Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis

Jiwon Seo
Stanford University
jiwon@stanford.edu

Jongsoo Park
Intel Corporation
jongsoo.park@intel.com

Jaeho Shin
Stanford University
jaeho.shin@stanford.edu

Monica S. Lam
Stanford University
lam@stanford.edu

ABSTRACT

Large-scale graph analysis is becoming important with the rise of world-wide social network services. Recently in SociaLite, we proposed extensions to Datalog to efficiently and succinctly implement graph analysis programs on sequential machines. This paper describes novel extensions and optimizations of SociaLite for parallel and distributed executions to support large-scale graph analysis.

With distributed SociaLite, programmers simply annotate how data are to be distributed, then the necessary communication is automatically inferred to generate parallel code for cluster of multi-core machines. It optimizes the evaluation of recursive monotone aggregate functions using a delta stepping technique. In addition, approximate computation is supported in SociaLite, allowing programmers to trade off accuracy for less time and space.

We evaluated SociaLite with six core graph algorithms used in many social network analyses. Our experiment with 64 Amazon EC2 8-core instances shows that SociaLite programs performed within a factor of two with respect to ideal weak scaling. Compared to optimized Giraph, an open-source alternative of Pregel, SociaLite programs are 4 to 12 times faster across benchmark algorithms, and 22 times more succinct on average.

As a declarative query language, SociaLite, with the help of a compiler that generates efficient parallel and approximate code, can be used easily to create many social apps that operate on large-scale distributed graphs.

1. INTRODUCTION

With the rise of world-wide social networks, many large-scale graph-oriented databases are now available. These graphs are large, making it necessary for them to be processed on large-scale distributed systems, thus raising the question of how these algorithms are to be programmed. MapReduce [13] demonstrated how a high-level language that masks the complexity of distributed processing can lead to the creation of a tremendous number of distributed applications. However, social network analyses such as community detection, link prediction, shortest-paths algorithms cannot be easily expressed in MapReduce. Pregel is one of the most well-known languages designed in response to these issues [27]. Inspired by the Bulk Synchronous Parallel (BSP) computing model [36], Pregel adopts a vertex-centric programming model. A Pregel program consists of iterations of vertex-oriented computations; each vertex processes messages sent from a previous iteration and sends messages to other vertices if necessary. The iterations are separated by a global synchronization point, when the messages from the previous iteration are passed between machines. The need for the programmer to think in vertex-centric programming model and manage the communication adds significant complexity to the programming.

Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/14... $ *10.00.*

1.1 Sequential SociaLite

Many of these graph analyses can be readily expressed in Datalog, a declarative logic programming language often used as a query language in deductive databases [35]. Datalog's support for recursion makes the expression of graph analysis natural; its high-level semantics makes it amenable to parallelization and optimization. However, Datalog's performance in the past has not been competitive. For example, the shortest paths problem was found to run over 30 times slower using LogicBlox [24], a state-of-the-art commercial implementation of Datalog, than a Java implementation of the Dijkstra's algorithm [32].

Recently, we proposed SociaLite, Datalog extensions for efficient graph analysis on sequential machines [32]. Through annotations, the programmers can specify that relations be represented with nested tables, a generalization that enables, for example, edges be compactly represented as adjacency lists. In addition, programmers can specify recursive aggregate functions for efficient evaluation of recursive queries. It was shown in our previous work [32] that recursive *monotone* aggregate functions can be computed efficiently using *semi-naive* evaluation [8]. (The definition of monotone aggregate functions will be given in Section 5.) We also demonstrated in [32] that these extensions can speed up Datalog implementations significantly on a single-core machine, delivering a performance that is comparable to highly optimized Java programs for a set of representative graph algorithms.

1.2 Distributed SociaLite

The analysis of large-scale graphs requires distributed execution across a large number of machines. With its high-level declarative semantics, SociaLite makes it possible for an implementation that hides the low-level complexity in distributed execution from the programmers. Building upon the sequential SociaLite research [32], this paper presents novel language extensions and optimizations for large-scale graph analysis on distributed machines. The contributions of this paper include the following.

Data distribution through sharded tables. A SociaLite programmer does not have to worry about distributing the computation and managing the communication. Instead, he simply specifies how the data are to be decomposed, or *sharded* across the different machines. From the data distribution, the distribution of the computation and required communication is inferred. SociaLite automatically manages the execution across the distributed machines, generates the message passing code, and manages the parallel execution within each instance.

Parallel recursive aggregate function evaluation with delta stepping. Delta stepping [28] has been shown in the past to be effective for parallelizing the shortest paths algorithm. By incorporating this technique in the SociaLite compiler, delta stepping is now made available to any recursive monotone aggregate functions, which is introduced in [32] for semi-naive evaluation.

Efficient approximation via delta stepping and Bloom filter representations. When operating on very large-scale graphs, it might be preferable to have partial answers in reduced execution time. Requiring no involvement on the programmers' part, SociaLite automatically returns partial answers as it processes the Datalog rules using semi-naive evaluation. Delta-stepping is also effective in delivering more accurate partial results. Furthermore, SociaLite allows Bloom filters [9] to be used to represent very large intermediate results approximately and compactly.

Experimental Evaluation. This paper validates the proposed techniques with extensive experimental evaluation. Experiments were performed on the largest real-life graph available to the authors: the Friendster social graph we used has over 120M vertices and 2.5G edges [15]. We also generated synthetic graphs up to 268M vertices for our weak scaling experiments on distributed machines. We experimented with two machine configurations: (1) a large shared memory machine that has a total of 16 cores and 256 GB memory, and (2) 64 Amazon EC2 cluster instances comprising a total of 512 cores. The input benchmark suite consists of 6 representative algorithms used commonly in social network analysis, including shortest paths and PageRank [10]. We also compared the performance of SociaLite with well-known parallel frameworks: Pregel (using Hama [6] and Giraph [4]) as well as MapReduce (using Hadoop [5] and HaLoop [11]).

All the algorithms scale well on the large shared memory machine and track the ideal weak scaling curve within a factor of two on the distributed machines. Our comparison with optimized Giraph (which showed fastest performance among the compared frameworks) shows that our implementation is significantly faster, while the length of the SociaLite programs is only 5% of that of Giraph. Finally, preliminary experiments with our proposed approximate evaluation techniques suggest they are effective in reducing the execution time by trading off a small amount of accuracy.

1.3 Paper Organization

Section 2 first summarizes the sequential SociaLite language [32], and introduces extensions in distributed SociaLite. Section 3 describes the data distribution in SociaLite. In Section 4, parallel rule execution is described. Section 5 explains how recursive aggregate functions are evaluated in parallel with delta-stepping algorithm, and Section 6 describes SociaLite's support of approximate computation. We evaluate the performance and scalability of SociaLite in Section 7. Related work is reviewed in Section 8 and we conclude in Section 9.

2. AN OVERVIEW OF SOCIALITE

Single-source shortest paths is a fundamental graph algorithm used in various graph analyses such as link prediction [22] and betweenness centrality [14]. We will use this algorithm as a running example to explain the Datalog extensions in sequential and parallel SociaLite.

2.1 A Datalog Program

The single-source shortest paths problem can be expressed succinctly in three rules of Datalog as shown in Figure 1(a). Rule (1) says that distance of a path to the source node with id 1 is 0. Rule (2) says that if there is a path from the source node to node s with distance d_1, and an edge from s to t of length d_2, then there is a path to node t with distance $d = d_1 + d_2$. Rule (3) finds the paths with minimum distances.

