An Adaptive SPARQL Engine with Dynamic Partitioning for Distributed RDF Repositories

Thesis by

Yasser Elsayed Ibrahim, B.Sc. Computer Science

Submitted in Partial Fulfillment of the Requirements for the degree of

Master of Science

King Abdullah University of Science and Technology

Mathematical and Computer Sciences and Engineering Division

Computer Science Program

Thuwal, Makkah Province, Kingdom of Saudi Arabia

July, 2012

The undersigned approve the thesis of Yasser Elsayed Ibrahim

Dr. Spiros Skiadopoulos Committee Member	Signature	Date
Dr. Basem Shihada Committee Member	Signature	Date
Dr. Panos Kalnis Committee Chair(Thesis Supervisor)	Signature	Date

King Abdullah University of Science and Technology

2012

ABSTRACT

The tremendous increase in the semantic data is driving the demand for efficient query engines. RDF data being generated at an unprecedented rate introduces a storage, indexing, and querying challenge. Due to the size of the data and the federated nature of the semantic web, it is in many cases impractical to assume a central repository, and more attention is being given to distributed RDF stores. This work is motivated by two major drawbacks of current solutions: 1) pre-processing part is very expensive and takes prohibitively long time for large datasets, and 2) current distributed systems assume that a static partitioning of the data should perform well for all kinds of queries, and do not consider fluctuations in the queryload.

In this paper we propose PHD-Store, an in-memory SPARQL engine for distributed RDF repositories. Our system does not assume any particular initial placement of the data and does not require pre-processing before running the first query. It analyzes incoming queries and adjusts data placement dynamically in such a way that communication among repositories is minimized for future queries. To achieve this flexibility, frequent query patterns are detected, and data are redistributed through a Propagating Hash Distribution (PHD) algorithm to ensure optimal placement for frequent query patterns. Our experiments with large RDF graphs verify that PHD-Store scales well and executes complex queries more efficiently than existing systems.

ACKNOWLEDGMENTS

I would like to sincerely thank my supervisor Dr. Panos Kalnis for his continuous guidance and encouragement throughout the course of this work. His enthusiasm and valuable feedback for research made my study very enjoyable and exciting and ultimately fruitful with rich experience. I would also like to thank him for providing me with an amazing research environment.

I thank my parents for their continuous encouragement and my siblings for bearing with me for my negligence towards them during this journey and their deep moral support at all times.

Lastly, I would like to thank the people at KAUST, Thuwal, Makkah Province, Saudi Arabia for providing support and resources for this research work.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter 1

INTRODUCTION

The Semantic Web links diverse data sources in a form easy to be processed by machines, in order to extract new facts and interesting connections. Sources vary from social networks and online retailers to scientific databases and wireless sensor readings. Such sources generate significantly large amounts of data that present a problem in the face of modern storage, indexing, and query processing technologies. Due to the nature and size of RDF data, the need rises for building distributed solutions to store and analyze large magnitudes of data in a short time to allow for rapid consumption of huge streams of data. Large-scale initiatives such as The Large Synoptic Survey Telescope [1] or The Large Hadron Collider [2] are expected to generate terabytes of data everyday that need fast analysis.

We focus our work on RDF [3], a versatile, graph-like representation of linked data that is gaining growing attention in a wide array of applications. Although a great deal of research has been directed to handling large relational data sets, only few works in the literature have tried to tackle the problem of parallel and distributed processing of large RDF data efficiently.

For RDF queries, the single most important operator is join. In a distributed setting, performance of join execution is bound by two factors: *computation* and

communication. Computation is usually optimized in one of two ways: first, directing each query to one worker to process simultaneously, so that the system benefits from multi-query optimization. Alternatively, the query is split between workers so that it is processed in parallel then the results are aggregated. PHD-Store adopts the second approach. On the other hand, communication is optimized by good placement of data, which minimizes the amount of data shared between different workers. It is critical to reduce needed communication for any distributed systems, since it can easily become a bottleneck for performance and scalability. However, achieving a good data placement is no simple task for RDF, since there is no schema which the system can rely on to guide the placement.

Early attempts revolved around hash distribution on a specific column in order to improve queries that involve joins on this column. Since RDF data resembles a large three-column table, a specific column can be chosen a-priori so that data are distributed according to the hash of this column. All queries involving joins on this column can then be carried out in parallel on all machines without communication. Consequently, hash distribution forces limitations on join columns to be the distribution column only. Queries involving different join columns, and more complicated queries with multiple joins will not benefit from the hash distribution. Therefore, more sophisticated solutions for the join had to be considered.

More recent attempts at better placement have considered applying some minimal cut algorithms. They rely on the idea that graph vertices that are related to each other will be queried together with a high probability. An input data set can be partitioned in such a way that minimizes cross-machine edges in order to increase query locality. In other words they argue that minimal cut partitioning of the data increases the probability that a query (or at least a big portion of it) is executed locally in each worker, thus reducing the total communication needed for solving the query.

