:*

1. *The maximum arity of IDB predicates in π' is twice the maximum arity of IDB predicates in Π.*
2. *The program can be evaluated in $O(\log n)$ rounds on a database of size n.*

The proof of Theorem 3.8 involves a similar construction as in the proof of Theorem 3.4 and a sequence of lemmas about this construction which are again similar with the lemmas for Theorem 3.4. Actually, their proof is easier than the proofs for Theorem 3.4 since the construction is more straightforward because we do not have to do a trick to save on the arity. Hence, we give the construction here with an example and we omit the lemmas and their proof.

Given a right-linear chain program Π, we construct a program Π' meeting the conditions of the theorem. The IDB predicates of Π' are the IDB predicates of Π plus:

$$\{t_{pq}(U_1,\ldots,U_k,V_1,\ldots,V_m) \mid p \text{ and } q \text{ are IDB predicates of } \Pi \text{ of arity } k \text{ and } m \text{ respectively } \}$$

The intent of t_{pq} is that $t_{pq}(a_1,\ldots,a_k,b_1,\ldots,b_m)$ is true if and only if $q(b_1,\ldots,b_m)$ implies $p(a_1,\ldots,a_k)$. The rules of Π' are as follows:

1. For all IDB predicates p, q, and r,

$$t_{pq}(X_1,\ldots,X_k,Y_1,\ldots,Y_m) \text{ :- } t_{pr}(X_1,\ldots,X_k,Z_1,\ldots,Z_n) \;\&\; t_{rq}(Z_1,\ldots,Z_n,Y_1,\ldots,Y_m)$$

2. If $p(X_1,\ldots,X_k)$:- $e(\hat{Z})$ & $q(Y_1,\ldots,Y_m)$ is a rule of Π (where $\hat{Z}$ is a vector of variables that may contain some variables X_i, some variables Y_i and other variables), then Π' has the rule

$$t_{pq}(X_1,\ldots,X_k,Y_1,\ldots,Y_m) \text{ :- } e(\hat{Z})$$

3. Each basis rule $p(X_1,\ldots,X_k)$:- $e(\hat{Z})$ of Π is also a rule of Π'.

4. If p and q are IDB predicates of Π, then Π' has the rule

$$p(X_1,\ldots,X_k) \text{ :- } t_{pq}(X_1,\ldots,X_k,Y_1,\ldots,Y_m) \;\&\; q(Y_1,\ldots,Y_m)$$

EXAMPLE 3.9. *From*

```
sg(X,Y) :- par(X,Z) & par(Y,Z)
sg(X,Y) :- par(X,Xp) & par(Y.Yp) & sg(Xp,Yp)
```

we get

```
implies(W,X,Y,Z) :- par(W,Y) & par(X,Z)
implies(W,X,Y,Z) :- implies(W,X,A,B) &
                        implies(A,B,Y,Z)
sg(X,Y) :- par(X,Z) & par(Y,Z)
sg(X,Y) :- implies(X,Y,W,Z), sg(W,Z)
```

Part 1 gives us the second rule; here predicate implies stands for $t_{sg,sg}$. Part 2 gives us the first rule. Part 3 gives us the third rule. Part 4 gives us the fourth rule.

The above program is easily seen to be equivalent with what we wrote before which we repeat here:

```
implies(W,X,Y,Z) :- par(W,Y) & par(X,Z)
implies(W,X,Y,Z) :- implies(W,X,A,B) &
                        implies(A,B,Y,Z)
sg(X,Y) :- implies(X,Y,Z,Z)
```

4. THE ITERATION-ARITY TRADEOFF

The number of attributes in the derived relations in a Datalog program (i.e., the arity of the IDB predicates) constrains what kind of queries can be expressed. Here we examine this matter in more detail. In some cases, a query can be expressed with IDB predicates of low arity, but only if the depth of the derivation trees (i.e., the number of rounds or iterations in a recursive implementation of the Datalog program) is high.

Next we will show that there is a tradeoff between the number of rounds and the arity.

