Parasol: An Architecture for Cross-Cloud Federated Graph Querying

Michael D. Lieberman[†], Sutanay Choudhury[‡], Marisa Hughes[†], Dennis Patrone[†], Robert T. Hider, Jr.[†], Christine D. Piatko[†], Matthew Chapman[†], J. P. Marple[†], David Silberberg[†]

Michael.Lieberman@jhuapl.edu   Sutanay.Choudhury@pnnl.gov

ABSTRACT

Large scale data fusion of multiple datasets can often provide insights that individual datasets cannot. However, when these datasets reside in different data centers and cannot be collocated due to technical, administrative, or policy barriers, a unique set of problems arise that hamper querying and data fusion. To address these problems, a system and architecture named Parasol is presented that enables federated queries over graph databases residing in multiple clouds. Parasol's design is flexible and requires only minimal assumptions for client clouds. Query optimization techniques are also described that are compatible with Parasol's lightweight architecture. Experiments on a prototype implementation of Parasol indicate its suitability for cross-cloud federated graph queries.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*distributed databases, query processing*

General Terms

Algorithms, Design, Performance

Keywords

Graph database, federated database, data fusion, query optimization

1. INTRODUCTION

Oftentimes, answering the most interesting questions about data at scale requires fusing data present in multiple datasets. Here, we focus on graph data, since graphs have a number of properties that facilitate data fusion, such as having easily extensible schemas and being able to represent many disparate types of data in graph form. Graphs also provide a natural means of expressing queries on data. Such a query might involve searching for local structures within a large graph—i.e., a subgraph query. This is also known as the *subgraph isomorphism* problem, which is known to be computationally and data intensive. The challenge is exacerbated in the context of data fusion, when parts of the query structure reside in different datasets. It is this last challenge that we address in this work.

Querying, analyzing, and fusing data at scale generally requires the use of large data clouds residing in data centers. However, when the datasets of interest exist in different clouds, possibly owned by different organizations, this affords multiple challenges. Ideally, all the datasets would be moved to a single cloud, but this may not be technically feasible. Data centers may be physically distant or rely on different cloud technologies. Likewise, administrative costs may be prohibitive for supporting such complex architectures. Finally, there may be policy issues acting as barriers to data collocation and integration, such as privacy or data sharing concerns.

With this work, we hope to call attention to the above challenges, and address some of the technical aspects. To this end, we present Parasol, a system and architecture for performing federated graph querying across multiple clouds. Our approach in designing Parasol was to be flexible by assuming minimal control over client clouds in the cross-cloud framework, and minimal knowledge of the data present in each cloud. For a cloud to participate, it implements a simple client interface, requiring the streaming of partial results for a small portion of the query. Parasol's central coordinator process, implemented using the Hadoop framework and the NoSQL Accumulo backend database, then gathers all results and merges them to form answers to the query. In this way, clouds maintain a large degree of independence from other client clouds and from Parasol itself. One caveat with a generic architecture like that of Parasol is that it limits the possibilities for using some types of query optimization. But, as we will show, even with such minimal assumptions, it is still possible to perform some query optimization and achieve reasonable performance, as indicated by our experiments on a prototype implementation of Parasol. We believe that Parasol's generic approach to federated graph queries is a promising first step toward scalable cross-cloud querying.

Parasol is related to previous work in federated databases and distributed querying, which have been a topic of research for many years. Traditional query optimization techniques, such as the Magic Sets algorithm [2] and others (e.g., [4, 12, 15]) have been developed for distributed query optimization and execution. An example of a recent distributed query algorithm approach is Horton [14]. One of its key design features is its graph partitioning scheme and use of memory to increase performance. In a similar vein, Sun et al. [16] describe a subgraph matching algorithm for very large graphs, in which they leverage a novel indexing scheme and make use of a shared distributed memory store. PowerGraph [5] is a more general graph computation framework tuned for power law graphs. Our work is also related to RDF triple stores for the Semantic Web. There has been recent focus on scalable querying of such RDF stores, using query languages such as SPARQL [6, 8], query cost-estimation techniques [13], and indexing schemes [3]. Finally, although we do not address it in this work, *schema matching* [11] may be required for successful data fusion of disparate datasets. All these approaches, and many others, rely on having tight control over data distribution, cluster topology, and other elements. In contrast, for our work, we make minimal assumptions about the clouds being queried.

