G-SQL: Fast Query Processing via Graph Exploration

Hongbin Ma*
eBay
Shanghai, China
honma@ebay.com

Bin Shao
Microsoft Research Asia
Beijing, China
binshao@microsoft.com

Yanghua Xiao
Fudan University
Shanghai, China
shawyh@fudan.edu.cn

Liang Jeff Chen
Microsoft Research Asia
Beijing, China
jeche@microsoft.com

Haixun Wang*
Facebook
Menlo Park, USA
haixun@gmail.com

ABSTRACT

A lot of real-life data are of graph nature. However, it is not until recently that business begins to exploit data's connectedness for business insights. On the other hand, RDBMSs are a mature technology for data management, but they are not for graph processing. Take graph traversal, a common graph operation for example, it heavily relies on a graph primitive that accesses a given node's neighborhood. We need to join tables following foreign keys to access the nodes in the neighborhood if an RDBMS is used to manage graph data. Graph exploration is a fundamental building block of many graph algorithms. But this simple operation is costly due to a large volume of I/O caused by the massive amount of table joins. In this paper, we present G-SQL, our effort toward the integration of a RDBMS and a native in-memory graph processing engine. G-SQL leverages the fast graph exploration capability provided by the graph engine to answer multi-way join queries. Meanwhile, it uses RDBMSs to provide mature data management functionalities, such as reliable data storage and additional data access methods. Specifically, G-SQL is a SQL dialect augmented with graph exploration functionalities and it dispatches query tasks to the in-memory graph engine and its underlying RDMBS. The G-SQL runtime coordinates the two query processors via a unified cost model to ensure the entire query is processed efficiently. Experimental results show that our approach greatly expands capabilities of RDBMs and delivers exceptional performance for SQL-graph hybrid queries.

1. INTRODUCTION

Relational databases have been widely used in a variety of industrial applications. Queries with multi-way joins are prevalent in real workloads. Example 1 illustrates one of such queries. As discussed in [1], database applications involving decision support and complex objects usually have to specify their desired results in terms of multi-way join queries. An earlier study on the workload of a real-life accounting system reveals that there are almost 4% quires contain join operations [2]. The analysis of TPC-DS benchmark [3] shows that 23.6% (22 out of 93) queries join multiple tables[1]. A more recent study [4] shows that real analytics workloads usually involve queries of graph-lets (a frequent subgraph) and these queries usually involve 5-10 self-joins on network data.

The execution process of join operations can be considered as explorations over links in an entity-relationship graph. For the mainstream relational databases, a single access to a graph node representing an entity stored in a table does not disclose information about its neighboring nodes (entities in other tables) in the graph. These neighbors have to be retrieved through additional table lookups, which makes graph exploration intrinsically slow due to poor cache locality.

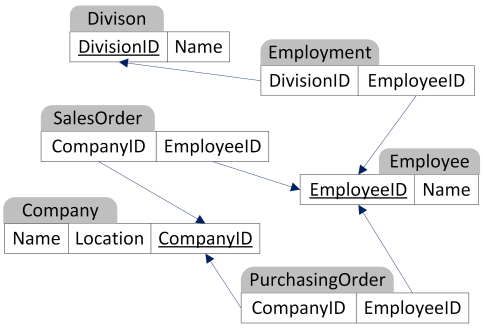


Figure 1: A database schema

Example 1. *Let us consider a simplified schema of a real-life enterprise database (Figure 1). The database records purchase and sales orders placed by employees of all divisions with different companies. Suppose we want to find divisions and employees who make purchase from companies in Seattle and make sales to companies in New York. The corresponding SQL query is shown in Figure 2.*

In general, it is not efficient for a relational database to perform a multi-way join of six tables to answer the query.

*This work was done in Microsoft Research Asia.

[1]Query 78 in Figure 10 of [3] even joins more than 20 tables.

Proceedings of the VLDB Endowment, Vol. 9, No. 12

Even if we create indexes for each foreign key column, the join process is still costly due to frequent disk I/O requests, especially when values are not selective enough to reduce the search space. The real-life database introduced in Example 1 has 0.86 million Company records and 2.7 million SalesOrder records, the query takes 1.3 seconds to complete.

```
SELECT Division.Name, Employee.Name
FROM   Division D, Employee E, Employment Y,
       PurchasingOrder P, Company M
       SalesOrder S, Company C
WHERE  E.EmployeeID = S.EmployeeID and
       S.CompanyID = C.CompanyID and
       E.EmployeeID = P.EmployeeID and
       P.CompanyID = M.CompanyID and
       Y.DivisionID = D.DivisionID and
       Y.EmployeeID = E.EmployeeID
       C.Location = 'New York' and
       M.Location = 'Seattle'
```

Figure 2: A multi-way join in a relational database

Most multi-way join queries essentially are pattern matchings in the entity-relationship graphs as illustrated in Example 1. Many graph-based query optimization techniques have been proposed in graph database to boost the performance of graph operations expressed by multi-way joins. Thus, we may wonder if we can *use graph database to boost the performance of multi-join queries* over relational database.

Basic Idea. In this paper, we present G-SQL that takes the advantage of both graph database and relational database to boost multi-way join queries on relational databases. The execution of multi-way join queries in general can be decomposed into two parts: *relation access* and *graph exploration*. The major concern of such an architecture is that we need to boost the performance of multi-way join queries without sacrificing the persistence, integrity, and many other desired property of relational databases. Hence, we resort to a hybrid architecture. Specifically, we use an in-memory graph computation engine (Trinity [5]) to provide fast graph exploration. We use RDBMSs or SQL engines, on the other hand, to process relational queries and provide additional functionalities such as data persistence, integrity and consistence. With Trinity, we transform multi-way join operations into subgraph matching or graph traversal operations on an in-memory graph. In other words, Trinity manages the graph topology while RDBMSs manage the data other than the topology.

Let us continue our running example shown in Figure 2, relation access can be efficiently implemented by employing B+ tree index built upon C.Location and M.Location. The join operation however can only be efficiently implemented by a graph traversal operation from a *Division* node to an *Employee* node, then to a *Client* node. The corresponding query expressed in G-SQL is shown in Figure 3. The graph traversal is dictated by the *MATCH* clause, where *Division*, *Employee* and *Client* are graph nodes and *-[Employees]-* and *-[Clients]-* are edges linking graph nodes. Relational database systems in general are not good at handling graph explorations expressed in multiple joins. A straightforward solution thus is decomposing a query into the graph exploration and traditional relation access parts. Then, using graph computation engine and relation database engine to process each part individually.

```
SELECT Division.Name, Employee.Name
FROM   Division, Employee,
       Company Client, Company Merchant
MATCH  Division−[Employees]→Employee
       −[Clients]→Client,
       Employee−[Merchants]→Merchant
WHERE  Merchant.Location = 'Seattle' AND
       Client.Location = 'New York'
```

Figure 3: A G-SQL example that integrates SQL with graph exploration

Comparison with Materializd View. One alternative solution to speedup the multi-way join queries is to use materialized views. A materialized view is a derived table defined on other base tables. Materialized view can be helpful in query optimization. However, materialized view has some limitations. First, a materialized view only works for a limited set of queries. In other words, an ad-hoc query cannot be accelerated if no corresponding materialized views are defined. In our approach, any queries with complex explorations can be accelerated since the relational database has been replicated to the graph store and the joins can be directly computed by the graph processing engine. Second, as we will show in the experimental study, by leveraging an in-memory graph engine with fast graph exploration support, in general we can have more performance gain than the materialized views.

Contributions. A big challenge we face is that G-SQL needs to coordinate two query processors, namely SQL and Trinity. By integrating the two execution engines, G-SQL needs a global cost model to coordinate the two engines. In this paper, we have made this following contributions: a) We have designed a unified system G-SQL that can dramatically speed up multi-join queries with graph explorations while allowing users to continue benefiting from mature relational technologies. b) We have designed a scheme that enables us to leverage prior solutions of incremental view maintenance to synchronize between the SQL database and the graph store upon updates. c) We have devised a unified model that allows G-SQL's runtime to choose a hybrid execution plan that interleaves SQL constructs and graph explorations to achieve global optimality.

