Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs

Mohamed Sarwat[1] Sameh Elnikety[2] Yuxiong He[2] Mohamed F. Mokbel[1]

[1]*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA*
[2]*Microsoft Research, Redmond, WA, USA*

ABSTRACT

Horton+ is a graph query processing system that executes declarative reachability queries on a partitioned attributed multi-graph. It employs a query language, query optimizer, and a distributed execution engine. The query language expresses declarative reachability queries, and supports closures and predicates on node and edge attributes to match graph paths. We introduce three algebraic operators, *select*, *traverse*, and *join*, and a query is compiled into an execution plan containing these operators. As reachability queries access the graph elements in a random access pattern, the graph is therefore maintained in the main memory of a cluster of servers to reduce query execution time. We develop a distributed execution engine that processes a query plan in parallel on the graph servers. Since the query language is declarative, we build a query optimizer that uses graph statistics to estimate predicate selectivity. We experimentally evaluate the system performance on a cluster of 16 graph servers using synthetic graphs as well as a real graph from an application that uses reachability queries. The evaluation shows (1) the efficiency of the optimizer in reducing query execution time, (2) system scalability with the size of the graph and with the number of servers, and (3) the convenience of using declarative queries.

1. INTRODUCTION

Graphs are widely used in many application domains, including social networking [31], software collaboration [4], geo-spatial road networks [37], interactive gaming [44], among others [13, 27]. For example in a social network graph, a node represents a person, photo, video, location, event, or group. An edge represents a binary relation between two persons such as friendship, family or work relations such as "advisor of" and "manager of". An edge shows that a person is tagged in a photo, is attending an event or is member of a group. Common queries include "finding Alice's photos taken in Singapore", "finding Bob's friends of friends", and "finding all people advised directly or indirectly by Prof. Carol".

Several emerging applications, e.g., Facebook Graph Search [16], allow users to issue interactive queries over a graph. In such applications, graph nodes and edges have several attributes, and users query the modelled entities and their relationships. Such queries can be expressed as graph paths, and we call this query class *reachability queries over an attributed multi-graph*. These applications introduce the following requirements: (1) *Usability*: A query should be declarative, rather than procedural. (2) *Low latency*: As applications are interactive, they require low query execution time. (3) *Scalability*: The employed graphs may not fit on a single server, motivating a distributed system design.

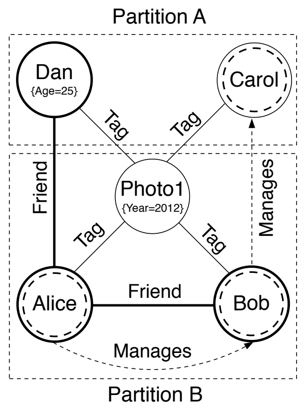
To motivate the need for a new system, we briefly discuss the existing categories of graph processing systems: (1) Relational database systems are not efficient in handling graph reachability queries because these queries are recursive and may contain closures. SQL needs to be extended with recursion to execute such queries. (2) Semi-structured data management systems, e.g. [36, 45], provide query languages such as SPARQL [43] to query RDF data, and XML query languages to query XML documents. SPARQL-based systems target a different class of queries, graph pattern matching rather than reachability. XML querying techniques [45] manage tree-structured data instead of graphs. (3) Centralized graph platforms, e.g. Grace [34] and GraphChi [20], require the graph to fit on one server. (4) Distributed graph platforms, such as Pregel [30], Giraph [19], Trinity [39] and PowerGraph [29], accept procedural programs to be computed over the graph. Such systems focus on graph computations rather than graph querying, and they assume the users to be expert programmers.

In this paper, we present the design, implementation and evaluation of Horton+, a distributed system for processing declarative reachability queries over a partitioned graph. Horton+ employs a declarative query language that uses regular language reachability to express reachability queries over an attributed multi-graph. We introduce three algebraic graph operators, *select*, *traverse*, and *join*, and use them to execute a query plan.

Executing graph reachability queries generates a random access pattern to the memory system; Horton+, therefore, manages the graph in the *main memory* of a cluster of servers. We build a distributed execution engine that executes the three graph operators in parallel and batches messages among the graph servers.

Since the query language is declarative, rather than procedural, there are multiple ways to execute a query. Horton+ is equipped with a query optimizer that reduces the query execution latency. The system maintains a set of graph statistics that are used by the optimizer. Among a rich space of possible query plans, the optimizer employs a cost model and selectivity estimation techniques to estimate the cost of executing each plan. The optimizer selects the execution plan that minimizes the expected number of visited graph nodes and reduces communication among the graph servers.

Horton+ is the first distributed query processor for processing



Graph:
Alice is a friend of both Bob and Dan.
Dan' s age is 25.
Photo1 is taken in year 2012.
Photo1 tags Alice, Bob, Carol, and Dan.
Alice is the manager of Bob.
Bob is the manager of Carol.

Query: 'Alice'-Friend-Person
Answer paths:
Alice-Friend-Bob
Alice-Friend-Dan

Query: 'Alice'(-Manages>-Person)*
Answer paths:
Alice-Manages>-Bob
Alice-Manages>-Bob-Manages>-Carol

Figure 1: Small fragment of a social graph.

```
     Query ::=  NodePred
                Query-EdgePred-Query
                (Query OR Query)
                Query (-EdgePred-Query)*
                (Query-EdgePred-)* Query
                Query (-EdgePred-Query)+
                (Query-EdgePred-)+ Query
  NodePred ::=  Id | NodeType | NodeType{(AttrPred)+}
                (NOT NodePred)
                (NodePred AND NodePred)
                (NodePred OR NodePred)
  EdgePred ::=  EdgeType | EdgeType{(AttrPred)+}
                (NOT EdgePred)
                (EdgePred AND EdgePred)
                (EdgePred OR EdgePred)
  AttrPred ::=  Operand BinaryOperator Operand
   Operand ::=  AttributeName | AttributeValue
  NodeType ::=  Node | TypeId
  EdgeType ::=  Edge | TypeId
```

Figure 2: Abstract syntax of the graph query language.

reachability queries over an attributed multi-graph. We implement Horton+ and evaluate it experimentally to show its scalability and efficiency in executing graph reachability queries using both real and synthetic graphs on a cluster of 16 graph servers. We also compare Horton+ with the Giraph [19] system to highlight benefits of Horton+ and its declarative query language and optimizer. An early version of the system, called Horton was demonstrated [38]. Horton+ introduces major additions, including the graph operators, query optimization, and different query plan and execution model with a formal query language.

In summary, the contributions of this paper are the following: (1) Horton+ as a full-fledged system for distributed processing of graph reachability queries. (2) A formal graph query language that supports reachability queries over an attributed multi-graph (Section 2.2). (3) A distributed query processor that executes query plans over multiple graph servers (Section 3). (4) A query optimizer, equipped with cost and selectivity estimation techniques (Section 4), that optimizes the issued query. (5) An experimental evaluation on a cluster of servers using both real and synthetic graphs and a comparison with Giraph (Section 5).

2. GRAPH MODEL & QUERY LANGUAGE

In this section, we give an overview of the graph model in Section 2.1, and describe the query language in Section 2.2.

2.1 Graph Model

We use a general graph model; an attributed multi-graph $G = \{\mathcal{V}, \mathcal{E}\}$ has a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$. A node represents an entity with a primary key (id), a categorical type (e.g., person, photo or event), and a set of arbitrary attributes. An edge is a binary relationship between two nodes, and it has a categorical type (e.g., friend-of or tagged-in), and a set of arbitrary attributes (e.g., edge direction and edge weight). Multiple edges may link two nodes, representing several relationships.

Figure 1 shows a fragment of a social graph as an example. There are two node types: Person (Alice, Bob, Carol, Dan) and Photo (Photo1), and three edge types: Friend, Tag and Manages. A node may have attributes, e.g., age of Dan is 25. The figure also shows two queries and their answers. The graph is partitioned: Nodes Carol and Dan are in partition A, and nodes Alice, Bob, and Photo1 are in partition B.

Horton+ manages both directed and undirected graphs in main memory with pointer-based representation. In the case of a directed graph, each node stores both inbound and outbound edges to allow queries to traverse both directions.

2.2 Query Language

The objective of the query language is to express graph reachability queries declaratively, rather than in a procedural manner, making developers more productive and allowing query optimization. The query language specifies relationships between entities as graph paths. Figure 2 depicts the abstract syntax of the language. The non-terminal `Query` is the start symbol, and a query starts with a node predicate and possibly followed by a sequence of edge and node predicate pairs. Closures specify paths of arbitrary length, and they are supported using Kleene star "*" (zero or more) and Kleene plus "$^+$" (one or more).

A node predicate specifies a node id, a node type such as *Photo* to match nodes of type photo, or *Node* to match any node type. A node predicate may contain predicates on node attributes, e.g., photos that are black and white taken this year (`Photo{color='B&W' AND year=2013}`). Node predicates can be composed. For example, the predicate `(Photo OR Video)`, which matches nodes of type photo or video, is composed of two predicates. Similarly, an edge predicate specifies an edge type. For instance, a *Tag*, *Friend*, or *Edge* matches a tag edge, friend edge, any edge, respectively. An edge predicate can also specify multiple predicates on edge attributes, e.g., a friendship relation since last year (`Friend{year=2012}`). We use the `"<"` and `">"` symbols to represent edge directions in a directed graph.

Q_1	`'Alice'-Tag>-Photo-Tag<-'Bob'`
Q_2	`Photo-Tag<-Person-Friend-'Alice'`
Q_3	`'Alice'-Tag>-Photo-Tag<-Person-Friend-'Alice'`
Q_4	`'Alice'(-Advice>-Person)*-Coauthor-'Bob'`

Table 1: Query examples.

