SociaLite: An Efficient Graph Query Language Based on Datalog

Jiwon Seo, Stephen Guo, and Monica S. Lam, *Member, IEEE*

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. Moreover, recursive aggregate functions make it possible to implement more graph algorithms that cannot be implemented in Datalog.

We evaluated SociaLite by running nine graph algorithms in total; eight for social network analysis (shortest paths, PageRank, hubs and authorities, mutual neighbors, connected components, triangles, clustering coefficients, and betweenness centrality) and one for biological network analysis (Eulerian cycles). We use two real-life social graphs, LiveJournal and Last.fm, for the evaluation as well as one synthetic graph. 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, with only 16% overhead for the largest benchmark, and 25% overhead for the worst case benchmark. Most importantly, being a query language, SociaLite enables many more users who are not proficient in software engineering to perform network analysis easily and efficiently.

Index Terms—Datalog, aggregates, query languages, graph algorithms, social network analysis.

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 [1].

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.

1.1 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 [2], [3], 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. $MIN is a pre-defined aggregate function in SociaLite.

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

$$\text{PATH}(t, d) \quad :- \quad \text{EDGE}(1, t, d). \tag{1}$$
$$\text{PATH}(t, d) \quad :- \quad \text{PATH}(s, d_1), \text{EDGE}(s, t, d_2), \; d = d_1 + d_2. \tag{2}$$
$$\text{MINPATH}(t, \$\text{MIN}(d)) \quad :- \quad \text{PATH}(t, d). \tag{3}$$

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

well as inefficient data structures. We ran this shortest-paths algorithm on LogicBlox [4], 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 [5] 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.

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 2, 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.

EDGE (int *src*:0..10000, (int *sink*, int *len*)).
PATH (int *sink*:0..10000, int *dist*).

$$\text{PATH}(t, \$\text{MIN}(d)) \quad :- \quad \text{EDGE}(1, t, d); \tag{4}$$
$$:- \quad \text{PATH}(s, d_1), \text{EDGE}(s, t, d_2), \; d = d_1 + d_2. \tag{5}$$

Fig. 2. 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 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.

Integration with imperative languages. SociaLite is intended to be used as a database component in a program written in a conventional imperative programming languages, such as Java. SociaLite queries can be embedded in a Java program. Data is exchanged between the Java and SociaLite programs via the input and output data structures, which are type checked dynamically. The output tables of the SociaLite program, whenever accessed, contain the results of applying the culmination of queries executed so far to the values of the input tables. The compiler uses incremental semi-naive evaluation to eliminate unnecessary re-evaluation of queries. In addition, SociaLite can also be extended by imperative functions. SociaLite supports a special *function* predicate, which invokes a pure external function to compute *return* values.

Comparison with other languages. We show that SociaLite subsumes two well-known Datalog extensions: choice-least and choice-most operators [6] for expressing greedy algorithms and monotonic aggregate functions [7]. In addition, we show that SociaLite is strictly more expressive than Pregel, a well-known graph analysis language.

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 Eulerian cycle, an important algorithm used in DNA sequence assembly. 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, as well as one synthetic graph, consisting of 2M nodes and 4M edges. Across the spectrum of graph algorithms, SociaLite programs outperform initial implementations in Java, and are also within 25% of their highly optimized Java counter-

parts. 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 incrementally. Then, Section 4 describes the integration with imperative languages, and Section 5 compares SociaLite with other Datalog extensions and a graph analysis language, Pregel. In Section 6, we explain how users can specify a desired evaluation order, and in Section 7 we evaluate the performance of SociaLite. Related work is reviewed in Section 8 and we conclude in Section 9.

2 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.

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*. 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 2. 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 optimize the column to be represented by array indices. 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 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.

Graphs in imperative programming are frequently represented as an adjacency list, which 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 and a form of denormalization. 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,

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

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 2 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.

2.3 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; a single test on the element suffices to determine if the corresponding tuples are matching tuples. 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.

