SociaLite: Datalog Extensions for Efficient Social Network Analysis

Jiwon Seo Stephen Guo Monica S. Lam
Computer Systems Laboratory
Stanford University
Stanford, CA, USA 94305
{jiwon, sdguo, lam}@stanford.edu

***Abstract*— With the rise of social networks, large-scale graph analysis becomes increasingly important. Because SQL lacks the expressiveness and performance needed for graph algorithms, lower-level, general-purpose languages are often used instead.**

For greater ease of use and efficiency, we propose SociaLite, a high-level graph query language based on Datalog. As a logic programming language, Datalog allows many graph algorithms to be expressed succinctly. However, its performance has not been competitive when compared to low-level languages. With SociaLite, users can provide high-level hints on the data layout and evaluation order; they can also define recursive aggregate functions which, as long as they are meet operations, can be evaluated incrementally and efficiently.

We evaluated SociaLite by running eight graph algorithms (shortest paths, PageRank, hubs and authorities, mutual neighbors, connected components, triangles, clustering coefficients, and betweenness centrality) on two real-life social graphs, LiveJournal and Last.fm. The optimizations proposed in this paper speed up almost all the algorithms by 3 to 22 times. SociaLite even outperforms typical Java implementations by an average of 50% for the graph algorithms tested. When compared to highly optimized Java implementations, SociaLite programs are an order of magnitude more succinct and easier to write. Its performance is competitive, giving up only 16% for the largest benchmark. Most importantly, being a query language, SociaLite enables many more users who are not proficient in software engineering to make social network queries easily and efficiently.

I. Introduction

In recent years, we have witnessed the rise of a large number of online social networks, many of which have attracted hundreds of millions of users. Embedded in these databases of social networks is a wealth of information, useful for a wide range of applications. Social network analysis encompasses topics such as ranking the nodes of a graph, community detection, link prediction, as well as computation of general graph metrics. These analyses are often built on top of fundamental graph algorithms such as computing shortest paths and finding connected components. In a recent NSF-sponsored workshop on Social Networks and Mobility in the Cloud, many researchers expressed the need for a better computational model or query language to eventually achieve the goal of letting consumers express queries on their personal social graphs [10], [36].

Datalog is an excellent candidate for achieving this vision because of its high-level declarative semantics and support for recursion. The high-level semantics makes possible many optimizations including parallelization and time-bounded approximations. However, the relational representation in Datalog is not a good match for graph analysis. Users are unable to control the data representation or the evaluation. Consequently, the performance of Datalog is not competitive when compared with other languages. For this reason, developers resort to using general-purpose languages, such as Java, for social network analysis. Not only is it more difficult to write analysis programs in general-purpose languages, these programs cannot be parallelized or optimized automatically.

This paper presents SociaLite, an extension of Datalog that delivers performance similar to that of highly optimized Java programs. Our proposed extensions include data layout declarations, hints of evaluation order, and recursive aggregate functions.

A. Performance of Datalog Programs

Consider the example of computing shortest paths from a source node to all other nodes in a graph. Using a previously proposed extension of aggregate functions [3], [24], shortest paths can be succinctly expressed in Datalog as shown in Figure 1. Here, the first statement declares that there is a path of length d from node 1 to node t, if there exists an edge from node 1 to node t of length d. The second statement is a recursive statement declaring that there is a path from node 1 to node t with length $d_1 + d_2$, if there is a path from node 1 to node s of length d_1 and an edge from s to t of length d_2. The shortest path from node 1 to node t is simply the shortest of all the paths from node 1 to node t, as expressed in the third statement. $\$\textsc{Min}$ is a pre-defined aggregate function in SociaLite.

$$\begin{array}{lcll}
\textsc{Path}(t,d) & :- & \textsc{Edge}(1,t,d). & (1)\\
\textsc{Path}(t,d) & :- & \textsc{Path}(s,d_1), \textsc{Edge}(s,t,d_2), & \\
 & & d = d_1 + d_2. & (2)\\
\textsc{MinPath}(t, \$\textsc{Min}(d)) & :- & \textsc{Path}(t,d). & (3)
\end{array}$$

Fig. 1. Datalog query for computing the single-source shortest paths. The source node has node id 1.

```
Algorithm Dijkstra $(G(V, E : V \times V \times I), s)$
    for each vertex $v \in V$
        $d[v] \leftarrow \infty$
    $d[s] \leftarrow 0$
    $Q \leftarrow s$
    while $Q \neq \emptyset$ do
        $u \leftarrow n \in Q$ with minimum $d[n]$
        $Q \leftarrow Q - u$
        for each $(u, v, l) \in E$
            if $d[v] = \infty$ then
                $Q \leftarrow v$
            $d' = d[u] + l$
            if $d' < d[v]$ then
                $d[v] \leftarrow l$
```

Fig. 2. Dijkstra's algorithm in an imperative programming language.

While the program in Figure 1 is succinct, it fails to terminate in the presence of cycles because the path lengths are unbounded. Even if the data contains no cycles, existing Datalog implementations are relatively slow, due to unnecessary computation of sub-optimal distances, as well as inefficient data structures. We ran this shortest-paths algorithm on LogicBlox [20], a state-of-the-art commercial implementation of Datalog. For a randomly generated *acyclic* graph with 100,000 nodes and 1,000,000 edges, the algorithm required 3.4 seconds to terminate on an Intel Xeon processor running at 2.80GHz.

In contrast, imperative programming languages provide users full control over the execution as well as the layout. For example, Dijkstra's algorithm in Figure 2 computes shortest paths in $O(m+n\log n)$ time, where n is the number of nodes and m is the number of edges. For the same acyclic graph used to evaluate LogicBlox, a Java implementation of Dijkstra's algorithm requires less than 0.1 second. The large performance gap with imperative languages makes Datalog not competitive for solving fundamental graph algorithms. More generally, join operations defined over relational databases do not seem to be a good match for graph algorithms. Graphs can be represented efficiently with linked lists, as opposed to relational tables. Additionally, join operations tend to generate many temporary tables that pessimize the locality of a program.

B. Contributions

This paper presents the SociaLite language, as well as the design, implementation, and evaluation of the SociaLite compiler. SociaLite is an extension of Datalog which allows concise expression of graph algorithms, while giving users some degree of control over the data layout and the evaluation order. For example, the SociaLite version of the shortest-paths algorithm, shown in Figure 3, terminates on cyclic graphs and is as efficient as a Java implementation of Dijkstra's algorithm. We shall use this program as a running example throughout this paper; details on this program will be described in subsequent sections. We summarize the contributions of this paper below.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textsc{Edge}\ (\text{int } \mathit{src}{:}0..10000,\ (\text{int } \mathit{sink},\ \text{int } \mathit{len})).}\\
\multicolumn{3}{l}{\textsc{Path}\ (\text{int } \mathit{sink}{:}0..10000,\ \text{int } \mathit{dist}).}\\
\textsc{Path}(t, \$\textsc{Min}(d)) & :- & \textsc{Edge}(1, t, d); \qquad (4)\\
 & :- & \textsc{Path}(s, d_1), \textsc{Edge}(s, t, d_2),\\
 & & d = d_1 + d_2. \qquad (5)
\end{array}$$

Fig. 3. SociaLite program for computing the shortest paths. The source node has node id 1.

