Map-Reduce Extensions and Recursive Queries

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

Vinayak Borkar
UC Irvine, USA
vborkar@ics.uci.edu

Michael Carey
UC Irvine, USA
mjcarey@ics.uci.edu

Neoklis Polyzotis
UC Santa Cruz, USA
alkis@cs.ucsc.edu

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

ABSTRACT

We survey the recent wave of extensions to the popular map-reduce systems, including those that have begun to address the implementation of recursive queries using the same computing environment as map-reduce. A central problem is that recursive tasks cannot deliver their output only at the end, which makes recovery from failures much more complicated than in map-reduce and its nonrecursive extensions. We propose several algorithmic ideas for efficient implementation of recursions in the map-reduce environment and discuss several alternatives for supporting recovery from failures without restarting the entire job.

Categories and Subject Descriptors

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

General Terms

Reliability, Theory, Algorithms, Performance

Keywords

Map-reduce, transitive closure, Datalog, recursion, polynomial fringe, task migration

1. MAP-REDUCE AND ITS EXTENSIONS

The introduction of map-reduce by Dean and Ghemawat [14] for parallel computation on commodity clusters focused a great deal of commercial and intellectual interest on this model and similar approaches to managing large-scale data. It is useful to reflect on what map-reduce brings to the table. Map-reduce itself is a convenient way for modestly skilled programmers to implement many data operations, including the operations of relational algebra and operations on sparse matrices and vectors (which are not too much different from joins of relations). However, the resulting parallel algorithms for relational operations are essentially the same as those known earlier. The secret sauce of map-reduce is not the algorithms it supports, but rather the way it handles failures during the execution of a large job.

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

1.1 Tasks and Compute-Node Failures

It is desirable to use cheap hardware to execute large parallel jobs, rather than using expensive parallel machines and/or specialized storage systems. However, in such a computing environment, it is normal to expect failures of the compute nodes — disk crashes or other component failures. In many cases there is a significant risk of software failure, e.g., a task that is written to use the latest version of Java finds itself running on a compute node whose Java has not yet been updated.

Map-reduce deals with these failures by restricting the units of computation in an important way. Both Map tasks and Reduce tasks have the *blocking* property:

- A task does not deliver output to any other task until it has completely finished its work.

The benefit of the blocking property is that if a task T_1 fails at any time during its execution, it can be restarted at another compute node. There is no risk that some output of the original T_1 has been passed to another task T_2, which erroneously will receive the same input from the restarted T_1 later. Without the blocking property, it would be necessary in the worst case to restart the entire job whenever any one task failed.

The blocking property is not unique to map-reduce; it has been exploited in a number of recent extensions. The point at which it breaks down is when the tasks are recursive. Recursive tasks need to deliver some output before finishing, because the output of a recursive task must have the opportunity to feed back to its input.

1.2 Generalizations of Map-Reduce

The cluster-computing environment normally is built on a distributed file system such as GFS [17] or HDFS [5]. These file systems divide enormous files into chunks of 64MB (typically), and replicate each chunk three or more times on different racks. They serve as a reliable storage system for a map-reduce system such as Google's original [14] or its open-source equivalent Hadoop [4].

The most natural extension of map-reduce is to allow tasks to be more general than the particular two kinds of tasks (Map tasks and Reduce tasks) found in map-reduce

itself. We are thus led to *dataflow* systems that support any collection of tasks connected in an acyclic manner. Typically, a dataflow system uses prototypes for each kind of task that the job needs, just as Map-Reduce or Hadoop use prototypes for the Map and Reduce tasks. A dataflow system replicates the prototype into as many tasks as are needed or requested by the user. Dataflow systems include Dryad [20] and its extension DryadLINQ [31] (Microsoft), Clustera [15] (U. Wisconsin), Hyracks [8] (UC Irvine), and Nephele/PACT [7] (T. U. Berlin).

Other extensions of map-reduce are high-level languages that compile into a sequence of map-reduce jobs. PIG [24] is an implementation of relational algebra from Yahoo!. Hive [6] (open-source) and SCOPE [12] (Microsoft) are implementations of limited forms of SQL.

1.3 Recursive Queries

While conventional wisdom may have relegated recursion to an uninteresting byway, consisting of calculations of ancestors or management chains, there are several classes of problems that have become interesting recently and are true recursions on very large relations.

1. The original problem for which Map-Reduce was developed — PageRank — is really a recursive calculation of the fixpoint of a sparse matrix-vector multiplication. We tend to think of it as an iteration because normally it is computed in discrete rounds.

2. The full transitive closure is necessary for studies of the structure of the Web, as in [10].

3. Several calculations about social networks involve the transitive closure of the Friends graph. Two examples are discovering communities or centrality of persons.