We continue with reviews of Parasol's architecture (Section 2), issues in query optimization (Section 3), experiments to test Parasol's effectiveness (Section 4), and conclude with lessons learned and ideas for future work (Section 5).

[†]The Johns Hopkins University Applied Physics Laboratory, 11100 Johns Hopkins Rd., Laurel, MD 20723

[‡]Pacific Northwest National Laboratory, 902 Battelle Blvd., Richland, WA 99354

DanaC'14, June 22 2014, Snowbird, UT, USA
Copyright 2014 ACM 978-1-4503-2997-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2627770.2627771

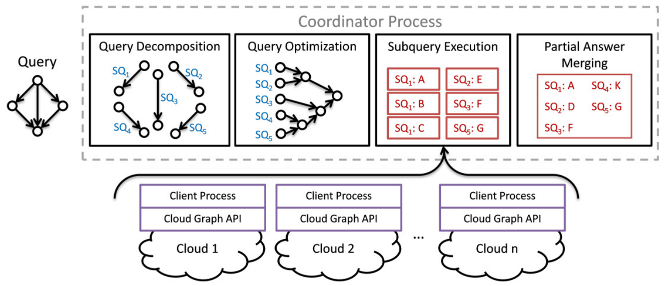

Figure 1: Parasol's architecture. A central coordinator process manages query execution over multiple client clouds.

2. ARCHITECTURE

Parasol's architecture is illustrated in Figure 1. The architecture is defined in terms of two lightweight processes: a *coordinator* process running in a coordinator cloud, and multiple *client* processes running in client clouds. Users interact with the coordinator by sending it graph queries. The coordinator then decomposes each query into appropriate subqueries, performs query optimization to devise a query plan (represented as a *SJ-tree* in Figure 1—see Section 3.1), and delegates subquery execution to client processes as appropriate. The clients execute the subqueries, returning their partial results to the coordinator, which then merges them to produce final answers that are returned to the originating user. The coordinator is implemented using Apache Hadoop, and uses the Accumulo NoSQL database for temporary storage of partial results. Merging is implemented as a sequence of MapReduce jobs, each of which merges two sets of partial results. The SJ-tree structure determines the job sequence.

Importantly, client processes assume no particular infrastructure for the client clouds. Instead, they only provide the logic to communicate with the coordinator, and a minimal API that the client clouds are responsible for implementing, which is a fairly standard practice in data integration. The API includes methods for computing answers to individual subqueries, which is the minimal functionality for a working federated graph query system. Also included are optional methods that enable query optimization (see Section 3 for details). While using this approach involves creating a new implementation for each new data source, and updating the implementation if the data source changes, the significant advantage is that each implementation can take advantage of site-specific information to speed queries (e.g., specialized indexes, MapReduce integration, etc.) which will be critical at cloud scales. Also, we do not assume a priori knowledge of the data within individual client clouds. If a data schema could be provided, additional execution and merging optimizations could be implemented using that knowledge (e.g., ignoring a subquery that not pertain to any information stored in a client cloud), at the cost of having to categorize and coordinate data descriptions across clouds.

To illustrate the workings of our architecture, we provide an example of end-to-end query execution, using Figure 3a as an example query Q_e. Q_e contains four variables and a total of five components, each of which represents a relationship between nodes. (For this discussion, we ignore constraints on node types.) First, the coordinator decomposes Q_e into its five components, SQ_1–SQ_5. The coordinator then performs query optimization, thereby ordering the join operations needed to construct final answers. Details of query optimization are provided in Section 3. Next, the coordinator sends SQ_1–SQ_5 to each client cloud, and collects the corresponding sets of matching partial results, P_1–P_5. We will refer to sets of partial results as *k-partials*, where k is the number of components satisfied by results in the set. Since P_1–P_5 have one matching component, we call these *1-partials*. In Figure 1, one such 1-partial is [SQ_1: A], indicating that the "A" edge in the data graph matches SQ_1.