Example. Table 1 shows four example queries. Q_1 finds all photos in which both Alice and Bob are tagged. Q_1 has three node predicates and two edge predicates: The first node predicate specifies the node id (`'Alice'`). The second node predicate provides the node type (`Photo`), and the third node predicate specifies the node id. The two edge predicates specify edge type (`Tag`). Similarly, Q_2 retrieves all photos in which a friend of Alice is tagged, and Q_3 finds all photos in which Alice is tagged with one of her friends. Q_4 finds whether Prof. Alice or her academic descendants

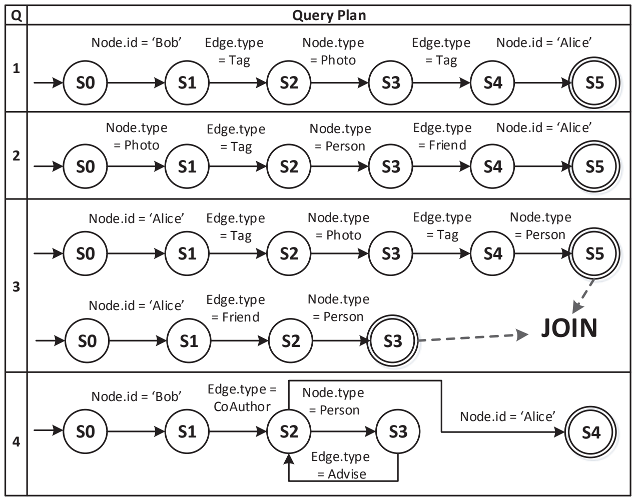


Figure 3: Query execution plan for Q_1, Q_2, Q_3, and Q_4.

have co-authored a paper with Bob. Q_4 is recursive with a closure (`(-Advice>-Person)`*) over a pair of predicates.

3. DISTRIBUTED QUERY PROCESSOR

The query processor receives an input query and returns the matched results. The input query is compiled into a query plan, which can be executed directly, or first optimized and subsequently executed. We introduce three algebraic graph operators, *select*, *traverse*, and *join*. Employing these operators is important: (1) The query processor becomes a composition of a few basic building blocks. Each operator has clearly defined functionality and efficient implementation. (2) The query optimizer builds a cost model for each operator using graph statistics to explore cost-efficient ways to combine them to answer a query. For clarity of presentation, we first describe the operators assuming a centralized environment, and we next present the distributed implementation.

3.1 Compilation into an Execution Plan

An input query is compiled into a plan containing one or more deterministic finite state automata (DFA) and graph operators. The query plan has a recursive tree structure: Each tree node is either a leaf node containing a DFA, or an intermediate node containing the join operator and two trees where each is a query plan.

Example. Figure 3 shows four execution plans for queries Q_1, Q_2, Q_3, and Q_4. The plan for Q_1 is a leaf node containing a DFA that has six states (S_0 to S_5). S_0 is the starting state, and *the set of starting nodes* for the query are the nodes which satisfy the first node predicate (*Node.id='Bob'*) on the transition from S_0 to S_1. The plan of Q_3 includes a join between two query plans, and Q_4's plan contains a loop because the query contains a Kleene star.

3.2 Algebraic Graph Operators

To execute the query plan, we introduce the following graph operators: The *select* operator locates the starting nodes from which path matching proceeds. The *traverse* operator traverses a set of nodes through their edges to a new set of nodes iteratively. The traversal is conditioned by the DFA transition predicates, and partial graph paths are accumulated. The *join* operator joins the result of two query plans to construct longer paths. The query optimizer introduces the join operator into the query plan, and it rewrites a DFA into one or several more efficient DFAs. DFA matching is performed first using the select operator to find the starting nodes, then a sequence of calls to the traverse operator.

Select Operator. The objective of the *select* operator is to determine the set of starting nodes efficiently. The operator takes a DFA and applies the transition predicate from the initial state on nodes to select the set of starting nodes. The operator employs a primary key index if the node primary key (id) is specified, or a hash index on the node type if node type is specified, and secondary index structures are exploited to match attribute predicates. If no index is available or predicate selectivity is very low, the *select* operator applies the predicates on all nodes of the given type.

Traverse Operator. The *traverse* operator is iterative; it receives a set of partial paths and the DFA. Initially the set of partial paths is the set of starting nodes from the *select* operator. In each iteration, the traverse operator matches each partial path into one or more longer paths if they satisfy the transition predicates of the DFA state. For the graph elements that satisfy the predicates, the *traverse* operator appends them to the partial paths, and the partial paths that are not extended in the iteration are dropped. Upon reaching an *accepting* state, the partial paths are returned as matching results for the DFA. This processing pattern results in traversing the graph in breadth-first manner from each starting node.

The complexity of the traverse operator iterations is upper bounded by the product of the number of start nodes, length of the query, and expected number of edges per node in the graph when the query has no closures. We discuss a more accurate cost estimation using graph statistics in Section 4.

The termination of the traverse operator iterations is an important property since the graph may contain cycles, and the DFA may contain loops (corresponding to closures). Each partial path maintains the DFA state at which each node and edge was matched along the path. Before a node is visited, the partial path is checked to ensure that the node is not visited again in the same DFA state. All queries terminate since each node is visited at most once in each state of the DFA for each constructed path.

Join Operator. The *join* operator receives two sets of matching paths from two query plans, and constructs longer paths by joining paths from the two sets. After each query plan is evaluated independently to produce its resulting paths, the join operation is performed based on the ids of the last nodes from the first path set and the ids of the last nodes of the second path set. For example, the two DFA's of Q_3 in Figure 3 are joined based on the `Person` id matched at S_5 of the first DFA and at S_3 of the second DFA.

3.3 Distributed Execution Engine

We use multiple partition servers (a) to query graphs that do not fit in the main memory of a single server, and (b) to evaluate each query in parallel on the partition servers. In this section, we discuss the architecture of the distributed execution engine and the implementation of the algebraic graph operators.

Architecture. The graph is partitioned into disjoint partitions, each managed by a *partition server* that is responsible for managing its own subset of graph data and associated indexes. Each edge is represented at both the source and destination nodes. A remote edge that connects two nodes in two different partitions specifies both the id of the remote node and the target partition where the remote node exists. A server is designated as the *coordinator*, and is responsible for query parsing, compilation and optimization. The coordinator uses a *directory service* that maintains two mappings: a mapping from a partition id to a server network address, and another mapping from node id to a partition id. The coordinator also maintains the graph statistics used by the optimizer.

Algorithm 1 Distributed query processing

```
1:  Function EVALUATE(QTree)
    /* Case 1: The query tree is a Join */
2:  if QTree is JOIN then
3:      L_AnswerPaths ← EVALUATE(QTree.LeftSubTree)
4:      R_AnswerPaths ← EVALUATE(QTree.RightSubTree)
5:      AnswerPaths ← JOINOPERATOR(L_AnswerPaths, R_AnswerPaths)
    /* Case 2: The query tree is a DFA */
6:  if QTree is Leaf DFA node then
7:      for all Partition $\mathcal{P}$ in AllPartitions do
8:          StartingNodes[$\mathcal{P}$] ← $\mathcal{P}$. SELECTOPERATOR(QTree.DFA)
9:          PartialPaths[$\mathcal{P}$] = StartingNodes[$\mathcal{P}$]
10:         if StartingNodes[$\mathcal{P}$] $\neq \varnothing$ then
11:             CurrentPartitions += $\mathcal{P}$
12:     if CurrentPartitions $\neq \varnothing$ then
13:         AnswerPaths ← TRAVERSEOPERATOR(QTree.QueryID,
                                          QTree.DFA, CurrentPartitions)
14: return AnswerPaths
```

Algorithm 2 Traverse operator

```
1:  Function TRAVERSEOPERATOR(QueryID, DFA, CurrentPartitions)
2:  AnswerPaths←$\varnothing$        DFACursor←0
3:  /* Sequence of Breadth First Search (BFS) levels (DFA states) */
4:  while CurrentPartitions $\neq \varnothing$ do
5:      /* STEPS 1 & 2: Local Computation & Global Communication */
6:      for all Partition $\mathcal{P}$ in CurrentPartitions do
7:          PARTITIONEXECUTIONENGINE($\mathcal{P}$, QueryID, DFA, DFACursor)
8:      CurrentPartitions ← $\varnothing$

9:      /* STEP 3: Bulk Synchronization/Coordination */
10:     PartitionsMsgs = WAITFORPARTITIONS(QueryID, DFA)
11:     for all Partition $\mathcal{P}$ in PartitionsMsgs do
12:         AnswerPaths += $\mathcal{P}$.FullPaths
13:         CurrentPartitions += $\mathcal{P}$.NextPartitions
14:     DFACursor ← ADVANCECURSOR(DFA, DFACursor)
15: return AnswerPaths
```

An effective graph partitioning algorithm assigns nodes to partitions to preserve locality in graph accesses, and it reduces communication overhead among partitions during query processing. Graph partitioning is not the focus of our work. Horton+ can, however, incorporate any existing graph partitioning scheme, including hashing (which is used by default) or more sophisticated partitioning tools [1].

3.3.1 Operators

Select Operator. The *select* operator determines the set of starting nodes by evaluating the first transition predicate of the DFA. In the distributed environment, the coordinator invokes the *select* operator on all partitions in parallel. A partition replies with a message to the coordinator indicating whether it finds matching nodes or not. The coordinator registers the partitions with matches as participants in the DFA execution. An important special case is when the DFA transition predicate is a primary key equality, providing the node id. The coordinator invokes the select operator at the target partition, determined by the directory service.

