SociaLite: Datalog Extensions for Efficient Social Network Analysis

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

ABSTRACT

Graph analysis is becoming increasingly important with the rise of social networks. Because SQL lacks the expressiveness and performance needed for graph algorithms, lower-level, general-purpose languages are often used instead, but can be difficult for users who are not proficient in software engineering.

For greater ease of use and efficiency, we propose SociaLite, a high-level graph query language based on Datalog. 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.

Our experiment demonstrates that SociaLite queries are succinct, requiring just one tenth the lines of code needed with Java. The optimizations proposed in this paper speed up almost all the algorithms by at least a factor of three and as much as 22-fold. SociaLite’s overhead is less than 16% when we compare its performance to that of highly optimized Java programs. This overhead is tolerable given its advantage in programming simplicity.

1. 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 [11, 39].

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, Datalog implementations are too slow to be competitive 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, but the 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. The extensions include data layout declarations, hints of evaluation order, and recursive aggregate functions.

1.1 Performance of Datalog Programs

Consider the example of computing shortest paths from a source node to all other nodes in a graph. Utilizing a previously proposed extension of aggregate functions [4, 27], 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. SociaLite syntax is used in the third statement for the aggregate function $\text{Min}$, which is indicated by the preceding dollar sign ($\$\$).

While the program is succinct, such a program fails to terminate in the presence of cycles because of unbounded path lengths. Even if the data contains no cycles, existing Datalog implementations are relatively slow, due to unnecessary computation for sub-optimal distances, as well as inefficient data structures. We ran this shortest-paths algorithm on LogicBlox [23], a state-of-the-art commercial implementa-
ation 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.

\[
\text{Path}(t, d) : = \text{Edge}(1, t, d). \quad (1)
\]

\[
\text{Path}(t, d) : = \text{Path}(s, d_1), \text{Edge}(s, t, d_2), d = d_1 + d_2. \quad (2)
\]

\[
\text{MinPath}(t, \text{Min}(d)) : = \text{Path}(t, d). \quad (3)
\]

Figure 1: Datalog query for computing single-source shortest path. Source node has node id 1.

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 typical shortest-paths algorithm written in Java requires less than 0.1 seconds of runtime when the Dijkstra’s algorithm is applied. 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.

Algorithm Dijkstra \((G(V, E : V \times V \times I), w, s)\)

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 \(d[v] = \infty\)

\(Q \leftarrow v\)

\(d'[v] \leftarrow d[v] + l\)

if \(d' < d[v]\)

\(d[v] \leftarrow l\)

Figure 2: Dijkstra’s algorithm in an imperative programming language.

1.2 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 the Java’s version of Dijkstra’s algorithm. We use this program as a running example throughout our paper. Further details about this program will be described in subsequent sections.

The contributions of this paper include the following:

- 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 aggregate functions which can be applied to recursively-defined SociaLite rules. We showed that semi-naive evaluation can be applied to recursive aggregate functions that are meet operations. Taken advantage of the commutativity of meet operation, we can speed up the convergence by prioritizing the evaluation.

- User-guided execution order. For many graph algorithms, evaluation order has a dramatic effect on the speed of convergence of a recursion. 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, and clustering coefficients. In addition, we also write a betweenness centrality application, one of the most complex and important graph analyses, in SociaLite. 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.

1.3 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.

\[
\text{Path}(t, \text{Min}(d)) : = \text{Edge}(1, t, d); \quad \text{Path}(s, d_1), \text{Edge}(s, t, d_2), d = d_1 + d_2. \quad (5)
\]

Figure 3: SociaLite Program for Computing Shortest Paths. Source node has node id 1.
introducing the concept of a data representation choice available to a SociaLite programmer by the number as an index into an array. We make this representation.

2.1 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.