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. Each rule can have multiple bodies, that are logically disjunctive; i.e., 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 2 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 immediate consequence operator [8], denoted as T_p, for Datalog program P is extended for a SociaLite program with recursive aggregate functions. Suppose we have a SociaLite rule with recursive aggregate function F_1, ..., F_k, having m bodies as follows.

$$\begin{aligned}\mathrm{P}(x_1,...,x_n,F_1(y_1),...,F_k(y_k)) &: -\mathrm{Q}_1(x_1,...,x_n,y_1,...,y_k);\\ &\dots\\ &: -\mathrm{Q}_m(x_1,...,x_n,y_1,...,y_k).\end{aligned}$$

Let I be a set of tuples satisfying predicates $\mathrm{Q}_1,...,\mathrm{Q}_m$; then $T_p(I)$ for the above rule yields

$$\{(x_1,...,x_n,z_1,...,z_k)|z_i{=}F_i(s_{y_i}) \wedge s_{y_i}{=}\{y_i|\forall(x_1,...,x_n,y_1,...,y_k)\in I\}\}$$

The arguments x_1, ..., x_n in predicate P are referred as *group-by parameters* for the aggregate functions F_1, ..., F_k. Note that the above definition of T_p can be easily extended for a rule having multiple predicates in its bodies.

3.2 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 $\leq$ and $\geq$, respectively. Summation, in contrast, is not a meet operation since it is not idempotent. Note that meet operations can be extended to operate on a set of elements through repeated application; since meet operations are associative, the order of the application does not matter.

Recursive aggregate functions in a SociaLite program are required to be meet operations for the program to have a well-defined fixed-point semantics. A SociaLite program is *stratified* by its recursive aggregate functions, such that all the operations in recursive rules in a stratum must be monotone under the partial orders of the recursive aggregate functions. If a stratum contains multiple recursive aggregate functions, the aggregate functions must be monotone to one another in the recursion; otherwise, the program is not considered as stratifiable. The following theorem gives a fixed point semantics to a stratified SociaLite program.

Theorem 1: Consider a stratum P with recursive aggregate functions in a stratified SociaLite program. Suppose that the immediate consequence operator T_p, denoted as h, is decomposed into g and f, such that g is the function that applies the aggregate functions to each set of tuples grouped by the group-by parameters, and function f represents the rest of operations in P. If g is a meet operation, inducing a partial order $\sqsubseteq$, and f is monotone under $\sqsubseteq$, 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 [9]. 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, thus is greater than any other elements.)

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

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

$h^k(\emptyset)$ 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. □

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.

With its fixed point semantics, recursive aggregate functions subsume other Datalog extensions, such as the mono-

tonic aggregate functions proposed by Ross and Sagiv [7]. This is discussed in Section 5.

A meet operation has a symmetric dual operation, called a *join* operation [9]. Like a meet operation, a join operation is idempotent, commutative, and associative; it defines a semi-lattice, inducing a partial order $\sqsupseteq$, such that the result of the operation for any two elements is the *least upper bound* of the elements.

Theorem 1 can be simply restated for a join operation, which would define least fixed point semantics and subsumes the traditional least fixed point semantics of Datalog. Function g for a Datalog program corresponds to the *union* operation, and the induced partial order is the subset relation; then the Theorem explains the least fixed point semantics of Datalog.

3.3 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.

Consider a stratum P of a stratified SociaLite program, and its transformation P' for the semi-naive evaluaton. For each rule in P, we add new *delta* rules to P' by replacing each intentional predicate in the rule body with a *delta* predicate that corresponds to the updated facts at previous iteration.

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

Input: A stratum P of a SociaLite program and its transformation P' with delta rules. Function g and f for P, as they are defined in Theorem 1. Function f is required to be distributive as well as monotone. Function f' that extends f for the evaluation of the delta rules in P'; f' takes an additional parameter which represents updated facts at previous iteration.

Output: Returns the greatest fixed-point solution for P.

Method:

$R_0 \leftarrow \emptyset$
$R_1 \leftarrow g\left(f(R_0) \cup R_0\right)$
$\Delta_1 \leftarrow R_1, i \leftarrow 1$
do
  $i \leftarrow i + 1$
  $R_i \leftarrow g\left(f'(R_{i-1}, \Delta_{i-1}) \cup R_{i-1}\right)$
  $\Delta_i \leftarrow R_i - R_{i-1}$
while $\Delta_i \neq \emptyset$
return R_i ■

As an example, the shortest-paths algorithm in Figure 2 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

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

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 $R_i = h^i(\emptyset)$. We use h_i to simply denote $h^i(\emptyset)$.
Basis: $R_1 = g\left(f\left(R_0\right) \cup R_0\right) = g \circ f\left(\emptyset\right) = h^1(\emptyset)$
Inductive step: Assuming $R_k = h^k(\emptyset)$ for all $k \leq i$,