The remaining part is organized as follows. Section 2 describes how to model graphs and graph queries in G-SQL. Section 3 overviews the query execution process of G-SQL. Section 4 elaborates the process of graph query processing. A unified cost model is elaborated in Section 5. Section 6 presents the experimental results. The related work is discussed in Section 7. Section 8 concludes.

2. DATA MODELING

In this section, we present our G-SQL's data model and the graph construction and data updating process.

2.1 Relation Based Graph Modeling

We use the extended SQL to specify a graph node on top of a relational database. The full specification of the language is shown in Appendix C. Figure 4 shows an example query that defines an Employee graph node type based on

the schema in Figure 1. The FROM and WHERE clauses are standard SQL statements. The SELECT clause is slightly different. First, a SELECT element can be a SQL sub-query that returns a collection of atomic values, e.g., the last three SELECT elements in Figure 4. Second, the AS renaming must specify what type a projected column is mapped to in the graph, i.e., ATTRIBUTE, EDGE annotations in Figure 4. Finally, we demand that one SELECT element be dedicated for nodes' identities. This column must be unique in the query's result, so that the nodes of this type can be uniquely identified in the graph.

```
SELECT E.EmployeeID as NODEID[EmployeeID],
       E.Name as ATTRIBUTE[Name],
       (
       SELECT Em.DivisionID
       FROM   Employment Em
       WHERE  Em.EmployeeID = E.EmployeeID
       ) AS EDGE[Divisions],
       (
       SELECT P.CompanyID
       FROM   PurchasingOrder P
       WHERE  P.EmployeeID = E.EmployeeID
       ) AS EDGE[Merchants],
       (
       SELECT S.CompanyID
       FROM   SalesOrder P
       WHERE  S.EmployeeID = E.EmployeeID
       ) AS EDGE[Clients],
FORM   Employee E
```

Figure 4: A SQL query defining the Employee node

We introduce to SQL a new MATCH clause[2] to specify graph pattern matching that is prominent in graph processing. A graph pattern consists of one or more paths. A path is a sequence of node aliases specified in the FROM clause. Node aliases are connected by named edges. A named edge is a special property of a graph node that is annotated by EdgeType attribute when the node is declared in the Trinity script. An example query is shown in Figure 3, in which the MATCH clause specifies two paths that form a tree pattern.

2.2 Graph Construction and Updating

Graph Construction. G-SQL allows users to declare a graph schema with the augmented SQL. With the schema defined in SQL, G-SQL will generate a graph schema specification script in TSL (Trinity Specification Language)[3]. Figure 6 shows a tsl script that specifies the Employee node generated from the SQL definition given in Figure 4. An employee has two attributes EmployeeID and Name, as well as 3 groups of outgoing edges Clients, Merchants, and Divisions. We use the "[...]" construct to annotate the construct that follows it. Each element of Clients and Merchants is a 64-bit identifier that references a graph node of type Company. Each element of Divisions is a 64-bit integer that references a graph node of type Division.

With a TSL script, the TSL compiler of Trinity graph engine[5] automatically generates dedicated object-oriented data access methods to access the fields stored in the in-memory blob. G-SQL also builds data access wrappers for visiting the properties residing in the RDBMS so that every field of a node can be accessed via unified interfaces as shown in Figure 5. Using the data access methods generated by TSL compiler, G-SQL *extracts*, *constructs*, and *stores* the graph in the graph engine to leverage its efficient memory storage for graph explorations.

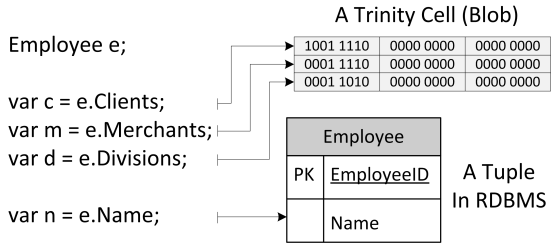


Figure 5: Unified data access interfaces

```
cell struct Employee {
   long EmployeeID;
   string Name;
   [EdgeType:SimpleEdge, ReferencedCell:Company]
   List⟨CellId⟩ Clients;
   [EdgeType:SimpleEdge, ReferencedCell:Company]
   List⟨CellId⟩ Merchants;
   [EdgeType:SimpleEdge, ReferencedCell:Division]
   List⟨CellId⟩ Divisions;
}
```

Figure 6: TSL script that defines Employee node

Graph Updating. After constructing the graph, we need to keep the graph up-to-date once the relational database is changed. G-SQL regards the declared graph topology as a special form of SQL nested views, which allows us to leverage previous solutions of incremental view maintenance to keep updated the Trinity-hosted topology when the SQL database changes. There is a wealth of literature in SQL view maintenance such as [6]. Let $V = Q(\mathcal{R})$ be a view, where Q is a SQL query and $\mathcal{R} = \{R_1, \ldots, R_n\}$ is a set of tables. A common approach to incrementally maintaining V is through differential tables. That is, every addition and deletion to a base table R_i is recorded in two differential tables, Δ^+R_i and Δ^-R_i, respectively. An update is equivalent to a combination of a deletion and an addition. Let $\Delta^+\mathcal{R} = \{\Delta^+R_1, \ldots, \Delta^+R_n\}$ and $\Delta^-\mathcal{R} = \{\Delta^-R_1, \ldots, \Delta^-R_n\}$. A maintenance algorithm takes input as $\mathcal{R}, \Delta^+\mathcal{R}, \Delta^-\mathcal{R}$ and computes differential tables Δ^+V and Δ^-V that can be applied to V incrementally.

A graph node specification in G-SQL can be decomposed into a root view corresponding to the outer query in Figure 4 and a number of child views corresponding to each sub-query in the SELECT clause. By using differential tables, we will be able to incrementally update these views and eventually propagate updates to the graph.

3. QUERY EXECUTION

In this section, we explain how query processing is divided between RDBMS and Trinity at the system level. We first

[2]The formal specification of the MATCH clause is given in Appendix D.

[3]http://www.graphengine.io/docs/manual/TSL/tsl-basics.html

present the basic procedure to execute queries in G-SQL then discuss the optimization of the execution procedure.

3.1 Basic Procedure

A G-SQL query is semantically equivalent to a SQL query. By using transformation rules, the G-SQL query can be rewritten to its SQL equivalence and completely evaluated by a SQL query engine. Conventional SQL databases, however, have limitations in graph explorations. Instead, G-SQL leverages Trinity's native graph store and execution engine to process graph query constructs, and combine them with conventional SQL constructs. In this subsection, we introduce the query processing architecture of G-SQL.

A G-SQL query, including both conventional SQL statements and extended query constructs, is parsed into a *join graph*. A join graph is a core component used by conventional SQL query optimizers to produce an execution plan. It describes a number of tables connected by join predicates, as well as a few peripherals, such as GROUP-BY, SUM/COUNT or ORDER BY. A join graph in G-SQL involves both graph nodes and relational tables. It can be divided into three components: the *graph pattern* described in the MATCH clause, conventional *table-join* graphs with their peripherals, and a number of *cut-set joins* that connect the graph pattern and the table-join graphs.

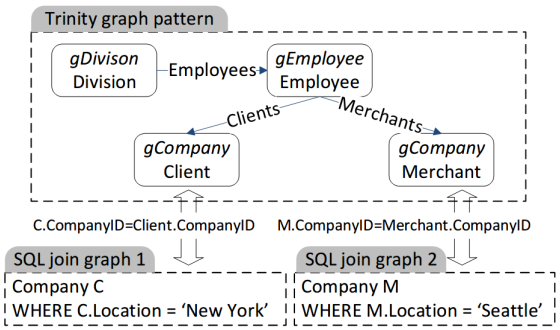


Figure 7: The join graph of the query in Figure 3

Figure 7 illustrates the join graph of the query in Figure 3. The graph pattern described in the MATCH clause is in the *Trinity graph pattern* region, and the remaining table-join graphs are in the others. We use the prefix 'g' to indicate the graph nodes loaded in the graph engine. In this example, the two table aliases Company C and Company M are not connected, so they form their own table-join graphs. The two table-join graphs are connected to the graph pattern through joins on attributes.