Tail-nested tables. We introduce a new representation, tail-nested tables, designed expressly for graphs. Singly nested tables are essentially adjacency lists. Edges from the same node s are represented by a single entry in the top-level table (s, t), where t is a table consisting of all destination nodes. Arbitrary levels of nesting are allowed, but only in the last column of each level. This representation reduces both the memory usage and computation time needed for graph traversals.

Recursive aggregate functions. SociaLite supports recursively-defined aggregate functions. We show that semi-naive evaluation can be applied to recursively-defined aggregate functions, if they are meet operations and that the rest of the rules are monotonic under the partial order induced by the meet operations. In addition, taking advantage of the commutativity of meet operations, we can speed up the convergence of the solution by prioritizing the evaluation.

User-guided execution order. The order in which a graph is traversed can have a dramatic effect on the performance of a graph algorithm. For example, it is useful to visit a directed acyclic graph in topological order, so that a node is visited only after all its predecessors have been visited. SociaLite enables users to hint at an efficient evaluation order, by referencing a sorted column in the database that contains nodes in the order to be visited.

Evaluation of SociaLite. All the optimizations presented in the system have been implemented in a SociaLite compiler. We show that a large collection of popular graph algorithms can be expressed succinctly in SociaLite, including PageRank, hubs and authorities, clustering coefficients, as well as betweenness centrality, one of the most complex and important graph analyses. Our experiments are performed on two real-life data sets, the LiveJournal social network, consisting of 4.8M nodes and 69M edges, and Last.fm, consisting of 1.8M nodes and 6.4M edges. Across the spectrum of graph algorithms, SociaLite programs outperform initial implementations in Java, and are also within 16% of their highly optimized Java counterparts. This demonstrates that users of SociaLite can enjoy the conciseness and ease of programming of a high-level language, with a tolerable degradation in performance.

C. Paper Organization

The rest of this paper is organized as follows. Section 2 describes our layout optimizations. Section 3 explains recursive aggregate functions and how they can be evaluated incrementally. In Section 4, we explain how users can specify a desired evaluation order. We put all the concepts together in Section 5 and evaluate the performance of SociaLite in Section

6. Related work is reviewed in Section 7 and we conclude in Section 8.

II. Data Representation

In traditional Datalog implementations, data is stored in relational tables, which are inefficient when it comes to graph algorithms. In this section, we present the data representation used in SociaLite and demonstrate how graph analysis is supported by fast join operations with this representation.

A. Data as Indices

A simple but highly effective technique used in imperative programming is to number data items sequentially and use the number as an index into an array. We make this representation choice available to a SociaLite programmer by introducing the concept of a data *range*. A range is simply a lower and an upper bound on the values of a field, and the bounds can be run-time constants.

Consider the single-source shortest-paths example in Figure 3. The first two statements in the program declare two relations: EDGE contains the source node, destination node and edge length for all edges in the graph, while PATH contains the length of the shortest path to each node in the graph. All data are represented as integers. The declaration indicates that the relations EDGE and PATH are to be indexed by the *src* and *sink* fields, respectively, both ranging from 0 to 10,000.

Our compiler uses the range as a hint to use the field as an index. Coupled with the notion of tail-nested tables introduced below, the compiler can simply allocate an array with as many entries as the given range, allowing it to be indexed directly by the value of the index.

B. Tail-Nested Tables

In conventional Datalog implementations or relational database systems, data are stored in a two-dimensional table of rows and columns. A column-oriented table stores the values in the same column contiguously, while a row-oriented table stores entire records (rows) one after another. To store information such as edges in a graph, the source nodes of edges must be repeatedly stored as shown in Figure 4.

Graphs in imperative programming are frequently represented as an adjacency list. As shown in Figure 4 (c), an adjacency list can compactly store edges of a graph, or any list of properties associated with a node. Not only does this representation save space, the program is more efficient because a single test suffices to compare the source node of all the edges in the same adjacency list.

We introduce the notion of a *tail-nested table* as a generalization of the adjacency list. The last column of a table may contain pointers to two-dimensional tables, whose last columns can themselves expand into other tail-nested tables. The nesting is indicated by parentheses in the table declarations. For example,

$$\begin{aligned}&R(\langle\text{type}\rangle c_1, \ldots, \langle\text{type}\rangle c_{n_1-1},\\ &\quad(\langle\text{type}\rangle c_{n_1,1}, \ldots, \langle\text{type}\rangle c_{n_1,n_2-1},\\ &\quad\ (\langle\text{type}\rangle c_{n_1,n_2,1}, \ldots, \langle\text{type}\rangle c_{n_1,n_2,n_3})))\end{aligned}$$

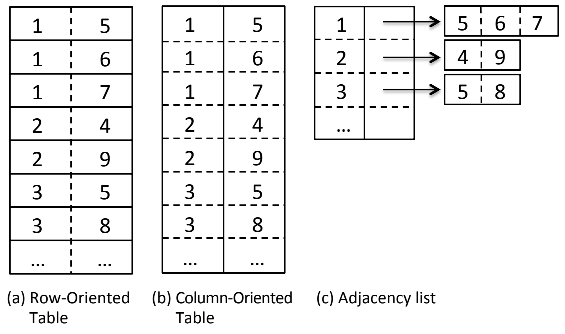

Fig. 4. Storage with different structures: the dotted lines indicate logical structure (columns and rows), and the normal lines indicate physical structure where the elements are stored together, contiguously.

declares a relation R with 3 nested tables where n_1, n_2, n_3 are the number of columns in each level.

The EDGE array in Figure 3 is declared as a tail-nested table, with the last column being a table containing two columns. Thus, EDGE is represented by a table of 10,001 rows, indexed by *src*. It has only one column containing pointers to a two-dimensional array; each row of the array stores a *sink* node and the length of the edge from source to sink. Note that the PATH table is not nested. The second column, *dist*, is applied to the \$MIN aggregate function, so it can only have one value. Therefore, the compiler can just dedicate one entry to one value of the sink node, and use the *sink* node as an index into the array.

C. Join operations

Let us now discuss how we accommodate tail-nested tables in nested loop joins and hashed loop joins. *Nested loop joins* are implemented by nesting the iteration of columns being joined. To iterate down the column of a tail-nested table, we simply nest the iterations of the columns' parent tables, from the outermost to the innermost tables. Observe that if the column being joined is not in the leaf tail-nested table, then each element visited may correspond to many more entries. Therein lies the advantage of this scheme.

In *hashed loop joins*, values of a column are hashed, so that we can directly look up the entries containing a value of interest. To support hashed loop joins for columns in nested tables, they each include a pointer back to its parent table and the record index for each nested table. In this way, from an entry in the column of a nested table, we can easily locate the record in the parent table(s) holding the record.

III. Recursive Aggregate Functions

As illustrated by the shortest-paths algorithm example in Section I, it is important that Datalog be extended with the capability to quickly eliminate unnecessary tuples so that faster convergence is achieved. To that end, SociaLite supports recursive aggregate functions, which help to express many graph analyses on social networks.

A. Syntax and Semantics

In SociaLite, aggregate functions are expressed as an argument in a head predicate. Each rule can have multiple bodies, and the aggregate function is applied to all the terms matching on the right. The aggregate function is recursive if the head predicate is used as a body predicate as well.