Traverse Operator. The *traverse* operator starts at each partition with one or more starting nodes as determined by the *select* operator. It then initiates a bulk synchronous breadth first search (BFS) [46] among the participant partitions, synchronized by the coordinator, to transition from one DFA state to the next while matching graph elements. The traverse operator iterates through a sequence of BFS levels that represent the DFA states. At each DFA state (BFS level), the *traverse* operator performs three main steps: (1) *Local computation* is the step in which each participant partition runs its own local query execution engine to check on the graph elements that satisfy the current DFA state. If an accepting DFA state is reached, signaling a matching path, the partition communicates the path to the coordinator. (2) *Global communication* is the step where graph partitions send messages to each other to prepare for the next DFA state (which could point to a node in a different partition). (3) *Bulk Synchronization* is the coordination/synchronization step needed to advance to the next DFA state. The coordinator implements a barrier, waiting for synchronization messages from all participant partition servers in order to advance the DFA to the next breadth first search level. When a partition cannot advance a DFA as no match is found, it sends a no-more-matches message to the coordinator. The distributed evaluation terminates when the coordinator receives all matching paths and no-more-matches messages from all participating partitions.

Join Operator. The coordinator either runs the *join* operator locally or assigns it to one of the partition servers with matching results. We apply a heuristic that selects the least-loaded partition server to process the paths matching the two query plans. The join is performed using sort-merge join, which takes $O(R\log(R) + S\log(S) + R + S)$ time to run at the coordinator where R and S denote the number of paths from the two plans to join.

3.3.2 Algorithm and Communication Patterns

Algorithm 1 shows the workflow of the distributed query execution using the operators. The algorithm takes the query execution plan $QTree$ as input, and returns the set of matching paths. The algorithm handles two cases:
Case 1: $QTree$ **is a join**: The algorithm recursively executes the left and right trees of $QTree$ representing two query plans. It then joins their results using the JOINOPERATOR.

The communication pattern of a *join* operator is as follows: (1) The coordinator starts the execution by sending the query plans to the partition servers; it also decides where to run the *join* operator and informs the partition servers. (2) After the evaluation of the two trees completes, the partition servers send their matching paths to the designated server that performs the join operation.

In summary, for a *join* operator, the coordinator sends P control messages to start the execution, and the partitions send their results back to the designated server with at most P data messages, where P is the number of graph partitions.
Case 2: $QTree$ **is a leaf DFA node**: The algorithm runs an initialization step (lines 6 to 11) where the coordinator broadcasts the query DFA to all partitions to invoke the select operator, SELECTOPERATOR, to find the starting nodes at each partition. Next, TRAVERSEOPERATOR (Algorithm 2) is invoked to find the matching paths. Finally, the matched paths are returned as answer.

Algorithm 2 gives the pseudo code of the TRAVERSEOPERATOR. The algorithm takes the following inputs: (1) *QueryID*: represents the ID of issued query, (2) *DFA*: denotes the query DFA, and (3) *CurrentPartitions* represents the set of participant graph partitions. TRAVERSEOPERATOR initializes the DFA cursor to the first DFA state (*DFACursor*←0). The algorithm iterates over a sequence of breadth first search (BFS) levels (lines 4 to 14).

STEP 1 & 2: Local Computation & Global Communication (lines 4 to 8): At each BFS level, the coordinator signals all partitions $\mathcal{P} \in$ *CurrentPartitions* to advance the *DFACursor* to the next DFA state. Each partition receives the partial path matches from other partitions with the ending nodes belonging to the partition, and it combines them with its local partial matches. Then, it checks for local graph elements to match the current DFA state. When the next DFA state points to a remote node (located in a different partition), the partition server buffers all the partial matches towards the same remote partition into a message, and sends the message to the remote partition. As each partition server sends at most one message to all other partitions, the total number of these data messages at each level of BFS is bounded by $P(P-1)$.

Q	Alternative query execution plans
1	**Plan1** : `'Alice' Tag Photo Tag 'Bob'` **Plan2** : `'Bob' Tag Photo Tag 'Alice'` **Plan3** : `('Alice' Tag Photo)⋈('Bob' Tag Photo)`
2	**Plan1** : `Photo Tag Person Friend 'Alice'` **Plan2** : `'Alice' Friend Person Tag Photo` **Plan3** : `('Alice' Friend Person)⋈(Photo Tag Person)`

3	**Plan1** : `'Alice' Tag Photo Tag Person Friend 'Alice'`
	Plan2 : `'Alice' Friend Person Tag Photo Tag 'Alice'`
	Plan3 : `('Alice' Tag Photo) ⋈`
	`('Alice' Friend Person Tag Photo)`
	Plan4 : `('Alice' Tag Photo Tag Person) ⋈`
	`('Alice' Friend Person)`

4	**Plan1** : `'Alice' (Advise Person)* Coauthor 'Bob'` **Plan2** : `'Bob' Coauthor (Person Advise)* 'Alice'`

Table 2: Examples of alternative query plans.

STEP 3: Bulk Synchronization (lines 9 to 14): The coordinator waits for messages from participant partitions *CurrentPartitions* signalling the end of local computation and global communication steps. The received messages *PartitionsMsgs* may contain two pieces of information: (1) *FullPaths*: represent a set of full matching graph paths if found. (2) *NextPartitions*: represent the set of partitions that will participate in the next BFS level. The partial matches are communicated only among partition servers; the coordinator does not send or receive any intermediate result. The TRAVERSEOPERATOR terminates when there is no participant partition at the next level (*CurrentPartitions*=∅).

In summary, each level of the *traverse* operator incurs (1) at most $2P$ control messages between the coordinator and the partitions to signal the start and end of the level, and (2) at most $P(P-1)$ data messages among the partitions to exchange intermediate results. The size of the control messages is small, and therefore, the workload of the coordinator is light without requiring any intensive computation or communication. The size of the data messages depends on each query, and it depends on the size of intermediate results that are distributed among the partition servers. The total levels of BFS is bounded by the length of the DFA if it does not contain a loop.

4. QUERY OPTIMIZATION

The declarative language of Horton+ makes query optimization possible and important. A declarative language only expresses the logic of a computation without describing its control flow: the query optimizer of Horton+ can choose to run a query among many implementations (or execution plans), preferably the one with the lowest cost. However, finding such an execution plan is not trivial, which requires an accurate cost model that estimates the cost of execution plans by taking into account of graph statistics and a computationally-efficient enumeration algorithm that finds the lowest-cost solution in a short amount of time. This section describes query optimizer of Horton+, which efficiently finds an optimal query execution plan (visiting the fewest number of graph nodes) for queries without closure operators. Moreover, we develop a heuristic algorithm to perform optimization for queries with closure operators, and we discuss how the optimizer takes communication cost into consideration for distributed graphs.

4.1 Space of Query Plans

The query optimizer takes a compiled query plan as input and outputs an efficient query execution plan. The output plan is represented as a tree with DFAs in the leave nodes and join operators as the intermediate nodes. To produce an efficient plan, the optimizer enumerates various execution plans, estimates their costs and returns the lowest-cost plan. Here we define the cost of an execution plan by estimating the total number of nodes it visits. The fewer the number of visited nodes, the more efficient the execution plan.

For a graph query $Q = \langle N_1, E_1, \cdots, N_i, E_i, N_{i+1}, \cdots, E_{k-1}, N_k \rangle$, with k node predicates and $k-1$ edge predicates, Horton+ query optimizer first considers k possible plans as follows: (1) One plan is to execute the query starting from N_1 to N_k. (2) Another plan is to execute the query in the reverse order, from N_k to N_1. (3) $k-2$ plans as dividing Q at node N_i, $1 < i < k$, into two subqueries as follows: (a) a subquery that starts from N_1 and ends at N_i, and (b) a subquery that starts from N_i and ends at N_k. The results from the two subqueries are joined to produce the final answer. Each of the two subqueries can be recursively optimized by considering its execution in the forward and reverse order, as well as further splitting into shorter subqueries. However, for simplicity illustration, we first describe a simple version of the query optimizer where the first subquery is executed in the forward order from N_1 to N_i while the second subquery is executed in the reverse order, from N_k to N_i. We present the complete recursive query optimizer in Section 4.6. A query can be executed in the forward or reverse order because each edge (whether directed or not) can be accessed from the its two nodes.

Example. Table 2 gives all non-recursive query plans that the query optimizer considers for queries Q_1 to Q_4. For example, Q_3 has four node predicates, and hence four possible plans as follows: (1) The forward order from N_1 to N_4, which finds the graph node for `Alice`, then finds photos in which `Alice` is tagged. From these photos, we find all persons tagged in any of these photos. Among these persons, we find the ones who are friends with `Alice`. (2) The reverse order, from N_4 to N_1, which finds the graph node for `Alice`, then all friends of `Alice`. For these friends, we find all photos in which they are tagged. Among these photos, we find the ones in which `Alice` is tagged. (3) A join at N_2, where we have two subqueries. The first subquery finds all photos in which `Alice` is tagged (the forward order from N_1 to N_2), while the second subquery finds all `Alice` friends, and then finds all photos in which `Alice` friends are tagged (the reverse order, from N_4 to N_2). The outputs of the two subqueries (set of photos) are joined to get the intersection. (4) A join at N_3, where we have two subqueries. The first subquery goes from `Alice` to all photos she is tagged in, and then all persons who are tagged in these photos (the forward order from N_1 to N_3), and the second subquery goes from `Alice` to all her friends (the reverse order, from N_4 to N_3). The outputs of the two subqueries are joined.

4.2 Graph Statistics

This section outlines four main *statistics* functions, $S(N_i)$, $T(N_i)$, $F(N_i, E_j, N_h)$, and $G(N_i, E_j, N_h)$ that are used to estimate the cost of each considered query plan.

***S(N$_i$)* and *T(N$_i$)*.** Given a node predicate N_i, $S(N_i)$ estimates the number of nodes that satisfy predicate N_i while $T(N_i)$ estimates the number of nodes that need to be visited to find the ones satisfying N_i. Here $S(N_i)$ represents the selectivity of a node predicate indicating the number of successful matches while *T(N$_i$)* represents the cost to find the successful matches. We use rules similar to those used in the query optimizer of relational database systems. If the node predicate is an id equality, indexed as a primary key, then $S(N_i) = T(N_i) = 1$. If the predicate is on a non-indexed field and one tenth of the nodes in this node type satisfying the predicate, then $S(N_i) = 0.1m$ and $T(N_i) = m$ where m is the total number of nodes of this node type. If a histogram is maintained, we can get a better accuracy on estimating $S(N_i)$, yet $T(N_i)$ depends on the index availability.