While this Datalog program is succinct, it is inefficient and cannot handle cyclic graphs. The program wastes much of its execution time, because it first finds all the possible paths with different distances before computing the shortest paths. The program might not even terminate if there are cycles in the input graph.

(a) Datalog:

$$\textsc{Path}(t,d) \ :-\ t=1, d=0.0. \quad (1)$$
$$\textsc{Path}(t,d) \ :-\ \textsc{Path}(s,d_1), \textsc{Edge}(s,t,d_2), d{=}d_1{+}d_2. \quad (2)$$
$$\textsc{MinPath}(t,d) \ :-\ \textsc{Path}(t,\$\textsc{Min}(d)). \quad (3)$$

(b) Sequential SociaLite:

Edge (int s:0..9999, (int t, double $dist$)).
Path (int t:0..9999, double $dist$).

$$\textsc{Path}(t,\$\textsc{Min}(d)) \ :-\ t=1, d=0.0; \quad (4)$$
$$:-\ \textsc{Path}(s,d_1), \textsc{Edge}(s,t,d_2), d{=}d_1{+}d_2. \quad (5)$$

(c) Parallel SociaLite:

Edge [int s:0..9999] ((int t, double $dist$)).
Path [int t:0..9999] (double $dist$).

$$\textsc{Path}[t](\$\textsc{Min}(d)) \ :-\ t=1, d=0.0; \quad (6)$$
$$:-\ \textsc{Path}[s](d_1), \textsc{Edge}[s](t,d_2), d{=}d_1{+}d_2. \quad (7)$$

Figure 1: Single-source shortest paths in (a) Datalog, (b) Sequential SociaLite, (c) Parallel SociaLite. Source node has node id 1.

2.2 Sequential SociaLite Extensions

The sequential version of SociaLite in [32] has two extensions over Datalog: *tail-nested tables* and *recursive aggregate functions*. Due to space constraints, we will only provide a high-level summary of the extensions using the shortest-paths example; details can be found in [32]. Figure 1(b) shows the shortest paths program written in sequential SociaLite.

Tail-nested tables. Graphs are often represented as adjacency lists in imperative programs and not relational tables for efficiency. Sequential SociaLite gives user control over the representation by way of layout annotations, so that data structures like edges can be represented efficiently as adjacency lists. In this example, the relation EDGE is represented as a table with source s in the first column; all edges sharing the same source s are stored in a table, consisting of two columns t (sink) and *dist*. These tables can be *tail-nested*; that is the last column of each table can itself be represented as a table. This representation reduces both the memory usage and computation time needed for graph traversals.

Recursive aggregate functions. Sequential SociaLite supports *recursive* aggregate functions, where an aggregate function like \$MIN can depend on itself. In this example, PATH has two clauses. Rule 4 is the base case where the source node is defined to have path length 0; Rule 5 is the inductive step that says the shortest path length to a sink t is the minimum of $d_1 + d_2$ over all neighboring nodes s, where d_1 is the minimum path length reaching s and d_2 is the length of the edge from s to t.

More formally, the rules of the form

$$\begin{array}{lll} \mathrm{P}(x_1, ..., x_n, F(z)) & :- & \mathrm{Q}_1(x_1, ..., x_n, z); \\ & ... & \\ & :- & \mathrm{Q}_m(x_1, ..., x_n, z). \end{array}$$

yields

$$\{(x_1, ..., x_n, z) | z = F(z'), \forall_{1 \le k \le m} Q_k(x_1, ..., x_n, z')\}$$

The recursive aggregate construct in this example makes clear to SociaLite that it is not necessary to compute all the possible paths, as long as the minimum can be found. This principle lets SociaLite compute the shortest paths even in the presence of cycles; this is also the same basic principle that leads to the Dijkstra's shortest-paths algorithm where computations involving the shortest paths are prioritized. Whereas the Datalog program in Figure 1(a) is a description of the Bellman-Ford algorithm, the sequential SociaLite version shown in Figure 1(b) can be optimized to run with the performance of Dijkstra's algorithm, providing a dramatic performance improvement. This algorithmic transformation is incorporated into the SociaLite compiler and made available to all recursive aggregate functions that are monotone.

2.3 Parallel SociaLite Extensions

Parallel SociaLite asks programmers to indicate how the data are to be partitioned across the distributed machines. It introduces a *location operator* ([]) to be applied to the first column of a data declaration. As illustrated in Figure 1(c), parallel SociaLite is very similar to sequential SociaLite.

The declaration PATH[int t](double *dist*) specifies that PATH is a relation with two columns and that the table is horizontally partitioned, or *sharded*. Suppose the target system has 10 machines, then paths for nodes whose ID is less than 1000 are placed on machine 0; those between 1000 and 1999 are placed on machine 1, and so on. In this case since both PATH and EDGE are similarly declared, the path to node n shares the same machine as edges originating from node n.

For the sake of readability, any mention of a sharded column in a rule is also enclosed with the location operator ([]). In this example, the sharded columns of both PATH and EDGE are joined together, requiring no data movement. However, since the result of the join operation is to be stored in a possibly different location, depending on the value of t (the destination of the edge), data transfers may be necessary.

3. DATA DISTRIBUTION IN SOCIALITE

This section presents the concept of sharding and how the compiler automatically distributes the computation and generates the necessary communication for a distributed machine with the help of programmer-supplied sharding specifications.

3.1 Range-based and hash-based shards

Shards, each containing a number of rows, are placed on different machines in a distributed system. The *location operator*([]) can only be applied to the first column of a relation; the value of the first column in a relation dictates where the relation is located. We refer to the value of the first column in a sharded relation as the *shard key*. We define a function SHARDLOC(r, x) which returns the machine number based on the value of the shard key x in relation r.

There are two kinds of sharding: *range-based* and *hash-based*. If the first column of a sharded array has a range, then the range is divided up into consecutive subranges and evenly distributed across the machines. Suppose the shard key in relation r has range $l..u$, then

$$\textsc{ShardLoc}(r, x) = \left\lfloor \frac{x - l}{\lceil \frac{u-l+1}{n} \rceil} \right\rfloor$$

where n is the number of machines in this system. If no range is given, we use a standard hash function to map the shard key to a machine location; the range of the hashed values is also evenly distributed across all the machines.

The location operator is also used in rule specifications to make it apparent to the programmer where the operands and results are placed.

3.2 Distribution of Computation

Distributed SociaLite makes it easy for users to control the communication pattern without having to write the tedious message passing code. Without loss of generality, consider a join operation with two operands, such as

$$\begin{array}{ll} \mathrm{BAR}[\text{int } x{:}0..9999](\text{int } z). & \\ \mathrm{BAZ}[\text{int } z{:}0..9999](\text{int } y). & \\ \mathrm{FOO}[\text{int } x{:}0..9999](\text{int } y). & \\ \mathrm{FOO}[x](y) \quad :- \quad \mathrm{BAR}[x](z), \mathrm{BAZ}[z](y). & (8) \end{array}$$

Rule 8 specifies that data BAR$[x](z)$ are to be transferred to a machine with ID SHARDLOC(BAZ, z), where the join operation with BAZ$[z](y)$ is performed. The result from the join operation is then transferred back to a machine with ID SHARDLOC(FOO, x).

The goal of this compiler is to let the programmer easily control the parallel execution; using the location operator and re-ordering join operations in a rule body, programmers can easily experiment with different communication patterns without affecting the correctness of a program. For example, Rule 8 could have been written as:

$$\mathrm{FOO}[x](y) \quad :- \quad \mathrm{BAZ}[z](y), \mathrm{BAR}[x](z). \qquad (9)$$

Rule 9 will require broadcasting the table BAZ to all the machines. Whether this performs better depends on the relative sizes of the tables, which is a property that users typically have some understanding of. It is potentially possible to auto-tune the computation dynamically based on the sizes of the data, but such optimizations are outside the scope of this paper.