For example, the shortest-paths algorithm shown in Figure 3 is specified with the help of a recursive $MIN aggregate function. Each evaluation of the rule returns the minimum of all the distances d for a particular destination node t. The distances d computed in the rule body are grouped by the node t, then the $MIN aggregate function is applied to find the minimum distance. In this example, rule 4 states the base case where the distances of the neighbor nodes of the source node are simply the lengths of the edges. Rule 5 recursively seeks the minimum of distances to all nodes by adding the length of an edge to the minimum paths already found.

This example demonstrates how recursive aggregate functions improves the ability of SociaLite to express graph algorithms. *Without* recursive aggregation, the program in Figure 1 first generates all the possible paths before finding the minimum. It does not terminate in the presence of cycles as the path lengths are unbounded. *With* recursive aggregation, the SociaLite program specifies that only the minimum paths are of interest, thus eliminating cycles from consideration.

The operational semantics of a SociaLite program is defined similarly as Datalog: all rules are to be repeatedly evaluated until convergence is achieved. Application of rule

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

yields

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

Only one argument of a head predicate can be an aggregate function; we refer to all the other arguments in the predicate as *qualifying parameters* for the aggregate function.

B. Greatest Fixed-Point Semantics

Definition 1: An operation is a *meet* operation if it is idempotent, commutative, and associative. A meet operation defines a semi-lattice; it induces a *partial order* $\sqsubseteq$ over a domain, such that the result of the operation for any two elements is the greatest lower bound of the elements with respect to $\sqsubseteq$.

For example, minimum and maximum are meet operations; the induced partial orders are $\le$ and $\ge$, respectively. Summation, in contrast, is not a meet operation since it is not idempotent.

Theorem 1: Given a SociaLite program with an aggregate function, let g be the function that applies the aggregate function to each set of qualifying parameters, and let f represent the rest of the rules. If g is a meet operation, inducing a partial order $\sqsubseteq$, and f is monotone with respect to $\sqsubseteq$, then there exists a unique greatest fixed point R^* such that

1) $R^* = h(R^*) = (g \circ f)(R^*)$
2) $R \sqsubseteq R^*$ for all R such that $R = h(R)$.

Furthermore, iterative evaluation of SociaLite rules will yield the greatest fixed-point solution R^* if it converges.

Proof: If f is monotone under g, then h is also monotone under g, which implies the existence of a unique greatest fixed-point solution [34]. Let h^i denote i applications of h. It follows from the monotonicity of h that $h^i(\emptyset) \sqsubseteq h^{i-1}(\emptyset)$ under g. ($\emptyset$ is the top element in the semi-lattice defined by g, and is thus greater than any other elements.)

If the meet-semilattice defined by g is finite, then there must be a finite k such that

$$\emptyset \sqsupseteq h(\emptyset) \sqsupseteq h^2(\emptyset) \sqsupseteq \ldots \sqsupseteq h^k(\emptyset) = h^{k+1}(\emptyset)$$

$h^k(\emptyset)$ is an inductive fixed-point solution. Using mathematical induction, we can show that the inductive fixed-point is greater than any other fixed-point under g as long as f is monotone. Therefore, the inductive fixed-point $h^k(\emptyset)$ from iterative evaluation is the greatest fixed-point solution. ■

Note that the monotonicity of f under g is required for the existence of a unique greatest fixed-point solution. If f is not monotone, the inductive fixed-point might be different from the greatest fixed-point. Since the iterative evaluation may reach a solution that is lower than the greatest fixed-point solution, it may converge to a suboptimal solution.

C. Semi-Naive Evaluation

Semi-naive evaluation is an optimization critical to the efficient execution of recursive Datalog rules. It avoids redundant computation by joining only subgoals in the body of each rule with at least one new answer produced in the previous iteration. The final result is the union of all the results obtained in the iterative evaluation. Semi-naive evaluation can be extended to recursive aggregate functions if they are meet operations.

Algorithm 1: Semi-naive evaluation of recursive aggregate functions.

Input: A SociaLite program $h = g \circ f$, where $g : R \to R$ applies an aggregate function to each set of qualifying parameters and $f : R \to R$ represents the rest of the rules. g is a meet operation and f is monotone with respect to the partial order induced by g.

Output: Returns the greatest fixed-point solution for the SociaLite program h.

Method:

```
$R_0 \leftarrow \emptyset, \Delta_0 \leftarrow \emptyset$
$i \leftarrow 0$
do
    $i \leftarrow i + 1$
    $R_i \leftarrow g\,(f(\Delta_{i-1}) \cup R_{i-1})$
    $\Delta_i \leftarrow R_i - R_{i-1}$
while $\Delta_i \neq \emptyset$
return $R_i$
```
■

As an example, the shortest-paths algorithm in Figure 3 can be expressed as $h = g \circ f$, where

$$f(R) = \{\langle t, d\rangle | \textsc{Edge}(1, t, d) \vee (\langle s, d_1\rangle \in R \wedge \textsc{Edge}(s, t, d_2) \wedge d = d_1 + d_2)\}$$

computes the new path lengths by adding one more edge to the minimum paths found so far.

$$g(R) = \{\langle t, \min_{\langle t,d_1\rangle \in R} d_1\rangle | \langle t, d\rangle \in R\}$$

finds the minimum path for each destination node. Since minimum is a meet operation, we can apply Algorithm 1 to this program.

$$\Delta_i = \{\langle t, d\rangle | \langle t, d\rangle \in R_i \wedge (d \neq d_1 | \langle t, d_1\rangle \in R_{i-1})\}$$

represents all the newly found shortest paths. Clearly, there is no value in applying f to paths that were already found in R_{i-1}, thus

$$R_i = g(f(\Delta_{i-1}) \cup R_{i-1}) = g(f(R_{i-1})) = h(R_{i-1})$$

This means that the semi-naive evaluation in Algorithm 1 yields the same result as naive evaluation for the shortest-paths program.

Theorem 2: Algorithm 1 yields the same greatest fixed point as that returned with naive evaluation.

Proof: We use mathematical induction to show that $R_i = h^i(\emptyset)$.
Basis: $R_1 = g(f(\Delta_0) \cup R_0) = g \circ f(\emptyset) = h^1(\emptyset)$
Inductive step: Assuming $R_k = h^k(\emptyset)$ for all $k \leq i$,

$$\begin{aligned} R_{i+1} &= g\left(f\left(\Delta_i\right) \cup R_i\right) \\ &= g\left(f\left(h^i(\emptyset) - h^{i-1}(\emptyset)\right) \cup h^i(\emptyset)\right) \\ &= g\left(f\left(h^i(\emptyset) - h^{i-1}(\emptyset)\right) \cup g \circ f \circ h^{i-1}(\emptyset)\right) \\ &= g\left(f\left(h^i(\emptyset) - h^{i-1}(\emptyset)\right) \cup f \circ h^{i-1}(\emptyset)\right) && (6) \\ &= g \circ f\left(\left(h^i(\emptyset) - h^{i-1}(\emptyset)\right) \cup h^{i-1}(\emptyset)\right) && (7) \\ &= h^{i+1}(\emptyset) \end{aligned}$$

Line 6 is true because g is a meet operation, and Line 7 is true because f is distributive. ■

It is easy to extend the theorem for a program with multiple recursive aggregate functions. We denote a program with n aggregate functions as $h_n \circ h_{n-1} \circ ... \circ h_1$, where $h_i = g_i \circ f_i$ for all $i \leq n$, such that g_i and f_i satisfy the same conditions as above. Then, following steps similar to those in the above proof, we can see that semi-naive evaluation gives the same results as naive evaluation for the program with n recursive aggregations.