***F(N$_i$, E$_j$, N$_h$)* and *G(N$_i$, E$_j$, N$_h$)*.** Given two node predicates N_i and N_h and an edge predicate E_j, $F(N_i, E_j, N_h)$ estimates the number of nodes that are reachable from N_i through the edge predicate E_j and satisfy the predicate N_h while $G(N_i, E_j, N_h)$ estimates the number of nodes that need to be visited to find these nodes, i.e., the number of reachable nodes from N_i to the node type of N_h using the edge predicate E_j. Again, $F(N_i, E_j, N_h)$ is a measure of selectivity indicating the number of successful matches while $G(N_i, E_j, N_h)$ measures the cost to find the successful matches. $G(N_i, E_j, N_h)$ is computed by utilizing few statistics maintained for the number of edges of each type connected to each node type. This number is then divided by the selectivity of the predicate at node N_h to compute $F(N_i, E_j, N_h)$. For example, if N_i is of id `Alice`, then, we know that there is only one node satisfying predicate N_i. Then, if E_j is of type `Tag`, and we know from our statistics that `Alice` is tagged in 20 photos, then, we say that we will visit 20 nodes matching N_h so $F(N_i, E_j, N_h)$ = $G(N_i, E_j, N_h)$ = 20. However, if the predicate N_h includes only black & white photos, and we know that only 10% of the photos are black & white, then $F(N_i, E_j, N_h)$=2 while the total number of visited nodes is $G(N_i, E_j, N_h)$=20 as we have to visit all of the 20 photos in order to find the black & white ones.

Collecting Statistics. Since the underlying graph is partitioned and distributed among multiple graph partition servers, collecting the aforementioned statistics is performed as follows: (1) The graph directory service (DS) sends a statistics collection request to all graph partition servers. (2) Each graph partition server, in parallel, calculates the graph statistics $S(N_i)$, $T(N_i)$, $F(N_i, E_j, N_h)$, and $G(N_i, E_j, N_h)$ for all graph nodes and edges stored locally on that partition. (3) Then, each partition sends back a message to the graph directory service reporting its own local graph statistics. (4) Finally, the graph directory service aggregates the statistics from the partitions to generate the global graph statistics.

4.3 Objective Function and Cost Model

Given the space of k query plans for any query Q with k node predicates, it is the objective of the query optimizer to find the plan with the lowest estimated cost in terms of the number of visited nodes. Formally, Horton+ aims to minimize the objective function $Cost(Q[1,k])$, represented as:

$$Cost(Q[1,k]) = \min_{1 \le i \le k} (Cost(Q[1,i]) + Cost(Q[k,i]) + Join(Q,i))$$

where $Cost(Q[1,i])$ is the cost of executing a subquery of Q in the forward order from N_1 to N_i, $Cost(Q[k,i])$ is the cost of executing a subquery of Q in the reverse order, from N_k to N_i, and $Join(Q,i)$ is the cost of joining the results of these two subqueries. The trivial cases of $i = k$ and $i = 1$ correspond to the query plans with forward and reverse orders, respectively, where no join operation is involved, i.e., $Join(Q,i) = 0$.

Given the functions $S(N_i)$, $T(N_i)$, $F(N_i, E_j, N_h)$, and $G(N_i, E_j, N_h)$ (described in Section 4.2), $Cost(Q[1,i])$, $Cost(Q[k,i])$, and $Join(Q,i)$ can be calculated as follows:

***Cost(Q[1,i])*.** For the case when i=1, where $Cost(Q[1,i])$ is set to zero, corresponding to the execution of Q in the reverse order, from N_k to N_1. For the case when $i > 1$, we first need to visit $T(N_1)$ nodes to find the $S(N_1)$ nodes that satisfy the first node predicate N_1. Then, for the second node N_2, the cost is $S(N_1) \times G(N_1, E_1, N_2)$, which corresponds to the number of qualified nodes from N_1 multiplied by the number of nodes we visit to satisfy the predicate N_2. Then, for the third node N_3, we visit a total number of nodes $S(N_1) \times F(N_1, E_1, N_2) \times G(N_2, E_2, N_3)$, which corresponds to the number of qualified nodes from N_2, which is $S(N_1) \times F(N_1, E_1, N_2)$, multiplied by the number of nodes we need to visit to satisfy the predicate N_3, which is $G(N_2, E_2, N_3)$. The total number of visited nodes for $Q[1,i]$ is the sum of the number of visited nodes at each predicate N_j, which is formally presented as:

$$Cost(Q[1,i]) = \begin{cases} T(N_1) + S(N_1) \times G(N_1, E_1, N_2) + \\ S(N_1) \displaystyle\sum_{j=2}^{i} \left(G(N_j, E_j, N_{j+1}) \prod_{h=1}^{j-1} F(N_h, E_h, N_{h+1}) \right) & i > 1 \\ 0 & i = 1 \end{cases}$$

***Cost(Q[k,i])*.** Similar to Cost(Q[1,i], but Cost(Q[k,i]) estimates the cost execution in the reverse order. For the non-trivial case of $i < k$, we start by getting the number of visited nodes of type N_k as $T(N_k)$. Then, we follow the nodes in the reverse order, e.g., for node N_{k-1}, we visit $S(N_k) \times G(N_k, E_{k-1}, N_{k-1})$ nodes. For node N_{k-2}, we visit $S(N_k) \times F(N_k, E_{k-1}, N_{k-1}) \times G(N_{k-1}, E_{k-2}, N_{k-2})$, and so on. Formally:

$$Cost(Q[k,i]) = \begin{cases} T(N_k) + S(N_k) \times G(N_k, E_{k-1}, N_{k-1}) + \\ S(N_k) \displaystyle\sum_{j=i}^{k-2} \left(G(N_{j+1}, E_j, N_j) \prod_{h=j+1}^{k-1} F(N_{h+1}, E_h, N_h) \right) & i < k \\ 0 & i = k \end{cases}$$

***Join(Q,i)*.** A trivial case is when i=1 or i=k, where $Join(Q,i)$ is set to zero, indicating that the query plan corresponds to either the reverse or forward order, respectively. For the non-trivial case ($1 < i < k$), $Join(Q,i)$ is computed as the Cartesian product of the two sets involved in the join. The first set includes the estimated number of nodes satisfying all the predicates in the forward order from N_1 to N_i, which is: $S(N_1) \prod_{j=1}^{i-1} F(N_j, E_j, N_{j+1})$. Similarly, the second set includes the estimated number of nodes satisfying all the predicates in the reverse order from N_k to N_i, which is $S(N_k)$ $\prod_{j=i+1}^{k} F(N_j, E_{j-1}, N_{j-1})$. Formally:

$$Join(Q,i) = \begin{cases} S(N_1)S(N_k) \displaystyle\prod_{j=1}^{i-1} F(N_j, E_j, N_{j+1}) \prod_{j=i+1}^{k} F(N_j, E_{j-1}, N_{j-1}) & 1 < i < k \\ 0 & i = 1 \, OR \, i = k \end{cases}$$

4.4 Numerical Example

Figure 4-a gives examples of some collected statistics that are enough for the query optimizer to decide on the best execution plan for Q_1, Q_2, and Q_3 of Table 1. For simplicity and ease of illustration, we assume that $T(N_i)$ and $G(N_i, E_j, N_h)$ are equivalent to their counterparts $S(N_i)$ and $F(N_i, E_j, N_h)$. In our example, *S('Alice')* is set to one where there is only one node with id `Alice`. *S(Photo)* is set to one million, indicating the number of nodes of type `Photo` in the whole graph. *F('Alice', Friend, Person)* is set to 10 as `Alice` has only 10 friends of type `Person`. Statistics are bi-directional as *F(Person, Friend, 'Alice')* is set to 150 as the *average* number of friends for each person. *F(Person, Tag, Photo)* and *F(Photo, Tag, Person)* are set to 20 and 3 as the *average* number of "photos per persons" and "persons tagged in a photo", respectively.

Figure 4-b gives the cost of each query plan for Q_1, Q_2, Q_3 based on the statistics of Figure 4-a. As an example, we describe the optimal plan of Q_3 as follows:

***Q*$_3$.** Plan 4 (Join at the third node N_3) has the lowest cost, computed as the sum of three parts: (a) The cost of going

T(Alice) = S(Alice)	1
T(Bob) = S(Bob)	1
T(Photo) = S(Photo)	1M
G,F(Alice, Friend, Person)	10
G,F(Alice, Tag, Photo)	50
G,F(Bob, Tag, Photo)	2
G,F(Person, Friend, Alice)	150
G,F(Person, Tag, Photo)	20
G,F(Photo, Tag, Alice)	50
G,F(Photo, Tag, Bob)	2
G,F(Photo, Tag, Person)	3

(a) Statistics

Q	P	Cost
1	1	1 + 1×50 + 1×50×2=151
	2	1 + 1×2 + 1×2×50=103
	3	(1+1×50) +(1+1×2)+1×1×50×2=154
2	1	1 + 1×10 + 1×10×20=311
	2	1M + ……..
	3	1M + ……..
3	1	1+1×50+1×50×3+1×50×3×150=22701
	2	1+1×10+1×10×20+1×10×20×50=10211
	3	(1+1×50)+(1+1×10+1×10×20)+1×1×50×10×20=10262
	4	(1+1×50+1×50×3)+(1+1×10)+1×1×50×3×10=1712

(b) Query Plan Cost for Q_1, Q_2, Q_3 (optimal plans are shaded)

Figure 4: Example of statistics and cost of query plans.

in the forward order from N_1 to N_3 as *Cost(1,3)* = $T(N_1)$ + $S(N_1)$ $G(N_1, E_1, N_2)$ + $S(N_1)$ $F(N_1, E_1, N_2)$ $G(N_2, E_2, N_3)$, which is equivalent to: *T('Alice')* + *S('Alice') G('Alice',Tag,Photo)* + *S('Alice') F('Alice',Tag,Photo) G(Photo,Tag,Person)* = 201. (b) The cost of going in the reverse order, from N_4 to N_3, as *Cost(4,3)* = $T(N_4)$ + $S(N_4)$ $G(N_4, E_3, N_3)$, which is equivalent to: *T('Alice')* + *S('Alice') G('Alice',Friend,Person)* = 11. (c) The cost of joining the results from the two previous parts as *Join(3)* = $S(N_1)$ $S(N_3)$ $F(N_1, E_1, N_2)$ $F(N_2, E_2, N_3)$ $F(N_4, E_3, N_3)$, which is equivalent to: *S('Alice') S('Alice') F('Alice',Tag,Photo) F(Photo,Tag,Person) F('Alice',Friend,Person)* = 1500. Finally, the total cost of this plan is the sum of these three costs as 201+11+1500 = 1712.