3.3 Batching the Messages

The SociaLite compiler batches the communication so that data intended for each destination is placed in its own table and sent when enough tuples are stored. For simplicity, the compiler rewrites the rules so that all the computation is performed on local data, and communication is necessary only for sending the result to the right machine.

For example, the compiler will rewrite Rule 8 as:

$$\textsc{Bar}'[\text{int } l{:}0..9999](\text{int } x, \text{ int } z).$$
$$\textsc{Bar}'[z](x,z) \quad :- \quad \textsc{Bar}[x](z). \tag{10}$$
$$\textsc{Foo}[x](z) \quad :- \quad \textsc{Bar}'[z](x,z), \textsc{Baz}[z](y). \tag{11}$$

Rule 10 distributes all the relations of Bar on each machine into a sharded relation Bar′. Each shard destined for a different machine, i.e. $\textsc{ShardLoc}(\textsc{Bar}', z) \neq \textsc{ShardLoc}(\textsc{Bar}, x)$, is sent to the appropriate machine in one message. The results from Rule 11 are similarly sharded and distributed to the corresponding machine.

4. PARALLEL EXECUTION ENGINE

Sequential Datalog programs have a simple execution model. If a Datalog program is *stratifiable*, i.e. there are no negation operations within a recursive cycle, there is a unique greatest fix point solution to its rules. We assume that all our input programs are stratifiable, and that the rules are broken up into a sequence of *strata* such that all negations are applied to results produced from a previous stratum. Each stratum can be executed in sequence, and rules in each stratum can be repeatedly applied until no changes are observed in order to reach the greatest fix point solution. *Semi-naive evaluation* is a well-known and important evaluation optimization technique. It is not necessary to evaluate the rules with the entire solution result over and over again; because the join operations are monotone with respect to the semi-lattice of the solution space, we need only to incrementally compute with the new solutions discovered along the way [32].

Datalog, hence SociaLite, operates on a relation at a time, exposing plenty of opportunities for parallelism by virtue of the size of the data sets. In addition, the functional aspect of the language makes it easy for parallelism to be exploited across rules as well. At a high level, all the machines need to do are to participate in the semi-naive evaluation of the rules until the fix point solution is reached.

Our target machine is assumed to be a network of machines, each of which may have multiple cores. The SociaLite runtime handles these two levels of parallelism efficiently by using message passing across distributed machines and lock-based synchronization across cores on each shared-memory machine.

4.1 Distributed System Architecture

The SociaLite parallel engine consists of a master machine which interprets the Datalog rules and issues work to a collection of slave machines (Figure 2). For large scale operations that involve many machines for a substantial amount of time, it is also important that intermediate work be check pointed occasionally and restorable as needed. We use a fault-tolerant distributed file system [17] for the check pointing. If one or more workers fail, the intermediate states are restored from the latest checkpoint and the evaluation is resumed from that point.

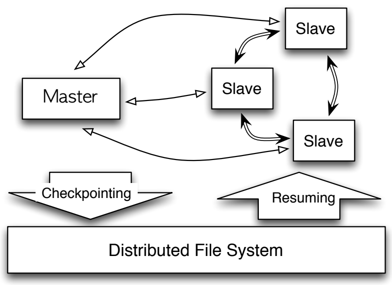


Figure 2: Distributed System Architecture of SociaLite.

The master compiles the SociaLite program into a dependence graph, where each node corresponds to a join operation and each edge represents a data dependence. After finding the strongly connected components, the master organizes all the nodes within a stratum into *epochs*. Recursive strongly connected components are each placed into their own epoch and non-recursive rules can be combined into epochs with the constraint that dependences across the epochs form an acyclic graph. The master then visits the epoch graph in a topological order and instructs the slaves to work on an epoch at a time. Applying semi-naive evaluation, each slave node repeatedly executes the rules upon the arrival of communication from other nodes and updates the internal tables or sends messages to remote nodes as needed.

The following protocol is used to detect when the slaves have quiesced, signifying the completion of an epoch:

1. slaves report its idle status to the master along with a timestamp if there are no more rules to execute and no more data to send. A message is considered sent only when the receiver acknowledges the receipt of the data.
2. Upon receiving an idle status from all the slaves, the master confirms with each slave that it is still idle with the same last reported timestamp. This process is repeated until confirmations from each slave are received.

4.2 Multiple Cores

To support parallelism within a shared memory machine, the sharding hints provided for distributed processing are used to define coarse-grain locks on the data tables in each machine. Each sharded table is further *subsharded*, with each subshard protected by its own lock. A shard is subsharded $32n$ ways, where n is the number of cores supported on the machine. SociaLite breaks up tasks into units that operate on a subshard at a time and are placed in a dynamic task queue.

Each machine has a manager responsible for accepting epoch assignments, reporting and confirming idle status with the master, accepting inputs, placing the corresponding tasks on the work queue, and sending data intended for other machines. Each worker fetches tasks from the work queue, performs the task and updates the resulting data tables.

We have developed two optimizations to minimize the synchronization overhead:

1. Non-recursive epochs are further broken down into *sub-epochs* whose rules are totally independent of each other. Synchronization is necessary only to enforce mutual exclusion between result update operations.
2. No synchronization is necessary if the updated shard is guaranteed to be accessed by only one worker. Consider Rule 8 for example. Since tables Bar and Foo are sharded using the same criteria, each shard in the Foo table is accessed by only one worker as illustrated in Figure 3. The figure shows the evaluation of Rule 8 in a 3-core processor with 3 subshards. The color of the rows/shards indicates that they are accessed by the core with the same color. Different cores write to different subshards, so no synchronization is needed, leading to faster performance.

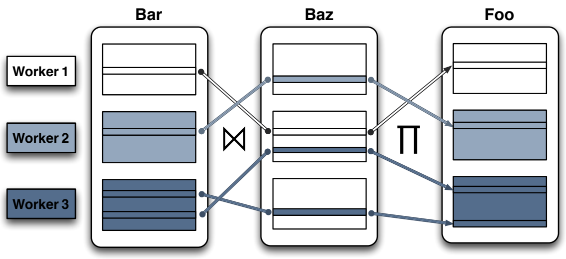


Figure 3: Parallel Evaluation of Rule 8, Foo $[x](y)$ **:-** Bar $[x](z)$**,** Baz $[z](y)$**.**

5. PARALLELIZING RECURSIVE AGGREGATE FUNCTIONS

As noted above, semi-naive evaluation is an important optimization to eliminate redundant evaluation in Datalog. Semi-naive evaluation can also be applied to some aggregate functions, such as the minimum operation. We keep track of the current minimum value, and we can simply update it with a lower value if one is discovered in the course of the semi-naive evaluation process. Note that this does not work for summation unless we know that the new contribution has not already been accounted for in the running total. The main distinction between the two is that minimum is a *meet* operator (i.e. it is idempotent, commutative, and associative), and summation is not.

More formally, semi-naive evaluation can be used to compute the greatest fix point of a Datalog program with an aggregate function g if g is a *meet* operator and the rest of the program is monotone with respect to the partial oder defined by g [32]. In short, we refer to such aggregate functions as *monotone*.

For monotone aggregate functions, the sequential SociaLite compiler uses a priority queue to keep track of all the newly created tuples to be evaluated. By operating on the lowest values (with respect to the semi-lattice of solutions), the solution converges quickly to the greatest fix point, yielding a behavior similar to that of Dijkstra's shortest-paths algorithm.

