Using semijoin programs to solve traversal queries in graph databases

Norbert Martinez-Bazan
Sparsity Technologies
Barcelona
norbert@sparsity-technologies.com

David Dominguez-Sal
Sparsity Technologies
Barcelona
david@sparsity-technologies.com

ABSTRACT

Graph data processing is gaining popularity and new solutions are appearing to analyze graphs efficiently. In this paper, we present the prototype for the new query engine of the Sparksee graph database, which is based on an algebra of operations on sets of key-value pairs. The new engine combines some regular relational database operations with some extensions oriented to collection processing and complex graph queries. We study the query plans of graph queries expressed in the new algebra, and find that most graph operations can be efficiently expressed as semijoin programs.

1. INTRODUCTION

The use of graphs in analytic environments is getting more generalized, with real applications in many different environments like social network analysis, fraud detection, network traffic optimization, etc. Graph databases are one important solution to consider in the management of large datasets. A graph database (GDB) is any storage system that uses graph structures with nodes, edges, and properties to represent and store data. Some graph database industrial projects are, for example, Neo4J [4], a Java-based open-source graph database engine; Sparksee [6] (formerly DEX), a multi-platform graph database management system for efficient graph management in memory constrained environments; or Titan [8] a distributed and cloud-enabled graph database. There are other solutions to store and query graphs, such as the Resource Description Framework [5] (RDF) triple storages, or distributed engines specialized in the efficient processing of complex graph algorithms over large graphs, such as Pregel [14] with the vertex-centric computation model, or GraphLab [13], a parallel computation abstraction tailored to machine learning.

One of the most important challenges for graph databases is how to express graph queries and how to solve them efficiently. While other solutions have standard languages, such as SPARQL [7] for RDF, or domain-specific languages to write parallel graph analysis algorithms as Green-Marl [12], there is still a lack of formalization and standardization in the area of graph databases. In recent years several proposes have raised from the research community, for example GSPARQL [18], with graph extensions for SPARQL, or GQL [11] for querying graph patterns, but these solutions are still far for being adopted by the community. Instead, there is an important gap between the current industrial approach with libraries of very efficient APIs for procedural languages as Java, and high level languages like Gremlin [2] and Cypher [1] that combine pipes or data flows of results from the execution of graph APIs. It is very difficult to optimize complex graph oriented programs based on direct calls to low level APIs. Thus, one of the current challenges for graph databases is to define an algebra with a set of query operations that, combined into the form of query plans, can be optimized and executed efficiently in a query engine to solve different flavors or graph queries such as edge traversals (hops), graph pattern matching, graph algorithms, etc.

In this paper, we propose an algebra with a set of operations to solve graph queries in Sparksee, and we introduce extensions to SQL-like features to compute recursive programs and collection-oriented procedures typical for graph algorithms. The operations of this algebra are capable to reproduce the behavior of the current Sparksee APIs and, at the same time, are flexible enough to be combinable in the form of query plans that are suitable for optimization using the most usual database query optimization techniques as well as future optimizations more specific of graph patterns and graph traversals. Also, by adapting the operations to the graph data representation of the Sparksee engine, the query runtime will be able to take advantage of compressed bitmap processing and combination.

An important property of the operations is that most of them belong or have been adapted from the relational algebra, and the data structures in the form of key-value pairs are close to the relational model with multivalued attributes (Non First Normal Form relations). In particular, our main operation is the semijoin, which has been widely used in database technology for distributed systems, and semijoin program optimization has been studied and formalized in detail [10, 19]. Thus, our proposal is a compromise between the relational model and the noSQL and, in more detail, key-value pairs storages and graph querying systems. On the other hand, our procedural extensions for parallel subquery execution and recursion can be used to simulate graph analytical frameworks for the computation of complex graph algorithms over huge graphs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Proceedings of the Second International Workshop on Graph Data Management Experience and Systems (GRADES 2014), June 22, 2014, Snowbird, Utah, USA.
Copyright 2014 ACM 978-1-4503-2982-8 ...$15.00.