Consider the SociaLite program for the single-source shortest path example in Figure 3. The first two statements in the program declare two relations: 

\[ \text{Edge} \] contains the source node, destination node and edge length for all edges in the graph, while 

\[ \text{Path} \] contains the length of the shortest path to each node in the graph. All data is represented as integers. The declaration indicates that the relations \[ \text{Edge} \] and \[ \text{Path} \] are to be indexed by the \[ \text{src} \] and \[ \text{sink} \] fields, respectively, both ranging from 0 to 10,000.

The concept of a range is useful for different purposes, but currently our compiler utilizes it solely as a hint to use the field as an index. Note that the data range can be determined dynamically at run time. 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.

2.2 Tail-Nested Tables

In conventional Datalog implementations or relational database systems, data is 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, a relation with 3 levels of tail-nesting will be declared as follows, where \( n_1, n_2, n_3 \) are the number of columns in each level:

\[ R((\text{type})c_1, \ldots, (\text{type})c_{n_1}, \ldots, (\text{type})c_{n_1, n_2}, \ldots, (\text{type})c_{n_1, n_2, n_3})) \]

For example, the \[ \text{Edge} \] array in Figure 3 is declared as a tail-nested table, with the last column being a table containing two columns. Thus, \[ \text{Edge} \] is represented by a table of 10,001 rows, indexed by \[ \text{src} \]. It has only one column containing pointers to a two-dimensional array; each row of the array stores a \[ \text{sink} \] node and the length of the edge from source to sink. Note that the \[ \text{Path} \] table is not nested. The second column, \[ \text{dist} \], is applied to the \[ \text{MIN} \] 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 \[ \text{sink} \] node as an index into the array.

2.3 Join operations

Let us now discuss how we accommodate tail-nested tables in nested loop joins and hashed loop joins. \[ \text{Nested loop joins} \] 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 \[ \text{hashed loop joins} \], 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, we record for each column, the parent table, as well as 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.

3. RECURSIVE AGGREGATE FUNCTIONS

As illustrated by the shortest-paths algorithm example in Section 1, 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.
3.1 Syntax and Semantics

In SociaLite, aggregate functions are expressed as an argument in a head predicate. For example, in Figure 3, the aggregate function specified is the $\$Min$ 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 source node $i$, then the $\$Min$ aggregate function is applied to find the minimum distance. Each rule can have multiple bodies, and the aggregate function is applied to all the terms matching on the right. 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.

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

\[
P(x_1, ..., x_n, F(z)) : = Q_1(x_1, ..., x_n, z). \quad \ldots \quad : = Q_n(x_1, ..., x_n, z).
\]

yields

\[
\{(x_1, ..., x_n, z) | z = F(z'), \forall 1 \leq k \leq m Q_k(x_1, ..., x_n, z')\}
\]

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.

Recursive aggregate functions greatly improve the ability of SociaLite to express graph algorithms. Let us return to the shortest-paths algorithm in Figure 3 as an example. Without recursive aggregation (Figure 1), we need to generate all the paths before we can find the minimum. The program does not terminate with cycles as the path lengths are unbounded. With recursive aggregation, the SociaLite program only keeps track of the minimum paths, which automatically eliminates cycles from consideration.

3.2 Semi-Naive Evaluation

Semi-naive evaluation is a critical optimization to the efficient execution of recursive Datalog rules. It avoids redundant joins by ensuring that the join of the subgoals in the body of each rule must involve 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.

We can extend semi-naive evaluation to recursive aggregate functions if they are meet operations. A meet operation is idempotent, commutative, and associative. Meet operations induce a partial order such that the result of the operation for any two elements is the greatest lower bound of the elements with respect to this partial order.

Each iteration in a semi-naive evaluation with aggregate functions involves:
1. Join the subgoals containing new results from the last iteration
2. Apply the aggregate function to the new results as well as the answer from the last iteration, for each set of qualifying parameter values.

As long as the aggregate function is a meet operation, semi-naive evaluation computes the same greatest lower bound returned with naive evaluation.