4.1 The Reachability Query

Let us begin by examining the reachability query

```
reach(X) :- reach(Y) & arc(Y,X)
reach(X) :- source(X)
```

As we mentioned in Section 1.1, the IDB predicate *reach* is of arity 1, and it could take $n-1$ rounds to compute *reach* on a graph of n nodes. Now, we shall prove that this tradeoff is inherent; any Datalog program Π with IDB predicates of arity 1 that computes *reach* from EDB predicates *source* and *arc* must take $\Omega(n)$ rounds on an n-node graph.

DEFINITION 4.1. *Consider a derivation tree T for the program Π and its expansion E. Note that E has only arc and source subgoals, although T may have any number of IDB predicates of arity 1, of which one is* reach. *In E we say there is a* path *$X_0, X_1, \ldots, X_k$ from variable X_0 to variable X_k if for all $i = 0, 1, \ldots, k-1$ there is a subgoal $arc(X_i, X_{i+1})$. The* length *of this path is k.*

The following theorem says that there is no Datalog program of maximum IDB arity equal to one that computes the reachability query and is such that each true fact can be computed in less than a linear number of rounds.

THEOREM 4.2. *Suppose* Π *is a Datalog program for reachability with IDB predicates of arity 1. Then there is a constant* $c > 0$*, depending on* Π*, such that when* Π *is applied to EDB relations arc and source, the height of any derivation tree for the fact* $reach(a)$ *is at least* c *times the length of the shortest path from the source node to* a*.*

PROOF. We first prove the following inductive hypothesis by induction on h:

Inductive hypothesis: A node $p(X)$ with a derivation tree of height h has a corresponding expansion in which for every variable Y, either Y has no path to X, or Y is connected to X via a path of length at most kh, where k is the size of the largest body in Π.

The basis, $h = 0$, is vacuous. To prove the inductive step, consider a tree of height $h \geq 1$ with root $p(X)$, and assume the hypothesis for all trees of length less than h. There are at most k children of the root, so any variable Y, found among the children, that has a path to X will be connected by a path of length at most k. Suppose $q(Y)$ is an IDB subgoal that is a child of the root. All variables in its subtree must have a path to X through Y (if they have a path to X at all), because all the variables in that subtree, other than Y were nondistinguished variables introduced at some stage of the expansion and do not appear either among the children of the root or in any other subtree. Let Z be any such variable. By the inductive hypothesis, if there is a path from Z to Y it is of length at most $k(h-1)$. But any path from Z to X must go through Y, and therefore, if there is any path from Z to X there is a path of length at most $k(h-1)+k = kh$.

We have proved that any expansion of a program with IDB predicates of arity 1 and EDB predicates *arc* and *source* contains paths that reach the variable in the root node and are of length at most kh, where h is the height of the corresponding derivation tree. Now we shall prove that given a database D over relations *arc* and *source*, there is no derivation tree for the fact $reach(a)$ of height less than c times the length of the shortest path in D from a source node to a, where $c = 1/k$.

Let D be a database that has $source(v_0)$ and $arc(v_i, v_{i+1})$ for $i = 1, 2, \ldots, n-1$. That is, force D to be a single path. Now, we know $reach(v_n)$ is true, so we must have an expansion E with head $reach(X)$ and subgoals that are either *arc* or *source* subgoals; there is at least one *source* subgoal. Further, E maps homomorphically, by a homomorphism μ from variables of E to nodes of the database D in such a way that all the subgoals are made true. In particular, X maps to v_n.

Suppose this expansion corresponds to a tree of height less than cn, where $c < 1/k$, and k is the parameter associated with constant c and the particular Datalog program Π. Then any variable in E that has any path to X has a path to X of length less than n. There are two cases:

Case 1: E has a subgoal $source(Y)$, Y has a path to X, and the homomorphism μ maps Y to v_0. This path involves fewer than n variables of E. When we apply μ to these variables we get a sequence of nodes in the database D that are connected by *arc* facts and connect v_0 to v_n. However, we have a contradiction, because we know that in D there is no path of length less than n that connects v_0 to v_n.