This paper is organized as follows. In Section 2, we introduce briefly the current graph data model and implementation of Sparksee. Section 3 presents the mapping of the Sparksee structures to key-value pairs, and details the algebra operations and their use to rewrite the Sparksee APIs. In Section 4 we analyze different queries from a public graph benchmark, focusing in the use and optimization of semijoin programs. Finally, Section 5 summarizes the conclusions and proposes the future work.

2. SPARKSEE GRAPH DATABASE

Sparksee [15, 16] (formerly DEX) is a graph database defined and implemented using a combination of several specialized structures that allow for an efficient management of very large graphs. The logical data model in Sparksee is a labeled and attributed directed multigraph, also known as *Property graph*. In a Sparksee graph each node and each edge has a unique label that denotes the object type. Edges point towards a direction, from the *tail* or source node to the *head* or destination node. Nodes and edges have also a variable list of attribute values. Figure 1 shows a simple graph that models a Social Network. In this example, the graph has three types of nodes: *Person*, *Post* and *Tag*; and five types of edges: relationships between people (*knows*), posting (*hasCreator*), tagging (*hasTag*), people preferences (*hasInterest* and *likes*). Each node and edge has some attributes associated, such as the *name*, the *date* or the *content*.

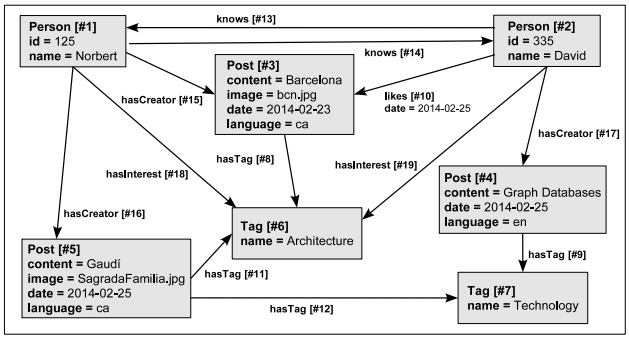

Figure 1: Example of a Sparksee graph.

Our proposal to manage large graphs with a limited amount of memory is based on object identifiers (*oids*) and two types of structures: compressed *bitmaps* and *maps*. Each object (a node or an edge) in a Sparksee graph has a unique identifier in the graph, called *oid*. An oid is represented by a positive integer number, and a collection of objects is a set of distinct oids. There exists an oid generator to provide consecutive unique oids to nodes and edges when they are created. A *bitmap* or bit-vector is a collection of presence bits that denotes which objects are selected or related to other objects. These bitmaps are compressed as described in [16]. A *map* is an inverted index with key values associated to bitmaps or data values, and it is used as an auxiliary structure to complement bitmaps, providing full access to all the data stored in the graph. We combine these two types of structures to build a more complex one: the *link*. A link is a binary association between unique identifiers and data values. It provides two basic functionalities: given an oid, it returns the value; and, if it is fully indexed, given a value it returns all the oids associated to it.

Following a partitioned storage model, a graph in Sparksee is represented as a combination of links. Figure 2 depicts the structures that represent the graph in the Social Network example presented in Figure 1. The links have been graphically divided into three groups: the first group, OBJECTS, contains the links that allows for accessing the nodes and edges in the graph; the second group, RELATIONSHIPS, is divided into two sets of links (TAILS and HEADS) that are related to the connectivity among nodes; and the third group, ATTRIBUTES, contains all the links for accessing attribute values in an object or to obtain the objects related to a certain value. In this figure each link is shown with both indexes (from oid to value and from value to oids). Bitmaps are depicted as uncompressed although Sparksee compresses them for efficiency.