Formally, let $F$ be the given aggregate function, $s_i$ be the set of inputs aggregated in iteration $i$ for a given set of qualifying parameters, and $S_i = F(s_i)$. Suppose $s'_{i+1}$ is the set of new delta values to be aggregated, for the same set of qualifying parameters, by joining the new results from the $i$th step. If $F$ is a meet operation,

\[
s_{i+1} = s_i \cup s'_{i+1} \implies S_{i+1} = F(s_{i+1}) = F(\{S_i\} \cup s'_{i+1})
\]

3.2.1 Example: Minimum

Let us consider the shortest-paths algorithm in Figure 3 as an example. Note that $\$Min$ is a meet operation, inducing a $<$ partial order on the integer domain. Let $D(t, i)$ denote the shortest path for node $t$ in iteration $i$. When Rules 4 and 5 are evaluated in iteration $i+1$ under naive evaluation, we obtain:

\[
D(t, i + 1) = \min\{D(t, i) \cup \{D(s, i) + d(t, d)\}\}.
\]

Suppose a new minimum is found for node $X$. Applying semi-naive evaluation to this rule, we only need to evaluate the right-hand-side where the source node is $X$. The new minimum for node $t$ is simply the minimum of the latest path length for $t$ and the sum of the path length of $X$ and the length of edge $(X, t)$ if one exists.

\[
D(t, i + 1) = \min\{D(t, i) \cup \{D(X, i) + d(X, t)\}\}.
\]

Clearly, the semi-naive evaluation returns the same result as naive evaluation.

3.2.2 Non-Example: Summation

In contrast, summation is not a meet operation, as it is not idempotent. Semi-naive evaluation does not apply to recursive summations. Consider the example in Figure 5 where we count the number of shortest paths leading to any node from a single source, a step in the important problem of finding betweenness centrality [12]. 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 $p$. $\text{PATHCOUNT}(n, c)$ is true if $c$ shortest paths reach node $n$. The base case is that 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 $n$. Here the aggregate sum function is used in the recursive definition of $\text{PATHCOUNT}$.

\[
\text{PATHCOUNT}(n, \$\text{SUM}(c)) : = \text{SOURCE}(n), c = 1;
\]

\[
: = \text{SP}(n, d, p), \text{PATHCOUNT}(p, c).
\]

Figure 5: Recursive aggregate function in $\text{PATHCOUNT}$, a step in betweenness centrality

Whenever the $\text{PATHCOUNT}$ changes for a node, we have to re-evaluate the $\text{PATHCOUNT}$ for all the successors along the shortest paths. Semi-naive evaluation cannot be used. If we simply added the new modified path count to the current path count of a successor, the answer would be wrong as some paths will be doubly counted. We need to re-apply the whole rule by summing up all the path counts for all the predecessors. The key, as we have noted, is that summation is not idempotent, and therefore not a meet operation.

3.3 Combinations of Aggregate Functions
Given aggregate functions that are meet operations, we can model the solution space to the evaluation of a SociaLite program as a meet-semilattice. In the case where there is just one aggregate function in a SociaLite program, the solution space is a cross-product of meet-semilattices, one for each qualifying parameter and the result of the meet operation itself. This model allows us to handle mutually recursive aggregate functions. If a meet-semilattice can be constructed accommodating all the semi-lattices corresponding to each aggregate function, then the solution will decrease monotonically in iterative evaluation according to the partial order imposed by the meet operation. For example, a $\text{MIN}$ function applied to the lengths can be combined with a $\text{MAX}$ function applied to the costs. However, if one rule seeks the minimum and another seeks the maximum of the same variable, we cannot construct a semi-lattice to model the evaluation. As a matter of fact, such a program may oscillate and not terminate, in the same way a set of recursive rules with negation may behave.

### 3.4 Optimizations

The high-level semantics in SociaLite programs enables important optimizations easily. We have developed several optimizations for evaluating recursive aggregate functions: prioritized evaluation, distribution of meet operations, and pipelining.