It is not suitable for distributed SociaLite to use a priority queue, as it will serialize the evaluation of aggregate functions. Meyer and Sanders introduced a technique called *delta stepping* that has shown to be effective in computing shortest paths for large-scale graphs [28]. Madduri and Bader have shown that the parallel implementation of the delta-stepping algorithm gives near linear speedup on Cray MTA-2 [26]. We have generalized this technique and incorporated it into our SociaLite Parallelizing Compiler.

5.1 Delta Stepping

Dijkstra's algorithm can be considered as a series of relaxations. It initializes all tentative distances from a single source to ∞. Starting with the source node, each relaxation step picks the node with the shortest, newly found distance from the source to update the distances for its immediate neighbors. The algorithm is serial because relaxation is applied to only one node at a time.

Delta stepping eliminates the serialization by relaxing multiple nodes in each *delta step*. The algorithm is parameterized by the delta value, Δ. It is an iterative algorithm where each step i targets all the nodes whose shortest distance from the source is between $(i-1)\times\Delta$ and $i\times\Delta$. This step is repeated until all the minimum paths are found.

Furthermore, each delta step i is separated into two substeps. The first is an iterative step that applies relaxation to all the *light* edges, edges shorter than Δ. The resulting distance may be within length $i\times\Delta$, thus adding more nodes to be relaxed. The second applies relaxation to all the *heavy* edges, edges no shorter than Δ. This second step needs not be repeated because the updated distances are guaranteed to exceed the range considered in the current step.

Note that as Δ approaches zero, the algorithm becomes identical to Dijkstra's. In other words, this technique expands the number of paths that can be evaluated at a time, thus allowing for more parallelism.

5.2 Delta Stepping in SociaLite

SociaLite only optimizes linear recursions involving aggregate functions which the programmer have declared as monotone. It uses heuristics to statically determine the equivalent of light and heavy edges in the recursive Datalog rules. Then it samples the variables contributing to the new aggregate value, the edge length in this case, and chooses a 20% percentile value as the threshold separating light from heavy values. Where the heuristics fail in identifying the contributing parameters, delta stepping is simply run without distinguishing between light and heavy edges.

Our SociaLite runtime system uses the same delta-stepping algorithm as described in Section 5.1 in parallelizing recursive monotone aggregate functions. The updated tuples are stored in prioritized buckets according to the values of the results, where the priority order is inferred from the partial order defined by the monotone aggregate function. We adopt a combination of coarse and fine-grain bucketing scheme to handle the cross-machine and intra-machine levels of parallelism. On a shared memory machine where the tasks are dynamically scheduled to balance the load across cores and synchronization overhead is low, fine-grain bucketing is desired. With longer network latency and static partitioning across machines, coarser-grain bucketing with larger Δ values is a better choice.

This optimization shows off the power of SociaLite as a high-level language. The user only has to annotate how the data are distributed and whether the recursive aggregate functions are monotone. The compiler automatically takes care of the distribution of computation, the communication, as well as bringing to bear algorithmic transforms like delta stepping. This is particularly significant considering the complexity of writing and debugging parallel code.

6. APPROXIMATE COMPUTATION

The social graphs today are so large that it may not be computationally feasible to perform even innocent-looking queries. Take, for example, the question of the size of one's online social network. The average distance of any two Facebook users is found to be only 4.7 in May 2011 [34]. It was also found that 99.91% of Facebook users were interconnected, forming a large connected component [34]. It is thus computationally expensive even to compute one's friends of friends let alone all the people we can reach transitively through friendships.

It is therefore important that we can compute the results in an approximate fashion. The high-level semantics of SociaLite makes it relatively easy to implement approximations automatically.

6.1 Early Termination

Consider the simple example of querying for one's friends of friends. The size of results varies considerably depending on the person asking the query. While 10% of the users have less than 10 friends in Facebook, the average number of friends is 190, and 10% have over 500 friends [34]. So an average friends-of-friends list would have 36,000 entries, with the size ranging from very small to over 100,000 entries. Not only would it be expensive to create all these results for all users, it is also a waste of resources if the users intend to peruse the list manually. In addition, for the sake of a faster response, it might be desirable to provide partial results to the user instead of waiting until all the results are available.

The semi-naive evaluation of SociaLite rules supports approximate computation trivially. Approximation is achieved by simply terminating each epoch of the execution before the fix point is reached. Results can be presented to the end users as they are generated. In the case of recursive monotone aggregate functions, delta-stepping based prioritization not only speeds up the computation, it also improves the quality of the approximation greatly. Typically, the quality improves quickly at the start, and the marginal improvement slows down as time goes on. Empirical evidence of this trend is shown in Section 7.

6.2 Bloom-Filter Based Approximation

It has been well known that one's friends of friends network is important in how one finds new information [18]. However, due to small-world characteristics of social networks, the size of one's friends of friends network may be very large; hence, it may not be feasible to perform friends-of-friends queries efficiently. However, if the friends-of-friends subgraph is just an intermediate result used to answer further queries that have a small answer, such as the number of friends of friends that have a certain attribute (Figure 4), we can approximate this answer quite accurately and quickly by using *Bloom filters* [9].

We introduce the use of Bloom filter as a means to provide a quick approximation to the case where the final result may be relatively small, but the intermediate results are large. An important operation in semi-naive evaluation is to find

$$\textsc{Foaf}(n, f_2) \quad :- \quad \textsc{Friend}(n, f), \textsc{Friend}(f, f_2). \quad (12)$$
$$\textsc{FoafSum}(n, \$\textsc{Sum}(a)) \quad :- \quad \textsc{Foaf}(n, f_2), \textsc{Attr}(f_2, a). \quad (13)$$

Figure 4: A SociaLite Program Computing Local Aggregation

the difference between the result of a rule application from the results obtained so far. If the result accumulated so far is large, this can be very expensive both in terms of computation time and memory consumption.

A Bloom filter is a space-efficient probabilistic data structure for testing set membership. A Bloom filter is a bit array, all set to zero initially. Insertion of an element involves setting k locations in the array according to the results of applying k hash functions to the element. To test for membership, one needs only to check if those k positions hashed to by an element are 1. Thus, false positives are possible but not false negatives. That is, the Bloom filter may indicate that a member is in the set while it is not, but vice versa is not possible. The chance of false positives increases with the size of the set represented. It is not possible to remove elements from a set represented by a Bloom filter, nor is it possible to enumerate the elements in the set efficiently.

When intermediate sets get too large to be represented exactly, we can use a Bloom filter to represent them compactly. It is efficient to check if newly generated tuples are already included in the Bloom filter. There is a possibility that an element is erroneously reported to be in the set when it is not. However, if we have to approximate anyway because of the cost, the use of Bloom filter will let us generate approximate results in a shorter amount of time in the end.

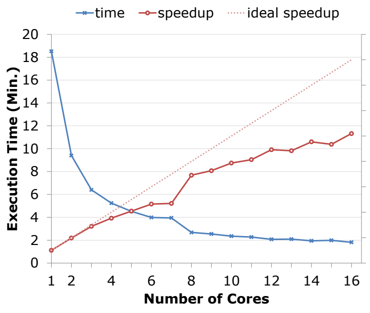
The use of Bloom filter as an intermediate representation is coupled with pipelined evaluation of the relevant SociaLite rules, since it is not possible in general to enumerate the results in a Bloom filter. (Pipelined evaluation was introduced to execute SociaLite rules in lock step to improve temporal locality [32].) In the example in Figure 4, each member of the second column of the Foaf table is represented as a Bloom filter. Rules 12 and 13 are pipelined so that as new friends-of-a-friend are found in rule 12, the tuples are pipelined to rule 13 to compute the sum of an attribute.