4.5 Query Optimization Algorithm

Algorithm 3 gives the pseudo code of the query optimizer. The input to the algorithm is a query Q with k node predicates and $k-1$ edge predicates. The output is a *join pointer* on where to split the query to achieve the best performance in terms of the number of visited nodes. A *join pointer* value of k or 1 indicates that the best query plan is the forward or reverse order, respectively, without any join. A basic algorithm to find the lowest cost would compute the cost of k different execution plans individually where each plan can cost up to $O(k^2)$ number of addition and multiplication operations, which gives a total cost of $O(k^3)$, where k is the number of nodes in the input path query. Here, we present an algorithm with a total cost of only $O(k)$, which exploits common subcomputations to reduce computational complexity.

The algorithm has three main parts: The first part (Lines 2 to 8) incrementally fills four arrays, *CostF*, *JoinF*, *CostR*, and *JoinR*, each of size k. An item $i > 1$ in any of the two arrays, *CostF[i]*, *JoinF[i]*, maintains the cost of query evaluation in the forward order from N_1 to N_i as *CostF[i]*= $T(N_1)$ + $S(N_1)$ $G(N_1, E_1, N_2)$ + $S(N_1)$ $\sum_{j=2}^{i}$ $(G(N_j, E_j, N_{j+1})$ $\prod_{h=1}^{j-1} F(N_h, E_h, N_{h+1}))$, and if there is a join at node i, the additional cost is *JoinF[i]* $=$ $S(N_1)$ $\prod_{h=1}^{i} F(N_h, E_h, N_{h+1})$. Similarly, an item $i < k$ in any of the two arrays, *CostR[i]*, *JoinR[i]*, maintains the cost of query evaluation in the reverse order, from N_k to N_i, and part of the join cost should we decide to join at node i. The second part of the algorithm (Lines 9 to 14) computes the cost of forward and reverse execution order of Q. As the costs are computed incrementally, the forward and reverse order costs are computed by adding two terms: (1) The cost encountered to reach to node N_{k-1} and N_2, which is *CostF[k-1]* and *CostR[2]*, respectively, and (2) The number of nodes to visit to reach to N_k and N_1, which is computed as the number of paths we have, i.e., join cost, till N_{k-1} and N_2 (*JoinF[k-1]* and *JoinR[2]*) multiplied by the number of output nodes of each path to reach N_k and N_1 ($G(N_{k-1}, E_{k-1}, N_k)$ and $G(N_2, E_1, N_1)$), respectively. The minimum of the forward and reverse order costs is set as the current optimal plan with the *join pointer* set as k and 1, respectively. The third part of the algorithm (Lines 15 to 18) iterates over all nodes to compute the total cost of joining at each node i as *CostF[i]+CostR[i]+JoinF[i]*×*JoinR[i]*. The value of i that corresponds to the minimum total cost is returned as a *join pointer*.

For Q_1 in Table 1, *JoinF*={1,50,-}, *CostF*={1,51,-}, *JoinR*={-,2,1}, *CostR*={-,3,1}, which results in *ForwardCost* = 51 + 50×2 = 151 and *ReverseCost*=3 + 2×50 = 103. The cost of joining at node N_2 = 51 + 3 + 50×2 = 154. Hence, *join pointer* is set to 1.

Algorithm 3 Query optimizer

```
1: Function QUERYOPTIMIZER($Q = \langle N_1, E_1, \cdots, E_{k-1}, N_k \rangle$)
2: JoinF[1] ← $S(N_1)$; CostF[1] ← $T(N_1)$;
3: JoinR[k] ← $S(N_k)$; CostR[k] ← $T(N_k)$;
4: for $i = 2$ to $k-1$ do
5:     JoinF[i] ← JoinF[i-1] × $F(N_{i-1}, E_{i-1}, N_i)$
6:     CostF[i] ← CostF[i-1] + JoinF[i-1] × $G(N_{i-1}, E_{i-1}, N_i)$
7:     JoinR[k-i+1]←JoinR[k-i+2] ×$F(N_{k-i+2}, E_{k-i+1}, N_{k-i+1})$
8:     CostR[k-i+1] ← CostR[k-i+2] + JoinR[k-i+2] ×$G(N_{k-i+2}, E_{k-i+1}, N_{k-i+1})$
9: ForwardCost ← CostF[k-1] + JoinF[k-1] × $G(N_{k-1}, E_{k-1}, N_k)$
10: ReverseCost ← CostR[2] + JoinR[2] × $G(N_2, E_1, N_1)$
11: if ForwardCost < ReverseCost then
12:     MinCost ← ForwardCost ; JoinPointer ← $k$;
13: else
14:     MinCost ← ReverseCost; JoinPointer ← 1;
15: for $i = 2$ to $k-1$ do
16:     TotalCost ← CostF[i] + CostR[i] + JoinF[i] × JoinR[i];
17:     if TotalCost < MinCost then
18:         MinCost ← TotalCost; JoinPointer ← $i$;
19: Return JoinPointer;
```

4.6 Query Plans with Recursive Joins

We have described a simple version of Horton+ query optimizer that only considers splitting a given query into two subqueries. However, the full version of Horton+ query optimizer considers recursive splits of subqueries and produces an *optimal* execution plan based on the graph statistics. More specifically, it takes each subquery and recursively considers it for optimization, i.e., a subquery can be evaluated in the forward or reverse order, or can be split again to another two subqueries. The output of the recursive query optimization is a query execution plan represented as a tree: Each leaf node is a DFA representing a traversal of a subquery, and each intermediate node is a join operator that joins the results of two subqueries. Denoting $RCost(Q(p,q))$ as the cost of a query (or subquery) using recursive optimization where $p \leq q$, we present the recursive formulation of the query optimizer:

$$
\begin{aligned}
&RCost(Q(p,q)) = \\
&\quad \min\{SCost(Q(p,q)), SCost(Q(q,p)), \\
&\quad \min_{p<i<q}\{RCost(Q(p,i)) + RCost(Q(i,q)) + Join(Q(p,i), Q(i,q))\}\}
\end{aligned}
$$

Here, $SCost(Q(p,q))$ and $SCost(Q(q,p))$ are the cost of evaluating the query $Q(p,q)$ in the forward and reverse orders respectively, and $Join(Q(p,i), Q(i,q))$ is the cost of joining the subqueries $Q(p,i)$ and $Q(i,q)$. We present their formulation as follows.

$$
SCost(Q[p,i]) =
$$

$$
\begin{cases}
T(N_p) + S(N_p) \times G(N_p, E_p, N_{p+1}) + & \\
S(N_p) \displaystyle\sum_{j=p+1}^{i} \left(G(N_j, E_j, N_{j+1}) \prod_{h=1}^{j-1} F(N_h, E_h, N_{h+1}) \right) & p < i \\
0 & p = i
\end{cases}
$$

$$Join(Q(p,i),Q(i,q)) =$$

$$\begin{cases} S(N_p)S(N_q)\prod_{j=p}^{i-1} F(N_j,E_j,N_{j+1}) \quad \prod_{j=i+1}^{q} F(N_j,E_{j-1},N_{j-1}) & 1 < i < k \\ 0 & i = 1\, OR\, i = k \end{cases}$$

The cost of solving $RCost(Q(1,k))$ naively is $\Omega(k!)$, where k is the number of query node predicates. We use a *dynamic programming* framework to solve the problem efficiently: it computes and stores the optimal solutions for subqueries and uses those to construct the optimal solution for a bigger problem. Dynamic programming effectively reduces the computational cost of obtaining an *optimal* execution plan to $O(k^3)$, which is rather affordable as query length is often not that large (even long queries have length under 20 in most cases). Recursive splits can be mostly beneficial when there are many selective nodes within a long query.

4.7 Closure Operators

Estimating the cost of a query with closure operators is complex because a query optimizer does not know the number of recursive steps a query would take to complete without actually running the query. Thus, we develop a heuristic algorithm: (1) it optimizes the non-closure part of a query by exploring different traverse orders and join sequences using the techniques we presented earlier, and (2) it further exploits the closure part of the query with different number of recursive steps. This algorithm includes three phases. While illustrating the phases, we use an example query $N_1 - E_2 - N_2 - (E_3 - N_3)^* - E_4 - N_4 - E_5 - N_5$, and we assume that we consider k number of recursive steps where $k = 0, 1, 2, 3$.