Case 2: There is no such subgoal $source(Y)$ for which there is a path from Y to X. Then consider a database D' constructed from D as follows. D' has all the nodes $v_0, v_1, \ldots, v_n$ and also has new nodes $u_0, u_1, \ldots, u_n$. In D', the fact $arc(v_i, v_{i+1})$ is true, and so are $arc(u_i, u_{i+1})$ and $arc(v_i, u_{i+1})$ for $i = 0, 1, \ldots, n-1$. Note that $arc(u_i, v_{i+1})$ is not true. In D', $source(u_0)$ is true, but not $source(v_0)$. Consider a new homomorphism μ' from E to D'. If Z has a path to X in E, then $\mu'(Z) = \mu(Z)$. In particular, $\mu'(X) = v_n$. However, if Z has no path to X, then $\mu'(Z)$ is that u_i such that $\mu(Z) = v_i$. Whenever $source(Z)$ is a subgoal of E, we know Z has no path to X. We also know $\mu(Z) = v_0$. Thus, $\mu'(Z) = u_0$, which makes this subgoal true. Whenever $arc(W, Z)$ is a subgoal, we know that if Z has a path to X then W has a path to X. We also know μ mapped this subgoal to a true fact in D. Therefore, μ' will map it to a true fact in D'. We conclude that Π derives $reach(v_n)$ when applied to database D', since in the expansion E the homomorphism μ' maps X to v_n and maps the body of E to true facts of D'. Since in D', v_n is not reached from any source, we again have a contradiction. □

4.2 The Same-Generation Query

Now we consider the same-generation query

```
sg(X,Y) :- par(X,Z) & par(Y,Z)
sg(X,Y) :- par(X,Xp) & par(Y,Yp) & sg(Xp,Yp)
```

The following theorem essentially says that any Datalog program with IDB predicates of arity 2 that computes *sg* from EDB predicate *par* cannot take $o(n)$ rounds on all databases of size n for all n.

A fact $sg(A, B)$ in a database D can be proved in various different ways, e.g., either because there is a common parent, (i.e., $par(C', A)$ and $par(C', B)$ are true), because there is a common ancestor C which is a grandparent to both i.e., $par(C, A_1)$, $par(A_1, A)$ $par(C, B_1)$ and $par(B_1, B)$ are true, and so on. All the ways have a common feature: a pair of paths of the same length leading to two individuals from a common ancestor. Thus we say that a pair of nodes A, B has an *equal-length pair of paths* if there are two sequences of facts $par(C, A_1), par(A_1, A_2), \ldots par(A_n, A)$ and $par(C, B_1), par(B_1, B_2), \ldots par(B_n, B)$. The number of facts in each sequence (i.e, $n+1$ here) is called the *length* of the equal-length pair of paths.

THEOREM 4.3. *Suppose* Π *is a Datalog program for same-generation with all IDB predicates of arity less than 3. Then there is a constant* $c > 0$*, depending on* Π*, such that when* Π *is applied to EDB relation* arc*, the height of any derivation tree for the fact* $sg(a_1, a_2)$ *is at least* c *times the length of the shortest equal-length pair of paths for the pair of nodes* (a_1, a_2)*.*

The proof of the theorem is bases on focusing on *type 1* nodes in the derivation tree. These are nodes such that both their variables have paths in the corresponding expansion to the variables in the root of the derivation tree. First we prove a lemma which says that if, in the derivation tree of expansion E, nodes of type 1 contain "short" paths (i.e., of length bounded by a constant c) in their expansions then all paths in E are of length at most linear on the height of the corresponding derivation tree of E. To prove the

lemma we prove an inductive hypothesis which is similar to the inductive hypothesis in the proof of Theorem 4.2 only more complicated. Then we consider two cases: The first case (Case A) assumes an expansion E with corresponding derivation tree (a particular expansion that computes the *sg* query on a particular database) where all type 1 nodes contain short paths in their expansion. Thus we can use the lemma and proceed with the proof in this case, in a similar way as we proceed with the proof of Theorem 4.2 – again with more complications because of the arity being greater than 1. The second case (Case B) is proved to be impossible by using a pumping argument.