G-SQL's query processor takes as input a join graph, derives an execution plan that assigns the join graph's components to the SQL and Trinity query engines, and coordinates the two engines' results at runtime. At the high level, the query processor uses Trinity's execution engine to process graph patterns. For each table-join graph, G-SQL formulates a new equivalent SQL query and sends it to the relational database for evaluation. G-SQL's query processor evaluates cut-set joins by pushing values from one engine to the other and calling the join API in the target engine. In the SQL query engine, this API is implemented by storing the pushed values into a temporary table and invoking a join with it in a formulated SQL query. In Trinity, this API is implemented by using the pushed values to update the graph exploration's runtime states. As we will show later, unlike the SQL execution engine that materializes intermediate results of each physical operator at the earliest convenience, G-SQL maintains all explorations' states and do not discard them until all bindings of the nodes in the graph pattern have been established.

3.2 Optimization

A core optimization of the G-SQL's query processor is to determine the order of cut-set joins that exchange intermediate results between the two engines. G-SQL's query processor assigns table-join graphs and graph patterns to the SQL engine and Trinity respectively. It relies on their own execution engines to optimize and execute individual components. However, the join order of cut-set joins will ultimately change individual components' execution plans and the query's overall cost.

To understand the effects of cut-set joins, consider the example in Figure 7. When G-SQL's query processor decides to evaluate the second table-join graph first, an equivalent SQL query is formulated and processed by the SQL execution engine, as shown by the first query in Figure 8. G-SQL processes the graph pattern first and passes the Merchant node's bindings into the SQL engine to perform the join M.CompanyID = Merchant.CompanyID, the formulated SQL query becomes the second query in Figure 8. The second SQL query is likely to have a different execution plan from the first. Assuming there is no secondary index on the Location column, the second query tends to have a lower cost because the input tables are smaller. By comparison, the SQL query in the first plan may be more expensive. But the SQL query's results will help Trinity filter the bindings of the Merchant node. Trinity's query optimizer, in this case, will derive a new graph exploration plan that takes advantages of the filtering to reduce the graph explorations' complexity.

```
SELECT M.CompanyID
FROM   Company M
WHERE  M.Location = 'Seattle'
```

```
SELECT M.CompanyID
FROM   Company M
WHERE  M.Location = 'Seattle' AND
       M.CompanyID IN (Merchant nodes' binding set)
```

Figure 8: A SQL query w/ or w/o filtering by node bindings

We formalize G-SQL's query optimization problem as follows. We use $Q_i(R_1, \ldots, R_{m_i}), i = 1, \ldots, n$ to denote n table-join graphs and $G(v_1, \ldots, v_k)$ to denote the graph pattern. The execution plan of a G-SQL's query is a sequence of steps, each represented as $Q_i \rightarrow G$ or $G \rightarrow Q_j$:

$Q_i \rightarrow G$: Q_i is evaluated in the SQL engine, possibly with the input tables filled by the graph nodes' bindings generated from prior steps. Q_i's results are passed to Trinity to update the connected nodes' bindings.

$G \rightarrow Q_j$: Trinity starts from the nodes whose bindings have been established from the prior steps, and processes the remaining graph pattern until it establishes all the bindings of the nodes that are connected to Q_j's input tables. Taking these bindings as inputs, Q_j is evaluated in the SQL engine.

G-SQL's query optimizer aims to find a sequence of steps such that the overall execution time is minimized. In the

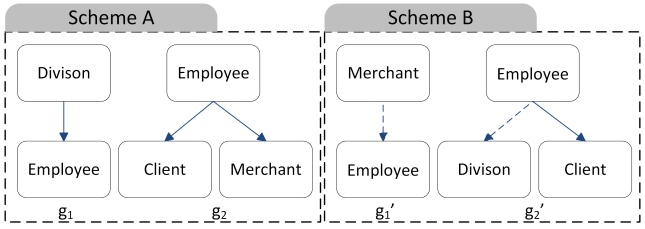


Figure 9: Different decomposition schemes

following two sections, we will first review the process of native graph query processing, and then present cost models of the two types of queries and a selection algorithm that generates an execution plan.

4. GRAPH QUERY PROCESSING

G-SQL leverages Trinity graph engine's distributed fast random data access capability to process graph pattern matching queries. In the G-SQL framework, Trinity functions as a distributed in-memory graph engine that provides:

- Distributed in-memory data management;
- Cluster management in the distributed setting;
- Distributed message passing framework.

On top of Trinity's data management and message passing interfaces, we implemented the gStep distributed pattern matching algorithm as the execution engine of G-SQL. In this section, we introduce this graph query processing framework.

4.1 gStep: Distributed Graph Matching Unit

A G-SQL query's MATCH clause specifies a graph pattern to match. G-SQL decomposes the graph pattern into a number of connected *gSteps* and evaluates them individually.

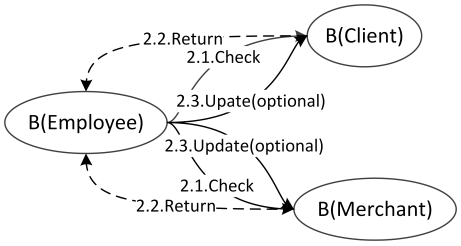
DEFINITION 4.1 (GSTEP). *A gStep, denoted by a triple* $g = \langle r_g, L_g, C_g \rangle$*, is a tree of height one. It consists of a root node* r_g *and* n *leaves* $L_g^i, 1 \leq i \leq n$ *and* $n \geq 1$*. A leaf* L_g^i *may have predicate(s)* C_g^i *that are specified in the query's* WHERE *clause.*

A gStep is a graph exploration primitive. It resembles the *STwig* concept in [7]. As elaborated in [7], any patterns including patterns with cycles can be decomposed into gSteps or STwigs. There are usually multiple ways to decompose a pattern into gSteps, especially when we also take reverse edges into consideration. For example, the graph pattern in Figure 3 can be decomposed in two ways, as shown in Figure 9. The two plans may have dramatic performance differences, depending on the cost of each individual gStep and the cardinalities of their results. We defer the discussion of finding an optimal decomposition to Section 5. In this section, we assume a decomposition is given, and present how G-SQL processes each individual gStep, and how to join them to generate the final results.

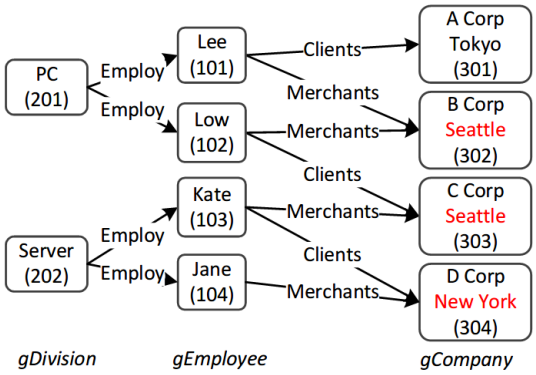
4.2 gStep Matching

A gStep $g = \langle r_g, L_g, C_g \rangle$ is evaluated by establishing node bindings for g's query nodes—the root r_g and each leaf L_g^i. A binding for a query node is a collection of matched graph node instances. It is established either by SQL constructs or by prior evaluations of other connected gSteps. A gStep's node bindings are execution states and can be passed to other connected gSteps to reduce their evaluation complexities. By using node bindings, we avoid materializing g's matches as intermediate results, and may reduce memory usage and message passing across machines. In the following, we denote the bindings of a query node by $B(r_g)$ or $B(L_g^i)$ and the matches for the gStep by $M(g)$.