$$\begin{aligned}
R_{i+1} &= g\left(f'\left(R_i, \Delta_i\right) \cup R_i\right) \\
&= g\left(f'\left(h_i, h_i - h_{i-1}\right) \cup h_i\right) \\
&= g\left(f'\left(h_i, h_i - h_{i-1}\right) \cup g \circ f(h_{i-1})\right) \\
&= g\left(f'\left(h_i, h_i - h_{i-1}\right) \cup f(h_{i-1})\right) && (6) \\
&= g\left(f'\left(h_i, h_i - h_{i-1}\right) \cup f'\left(h_{i-1}, h_{i-1}\right)\right) && (7) \\
&= g\left(f'\left(h_i \cup h_{i-1}, (h_i - h_{i-1}) \cup h_{i-1}\right)\right) && (8) \\
&= g\left(f'\left(h_i \cup h_{i-1}, h_i\right)\right) \\
&= g\left(f'\left(h_i, h_i\right)\right) && (9) \\
&= g\left(f\left(h_i\right)\right) = h^{i+1}(\emptyset)
\end{aligned}$$

Line 6 is true because g is a meet operation, and line 7 is true because f' is equivalent to f if R_{i-1} is used in place of Δ_{i-1}. Line 8 follows from f being distributive, and line 9 from $g \circ f'$ as well as h being monotone. □

The theorem can be extended 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.

3.4 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 leveraging 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 2 yields Dijkstra's shortest-paths algorithm.

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.

4 Integration with Imperative Languages

Intended to be used as a database query language in applications written in conventional languages, SociaLite is integrated into imperative programming languages in two ways. First, SociaLite queries can be *embedded* into imperative programs. Typically the imperative programs will prepare the input, write into the database and invoke queries on the database to compute the answers. Second, SociaLite can be *extended* with pure function predicates, written in imperative languages.

We use Java in this section as an example of an imperative language, but it can be applied to other imperative languages as well.

4.1 Embedding SociaLite

A SociaLite database is defined as a Java class SociaLiteDB. A Java program specifies a SociaLite program through the invocation of a series of *run* method. The Java program interfaces with the SociaLite database via input and output tables. The method *getTable* binds SociaLite tables to Java tables (Figure 3).

The *Table* class, whose instance corresponds to a SociaLite table, is defined as a set of *tuples*. Through methods in the *Table* class, Java programs can read from and write data into the SociaLite table (Figure 4). The *setTupleClass* method specifies the type of the tuples, so the types can be checked by the SociaLite database dynamically. The *insert* method adds a tuple to the table and the *read* method applies a visitor method to every tuple in the SociaLite table. Additionally, the table can be cleared via the *clear* method, so that the database can be used on different sets of inputs.

Method	Description
`run(query)`	Execute the given query
`getTable(table)`	Return the Java representation of the given SociaLite table

Fig. 3. Methods in SocialiteDB class

An example of a Java program with a SociaLite database is shown in Figure 5.

Method	Description
`setTupleClass(cls)`	Set the tuple class that defines the types of tuples in the table
`insert(tuple)`	Insert the tuple into the table
`read(visitor)`	Read the tuples in the table and pass them to the visitor class
`clear()`	Clear the tuples in the table

Fig. 4. Methods in Table class

1) This program first creates a new SociaLite database instance, declares EDGE table having two columns, src and target, with the latter represented as a nested table.
2) The SociaLite EDGE table is bound to the Java edge table with the getTable method. The nested structure in the SociaLite program is oblivious to the Java program. Here the Java program simply declares an Edge as a table of EdgeTuple objects, each of which has two integers. With the method invocation edge.setTupleClass(EdgeTuple.class), the Java type for edge is checked against the declaration in SociaLite, and an exception is thrown if the types do not match at run time. In this example, an edge with src 0 and target 1 is inserted into the edge table.
3) A SociaLite query is executed; the query declares that if there is an edge from node s to t, then there exists a reverse edge from node t to s. After the execution, the EDGE table represents an undirected graph.
4) The result in the edge table is then read with a callback method visit applied to every tuple in the edge table.

```
// 1. create a SocialiteDB with Edge table
SocialiteDB db=new SocialiteDB();
db.run("Edge(int src, (int target)).");
// 2. insert a tuple into the Edge table
class EdgeTuple extends Tuple {
    int src; int target;
}
Table edge = db.getTable("Edge");
edge.setTupleClass(EdgeTuple.class);
EdgeTuple t = new EdgeTuple();
t.src= 0; t.target = 1;
edge.insert(t);
// 3. run a SociaLite query
db.run("Edge(t,s) :- Edge(s,t).");
// 4. print tuples in the Edge table
edge.read(
  new TupleVisitor() {
    public boolean visit(Tuple _t) {
      EdgeTuple t=(EdgeTuple)_t;
      println(t.src+"->"+t.target);
      return true;
    }
  });
```

Fig. 5. Java program calling SociaLite

All the queries made to the SociaLite database are cumulative. Each invocation inserts an additional SociaLite declaration or rule to the accumulated set of rules. Each time a query is run, the database applies all the rules

accumulated so far to the values inserted into the tables. If a table has been "cleared", then its contains only those values inserted subsequent to the "clear" operation.