3. QUERYING SPARKSEE

In this section, we present a new graph query algebra that we are implementing for future Sparksee releases. These operations are combined to build query plans, which solve the most common graph queries: edge traversals (hops), pattern matching, graph statistics, etc. The main requirements for the algebra are: (i) to take advantage of modern multicore CPUs and their cache hierarchy; (ii) to allow the maximum degree of parallelization: intra-operator, intra-query and inter-query; (iii) to be suitable for a future Sparksee distributed graph database with partitioned graphs; (iv) to be able to map the current Sparksee API operations and data structures for compatibility issues; and (v), to adapt and reuse as much as possible the knowledge generated by more than 30 years of research in the area of relational, object oriented and RDF databases. Thus, we introduce first the data structures that represent the Sparksee links, and then the operations of the algebra classified in two groups: the relational-like operations that operate with links, and the procedural extensions to solve complex queries using collection oriented processing and recursion.

3.1 Links as key-value pairs

All data structures in the graph query pipeline are implemented as sets of key-value pairs (KVP). Each KVP set has two columns: the first one, named $\mathcal{K}$, contains unique keys that are either an oid or scalar datatype (e.g. a literal string, an integer, a real number, a boolean, etc.); the second one, named $\mathcal{V}$, contains a tuple formed by one or more elements, which are in turn a scalar datatype, an oid or collections of oids. In the new version of the engine, we represent the current Sparksee links presented in Section 2 with one or two KVP. The first one, which we call oid-to-value (OV), maps the left part of the link and sets $\mathcal{K}$ as an oid and $\mathcal{V}$ as a label, oid or scalar datatype. The second one, which we call value-to-oids (VOS), sets $\mathcal{K}$ as a label, oid or scalar datatype to a collection of oids. The first one is always present, but the second one only exist if indices are created.

In the paper, we identify the persistent KVPs that represent the links with a prefix that indicates its type (OV_ or VOS_), followed by the name of the content of the KVP (e.g. *LABELS*, *TAILS* or *ATTR*) and the object type (e.g. *hasTag* or *Post*). For instance, Figure 3 depicts the four KVPs that represent the relationship `hasTag` and `language` shown in Figure 2, with the unique keys underlined. Then, the new KVP representation of Figure 2 uses two KVP for the labels, four for each edge type (tails and heads), and one or two for each attribute depending on the indexation.

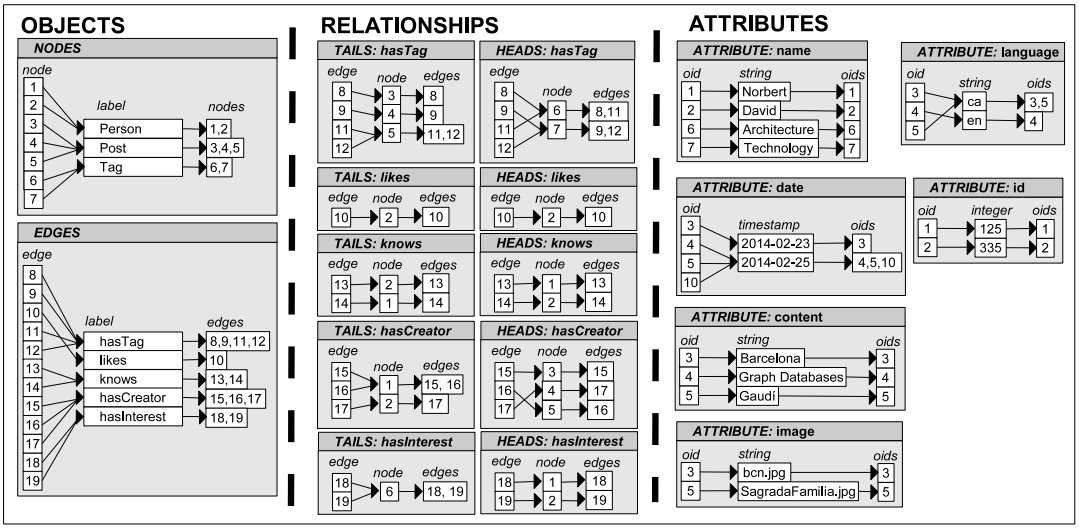

Figure 2: Social network graph represented as a collection of Sparksee links

Notice that an oid can be reached as a $\mathcal{K}$ in an OV or as a $\mathcal{V}$ from its corresponding VOS. And, each scalar datatypes or oid that appears as $\mathcal{V}$ in a OV is also as $\mathcal{K}$ in a VOS. This allows an optimizer to select which is the most convenient way to access to oids and scalar datatypes depending on their usage inside query plans.