The evaluation of a gStep $g = \langle r_g, L_g, C_g \rangle$ starts by initializing $B(r_g)$ and loading all matched node instances into it. The query processor then enumerates nodes in $B(r_g)$, and for each node, retrieves their neighbors as candidate nodes of L_g^i and sends them to $B(L_g^i)$. Upon receiving these candidates, $B(L_g^i)$ checks the candidates' eligibility, i.e., whether they satisfy the node type of L_g^i and the predicates C_g^i associated with L_g^i, and only keeps those satisfied. $B(L_g^i)$'s satisfied node instances are sent back to $B(r_g)$ to further prune $B(r_g)$: a node instance in $B(r_g)$ that does not receive responses from all $B(L_g^i), i \geq 1$ is removed from $B(r_g)$, because its neighbors do not satisfy L_g.



(a) Graph exploration sketch



(b) An example graph

Phase	B(Employee)	B(Merchant)	B(Client)
Prior	{101,102,103,104}	{}	{}
Init	{101,102,103}	{}	{}
2.1	{101,102,103}	{302,303}	{304}
2.2	{103}	{302,303}	{304}

(c) Dynamic bindings of query nodes

Figure 10: The evaluation of an example gStep

Figure 10a illustrates the process of evaluating g_2 (defined in Figure 9) in Plan A for the data graph shown in Figure 10b. Each graph node in the figure is labeled with a unique ID in the parenthesis. The arrows in Figure 10a indicate the message passing between different bindings. In the first stage, B(Employee) is filled with $\{101, 102, 103, 104\}$, a binding given by g_1's evaluation. For each node instances in B(Employee), the query processor explores its neighbors.

Node 104 is immediately pruned in the 2nd row in Figure 10c, because it does not have a Clients neighbor at all. The others emit $\{302, 303\}$ as a message to B(Merchant) and $\{301, 303, 304\}$ to B(Client), as shown in the 3rd row in Figure 10c. B(Merchant) receives candidates, verifies their eligibility, and returns $\{302, 303\}$ to B(Employee) as qualified merchants. B(Client) acts similarly. B(Employee) eventually collects responses from B (Merchant) and B(Client), and detects that only node 103 points to neighbors in both bindings. Hence, only node 103 is kept (the last row of Figure 10c).

4.3 Joining Multiple gSteps

Matches for the sequence of decomposed gSteps are retrieved as $M(g_1)$,$M(g_2)$... respectively. However, they are matches for a single gStep rather than the matches for the whole pattern graph. An extra step is required to get the final results: we need to join all the gStep matches. The graph processing engine is responsible for merging all the gStep matches.

Even if we make lots of binding prunings during exploration, invalid intermediate results still remain. For example $(201, \{101\})$ is added to $M(g_1)$ after the evaluation of gStep g_1 but it turns out to be an invalid intermediate result after g_2 is evaluated. However, it is our design philosophy not to remove it because we assume each gStep's evaluation does not attempt to backtrack previous gSteps' matches or previous query nodes' bindings. This design avoids tracking too much historical information during the graph exploration, which is costly to communicate. The tradeoff for such design is that a final join is required after all gSteps have been evaluated. Fortunately, our voluntary binding pruning within each gStep evaluation will greatly reduce the intermediate results, which makes the final join relatively light weight compared to traditional relational solutions.

We call a set of gSteps *linearly ordered* if these gSteps decomposed from a query graph are executed in an order that each gStep's root node exists in the preceding gStep's leaf nodes. Under this condition, joining in the reverse order of execution tends to be the best choice because it guarantees that every intermediate join output will be in the final results. In other words, no intermediate output will be generated in vain. However, gSteps in real situations may not simply exhibit a linear pattern. We adopt a first-executed last-joined strategy to join the final results since we assume the scale of final join is relatively small.

5. COST MODEL AND QUERY PLANNING

In this section, we present G-SQL's query optimizer. We first introduce cost models for SQL queries and gSteps that are measured by elapsed time and result cardinality. We then present a dynamic programming algorithm that finds an optimal execution plan.

5.1 Cost Model for SQL Constructs

A G-SQL query's execution plan is a sequence of $Q_i \rightarrow G$ and $G \rightarrow Q_i$ steps. A table-join graph Q_i is evaluated with or without node bindings as input tables. When there are no input tables, Q_i is a conventional SQL query and its cost and cardinality is given by the SQL optimizer. In the following, we focus on Q_i with input tables. In such cases, the input tables are joined with Q_i through IN clauses (as demonstrated in Figure 8).

We use a simple model to estimate the cost of Q_i when Q_i has a binding set as the input table. The idea is derived from the merge join and the nested loop join in the relational world: when we join two tables, if one is small, the nested loop join is preferable. The execution plan iterates through each tuple of the smaller table and looks up the other; if one table's size is comparable to the other's, the merge join is often better. Likewise in our case, if Q_i's input table is small, it's preferable to iterate through each tuple in the input and feeds it into Q_i; if the input size is large, the execution plan could evaluate Q_i completely and join with the input table thereafter.

Given a table-join graph Q_i, the cost model estimates the time for executing Q_i, denoted as $ET(Q_i)$, and the result cardinality, denoted as $ER(Q_i)$. The model relies on two SQL queries, $Q_i^{\{\}}$ and $Q_i^{\{1\}}$, whose costs are given by the SQL optimizer. $Q_i^{\{\}}$ is the full query describing the table-join graph, e.g., the first query in Figure 8. Its cost corresponds to the plan using the merge join. $Q_i^{\{1\}}$ adds to Q_i a 1-node binding set in an IN clause, e.g., the second query in Figure 8. This query describes the unit cost of evaluating a node binding when using the nested loop join. Putting $Q_i^{\{\}}$ and $Q_i^{\{1\}}$ together, we have the following equation, where x is the estimated size of the node binding set, to estimate the time cost:

$$ET(Q_i) = \begin{cases} x \cdot ET(Q_i^{\{1\}}) & x < \dfrac{ET(Q_i^{\{\}})}{ET(Q_i^{\{1\}})} \\ ET(Q_i^{\{\}}) & x \geq \dfrac{ET(Q_i^{\{\}})}{ET(Q_i^{\{1\}})} \end{cases}$$

We use the following equation to estimate Q_i's result cardinality: $ER(Q_i) = x \cdot \frac{ER(Q_i^{\{\}})}{nc(T)} + 1$, where T is the type of the graph node connected to the table join graph Q_i and $nc(T)$ is the number of node instances in the graph. The intuition of the equation is as follows: when there are no node bindings, $ER(Q_i)$ is $ER(Q_i^{\{\}})$ and all graph nodes of type t contribute to $Q_i^{\{\}}$'s results. We assume that these nodes contribute to $Q_i^{\{\}}$'s results equally. So when only a portion of them are selected (by graph patterns), $ER(Q_i)$ is proportional to how many nodes are selected, which is characterized by $\dfrac{x}{nc(T)}$.

5.2 Cost Model for gSteps

The graph query processing engine preprocesses the data graph and uses the following statistics to guide the optimizer: a) Number of instances of a node type T_n, denoted by $nc(T_n)$; b) Number of instances of an edge type T_e, denoted by $ec(T_e)$; c) Shrink ratio of an edge type T_e, denoted by $sr(T_e)$. Edge instances of the same edge type may point to identical nodes. And we use shrink ratio to capture the degree of overlapping. It is equal to the number of unique nodes pointed by any edge of type T_e, divided by $ec(T_e)$.

Recall that the evaluation of a gStep starts from loading all members in $B(r_g)$, which is the binding for the root node of the gStep g, and goes through the following four stages:

- Enumerate binding members to find their neighbor candidates.

- Locally serialize these candidates to multiple packages, and send them to the target machines through network/loopback.
- The receivers unpack the packages and check each candidate's eligibility.
- The receivers send back the checked results.