The SociaLite compiler applies incremental semi-naive evaluation to minimize re-evaluations across runs. Normally, the evaluation of a query can simply be applied to all the results accumulated so far until convergence. However, if a clear operation has been invoked on table t since the last run, all results dependent on t are also cleared before running the queries.

4.2 Extending SociaLite

SociaLite can be extended with pure functions written in conventional programming languages, with the help of *function predicates*. A function predicate is expressed as $(v_1, \ldots, v_m) = \$f(a_1, \ldots, a_n)$, where f is the name of an external pure function with n input parameters and m returned values. All the input variables $a_1, \ldots, a_n$ must be bound to non-function predicates or in return variables of preceding function predicates. Also, the return variables $v_1, \ldots, v_m$ must not appear as input parameters or return variables of preceding predicates. If a function predicate is used in recursive rules involving an aggregate function, the external function must be monotonic with respect to the aggregate function to ensure the existence of a greatest fixed-point solution.

5 Comparison with Other Languages

This section describes how SociaLite subsumes two important Datalog extensions for graph algorithms: the choice operators and monotonic aggregate functions. By showing how a Pregel execution can be specified in two SociaLite rules, we prove that SociaLite subsumes Pregel as well.

5.1 Greedy Algorithms Using Choice Operators

Greedy algorithms, where optimization heuristics are used locally to approximate globally optimal solutions, are important and prevalent. The inability of Datalog in expressing greedy algorithms has prompted the proposal of new Datalog extensions called the greedy choice operators [6], [10]. These operators, *choice-least* and *choice-most*, are themselves derived from the *choice* operator [11], [12] proposed to add non-determinism to Datalog. The choice operator chooses a tuple nondeterministically in its evaluation, thus allowing exponential search to be expressed simply. More formally, a subgoal of the form $\text{CHOICE}((X), (Y))$ in a rule states that the set of all tuples derived from the rule must follow the functional dependency $X \rightarrow Y$. The arguments X and Y are vectors of variables, where X optionally can be an empty vector.

The choice-least and choice-most operators introduce the use of a heuristic function for choosing a tuple among possible candidates. A subgoal of the form $\text{CHOICE-LEAST}((X), (Y))$ in a rule states that the set of all tuples derived from the rule must follow the functional dependency $X \rightarrow Y$, and that Y has the least cost among all the possible candidates. CHOICE-MOST is defined analogously. The introduction of CHOICE-LEAST and CHOICE-MOST eliminates the backtracking necessary to simulate a non-deterministic algorithm, turning it into a greedy algorithm.

Datalog programs with greedy choice operators can be implemented in SociaLite, with the help of the recursive aggregate functions, as follows. Given a Datalog program with a choice-least operator of the following form,

$$\begin{aligned} \text{P} :- \text{Q}, &\text{CHOICE}((X_1), (Y_1)), \ldots, \text{CHOICE}((X_i), (Y_i)), \\ &\text{CHOICE-LEAST}((X), (C)), \end{aligned}$$

where Q is the conjunction of all the goals in the rule, X, X_1, Y_1, $\ldots X_i$, Y_i denotes vectors of variables, and C is the cost, a single variable ranging over an ordered domain. Note that CHOICE operators here do not induce any non-determinism, because the CHOICE-LEAST operator ensures deterministic evaluation through the greedy selection; the CHOICE operators simply make an arbitrary selection among candidate tuples that yield the minimum cost value. Here is the equivalent SociaLite program for the above program:

$$\begin{aligned} &\text{R}(X, \$\text{MIN}(C)) \ :- \ \text{Q}. \\ &\text{P}_1(X_1, \$\text{CHOICE}(Y_1)) \ :- \ \text{Q}, \text{R}(X, C). \\ &\qquad \ldots \\ &\text{P}_i(X_i, \$\text{CHOICE}(Y_i)) \ :- \ \text{Q}, \text{R}(X, C). \\ &\text{P} :- \text{Q}, \text{P}_1(X_1, Y_1), \ldots, P_i(X_i, Y_i), R(X, C). \end{aligned}$$

where \$CHOICE is a recursive aggregate function whose partial order depends on the evaluation order, such that an element evaluated prior is considered smaller than an element evaluated posterior. Note that it is trivial to implement \$CHOICE; it always returns the first tuple found to satisfy the subgoal. \$CHOICE enforces functional dependency from its group-by parameters to its argument. Datalog programs with choice-most operator can be implemented with the function \$MAX analogously.