Despite the obvious advantage of this approach over traditional hash distribution, we argue that it is not enough to result in a good data placement. The main drawback of this approach is that it is static in the face of query load changes. Moreover, RDF graphs tend to be well connected, and for any particular partitioning of the data, there can be queries that require communication between several workers. If such queries dominate the workload, this partitioning may not be any more useful than random partitioning.

Aside from the aforementioned drawbacks, one of the greatest challenges with RDF, and graph data generally, is the complexity of the pre-processing steps for the distributed setting, mainly due to the natural locality characteristics of the data. In this work, we use the term pre-processing for all kinds of data preparation, statistics collection, and indexing that are run against the data before the system starts answering the first query. Minimal cut algorithms are particularly expensive when working in the distributed setting. Considering the very large sizes of RDF datasets, such an expensive pre-processing step can be prohibitively long, possibly taking days before running the first query. In addition, even if only few queries will need to be run, or if only part of the graph is interesting for querying, the entire graph still needs to be indexed.

We argue that: 1) In many cases, large amounts of data need fast analysis for extraction of quick facts or answering a limited number of queries. Therefore the cost should not be paid all up front. Rather, data can be indexed in a pay-as-you-go fashion, favoring more important parts of the graph, and more frequent queries. 2) Static partitioning is not necessarily a good fit for all kinds of queries. As explained above, there can be query workloads which access many connections across different workers no matter how partitioning is done.

We introduce PHD-Store, an in-memory SPARQL engine for distributed RDF

repositories. PHD-Store is an adaptive system that does not rely on static pre-partitioning. Instead, starting from any initial partitioning (which may well be random), it starts answering the first query immediately after loading the data to memory. The first few queries do not use any indexing, and are answered by simple distributed joins. Over time, queries are monitored and analyzed, and are evaluated according to some defined *importance* measure. The system then gradually adjusts the data placement dynamically through replication in order to reduce the cost of important queries. Our work relies on the realization that the queries expose the actual inherent relationships in the data, building up a virtual schema for the data. Our system tries to capture these relationships by performing proper replication of the data. We introduce a technique for redistributing data associated with execution of a query called propagating hash distribution (PHD), which captures the main relationships of a query, and tries to build *data clusters* around its main entities. The main idea is that a very well connected (high degree) entity will have its neighboring less degree entities copied on the same compute node, and those will in turn have their low degree neighbors on the same node and so on in a tree-shaped distribution. The system also tries to minimize the resulting replication by smartly analyzing the query graph using collected statistics. We summarize the key contributions of this work as follows:

- We introduce an efficient distributed system for analyzing RDF data that eliminates pre-processing time entirely, and can start querying data immediately after loading.
- We build an adaptive data redistribution technique (PHD) that is dependent on the query load and adjusts data placement to favor frequent queries.
- We describe a statistics-guided algorithm for analyzing queries to minimize the

replication that results from the PHD redistribution of data.

- We include simple policies for limiting replication by specifying a custom memory budget, and explain how the system can sustain its indexing for important queries.

- We evaluate our system against a very promising system [4] that uses smart replication to minimize communication, and show the effectiveness of our approach in eliminating pre-processing time and achieving better or comparable query execution. In our experiments, we have the advantage of running queries without pre-processing, achieving up to several hours of a head-start. We run queries in less or comparable time than the competitor in most cases, and we use orders of magnitude less replication.

The rest of the thesis is organized as follows. Chapter 2 introduces needed background and preliminaries, Chapter 3 discusses the related work. Chapter 4 presents the architecture of our system. Chapter 5 details the redistribution algorithm (PHD) and the heuristics we use to optimize replication, and Chapter 6 discusses the process of query execution. In Chapter 7, we discuss our experimental results and comparison with competitor systems. Finally, Chapter 8 concludes this work.

Chapter 2

BACKGROUND AND PRELIMINARIES

Before we discuss the main contribution of this work, this chapter details some necessary background. As discussed in the introduction, we work on RDF as the standard format for unstructured data shared on the semantic web, and queried by the standard SPARQL language.

2.1 RDF

The Resource Description Framework is a W3C initiative towards a standard for encoding and exchanging knowledge between various data sources in the World Wide Web. It does not enforce schemas or regulate semantics on different data repositories, providing a simple way of expressing facts across the web. Further, it acts as a standardized way of publishing data vocabularies which are both human readable and machine processable. Its design emphasizes simplicity of expression, which drew the attention of many communities from different areas such as systems security, physical and life sciences, sensor networks and many others.

RDF data are compilations of triples, each consisting of three columns in the format <*subject*> <*predicate*> <*object*>. The subject column of a triple t