Moreover, these recursive calculations are really operations on relational data. A sparse matrix is naturally represented as a relation. The same is true of a sparse graph, such as the graphs of the Web or a social network.

1.4 Overview of the Paper

In this paper, we shall explore the problems of implementing recursion, especially recursive queries, on a computing cluster. We begin with the data-volume cost model, in which we can evaluate different algorithms for executing queries on a cluster, whether recursive or not.

We look at algorithms for implementing recursive queries, starting with transitive closure as a fundamental example, and then looking at its generalization to all recursive queries. Next, we explore the space of approaches to implementing recursion on a cluster. The central issues are:

1. Execution of a recursion as an iteration may not be the most efficient way to proceed. We look at some other options and the problems they present.

2. In Section 1.1 we noted that recursive tasks cannot have the blocking property. We look at options for coping with compute-node failures when tasks are recursive.

3. We consider the problem of the *endgame*: many recursive queries generate most of the answer tuples in the early rounds, but these productive rounds are followed by many rounds that each yield only a small number of new tuples. Because file transfer between compute nodes involves substantial overhead, there is a need to cope with the plethora of small files. We address:

   (a) Modifying the underlying algorithm to reduce the number of rounds; see Section 4.

   (b) Modifying the behavior of the tasks executing the recursion as the number of rounds increases. These approaches include merging tasks, migrating tasks, and introducing new tasks that allow the combination of many small files into a few large files; see Section 7.

2. DATA-VOLUME COST

When evaluating algorithms on a computing cluster, we need to measure the total running time of all the tasks that collaborate in the execution of the algorithm. That sum is the cost of renting time on processors from a public cloud, and is therefore a fair way to evaluate algorithms in the cluster-computing environment. As a simplified surrogate for the running time, we shall focus on the sum of the sizes of the inputs to all the tasks that collaborate on a job, which we call the *data-volume* cost of an algorithm. A discussion of this model is in [3], where it is called the communication-cost model.

2.1 Details of the Model

To apply data-volume cost meaningfully, we need to assume there is an upper limit on the amount of data that can be input to any one task. Without this assumption, the optimal algorithm would be a single task with no communication among tasks. But more importantly, there is a great value in restricting the input size for a task. For example, this restriction may enable all computation to be done in main memory.

The model also assumes a significant, fixed overhead cost for sending any file to a task. Equivalently, there is a lower bound b which we add to the data-volume even if an input file is smaller than b. We discuss how to avoid small files as input in Section 8.

2.2 Other Costs

The execution time of a task could, in principle, be much larger than its input data volume. However, for the kinds of algorithms we shall focus on, including all kinds of SQL operations, there are implementations, e.g., hash join, that perform operations in time proportional to the input size plus output size.

The data-volume model counts only input size, not output size. In justification, the output of each task is either:

1. The input to one or more other tasks, and thus will be counted as input to that operation, or

2. Output of the job as a whole. In this case, it is likely that the output will be aggregated or otherwise collapsed to produce a query result that can be read and used by a human analyst. The cost of generating this relatively small output can be neglected.

3. DATALOG AND RECURSIVE QUERIES

There has been a surprising resurgence of interest in Datalog for large-scale data-processing applications, including networking [18], analysis of very large programs [22], and distributed (social) networking [25]. Datalog notation also allows us to express a rule that gives a lower bound on execution time for queries, whether recursive or not; this point is discussed in Section 3.2. We shall use the notation of [29] to express Datalog queries in what follows.

3.1 Transitive Closure

Each of the recursions mentioned in Section 1 is similar to the standard calculation of transitive closure (hereafter TC) on a directed graph. There are several ways to express TC as a recursive Datalog program. Suppose that $E(x, y)$ is a relation representing the directed edges of a graph; i.e., there is an edge from a to b if and only if (a, b) is a tuple of E. The *nonlinear* form of TC rules is:

```
P(x,y) <- E(x,y)
P(x,y) <- P(x,z) AND P(z,y)
```

One can also replace the subgoal $P(x, z)$ in the recursive rule by $E(x, z)$ to get the *left-linear* version of TC, or replace the subgoal $P(z, y)$ by $E(z, y)$ to get the *right-linear* version.

Although the nonlinear form is not even supported by the SQL standard, and was not considered in the classic study of TC in [13], when we think about execution on a computing cluster there is good reason to consider the nonlinear version. The advantage of using a nonlinear TC is that the number of needed *rounds* (iterations of the recursive rule, whether synchronous or not) is much smaller than for the linear versions — $O(\log n)$ rather than $O(n)$ on an n-node graph. Thus, this method helps deal with the endgame problem where many small files need to be communicated at many of the rounds of a recursion.