After the coordinator has collected all 1-partials from the clients, it proceeds by iteratively merging these into higher order partials, merging those whose variables are consistent. Visually, this can be understood as joining edges in the query that share nodes. For example, the 1-partials [SQ_1: A] and [SQ_2: E] may be merged to form the 2-partial [SQ_1: A, SQ_2: E] as long as their shared variable ("P") maps to the same data node. After creating all consistent combinations, the original partials are deleted. The coordinator continues to merge partials until reaching a final set of k-partials, where k is the number of query components. These solutions are returned to the user.

3. QUERY OPTIMIZATION

In this section, we discuss Parasol's generic framework for query optimization. As mentioned before, Parasol was designed with flexibility in mind, making minimal assumptions about each client cloud's architecture and capabilities, which provides for easier cloud integration. This genericity extends to Parasol's query optimization design. Parasol's coordinator asks each client cloud for statistics related to execution of a particular query. The client clouds can but are not required to provide such statistics in return, and Parasol will use whatever information is returned. Of course, better statistics will result in a better query plan. We have created a limited implementation of query optimization for our Parasol prototype, described below; however, future improvements are planned (see Section 5).

To ground our discussion, we return to the query of Figure 3a. The coordinator receives sets of 1-partials of five types (SQ_1–SQ_5) from the clouds, and merges the partials that share a common node. Formally, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we define the *join* operation $G_1 \bowtie G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. Applied to the subgraphs of Figure 3a, for example, SQ_1 shares a common node with SQ_3 ("P") and SQ_4 ("F"). Given that SQ_3 and SQ_4 also share a common node ("L"), we can reach a 3-partial corresponding to [SQ_1, SQ_3, SQ_4] following any of three different join orders: ($SQ_1 \bowtie SQ_3$) $\bowtie SQ_4$, ($SQ_1 \bowtie SQ_4$) $\bowtie SQ_3$, and ($SQ_3 \bowtie SQ_4$) $\bowtie SQ_1$.

Unless a unique join order is established, we will repeatedly produce the same output following different join orders. This brings us to the well known *join order selection* problem [10], which asks for a join order that minimizes the number of k-partials created in the intermediate stages of query processing. Unfortunately, typical cost models used in relational database optimizers rely on indexing statistics or scanning the data, and many of these models are less readily applicable to cloud-scale graph databases. Many of the prominent solutions to subgraph isomorphism (e.g., [19]) rely on indexing the neighborhood of individual vertices for structural and semantic features. Typically, these algorithms explore a k-neighborhood of each vertex (k is often set to 2). However, the cost of indexing becomes prohibitive as the number of vertices approaches billions [16].

Instead, our current work is restricted to using a simple greedy heuristic for join order determination inspired by work in relational databases: *left-deep* joins. In a relational context, this heuristic calls for using a new base table as input to each subsequent join operation, rather than the results of intermediate joins. The term comes from the resulting join plan when viewed as a tree (see Section 3.1). For our work, by analogy, this means using at least one set of 1-partials as input to each join. Using this heuristic was motivated by an extensive survey of the literature on optimal join order determination in relational databases (e.g., [7, 9, 18]). A key conclusion of the survey states that left-deep join plans are among the best performing heuristics. As alternatives, the above mentioned studies point to a large body of research using techniques such as dynamic programming and genetic algorithms to find the optimal join order. Approaches based on minimum spanning trees or approximate vertex cover can also provide initial paths forward. Nonetheless, finding the lowest cost join order or using a cost-driven join order determination remains a difficult problem in graph databases, and we adopt a simpler approach.