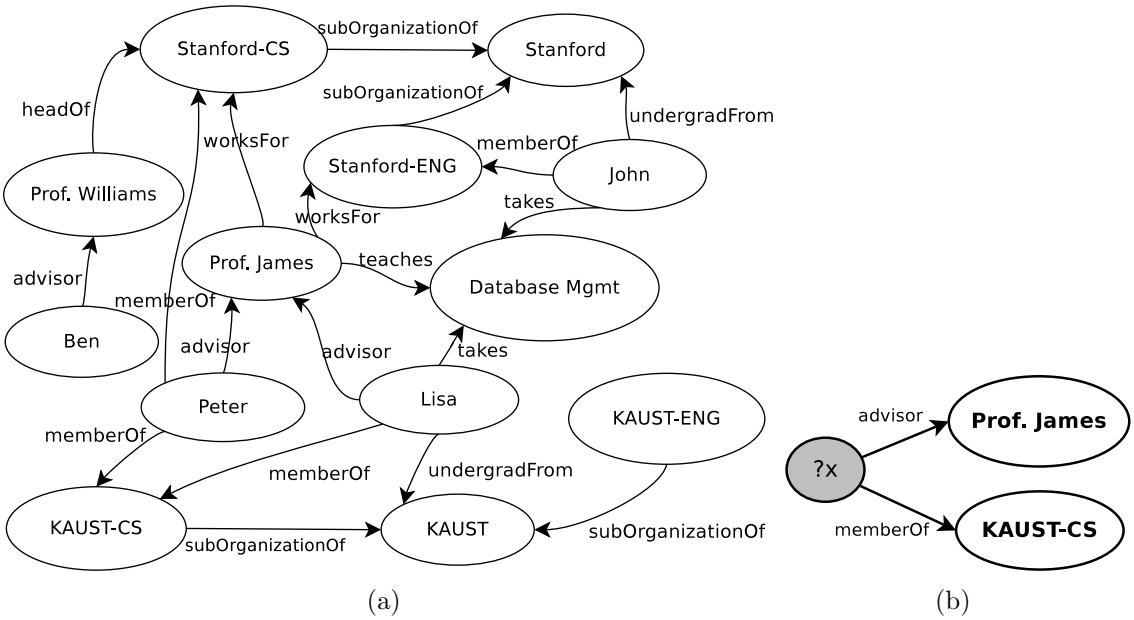

Figure 2.1: (a) An RDF graph. Each directed edge is a relationship between two entities. (b) A SPARQL query: "Find all students at KAUST-CS whose advisor is Professor James"

is denoted as $t.s$, and the predicate and object are $t.p$ and $t.o$ respectively. A triple expresses a relationship between two resources, or an attribute and a value for an entity (the subject and object, respectively). RDF data can be viewed as a directed labeled graph, where vertices represent data resources, and edges–labeled by predicates–represent connections to other resources or attribute values. Figure 2.1(a) shows a very simple RDF graph representing students in an academic network. The connection from "John" to "Stanford" is expressed as the following RDF triple: <John> <undergraduateDegreeFrom> <Stanford>.

2.2 SPARQL

The Simple Protocol and RDF Query Language (SPARQL) is a standard query language for RDF. A SPARQL query looks like a set of RDF triples in that its triples consist of three columns, a subject, a predicate, and an object, but with the exception

of having one or more variables in the place of some of its columns. For example, the following query finds all students studying in KAUST-CS whose advisor is Professor James. It expresses this through two RDF statements as shown, we call each of which a sub-query. The first statement returns all triples where the predicate is "memberOf" and the object is "KAUST-CS". The `?x` variable means there are no constraints on the subject, i.e. it can be substituted for any literal in the data. Likewise, the second RDF triple returns all triples which represent entities linked to *Professor James* via an *advisor* relationship. The final query result is obtained by joining the results of all sub-queries on the matching column names, `?x` in this case:

```
SELECT ?x WHERE {
    ?x  <memberOf>  <KAUST-CS> .
    ?x  <advisor>   <Professor James> .
}
```

It is worth mentioning that SPARQL queries are similarly modeled as small RDF graphs, such as the one in Figure 2.1(b), rendering query execution as an instance of the subgraph matching problem. The graph in Figure 2.1(b) is answered with matching patterns in the main RDF graph. These patterns have the same labels on nodes and edges as in the query, but with replacing variables with labels from the data. Answers to the query in Figure 2.1(b) are shaded in the main RDF graph in Figure 2.1(a).

SPARQL queries which comprise of a single sub-query can be answered directly by scanning all RDF triples or using an index, and returning all such triples that qualify the query, substituting any value for variable columns. SPARQL queries consisting of more than one sub-query are solved by first answering each sub-query individually, then performing joins between all of the them on shared columns. Like joins in

relational tables, executing joins in SPARQL queries consists of first determining the best join order of the comprising sub-queries, then performing the actual join by choosing the best algorithm according to statistics and data distribution. The first step is a very rich area in the database research, and has received tremendous analysis and contribution by the relational database community, including how to choose the best join plan using estimated cost models and how to trim the search space to make plan generation efficient [5]. We therefore do not discuss it more in this work. We also use a simple semi-join algorithm that minimizes communication by sending only the join column(s) and receiving a list of triples which qualify in the join, and then performing the actual join. These steps have been discussed for the case of RDF data in several works, we mention RDF-3X [6] as an example which shows great performance in answering SPARQL queries over locally indexed RDF data, and has gained significant research attention the past few years.