OV_TAILS_hasTag	
K	**V**
8	{ 3 }
9	{ 4 }
11	{ 5 }
12	{ 5 }

VOS_TAILS_hasTag	
K	**V**
3	{ 8 }
4	{ 9 }
5	{ 11, 12 }

OV_ATTR_language	
K	**V**
3	ca
4	en
5	ca

VOS_ATTR_language	
K	**V**
ca	{ 3, 5 }
en	{ 4 }

Figure 3: Two links represented as OV and VOS

3.2 Basic operations

In our algebra, each operation receives one or more arguments such as KVPs, filter constants or configuration parameters, and returns one single KVP with the result of the operation. Thus, all operations can be combined by passing the results of one or more operations as parameters to another one. By convention, the reserved keywords of the algebra appears always in lowercase and the variables defined by the programmer are in uppercase.

The basic operations operations of the algebra are relational operations that manage KVP as relations with two columns. With these operations, it is possible to process all the simple graph API such as edge traversal or attribute management. The operations are:

- *Scan(<KVP_name>)*: retrieves one persistent KVP by name. In the paper, we denote the string concatenation as '||' to build the KVP names.
- *Select(R, Exp)*: filters R to restrict the result to those pairs that match the expression Exp.
- *Semijoin(R, S, Exp)*: performs a join between the pairs in R and S that matches the expression Exp, and returns the projection of the columns of the left operand.
- *Equijoin(R, S)*: combines R and S by finding common elements in the $\mathcal{K}$ columns.
- *Group(R, [G_1, ..., G_n], [A_1, ..., A_m])*: computes the aggregates A_i over the pairs in R, optionally grouped by the grouping columns G_j.
- *Sort(R, [S_1, ..., S_n])*: sorts R based on the S_i sorting conditions.
- *Limit(R, N)*: restricts R to a maximum of N pairs.
- *Project(R, Exp, [Exp_1, ..., Exp_n])*: returns a projection over $\mathcal{K}$ and/or $\mathcal{V}$ from R. Expression Exp is the new key (must be unique), and expressions $Exp_{1...n}$ are the new elements of $\mathcal{V}$. If the new key is a collection of oids, then it is split in as many pairs as oids in the collection.
- *Union(R, S), Intersection(R, S)* and *Difference(R, S)*: the classical set-oriented operations over the pairs of two KVPs.
- *Product(R, S)*: returns the cartesian product of the keys and values in R and S, with temporary oid keys for each new pair.

For readability, we write temporary results T_i as "*let* $T_1 = Exp_1, \ldots, T_n = Exp_n$ *in* R". This evaluates the expressions Exp_i and stores their results in the temporary variables T_i. After that, it evaluates R using the T_i temporary results as parameters.

Writing basic graph operations: It is easy to express basic graph queries with the presented operations. For example, we retrieve all the nodes with a certain label or the label associated to a node as:

```
NODES(T) ::= select(scan("VOS_LABELS"), k=T)

NODE_TYPE(N) ::= select(scan("OV_LABELS"), k=N)
```

Finding the edges of a node involves a selection on the VOS_TAILS for the outgoing, or on a VOS_HEADS for the ingoing. If the parameter is a set of nodes instead of a single

one, then it uses a semijoin, which is a generalization of a restriction [10]. For example, to find the outgoing edges of type E starting at a certain tail node or the ingoing from a set of head nodes:

```
EDGES_OUT(N, E) ::=
 select(scan("VOS_TAILS_" || E), k=N)

EDGES_IN(R, E) ::=
 semijoin(R, scan("VOS_HEADS_" || E), k=k)
```

A more frequent operation is to find the neighbors or adjacent nodes following the edges of a certain type. This operation is also known as 1-hop. In our case, it previously obtains the edges, and then looks for the adjacent nodes.

```
NEIGHBORS_OUT(N, E) ::=
 semijoin(scan("OV_HEADS_" || E), EDGES(N, E), k=k)
```

Notice that this traversal involves two operations: one semijoin and one select, or two semijoins when there are more than one starting node. In some cases it is better to have an extra index that materializes the neighborhoods of the nodes by edge type. These optional KVP indexes are named as OUT and IN. Figure 4 details an example of them for the *hasTag* edge type shown on Figure 3.