7. EVALUATION

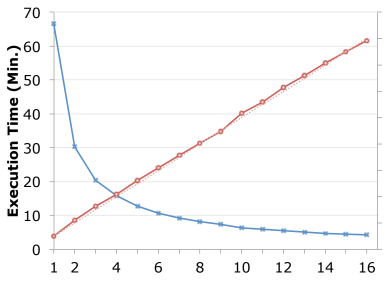
We have a fully working SociaLite compiler and runtime system that automatically translates the SociaLite code to Java code for a distributed system of multi-core machines. Here we present five sets of experimental results. We evaluated a set of core graph analyses on a large multi-core machine as well as 64 Amazon EC2 instances. We also compared four parallel/distributed frameworks (Hama [6], Giraph [4], Hadoop [5], and HaLoop [11]) using the shortest-paths algorithm. Then we compared SociaLite with Giraph that showed the fastest performance among the four frameworks. Finally, we present some preliminary experiments with our approximate evaluation optimizations.

7.1 Benchmark Algorithms

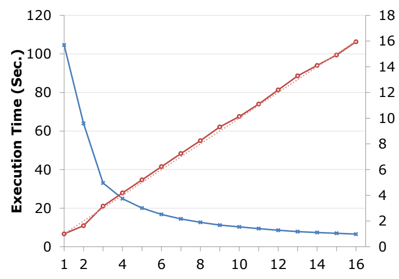
For our evaluation, we use six core graph analyses that have been shown to be useful for a wide range of problems such as community detection, link prediction, and other general graph metrics [14, 22, 31]. The first two operate on



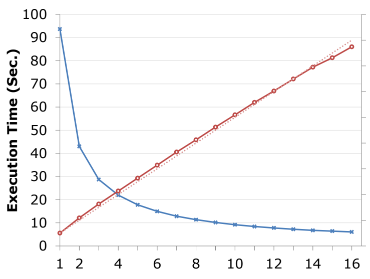
(a) Shortest Paths



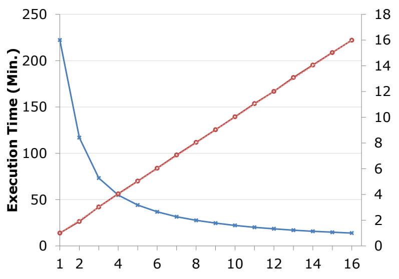
(b) PageRank



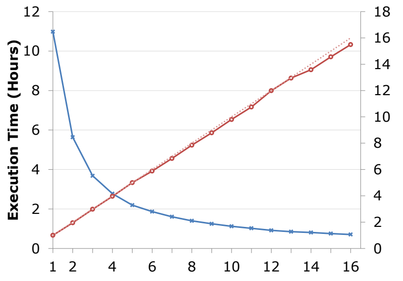
(c) Mutual Neighbors



(d) Connected Components



(e) Triangles



(f) Clustering Coefficients

Figure 5: Speed up on a multi-core machine with 16 cores.

directed graphs and the rest work on undirected graphs. Edges in the latter are generally represented as a pair of unidirectional edges.

Shortest Paths: Find shortest paths from a source vertex to all other vertices in the graph. This is used in many analyses including betweenness centrality [14].

PageRank: PageRank [10] is a link analysis algorithm that computes the importance of vertices in a graph.

Mutual Neighbors: Find all vertices that are common neighbors of a pair of vertices. This algorithm is typically used for link prediction [22].

Connected Components: Find all connected components in a graph. A connected component is a maximal connected subgraph in which every pair of vertices is connected by at least one path.

Triangles: Counts all triangles (i.e., cliques of size three) in a graph. Triangles serve as a graph metric as well as the basis of analysis algorithms such as the clique percolation algorithm [31].

Clustering Coefficients: A graph metric that measures connectivity of a graph. Local clustering coefficient and network clustering coefficient are computed. The former shows how connected the neighbors of a vertex are. The latter is the average of the local clustering coefficients of all vertices.

All benchmark algorithms are executed on the entire data set, except for Mutual Neighbors, which is evaluated on selected pairs of vertices (the number of pairs is set to be same as the number of vertices in a given graph). In Connected Components, we used an algorithm that broadcasts its identity from each vertex, and a receiving vertex takes a minimum value, which represents the identity of the component.

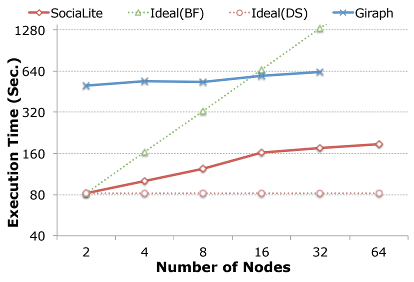
7.2 Multi-Core Parallelization

Our first experiment is to evaluate the effectiveness of SociaLite on a multi-core machine with a large memory. The machine we use is a dual-socket Intel Xeon E5-2670 with Sandy Bridge micro-architecture with 256GB memory. Each socket has 8 cores with 20 MB LLC (last-level cache), hence in total the machine has 16 cores. For this experiment, we use as input the social graph from Friendster, the largest real-world social graph available to us. Consisting of 120M vertices and 2.5G edges [15], about an eighth the size of the largest existing social graph, the Friendster graph fits in the memory of a target machine.

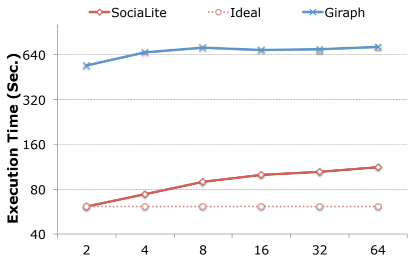
We show the execution time and speed up for each application in Figure 5. We first note some of these algorithms take a long time to execute on a single core. Both Connected Components and Mutual Neighbors take about 100 seconds, which is about the time to perform a small number of operations per vertex in the graph. Next, Shortest Paths, PageRank, and Triangles take 18, 68, 225 minutes, respectively. Finally, Clustering Coefficients takes about 11 hours to complete. Clearly, all these algorithms, especially the latter four, could benefit from faster execution.

All but the Shortest Paths program are data-parallel algorithms, and they all scale nearly linearly up to 16 cores. The strategy of sharding the data arrays locally and using a dynamic scheduler appears to balance the load well across the cores, resulting in nearly perfect speedup.

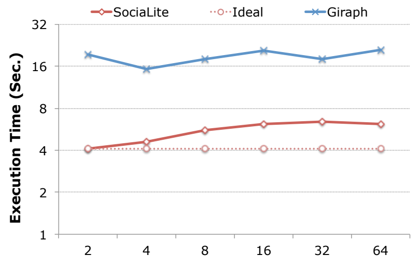
Shortest Paths is much harder to parallelize effectively because of the data dependences in the program. The speed up starts out behaving almost linearly and gradually settling to a speed up of about 10 for 16 cores. As described in Section 5, delta stepping is used in evaluating the Shortest Paths problem. The amount of parallelism available in each delta step varies according to the characteristics of the graph. Although the speed up is not perfect, the reduction of execution time from over 18 minutes to under 2 minutes makes a significant difference to the practicality of computing shortest paths.