**Prioritized evaluation.** For recursive aggregate functions that are meet operations, we can speed up its convergence by taking advantage of its 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 the 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:

$$\text{Bar}(a, \text{MIN}(b)) \rightarrow \text{Foo}(a, c), \text{Bar}(c, b).$$

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 we do the join. This reduces both memory usage and execution time. Especially in the case where $\text{Bar}$ 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 interleaves rule evaluation instead of evaluating one statement in its entirety before the next. For example, given a pair of rules, $R_1$ and $R_2$, such that $R_2$ depends on $R_1$, the new intermediate results obtained in computing rule $R_1$ are used immediately to evaluate rule $R_2$, without waiting for all the results of $R_1$ to finish. Pipelining has the effect of improving data locality. 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.

### 4. ORDERING

The control of evaluation order is very important to the performance of a graph analysis routine. For example, depth-first ordering is used in efficient algorithms for finding strongly connected components. A topological sort is used in directed acyclic graphs for computations that depend on a node’s predecessors, such as the path counts example in Figure 5. The importance of graph ordering warrants that SociaLite provides users with some degree of control over it. This is accomplished by enabling users to declare sorted data columns, and permitting them to include these columns in SociaLite rules, as a hint to the execution order.

#### 4.1 Ordering Specification

The syntax we have adopted is borrowed from SQL. The declaration is of the form:

$$R((\text{type})f_1, \ldots, \text{type}f_n) : \text{orderby}f_1[\text{asc|desc}], f_2[\text{asc|desc}], \ldots$$

The scope of the ordering of a column is confined to the column within the enclosing nested table.

#### 4.2 Evaluation Ordering

To start, let us return to the example of path counts in Figure 5. 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 a 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 6.

```sql
DIST(int d) : orderby d

DIST(d)

PATHCOUNT(n, $SUM(c))

Figure 6: Ordering of evaluation in SociaLite.
```

Notice that the correctness of the SociaLite rule is independent of the execution order. The user provides a hint regarding 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.

#### 4.3 Condition Folding

When the column is sorted, we can easily iterate through values within a range. Consider for example a statement such as:

$$\text{Bar}(\text{int } a, \text{int } b) : \text{sortedby } b$$

$$\text{Foo}(a, b) : - \text{Bar}(a, b), b > 10.$$  

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.

### 5. 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.
5.1 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 $\Sigma$, $\min$ 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 the v argument into the running accum argument.