OUT_hasTag	
K	**V**
3	{ 6 }
4	{ 7 }
5	{ 6, 7 }

IN_hasTag	
K	**V**
6	{ 3, 5 }
7	{ 4, 5 }

Figure 4: Neighborhood KVPs

While with these indexes the expressions are simpler and their evaluation will be more efficient, their construction and maintenance is more expensive, in particular for updates and deletes. Their use should be restricted only to frequently traversed edge types with infrequent removals, which is the case of social networks where relationships are added massively but only a few of them are removed. With the index, the neighborhood operation for a single node is rewritten as:

```
NEIGHBORS_OUT(N, E) ::=
 select(scan("OUT_" || E), k=N)
```

When multiple types of edges are required then the result is the `union` of the same expression executed for each distinct edge type.

A *k-hop* expression is expressed similarly by concatenating *k-1* semijoins after the first select. Thus, by combining multiple traversals and filters we express the graph query *friends of a friend* (FOAF) or, which is the same, the 2-hops from a node:

```
FOAF(N, E) ::=
 let R = scan("OUT_" || E)
 in select(semijoin(R, select(R, k=N), k=v), k<>N)
```

Another graph function, the degree of the outgoing edges of a node, uses aggregates to compute the degree of a node.

```
DEGREE_OUT(N, E) ::=
 group(NEIGHBORS_OUT(N, E), [], [sum(length(v))])
```

Attributes are managed like nodes, edges and relationships. Thus, the value of a node for an attribute is retrieved as:

```
ATTR(N, A) ::= select(scan("ATTR_" || A, k=N)
```

3.3 Procedural operations

In the previous subsection, we have seen how we express simple graph queries with the combination of the operations of KVPs. While some of these operations have an inherent parallelism because they can be transformed into the union of the operation over chunks or partitions of the original KVP (e.g. in select, semijoin, project, etc.), in other queries there is a need for collection oriented computation. For example, to compute all the triangles in a subgraph it is necessary to evaluate a query for each node (or edge) of a subgraph and merge the results in a single KVP.

Furthermore, there are many graph algorithms that obtain a set of results iteratively until an exit condition or some threshold have been reached, such as Breadth First Search (BFS) traversal. For these complex queries, we provide specific procedural operations that receive one or more subqueries that will be executed under certain conditions.

The first procedural operation "*foreach P in R do Q*" evaluates the subquery Q for each pair P in R, and the results are returned as a single KVP merged during a reduce *union* phase.

The second procedural operation is the recursive, which is defined as:

with V_1=P_1, ..., V_n=P_n
do $T_1 = Exp_1$,..., $T_m = Exp_m$ *[until* C | *steps* K*]*
return T_x

where V_i are the initial parameters of the recursive procedure with the contents of the P_i expressions. At each step, the expresions Exp_j after the `do` keyword are evaluated sequentially and the results stored in the temporary variables T_j. The process is repeated until the expression C evaluates to an empty result or an optional maximum number of K iterations have been executed. Thus, the T_j expressions are evaluated at least once with the initial parameters, and the result of the recursive operation is the last content of one of the temporary variables T_x.