D. Optimizations

Taking advantage of the high-level semantics of SociaLite, we have developed several optimizations for evaluating recursive aggregate functions, as described below.

Prioritized evaluation. For recursive aggregate functions that are meet operations, we can speed up convergence by taking advantage of commutativity. We store new results from the evaluation of aggregate functions in a priority queue, so that the lowest values in the semi-lattices are processed first. This optimization when applied to the shortest-paths program in Figure 3 yields Dijkstra's shortest-paths algorithm.

Distributing meet operations. When the aggregate function is a meet operation, we can take advantage of distributivity to prune out redundant tuples to reduce the execution overhead. Given a rule of the following pattern:

$$\textsc{Bar}(a, \$\textsc{Min}(b)) \quad :- \quad \textsc{Foo}(a, c), \textsc{Bar}(c, b).$$

instead of applying the join operation before finding the minimum for each value of a, we can take advantage of distributivity by finding the minimum for each value of a as each join is performed. This reduces both memory usage and execution time. Especially in the case where BAR is a nested table, the code generated will simply compare the c values once and return the minimum of b in the nested table.

Pipelining. This optimization improves data locality by interleaving rule evaluation, instead of evaluating one statement in its entirety before the next. If rule R_2 depends on rule R_1, pipelining applies R_2 to the new intermediate results obtained from R_1, without waiting for all the results of R_1 to finish. While pipelining is not specific to aggregate functions, it is particularly useful for recursive and distributive aggregate functions whose bodies have multiple parts. This enables prioritization across statements, which can translate to significant improvement.

IV. Ordering

SociaLite lets users control the order in which the graphs are traversed by declaring sorted data columns and including these columns in Socialite rules as a hint to the execution order.

A. Ordering Specification

The SociaLite programmer can declare that a column in a table is to be sorted by writing

$$R(\langle\text{type}\rangle f_1, \langle\text{type}\rangle f_2, \ldots)\ \text{orderby} f_i[\text{asc}|\text{desc}], \ldots$$

This syntax is familiar to programmers well versed in SQL. Note that in the case where the declared column belongs to a nested table, the scope of the ordering is confined within that nested table.

B. Evaluation Ordering

When a sorted column is included in a Datalog rule, it indicates to the compiler that the rows are to be evaluated in the order specified. Suppose we wish to count the number of shortest paths leading to any node from a single source, a step in the important problem of finding betweenness centrality [11]:

$$\begin{aligned} \textsc{PathCount}(n, \$\textsc{Sum}(c)) \quad &:- \quad \textsc{Source}(n), c = 1; \\ &:- \quad \text{SP}(n, d, p), \textsc{PathCount}(p, c). \end{aligned}$$

The predicate $\text{SP}(n, d, p)$ is true if the shortest path from the source node to node n has length d and the immediate predecessor on the shortest path is node p. $\textsc{PathCount}(n, c)$ is true if c shortest paths reach node n. The base case is that

$$
\begin{array}{lll}
\textsc{Dists}(\text{int } d) : \text{orderby } d & & \\
\textsc{Dists}(d) & :- & \textsc{SP}(\_, d, \_). \\
\textsc{PathCount}(n, \$\textsc{Sum}(c)) & :- & \textsc{Source}(n), c = 1; \\
 & :- & \textsc{Dists}(d), \textsc{SP}(n, d, p), \\
 & & \textsc{PathCount}(p, c).
\end{array}
$$

Fig. 5. Ordering of evaluation in SociaLite.

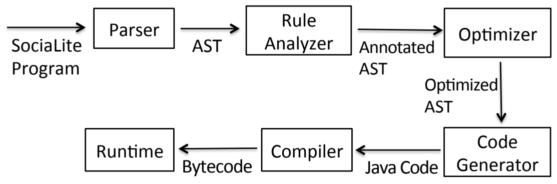

Fig. 6. SociaLite system overview

the path count of the source node is 1; the path count of a node n is simply the sum of the path counts of all its predecessors along the shortest paths leading to n.

Notice that the second rule is recursively dependent on PathCount. Since \$Sum in the rule head is not a meet operation (because it is not idempotent), the recursion here indicates that whenever the PathCount changes for a node, we have to reevaluate the PathCount for all the successors along the shortest paths. If the evaluation is ordered such that many reevaluations are required, it may incur a large performance penalty.

We note that the shortest paths from a single source to all nodes form an acyclic graph. We can compute the path count just once per node if we order the summations such that each node is visited in the order of its distance from the source node. We can accomplish this by including a sorted column with the right ordering in the SociaLite rule, as shown in Figure 5.

Notice that the correctness of the SociaLite rule is independent of the execution order. The user provides a hint regarding the desired execution order, but the compiler is free to ignore the desired order if it sees fit. For example, if a SociaLite program is to be executed on a parallel machine, then it may be desirable to relax a request for sequential execution ordering.

C. Condition Folding

Our compiler takes advantage of the sorted columns to speed up computations predicated on the values of the data. Consider a statement such as:

$$
\begin{array}{lll}
\textsc{Bar}(\text{int } a, \text{int } b) : \text{sortedby } b & & \\
\textsc{Foo}(a, b) & :- & \textsc{Bar}(a, b), b > 10.
\end{array}
$$

We can use binary search to find the smallest value of b that is greater than 10, and return the rest of the tuples with no further comparisons.

V. Putting It All Together

Having presented the high-level concepts in this paper, we now combine everything together and describe the prototype SociaLite compiler that we have developed.

A. User-Specified Functions

Users can supply natively implemented functions and use them in SociaLite rules. A Java function F with n arguments can be invoked in SociaLite with $\$F(a_1, \ldots, a_n)$, which can return one or more results.

SociaLite has a number of pre-defined aggregate functions such as \$Sum, \$Min, and \$Max. We also allow users to define their own aggregate functions in Java. Users can supply a custom aggregate function as a Java class, where the Java class has the aggregate function as its class name. Users can indicate that a given aggregate function is a meet operation, by subclassing from the pre-defined *MeetOp* class, instead of the more general *AggregateOp* class.

An aggregate class has:

- an *identity*() method, which returns the initial value of the accumulated value.
- an *invoke*(*accum*,*v*) method, which returns the result of aggregating argument *v* into the running *accum* argument.

For example, the union aggregate function is defined as:

```
class Union: MeetOp {
   identity(): return {};
   invoke(set accum; elem v): return accum+{v};
}
```

Our compiler does not check if the user has correctly defined a meet operation. In addition, we assume that if a set of recursively defined SociaLite rules have two or more aggregated functions declared to be meet operations, it is safe to apply semi-naive evaluation to those aggregate functions.

B. System Overview

The SociaLite compiler accepts a SociaLite program, together with additional Java functions, and translates it into Java source code. The generated code is then compiled by a regular Java compiler into byte code, which is executed with the SociaLite runtime system, as shown in Figure 6. The SociaLite compiler parses the code into an abstract syntax tree (AST), performs syntactic and semantic analysis, optimizes it, and generates code.