(1) For each recursive level k, we remove the closure operator and expand the query according to the value of k. For example, with $k = 2$, our example query has a form of $Q(k = 2) = N_1 - E_2 - N2 - (E_3 - N_3 - E_3' - N_3') - E_4 - N_4 - E_5 - N_5$. For the expanded query instance, we compute a good plan, denoted as plan$_k$. The plan is computed similarly as in Section 4.6, however, we exclude those plans that would perform a join operation inside the recursive block, e.g., $E_3 - N_3 - E_3' - N_3'$ is the recursive block for Q(k=2). In other words, we treat the recursive block as an atomic unit: we can execute it in the forward or reverse order but we do not perform any join inside it. For the other parts of the query, we still consider the join operator based on the cost estimates.

(2) If all the plans, plan$_k$ for $k = 0, 1, 2, 3$, have equivalent execution sequence, we use this sequence to execute the recursive query. Here we define two plans have equivalent execution sequence if they have join on the same node predicates and have the same evaluation order for the same subqueries.

(3) However, if for different k values, their optimized plans have different execution sequences, we estimate the cost of using each "local optimal" execution sequence in the recursive query, and we call this cost TotalCost(plan$_k$) for a given plan$_k$. Among all plan$_k$ where $k = 0, 1, 2, 3$, we find the plan with the minimum TotalCost(plan$_k$), and use plan$_k$ for the recursive query.

Example. We give an example on how to compute TotalCost(plan$_k$). Suppose that $k = 1$ and plan$_1$ is to execute the query from left to right. The total cost of applying plan$_1$ to the recursive query is as follows:

$$\begin{aligned} TotalCost(plan_1) \;=\; & SCost(Q(k=0)) + SCost(Q(k=1)) \\ & + SCost(Q(k=2)) \\ & - 2SCost(N_1 - E_2 - N_2)\,. \end{aligned}$$

We remove the additional sequential cost of processing the subquery Q($N_1 - E_2 - N_2$) because this cost is incurred only once in the recursive execution.

4.8 Optimization for Distributed Execution

Horton+ takes into account the communication cost when optimizing the input query. It distinguishes between: (1) local edge: two nodes of a local edge reside on the same graph partition, and (2) remote edge: two nodes of a remote edge are stored on different graph partitions. Horton+ can assign higher cost for accessing remote edges and lower cost for local edges. We achieve it by incorporating the local/remote information into graph statistics thus influencing our cost estimation and final decision. For example, as described in Section 4.2, $G(N_i, E_j, N_h)$ estimates the number of nodes that need to be visited through the edge predicate E_j to satisfy the predicate N_h. We can revise its cost to reflect the cost difference of remote and local edge accesses with additional statistics P(N$_i$, E$_j$, N$_h$) that defines the probability of remote edges for edge predicates E$_j$. Supposing that the cost of remote and local access is c : 1, the revised cost of $G(N_i, E_j, N_h)$ is $G'(N_i, E_j, N_h) = c \times G(N_i, E_j, N_h) \times P(N_i, E_j, N_h) + G(N_i, E_j, N_h) \times (1 - P(N_i, E_j, N_h))$. By using the new statistics $G'(N_i, E_j, N_h)$, we apply the same optimization procedure as described earlier to decide efficient query plan considering communication costs.

5. EXPERIMENTAL EVALUATION

This section presents an experimental evaluation of Horton+ [38]. Our objective is to assess three aspects: system efficiency (query optimization), scalability (distributed execution), and usability (declarative querying). We also provide a comparison with Giraph [19]; a graph processing system built on top of Hadoop. Horton+ is implemented in C# in 30K lines of code. The implementation includes the client interfaces, query language parser and compiler, query optimizer, and distributed query processor. We use two graph types:

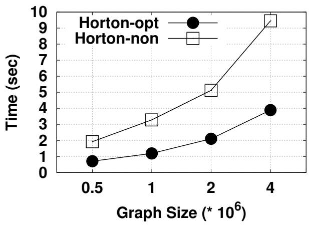
(1) We use a *real graph* from a software collaboration system, called Codebook [4], which models software engineers and their software artifacts, including source code, bug reports, projects, and their relationships. The graph has 2,910,535 nodes, 13,612,406 edges, 8 node types, and 11 edge types. It is generated by crawling multiple data sources including source code repositories, and employee directory and document databases. Each node and edge is associated with a large number of attributes. The graph data is represented natively in main memory as C# objects. The memory footprint of the graph is around 12 GB, including object overheads and the intermediate results while evaluating queries.

(2) We generate *synthetic graphs* with different sizes using the RMAT graph generator [5] that produces scale-free graphs. The graph schema including the types and attributes of nodes and edges are set to mimic the real graph schema. The node and edge types and attributes are generated using the Zipf distribution which models the popularity of attribute values, and we set the number of edges to five times the number of nodes.

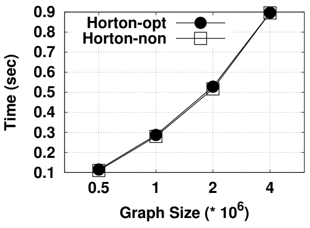
Workload. There is no standard benchmark for reachability queries. We, therefore, characterize the queries from Codebook, and classify them into four categories. For each category, we show one of most frequent queries in Table 3. (1) *Short queries* have a small number of predicates as in query Q_1. Since the query length is short, the query optimizer searches a small space. (2) *Selective queries* have one or more selective predicates. For example query Q_2 contains two id predicates `'Dave'` and `'Tim'`. Due to the high selectivity, selective queries traverse a small number of paths to compute the final answer. (3) *Report queries* return a large result set, such as query Q_3. Report queries are the most expensive to execute. (4) *Closure queries* require recursive graph traversal. Query

Query	Query in Plain English	Query in Horton+
Short (Q_1)	Find the person who committed checkin 400 and the WorkItemRevisions it modifies	`Person-Committer-Checkin{id=400}- Modifies-WorkItemRevision`
Selective (Q_2)	Find Dave's checkins that modified a WorkItem create by Tim	`Person{id='Dave'}-Committer-Checkin -Modifies-WorkItem -CreatedBy-'Tim'`
Report (Q_3)	For each checkin, find the person (along with his manager) who committer it as well as all the work items (along with their WebURLs that are modified by that checkin)	`Person-Manages-Person- Committer-Checkin-Modifies -WorkItemRevision-Modifies-WorkItem -Links-WebURL`
Closure (Q_4)	Retrieve all checkins that any employee in Dave organizational chart (working under him) committed.	`Person{id='Dave'} (-Manages-Person)*-Checkin`

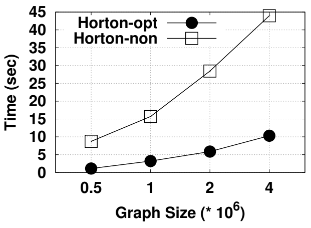
Table 3: The graph queries used in the experiments.



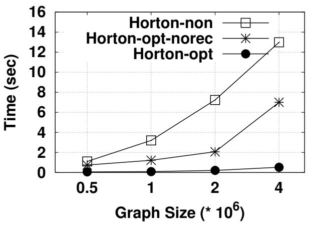
(a) Short Query.



(b) Selective Query.



(c) Report Query.



(d) Report Query (recursive).

Figure 5: Impact of query optimization on the query execution time (using the synthetic graphs) on a single server.

Q_4 is a closure query and it retrieves the management hierarchy of a person using Kleene star.

Performance Metrics. Our main performance metric is the query execution time. We also examine the computation and communication costs to process the queries.

Experimental Environment. All experiments are run on a cluster of 16 graph partition servers plus two servers as the coordinator and client. Each server has an Intel QuadCore 2.9 GHz CPU, 16 GB RAM, and runs Windows Server 2008. The servers are connected by a Gigabit Ethernet switch.

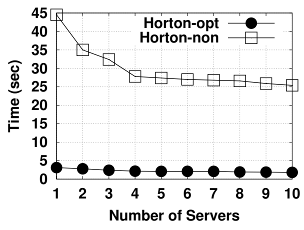
5.1 Efficiency (Query Optimization)

In this section, we study the benefits of the query optimizer on reducing query execution time for a graph deployed first on a single server, and then on multiple servers. We run two versions of Horton+: `Horton-opt` is the full version of Horton+ including its query optimizer. `Horton-non` represents Horton+ with the optimizer turned off, i.e., queries are executed in the forward order. The experimental results demonstrate that optimization reduces the latency of many queries by a factor of 5 — 15 times. Moreover, the larger the graph size, the higher the optimization benefits.

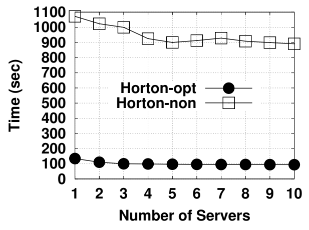
5.1.1 Deployment on a Single Graph Server

Short query Q_1. Figure 5(a) shows the performance of executing query Q_1 on synthetic graphs. The X-axis is the size of the synthetic graph (0.5, 1, 2, and 4 million nodes), and the Y-axis is execution time. `Horton-opt` outperforms `Horton-non` for all graph sizes because `Horton-opt` splits Q_1 at the middle selective node predicate `Checkin{id=400}` and processes two subqueries. The query execution time in both `Horton-non` and `Horton-opt` becomes higher with increasing graph size because the query execution engine visits more graph nodes when the graph size becomes larger. The benefits of optimization become more significant with the increase in graph size.

Selective query Q_2. Figure 5(b) shows the results of running query Q_2. `Horton-non` and `Horton-opt` give the same performance, as they both execute the forward execution plan. The results also show that the overhead of running the query optimizer is almost negligible, compared with the total query execution time. The optimized and non-optimized plans for Q_4 are the same with equal execution time. We omit Q_4 as it is similar to Figure 5(b).



(a) Short Query.