With these extensions we find largest hub among all nodes for a given node and edge type:

```
HUB(N, E) ::=
 let D = foreach X in NODES(N)
            DEGREE_OUT(X, E)
 in semijoin(D, group(D, [], [max(v)]), v=v)
```

Additionally, a *BFS* traversal that departs from a node up to a certain depth level K is:

```
BFS(N, E, K) ::=
 with R=N
 do S=R,
    R=NEIGHBORS_OUT(S, E)
 until semijoin(R, S, k<>k)
 steps K
 return R
```

In this section, we have presented the algebra of operations used to build the query engine of the Sparksee graph database engine. We have seen that the traversal of edges, one of the most frequent operations in graph queries, is solved with the concatenation of two or more semijoins, and even a recursive k-hop algorithm performs semijoins iteratively until another semijoin raises the end condition. In the next section we analyze this use of semijoins as the pivotal operation inside Sparksee query plans.

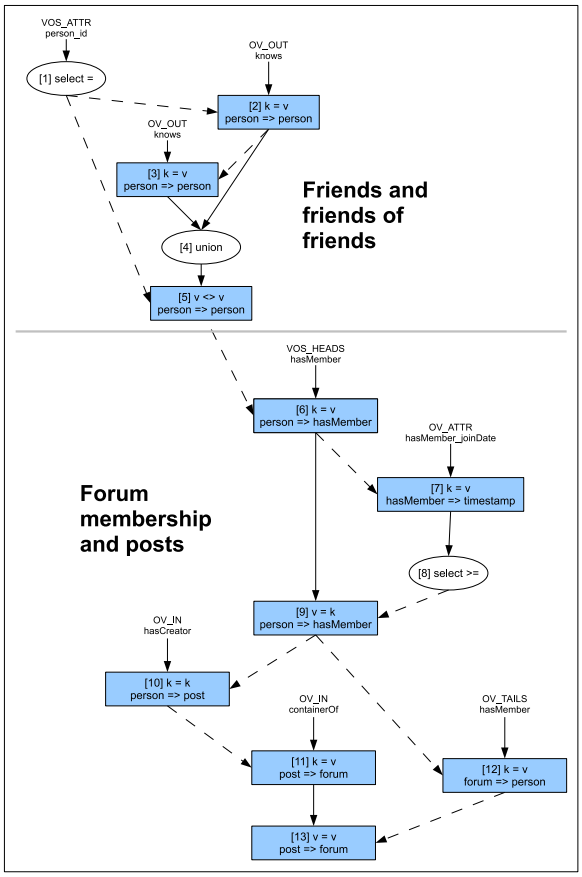

Figure 5: SN Benchmark Q5

4. QUERY ANALYSIS

In this section, we analyze the usage of semijoins in real graph queries, and we pinpoint some optimizations that could be evaluated and implemented in future Sparksee versions. For the evaluation of the proposal, we use the Interactive Social Network Benchmark proposed by the Linked Data Benchmark Council [9] (LDBC). This benchmark aims at evaluating RDF and graph database technologies.

We analyze the first interactive queries documented in the SNB workload generator (QGEN) to test our proposal. LDBC also provides a synthetic dataset generator (DBGEN) that builds a complex social network, which mimics real scenarios such as Facebook or Twitter. We loaded in Sparksee v5.0 the validation datasets with 100K users during 3 years of activity to verify the correctness of the query plans. The latest versions of DBGEN and QGEN were downloaded at the end of March 2014 from the LDBC GitHub public repository [3]. For the analysis of semijoin programs in Sparksee, first we examine some common graph operations that translate into semijoin programs, and later we quantify the usage of semijoins and other operations in LDBC queries.

4.1 Semijoin programs

One of the LDBC queries, Q5, is a good example of typical graph operations that translate into semijoins in Sparksee. This query *looks for the forums that the friends and friends of the friends of one person have joined after a certain date; for each of these forums, the query returns the number of posts they made there*. Figure 5 shows an optimized query plan, where semijoins are represented as boxes and the other operations as ellipses. Each operation has a numeric identifier. A semijoin always has two inputs: the vertical solid line is for the left operand, which is going to be the source for the result, while the dashed lateral line comes from the right operand as the filter of the semijoin. Semijoins also have extra information: the two columns of the comparison (k for key or v for values), the signature of the result (node or edge types, datatypes...). The figure is also divided in two blocks, which are the two examples that we are use to analyze the semijoin usage: the upper of the query plan is a 2-hop operation, the classical friends and friends of a friend search, and the rest of the query plan is the equivalent of a pattern matching expression with a filter on the edges.