The optimizer analyzes the dependency in the program and evaluates the strongly connected components in topological order. The compiler implements all the optimizations described in the previous sections: data layout and optimizations of tail-nested tables, prioritized evaluation of recursive aggregate functions that are meet operations, distribution of meet operations, and pipelining and ordering the execution as hinted by sorted columns in the rules.

VI. Experiments

In this section, we present an evaluation of our SociaLite compiler. We start by comparing SociaLite to other Datalog engines, using the shortest-paths algorithm, and establish that our baseline implementation is competitive. We then evaluate

Programs	Exec Time(sec)
Overlog	24.9
IRIS	12.0
LogicBlox	3.4
SociaLite	2.6
SociaLite (with layout opt)	1.2
SociaLite (plus recursive min)	0.1
Java (Dijkstra's algorithm)	0.1

Fig. 7. Comparing the execution time of the shortest-paths program on representative Datalog engines.

the compiler with seven core graph analysis routines and a complete algorithm for computing betweenness centrality.

A. Comparison of Datalog Engines

To evaluate how SociaLite compares with state-of-the-art Datalog engines, we experimented with three representative systems: Overlog [8], IRIS [14], and LogicBlox [20]. Overlog is a research prototype designed to explore the use of declarative specification in networks, IRIS is an open-source Datalog engine, and LogicBlox is a commercial system.

None of the other Datalog engines support recursive aggregate functions. We added *nonrecursive* aggregate functions, which are supported by Overlog and LogicBlox, to IRIS in a straightforward manner for the sake of comparison. Without recursive aggregation, our choice of a graph algorithm benchmark was limited. To approximate graph analyses as closely as possible, we selected the shortest-paths program in Figure 1 as the benchmark and ran it on an acyclic graph, since it would not terminate otherwise. Note that the LogicBlox Datalog engine warns users that the program may not terminate. Since real-world graphs often contain cycles, a randomly generated *acyclic* graph with 100,000 nodes and 1,000,000 edges was used as input to all the programs. We authored the programs for Overlog and IRIS ourselves; the program for LogicBlox was written with the help of a LogicBlox expert.

We ran the shortest-paths algorithm on a machine with an Intel Xeon processor running at 2.80GHz. Figure 7 compares the execution times of all the four Datalog engines, including SociaLite. LogicBlox ran in 3.4 seconds, which is significantly faster than Overlog and IRIS. In comparison, SociaLite executed in 2.6 seconds, showing that our baseline system is competitive. With the data layout optimizations described in Section II, the program ran in 1.2 seconds. Had we written the SociaLite program using recursive aggregate functions, as shown in Figure 3, the performance achieved with all the optimizations described in this paper would be 0.1 seconds, which is similar to the performance of Dijkstra's algorithm in Java.

B. Graph Algorithms

Our experimentation with different graph algorithms began with a survey of the literature on social network analyses. Common graph algorithms include computing the importance of vertices, community detection, link prediction, and other general graph metrics [11], [18], [26], [28]. We selected seven representative graph analysis routines, three of which operate on directed graphs:

Shortest Paths: Find shortest paths from a source node to all other nodes in the graph. This is fundamental to many other algorithms, such as link prediction [18] and betweenness centrality [11], which itself is useful for computing the importance of vertices and detecting communities.

PageRank: PageRank [6] is a link analysis algorithm (used for web page ranking) which computes the importance of nodes in a graph. In general, a node is considered important if other important nodes point to it. This algorithm is used ubiquitously in information retrieval, data mining, and computational social science.

Hubs and Authorities: Hyperlink-Induced Topic Search (HITS) [16] is another link analysis algorithm (and a precursor to PageRank) that can be used to compute the importance of nodes in a graph. Two scores, a hub score and an authority score, are assigned to each node. A node is a good hub if it points to a large number of authoritative nodes; a node is a good authority if it is pointed to by a large number of good hubs.

The rest of the benchmarks operate on undirected graphs. Note that an undirected edge is typically represented by a pair of unidirectional edges.

Mutual Neighbors: Find all common neighbors of a pair of nodes. The number of common neighbors between two nodes is an important metric often used for link prediction [18].

Connected Components: Find all connected components in a graph. A connected component is a subgraph in which every pair of nodes is connected by at least one path, and no node in the component is connected to any node outside the component. This is used in many graph analysis routines and fields such as computer vision and computational biology.

Triangles: Find all triangles (i.e., cliques of size three) in the graph. Triangles are used in many graph algorithms: they can define similarity between two graphs [28] or similar sub-structures in a graph [29]. They are also used in the clique percolation algorithm for detecting communities [26].

Clustering Coefficients: We compute the local clustering coefficient of each node, as well as the network average clustering coefficient. In general, the local clustering coefficient is a measure of how well a node's neighbors are connected with each other. The network average clustering coefficient is the average of the local clustering coefficients of all vertices.

C. SociaLite Programs

All the benchmarks in this study can be succinctly expressed in SociaLite. To give readers a flavor of what SociaLite programs look like, we show several representative programs in Figure 8. Whereas these SociaLite programs range from

PageRank (Iteration $i+1$)

```
int N = 4847571. // # of nodes in LiveJounal data
EDGE      (int src: 0..N, (int sink)).
EDGECOUNT (int src: 0..N, int cnt).
NODES     (int n: 0..N).
RANK      (int iter: 0..10, (int node: 0..N, int rank)).
RANK(i + 1, n, $SUM(r))  :-  NODES(n), r = 0.15/N;
                         :-  RANK(i, p, r1), EDGE(p, n),
                             EDGECOUNT(p, cnt),
                             cnt > 0, r = 0.85 × r1/cnt.
```

Connected Components

```
int N = 1768195. // # of nodes in Last.fm data
EDGE      (int src: 0..N, (int sink)).
NODES     (int n: 0..N).
COMP      (int n: 0..N, int root).
COMPIDS   (int id).
COMPCOUNT (int cnt).
COMP(n, $MIN(i))        :-  NODES(n), i = n;
                        :-  COMP(p, i), EDGE(p, n).
COMPIDS(id)             :-  COMP(_, id).
COMPCOUNT($SUM(1))      :-  COMPIDS(id).
```

Triangles

```
int N = 1768195. // # of nodes in Last.fm data
EDGE     (int src: 0..N, (int sink)) orderby sink.
TRIANGLE (int x, int y, int z).
TOTAL    (int cnt).
TRIANGLE(x, y, z)  :-  EDGE(x, y), x < y,
                       EDGE(y, z), y < z, EDGE(x, z).
TOTAL($SUM(1))     :-  TRIANGLE(x, y, z).
```

Fig. 8. Sample SociaLite programs.

	Hand-optimized Java	SociaLite
Shortest Paths	161	4
PageRank	92	8
Hubs and Authorities	104	17
Mutual Neighbors	77	6
Connected Components	103	9
Triangles	83	6
Clustering Coefficients	84	10
Total	704	60

Fig. 9. Number of non-commented lines of code for optimized Java programs and their equivalent SociaLite programs.

4 to 17 lines, with a total of 60 lines; Java programs for these algorithms with comparable performance range from 77 to 161 lines of code, with a total of 704 lines (Figure 9). SociaLite programs are an order of magnitude more succinct than Java programs and are correspondingly easier to write. (More details on these Java programs will be presented in Section VI-F.)