2.3 Data distribution and placement

When working with larger RDF data sets distributed over a big cluster of machines, communication tends to become the bottleneck for scalability and performance, assuming the existence of proper and efficient local indices. Therefore, the partitioning of the data plays a vital role in efficiency of query execution, since it determines the amount of communication needed to perform the joins of a query. A poor distribution of data could result in large amounts of data traveling between compute nodes to execute the joins, whereas good data localization can reduce communication significantly. In fact, joins can in many cases be done locally, completely in parallel without the need for communication, if it is known in advance that joining with data on other workers will not produce any results, and that the results of a join are completely

contained within the data on the local machine. This last scenario is the goal of the more recent work on proper storage and indexing of RDF in the distributed setting, and is also the goal of this work.

In the relational model, data distribution has been studied in two ways: *replication* and *partitioning*. Data replication is one technique for increasing data locality. This is achieved by replicating parts of the data on more than one machine, in an attempt to reduce the amount of data that need to be retrieved from other machines. Partitioning can be done either vertically or horizontally. In vertical partitioning, the columns of a relation are split out to form several relations which can be joined later on to restore the original triples. While this approach shows promising results for indexing on a single machine, it incurs high costs when applied to a distributed environment, since reconstructing the triples for each sub-query will require expensive communication. Horizontal partitioning, on the other hand, splits the relation in rows according to some partitioning policy, such as round-robin, range or hash policies.

Hashing is one popular form of horizontal partitioning for RDF data, where data can guarantee perfect locality with respect to joins by performing hash distribution on a particular join column. When having a small number of columns as in the case of RDF, hash distribution can be more useful. For example, if data is hash distributed on the subject, then the join of the query in Figure 2.1(b) will be executed locally on all machines without the need to send any data over the network. This is because distributing triples on the subject column guarantees that triples with the same value of `?x` will go to the same machines. Therefore, communication is not needed since joining with the data on other machines will give no results. Nevertheless, it is obvious that hash partitioning is useful only in limited cases, where the join column is known in advance (specifically during the data distribution phase), and where the queries are star-shaped around the same column, which are both not realistic assumptions

to make. Queries linking objects to subject of more than two consecutive sub-queries (chain-shaped queries) are usually very interesting in many data sets, and no hash distribution of the data can satisfy such queries, we will discuss an example of such queries later in Figure 5.1. Further, queries can include joins on a column different from the one on which data is distributed. Therefore, a smarter data distribution scheme is required that can handle different kinds of queries.

In some cases, partitioning can provide good data locality–and therefore a better performance–without increasing the size of the graph when done properly. However, there can be no single partitioning scheme which satisfies most queries. Adaptive systems will try to modify the way data are partitioned or replicated to better suit more recent queries. In this work, we use replication to provide for adaptivity in the face of changing query workloads, and avoid partitioning since it incurs higher cost to use and maintain on large graphs.

Chapter 3

RELATED WORK

3.1 Single processor

Earlier works started by using Relational Database Management Systems with RDF data, such as Sesame [7] and RDFSuite [8]. In these works, the entire RDF data set is stored as one very large relational table, usually with indices on all three columns. Jena [9] added support for rich features, such as inference and reasoning, and proposed data structures called property tables to optimize unconstrained joins. In [10], Abadi et al. use vertical partitioning to split the large table into many small tables, where each table contains all the RDF Triples with the same property (predicate). Each table is sorted over the subject column, and is managed by a column-based DBMS. However, the need quickly rose for an RDF data management system that can store and index data natively, while avoiding the unnecessary overhead of extra functionalities such as storage optimizers, transaction engines, and performance monitoring and tuning. YARS2 [11] and HPRD [12] are among the first attempts to build a native RDF store. Both systems, however, lack important optimization features such as planning for optimal join order, which is a very important optimization for complicated RDF queries.

RDF-3X [6] and Hexastore [13] proposed more promising designs. They make use of the fact that RDF has a fixed number of columns (three) to perform comprehensive indexing. They extend the work of Abadi et al. by building comprehensive indices that cover all six permutations of single columns and pairs of columns to make disk access more efficient. RDF-3X in specific managed to achieve an excellent performance and is considered the state of the art in single processor RDF stores. In addition to the six-way indexing, they also build aggregated indices for count queries. The system uses rigorous byte-level compression techniques to keep the indices from exploding in size, and actually manages to have the size of the resulting compressed database less than its original size.

Having indices on all permutations of the triples allows RDF-3X to answer all sub-queries of a SPARQL query very efficiently with one index scan. Because indices are used to answer sub-queries, the query planner can make use of that to perform very fast merge joins in most cases. RDF3X's query optimizer makes use of a cost model that is specially designed for RDF triples to do query optimizations. They propose two kinds of statistics, a generic type, and a frequent-paths based type, which tries to give better predictions for certain join paths whenever they exist in queries. Using the query optimizer along with the smart cost model particularly results in excelling performance, without making many assumptions about the data or queries. RDF3X is generally argued to be state of the art in single processor RDF indexing, despite the expensive loading and indexing step.