First, notice that in both examples all semijoins operations compare oids or collections of oids, and values of other datatypes are accessed only inside `select` filters. It is important to remember that the internal implementation of the persistent OV and VOS KVPs in Sparksee have the unique keys indexed, and that collections of oids are always compressed in the form of bitmaps. Thus, we distinguish two different combinations of semijoins in query plans:

(a) The left operand is the key. This is usually the case when the left argument of the semijoin is a persistent KVP and the key is indexed:

- `k=v` in 2, 3, 6, 7, 11 and 12: the semijoin selects the left keys that exists in at least one of the right collections of oids. Depending on the cardinalities there are multiple optimizations, such as iterating over the right collections to lookup in the left index for the keys, or a more specific optimization for bitmaps that first obtains the union of all the right collections and then test the keys only once onto this merged collection.

- `k=k` in 10: this is a particular case of the previous one, where the right collections have only a single member.

(b) The left operand is a collection of oids. This is usually the case when the left argument of the semijoin is an intermediate result inside the query pipeline:

- `v=v` in 13 and v=k in 9: this is a particular case of the first one with a collection of oids instead of single keys, but by default there is no index available for lookups unless the query engine indexes temporary results on the fly. Thus, one optimization is to perform intersections between the bitmaps of the left and right collections of oids. Also, all the right collections can be merged in a single bitmap to reduce the number of comparisons.

- `v<>v` in 5: as before, but now with the set-difference operation instead of the intersection.

In the query plan we also observe the appearance of semijoin programs [10], which are sequences of semijoins. Semijoin programs are interesting because they reduce significantly the size of the intermediate results, in contrast to joins, and allow optimizations based on the swapping of operations. Also, in a distributed architecture, semijoins can be computed with less intersite data transfer joins.

4.2 Semijoin usage in queries

We analyze quantitatively each algebraic operation that appears in the subset of queries of SNB. Table 1 reports the frequency per query. There are two extra columns: one for the total count of each operation, and another with the percentage. Values in bold identify the largest value in a column. In this table, we observe that in five of the six queries the most frequent operation is the semijoin, and in Q3 it is close to the most frequent. Furthermore, semijoin accounts for 28.1% of the total operations, while another 25.3% are scans. We detected that many of those scans are the input of semijoin programs. This means that more than 50% of the operations are scan and semijoin, and this is the first hint of potential optimizations in the access methods to the KVP and in their interactions with semijoins.

Operation	Q1	Q2	Q3	Q4	Q5	Q6	Total	%
EQUIJOIN	16	6	4	1	1	1	29	11.6%
FOREACH	3	4	1				8	3.2%
GROUP	4		2	1	1	1	9	3.6%
LIMIT	1	1		1	1	1	5	2.0%
PRODUCT	4	3	2				9	3.6%
PROJECT	12	7	5	1	1	1	27	10.8%
RECURSIVE							0	0.0%
SCAN	23	**14**	7	6	7	6	63	25.3%
SELECT	3	3	4	3	2	2	17	6.8%
SEMIJOIN	**30**	13	**11**	**7**	**8**	**10**	**79**	**28.1**%
SORT	1	1		1	1	1	5	2.0%
UNION	1	3	1		1	1	7	2.8%

Table 1: Operations

Table 2 counts the semijoins grouped by the join columns. The first column in the table identifies the columns of the KVP involved in the semijoin: k for the key, and v for the value. We observe that more than 90% of the semijoins have a key as the left operand in the comparison, which is an oid from a persistent KVP. This is a second hint of optimization based on indexed access methods to the Sparksee graph storage. Also, more than 54% of the semijoins involve collections of values, which can be solved by using intersections of compressed bitmaps. We noticed that in all joins where a k appears, this column always contains oids, though our model assumes no limitation on this.