3.2 A Lower Bound on Query Execution Cost

When we take a join, we expect that the execution cost is no less than the number of pairs of tuples that join successfully. This statement is correct only if there is no other way to execute the same query that avoids taking the join. The following rule, from [1], expresses this intuition precisely, in Datalog. Given a Datalog program, the number of its *derivations* is the sum, over all the rules in the program, of the number of ways we can assign values to the variables in order to make the entire body (right side of the rule) true. The key property of the count of derivations is that:

- An implementation of a Datalog program that executes the rules as written must take time on a single processor at least as great as the number of derivations.

Of course, one can sometimes rewrite the rules to do better, but unless we do so, every derivation must be considered in a correct execution of the rules. It is known that seminaive evaluation of Datalog rules (see [27] for the definition of *seminaive*) will take time on a single processor that is proportional to the number of derivations, since seminaive evaluation avoids considering any combination of tuples that have been combined before.

[16] explored parallel implementation of Datalog queries in a setting that is not too different from a modern cluster environment. They defined a parallel implementation to be *seminaive nonredundant* if (in our terms) the data-volume cost is proportional to the number of derivations. We shall see several algorithms for TC that have this property. First, let us consider some examples of counting derivations.

EXAMPLE 3.1. *Consider the left-linear TC mentioned in Section 3.1:*

```
P(x,y) <- E(x,y)
P(x,y) <- E(x,z) AND P(z,y)
```

The body of the first rule is just $E(x, y)$, so the body can be made true by any substitution for x and y that forms an edge of the graph. That is, this rule admits $|E|$ derivations.

The second rule body, $E(x, z)$ AND $P(z, y)$, is made true whenever we can find nodes a, b, and c such that there is an edge from a to c and a path from c to b. Thus, we can describe the number of derivations for this rule as the sum over all nodes c of the in-degree of c times the number of nodes reachable from c. Typically, the number of derivations obtained from the second (recursive) rule greatly exceeds the number from the first (basis) rule. In what follows, we shall ignore the basis rule and consider only the recursive rule.

For the right-linear version of TC, where the body of the recursive rule is $P(x, z)$ AND $E(z, y)$, the recursive rule has a number of derivations equal to the sum over all nodes c of the number of nodes that can reach c times the out-degree of c. That figure can be more or less than the figure for left-linear TC, but it is rarely the same.

The nonlinear TC program has a number of derivations equal to the sum over all nodes c of the number of nodes that can reach c times the number of nodes c can reach. This quantity is different from the numbers of derivations for either the left- or right-linear TC programs, but is never less than those and can be much greater. Section 4.2 discusses ways to lower the derivation count for nonlinear TC significantly. When we consider the inherent advantages, in the cluster environment, of reducing the number of rounds, the nonlinear version of TC can be superior in practice.

4. TRANSITIVE CLOSURE ALGORITHMS

We shall now consider two specific algorithms for taking the transitive closure. The first is a direct implementation of the nonlinear TC program, and the second is the improvement, promised above, that greatly cuts down on the data-volume cost.

4.1 An Implementation of Nonlinear TC

We use two kinds of tasks: *Join tasks* and *Dup-elim tasks*. Each task receives inputs from tasks of the one kind and delivers input to the tasks of the other kind.[1] We require two hash functions, h and g. The first, h, takes a node of the graph as an argument and produces a bucket number. Each Join task corresponds to one bucket. The hash function g takes a pair of nodes and produces a bucket number, which corresponds to one Dup-elim task.

Initialization: Each tuple (a, b) in the relation E is sent to the Dup-elim task $g(a, b)$, where it is treated as a tuple of the relation P.

[1] We could combine the two kinds of tasks into one kind, but using both kinds allows us to recover from single task failures in a way we could not do if we used only one kind of task. In particular, we can restart tasks of either kind using the outputs of tasks of the other kind.

Join tasks: Join task i receives all and only the tuples (a, b) such that either $h(a)$ or $h(b)$, or both, is i. This task:

1. Adds (a, b) to its local store. It will never be the case that this tuple is already stored.
2. Searches its store for tuples (b, c), provided $h(b) = i$. For each such tuple found, it sends (a, c) to the Dup-elim task $g(a, c)$.
3. Searches its store for tuples (c, a), provided $h(a) = i$. For each such tuple found, it sends (c, b) to the Dup-elim task $g(c, b)$.

Dup-elim tasks: These store all received tuples locally. When (a, b) is received, it checks whether this tuple was recieved previously. If so, the tuple is ignored. If not, the tuple is stored and sent to Join tasks $h(a)$ and $h(b)$.