For example, the union aggregate function is defined as

```java
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.

5.2 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 7. 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.

<table>
<thead>
<tr>
<th>Datalog Engine</th>
<th>Exec Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlog</td>
<td>24.9</td>
</tr>
<tr>
<td>IRIS</td>
<td>12.0</td>
</tr>
<tr>
<td>LogicBlox</td>
<td>3.4</td>
</tr>
<tr>
<td>SociaLite</td>
<td>2.6</td>
</tr>
<tr>
<td>SociaLite (with layout opt)</td>
<td>1.2</td>
</tr>
<tr>
<td>SociaLite (plus recursive min)</td>
<td>0.1</td>
</tr>
<tr>
<td>Java (Dijkstra’s algorithm)</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 1: Comparing the Execution Time of Shortest-Paths Program on Representative Datalog Engines

6. EXPERIMENTS

Having presented the system overview of our prototype SociaLite compiler, we now discuss experimental evaluation of our system. The SociaLite compiler is initially evaluated against other Datalog engines, using the shortest-paths algorithm, to establish that our baseline implementation is competitive. We then present the full evaluation of the SociaLite compiler with seven core graph analysis routines, followed by an evaluation of a complete graph algorithm for computing betweenness centrality.

6.1 Comparison of Datalog Engines

To evaluate how SociaLite compares with state-of-the-art Datalog engines, we experimented with three representative systems: Overlog [10], IRIS [15], and LogicBlox [23]. 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 is limited. To approximate graph analysis as closely as possible, we picked 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 the users that the program may not terminate. Since real-life graphs often contain cycles, we use as an input the same randomly generated acyclic graph with 100,000 nodes and 1,000,000 edges. We authored the programs for Overlog and IRIS ourselves, and 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. Table 1 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 executes in 2.6 seconds, showing that our baseline system is competitive. With the data layout optimizations described in Section 2, the program runs 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.

6.2 Graph Algorithms
Our experimentation began with a survey of the literature on graph algorithms in social networks. Common algorithms include computing the importance of vertices, community detection, link prediction and other general graph metrics [12, 21, 32, 33, 31]. We selected seven of the core algorithms as representative of graph analysis routines. Three of these are fundamental algorithms for directed graphs:

**Shortest Paths**: Find shortest paths from a source node to all other nodes in the graph. This is a core algorithm fundamental to many other algorithms, such as link prediction [21] and betweenness centrality [12], which itself is useful for computing the importance of vertices and detecting communities. The longest shortest path also defines the diameter of a graph, which is an important metric.

**PageRank**: PageRank [8] 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. As an algorithm, the importance of PageRank cannot be over-stated; it is now ubiquitous in many research areas, such as information retrieval, data mining, and computational social science.

**Hubs and Authorities**: Hyperlink-Induced Topic Search (HITS) [19] is another link analysis algorithm (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, and it is a good authority if it is pointed to by a large number of good hubs.

We also consider four algorithms for undirected graphs. Note that an undirected edge is typically represented by a pair of unidirectional edges.

**Mutual Neighbors**: Find all nodes which are common neighbors of a pair of nodes. The number of common neighbors between two nodes is an important metric often used for link prediction [21].

**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 a basic algorithm used in many graph analysis routines, as well as other 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 nodes, be over-stated; it is now ubiquitous in many research areas, such as information retrieval, data mining, and computational social science.

**Clustering Coefficients**: We compute the local clustering coefficient of each node, as well as the network average clustering coefficient. The local clustering coefficient of a node \( x \) is defined to be \( \frac{|\{y,z\}|}{k(k-1)} \), where \(|\{y,z\}|\) is the number of pairs of neighbors \( \{y,z\} \) of \( x \) which are directly connected by an edge, while \( k \) is the number of neighbors of \( x \). 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 simply the average of the local clustering coefficients of all vertices.

### 6.3 SociaLite Programs

All these algorithms can be succinctly expressed in 4 to 17 lines of SociaLite; in contrast Java programs of comparable performance requires many more lines of codes, as shown in Table 2. (We compare SociaLite with Java further in Section 6.6.) To give readers a flavor of what SociaLite programs look like, we show several representative programs in Figure 8.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Hand-optimized</th>
<th>SociaLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest Paths</td>
<td>161</td>
<td>4</td>
</tr>
<tr>
<td>PageRank</td>
<td>92</td>
<td>8</td>
</tr>
<tr>
<td>Hubs and Authorities</td>
<td>104</td>
<td>17</td>
</tr>
<tr>
<td>Mutual Neighbors</td>
<td>77</td>