5.2 Monotonic Aggregate Functions

Monotonic aggregate functions, proposed by Ross and Sagiv [7], are another well-known Datalog extension for graph algorithms. Ross and Sagiv define aggregate functions as *monotonic* if adding more elements to the multi-set being operated upon can only increase (or decrease) the value of the function. For example, according to Ross and Sagiv, both maximum and summation are monotonic aggregate functions. Note that in our terminology, maximum is monotonic but not summation. Summation is not idempotent, which means that adding incremental values to a partial sum without removing duplicates will yield the wrong answer.

Because of their definition of monotonic functions, Ross and Sagiv impose an addition constraint on their monotonic aggregate functions before they can be used recursively. Namely, 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. For example, our shortest-paths algorithm shown in Figure 2 would fall outside their formulation.

Note that any aggregate function satisfying Ross and Sagiv's restriction is also a meet operation. Functional dependency guarantees idempotency trivially. Thus, recursive aggregate functions are a strict generalization of monotonic aggregate functions.

5.3 Comparison with Pregel

Pregel [13] is a well-known graph analysis framework developed at Google. A Pregel program is vertex-centric; in each iteration of the computation, a local operation is performed on a vertex based on the data received in the previous iteration and messages are sent to neighboring vertices. When all the messages reach their destination, the next iteration begins.

The Pregel execution model can be succinctly expressed in SociaLite. The ith iteration of a Pregel computation can be written with the following two rules:

$$\begin{aligned} \textsc{State}(s, i+1, \$\textsc{Accuml}(m, v)) \ :- \ & \textsc{State}(s, i, v), \\ & \textsc{Msg}(s, i, m). \\ \textsc{Msg}(t, i+1, m) \ :- \ & \textsc{State}(s, i+1, v), \textsc{Edge}(s, t), \\ & m = \$\textsc{ComputeMsg}(v, t). \end{aligned}$$

The first rule updates the state of vertex s in iteration $i+1$ based on its value v and new messages received in iteration i, using the aggregate function \$ACCUML. The second rule uses \$COMPUTEMSG to create messages for its neighboring vertices t in the next iteration. This shows that SociaLite is more expressive than Pregel.

6 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.

6.1 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.

6.2 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 finding betweenness centrality [14]:

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

Fig. 6. Ordering of evaluation in SociaLite.

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

The predicate $\textsc{SP}(n, d, p)$ 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 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 6.

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.

6.3 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:

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

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.

7 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 the compiler with seven core graph analysis routines and a complete algorithm for computing betweenness centrality. Finally, we experiment with using SociaLite to find Eulerian cycles, an important algorithm in bioinformatics for DNA assembly.

7.1 Comparison of Datalog Engines

To evaluate how SociaLite compares with state-of-the-art Datalog engines, we experimented with three representative systems: Overlog [15], IRIS [16], and LogicBlox version 3.7.10 [4]. 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 2, the program ran in 1.2 seconds. Had we written the SociaLite program using recursive aggregate functions, as shown in Figure 2, 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.

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.

7.2 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, and other general graph metrics [14], [17], [18]. 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 a graph.

PageRank: PageRank [19] is a link analysis algorithm (used for web page ranking) which computes the importance of nodes in a graph.

Hubs and Authorities: Hyperlink-Induced Topic Search (HITS) [20] is another link analysis algorithm that computes the importance of nodes in a graph.

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.

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.

Triangles: Find all triangles (i.e., cliques of size three) in a graph.

Clustering Coefficients: We compute the local clustering coefficient of each node, which is a measure of how well a node's neighbors are connected with each other.

7.3 SociaLite Programs

All the benchmarks in this study can be succinctly expressed in SociaLite. Whereas SociaLite programs for the benchmarks range from 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 8). 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 7.6.)

7.4 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 LiveJournal, a website that enables individuals to keep a journal and read friends' journals [21]. 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 [22]. The Last.fm dataset is

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

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

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 9. 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.

7.5 Analysis of the Optimizations

Our next set of experiments evaluates the contribution of the different optimizations proposed in this paper.