The data-volume cost for this algorithm is of the same order as the number of derivations of the nonlinear Datalog program. To see why, suppose the graph has a path from node a to node c and a path from node c to node b. Then there is a derivation in which a, b, and c substitute for x, y, and z, respectively, in the recursive rule. The Join task numbered $h(c)$ will eventually receive the tuples $P(a, c)$ and $P(c, b)$. When the second of these arrives, it will discover $P(a, b)$ and communicate this tuple to the Dup-elim task $g(a, b)$. No other Join task receives both $P(a, c)$ and $P(c, b)$, so this derivation is never again used (although there may be other derivations of the same fact $P(a, b)$).

When the Dup-elim task $g(a, b)$ receives $P(a, b)$ for the first time, it transmits it to two Join tasks $h(a)$ and $h(b)$. But if received a second time, $P(a, b)$ is ignored. Thus, each derivation causes the communication of between one and three tuples, but never causes more than three. Thus, the data-volume cost is proportional to the number of derivations. Additionally, if sensible data structures are used for the various tasks, the execution time at each task is proportional to the number of tuples received or sent. Thus, total cost is linear in the number of derivations.

4.2 TC by Recursive Doubling

Recall that the motivation for considering the nonlinear version is that the lower the number of rounds, the less likely it is that the overhead of delivering a large number of tiny files will dominate the total cost. However, the linear versions of TC have an advantage over the nonlinear TC in that the former discover each shortest path only once. That is not the same as discovering each P-fact only once, but it seems the most restrictive exploration of paths that we can do in general.

It is, in fact, possible to implement nonlinear TC in a way that discovers each shortest path only once. The algorithm *Smart Transitive Closure* appears in [30] and [19]. In [21] it was shown to be highly efficient as a serial algorithm. Intuitively, Smart-TC breaks each path into a prefix, whose length is a power of 2 and a suffix whose length is no greater than the length of the prefix. The details are shown in Fig. 1. There are other options, which we shall not discuss in detail, for forcing paths to be discovered only once. For example, we could insist that the length of the prefix be exactly equal to the length of the suffix (plus 1 for odd lengths).

To simplify the description of the Smart-TC algorithm, we shall assume that the graph is first made acyclic by collapsing strong components into single nodes, as was suggested in the analysis of TC algorithms [13]. The algorithm shown in Fig. 1 is an iteration, where in round $i \geq 0$ we compute:

1. $P_i(x, y)$ = the pairs of nodes (x, y) whose shortest path from x to y is of length between 0 and $2^i - 1$.
2. $Q_i(x, y)$ = the pairs of nodes (x, y) such that the shortest path from x to y is of length exactly 2^i.

```
1)   Q0 := E;
2)   P0(X,X) := {(x,x)|x is a graph node};
3)   i := 0;
4)   repeat {
5)        i := i + 1;
6)        Pi(x,y) := π_{x,y}(Q_{i-1}(x,z) ⋈ P_{i-1}(z,y));
7)        Pi := Pi ∪ P_{i-1};
8)        Qi(x,y) := π_{x,y}(Q_{i-1}(x,z) ⋈ Q_{i-1}(z,y));
9)        Qi := Qi − Pi;
     }
10)  until (Qi == ∅)
```

Figure 1: Transitive closure by recursive doubling

The basis is lines (1) through (3). Line (1) initializes Q_0 to be the edges of the graph, i.e., those pairs of nodes whose shortest path is of length $2^0 = 1$.[2] Line (2) initializes P_0 to be the paths of length 0, that is, all tuples $P_0(x, x)$. Finally, line (3) sets i, the iteration counter, to 0.

Lines (4) through (10) form a loop that iterates until at some stage no more Q-tuples are discovered. After incrementing i at line (5), we compute P_i at lines (6) and (7). Line (6) joins Q_{i-1} and P_{i-1}, thereby discovering all paths of length between 2^{i-1} and $2^i - 1$. Line (7) then adds in the P-facts discovered on previous rounds, i.e., those paths of length less than 2^{i-1}. Line (8) computes Q_i to be the join of Q_{i-1} with itself. That will surely discover all paths of length 2^i. However, it also will include some pairs that have a path of length 2^i but also have a shorter path. These pairs are eliminated by line (9).

To implement this algorithm, we can use one collection of the Join and Dup-elim tasks mentioned in Section 4.1 for P and a distinct collection of the same kinds of tasks for Q. The data-volume cost of this implementation is no greater than the sum, over all nodes c and integers i, of the number of nodes that can reach c by a shortest path whose length is 2^i, times the number of nodes c can reach by a path that is no longer. This quantity is surely less than the cost of the straightforward nonlinear TC algorithm. More importantly, it shares with the implementations of the left- or right-linear versions of TC the good property that each path can be constructed in only one way.