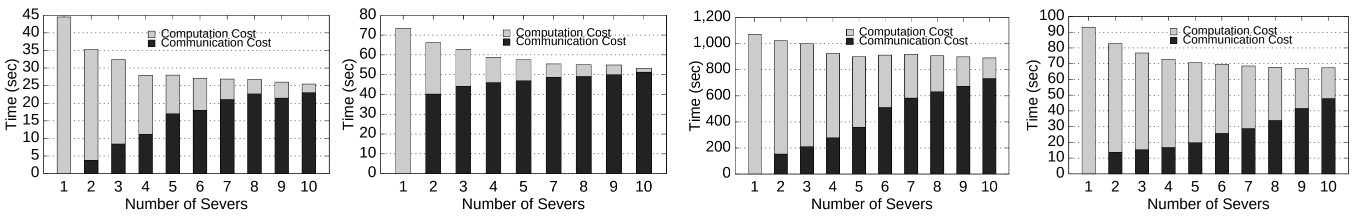


(b) Report Query.

Figure 6: Impact of optimization on execution time (using the real graph), number of servers varies from 1 to 10.

Report query Q_3. Figure 5(c) shows the performance of query Q_3. The optimizer provides lower execution time, and the benefits increase with the graph size.

Recursive split for query Q'_3. Figure 5(d) shows interesting results: We change Q_3 into Q'_3 by adding two predicates to the third (`CheckIn{id=390}`) and fourth (`WorkItemRevision{id=610}`) node predicates. `Horton-opt` exploits these predicates to reduce the execution time by a factor of 13. `Horton-opt` chooses a query plan that includes recursive splits at two nodes. First, `Horton-opt` splits the query at the third node (`CheckIn{id=390}`) into two subqueries. Next for the second subquery, `Horton-opt` performs a recursive split, where this second subquery is split at the fourth node (`WorkItemRevision{id=610}`). The recursive split is the main reason behind the impressive performance of `Horton-opt` for Q'_3. To measure the benefits of recursive splits, we run a third version of Horton+, termed `Horton-opt-norec`, which uses the same optimizer but without recursive splits; thus the optimized query plan contains at most one split. `Horton-opt-norec` produces a plan for Q'_3 with a single split at the third node. Figure 5(d) shows that



(a) Short Query. (b) Selective Query. (c) Report Query. (d) Closure Query.

Figure 7: Query execution time (using the real graph) while varying number of partition servers from 1 to 10.

`Horton-opt` yields up to 7 times better performance than that of `Horton-opt-norec`, which shows the benefits of employing recursive splits in the query optimizer.

5.1.2 Deployment on Multiple Graph Servers

Figures 6(a) and 6(b) depict the query execution time of `Horton-opt` and `Horton-non` for queries Q_1 and Q_3 executed over the real graph, while varying the number of servers from 1 to 10 over the X-axis. We omit the results of Q_2 and Q_4 since the optimized query plans are the same as the non-optimized plans.

For query Q_1, `Horton-opt` consistently achieves from 12 to 14 times better performance compared with `Horton-non` because in `Horton-opt` the optimizer splits Q_1 at the middle node predicate `CheckIn{id=400}` and executes the two subqueries. The optimized plan is substantially more efficient than the forward execution of Q_1.

For query Q_3, `Horton-opt` consistently outperforms `Horton-non` by a factor of 8 to 10 times as `Horton-Opt` uses an optimized plan that splits the query at the middle node predicate `CheckIn`. These results show that query optimizer produces good plans, suitable for deployments both on a single server and on multiple servers.

5.2 Scalability (Distributed Processing)

We study the performance of distributed query processing in two cases. First, we use the real graph which fits in a single server, and study the distributed execution overhead with the number of servers. Second, we use a large synthetic graph that does not fit in a single server to show query processing times.

(1) Real Graph. Since the real graph fits in the main memory of single server, this constitutes a challenging environment for evaluating a distributed system: (1) There is no benefit from the aggregated main memories of the servers, and (2) the communication and synchronization inefficiencies are emphasized. A single server could be more efficient as it incurs no messaging overhead.

We vary the number of partition servers from 1 to 10. Figures 7(a) to 7(d) show the execution time of queries Q_1, Q_2, Q_3, and Q_4. The query execution time has two components: computation time and communication time. The computation time is time used for local computations at the servers, and the communication time is the time spent in message passing among the servers. The performance of Q_1, Q_2, Q_3, and Q_4 improves with increasing the number of servers. The improvement comes from executing the query in parallel on more servers, reducing the parallel computation time component as the number of servers increases as depicted in the figures. More graph partitions lead to a reduction in the number nodes and edges per partition, further reducing the amount of local computation per server.

However, the performance gain shows diminishing returns because the communication time increases with the number of servers. The communication time is zero for a single server, and it increases as more servers are added because more messages are exchanged among the graph servers during query execution. The communication cost is dominated by the messages exchanged among the graph servers during the global communication step performed by the *traverse* operator. These results show that the system is efficient, and query execution time improves with the number of servers even if the graph fits in the memory of one server.

Query	Total execution	Communication	Computation
Short Query(Q_1)	47.588 sec	0.723 sec	46.865 sec
Selective Query(Q_2)	6.294 sec	0.693 sec	5.601 sec

Table 4: Execution time for 1024 million nodes, 5120 million edges synthetic graph deployed on 16 partition servers.

(2) Synthetic Graph. We use a synthetic graph with 1024 million nodes and 5120 million edges with an aggregate memory footprint of 145 GB, partitioned on a cluster of 16 servers. This experiment shows that Horton+ processes queries over graphs that do not fit on a single server, and it exploits multiple servers to execute a single query in parallel.

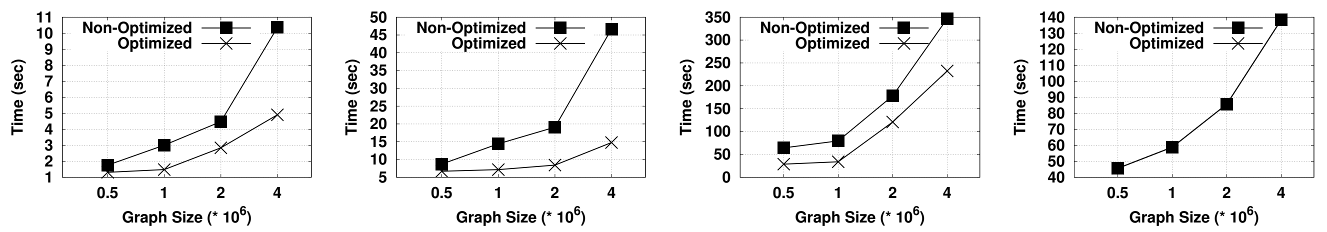
Table 4 shows the execution time of queries Q_1 and Q_2. The execution time of Q_1 is approximately 48 seconds. Q_1 execution plan splits the query into two subqueries at the (`CheckIn{id=400}`) node predicate. Both subqueries are executed separately and the their outputs are joined to form the final answer. Even though the graph is partitioned on 16 servers, only 1.5% of the query execution time is spent in communication and the remaining 98.5% is spent for local computation. This is in contrast to the findings we observe in Figures 7(a) and 7(c), where the communication cost is dominant for only 10 servers. The reason is that with larger graph sizes, partitioning the graph among 16 servers provides enough work for each server to parallelize query processing.

Query Q_2 shows similar benefits to Q_1, as the majority of the time is spent in computations rather than in communication. The computation cost comes mainly from traversing the graph elements at the graph partition servers during the local computation steps performed by the *traverse* operator. These results show that Horton+ efficiently parallelizes execution over a cluster of servers.

5.3 Usability (Declarative Queries)

Writing a procedural program takes more effort compared with expressing an equivalent declarative query, particularly for novice users. The procedural program is harder to write, debug, and maintain. Moreover, expressing a query directly into a procedural program may not lead to an efficient execution, and it is well-known that procedural programs are hard to optimize automatically.

We support this argument with anecdotal evidence: Figure 9 depicts query Q_3 (from Table 3) in a procedural language (i.e., `Java`) in Giraph. We make two observations: (1) The procedural program is longer and more complex. (2) Comparable graph systems such as Giraph [19], Pregel [30], and Trinity [39] require



(a) Short Query. (b) Selective Query. (c) Report Query. (d) Closure Query.

Figure 8: Impact of optimization on execution time in *Giraph* (using the pseudorandom synthetic graph) on a 10 servers cluster.

```
public void compute(Iterable<Text> m) throws IOException{
 Text message = m.next().get(); int st = getSuperstep();
 if (st == 0 && getValue().get() == "Person") {
   for (Edge<LongWritable, Text> edge : getEdges())
    if (edge.getValue().get() == "Manages")
     sendMessage(edge.getTargetVertexId(),formatMsg(m));
 } else if (st == 1 && getValue().get() == "Person") {
   for (Edge<LongWritable, Text> edge : getEdges())
    if (edge.getValue().get() == "Committer")
     sendMessage(edge.getTargetVertexId(),formatMsg(m));
 } else if (st == 2 && getValue().get() == "Checkin") {
   for (Edge<LongWritable, Text> edge : getEdges())
    if (edge.getValue().get() == "Modifies")
     sendMessage(edge.getTargetVertexId(),formatMsg(m));
 } else if (st == 3
          && getValue().get() == "WorkItemRevision") {
   for (Edge<LongWritable, Text> edge : getEdges())
    if (edge.getValue().get() == "Modifies")
     sendMessage(edge.getTargetVertexId(),formatMsg(m));
 } else if (st == 4 && getValue().get() == "WorkItem") {
   for (Edge<LongWritable, Text> edge : getEdges())
    if (edge.getValue().get() == "Links")
     sendMessage(edge.getTargetVertexId(),formatMsg(m));
 } else if (st == 5 && getValue().get() == "WebURL") {
 }
 voteToHalt();
}
```

Figure 9: *Giraph* Program (Java pseudo code) for Query Q_3.

programmers to write procedural programs with explicit communication messages for queries like Q_3. The procedural program for Q_3 is likely to have a high execution time, similar to the time of the non-optimized execution plan in Figure 5(c), which is almost an order of magnitude higher than the optimized plan. Furthermore, writing an efficient program requires the user to be closely familiar with both the underlying graph and the execution engine; for example it is challenging for the programmer to split a query into multiple subqueries at the right node predicates and to write code to join the outputs in the right order. Horton+, on the other hand, allows users to declaratively express queries, and optimizes them.