The estimated time cost for evaluating a gStep is: $ET(g) = |B(r_g)| \cdot enumcost + \sum_i |Candidate(L_g^i)| \cdot (passcost * 2 + checkcost)$. In this equation, *enumcost*, *passcost*, *checkcost* are amortized time for the aforementioned step 1,2,3 respectively. The cost of step 4 is regarded to be equal to that of step 2, so *passcost* is doubled in the equation. $|Candidate(L_g^i)|$ is the number of candidate neighbors that are supposed to be sent to the ith machine. The sum of the candidate neighbors can be easily estimated given $|B(r_g)|$ and the average degree for each edge type. We can safely ignore the cost for enumerating because normally *enumcost* is negligible since it can be done in Trinity's memory storage efficiently. Thus, we have $ET(g) \approx \alpha \cdot \sum_i |Candidate(L_g^i)|$, where α is a constant factor and can be estimated by profiling.

The estimation of the root node and leaf node binding sizes are also required. To illustrate it we first consider a gStep that has only one leaf node denoted as L_g. Assume the edge connecting r_g and L_g is of edge type t_e, L_g of node type t_l and r_g of node type t_r. We also assume that the predicate on L_g is checked by cut-set joining with the results from SQL construct Q. The problem here is to estimate the size of $B(r_g)$ and $B(L_g)$ after the gStep gets evaluated, denoted as $ER(r_g)$ and $ER(L_g)$ respectively. First of all, we define two parameters: $hitratio = \frac{ER(Q^{\{\}})}{nc(t_l)}$ and $avgdegree = \frac{ec(t_e)}{nc(t_r)}$. The *hitratio* parameter indicates the likelihood of a random t_l typed node satisfying the predicate on L_g, and the *argdegree* parameter represents how many t_e edges each t_r node has on average. Given these two parameters, $ER(r_g) = (1-(1-hitratio)^{avgdegree}) \cdot |B(r_g)| + 1$ and $ER(L_g) = hitratio \cdot avgdegree \cdot |B(r_g)| \cdot sr(t_e) + 1$.

After gStep matching, the members in $B(r_g)$ gets pruned if none of its neighbors satisfy the predicate on L_g. In other words, the members keep staying in $B(r_g)$ as long as at least one of its *avgdegree* neighbors is qualified, thus a $1-(1-p)^n$ probability factor is applied in $ER(r_g)$.

$B(L_g)$ is essentially the result of propagating $B(r_g)$'s neighbors with a likelihood, so $ER(L_g)$ can be estimated by multiplying the count of neighbors($avgdegree \cdot |B(r_g)|$) with *hitratio*. It is further multiplied by a shrink ratio factor $sr(t_e)$ to eliminate duplications.

Both $ER(r_g)$ and $ER(L_g)$ are added by one for normalization. The estimation of gSteps with more than one leaf nodes works similarly. We assume the independence of hit ratios among different leaf nodes.

5.3 Execution Plan Optimization

G-SQL's query optimizer aims to find an optimal sequence of SQL constructs and gSteps. G-SQL translates it into a dynamic programming(DP) problem as follows: For any G-SQL queries, we have a pattern graph and multiple table join graphs connecting to the nodes in the pattern graph by cut-set joins. If we degrade each table join graph into a special *SQL node*, we get an *extended pattern graph*, or *search graph*. The problem is now to sequentially pick edges from the search graph so as to keep the cost at the minimum.

If the picked edge belongs to the original pattern graph, the edge corresponds to a gStep to be added into the execution list, and the incurred cost of such behavior equals the estimated cost of the gStep. Otherwise it is mapped to a SQL construct and the cost equals the estimated cost of the SQL construct under the current binding situation. Note a gStep can contain multiple edges, so we also allow picking more than one edges at a time.

We use ξ to denote the set of selected edges from the search graph. ξ can be used to denote a state in DP. Each state maintains the binding estimation for each query node. After edge selections, both the cost and binding estimations are updated. G-SQL starts from the states of size one, that is, G-SQL starts from choosing a one-leaf gStep or a SQL subquery to form a ξ of cardinality one. Once the computation for the state of size k is finished, G-SQL starts to compute the subgraphs of size $k+1$. The states of the same size get merged if their covering edges are identical. Specifically, if a gStep g with i leaf nodes is selected by a state s of size $k+1-i$. A new state s' of size $k+1$ is created, and we have: $cost(s') = min\{cost(s'), cost(s) + ET(g)\}$. Here, $cost(s)$ stands for the accumulated time consumptions for state s. The selected new component of the state can be disconnected with the original subgraph, because for many scenarios G-SQL needs to explore from multiple graph nodes to fully take advantage of the selective SQL constructs connected to them.

6. EVALUATION

In this section, we present our experimental results. We have done experiments on three real-life data sets.

- First, we compare the performance of G-SQL with Microsoft (R) SQL Server on the Microsoft Academic Graph (MAG for short) data set[4] (2015-11-06 snapshot is used). The decompressed raw data size is 99,172,505,167 bytes in tab-separated values format.
- Second, we compare the performance of G-SQL with SQL Server on a real-life enterprise CRM database.
- Third, we evaluate the performance of G-SQL on Microsoft Satori knowledge graph[5] in a distributed setting.

All these three data sets are graphs with power-law distribution: even the average degree is small, some nodes' degrees are large.

Table	Row Count
Papers	120,383,707
Authors	119,890,853
FieldsOfStudy	53,834
PaperKeywords	157,052,442
PaperReferences	952,364,264
PaperAuthorAffliation	312,274,259

Table 1: Statistics of the MAG data set

We have implemented G-SQL in C# and we have three deployments:

- For the experiments on the MAG data set, both G-SQL and SQL Server are deployed to the same machine that has a 2.60 GHz Intel(R) Xeon(R) E5-2690 v3 CPU, with 256 GB DDR3 RAM.

[4]http://research.microsoft.com/en-us/projects/mag/
[5]https://en.wikipedia.org/wiki/Knowledge_Graph

- For the experiments on the CRM database, both G-SQL and SQL Server are deployed to the same machine that has two 2.67 GHz Intel(R) Xeon(R) E5650 CPUs, with 98 GB DDR3 RAM.
- For the experiments on the Satori knowledge graph, G-SQL is deployed to a 16-machine cluster. The specs of each machine are the same as the machine used in the CRM experiments. The servers are connected by 40Gb/s InfiniBand Network adapters.

The operating system used in our experiments is 64-bit Windows Server 2008 R2 Enterprise with service pack 1. The relational database management system in comparison is Microsoft(R) SQL Server 2012. We built indexes on both primary and foreign keys for each table to improve SQL Server's performance. For the experiments with "Warm Cache", we reserved enough memory for SQL server and scanned every table before the experiments to warm up the SQL Server.

6.1 Experiments on MAG

We conducted comparison experiments on the MAG data set for both SQL and G-SQL. The statistical information of the MAG tables used in our experiments is shown in Table 1. We carefully chose 8 queries as listed in Appendix A. These 8 queries mimicked the common query patterns for Microsoft Academic portal[6]. The statistical information of these queries is shown in Table 2.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
table #	5	4	4	4	4	4	5	7
table join #	4	3	3	3	3	4	4	6
cut-set join #	2	1	1	1	1	0	1	2

Table 2: Statistics of MAG queries

Warm Cache. The first two columns for each query in Figure 11 are the execution time of SQL and G-SQL respectively. Note that log scaling is used for the vertical axis. For the MAG data set, G-SQL outperforms SQL by at least 5 times. For certain queries such as Q6 and Q8, G-SQL even outperforms SQL by two orders of magnitudes. The composition of the G-SQL cost is shown in the last two columns for each query. Column G-SQL(SQL) represents the component cost of accessing SQL Server during the execution of a G-SQL query, while G-SQL(REST) includes the Trinity runtime cost and the coordination cost.

Generally speaking, G-SQL delivers better performance than pure SQL, thanks to the fast graph exploration mechanism that offloads the costly multi-way joins onto the graph engine. G-SQL performs exceptionally well on certain type of queries, such as Q2, Q3, Q6 and Q8. The reason is elaborated in the introduction section.