7.5.1 Data Layout Optimizations

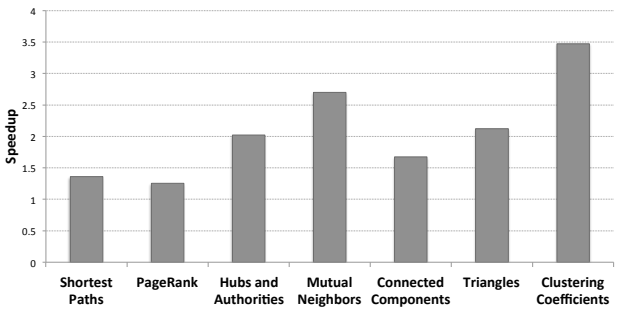

Fig. 10. Speedup due to tail-nested tables and data indexing over column-orientation, with other optimizations applied.

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 10). 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.

7.5.2 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 (Figure 11). 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.

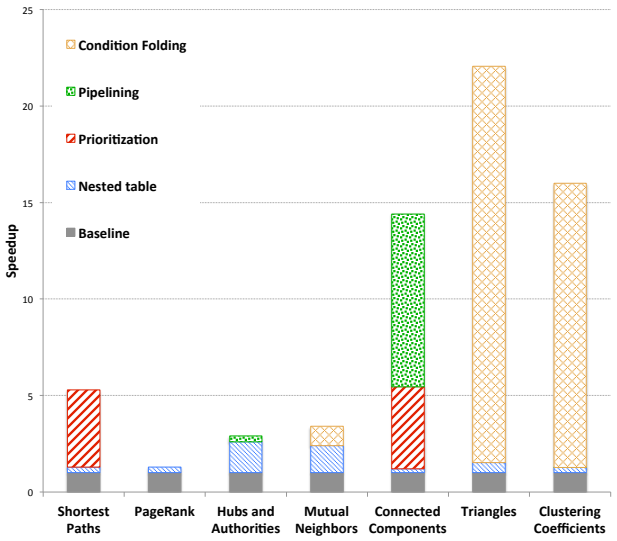

Fig. 11. Speedups from optimizations. Baseline is SociaLite with column orientation.

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 10 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 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.

7.6 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.

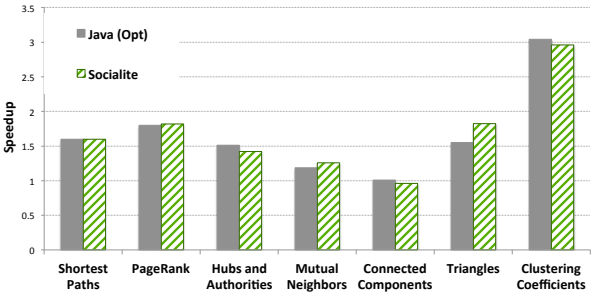

Fig. 12. Performance of optimized SociaLite programs and optimized Java programs relative to initial implementation in Java.

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 12, 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 8, 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.

7.7 Betweenness Centrality

Besides the seven core algorithms, we also experimented with using SociaLite to implement a full application, betweenness centrality [14]. Betweenness centrality is a

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. 13. Betweenness Centrality: Java vs SociaLite

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 [23], 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 6. Similarly, 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 13 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.

7.8 Eulerian Cycles

Bioinformatics is another interesting application domain for SociaLite, since it also requires analyses of networks, such as the protein interaction networks [24]. Of great importance in this domain is DNA sequence assembly, which infers full-length (unknown) DNA sequences from experimentally-obtained short fragments of DNA sequences. A core routine used is the creation of a De Bruijn graph [25] from the genome sequence fragments [26]. In a k-dimensional De Bruijn graph, the edges of the graph are unique genome subsequences of a given length k within the fragments, and the nodes of the graph represent genome subsequences of length k-1 within the fragments. Thus, two nodes in the graph are connected if the suffix of a node is an exact match of length k-2 of the prefix of another node [26]. The Eulerian cycle in such a graph represents the original DNA sequence [26].

To find the Eulerian cycle in the De Bruijn graph, Hierholzer's algorithm can be used, which follows a trail of edges from a node having unvisited edges until returning to the starting node; this is repeated until all the edges are visited [27].