<td>6</td>
</tr>
<tr>
<td>Connected Components</td>
<td>103</td>
<td>9</td>
</tr>
<tr>
<td>Triangles</td>
<td>83</td>
<td>6</td>
</tr>
<tr>
<td>Clustering Coefficients</td>
<td>84</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>704</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 2: Number of non-commented lines of code for optimized Java programs and their equivalent SociaLite programs.

- **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:

\[
\text{Rank}(i + 1, n) = \frac{1 - d}{N} + d \sum_{p \in \text{Edge}(p,n)} \left( \frac{\text{Rank}(i,p)}{\text{Edge}(p,n)} \right),
\]

where \( N \) is the number of nodes in the graph, and \( d \) is a parameter called the damping factor, which is typically set to be 0.85 [8]. The SociaLite program expresses the formula directly and simply in two rules; the first computes the contributions from all the predecessor nodes, and the latter adds the constant term. The \$\text{Sum} \$ aggregate function is computed once for each node; since there is no recursion within each step of the iteration, the fact that \$\text{Sum} \$ is not a meet operation is inconsequential.

- **Connected Components**: In this algorithm, 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. Since \$\text{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 all the unique component IDs and count them.

- **Triangles**: Finding triangles, or cliques of size 3, can be easily described in SociaLite. We simply specify the condition for having a triangle in the first rule and store the...
nodes constituting triangles. 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. The fact that all these graph algorithms can be succinctly specified in SociaLite is no surprise. The primary question of interest is whether these algorithms perform well, which we will discuss next. In imperative programming, the program lays out all the data structures (dealing with references and objects) and controls the order in which every piece of data is accessed. SociaLite programmers need only describe the relationships between the data structures declaratively as recursive SociaLite rules. The programmers also need to decide on the use of data indexing, ordering, and nested tables; the choice is very limited and the decisions are quite 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 Triangles, 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, the sorting occurs within the column of each table, which is exactly what we need for this algorithm.

6.4 Overall Performance

For our experiments with directed graph algorithms (Shortest Paths, PageRank, and Hubs and Authorities), we use a real-world corpus from the LiveJournal social network [22]. LiveJournal enables individuals to keep a journal and read friends’ journals. Our LiveJournal dataset is a directed graph with 4,847,571 nodes and 68,993,773 edges. For undirected graph algorithms, we use an undirected social network from Last.fm [20]. Last.fm is a social music website, connecting users with similar musical tastes. The Last.fm dataset is an undirected graph consisting of 1,768,195 nodes and 6,428,807 edges.

All applications are 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 is instead evaluated on 2,500,000 randomly selected node pairs. We execute 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 compare fully optimized SociaLite programs with non-optimized SociaLite programs. We also compare 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 Table 3. 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 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 finish under 31 seconds. The speed-up observed ranges from 1.3 times for simpler algorithms like Pagerank up to 22.1 times for Triangles.

6.5 Analysis of the Optimizations

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

6.5.1 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: (a) the performance with all optimizations, and (2) the performance with all but data layout optimizations, and column-oriented relational tables are used instead. The speedup of the former to the latter measures the effect of data layout in the presence of all the optimizations (Figure 9). We see that the data layout optimization provides a considerable improvement across the board, with the speed-up over column orientation ranging from 1.3 to 3.5. The reasons for the speed-up are easier to explain when we see the results of the next experiment.

6.5.2 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. The baseline of this experiment was obtained using no optimizations and a column-oriented layout.

<table>
<thead>
<tr>
<th>SociaLite Programs</th>
<th>Unoptimized (row)</th>
<th>Unoptimized (column)</th>
<th>Optimized</th>
<th>Speed-Up (over column)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest Paths</td>
<td>37.9</td>
<td>35.2</td>
<td>6.6</td>
<td>5.3</td>
</tr>
<tr>
<td>PageRank</td>
<td>55.4</td>
<td>24.1</td>
<td>19.2</td>
<td>1.3</td>
</tr>
<tr>
<td>Hubs and Authorities</td>
<td>114.5</td>
<td>96.5</td>
<td>30.9</td>
<td>3.0</td>
</tr>
<tr>
<td>Mutual Neighbors</td>
<td>7.7</td>
<td>5.1</td>
<td>1.5</td>
<td>3.4</td>
</tr>
<tr>
<td>Connected Components</td>
<td>25.9</td>
<td>18.7</td>
<td>1.3</td>
<td>14.4</td>
</tr>
<tr>
<td>Triangles</td>
<td>158.1</td>
<td>106.1</td>
<td>48</td>
<td>22.1</td>
</tr>
<tr>
<td>Clustering Coefficients</td>
<td>353.7</td>
<td>245.8</td>
<td>15.4</td>
<td>15.9</td>
</tr>
</tbody>
</table>

Table 3: Execution times of unoptimized and optimized SociaLite programs (in seconds).
PageRank (Iteration $i + 1$)