PageRank. Unlike most other graph algorithms that seek a fixed-point solution, PageRank is an iterative algorithm which runs until a convergence threshold is achieved. Shown in Figure 8 is the code for iteration $i+1$. Let $r = \text{RANK}(i,n)$ be the rank of node n in iteration i, expressed as $\text{RANK}(i,n,r)$ in the SociaLite program. In each step, the new PageRank is computed with the following formula:

$$\text{RANK}(i+1,n) = \frac{1-d}{N} + d \sum_{p|\text{EDGE}(p,n)} \frac{\text{RANK}(i,p)}{|\text{EDGE}(p,n)|}$$

where N is the number of nodes in the graph, and d is a parameter called the damping factor, which is typically set to 0.85 [6]. The SociaLite program expresses the formula directly and simply in two rules; the first computes the constant term, and the latter adds the contributions from all the predecessor nodes. The \$SUM aggregate function is computed once for each node; since there is no recursion within each step of the iteration, the fact that \$SUM is not a meet operation is inconsequential.

Connected Components. The connected component ID for each node is simply the minimum of all the node IDs that the node is connected to. The algorithm is expressed simply and directly in four rules. The first rule initializes the component of each node to its own ID. The second rule recursively sets the component ID to the minimum of the component IDs it is connected to directly. Since \$MIN is a meet operation, semi-naive evaluation can be applied. Because of the priority queue used to keep track of the component IDs, the lowest values propagate quickly to the neighboring nodes. The third and fourth simply collect up all the unique component IDs and count them.

Triangles. Finding triangles can be easily described in SociaLite. We specify the condition for having a triangle in the first rule. To avoid counting the same triangle multiple times, the edges of the triangles are sorted from smallest to largest. The second rule simply counts up all the triangles.

Discussion. In imperative programming, the programmer lays out all the data structures and controls the order in which every piece of data is accessed. SociaLite programmers simply describe the computations declaratively as recursive SociaLite rules; they have some control over the layout and execution order by declaring how the data are to be indexed, ordered and nested. The choice is limited and the decisions are often obvious.

For example, we see from the sample SociaLite programs that indexed arrays are used to represent properties of nodes (such as source nodes in graph edges and the iterations for PageRank). Relations expected to have common columns, such as graph edges, have nested structures. Also, for the Triangles program, due to the comparison in the rules, it is useful to sort the *sink* field, so that a binary search can be used to quickly determine the range of *sink* values that will satisfy the predicate. Note that because the *sink* field is represented by nested tables, sorting is applied to the column of each table, which is exactly what we need for this algorithm.

D. Overall Performance

We used two real-world graphs for our experiments, since the first three benchmarks need a directed graph, and the rest need an undirected graph. Our first graph was extracted from

SociaLite Programs	Unoptimized (row)	Unoptimized (column)	Optimized	Speedup (over column)
Shortest Paths	37.9	35.2	6.6	5.3
PageRank	55.4	24.1	19.2	1.3
Hubs and Authorities	114.5	93.5	30.9	3.0
Mutual Neighbors	7.7	5.1	1.5	3.4
Connected Components	25.9	18.7	1.3	14.4
Triangles	158.1	106.1	4.8	22.1
Clustering Coefficients	353.7	245.8	15.4	15.9

Fig. 10. Execution times of unoptimized and optimized SociaLite programs (in seconds).

LiveJournal, a website that enables individuals to keep a journal and read friends' journals [19]. Our LiveJournal dataset is a *directed* graph with 4,847,571 nodes and 68,993,773 edges. Our second graph was extracted from Last.fm. a social music website that connects users with similar musical tastes [17]. The Last.fm dataset is an *undirected* graph consisting of 1,768,195 nodes and 6,428,807 edges.

All applications were executed on the entire data set, except for Mutual Neighbors. Since finding mutual neighbors for all pairs of nodes in the Last.fm graph is expensive, the algorithm was instead evaluated on 2,500,000 randomly selected node pairs. We executed the directed graph algorithms on a machine with an Intel Xeon processor running at 2.80GHz and 32GB memory, and the undirected graph algorithms on a machine with an Intel Core2 processor running at 2.66GHz and 3GB memory.

To evaluate the optimizations proposed in this paper, we compared fully optimized SociaLite programs with non-optimized SociaLite programs. We also compared the performances of two SociaLite variants, one using row-oriented tables and one using column-oriented tables. Note that the row-oriented tables are implemented as arrays of references pointing to tuples in the tables; the column-oriented tables store each column in an array. Our experimental results are shown in Figure 10. The execution times of the unoptimized graph algorithms range from 8 seconds to 354 seconds for the row-oriented implementation. The column-oriented implementation runs up to two times faster. Since the column-oriented implementation is consistently better than the row-oriented counterpart, we use the column version as the baseline of comparison in the rest of our experiments.

The experimental results show that our optimizations deliver a dramatic improvement for all the programs, even over the column-oriented implementation. In particular, all programs finished under 31 seconds. The speedup observed ranged from 1.3 times for simpler algorithms like Pagerank and up to 22.1 times for Triangles. Across all the programs, optimized SociaLite outperformed the column-oriented implementation optimizations by a harmonic mean of 4.0 times.

E. Analysis of the Optimizations

Our next set of experiments attempts to determine the contribution of the different optimizations proposed in this paper.

1) Data Layout Optimizations: Because the data layout interacts with all optimizations, we wished to isolate the effect of data layout optimizations. We obtained two measurements: (1) the performance with all optimizations, and (2) the performance with all but data layout optimizations, and column-oriented relational tables being used. The speedup of the former to the latter measures the effect of data layout in the presence of all the optimizations (Figure 11). We see that the data layout optimization provides a considerable improvement across the board, with the speedup over column orientation ranging from 1.3 to 3.5. The reasons for the speedup are easier to explain when we observe the results of the next experiment.

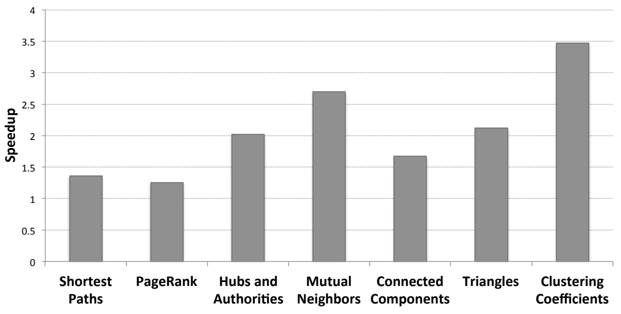

Fig. 11. Speedup due to tail-nested tables and data indexing over column-orientation, with other optimizations applied.

2) Effects of Individual Optimizations: We discovered through experimentation that all optimizations are mostly independent of each other, except for the data layout. This allowed us to understand the contribution of each optimization by simply compounding them one after the other. We ran a series of experiments where we measured the performance of the benchmarks as we added one optimization at a time (Figure 12). The baseline of this experiment was obtained using no optimizations and a column-oriented layout. We then added optimizations in the following order:

1) nested tables and data indexing,
2) prioritization in aggregate functions,
3) pipelining, and
4) conditional folding.