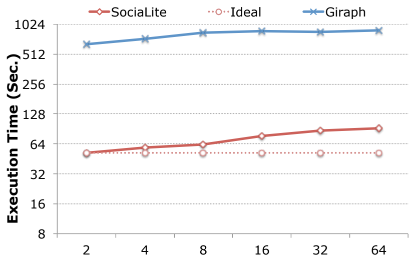
(a) Shortest Paths



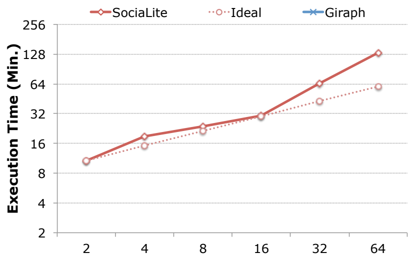
(b) PageRank



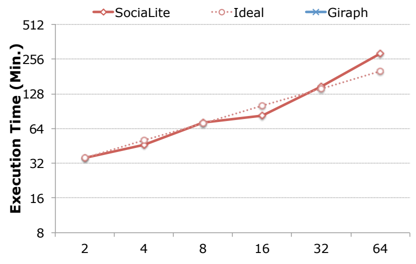
(c) Mutual Neighbors



(d) Connected Components



(e) Triangles



(f) Clustering Coefficients

Figure 6: Weak scaling of SociaLite and Giraph programs (Input graphs for 2, 4, 8, 16, 32, 64 instances have 8M, 17M, 34M, 67M, 134M, and 268M vertices, respectively.)

7.3 Parallelization On a Distributed Machine

As data sizes scale up, it is necessary to run the analysis on distributed machines, for both increased memory and computational power. Our experiment is performed on 64 Amazon EC2 High-Performance Cluster (HPC) instances. The HPC instances are dedicated machines as opposed to typical EC2 instances which are virtual machines. The instances are connected via a 10 Gigabit Ethernet network, and each instance has a dual socket Intel Xeon X5570 and 23GB of memory. Each X5570 processor has 4 cores, hence a single machine has 8 cores; in total the 64 instances have 512 cores. Out of the 23GB memory, 18GB is given to SociaLite slave nodes; the rest is used by the distributed file system. Out of the 8 cores, we dedicated 6 cores for computation (using 6 computation threads), and 2 cores for network communication.

With distributed machines, we are interested mainly in *weak scaling* [19], the ability to use more machines to handle larger data sets. For uniformity across data sets of different sizes, we generate synthetic graphs as inputs to our graph analyses, using the RMAT algorithm [12]. Using the Graph 500 graph-generator [33], we generated six graphs of 8.4M, 16.8M, 33.6M, 67.1M, 134.2M, and 268.4M vertices, all having sixteen edges per vertex on average. We double the input size for each doubling of machine instances to keep the data per instance to be constant for the weak scaling experiment.

The ideal weak scaling graphs for the benchmarks and the performance of the SociaLite implementations, up to 64 instances, are shown in Figure 6.

As PageRank, Mutual Neighbors, Connected Components are linear algorithms, the weak scaling graph is simply flat. For shortest paths, delta stepping for a graph with random edge weights is linear for all but degenerate cases [28]. For the sake of comparison, we note that the ideal scaling graph of the Bellman-Ford algorithm is $O(p)$, where p is the number of instances if the number of edges is $O(n)$.

Shortest Paths, PageRank, Mutual Neighbors, and Connected Components all share the same communication pattern. Namely, communication is necessary for each edge whose source and destination are on different machines. Given $O(n)$ edges, the communication volume is $O\left(\frac{p-1}{p}\cdot n\right)$ where p is the number of machines. The amount of communication grows from $n/2$, $3n/4, \ldots,$ to n asymptotically. Thus, the time of execution grows with an increased number of instances, more so at the beginning, and settles at twice the time needed with ideal weak scaling.

Note that for Shortest Paths, the SociaLite program runs significantly faster than the Bellman Ford algorithm. This is notable because the programmer is essentially writing a Bellman Ford algorithm and SociaLite automatically turns this into a delta-stepping algorithm. This illustrates the advantage of using a high-level language with an optimizing compiler.

In Triangles, for each edge $(x, y) \in E$, we need to check if any of the neighbors of x is connected to y. Using binary search, the number of memory operations for Triangles is thus $O\left(\sum_{\langle x,y\rangle\in E} d_x \log(d_y)\right)$, where d_x is the degree of the vertex x. For RMAT graphs with a power-law degree distribution, the summation is experimentally measured to be $O(n^{\frac{3}{2}})$. The complexity of Clustering Coefficients is essentially the same as Triangles, thus the weak scaling for both algorithms is $O(p^{\frac{1}{2}})$.

The communication volume of Triangles and Clustering Coefficients is $O\left(\frac{p-1}{p}\sum_{\langle x,y\rangle\in E} d_x\right)$, where p is the number of machines, E is the set of edges, and d_x is the degree of the vertex x. For RMAT graphs, we determine empirically that the volume is close to $O(n^{\frac{3}{2}})$. This suggests that the communication volume increases faster than the weak scaling for the algorithms ($O(p^{\frac{1}{2}})$). In our experimental result, both Triangles and Clustering Coefficients track the weak scaling curve up to 16 nodes and start to deviate subsequently, as the communication volume increases relatively faster, hence the network bandwidth becomes bottleneck in the performance.

7.4 Comparison with Other Frameworks

To evaluate how distributed SociaLite compares with other parallel frameworks, we perform two experiments. The first is to compare the performance of a number of common frameworks in executing Shortest Paths, one of the more challenging graph algorithms. We then take the one found to have the best performance and compare its performance for all the benchmarks with that of SociaLite.

7.4.1 Comparison of Existing Parallel Frameworks

MapReduce and Pregel are two of the most popular large-scale distributed frameworks. Because we have no access to these proprietary systems, we performed our experiments on open-source implementations of these execution models: Hadoop and HaLoop for the MapReduce programming model, and Hama and Giraph for the Pregel-like vertex-centric programming model. Note that Hadoop and Giraph are actively used in Facebook and Yahoo!.

Shortest Paths for Hama and Giraph is implemented as described in the paper on Pregel [27]; each vertex iterates over messages from neighbors to update its current distance, and the vertex sends its neighbors their potential distance values. To our surprise, our first implementation in Hama and Giraph ran more than two orders of magnitude slower than the SociaLite version on a single machine. Upon close examination, we found that the Shortest Paths program has very large memory footprints in both Hama and Giraph. Specifically, they store primitive type values as boxed objects, incurring significant penalty on performance and memory usage. We reimplemented several data structures of Hama and Giraph so as to store the primitive types directly. This optimization proved to be very effective; for example, on a graph with 131K vertices and 1M edges, optimized Shortest Paths in Hama took only 5 seconds, while an unoptimized version took more than 30 seconds.

It required considerably more effort to implement Shortest Paths in MapReduce model than Pregel. Our implementation takes weighted edge list and a distance value of a vertex as input to a mapper, and the reducer receives vertex ID and its distances to compute the minimum distance for the vertex. We iterate the map/reduce tasks multiple times. The Hadoop version requires another map/reduce stage between the iterations to determine if a fix point is reached, whereas in HaLoop the fix point is detected by its built-in support that compares the reducer outputs of the current iteration with the cached ones from the previous iteration.

Figure 7 compares the execution times of Shortest Paths in the four frameworks. The Hama and Giraph programs are implemented with the optimizations described earlier. We can see that the optimized Giraph version is significantly faster than other implementations, 2 to 4 times faster than the Hama version, and 20 to 30 times faster than the Hadoop/HaLoop versions.