5. BEYOND LINEAR RECURSION

Besides linear programs, there is a wider class of programs that can be efficiently parallelized. It is the class of programs that have the *polynomial-fringe property* (hereafter PFP). [29] defined the PFP, and showed that all Datalog programs with the PFP are in **NC**. A program has the PFP if all true facts have a proof tree with a number of leaves that is polynomial in the data size. For example, all linear recursions have the PFP. [5] examined the common case of single chain-rule Datalog programs (rules of the form $H(x_0, x_n) \leftarrow P_1(x_0, x_1)$ `AND` $\cdots$ `AND` $P_n(x_{n-1}, x_n)$) and divided them completely between those that are in **NC** and those that are P-complete. The algorithm in [29] can be used for any program with the PFP to obtain an equivalent program that can be executed in polylog rounds and in which TC is an integral part of (in a similar way as when we transform linear programs [23] to take advantage of TC). Thus we can state the theorem:

THEOREM 5.1. *Let* Π *be a program with the PFP. Then there is another program* Π' *which is equivalent to* Π *and can be executed in* $\log^2 n$ *rounds.*

The proof is based on the following lemma whose proof comes from the techniques of [29]:

LEMMA 5.2. *If a true fact can be derived from a derivation tree of* Π *of fringe size F, then it can be derived from a derivation tree of* Π' *using the same fringe but in* $\log^2$ *rounds.*

6. CONCLUSIONS

We have addressed the endgame problem — reducing the number of rounds of a recursion to avoid sending large numbers of small files between compute nodes. For transitive closure, it is possible to solve the endgame problem by using a nonlinear recursion. Modifications to the standard nonlinear transitive closure that have the unique-decomposition property can be expected to do approximately the same amount of work as linear TC, and yet use many fewer rounds. The best choice among unique-decomposition algorithms is data dependent, and it is impossible to declare one algorithm best under all circumstances.

Unfortunately, not all linear recursions are like TC. While it is possible to reduce any linear recursion to TC, doing so can square the number of facts that the recursion must infer. We have investigated whether it is possible to turn a linear recursion into a nonlinear equivalent that takes a logarithmic number of rounds to complete and yet does not increase by more than a constant factor the total work (data-volume cost) of execution. We proved that all right-linear chain recursions have this desired property. However, for two significant examples — reachability and same-generation — we showed that it is impossible to get a sublinear number of rounds without increasing the arity of the recursive predicate(s) and thus increasing significantly the number of facts with which the recursion must deal.

Reachability has a lot of applications including social networks, XML databases, bio-informatics, model verification. The bibliography on this topic, both past and recent is huge; some recent work on indexing techniques for fast reachability computation can be found in [10, 14, 15, 24, 32, 33].

6.1 Open Questions

We have begun investigation into the matter of when a linear recursion can be replaced by a nonlinear recursion with similar data-volume cost that needs only logarithmic (in the data size) rounds to complete. We suggest:

1. Find classes of linear recursions more general than the right-linear chain recursions (or the obviously equivalent left-linear class) for which one can guarantee logarithmic rounds with no increase in the arity of recursive predicates.

2. For any such class discovered in (1), are there unique-decomposition variants of the nonlinear recursion? Can we argue that these variants are comparable in data-volume cost to the original linear recursions?

3. Find general classes of linear recursions for which we can prove no equivalent recursion that completes in logarithmic rounds can use only predicates of the same arity as the linear recursion.

4. It is reasonable to assume that when the arity of recursive predicates is increased, the number of facts deduced during the recursion grows significantly, and such is the case in the examples we have examined. Is this intuition correct in all cases?