We observe that data layout optimizations on their own have limited improvement, except for Hubs and Authorities and Mutual Neighbors. The reason for the improvement is that the representation of edges is more compact, and we can iterate through the edges of the same source node without testing the source node for each edge. Comparison with Figure 11 shows that data layout optimizations make all the other optimizations more effective.

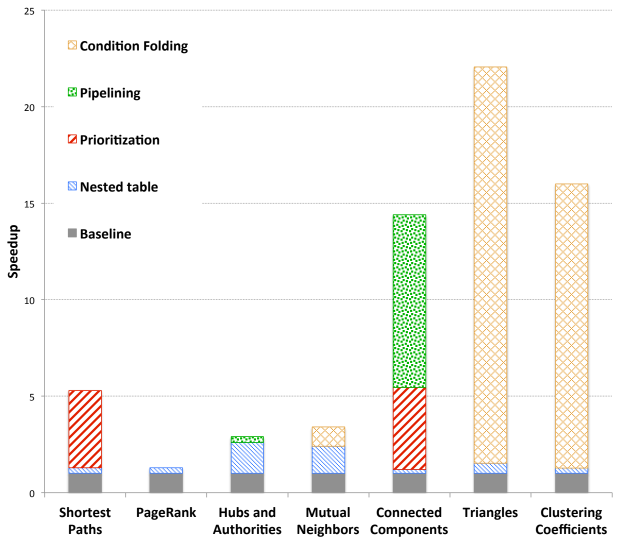

Fig. 12. Speedups from optimizations. Baseline is SociaLite with column orientation.

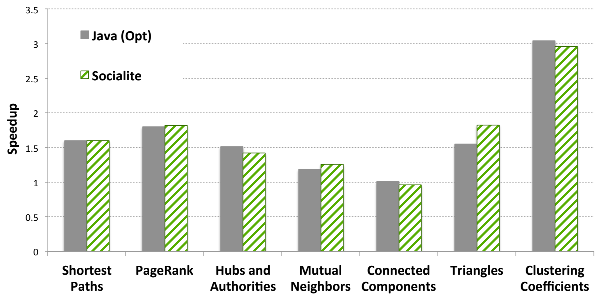

Fig. 13. Performance of optimized SociaLite programs and optimized Java programs relative to initial implementation in Java.

Both Shortest Paths and Connected Components use the $MIN aggregate function and can therefore benefit from the prioritization optimization. For Shortest Paths, the use of a priority queue provides a large speedup, transforming it from a Bellman-Ford algorithm to Dijkstra's algorithm. For Connected Components, the priority queue allows the lowest-ranked component ID to propagate quickly through the connected nodes. In both cases, we observe more than a 5-fold speedup. For Connected Components, pipelining increases the speedup 14-fold. The reason for this tremendous improvement is that the two parts of the recursive definition of Connected Components are pipelined. If the base definition is run to completion before the recursive computation, the priority queue is filled with component ID values that are rendered obsolete almost immediately. Hence for Connected Components, prioritization together with pipelined evaluation provides a large performance improvement. Finally, both Triangles and Clustering Coefficients benefit from condition folding; this optimization returns a significant speedup when coupled with data layout optimizations.

F. Comparison with Java Implementations

To understand the difference between programming in Datalog and imperative programming languages like Java, we asked a colleague who is well versed in both graph analysis and Java to write the same graph analysis routines in Java.

The first implementation of the algorithms in Java is significantly faster than the unoptimized Datalog programs. However, with the optimizations proposed in this paper, our SociaLite programs surpassed the performance of the first implementations in Java. As shown in Figure 13, SociaLite is faster than the first implementation in 6 out of the 7 cases, with speedup ranging from 1.25 to almost 3 times. The harmonic mean in speedup for SociaLite over unoptimized Java for all the programs is 1.52. Note that the original shortest-paths algorithm in Java did not finish within a reasonable amount of time; we improved the implementation by substituting the priority queue in the standard Java library with a custom priority queue. Even with this improvement, it is more than 50% slower than the SociaLite program.

Conceptually, it is always possible to duplicate the performance obtained with SociaLite in a Java program; after all, our compiler translates SociaLite into a Java program. We asked our Java programmer to optimize his Java programs using the concepts in our SociaLite compiler. With considerably more effort, the programmer created optimized Java versions that perform similarly as the SociaLite counterparts, with a harmonic speed up of 1.51 over the unoptimized Java versions.

As shown in Figure 9, the code size of SociaLite programs is much smaller than that of the optimized Java programs. The ratio of the Java to SociaLite code size has a harmonic mean of 11.1. Whereas it took a few minutes to implement the SociaLite programs; it took a few hours for the Java programs. The complexity of the optimizations makes it much harder to get the code to run correctly.

G. Betweenness Centrality

Besides the seven core algorithms, we also experimented with using SociaLite to implement a full application, betweenness centrality [11]. Betweenness centrality is a popular network analysis metric for the importance of a node in a graph. The betweenness centrality of a node v is defined to be $\sum_{s\neq v\neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where $\sigma_{st}(v)$ is the number of shortest paths from s to t passing through v and σ_{st} is the total number of shortest paths from s to t.

We implemented Brandes's algorithm [5], which is the fastest known algorithm for computing betweenness centrality. The algorithm is an iterative process. Each iteration begins with a single-source shortest-paths computation from a source node, followed by path counting (which visits all nodes in increasing order of their distances from the source). It ends with computing the fraction of paths passing through each node, which requires visiting all nodes in the opposite order (e.g., decreasing order of distance from the source). Note that we have already shown how we can control the order of evaluation for finding path counts in Figure 5. Similarly,

Comparison	Java	SociaLite
Development time for Shortest Paths (hours)	10	0.1
Total development time (hours)	12	0.4
Lines of code	258	21
Execution time (hours)	1.8	2.1

Fig. 14. Betweenness Centrality: Java vs SociaLite

we can reverse the order of evaluation by sorting distances in decreasing order.

We used the Last.fm graph for this experiment. It is too expensive to compute centrality exactly for this large graph, as it requires finding the shortest paths from all nodes. Instead, we computed an approximate centrality by running the shortest paths algorithm from 1,000 randomly selected nodes.

To understand how SociaLite compares with an imperative programming language, one author of this paper wrote the code in SociaLite and the other in Java. Figure 14 compares the two implementations. The SociaLite version took about 24 minutes from start to finish. The Java version took about 12 hours, 10 of which were spent in optimizing the shortest-paths algorithm. The program size of the SociaLite version is much smaller than that of the Java version: the SociaLite version uses 21 lines, whereas the Java program requires 258 lines.

The SociaLite implementation is slower than the Java version, but by only 16%. Around 6% of the overhead is due to the overhead of computing ordering hints; the Java version is faster because it determines the ordering as the shortest paths are found. The rest of the slowdown can be attributed to the computation of the shortest paths. Overall, this experiment shows that programming in SociaLite is simpler and faster than coding in Java and the performance overhead is tolerable.

VII. Related Work

Aggregate functions in Datalog. Various attempts have been made in the past to allow incremental analysis of aggregate functions in Datalog [12], [15], [33]. Ganguly et al. showed how a non-recursive minimum or maximum function can be rewritten with a set of recursive rules involving negation, and proved that incremental analysis will yield the same result [12].