5.4 Comparing Horton+ with Giraph

This section compares Horton+ with Apache Giraph, which is a large-scale graph processing system built on-top of Hadoop. Both Horton+ and Giraph store the graph in the main memory of a cluster of servers. Also, both employ the bulk synchronous parallel execution paradigm to process queries in parallel over the distributed graph. By default, Giraph loads the graph from the Hadoop HDFS file system, and then deploys it to the cluster for each submitted graph processing job. Giraph users write queries as procedural Java programs which are executed directly without optimization, whereas Horton+ maintains graph statistics to optimize the queries.

We write a procedural program for each query in Table 3. For example, Figure 9 shows part of the Java code equivalent to Q_3. The actual code is longer, and it contains additional lines for reading graph data from and writing the results to Hadoop HDFS. In addition, we use the Horton+ optimizer to generate an optimized query plan and we write an optimized program in Giraph emulating the optimized plan.

We study the execution time of these procedural programs on Giraph. Our objective is not to compare the performance of Giraph with Horton+ directly because they use different software stacks of managed and unmanaged components (such as JVM and CLR/.Net framework) and different communication primitives and libraries. Instead, our objective is to (1) put Horton+ performance in perspective, and (2) show that Horton+ optimization strategies can also be used to guide writing better procedural programs for Giraph.

We deploy graphs of sizes 500K, 1M, 2M, and 4M nodes (number of edges is five times the number of nodes) generated using the the pseudorandom synthetic graph benchmark provided by Giraph, over 10 servers in a Hadoop cluster running `hadoop-0.20.203.0` with 30 mappers. Figures 8(a) to 8(d) show the performance of the procedural programs. We compare the plain Giraph Java programs (labelled `Non-Optimized`) with the Java programs written following the Horton+ optimized plans (labelled `Optimized`). Horton+ produces optimized plans different from prior plans because the graph statistics of the pseudorandom synthetic graph are quite different from the statistics of the prior real and synthetic graphs. The `Optimized` programs outperforms the `Non-Optimized` for Q_1, Q_2, Q_3 because the `Optimized` programs traverse fewer graph nodes and edges, incurring less computation and communication overheads. For Q_4, both `Optimized` and `Non-optimized` achieve the same performance as the optimized plan is equivalent to the forward plan. These results show that the Horton+ optimization techniques are general, and its optimizer can provide guidance in writing more efficient procedural programs for Giraph.

6. RELATED WORK

Graph Query Languages. Graph query languages are based on either regular expressions [10, 11, 22], SQL-like languages [3, 36, 40], or a procedural languages [23]. Horton+ uses a formal declarative query language to express reachability queries, and more importantly it provides an efficient distributed execution engine to execute its declarative queries.

Graph Processing Algorithms. In-memory graph processing algorithms include computationally-intensive algorithms, e.g., graph mining [32, 42], dense subgraphs [18, 35], and pattern matching [14], where the emphasis is on having reasonable latency for problems that are likely to be NP-complete. Online graph algorithms support simple graph queries, e.g., shortest path queries [17, 47], reachability queries [8, 15, 24], smaller versions of complex queries, e.g., pattern matching queries [15, 50], or approximate queries on a streaming environment [2, 49]. Horton+ focuses on processing reachability queries over a partitioned graph, and provides a query language, optimizer and distributed execution engine.

Distributed Graph Query Processing. Research in distributed graph query processing has focused on either leveraging the MapReduce paradigm [12] to support graph operations [7, 9, 25] or building distributed computation models for graph queries, e.g., Pregel [30], Trinity [39], GraphChi [20], and PowerGraph [29]. Horton+ is different because (1) it supports a declarative query language and (2) it optimizes query execution. In contrast, systems like Pregel provide an API for developers to write procedural programs, which are harder to write, debug, maintain and optimize.

Graph Query Optimization. Existing graph query optimization techniques focus on either building index structures [48, 50], or on developing selectivity estimation modules for certain graph queries [33, 49]. These techniques are complementary to Horton+, and it can employ such techniques. Several optimization techniques on tree structures, such as for XML documents, are not applicable to graphs, which contain cycles.

Graph Libraries. Graph libraries provide various graph algorithms within a single framework [6, 21, 26, 28, 41]. Horton+ provides a query language rather than a set of graph algorithms. It consists of multiple components including compiler, optimizer, and distributed query processor.

7. CONCLUSION

This paper presents the design, implementation and evaluation of Horton+, a distributed system for processing reachability queries on a partitioned attributed multi-graph. The system has a declarative query language, distributed query processor, and query optimizer. The query language expresses reachability queries and supports closures and predicates on the attributes of nodes and edges. The distributed query processor executes a query plan using three algebraic graph operators, *select*, *traverse* and *join* to find paths that match the user query. The query optimizer employs a cost model and selectivity estimation techniques to rewrite the query plan. Experiments on real and synthetic graphs on a cluster of servers show system the scalability and efficiency.

8. REFERENCES

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, 2006.

[2] C. C. Aggarwal, Y. Li, P. S. Yu, and R. Jin. On Dense Pattern Mining in Graph Streams. *PVLDB*, 3(1):975–984, 2010.

[3] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *ICDE*, 1998.

[4] A. Begel, K. Y. Phang, and T. Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *ICSE*, 2010.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, Apr. 2004.

[6] A. Chan, F. K. H. A. Dehne, and R. Taylor. CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *IJHPCA*, 19(1):81–97, 2005.

[7] R. Chen, X. Weng, B. He, and M. Yang. Large Graph Processing in the Cloud (Demo). In *SIGMOD*, 2010.

[8] Y. Chen and Y. Chen. An Efficient Algorithm for Answering Graph Reachability Queries. In *ICDE*, 2008.

[9] J. Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[10] M. P. Consens and A. O. Mendelzon. GraphLog: a Visual Formalism for Real Life Recursion. In *PODS*, 1990.

[11] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A Graphical Query Language Supporting Recursion. In *SIGMOD*, 1987.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[13] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-law Relationships of the Internet Topology. In *ACM SIGCOMM*, 1999.

[14] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental Graph Pattern Matching. In *SIGMOD*, 2011.

[15] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding Regular Expressions to Graph Reachability and Pattern Queries. In *ICDE*, 2011.

[16] Facebook Graph Search. `https://www.facebook.com/about/graphsearch`.

[17] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational Approach for Shortest Path Discovery over Large Graphs. *PVLDB*, 5(4):358–369, 2011.

[18] D. Gibson, R. Kumar, and A. Tomkins. Discovering Large Dense Subgraphs in Massive Graphs. In *VLDB*, 2005.

[19] Giraph. `http://incubator.apache.org/giraph/`.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[21] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *POOSC*, 2005.

[22] R. H. Güting. GraphDB: Modeling and Querying Graphs in Databases. In *VLDB*, 1994.

[23] H. He and A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *SIGMOD*, 2008.

[24] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing Label-constraint Reachability in Graph Databases. In *SIGMOD*, 2010.

[25] U. Kang, D. H. Chau, and C. Faloutsos. Mining Large Graphs: Algorithms, Inference, and Discoveries. In *ICDE*, 2011.

[26] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: A Scalable and General Graph Management System. In *KDD*, 2011.

[27] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The Web as a Graph: Measurements, Models, and Methods. In *Conference on Computing and Combinatorics, COCOON*, 1999.

[28] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, 1993.

[29] A. Kyrola, G. Blelloch, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[30] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, 2010.

[31] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random Graph Models of Social Networks. *Proceedings of the National Academy of Sciences of the USA*, 99(1):2566–2572, Feb. 2002.

[32] J. Pei, D. Jiang, and A. Zhang. On Mining Cross-graph Quasi-cliques. In *KDD*, 2005.

[33] Y. Peng, B. Choi, and J. Xu. Selectivity Estimation of Twig Queries on Cyclic Graphs. In *ICDE*, 2011.

[34] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, 2012.

[35] S. Ranu and A. K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *ICDE*, 2009.

[36] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs. In *CIKM*, 2012.

[37] J. Sankaranarayanan and H. Samet. Distance Oracles for Spatial Networks. In *ICDE*, 2009.

[38] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online Query Execution Engine for Large Distributed Graphs (Demo). In *ICDE*, 2012.

[39] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, 2013.

[40] L. Sheng, Z. M. Özsoyoglu, and G. Özsoyoglu. A Graph Query Language and Its Query Processing. In *ICDE*, 1999.

[41] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library - User Guide and Reference Manual*. C++ in-depth series. Pearson / Prentice Hall, 2002.

[42] A. Silva, W. M. Jr., and M. J. Zaki. Mining Attribute-structure Correlated Patterns in Large Attributed Graphs. *PVLDB*, 5(5):466–477, 2012.

[43] SPARQL. `http://www.w3.org/TR/rdf-sparql-query/`.

[44] G. Szabo and G. Fath. Evolutionary Games on Graphs. *Physics Reports*, 446(4-6):97–216, July 2007.

[45] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[46] L. G. Valiant. A bridging Model for Parallel Computation. *Communincations of ACM*, 33(8):103–111, 1990.

[47] F. Wei. TEDI: Efficient Shortest Path Query Answering on Graphs. In *SIGMOD*, 2010.

[48] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD*, 2004.

[49] P. Zhao, C. C. Aggarwal, and M. Wang. gSketch: On Query Estimation in Graph Streams. *PVLDB*, 5(3):193–204, 2011.

[50] P. Zhao and J. Han. On Graph Query Optimization in Large Networks. *PVLDB*, 3(1):340–351, 2010.