The program sizes are more or less the same – they are around 200 lines of code (253 for Hama, 232 for Giraph, 215 for Hadoop, and 185 for HaLoop). If we include the effort for the optimizations in Hama and Giraph, the programming complexity in the four frameworks was, in our experience, more or less the same.

# Instances	Opt. Hama	Opt. Giraph	Hadoop	HaLoop
2	20.5 (min.)	8.3	230.8	183.7
4	25.0	8.9	259.9	208.2
8	33.7	8.9	275.2	220.2
16	42.5	9.9	290.9	235.1

Figure 7: Comparison of the execution times (in minutes) of Shortest Paths on EC2 instances with graphs having 8M, 17M, 34M, and 67M vertices.

7.4.2 Comparison with Optimized Giraph

We selected optimized Giraph for the full comparison with SociaLite since it showed the fastest performance in executing Shortest Paths. Hereafter, we simply refer to the optimized Giraph as Giraph. All the benchmarks described in Section 7.1 are implemented in Giraph. We found that it was nontrivial to implement Triangles and Clustering Coefficients in Giraph as they cannot be easily expressed using the message passing abstraction. As shown in Figure 8, SociaLite programs are 11 to 58 times more succinct compared to the Giraph programs, on average 22 times succinct. If the number of lines of code is indicative of the programming complexity, then it is much easier to write these analyses in SociaLite than in Giraph.

	Giraph	SociaLite	Ratio
Shortest Paths	232	4	58
PageRank	146	13	11
Mutual Neighbors	169	6	28
Connected Components	122	9	13
Triangles	181	4	45
Clustering Coefficients	218	12	18
Total	1,068	48	22

Figure 8: Number of non-commented lines of code for Giraph and SociaLite programs, and the ratio of the size of Giraph programs over that of SociaLite programs.

Figure 6 compares the execution times of SociaLite and Giraph programs on 2 to 64 instances. Similarly as in SociaLite programs, we dedicated 6 cores for computations in Giraph programs; the rest 2 cores are used for network communication.

For two of the benchmarks (PageRank, Connected Components), SociaLite programs performed almost an order of magnitude faster than Giraph counterparts across all the machine configurations. For Shortest Paths and Mutual Neighbors, SociaLite programs performed significantly faster although not an order of magnitude faster.

Note that the Giraph programs for Triangles and Clustering Coefficients ran out of memory, so no results are reported. The Shortest Paths program also failed similarly when run on the largest graph with 64 instances. Because of the way they adopted the Bulk Synchronous Parallel (BSP) model in Giraph (and Pregel), each machine must have enough physical memory to buffer all the messages from one iteration so as to process them in the next. This memory requirement turned out to exceed the memory capacity in the case of Triangles, Clustering Coefficients, and the largest configuration for Shortest Paths.

7.5 Approximate Evaluation

In this section, we present some empirical evidence to show that our proposed approximate computation techniques can provide a good tradeoff between time and accuracy. The experiments described here are performed on the LiveJournal social graph [23], which has 4.8M vertices and 68.9M edges, running on an Intel Xeon E5-2640 having 6 cores with 80GB memory.

7.5.1 *Early Termination with Delta Stepping*

Single-source shortest paths, used in important graph analyses like betweenness centrality [14] and link prediction [22] are quite expensive, as seen from Figure 5. The cost for computing shortest paths for just one source is substantial, taking over 100 seconds on a single machine on a relatively small graph compared to the typical social graphs in Facebook or Google. Imagine now that we have to perform this computation for every user of interest!

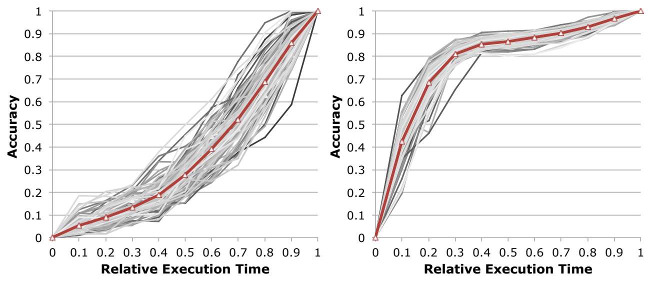
On the LiveJournal data, we applied the shortest-paths algorithm to 100 randomly selected source vertices with and without delta stepping. We found that the execution time ranges from 9.3 to 15.9 seconds, taking an average of 12.3 seconds, without delta stepping. With delta stepping, the executing time ranges from 4.1 to 7.6 seconds, with an average of 5.1 seconds. Delta stepping improves the performance by an average of 2.4 times.

What if we cannot devote an average of 5.1 seconds to each user? Thanks to semi-naive evaluation, we can just read off the answer derived so far at any given instant as an approximate solution. For the shortest paths problem, we measure accuracy as the fraction of the number of vertices whose path lengths are found to be within 10% of the optimal. For each of the 100 trials, we ran the shortest paths algorithm for different lengths of time and measured the accuracy of the results obtained. The tradeoffs between time and accuracy for each of these trials, with and without delta stepping, are shown in Figure 9. The red, bold line shown in the graph represents the average across the 100 different sources. Note that all the execution times are normalized; the graph plots the accuracy attained with different fractions of the time needed to compute the perfect answer.

We see distinctly that the graph without delta stepping is concave, whereas the graph with delta stepping is convex. By iterating over the shorter paths first in delta stepping, accuracy improves quickly at the beginning; also less effort is wasted, resulting in a faster total execution time. With delta stepping, it takes less than 30% of the computation time for 80% of the vertices to get within 10% of the optimal; it takes 90% of the computation without delta stepping to achieve the same accuracy. Accounting for the absolute differences in execution time, delta stepping takes on average just 1.5 seconds to provide the same answer as the algorithm without delta stepping in 11.1 seconds. This represents a speedup of 7.4 times. This suggests that delta stepping is useful for both full and approximate answer calculations for recursive monotone aggregate functions.

7.5.2 *Bloom-Filter Based Approximation*

For the sake of testing the effectiveness of the Bloom filter, we also ran the friends-of-friends query shown in Figure 4 on the LiveJournal data. Without approximation, the program required 26 GB of memory to process a graph with 4.8M vertices and 68.9M edges, as shown in Figure 10. With the use of a Scalable Bloom filter [1] to represent the FOAF table, the program required only 3 GB of memory, and it ran 32.8% faster than the exact execution. With approximation, it is possible to miss the contribution of some friends of friends, hence the results may be smaller than the exact answers. We found that the answers for 92.5% of the vertices in the graph have less than 10% errors when the Bloom Filter initially uses 5.5 bits per element. At least for this example, the tradeoff between speed and accuracy appears to be worthwhile.



(a) Without Delta Stepping (b) With Delta Stepping

Figure 9: Accuracy and computation time tradeoffs of Shortest Paths Program. The red, bold line shows the average of all the 100 executions.

	Exact	Approximate	Comparison
Exec Time (Min.)	28.9	19.4	32.8% faster
Memory Usage (GB)	26.0	3.0	11.5% usage
Accuracy (≦ 10% error)	100.0%	92.5%	

Figure 10: Effect of Bloom Filter on Execution Time and Accuracy for the Program in Figure 4

7.6 Discussion

Our experiments show that SociaLite is succinct and delivers high performance on both parallel and distributed machines. In our experience, students in one of our research projects found the syntax of SociaLite foreign initially. However, because SociaLite allows recursive graph relationships be expressed naturally and directly, the students learned quickly and implemented many queries with little difficulty. To write multi-threaded or distributed code, programmers only need to decide how data are to be sharded to minimize communication. They can simply add the sharding specification into the sequential SociaLite programs, which typically requires no additional lines of code. In contrast, even programmers well versed in the MapReduce or vertex-centric programming models like Pregel and Giraph would require significant effort to express graph algorithms in those models. We found that the SociaLite benchmarks are 11 to 58 times more succinct than Giraph programs.