3.2 Early distributed systems

Less research work has been made toward implementing distributed RDF data stores. In addition, all existing distributed solutions claim that data partitioning is needed

prior to query execution in order to optimize communication, and they have proposed different ways of performing the static partitioning. Hash partitioning has been used to provide better data locality for distributed joins in [14], [11], and [15]. Although most of those distributed systems include sophisticated modules for cost modeling, plan generation, and query execution, the communication cost is still a dominating factor when executing queries in a distributed setting. Therefore, the way data are partitioned in the beginning of the system runtime plays a crucial role in query processing, and can easily become a bottleneck for large data sets, which can be overcome with a smart data partitioning technique. Nevertheless, standard hash partitioning is not sufficient for different kinds of queries as explained earlier because of the limitations it forces on the join columns and the simplicity of queries.

Still it is worth noting that performing distributed hash joins on data that are partitioned according to the join column is the most efficient approach to do distributed join. This is for a number reasons: first, workers do not need to exchange any data in the process. For the data on one machine, joining with data on any other machine will not give any result, since the join keys have to be different. All workers, therefore, spend all the time in processing joins locally in parallel. Second, this utilizes all compute power of the cluster, instead of localizing all data needed for the join on one of a few machines. Finally, hash partitioning results in good load balancing on average, given a set of values for the join key which are not skewed. This work uses an extended form of hash partitioning that supports more complicated query patterns.

3.3 N-Hop guarantees

Huang et al. [4] try to reduce communication required for queries by replicating parts of the RDF graph which are needed by multiple workers. They first try to find

a good partition of the data, using the popular graph partitioning algorithm METIS [16]. This partitioning results in a minimal number of edges crossing between workers. Since RDF data is edge-based rather than vertex-based, a crossing edge means that a certain entity does not have all the edges where it is subject or object on one machine, and might have them distributed on several machines, such that they need shipping in case of a join. After partitioning the graph, they apply an N-hop replication, which for any vertex v will copy all triples relating to vertices which are N hops (edges) away from v to its machine. While this is an expensive step, it guarantees that any query whose diameter (maximum path length) is N hops or less can be answered locally in parallel on all machines without communication. Queries which are longer are carefully split into multiple smaller queries which are all N hops long or less, then each is solved in parallel. The final results are joined together using normal distributed join techniques in a set of Hadoop [17] jobs. It is obvious that this will result in redundant triples in the final result, and they solve this problem by injecting "ownership" triples, which are essentially flags that indicate whether a certain vertex is owned by the machine it exists on. This adds an extra join to each query, since final results of any query need to be filtered so that only the "owned" copies of the data are returned. The authors also propose an optimization to reduce the amount of replication the system makes, which simply ignores the high-degree vertices from the replication rule, since they result in most replication. Although this optimization reduces replication significantly, it makes the system unable to solve many queries that include high-degree vertices, which are likely to appear in prospective workloads.

We find several drawbacks to this approach. First, it does not scale well with more complicated queries that are relatively long in the number of maximum hops, since the system will incur severe replication of data, which is not practical for larger data sets. Even a query which is only a few hops long will need significant replication,

reaching up to several times the size of the original data [4]. Second, the approach depends heavily on the N parameter (number of hops guaranteed), which is sensitive to the types of queries, while in fact indexing the data takes place in offline time, prior to running the first query. This requires the administrator to either limit incoming queries to N hops only, or have a-priori knowledge about the nature of queries. Finally, this system is not suitable for well connected graphs, where the average degree of most vertices is relatively high, since this will result in heavy replication of the graph data. Overall, this approach achieves good performance when the data and queries fit specific assumptions, and when enough time is given for pre-processing.

3.4 Schism

It became quickly noticed that relationships in data cannot be captured in a straightforward manner by splitting the data across worker machines, even if the partitioning follows some sophisticated plan. Partitioning and replication schemes needed to adapt to the way a graph is queried in order to reach a good placement of the data. In [18], Curino et al. propose a workload-driven solution for the placement problem in relational data, by running a batch of queries in a traditional way, then constructing a graph that captures all the relationships that the queries imply. They then execute a minimal cut algorithm to reduce cross-partition edges. Their approach is very close to the static partitioning approach, since it performs an expensive partitioning operation before running the significant portion of the queries, but it represents a step towards query-sensitive data placement.

3.5 Sedge

In a more recent paper, Yang et al. [19] propose keeping multiple instances of Pregel [20], the Google vertex-centric framework for graph processing, which have different partitions of the same graph, then directing queries to the Pregel instance which minimizes cross-partition edges for this specific query. Sedge appears to be highly adaptive when changing query loads, using online statistics to constantly create and remove partitions of the data to better suit more recent queries. While this approach has a good chance of answering arbitrary queries with minimal communication cost, it comes at the expense of severe replication of the data, and maintaining several independent instances of the Pregel framework running.

3.6 Eventual indexing

An interesting new trend has risen recently in relational databases for building systems that tries to avoid the time necessary for data indexing, before the first query can be executed. In [21], Idreos et al. introduce the concept of eliminating the data-to-query time as they call it, and use the incoming queries to slowly and incrementally build an index over the data by reordering data tuples based on queried parameters. Each query incurs some extra cost for data rearrangement, and after enough queries have been answered, the data are sorted, and the system can use more efficient query execution techniques.