```c
int N = 4847571. // # of nodes in LiveJournal data
Edge (int src: 0..N, (int sink)).
EdgeCount (int src: 0..N, int cnt).
Rank (int iter: 0..10, (int node: 0..N, int rank)).
RankTmp (int node: 0..N, int rank).

RankTmp(n, $\Sigma(r)$) = Rank(i, p, r1), Edge(p, n),
                    EdgeCount(p, cnt),
cnt > 0,
r = r1/cnt.

Rank(i + 1, n, r) = RankTmp(n, r1),
r = 0.15/N + 0.85 \times r1.
```

Connected Components

```c
int N = 1768195. // # of nodes in Last.fm data
Edge (int src: 0..N, (int sink)).
Nodes (int node).
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($\Sigma(1)$) = CompIDs(id).
```

Triangles

```c
int N = 1768195. // # of nodes in Last.fm data
Edge (int src: 0..N, (int sink)) order by 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($\Sigma(1)$) = Triangle(x, y, z).
```

The results are presented in Figure 10.

We observe that data layout optimization on its own has 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 9 shows that data layout optimizations make all the other optimizations more effective.

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 gives it a large speed-up, transforming it from a Bellman-Ford algorithm to Dijkstra’s algorithm. For Connected Components, the priority queue also allows the lowest-ranked component ID to propagate quickly through the connected nodes. In both cases, we observe more than a 5-fold speed-up. For Connected Components, pipelining increases the speed-up 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 speed-up when coupled with data layout optimizations.

6.6 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. We then went back to improve the Java programs using the knowledge we gained from our SociaLite implementation. The comparison in performance
is shown in Figure 11. The figure shows that most of the Socialite programs are within 5% overhead of optimized Java programs, and are much faster than the unoptimized Java programs. Note that the original shortest-paths algorithm did not finish within a reasonable amount of time; the performance we reported here already included a custom priority queue developed as an optimization. Still, it is more than 50% slower than both the Socialite program and the optimized Java 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. Our Java programmer put in significant effort to improve his first implementation. For example, the shortest-paths program was first implemented with the priority queue in the standard Java library. The general-purpose priority queue was too inefficient, so the program had to be re-implemented with a custom priority queue. Then, the data structures in the program were optimized for primitive data types. Finally, the program was optimized to use arrays, instead of linked lists, to store neighboring nodes in the graph for better locality. The final version of the Java shortest-paths program has similar performance as the Socialite program.

The Java programs also took much longer to program correctly; the optimizations added significantly to the programming complexity. As shown in Figure 2, the code size of Socialite programs is much smaller than that of the optimized Java programs, 90% smaller in most programs. The development time was approximately proportional to the code size; it took a few minutes to implement the Socialite programs and a few hours for the Java programs.

6.7 Betweenness Centrality

Besides the seven core algorithms, we also experimented with using Socialite to implement a full application, betweenness centrality [12]. 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 $\sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where $\sigma_{st}(v)$ is the number of shortest paths from $s$ to $t$ passing through $v$ and $\sigma_{st}$ is the total number of shortest paths from $s$ to $t$.

We implemented Brandes’ algorithm [7], 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), and finally 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 6. Similarly, we can reverse the order of evaluation by sorting distances in decreasing order.

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

To understand how Socialite compares with an imperative programming language, we performed an experiment where two of the authors of this paper sat in the same room to implement the algorithm, one in Socialite and one in Java. Table 4 compares the two implementations. The optimized shortest-paths algorithm was developed in 10 hours using Java and 5 minutes in Socialite. In addition, over two hours were needed to write the betweenness centrality algorithm, bringing the total to 12 hours. Socialite, on the other hand, took a total of 0.4 hours. 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.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Java</th>
<th>Socialite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development time for Shortest Paths (hours)</td>
<td>10</td>
<td>0.1</td>
</tr>
<tr>
<td>Total development time (hours)</td>