5. THE GENERAL ITERATIVE SOLUTION

The organization of recursive tasks from Section 4.1 carries over to every Datalog program, recursive or not. Here is an outline of how to do so.

[2]Note we assume the graph is acyclic, and thus has no loops. If not, we would have to remove $Q_0(x, x)$ whenever there is a loop from x to itself.

5.1 Implementing Any Datalog Program

For each rule we create a collection of tasks that are responsible for applying that rule and producing new facts for the head of the rule. Some hashing scheme is used to divide the work among tasks. This scheme must identify buckets by vectors of values, and each component of the vector is obtained by hashing a certain variable that appears in the body of the rule. Suppose $P(\cdots)$ is a subgoal of the rule, and this subgoal has k arguments. A task receives all facts $P(a_1, a_2, \ldots, a_k)$ discovered by any task, provided that each component a_i meets a constraint: if the ith position of the subgoal corresponds to a variable that participates in the hashing scheme for the rule, then a_i has a hash value that agrees with the ith component of the vector associated with that task. If the ith position's variable does not participate in the hashing scheme, then a_i can be any value.

EXAMPLE 5.1. *Consider a rule*

```
P(x,y) <- Q(x,z) AND R(z,w) AND S(w,y)
```

Suppose we use 100 tasks to evaluate this rule. We might choose to hash z five ways and w 20 ways, while x and y do not participate in the hashing scheme. If a new Q-fact, say $Q(a, b)$ is discovered by some other task, then this fact is sent to all those tasks for the rule above whose bucket is identified by a vector $[h(b), v]$. Here, $h(b)$ is the hash function for variable z applied to b (resulting in one of five possible values), and v is any of the 20 values into which w can be hashed. I.e., Q-facts are sent to 20 tasks; R-facts are sent to one task, and S-facts are sent to five tasks.

We can avoid having to send newly discovered facts to more than one task of a collection if we rewrite the rules to have bodies of at most two subgoals (i.e., we force all joins to be binary). Additionally, the results from several rules for the same predicate must be combined, using tasks that perform a union. The implementation must make sure that facts discovered more than once are not propagated the second or subsequent times, using tasks like the Dup-elim tasks from Section 4.1. If we handle these aspects correctly, then we can claim that the data-volume cost of the algorithm is proportional to the number of derivations for the rules.

5.2 The Polynomial-Fringe Property

It is highly unlikely that all Datalog programs can be implemented in parallel in *polylog time* (time a power of the logarithm of the input size). The division between those that can and those that (almost surely) cannot was addressed in [28], which defined the *polynomial-fringe property* (hereafter PFP). 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. [28] showed that all Datalog programs with the PFP can be parallelized in polylog time. [2] examined the common case of single chain-rule Datalog programs and divided them completely between those that have polylog-time parallel algorithms and those that are P-complete and therefore almost surely cannot be parallelized.

Using the algorithm of [28], we can implement any Datalog program with the PFP in only $\log^2 n$ parallel computation steps (n is the data size). Most of the work involves $\log n$ executions of TC, which can be done in the efficient way suggested by Section 4.2. This figure does not exactly match the $\log n$ rounds needed for transitive closure, but it does suggest that we can do much better than $O(n)$ rounds that might be needed in general.

6. IMPLEMENTATIONS OF RECURSION

Several software systems have addressed the problem of implementing recursions in a cluster environment while coping efficiently with failures. We discuss each briefly, with an eye toward how they deal with failures during a recursion.

6.1 The HaLoop Approach

HaLoop [11] replaces recursion by an iteration of map-reduce passes. This system tries to minimize communication by considering how data is passed from tasks at one pass to the next and locating intermediate files and tasks appropriately. Since tasks exist for only a single pass, the problem of recursive tasks disappears. That is, the ordinary failure-recovery methods of a map-reduce implementation can be relied on to handle failures. The problem with this approach is that tasks must operate in synchronous rounds, and the output of one task must still be passed, even if locally, to its successor task in the next map-reduce phase.

6.2 The Pregel Approach

Pregel [23] assumes a recursive algorithm executes on a graph. Tasks correspond to nodes of the graph and can send data to any other task. These messages are organized into *supersteps*, which are essentially rounds. Pregel checkpoints the state of all tasks periodically, after some chosen number of supersteps. When a task fails, all tasks are rolled back to the previous checkpoint, and the failed task(s) are restarted at different compute nodes. That approach not only throws away perfectly good computation, but it does not scale completely. That is, the more compute nodes are involved, the more frequently we must checkpoint if we want a fixed probability of failure between consecutive checkpoints.