A more recent work by the same authors [22] discussed extending this approach to executing queries on raw files, and similarly building some kind of an indexing scheme in the process. This index contains simple positional data that makes future queries faster by doing guided file accesses to skip tuples and/or attributes. Nevertheless, such

approaches have never been applied to RDF or graph data, despite the fact that pre-processing steps for graph data can be extremely expensive. The problem of indexing large amounts of data before running the first query shows clearly as datasets become larger, and is even more exacerbated when dealing with graph-like data, which cannot be indexed traditionally, and require running complicated algorithms for partitioning.

3.7 PHD-Store

Our work tries to avoid the drawbacks mentioned in previous work by performing adaptive query-based partitioning in the run time. We take on the principle of eventual indexing to eliminate the data-to-query time, therefore the system can start answering the first query immediately after loading the data from disk. Afterwards, the system constantly adapts to the query workload by adjusting its data placement to better serve future queries, using replication to improve the data locality with respect to the workload. We allow specifying a replication budget that the system abides by while generating replicas from data, such that it can operate with limited memory. We try to provide looser guarantees than the ones introduced by the N-Hop approach, where data placement serves indexed queries, or similar ones, but does not need to replicate for any type of queries. This gives us both the advantage of adaptivity, even for larger and more complicated queries, and at the same time avoids unnecessary replication, which can be very expensive when dealing with very big data sets. In the following sections we explain our model and implementation.

Chapter 4

SYSTEM ARCHITECTURE

The system architecture is depicted in Figure 4.1. The master node contains a statistics module that gathers some simple information about the data, which will help guide the redistribution process. A query index is responsible for analyzing new queries to decide whether they can, completely or in part, be solved in parallel without communication. It also keeps an index of recent queries in order to keep the redistribution in favor of hot queries, but also within the allowed replication budget. A query planner uses the information provided by the statistics manager and the query index to build a plan for execution. The plan is then passed to the query executor, and the result is returned to the user.

Every worker node contains two main data structures: the main data index, and the replica index. The main index contains the original data assigned to this worker without any kind of redistribution. This index is used every time a new query is answered. The replica index contains copies of data used in the replication process to make future queries faster. After replication for a certain query, we call such a query *redistributed*. This separation into two indexing data structures is an important design decision for a better replication management in PHD-Store. As will be discussed in section 6, an independent replica index effectively helps keep the cost of join and

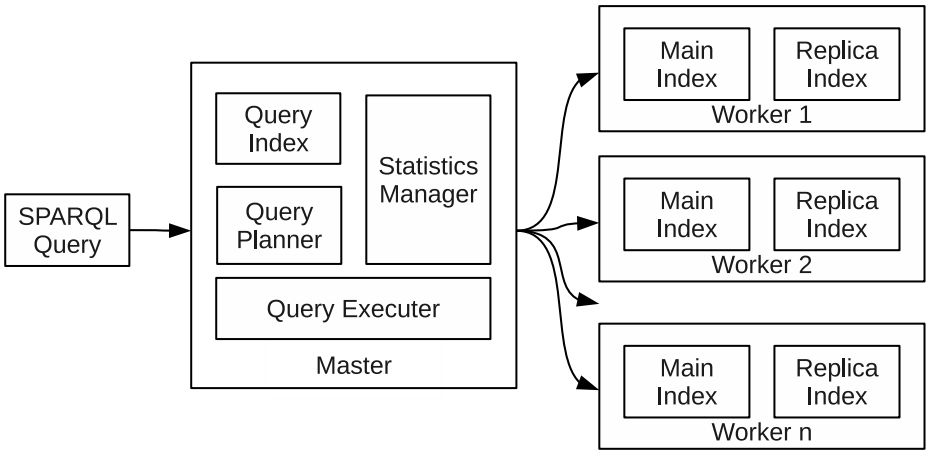

Figure 4.1: System Architecture

index maintenance low.

PHD-Store uses a simple workflow consisting of three phases to control the placement of data across the worker nodes. First, when a new query is posed to the system, a statistics manager is consulted which contains metadata about the current data placement as well as statistics about different worker nodes. A plan is formulated which dictates the order of execution of joins, and whether the query should be executed using a distributed join method with communication, or in an embarrassingly parallel fashion. The plan is then handled by a query executor to get the final results returned to the master node.

The second step is identifying communication patterns and important relationships between different parts of the data. PHD-Store does this by analyzing and indexing incoming queries in order to identify new *hot queries*. It calculates a certain *core* vertex in the query graph, and then performing a graph walk based on the importance of the entities in the graph.

The third step is the redistribution for hot queries. PHD-Store uses our proposed Propagating Hash Distribution (PHD) algorithm to distribute the data of a query in a special way. The goal of this algorithm is minimizing communication needed for

answering frequent queries in the future. The system follows certain heuristics based on smart analysis of the query graph to reduce the resulting replication of data. A simple policy exists to define a memory budget for the replication so that the system can run in memory-limited settings. The next two chapters discuss in detail how the PHD algorithm works and the query execution process.