Cold Start. Figure 12 shows the comparison results without cache warm-up. The overall performance decreases dramatically because extra disk IO operations are incurred. In cold start, G-SQL outperforms pure SQL queries by at least 7 times. For most of the queries except for Q3 and Q8, the execution time of G-SQL(SQL) is dominating (larger than 79%).

[6]http://academic.research.microsoft.com/

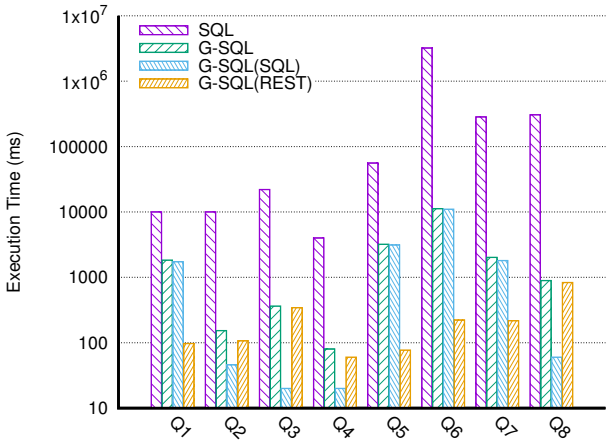


Figure 11: Execution time for Warm Cache (MAG)

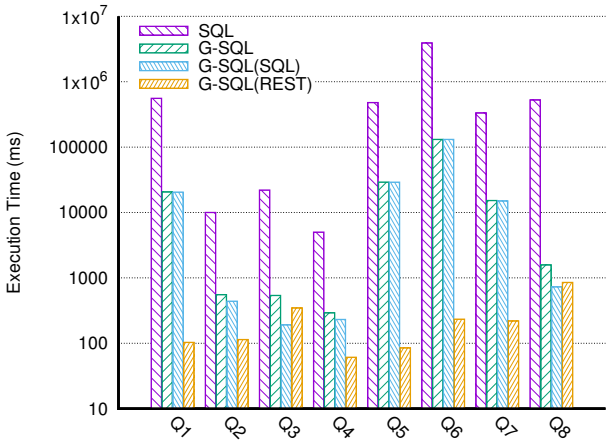


Figure 12: Execution time for Cold Start (MAG)

Materialized View. To compare G-SQL and materialized view, we have created indexed materialized views in SQL Server. We rewrote Q1, Q4, Q5, Q7, Q8 to leverage the materialized view listed in Appendix B. The comparison performance numbers are shown in Figure 13, the columns from left to right for each query are: SQL (cold start), SQL (warm cache), G-SQL (cold start), G-SQL (warm cache), SQL with materalized view (cold start), SQL with materialized view (warm cache). For Q1, the execution time with materialized view is dramatically less than that without materialized view, especially in cold start. For Q4 and Q5, the performance with materialized view is slightly better. For Q7, the execution times with/without materialized view are almost the same. Interestingly, the performance numbers with materialized view for Q8 are worse than those without materialzied view. Our observation is that materialized view can sometimes dramatically boost the query processing performance; but it is not always the case.

6.2 Experiments on CRM database

The CRM used in the experiments has very complex relations among its entities. The schema of the CRM database consists of 38 tables and 40.45 million rows. There are over 34 million nodes and over 150 million edges in the constructed

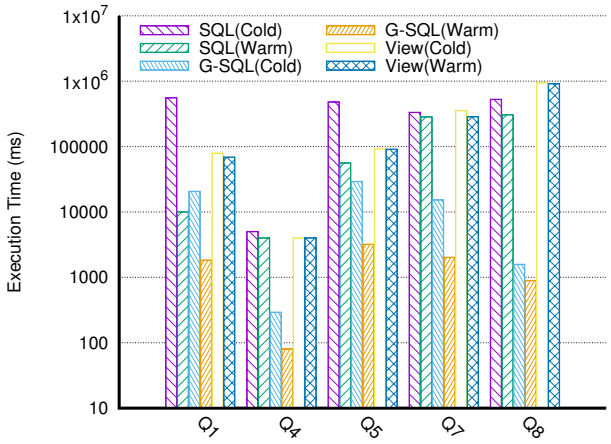


Figure 13: Performance comparison between SQL, G-SQL, and SQL with Materialized View (MAG)

```
SELECT a.Attr1, b.Attr2
FROM   TableA a, TableB b,
       TableC c, TableD d,
MATCH  a-[FK1]->b-[FK2]->c,
       d-[FK3]->c
WHERE  a.Attr3 = 'XXX' AND
       d.Attr4 = 'YYY'
```

```
SELECT a.Attr1, b.Attr2
FROM   TableA a, TableB b,
       TableC c, TableD d,
MATCH  a-[FK1]->d-[FK2]->b,
       c-[FK3]->d
WHERE  a.Attr3 = 'XXX' AND
       b.Attr4 = 'YYY' AND
       c.Attr5 = 'ZZZ'
```

Figure 14: Example query

graph. We selected seven queries whose patterns are exemplified in Figure 14. The table names and attributes are anonymized. The statistical information for each query is shown in Table 3.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
table #	4	3	4	3	4	4	4
table join #	3	2	3	2	3	3	3
cut-set join #	2	1	2	2	2	3	2

Table 3: Statistics of SQL queries

Warm Cache. The experimental results are shown in Figure 15. Because the data size of CRM is much smaller than that of MAG, the performance gain of G-SQL is not as dramatic as that in the MAG experiments: G-SQL outperforms SQL by 3.87 times on average. However, it is still clear that G-SQL(SQL) accounts for a large portion of the total cost. The ratio is about 56.6% on average.

For the CRM data set, G-SQL performs exceptionally well for certain type of queries, such as Q3, Q4 and Q6. But, for queries such as Q1 and Q2, G-SQL only performs slightly better than SQL. This is because these queries require little exploration along primary key and foreign key relations. In this case, the execution of SQL dominates the overall cost and the graph engine cannot help much.

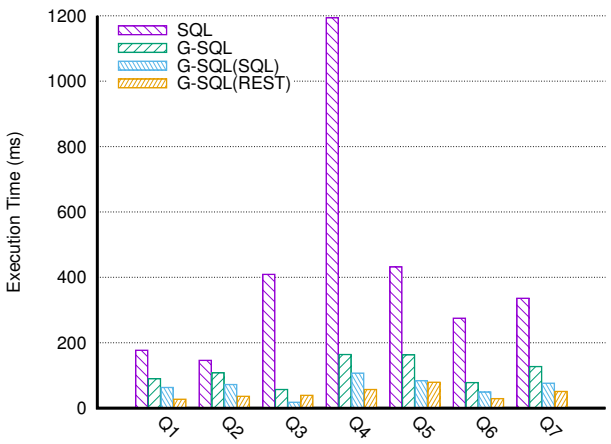


Figure 15: Execution time for Warm Cache (CRM)

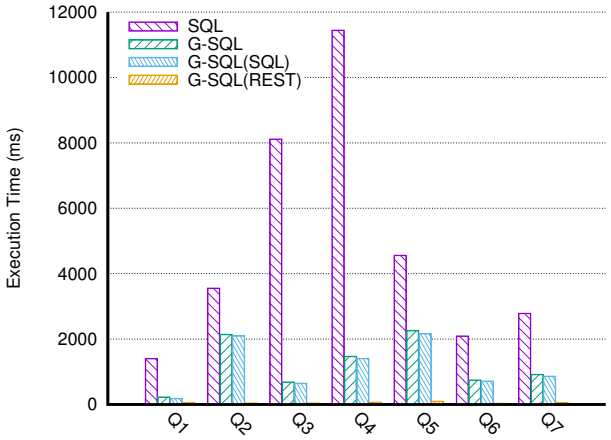


Figure 16: Execution time for Cold Start (CRM)

Cold Start. Figure 16 shows the comparison results without cache warm-up. For the CRM queries, G-SQL(SQL) accounts for about 93.3% of the G-SQL's cost. G-SQL outperforms pure SQL by at least 4 times, which is even more significant than that in the warm cache setting.