3.1 SJ-Tree

We express our query plan as a *Subgraph Join tree (SJ-tree)*, which defines the order in which partial results for a query graph Q will be joined. Figure 2 is an SJ-tree for the query in Figure 3a. An SJ-tree T for Q is a binary tree comprised of the node set N_T, where each $n \in N_T$ corresponds to a subgraph of Q. The subgraph corresponding to the root of T is isomorphic to Q, and the subgraph corresponding to any $n \in N_T$ of T is isomorphic to the output of joining the subgraphs corresponding to n's children. We use a binary tree struc-

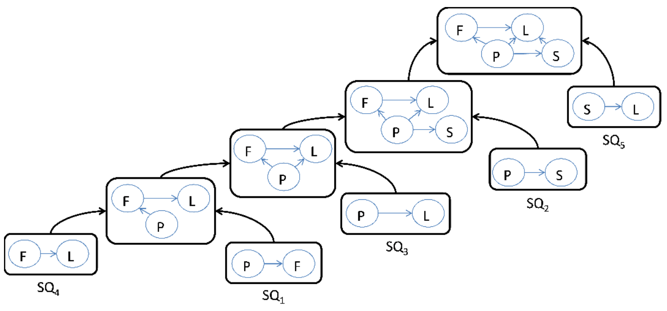

Figure 2: A sample SJ-tree for the query in Figure 3a.

ture, rather than a k-ary tree, for the sake of simplicity and to avoid the combinatorial cost of joining matches from multiple children.

Thus, query planning amounts to constructing an SJ-tree for the given query Q. Each leaf of the SJ-tree represents a subgraph that we will request from individual clouds to perform subgraph isomorphism, the results of which will consist of 1-partials in our setup. Internal nodes of the SJ-tree represent subgraphs that result from the joining of subgraphs returned by the subgraph isomorphism operations. The SJ-tree will be built greedily from the bottom up, with the most selective primitives being added first, leveraging the left-deep heuristic by joining a new set of 1-partials at each step. This is reflected in the SJ-tree in Figure 2. Also, note that the SJ-tree is not used for data indexing, but rather for query planning. Of course, producing accurate selectivity estimates for use in SJ-tree construction may require indexes, but this is left to each client cloud's discretion.

3.2 SJ-Tree Construction

We introduce an algorithm for SJ-tree construction, BUILD-SJ-TREE (Algorithm 1). The algorithm assumes the availability of frequency information of small subgraphs termed *primitives*, such as single edges, 2-edge paths, or triangles [1, 17], which can be efficiently queried from the data graph G. For each primitive g_M, we also assume availability of a *selectivity* estimate for g_M; higher selectivity implies use of g_M will keep the number of intermediate k-partials small. The client clouds in our prototype implementation of Parasol provide single edge primitives, and the selectivity of each g_M is computed as $1/n(g_M, G)$, where $n(g_M, G)$ is the number of occurrences of g_M in G. More sophisticated selectivity measures can be used, but clouds will vary as to what estimation capabilities they have. For our proof-of-concept, a simple count suffices.

Inputs to Algorithm 1 are the query graph Q and a set of primitives M, sorted by descending selectivity. Our goal is to decompose Q into a collection of (possibly repeated) subgraphs chosen from M, which will in turn define the SJ-tree and the join ordering. The algorithm begins by initializing the output SJ-tree T as a single node with a subgraph of Q that matches $M[1]$, the most selective primitive, which is then removed from Q (lines 1–2). The algorithm iterates by removing portions of Q and adding them to the final SJ-tree T in a bottom-up manner, until Q is empty (lines 3–9). In each iteration, we loop over the primitives (lines 4–7), and for each primitive g_M we attempt to find a subgraph g_{sub} of Q that is isomorphic to g_M (line 5). If we succeed, and further, g_{sub} shares a vertex (i.e., overlaps) with a subgraph in T (line 6) then we break out of the for loop, build the next T as a node with the current T and g_{sub} as children, and remove g_{sub} from Q (lines 8–9). At termination, we have an SJ-tree which will serve as our query plan.