<td>12</td>
<td>0.4</td>
</tr>
<tr>
<td>Lines of code</td>
<td>258</td>
<td>21</td>
</tr>
<tr>
<td>Execution time (hours)</td>
<td>1.8</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Table 4: Betweenness Centrality: Java vs Socialite

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. Although there is a performance overhead, it is tolerable for the programs that we have written.

7. RELATED WORK

Aggregate functions in Datalog. Various attempts have been made in the past to allow incremental analysis of aggregate functions in Datalog [18, 13, 37]. Ganguly et al. showed how a non-recursive minimum/maximum aggregate function can be rewritten recursively with a set of rules involving negation, and proved that incremental analysis will yield the same result [13]. In contrast, our approach lets the users describe the aggregate functions recursively with a multi-part body. Our formulation also applies to all aggregate functions that are meet operators.
Ross and Sagiv proposed a language semantics that, like ours, allows aggregate functions to be defined recursively [34]. 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 3, 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 operator. Thus, our formulation is a strict generalization of theirs.

**Other Datalog Research.** Recently Datalog research has been revived in many domains including security [35, 26], programming analysis [38], and network/distributed systems [24, 2]. Datalog is particularly well employed in the domain of network and distributed systems to implement, for example, network protocols like distributed consensus (Paxos). Datalog engines for those domains are extended with features for network programming. Dedalus [3] for example, has incorporated the notion of time as a language primitive, which helps reasoning with distributed states.

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 easily write efficient analysis programs.

**Data layout.** Various projects in the past have explored nested data structures. NESL is a parallel data-parallel programming language with nested data structures [5]. Nested data structures are also used in object-oriented databases [14]. More recently, nested structures have also 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 [30]. 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.** Other proposals of using high-level query languages for graph analysis include Cypher Query Language for Neo4j [29] and SPARQL for AllegroGraph [1]. In these cases, users have no control over the data layout or evaluation order. The convenience in using a high-level language is unfortunately accompanied with a significant loss in performance.

Because the popular MapReduce computation model does not support graph analysis well [39], 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 [9]. 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 [25]. Parallelization of SociaLite is promising because of the language’s high-level semantics. However, we currently only have a sequential implementation and parallelization is work in progress.

An alternative approach to simplify graph analysis development is the use of graph libraries. Examples of open-source libraries include the Boost Graph Library (BGL) [6], JGraphT [16], JUNG [17] and SNAP [36]. These libraries include highly optimized routines for common graph algorithms. However, users have a limited choice of algorithms and any deviations from the canned set of libraries require significant effort. The programming skill requirement of this approach is much higher compared to the approach of using database languages like Datalog. The latter enables database professionals to formulate new queries quickly, which is important in the data mining process.

Language-Integrated Query (LINQ) is an extension to the C# programming language that adds declarative style queries to the language. LINQ queries are written declaratively in SQL-like languages. Various techniques have been developed to optimize these queries [40] and make them run as fast as loop-based imperative code [28].

8. CONCLUSION

Database languages are powerful as they enable non-expert programmers to formulate data 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 data structures; examples include the use of data as an array index and nested tables.

In our evaluation of graph algorithms in SociaLite, we found that the optimizations proposed sped up almost all of the applications by at least a factor of three and up to 22-fold. 6 out of 7 graph algorithms run faster in SociaLite by 25% to almost 200% than the first implementations in Java. On the betweenness centrality application, SociaLite 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. Finally, Java implementations required about 10 times as many lines as SociaLite. This demonstrates that graph analysis can be succinctly and efficiently implemented in SociaLite.

9. ACKNOWLEDGMENTS
10. REFERENCES