Effects of Query Plan Optimization. Figure 17 shows the overall time cost for each G-SQL query, together with the time spent on SQL. The first and the second column show the results for the naive approach denoted as G-SQL*; The third and the fourth column are for G-SQL with optimization based on our cost model. The effect is most dramatic for Q3, where G-SQL outperforms naive G-SQL* by 67 times. G-SQL can take advantage of the binding established by prior graph exploration to replace table scans with lookups. Since execution cost of SQL usually dominates a G-SQL query, the overall performance gain is dramatic. The same situation applies for Q1, Q6, and Q7. For queries such as Q2, Q4, and Q5, G-SQL is spending as much time as G-SQL* on SQL operations, this is where the size of estimated binding set grows so large that the G-SQL coordinator considers it

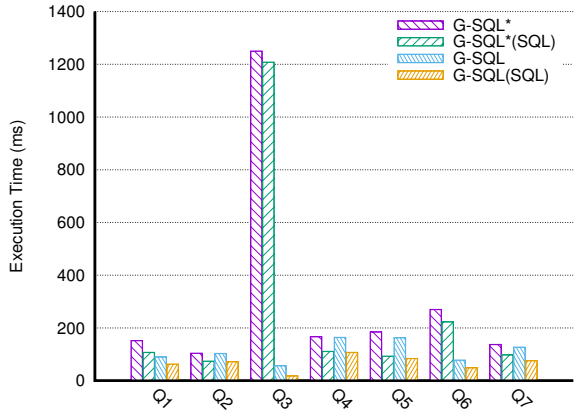


Figure 17: Comparison of different query plans

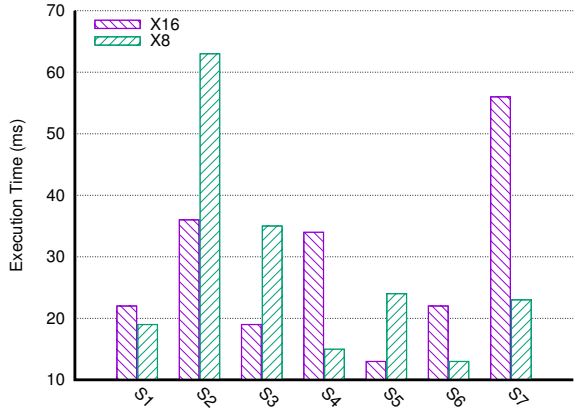


Figure 18: Execution time for a large RDF data set

worthless to leverage any bindings as input table for the SQL query.

To summarize, G-SQL brings significant performance gain to the traditional SQL databases without comprising its functionality. The performance gains are obtained by replacing SQL joins with graph explorations. It is worth mentioning that in our experiments we ensure all SQL Server tables are cached in the main memory for fair comparisons.

6.3 Experiments on distributed RDF database

We have a real RDF dataset that contains 11 billion tuples. The dataset ends up more than 3.5 billion nodes in the constructed graph. It takes about 460 GB main memory over a cluster of Trinity servers. We prepared seven queries for this dataset (S1 $\sim$ S7), e.g., to find persons who is engaged in a romantic relationship with celebrity Winona Ryder, or to get a list of actors who have costarred with Harrison Ford in a film. The experiments are conducted in two settings, one uses 8 machines (denoted as X8) and the other uses 16 machines (denoted as X16).

As shown in Figure 18, the execution times of the seven G-SQL queries are within 100 ms under X8 and X16 settings. With more machines, the computation power and system resources increase; but meanwhile the coordination costs and the number of network messages increase as well. Therefore, X16 does not necessarily outperform X8 for all queries.

7. RELATED WORK

Applications empowered by large graph processing are rising. Graphs are the natural representations for many categories of problems. First, graph can be used to model complex networks such as web graphs and social networks. The most well-known application for the web graph is the PageRank algorithm, which treats web documents as nodes and hyper links as edges. Graphs are also adopted by social network mining algorithms to accomplish tasks like community detection, which relates to the *node clustering* problem on large graphs [8]. Second, graph can be used to represent chemical and biology structures. The structure activity relationship (SAR) principle argues that the properties and biological activities of a chemical compound are related to its structure. Thus, graph mining may help reveal chemical and biology characteristics such as activity, toxicity, absorption, and metabolism [9]. Third, since the control flow in software systems can be modeled as *call graphs*, graph model can be used for software bug localization [10, 11], which aims to mine such call graphs in order to determine the bugs in the underlying programs. Similarly, some distributed computation frameworks like GraphLab [12] adopt graphs to represent the computation dependencies to facilitate parallel computing.

A number of graph stores/databases are developed to meet the growing market demands. Pregel [13], PowerGraph [14], Neo4j [15], and GraphChi [16] are the representative ones. Many research efforts are devoted to graph query languages. GraphLog [17] represents both data and queries as graphs. GraphQL [18] is a query language for graph databases that supports arbitrary attributes on vertices, edges, and graphs. In GraphQL, graphs are the basic unit of information and each query manipulates one or more collections of graphs. Some graph query languages, such as GOOD [19], GraphDB [20], and GOQL [21], adopt a graph-oriented object data model and provide functionalities for manipulating the graph data. As to the Resource Description Framework(RDF), SPARQL query language was made a standard by World Wide Web Consortium (W3C) and is recognized as one of the key technologies in the semantic web.

A lot of graph research interests have been spurred by Semantic Web and the accompanying RDF model. Many researcher [22, 23, 24, 25] investigated the possibility of querying web scale RDF data by storing it into native graph databases or object oriented databases. R. Angles et al. [24] argued that graph database models are closer to RDF models in motivations and use cases, and studied incorporating new query language primitives to support graph-like queries on RDF data. V. Bonstrom et al. [25] argued that RDF should be stored as a graph instead of triples to enable the semantic interpretation of RDF schema. They built a prototypical RDF system on an object oriented database. J. Hayes et al. [22] proposed a novel graph model called *RDF bipartite graphs* as an alternate graph representation for RDF data. Recently K. Zeng et al. [26] leveraged distributed graph exploration for querying RDF data stored as a graph and achieved a significant performance gain. Their implementation adopted special filters on nodes/edges as an optimization technique for pipelined execution plans. This work shares the same design philosophy to perform efficient graph pattern matching over a server cluster.

Interestingly, the combination of relational databases and graph computation is precedented, yet in an opposite way

to our work. Many efforts were devoted to tackling graph problems by leveraging existing relational database functionalities. Take the RDF problem for an instance: RDF model describes a graph by a set of triples, each of which describes an (attribute, value) pair or an interconnection between two nodes. Many existing systems including SW-Store [27], Hexastore [28], and RDF-3x [29] resort to SPARQL queries by storing RDF data as triples in relational databases and use SQL joins to simulate graph matching. However, such approaches suffer from the cache locality issues when exploring a large graph. A recent work [30] realized this problem and tried to build a special table to promote locality by putting an entity's predicates in the same row. In our point of view, it simulated a graph layer on top of relational databases by flattening the graph nodes in the table rows. Their method relieves the locality issues to some degree, but still requires building extra tables to fit the multi-valued predicates due to the limitations of relational databases.

8. CONCLUSION

We presented G-SQL, an integrated approach to accelerating the processing of expensive SQL-graph hybrid queries. G-SQL employs an in-memory graph engine on top of a relational database. G-SQL well leverages the maturity of relational database technology and the fast graph exploration power of the in-memory graph engine. We introduced a unified cost model to coordinate the underlying SQL execution engine and the native graph query processing engine. The in-memory graph engine utilizes fast graph traversal to avoid costly multi-way join queries on relational tables. The dramatic performance gain introduced by the graph-empowered G-SQL is confirmed by our comprehensive experiments.