Chapter 5

PROPAGATING HASH DISTRIBUTION

5.1 Extending hash distribution

In this work, we claim that indexing graph data should not follow the characteristics of the dataset itself. It should rather find a good way of detecting important relationships between entities in the data. Thus, we choose to build a data distribution model that is based on the analysis of the query workload itself. As the system answers more queries, it approaches a state which is optimal for all recent queries, and which is likely to answer a new query efficiently. In order to achieve this, we introduce our Propagating Hash Distribution (PHD) algorithm. The PHD is the main contribution of this work. The goal of the algorithm is to provide a better way of clustering that is guided by queries rather than data, and leading to better query performance for future similar queries. Consider the schema of the popular LUBM benchmark [23], where a *university* has many departments, and is connected to many *students*, each department is in turn connected to many, albeit less, *students*.

Definition 1. *RDF Path. An RDF path is a set of ordered triples such that for any two consecutive triples* t_i *and* t_{i+1}, $t_i.o = t_{i+1}.s$. *A query consisting of triples which can be ordered to make such a path is called a Path Query.*

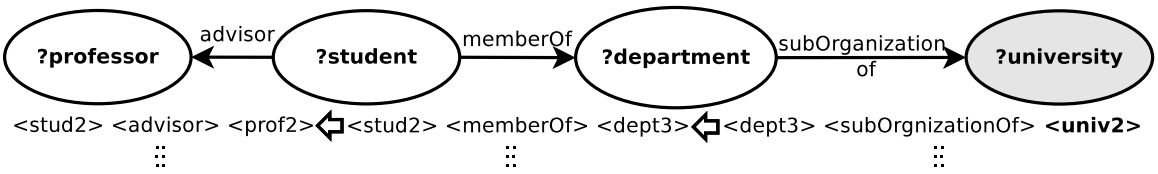

Figure 5.1: Simple Path Query

We analyze a simple path query such as the one in Figure 5.1. This query extracts the data of each *university* along with its *departments*, and each with its *students* who have *professors* as their advisors. The query is expressed in SPARQL in Figure 5.2.

```
SELECT ?univ, ?prof WHERE {
    ?dept       <subOgranizationOf>     ?univ .
    ?student    <memberOf>              ?dept .
    ?student    <advisor>               ?prof .
}
```

Figure 5.2: Path query in Figure 5.1 in SPARQL

Answering such a query requires performing two joins between its three comprising sub-queries. RDF data can be thought of as one very large relational table, over which semi-joins are executed. Therefore, the first join between (departments-universities) and (students-departments) will result in a list of universities along with all their departments and students. This list needs then to be further joined against (students-advisors) relationship to result in the required (universities, departments, students, advisors) tuples. Recall that the large relational table is likely to be distributed randomly across several machines. As the data set grows large, and when performing joins with high cardinalities, distributed join algorithms tend to suffer from a high communication cost, effectively a performance bottleneck for the overall execution of the query.

Let us first discuss how a simple star-shaped query can be answered efficiently

using hash-distribution. Such star-shaped queries lend themselves easily to hash-distribution, if edges are distributed according to the hash value of one specific column, making it easier to perform joins. Recall the simple query in Figure 2.1(b), where two RDF triples are joined over the common subject column `?x`. Assume that RDF data have been distributed according to the subject column. As a result, subject values will be split in groups across workers such that no two triples with the same subject will exist on two different workers. In order to execute the join in this case, it is enough that each machine carries out a local join between its triples. We argue that this is the most efficient way to do distributed join, since it parallelizes execution evenly (given only a good hash function) and at the same time does not incur any communication cost.

Nevertheless, it quickly shows that distributing the data on the hash of a certain column does not solve most of the cases of RDF queries. A simple path query which exceeds two edges (such as the one discussed above) will not benefit from simple hash distribution. Even two-edge queries such as the one in Figure 2.1(b) will not benefit if the current hash-distribution does not rely on their join-column, and will not execute without communication. Consecutively, such a distribution will only be effective if most of the queries are run on a specific column, which needs to be known a-priori. These constraints would greatly limit the performance and flexibility of a general purpose RDF store.

Our proposed PHD algorithm is inspired by traditional hash distribution. It promotes the *importance* of certain entities in the data, around which parts of the graph can be grouped to eliminate or minimize communication. The algorithm mainly starts by placing such important entities in the data according to their hash values. The next steps carry out a *propagation* of the placement to further related entities. Formally, PHD-Store executes the following steps for a given query Q:

1. Transform query graph Q into a vertex-weighted, undirected connected graph Q'.

2. Choose the most important vertex v' from which we start walking the query graph.

3. Hash-distribute all edges directly incident to v' according to the hash of v'.

4. Decompose Q' into a set of paths $\{P\}$ that start with v'.

5. Propagate the data distribution along each paths in $\{P\}$. This is described in more detail in the next section.

The first four steps are aimed at improving replication in the system, and they are discussed later in this section. We now analyze the propagation technique proposed in PHD-Store.