Semijoin	Q1	Q2	Q3	Q4	Q5	Q6	Total	%
k-k	23	6	2	4	1	2	38	46.9%
k-v	7	7	8	3	6	5	36	44.5%
v-k					2	2	4	4.9%
v-v			1		1	1	3	3.7%

Table 2: Traversals

Length	Q1	Q2	Q3	Q4	Q5	Q6
1	10	6	2		1	1
2	6	3	3		1	1
3	3	2		2	1	
4	2					
5			2			
6					1	
7						1

Table 3: Traversal lenghts

Finally, Table 3 contains the lengths of semijoin programs in each query. We see that, while most of the sequences of semijoins are short, there exists large semijoin programs with lengths up to 6 or 7 consecutive semijoins. Also, some semijoins results are reused in more than one posterior operation creating semijoin trees, as shown previously in Figure 5. This opens another optimization area based in the reorganization of semijoin programs by applying the properties of semijoins. For example, in some cases semijoins can be swapped in a semijoin program to reduce the number of comparisons. Other optimizations to be considered are those presented in [19] for the use of bit-arrays (bitmaps) in semijoins, or Bloom filters to reduce the number of candidate comparisons as proposed in [17].

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented an algebra for resolving graph operations in Sparksee. The basic construct of the query plans are semijoin programs that operate on pipelines of key-value pairs. Although most of the operations are relational based, the semijoins on collections of oids can take advantage on the compression capabilities of the Sparksee bitmap storage. Our next steps are to implement the prototype of the query engine with full support of the algebra, and to test and validate the performance of the proposal. After that, we will focus on query plan optimization for semijoin programs combined to our non-relational extensions for the resolution of complex queries over large graphs.

Acknowledgments

This work was partially supported by the EU FP7 projects LDBC (FP7-ICT-2011-8-317548) and CoherentPaaS (FP7-ICT-2013-10-611068). Norbert Martínez thanks Generalitat de Catalunya for the grant DI-13-069, and David Domínguez thanks Ministry of Science and Innovation of Spain for the grant Torres Quevedo PTQ-11-04970.

6. REFERENCES

[1] Cypher. http://www.neo4j.org/learn/cypher/.
[2] Gremlin. https://github.com/tinkerpop/gremlin/wiki/.
[3] LDBC SNB. https://github.com/ldbc/ldbc_socialnet_bm/.
[4] Neo4J. http://www.neo4j.org/.
[5] RDF. http://www.w3.org/RDF/.
[6] Sparksee. http://www.sparsity-technologies.com/.
[7] SPARQL. http://www.w3.org/TR/sparql11-query/.
[8] Titan. http://thinkaurelius.github.io/titan/.
[9] R. Angles et al. The Linked Data Benchmark Council: a Graph and RDF industry benchmarking effort. In *SIGMOD Record (accepted, unpublished)*. ACM, 2014.
[10] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *JACM'81*, 28(1):25–40, 1981.
[11] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. of the ACM SIGMOD'08*, pages 405–418. ACM, 2008.
[12] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH'12*, volume 40, pages 349–362. ACM, 2012.
[13] Y. Low et al. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.
[14] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *Proc. of the ACM SIGMOD'10*, pages 135–146. ACM, 2010.
[15] N. Martinez-Bazan et al. Dex: A high-performance graph database management system. In *Proc. of ICDEW'11*, pages 124–127, Washington, DC, USA, 2011. IEEE Computer Society.
[16] N. Martínez-Bazan et al. Efficient graph management based on bitmap indices. In *Proc. of IDEAS'12*, pages 110–119. ACM, 2012.
[17] J. K. Mullin. Optimal semijoins for distributed database systems. *Software Engineering, IEEE Transactions on*, 16(5):558–560, 1990.
[18] S. Sakr, S. Elnikety, and Y. He. G-sparql: a hybrid engine for querying large attributed graphs. In *Proc. of the ACM CIKM'12*, pages 335–344. ACM, 2012.
[19] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems (TODS)*, 9(1):133–161, 1984.