Algorithm 1 BUILD-SJ-TREE(Q, M)

```
1:  $T \leftarrow$ FIND($M[1], Q$)
2:  $Q \leftarrow Q \setminus T$
3:  while $|Q| > 0$ do
4:      for all $g_M \in M$ do
5:          $g_{sub} \leftarrow$ FINDISOMORPHIC($g_M, Q$)
6:          if $g_{sub} \neq \emptyset \wedge$ SHARESVERTEX($g_{sub}, T$) then
7:              break
8:      $T \leftarrow (T, g_{sub})$
9:      $Q \leftarrow Q \setminus g_{sub}$
10: return $T$
```

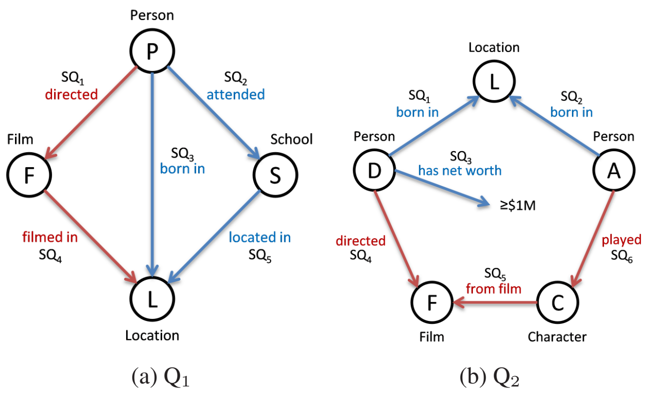

Figure 3: Queries used in our experiments. Red and blue correspond to data present in LMDB and BaseKB, respectively.

4. EXPERIMENTS

In this section, we describe some preliminary experiments for testing Parasol's efficacy on cross-cloud federated graph queries. All experiments were performed on two clouds. The first cloud contained three virtual machines, each with 4 Opteron 6172 CPUs at 2.1GHz and with 8G RAM. The second cloud had five compute nodes, each with 16 Xeon E5620 CPUs at 2.40GHz and 48G RAM. This configuration demonstrates the utility of our framework for querying across clouds with variable configurations.

4.1 Datasets

We chose two datasets with mostly distinct and some overlapping information: the Linked Movie Database (LMDB)[1] and BaseKB[2]. LMDB is a semantic database based on the Internet Movie Database, and contains information related to films, actors, directors, characters, etc. BaseKB is a linked database extracted from structured information present in Wikipedia. Persons and locations are shared by both datasets in a consistent way, enabling cross-dataset querying. We created subsets for our experiments, including only persons from the datasets who acted in at least 10 films, directed at least 5 films, or were associated with a location. Similarly, only characters associated with one of the above persons were included. The LMDB subset contained about 203k nodes and 820k edges, while the BaseKB subset had about 56k nodes and 230k edges. As we will show, these subsets contain enough variety to be able to test relatively complex queries.

4.2 Results

We designed two queries for our experiments, shown in Figure 3. Edge colors indicate the source dataset: red for LMDB, blue for BaseKB. The queries were designed to test different query complexities, and involving the different data types present in our datasets. Q_1 asks for a person P who directed a film F and also attended school S in the same location L as they were born. Q_2 asks for a director D with a net worth greater than \$1M who directed a film F, and an actor A who played character C in F and was born in the same location L as the director. Together, these queries demonstrate the relatively complex questions able to be answered with Parasol's design.

We executed our queries on our clusters and collected several metrics for each query. Table 1 summarizes our results. First, we note that although execution time was not a focus of our initial efforts, and despite the complexity and sizes of our datasets, each query finished in a reasonable time, indicating the practicality of our approach. Next, examining the numbers of 1-partials generated, all match a fairly large fraction of our datasets, which provides challenging use

[1]http://linkedmdb.org/
[2]http://basekb.com/

Table 1: Query results.