With the help of the recursive aggregate function $CHOICE defined in Section 5.1, we can use SociaLite to implement the Hierholzer's algorithm, as shown in Figure 14. This program is executed repeatedly, with each round excluding edges that have already been visited, until there are no edges left. Rule 10 states that there is a path from node S to node t found at time L, if there is an edge from S to t and the edge is not already in the trail. In the first iteration S is an arbitrary node and L is 1; in subsequent iterations, S is a node, with unvisited edges, that was visited at time L $-$ 1. To handle the negation in the predicate !PATH$(s, _, t)$, *inflationary* semantics [28] is assumed, which checks if a fact exists at the time when the negation is evaluated. In short, Rule 10 states that the first edge found for the trail is the edge from node S to t. The function $CHOICE enforces the functional dependency $(s, l) \rightarrow t$, ensuring that only a single edge is selected at a time. Rule 11 recursively states that if there is a path to node s found at time l_1, and there is an edge from s to t, then there is a path from s to t found at $l_1 + 1$ as long as the edge is not already part of the trail.

$$\text{PATH}(s, l, \$\text{CHOICE}(t)) :- s{=}\text{S}, l{=}\text{L}, \text{EDGE}(s, t), !\text{PATH}(s, \_, t); \quad (10)$$

$$:- \text{PATH}(\_, l_1, s), l = l_1 + 1, \text{EDGE}(s, t), !\text{PATH}(s, \_, t) \quad (11)$$

Fig. 14. A step in Hierholzer's algorithm for finding an Eulerian cycle in SociaLite.

As we did with the betweenness centrality application, one author of this paper implemented the algorithm in SociaLite and the other in Java. Figure 15 compares the two implementations on an input graph with 2M nodes and 4M edges. The input graph is synthesized by augmenting randomly generated cycles. We run the two programs on a Intel Core2 processor running at 2.66GHz.

The performance overhead of SociaLite compared to

Comparison	Java	SociaLite
Development time (hours)	4.5	0.3
Lines of code	105	16
Execution time (seconds)	2.8	3.5

Fig. 15. Eulerian Cycle: Java vs SociaLite

Java implementation is 25%. The majority of the overhead comes from having to iterate over the nodes to find a starting node with unvisited edges at the beginning of each iteration. While the overhead observed is higher than previous benchmarks, the Hierholzer algorithm has a short running time. The key takeaway is that recursive aggregate functions can be used to express calculations for which Datalog is unsuitable, thus making it possible for the whole algorithm to be expressed quickly and succinctly in SociaLite.

8 Related Work

Extending the semantics of Datalog. Various attempts have been made in the past to allow incremental analysis of aggregate functions in Datalog [29], [30]. 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 [29].

In Section 3 we described in detail the two well-known Datalog extensions: Datalog for greedy algorithms and Datalog with monotonic aggregate functions. In short, recursive aggregate functions in SociaLite extends Datalog with more expressive power than the two extensions.

Recently, Mazuran et al. proposed Datalog^{FS} that extends Datalog with frequency support goals, which allow counting the distinct occurrences satisfying given goals in rules. With the frequency support goals inside recursive rules, Datalog^{FS} can express large classes of algorithms that cannot be expressed in Datalog. The counting operation in Datalog^{FS} is a meet operation; the domain of the operation is $\{S, n\}$, where S is a set of tuples satisfying a given goal, and n is the number of distinct elements in S. For any two elements $\{S_1, n_1\}$ and $\{S_2, n_2\}$, the greatest lower bound by the counting operation is simply $\{S_1 \cup S_2, n_3\}$ where n_3 is the number of distinct elements in $S_1 \cup S_2$. (XXX: maybe too verbose.. we might not need to go into the details...) Since the counting operation is a meet operation, the frequency support goals can be implemented as recursive aggregate functions in SociaLite; hence any algorithm in Datalog^{FS} can be rewritten with recursive aggregate functions in SociaLite.

The semantics of the function predicates in SociaLite is similar to the previously proposed external predicates [31].

Other Datalog research. Recently Datalog research has been revived in many domains including security [32], programming analysis [33], and network/distributed systems [34], [35]. 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 [36].

In contrast, SociaLite has different goals. It aims to make graph analysis easy and efficient. The extensions of SociaLite 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 [37]. Nested data structures are also used in object-oriented databases [38]. 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 [39]. 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 [40] and GOQL [41]. 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 [1], 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 [42]. We showed in Section 5 how SociaLite is more expressive than Pregel, a vertex-centric framework for distributed graph analysis [13]. Parallelization of SociaLite while promising, due to its high-level language semantics, is outside the scope of this paper.

9 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. The integration with imperative languages enables wider class of applications to be implemented in SociaLite. 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 initial 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. For Eulerian cycle algorithm, an important algorithm in bioinformatics, the SociaLite implementation is 25% slower than the optimized Java program, which is within a reasonable range, considering the development time of 4.5 hours compared to just 15 minutes for SociaLite.

The most important contribution of SociaLite is that, as a query language, it makes efficient graph analysis 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. 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] 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.

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

[3] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The magic of duplicates and aggregates," in *VLDB*, 1990, pp. 264–277.