7. DEALING WITH TASK FAILURES

Now we turn our attention to the fact that recursive tasks do not have the blocking property, yet we need to be able to cope with failures of compute nodes or tasks without restarting the entire job. First, we look at options for managing the intermediate files that communicate data among recursive tasks. Then, we examine options for restarting tasks that have failed, especially how we deal with data that was already transmitted by the failed task.

7.1 File Management for Recursions

From the discussion of Section 4.1, you might imagine that tuples are passed among tasks as soon as they are generated. However, passing single tuples incurs severe overhead. Roughly, it is only economical to send tuples in packages of thousands. We assume that each task has an input queue of tuples, which from time to time is passed new input tuples by other tasks, or in some cases, by itself. Each task maintains one output file for each of the other tasks, into which it places discovered tuples destined for that task.[3]

[3]More realistically, there is only a need for one file per pair of compute nodes. That file can send data from all the tasks located at the source node to all the tasks located at the destination node. However, since our general desire for low wall-clock time implies that we should use many compute nodes, we shall tend to think of files as communicating from one task to another.

Because we must deal with failures, it is essential that each of these intermediate files be recoverable. They could be stored in the surrounding distributed file system, which will replicate them sufficiently that they are very unlikely to be lost. However, that solution is probably more than needed. An option, which we shall assume at the minimum, is that each task stores locally all the files it ever generates. Thus, if one task fails, its inputs can be reconstructed from the other tasks, as long as the failed task does not feed input to itself. However, we can design tasks to avoid self-feeding. An example should suggest the general technique.

EXAMPLE 7.1. *Let us recall the TC implementation of Section 4.1, where we used two kinds of tasks: Join and Dup-elim. Notice that each task feeds data only to tasks of the other kind. If we locate the Join tasks on different racks from those used by the Dup-elim tasks, then we can even survive a rack failure. If any number of Join tasks fail simultaneously, their inputs can be constructed from the Dup-elim tasks, and vice-versa.*

There are several other issues regarding file management. Section 8 will deal with the problem of small files. Here, we examine some options regarding how to decide when to pass files from one task to another.

7.1.1 Operation in Rounds

We can wait until each task has exhausted its input. At that time, all tasks transmit all their files to the proper destination task. This approach treats the recursion as an iteration. It is commonly used in map-reduce implementations. For example, we can see the common PageRank calculation, which is technically a recursion, as an iteration in which each step is carried out by a separate map-reduce job, with distribution of data interspersed. It is also the superstep approach used by Pregel. The disadvantage is that some tasks may finish early and must idle while other tasks finish.

7.1.2 Tasks Choose to Send Data

An alternative is to allow each task to decide when it is ready to send a file. It might send a file as soon as that file has reached a certain size, or send all its files after the total amount of output it has generated has reached a limit. There are two advantages of this approach. First, it is more likely that each task will have some input available at all times. Second, the communication network will be busy more of the time, rather than being idle between synchronous rounds.

7.1.3 A Global Decision

The controller for the tasks (*master* in the parlance of Hadoop) can decide when to pass data. It could call for data to be sent at regular time intervals, or it could monitor the total amount of data generated and call for transmission when the total amount of data reaches a set limit.

7.2 Recovery from Task Failures

A task can fail for many reasons, including software failures (e.g., the node on which it runs has the wrong version of Java), failure of the compute node itself, and failure of the communication facility for an entire rack of nodes. Here we discuss the approaches that can be used in an implementation of recursion on a computing cluster. No strategy, can eliminate the possibility of restart altogether. For example, a Hadoop job needs to be restarted when the node executing the master controller fails. However, a reasonable criterion for failure-management strategies is that the probability of having to restart the largest imaginable job is close to 0.

Pure Datalog, unlike most forms of recursion, has an idempotence property due to its reliance on the set model of data. That is, generating a tuple a second time has no effect on the eventual outcome of the recursion. Thus, if a task fails, we can restart it at a different node without affecting the global computation. Since the repeated task must (typically) run from the beginning, we need a mechanism to supply it with the same inputs that were received by the failed task. As discussed in Section 7.1, there are several options for making it almost certain that these files are still available to the restarted task.

Unfortunately, many recursions involve nonidempotent operations. For example, we may want the normal bag-model semantics of SQL. Or we may have an aggregation involved in a Datalog recursion, such as counting the number of paths in a graph or the number of nodes reachable from each node in a graph. If so, we not only need to reconstruct the inputs to a restarted task. We need to know which output files from the corresponding failed task were delivered to their destination, so we do not send the same data again. There are again several approaches.

7.2.1 Responsibility With the Master Controller

The master records the location of each file ever shipped from one task to another. A restarted task is told by the master which files the original task received as input.