Ross and Sagiv proposed a language semantics that, like ours, allows aggregate functions to be defined recursively [30]. They require that aggregate functions be monotonic, that is, adding more elements to the multi-set being operated upon can only increase the value of the function. For example, both minimum and summation are monotonic aggregate functions. As we have noted in Section III, we cannot simply add incremental values to a partial sum because summation is not idempotent and this may double count some values. To address this problem, they have an *additional* requirement that each cost argument (variable to be aggregated) must be *functionally dependent* on the rest of the tuple. This restriction means that there cannot be two tuples that differ only in the cost argument. With this restriction, removing all duplicates before applying the aggregate function will eliminate the double-counting problem. Unfortunately, such a formulation is too restrictive to be useful, since the point of recursion is often to iteratively refine the value of the variables in a program. For example, our shortest-paths algorithm shown in Figure 3 would fall outside their formulation. Note that any aggregate function satisfying their assumptions is also a meet operation. Thus, our formulation is a strict generalization of theirs.

Other Datalog research. Recently Datalog research has been revived in many domains including security [32], programming analysis [35], and network/distributed systems [1], [21]. Datalog is used in the domain of network and distributed systems to implement, for example, network protocols like distributed consensus. Datalog engines for those domains are extended with features for network programming. Dedalus, for example, has incorporated the notion of time as a language primitive, which helps reasoning with distributed states [2].

In contrast, SociaLite has different goals. It aims to make graph analysis easy and efficient. The extensions of SociaLite (tail-nested tables, recursive aggregate functions, and execution ordering) are designed and implemented to help programmers write efficient analysis programs easily.

Data layout. Various projects in the past have explored nested data structures. NESL is a data-parallel programming language with nested data structures [4]. Nested data structures are also used in object-oriented databases [13]. More recently, nested structures have been adopted in Pig Latin, a high-level language that allows users to supply an imperative program that is similar to a SQL query execution plan [25]. The language then translates the plan into map-reduce operations. In contrast, nested tables in SociaLite are strictly layout hints. The SociaLite rules are oblivious to the nesting in the representation. Users can treat elements in a nested table just like data in any other columns.

Graph analysis. A number of query languages have been proposed for graph databases, including GraphLog [9], G-Log [27], GOQL [31], and GrDB [23]. These query languages support functionalities that are useful for graph analysis, such as subgraph matching and node traversal. SociaLite is as expressive as, if not more, than these query languages, with its recursive aggregate functions and user-defined functions.

In terms of distributed frameworks for graph analysis, the popular MapReduce model does not support graph analysis very well [36], so a number of languages have been proposed to simplify the processing of large-scale graphs in parallel. HaLoop provides programming support to iterate map-reduce operations until they converge [7]. Pregel programs consist of a sequence of iterations, where every vertex in a graph can receive messages from a previous iteration, modify its state, and send messages to other vertices [22]. Parallelization of SociaLite while promising, due to its high-level language semantics, is outside the scope of this paper.

VIII. Conclusion

Database languages are powerful as they enable non-expert programmers to formulate queries quickly to extract value out of the vast amount of information stored in databases. With the rise of social networks, we have huge databases that require graph analysis. Analysis of these large databases is not readily addressed by standard database languages like SQL. Datalog, with its support for recursion, is a better match. However, current implementations of Datalog are significantly slower than programs written in conventional languages.

Our proposed language, SociaLite, is based on Datalog and thus can succinctly express a variety of graph algorithms in just a few lines of code. SociaLite supports recursive aggregate functions, which greatly improve the language's expressiveness. More importantly, the convenience of our high-level query language comes with a relatively small overhead. Semi-naive evaluation and prioritized computation can be applied to recursive aggregate functions that are meet operations. Another important feature of SociaLite is user-specified hints for data layout, which allow the SociaLite compiler to optimize the data structures.

In our evaluation of graph algorithms in SociaLite, we found that the optimizations proposed sped up almost all of the applications by 3 to 22-fold. The average speedups of SociaLite programs over unoptimized SociaLite and the first implementations in Java are 4.0 and 1.5 times, respectively. SociaLite is 11.1 times more succinct on average when compared to Java implementations of comparable performance. The SociaLite implementation of betweenness centrality is slower than the highly optimized Java version by just 16%, but it took 12 hours to write the Java application instead of half an hour.

The most important contribution of SociaLite is that, as a query language, it makes efficient social network queries accessible to users who are not proficient in software engineering.

Acknowledgment

We thank LogicBlox and especially Shan Shan Huang for her assistance with the LogicBlox comparison. We also thank Jeffrey D. Ullman, Stefano Ceri, and Jongsoo Park for useful discussions and feedback on this paper, and Junsang Cho for his help with the comparison with Java programs. 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 and Sony Ericsson, as well as a Samsung Scholarship.

References

[1] 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.

[2] 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.

[3] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (ldl1). In *PODS*, pages 21–37, 1987.

[4] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39:85–97, 1996.

[5] U. Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

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

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

[8] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: Metacompilation for declarative networks. *PVLDB*, 1:1153–1165, 2008.

[9] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in graphlog. In *Hypertext*, pages 269–292, 1989.

[10] S. Elnikety and Y. He. System support for managing large graphs in the cloud. In *Proceedings of the NSF Workshop on Social Networks and Mobility in the Cloud*, 2012.

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

[12] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programming. In *PODS*, pages 154–163, 1991.

[13] R. Hull. A survey of theoretical research on typed complex database objects. In *Databases*, pages 193–261, 1987.

[14] Iris, an open-source datalog engine. http://www.iris-reasoner.org/.

[15] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *ISLP*, pages 387–401, 1991.

[16] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46:604–632, 1999.

[17] Last.fm. http://last.fm/.

[18] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58:1019–1031, 2007.

[19] Livejournal. http://www.livejournal.com/.

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

[21] 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.

[22] 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, 2010.

[23] W. E. Moustafa, G. Namata, A. Deshpande, and L. Getoor. Declarative analysis of noisy information networks. In *ICDE GDM Workshops*, pages 106–111, 2011.

[24] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.

[25] 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.

[26] G. Palla, I. Derenyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814–818, 2005.

[27] J. Paredaens, P. Peelman, and L. Tanca. G-log: A graph-based query language. *IEEE Trans. Knowl. Data Eng.*, 7(3):436–453, 1995.

[28] J. W. Raymond, E. J. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45:631–644, 2002.

[29] N. Rhodes, P. W. 0002, A. Calvet, J. B. D. Jr., and C. Humblet. Clip: Similarity searching of 3D databases using clique detection. *Journal of Chemical Information and Computer Sciences*, 43(2):443–448, 2003.

[30] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.

[31] L. Sheng, Z. M. Özsoyoglu, and G. Özsoyoglu. A graph query language and its query processing. In *ICDE*, pages 572–581, 1999.

[32] M. Sherr, A. Mao, W. R. Marczak, W. Zhou, B. T. Loo, and M. Blaze. A3: An Extensible Platform for Application-Aware Anonymity. In *NDSS*, pages 247–266, 2010.

[33] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB*, pages 501–511, 1991.

[34] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

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

[36] C. Yu. Beyond simple parallelism: Challenges for scalable complex analysis over social data. In *Proceedings of the NSF Workshop on Social Networks and Mobility in the Cloud*, 2012.