9. ACKNOWLEDGEMENT

Yanghua Xiao is supported by National Key Basic Research Program of China under No. 2015CB358800, National NSFC under No. 61472085, by Shanghai Municipal Science and Technology Commission foundation key project under No. 15JC1400900, by Shanghai STCF under No.16511102102.

10. REFERENCES

[1] M. Chen, H. Hsiao, and P. S. Yu. On applying hash filters to improving the execution of multi-join queries. *VLDB J.*, 6(2):121–131, 1997.

[2] P. S. Yu, M. syan Chen, H. ulrich Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE TSE*, 18:347–355, 1992.

[3] M. Pöss, R. O. Nambiar, and D. Walrath. Why you should run TPC-DS: A workload analysis. In *VLDB*, pages 1138–1149, 2007.

[4] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[5] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[6] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2), 1995.

[7] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 2012.

[8] M. J. Rattigan, M. Maier, and D. Jensen. Graph clustering with network structure indices. In *ICML*. ACM, 2007.

[9] H. Kubinyi, R. Mannhold, H. Timmerman, H.-J. Böhm, and G. Schneider. *Virtual screening for bioactive molecules*. John Wiley & Sons, 2008.

[10] C. Liu, X. Yan, H. Yu, J. Han, and S. Y. Philip. Mining behavior graphs for" backtrace" of noncrashing bugs. In *SDM*, 2005.

[11] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 2005.

[12] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[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*. ACM, 2010.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[15] N. Developers. Neo4j. *Graph NoSQL Database [online]*, 2012.

[16] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

[17] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS*. ACM, 1990.

[18] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*. ACM, 2008.

[19] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In *PODS*, pages 417–424. ACM, 1990.

[20] R. H. Güting. Graphdb: Modeling and querying graphs in databases. In *VLDB*, volume 94, pages 12–15. Citeseer, 1994.

[21] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*. IEEE, 1999.

[22] J. Hayes and C. Gutierrez. Bipartite graphs as intermediate model for rdf. In *The Semantic Web–ISWC 2004*. Springer, 2004.

[23] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering sparql queries via subgraph matching. *VLDB*, 4(8):482–493, May 2011.

[24] R. Angles and C. Gutierrez. Querying rdf data from a graph database perspective. In *The Semantic Web: Research and Applications*. Springer, 2005.

[25] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *Web Congress, 2003. Proceedings. First Latin American*. IEEE, 2003.

[26] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *VLDB*. VLDB Endowment, 2013.

[27] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB*

Journal, 2009.

[28] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 2008.

[29] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proc. VLDB Endow.*, 2008.

[30] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *SIGMOD*. ACM, 2013.

APPENDIX

A. Queries on MAG

Q1: Find the names of the authors who have published more than 100 papers in the database field during 2011 to 2015.

```
SELECT A.Name
FROM   Authors A, Papers P, PaperKeywords K,
       FieldsOfStudy F, PaperAuthorAffliation PAA
WHERE  A.ID = PAA.AuthorId AND PAA.PaperId = P.ID
       AND P.ID = K.PaperId AND K.FieldStudyId = F.ID
       AND F.Name = 'Database'
       AND P.Year BETWEEN 2011 AND 2015
GROUP BY  A.ID, A.Name
HAVING  COUNT(*) > 100
```

Q2: Find all the co-authors of Michael Stonebraker.

```
SELECT A2.Name
FROM   Authors A1, PaperAuthorAffliation PAA1,
       PaperAuthorAffliation PAA2, Authors A2
WHERE  A1.ID = PAA1.AuthorId
       AND PAA1.PaperId = PAA2.PaperId
       AND PAA2.AuthorId = A2.ID
       AND A1.ID <> A2.ID
       AND A1.Name = 'Michael Stonebraker'
```

Q3: Find all the papers that cite Leslie Lamport's papers.

```
SELECT P.Title
FROM   Authors A, PaperAuthorAffliation PAA,
       PaperReferences R, Papers P
WHERE  A.ID = PAA.AuthorId
       AND PAA.PaperId = R.ReferenceId
       AND R.PaperId = P.ID
       AND A.Name = 'Leslie Lamport'
```

Q4: Find the fields of study of Judea Pearl.

```
SELECT DISTINCT F.Name
FROM   Authors A, PaperAuthorAffliation PAA,
       PaperKeywords K, FieldsOfStudy F
WHERE  A.ID = PAA.AuthorId
       AND PAA.PaperId = K.PaperId
       AND K.FieldStudyId = F.ID
       AND A.Name = 'Judea Pearl'
```

Q5: Find the top 10 cited papers in the database field.

```
SELECT TOP 10 P.Title
FROM   Papers P, PaperReferences R,
       PaperKeywords K, FieldsOfStudy F
WHERE  A.ID = PAA.AuthorId
       AND PAA.PaperId = K.PaperId
       AND K.FieldStudyId = F.ID
       AND F.Name = 'Database'
GROUP BY   P.Title
ORDER BY   COUNT(*) DESC
```

Q6: Find the top 10 authors with the highest self citation numbers.

```
SELECT TOP 10 A.Name
FROM   Authors A, PaperAuthorAffliation PAA1,
       PaperReferences R, PaperAuthorAffliation PAA2
WHERE  A.ID = PAA1.AuthorId
       AND PAA1.PaperId = R.PaperId
       AND R.ReferenceId = PAA2.PaperId
       AND PAA2.AuthorId = A.ID
GROUP BY   A.ID, A.Name
ORDER BY   COUNT(*) DESC
```

Q7: Find all the papers that have more than 10 authors in the field of mathematics.

```
SELECT P.Title
FROM   Papers P, PaperAuthorAffliation PAA
       PaperKeywords K, FieldsOfStudy F
WHERE  PAA.PaperId = P.ID AND P.ID = K.PaperId
       AND K.FieldStudyId = F.ID
       AND F.Name = 'Mathematics'
GROUP BY  P.ID, P.Title
HAVING  COUNT(*) > 10
```

Q8: Find the common fields of study of Leslie Lamport and Michael Stonebraker.

```
SELECT DISTINCT F.Name
FROM   Authors A1, PaperAuthorAffliation PAA1,
       PaperKeywords K1, FieldsOfStudy F, Authors A2,
       PaperAuthorAffliation PAA2, PaperKeywords K2
WHERE  A1.ID = PAA1.AuthorId
       AND PAA1.PaperId = K1.PaperId
       AND K1.FieldStudyId = K2.FieldStudyId
       AND K2.PaperId = PAA2.PaperId
       AND PAA2.AuthorId = A2.ID
       AND K1.FieldStudyId = F.ID
       AND A1.Name ='Leslie Lamport'
       AND A2.Name = 'Michael Stonebraker'
```

B. Materialized view defined for MAG

```
CREATE VIEW PaperFieldOfStudyView
       WITH SCHEMABINDING AS
SELECT SELECT P.ID, F.Name, COUNT_BIG(*) AS CNT
FROM   dbo.Papers P, dbo.PaperKeywords K,
       dbo.FieldsOfStudy F
WHERE  P.ID = K.PaperId
       AND K.FieldStudyId = F.ID
GROUP BY  P.ID, F.Name;
CREATE UNIQUE CLUSTERED INDEX
PaperFieldOfStudyViewIndex ON
PaperFieldOfStudyView(ID,Name)
```

C. The syntax specification of the augmented SQL

```
⟨node_spec⟩   ::= SELECT ⟨sel_element⟩+
                  FROM   R (, R)*
                  WHERE  cond (AND cond)*
⟨sel_element⟩::= R_i.col AS ⟨type⟩[name]
                  | (sql expression) AS ⟨type⟩[name]
⟨type⟩        ::= EDGE | ATTRIBUTE | NODEID
```

D. The syntax specification of MATCH clause

```
⟨pattern⟩::= ⟨path⟩ (, ⟨path⟩)*
⟨path⟩    ::= ⟨node⟩ (−⟨edge⟩? → ⟨node⟩)*
⟨node⟩    ::= string
⟨edge⟩    ::= string
```