There must be a way to reproduce the timing of the generation of output files; for example, if output files are generated in rounds, the restarted task can produce the same output files as the original version of the task, since it is given the same input files in the same sequence. The restarted task must execute the same steps as the original, in order to develop the same internal state as the original. However, when it generates an output file that was previously sent by the original version of the restarted task, the master does not deliver it; this file is "thrown away."

7.2.2 Responsibility With the Recipient

Instead of giving the master the responsibility for throwing away repeated output files, the recipients can be given that responsibility. Each task must record all the files it ever receives. It may not be necessary to record the file contents, but certainly it must record enough information to know if it receives an identical file, e.g., an identifier of the task and the round at which that file was sent.

8. THE ENDGAME

Recall that in later rounds of a recursion, the number of new facts derived at a round may drop considerably. There is significant overhead in transmitting files, so it is desired that each of the recursive tasks have thousands of tuples for each of the other tasks whenever we distribute data among the tasks. We are thus motivated to consider how to reduce the number of files needed, and that question in turn forces us to consider methods for migrating recursive tasks from one compute node to another.

8.1 Dealing With Small Files

The large number of tasks needed for early rounds of the recursion may lead to very small files at later rounds. Thus, in at least some applications there is need for an "endgame" strategy, where as the sizes of files shrink, the algorithm used to implement the recursion changes. We consider two possibilities here.

8.1.1 Using Fewer Compute Nodes

We may reduce the number of compute nodes used for the tasks and therefore cause many tasks to execute at the same node. Collect all the tuples at a compute node that are destined for tasks at a single other compute node into a single file (typical implementations do this step routinely). Ship that file to the destination and have that node sort the input into separate files for each of the tasks at that node. This option is an important motivation for considering the matter of task consolidation in Section 8.2.

8.1.2 Using Hub Tasks

We may create one or more *Hub tasks* whose only job is to receive files from the operational tasks and consolidate them into files destined for each task. As in the first strategy, each task can bundle the tuples destined for each other task (labeling the tuples by destination, of course) into a single file and ship them as one file to a Hub task.

8.2 Task Migration

Historically, task migration in multiprocessor systems has been discussed from the point of view of how compute nodes can inform other nodes of their load efficiently; see, e.g., [26]. However, in the environment we have been studying, there is typically a master controller that pings all tasks periodically anyway and can gather load information as it does. Nevertheless, there are some new issues that arise, especially for recursive tasks and tasks that deal with relational data. In the following sections we shall address two broad issues:

1. What are the mechanics of migrating tasks?
2. When should we choose to migrate tasks?

8.3 Merging and Splitting Relational Tasks

When we divide a relational operation among tasks, the normal way to do so is to use a hash function on some of the data. The buckets of the hash function correspond to the tasks, as in Example 5.1. If we are going to vary the number of tasks cooperating to perform an operation, we use a hash function h that generates a large number of bits, say 32. For any fixed number of tasks, say 64 tasks, we use only a prefix of the bit string produced by h; in this case, the first six bits would be used.

We need to manage indexes on the data as we merge and split tasks. When tuples of a relation are sent to only one task in the group (unlike Example 5.1), we can support merging and splitting of tasks easily. For an example, consider the natural join $R(A, B) \bowtie S(B, C)$. Tuples of both R and S are hashed on their B-values. We also need an index on B for both relations, so we can retrieve tuples of one relation that join with a given tuple from the other relation. Thus, suppose we have a hash function $h(b)$ that maps B-values to 32-bit strings, and assume the first six bits of that string determine which of 64 Join tasks gets a given tuple.

At a given Join task T, store the tuples of R and S in B-trees with key equal to the entire 32-bit string $h(b)$. That is, tuples of R, say (a, b), appear in the B-tree according to the order given by $h(b)$. Of course, all the R-tuples at T share the same first six bits, but the other 26 bits will order all the R-tuples at T. If we need to find all the tuples of R that have a particular B-value b, just search in the B-tree with key $h(b)$, and we shall find them quite efficiently. Tuples of S are stored analogously in another B-tree.

Now, suppose we decide to double the number of tasks, so there are 128 tasks. We identify each task by the first seven bits of $h(b)$, which means that the tuples for which T is responsible are divided into two groups — those with 0 in the seventh bit and those with 1. Say those with 0 remain at T, while those with 1 go to a new task U.

The benefit of the B-tree organization is now apparent. All the tuples that stay at T precede all those that are moved to U in the order. We must split the B-tree down the middle, which may involve restructuring at all levels, but the total work is proportional only to the number of levels, not to the number of tuples stored at task T.