The high-level semantics of SociaLite not only eases programming but also leads to better performance. SociaLite programs run 4 to 12 times faster than Giraph programs thanks to an efficient parallel execution engine and special optimizations for recursive aggregate functions. We also show that SociaLite can automatically derive approximate algorithms for some of these graph analyses. In contrast, high-level code optimization and transformation is difficult for imperative languages due to over-specification in such programs.

8. RELATED WORK

While Datalog was introduced in the 70s, it has gained popularity recently and has been used in commercial systems [24] as well as research projects. Its high-level semantics has been found to be useful to simplify programming across many domains, including program analysis [37], network systems [2, 3, 25], and modular robotics [7]. In the following, we focus on related work in parallel and distributed Datalog as well as graph analysis.

Parallel and Distributed Datalog Evaluation. There has been much work in the parallel evaluation of Datalog programs [16, 39]. Ganguly et al. proposed parallelizing Datalog evaluation by partitioning the set of possible variable substitutions; an extra constraint with a hash-based *discriminating* function is added to a rule body, so that each processor is responsible for only a portion of possible variable substitutions [16]. SociaLite's parallelization is based on sharding; shards of the first predicate of a rule are joined by parallel workers. High-level data decomposition is specified by users, giving a certain degree of control to users for improving efficiency.

More recently, Datalog has found its way into network and distributed system research [2, 3, 25]. For the sake of expressing distributed network state, NDlog and its successor Overlog extended Datalog with a location operator, which indicates the storage location of tuples [2, 25]. Dedalus [3] further extended Datalog with two features: mutable states as well as asynchronous processing and communication. These features help with the reasoning of distributed states, making it easy to implement distributed services, such as Paxos [21].

While there are similarities, the motivations, semantics, and design of the location operator in NDlog/Overlog are completely different from that in SociaLite. NDlog and Overlog are intended to be used where programmers want to describe the functions of the individual nodes in a distributed network. As such, the location operator denotes an actual machine address. In SociaLite, on the other hand, the use of a distributed machine is just a means of completing a task faster. The programmer has no need to dictate the computation on the individual nodes. SociaLite's location operator is just a simple hint to indicate that the given table is to be sharded with respect to the operand column. The details of the data distribution, such as the actual machine addresses of tuples, are abstracted away.

Distributed Data Analysis. Section 7.4 provides a comparison between the MapReduce model and the Pregel model. GraphLab is a distributed machine learning and data mining (MLDM) system that adopts vertex-centric programming model similar to Pregel, but with the support for asynchronous computation. Whereas Pregel, GraphLab, and MapReduce provide relatively low-level procedural constructs, SociaLite is an extension of a Datalog, a high-level declarative query language. This high-level semantics simplifies programming while facilitating optimizations like prioritized and approximate evaluation.

REX [29] supports incremental iterative computation with customized delta operations and handlers. With explicit processing of deltas, it is possible for users to set a customized termination condition for recursive queries or control the propagation of updates from one iteration to another. In SociaLite, with the generalization of delta-stepping algorithm, we *prioritize* the evaluation of updates, where the priority is automatically inferred from the recursive monotone aggregate functions.

Declarative Query Languages. The rising need for large-scale data analysis has prompted the development of a number of declarative query languages for distributed systems. Pig Latin [30] is a query language that compiles to run on the Hadoop infrastructure. DryadLINQ [38] is an SQL-like query language that extends the C# programming language. DryadLINQ queries are compiled to run on Dryad [20], which is Microsoft's distributed computing platform. Compared to the aforementioned query languages, SociaLite is better suited for graph algorithms, many of which can benefit from the support of recursion.

9. CONCLUSION

With online social networks such as Twitter and Facebook boasting of hundreds of millions and billions of vertices, it is necessary to harness the power of large-scale distributed systems in analyzing these networks. Vertex-centric computation, as embodied by Pregel, a state-of-the-art language for such analyses, requires programmers to manage the parallelism and communication at a very low level.

This paper shows that, with just a few annotations, programmers can describe these graph algorithms naturally in a few Datalog rules and that a compiler can manage the distributed computation effectively. The programmer simply specifies how tables are to be sharded across the machines, and SociaLite automatically decomposes the computation and generates the communication code. It also generalizes the delta stepping technique to optimize recursive monotone aggregate functions for parallel execution. The semi-naive evaluation framework in SociaLite can produce partial results trivially; this is especially important for social queries since fast response times are often more important than accuracy. In addition, it uses Bloom filters as an approximate data structure for storing large intermediate values.

We evaluated SociaLite with a suite of core algorithms used in many graph analyses. We found that all, except the shortest paths program, scaled linearly up to 16 cores on a shared memory machine. The shortest paths program showed a speed up of 10 on 16 cores. The programs tracked the ideal weak scaling curve within a factor of two in our experiment with 64 Amazon EC2 8-core instances. The SociaLite programs are found to be 22 times more succinct on average when compared to Giraph, an open-source alternative to Pregel. Our proposed approximate evaluation techniques are found to be effective on the couple of examples we experimented with.

The high-level semantics of SociaLite makes it possible for the system to parallelize the code effectively and to trade off accuracy for performance without user intervention. Furthermore, as a language for deductive databases, SociaLite makes it easy for programmers to write many interesting social applications that leverage the core graph algorithms such as those evaluated in this paper.

Acknowledgment

We thank Martin Rinard for discussions on this paper. This research was funded in part by NSF Programmable Open Mobile Internet (POMI) 2020 Expedition Grant 0832820, Stanford MobiSocial Computing Laboratory, which is sponsored by AVG, Google, ING Direct, Nokia, Samsung, and Sony Ericsson.

10. REFERENCES

[1] P. S. Almeida, C. Baquero, N. M. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

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

[3] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In *Datalog*, pages 262–281, 2010.

[4] http://incubator.apache.org/giraph.

[5] http://hadoop.apache.org.

[6] http://hama.apache.org.

[7] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A language for large ensembles of independently executing nodes. In *ICLP*, pages 265–280, 2009.

[8] F. Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems (Islamorada)*, pages 165–178, 1985.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, pages 422–426, 1970.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7*, pages 107–117, 1998.

[11] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

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

[14] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[15] http://archive.org/details/friendster-dataset-201107.

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

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[18] M. Granovetter. The Strength of Weak Ties. *The American Journal of Sociology*, 78(6):1360–1380, 1973.

[19] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[22] D. Liben-Nowell and J. M. Kleinberg. The link-prediction problem for social networks. *JASIST*, 58(7):1019–1031, 2007.

[23] http://www.livejournal.com.

[24] Logicblox inc. http://www.logicblox.com/.

[25] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.

[26] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, pages 23–35, 2007.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[28] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA*, pages 393–404, 1998.

[29] S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: Recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, 2012.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[31] G. Palla, I. Derenyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814, 2005.

[32] J. Seo, S. Guo, and M. S. Lam. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.

[33] http://www.graph500.org.

[34] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, 2011.

[35] J. D. Ullman. Principles of database and knowledge-base systems, volume ii. 1989.

[36] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[37] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[38] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[39] W. Zhang, K. Wang, S.-C. Chau, and S.-C. Chau. Data partition and parallel evaluation of datalog programs. *IEEE Trans. Knowl. Data Eng.*, 7(1):163–176, 1995.