With regard to point (4), we should note the probabilistic single-source reachability algorithm of [30] which achieves a low number of derivations although the arity is equal to two. It suggests picking $O(\sqrt{n} \log n)$ *distinguished nodes* from an n-node graph and searching forward from both the source node and all the distinguished nodes, but for only distance $\sqrt{n}$. Then, construct a graph on the distinguished nodes, with $u \rightarrow v$ if distinguished node u reaches distinguished node v in at most $\sqrt{n}$ steps. Take the transitive closure of this graph, using nonlinear TC. Then conclude that the source node reaches node w if the source reaches some distinguished node u in up to $\sqrt{n}$ steps, u reaches distinguished node v, and v reaches w in up to $\sqrt{n}$ steps.

When translated into Datalog, we have an algorithm in which any reachability fact has a derivation tree of depth $O(\sqrt{n})$. Although it has a binary predicate (for taking the TC of the graph of distinguished nodes), we can't derive more than $O(n \log^2 n)$ facts using this predicate, not the n^2 that we supposed would be possible for any algorithm that used a binary predicate. This algorithm works only with high-probability; it is not guaranteed to find all nodes reachable from the source. Thus, an interesting generalization of the theorems of Section 4 would take algorithms such as the one just sketched into account and find tradeoffs between

the number of derivations and the depth of the tree. E.g., this improvement on the data-volume cost despite the arity being two, gives motivation to the following question: Can reachability be computed with $O(n)$ data-volume cost? In a more algorithmic line, another question is to see whether we can get better balance in the probabilistic algorithm by choosing, e.g., $n^{2/3}$ distinguished nodes and search for distance $n^{1/3}$; that would give data-volume cost of $n^{4/3}$ instead of $n^{3/2}$.

7. REFERENCES

[1] Twister. http://www.iterativemapreduce.org/.

[2] F. N. Afrati. Bounded arity datalog ($\neq$) queries on graphs. In *PODS*, pages 97–106, 1994.

[3] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, 2011.

[4] F. N. Afrati, S. S. Cosmadakis, and M. Yannakakis. On datalog vs. polynomial time. *J. Comput. Syst. Sci.*, 51(2):177–196, 1995.

[5] F. N. Afrati and C. H. Papadimitriou. The parallel complexity of simple chain queries. In *PODS*, 1987.

[6] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.

[7] Apache. Hadoop. http://hadoop.apache.org/, 2006.

[8] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130, New York, NY, USA, 2010. ACM.

[9] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the IEEE International Conference on Data Engineering*, 2011.

[10] P. Bouros, S. Skiadopoulos, T. Dalamagas, D. Sacharidis, and T. K. Sellis. Evaluating reachability queries over path collections. In *SSDBM*, pages 398–416, 2009.

[11] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[12] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. Haloop: efficient iterative data processing on large clusters. In *VLDB Conference*, 2010.

[13] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *PODS*, pages 55–66, 1992.

[14] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[15] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[16] S. S. Cosmadakis. Inherent complexity of recursive queries. In *PODS*, pages 148–154, 1999.

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

[18] D. J. DeWitt, E. Paulson, E. Robinson, J. F. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.

[19] S. Ghemawat, H. Gobioff, , and S.-T. Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[20] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. In *Natl. Acad. Sci. USA 99*, page 78217826, 2002.

[21] Y. E. Ioannidis. On the computation of the transitive closure of relational operators. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 403–411, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, 2007.

[23] H. V. Jagadish, R. Agrawal, and L. Ness. A study of transitive closure as a recursion mechanism. In *SIGMOD Conference*, pages 331–344, 1987.

[24] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7, 2011.

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

[26] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD Conference*, 2010.

[27] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. 2010.

[28] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[29] J. D. Ullman and A. V. Gelder. Parallel complexity of logical query programs. In *FOCS*, 1986.

[30] J. D. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. In *SPAA*, pages 200–209, 1990.

[31] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Expert Database Conf.*, pages 271–293, 1986.

[32] H. Wang, H. He, J. Y. 0001, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.

[33] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.

[34] Y. Yu, M. Isard, D. Fetterly, M. Budiu, lfar Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In R. Draves and R. van Renesse, editors, *OSDI*, pages 1–14. USENIX Association, 2008.