Conversely, suppose that instead, we decide to reduce the number of tasks, merging our 64 tasks into 32 tasks. We start using only the first five bits of $h(b)$, which has the effect of merging all pairs of tasks identified with 6-bit strings that differ only in the sixth bit. Suppose that the bit-string for T has a 0 in its sixth bit, and V is the task whose bit-string agrees except for a 1 in the sixth bit. Then we must merge T and V. However, the keys for all the B-values at T will precede the keys for all the B-values at V. Thus, building an index for the merged task requires only that we stitch together two B-trees; no merging of values from the two trees is required. Again, the stitching may need some restructuring at each level, but the work is proportional to the levels, not the data size.

Matters are somewhat more complex when tuples have several components that together determine the task to which the tuple is assigned. Example 5.1 is typical; there, the hash function combines values corresponding to rule variables z and w. If we want, say, a hash function of 32 bits, then we need to interleave bits from a hash function that applies to z and another hash function that applies to w. Note that some tuples to not involve z or do not involve w, so in these tuples the bits from the missing attribute can be arbitrary and are determined only by the bit string associated with the task in question. We shall suggest the necessary extension in a concrete example, from which the general idea should be apparent.

EXAMPLE 8.1. *Suppose that we have 64 tasks implementing the Datalog rule from Example 5.1, and we wish to split tasks by taking the seventh bit of the hash function $h(w, z)$ into account. Consider the tuples for subgoal $Q(x, z)$ stored at task T, which we shall split into T and U. These tuples are arranged in a B-tree whose key is all the bits of h that come from z. If the seventh bit of h is a bit that comes from w, then the B-tree from T for tuples of Q is copied to U as well as remaining at T. However, if the seventh bit comes from z, then this B-tree is split as we described above.*

Conversely, if we merge T with another task V that differs only in the sixth bit of h, then we must again consider two cases, depending on whether the sixth bit comes from w or z. If from w, then the B-trees for Q at T and V are identical, since they contain exactly the same Q-tuples. We therefore

throw away one and keep the other. If the sixth bit is from z, then we concatenate the two B-trees as above.

8.4 When and How Should We Migrate?

We obviously need a strategy to determine when to migrate tasks. In this direction, we may leverage existing work in online computation and specifically the field of task systems [9]. The gist is to monitor the current performance of the computation and to run what-if scenarios on different possible configurations (migrating tasks to different nodes, scaling-up/down the degree of parallelism). Given this information, there are algorithms that can decide in a robust fashion (i.e., with a bounded competitive ratio) when to switch configuration based on the performance gains and also the cost of doing the switch.

9. FURTHER DEVELOPMENTS

We have argued that extending the map-reduce model of cluster computing to recursive jobs requires considerable work. Thought must be given to the underlying algorithms, including the possibility that nonlinear recursions offer substantial advantages in this environment. Conversion of linear to nonlinear recursions is easy in some cases, such as transitive closure, but may be impossible in others. Open questions include exploring where nonlinear recursion is possible and minimization of the data-volume cost for cases other than TC where the conversion is possible.

We have also explored a number of important implementation issues. One of these is the matter of how best to schedule the transmission of data among the various recursive tasks. Methods of recovery from task or compute-node failures needs to be explored and evaluated, especially in the case where the operations being performed are not idempotent. The problem of the "endgame," where recursions at the end require many rounds that produce little data, was proposed. There is a need to explore solutions involving file merging and task merging in various combinations.

10. REFERENCES

[1] F. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. Ullman. Cluster computing, recursion and datalog. To appear in a book based on the Datalog 2.0 Workshop (March, 2010), Oxford GB, to be published by Springer in 2011.

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

[3] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.

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

[5] Apache. Hdfs. http://hadoop.apache.org/hdfs/, 2008.

[6] Apache. Hive. http://wiki.apache.org/hadoop/Hive, 2008.

[7] 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.

[8] 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, to appear*, 2011.

[9] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[10] 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.

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

[12] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. In *VLDB*, 2008.

[13] S. Dar and R. Ramakrishnan. A performance study of transitive closure algorithms. In *SIGMOD Conference*, pages 454–465, 1994.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[15] 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.

[16] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. *SIGMOD Rec.*, 19:143–152, May 1990.

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

[18] J. M. Hellerstein. Datalog redux: experience and conjecture. In *PODS*, pages 1–2, 2010.

[19] 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.

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

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

[22] M. Lam and et al. Bdd-based deductive database. bddbddb.sourceforge.net, 2008.

[23] 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.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, 2008.

[25] S.-W. Seong, M. Nasielski, J. Seo, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, and M. S. Lam. The architecture and implementation of a decentralized social networking platform. http://prpl.stanford.edu/papers/prpl09.pdf, 2009.

[26] T. Suen and J. Wong. Efficient task migration algorithm for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 3(4):488 –499, jul. 1992.

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

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

[29] J. D. Ullman and J. Widom. *A first course in database systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

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

[31] 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.