[4] "Logicblox inc," http://www.logicblox.com/.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Num. Math.*, vol. 1, no. 1, pp. 269–271, 1959.

[6] S. Greco and C. Zaniolo, "Greedy algorithms in datalog." *TPLP*, pp. 381–407, 2001.

[7] K. A. Ross and Y. Sagiv, "Monotonic aggregation in deductive databases," *Journal of Computer and System Sciences*, vol. 54, no. 1, pp. 79–97, 1997.

[8] J. W. Lloyd, "Foundations of logic programming." Springer, 1987.

[9] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[10] S. Greco, C. Zaniolo, and S. Ganguly, "Greedy by choice," in *PODS*, 1992, pp. 105–113.

[11] F. Giannotti, D. Pedreschi, D. Sacca, and C. Zaniolo, "Non-determinism in deductive databases," in *Deductive and Object-Oriented Databases*, 1991.

[12] R. Krishnamurthy and S. A. Naqvi, "Non-deterministic choice in datalog," in *JCDKB*, 1988, pp. 416–424.

[13] 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*, 2010, pp. 135–146.

[14] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

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

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

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

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

[19] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *WWW7*, 1998, pp. 107–117.

[20] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, pp. 604–632, 1999.

[21] "Livejournal," http://www.livejournal.com/.

[22] "Last.fm," http://last.fm/.

[23] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[24] J. D. L. Rivas and C. Fontanillo, "Protein-protein interactions essentials: Key concepts to building and analyzing interactome networks." 2010.

[25] N. G. de Bruijn, "A combinatorial problem," *Koninklijke Nederlandse Akademie v. Wetenschappen*, vol. 49, pp. 758–764, 1946.

[26] R. M. Idury and M. S. Waterman, "A new algorithm for DNA sequence assembly." *Journal of Computational Biology*, vol. 2, pp. 291–306, 1995.

[27] C. Hierholzer and C. Wiener, "Ueber die mglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren," *Mathematische Annalen*, vol. 6, pp. 30–32, 1873.

[28] P. G. Kolaitis and C. H. Papadimitriou, "Why not negation by fixpoint?" *J. Comput. Syst. Sci.*, vol. 43, pp. 125–144, 1991.

[29] S. Ganguly, S. Greco, and C. Zaniolo, "Minimum and maximum predicates in logic programming," in *PODS*, 1991, pp. 154–163.

[30] S. Sudarshan and R. Ramakrishnan, "Aggregation and relevance in deductive databases," in *VLDB*, 1991, pp. 501–511.

[31] R. G. Chimenti, Danette and R. Krishnamurthy, "Towards on open architecture for ldl," in *VLDB*, 1989, pp. 195–203.

[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*, 2010, pp. 247–266.

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

[34] 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*, 2010, pp. 223–236.

[35] 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*, vol. 52, no. 11, pp. 87–95, 2009.

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

[37] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, pp. 85–97, 1996.

[38] R. Hull, "A survey of theoretical research on typed complex database objects," in *Databases*, 1987, pp. 193–261.

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

[40] M. P. Consens and A. O. Mendelzon, "Expressing structural hypertext queries in graphlog," in *Hypertext*, 1989, pp. 269–292.

[41] L. Sheng, Z. M. Özsoyoglu, and G. Özsoyoglu, "A graph query language and its query processing," in *ICDE*, 1999, pp. 572–581.

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

Stephen Guo is a data scientist in Silicon Valley. He received his BS and MS in Computer Science from Stanford University. His research interests span crowdsourcing, data mining, machine learning, and social networks.

Monica S. Lam has been a Professor in the Computer Science Department at Stanford University since 1988; she is also the founding Director of the Stanford MobiSocial Computing Laboratory. She received her BS in Computer Science from University of British Columbia and her PhD in Computer Science from Carnegie Mellon University. Her research interests spanned high-performance computing, computer architecture, compiler optimizations, security analysis, virtualization-based computer management, and most recently, open social networks. She loves working on disruptive startups–she was on the founding team of Tensilica (configurable processor cores) in 1998, and she was the founding CEO of both MokaFive (desktop management using virtual machines, 2005) and MobiSocial (open social networking, 2012). She is a co-author of the ”Dragon book” and an ACM Fellow.

Jiwon Seo received the B.S. degree in Electrical Engineering from Seoul National University in 2005, and the M.S. degree in Electrical Engineering from Stanford University in 2008, and is currently pursuing the Ph.D. degree in Electrical Engineering from Stanford University. His research interests include distributed systems, data mining, and social networks. Mr. Seo was a recipient of Samsung Scholarship.