	Q_1	Q_2
LMDB 1-partials	40.5K	257K
BaseKB 1-partials	50.4K	62.9K
Merge comparisons	8.32B	2.17B
Intermediate partials	385K	344K
Answers	55	1
Running time	02:35:23	01:32:06

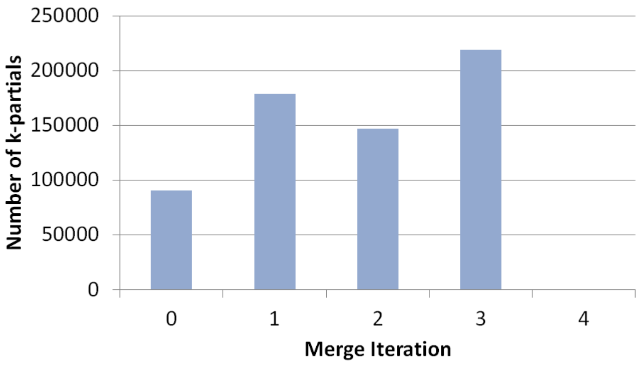

Figure 4: Number of intermediate k-partials for each merge iteration during execution of Q_1.

cases for our queries. In contrast to the 1-partial counts, the number of merge comparisons—how many pairs of partials were compared to see if they could merge—varied widely between queries, and it seemed that this dominated the overall execution time. Interestingly, the simpler query Q_1 required almost four times as many merge comparisons as the more complex Q_2. The number of intermediate partials generated during all the query executions was comparable.

We examined in more detail the number of k-partials during each partial merge iteration during the execution of Q_1. Results are shown in Figure 4. Prior to merging, the total number of partials was almost 100K, which steadily increased with each merge iteration. Interestingly, the total number of partials decreased at step 2. This can be explained as a favorable merge ordering decided by our prototype implementation of Parasol's query optimizer, and indicates the utility of query optimization in cross-cloud graph queries.

5. CONCLUSION AND FUTURE WORK

We have shown that a flexible architecture such as that of Parasol has promise to support effective cross-cloud graph querying. We are currently exploring additional improvements. First, Parasol could take advantage of additional optimization techniques (see Section 3) were some of its assumptions to be relaxed. For example, if clouds could report statistics related to partial results—e.g., number of results for each subquery component, distribution of node ids over query variables, etc.—these statistics could be used by the coordinator to inform the join order of partial results. Another assumption which could be relaxed is to allow client clouds to optionally merge partial results in place prior to its communication with the coordinator, reducing network traffic and resource contention. We also plan to further explore the benefits and drawbacks to our SJ-tree approach, and compare it with other query optimization techniques. While the difficulty of data fusion increases with data size and complexity, architectures such as that of Parasol can effectively address the need for cross-cloud federated querying.

6. REFERENCES

[1] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st Symposium on Principles of Database Systems*, PODS'12, pages 5–14, Scottsdale, AZ, May 2012.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS'86, pages 1–15, Cambridge, MA, Mar. 1986.

[3] F. Bugiotti, F. Goasdoué, Z. Kaoudi, and I. Manolescu. RDF data management in the Amazon cloud. In *Proceedings of the Workshop on Data Analytics in the Cloud*, DanaC'12, pages 61–72, Berlin, Mar. 2012.

[4] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE'02, pages 716–727, San Jose, CA, Feb. 2002.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Hollywood, CA, Oct. 2012.

[6] O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL queries over the web of linked data. In *Proceedings of the 8th International Semantic Web Conference*, ISWC'09, pages 293–309, Washington, DC, Oct. 2009.

[7] J. M. Hellerstein and M. Stonebraker. Predicate migration: optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD'93, pages 267–276, Washington, DC, May 1993.

[8] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, 4 (11):1123–1134, Aug. 2011.

[9] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB'86, pages 128–137, Kyoto, Japan, Aug. 1986.

[10] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment*, 6(2):133–144, Dec. 2012.

[11] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, Dec. 2001.

[12] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, May 1995.

[13] E. Ruckhaus, E. Ruiz, and M.-E. Vidal. Query evaluation and optimization in the semantic web. *Theory and Practice of Logic Programming*, 8(3):393–409, May 2008.

[14] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE'12, pages 1289–1292, Washington, DC, Apr. 2012.

[15] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, Sept. 1990.

[16] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, May 2012.

[17] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW'11, pages 607–614, Hyderabad, India, Mar. 2011.

[18] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of the 19th International Conference on Data Engineering*, ICDE'03, pages 443–454, Bangalore, India, Mar. 2003.

[19] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1–2):340–351, Sept. 2010.