5.2 Data propagation

For a given query Q, the system identifies the most important entity, which we call the *core vertex*, denoted as v'. We discuss the criteria of choosing v' later in this section. The query graph is then transformed into an undirected graph Q', and its vertices are assigned importance scores. Afterwards, the graph Q' is decomposed to independent paths, and after query decomposition, the system performs data propagation on single paths. Given a path that starts with the core vertex, the system will distribute the data using the PHD algorithm along this path. Consider again the path query in Figure 5.2 to be one of the paths in our example, assuming that `?univ` is selected as the core vertex. Before starting data propagation, the system first distributes all triples qualifying the sub-query `<?dept> <subOrganizationOf> <?univ>` (being the

only edge adjacent to the core vertex) by calculating the hash value of the column `?univ` modulo the number of workers, and sending the triple to the one with the resulting id.

The next step is then to propagate the data of the remaining edges along the given path. Consider the two triples $t_1 = <s, p, o>$ and $t_2 = <o, p', o'>$ placed consecutively in a path such that the object of t_1 is the subject of t_2. In order to explain the propagation, we first introduce the following definitions:

Definition 2. *Binding Column. The binding column of a triple $<s, p, o>$ is the column used to determine its placement, and is either its subject or object column.*

Definition 3. *Propagating Column. The object column of t_1 is called the propagating column, and it carries over the placement of t_1 to t_2, rendering the latter's subject column, `?o`, a binding column.*

The process is formally described in Algorithm 1. The algorithm runs in parallel on all the worker nodes. The first 5 lines of the algorithm perform hash distribution of the first sub-query, which is always adjacent to the core vertex, and is called *propagation level 0*. As illustrated in Figure 5.2, the vertex `?univ` was chosen to be the core. Let us assume a triple `<KAUST-CS> <subOrganizationOf> <KAUST>` will be placed according to the hash value of its object column, KAUST, to worker w. This column (core vertex) will become the binding column for the triple, and its other end, KAUST-CS, will be the propagation column. For each triple that satisfies the next sub-query in the path (propagation level 1), KAUST-CS is used as a binding column. For example, the triple `<Peter> <memberOf> <KAUST-CS>` will also be placed on worker w, following the placement of the first triple. Its column Peter will become propagating column for next level, and so forth.

Lines 6-15 describe the above steps for propagation formally. Propagation is done

Algorithm 1: Performing PHD on a given path P

Input: $P = \{E\}$; a path of consecutive edges, v'; index of the core vertex
Result: Data placed and propagated along path P

```
// hash-distributing the first (core-adjacent) edge
set prevData = getTriplesOfSubQuery(e_0);
foreach t in prevData do
    set m = hashColumn(e, v');
    sendToWorker(e, m);
set pc = getPropagationColumn(e_0, v');
// then propagate other edges
foreach i : 1 → |E| do
    set propColData = extractColumn(prevData, pc);
    set allColsData = allToAll(propColData);
    set currentEdge = getTriplesOfSubQuery(e_i);
    foreach c_j in allColsData do
        set data = join(c_j, currentEdge);
        sendToWorker(data, j);
    set allData = recvFromAll();
    addDataToIndex(allData);
    set prevData = getTriplesOfSubQuery(e_i);
```

through a series of semi-joins between each level in the path and the level directly before. This procedure will cause triples on level i to follow the placement of the triples in the parent level $i-1$. When answering a sub-query q_i, the set of result triples is denoted as RS_i. This result set is filtered, and only triples that satisfy the semi-join with the previous level are kept. We denote such triples by the term *propagation tripleSet* PTS_i. Formally, for worker w:

$$PTS_0 = \{\tau \in RS_0 : hash(\tau) = w\}$$

$$PTS_i = RS_i \ltimes PTS_{i-1}$$

Assume that the two universities in Figure 2.1(a) are hashed to two different workers such that `Stanford` goes to w_1 and `KAUST` goes to w_2. Therefore, all triples satisfying the first sub-query `<?dept> <subOrganizationOf> <?univ>` whose university is `Stanford` will be placed to worker w_1, and w_2 for triples with `KAUST` university. Table

Worker1
<Stanford-CS> <subOrganizationOf> <Stanford>
<Stanford-ENG> <subOrganizationOf> <Stanford>
<Peter> <memberOf> <Stanford-CS>
<John> <memberOf> <Stanford-ENG>
<Peter> <advisor> <Prof.James>
Worker2
<KAUST-CS> <subOrganizationOf> <KAUST>
<KAUST-ENG> <subOrganizationOf> <KAUST>
<Peter> <memberOf> <KAUST-CS>
<Ben> <memberOf> <KAUST-CS>
<Lisa> <memberOf> <KAUST-CS>
<Lisa> <advisor> <Prof.James>
<Ben> <advisor> <Prof.Williams>

Table 5.1: Distribution of RDF data in Figure 2.1(a) according to the path query in Figure 5.1

5.1 shows the placement of the triples related to our running path query example.

5.3 Minimizing replication

Depending on the nature and schema of the data, the propagation algorithm might, and in most cases will, result in replication. Consider again the data in Figure 2.1(a) when answering the path query in Figure 5.1. In this example, `Peter` is a member in two departments at the same time: `Stanford-CS` and `KAUST-CS`. As shown in Table 5.1, the triple `<Peter> <advisor> <Prof.James>` needs to exist on both workers at the same time, following its propagation column with the value `Peter`, which exists on both workers, since this student is member of two different departments. It follows that any consecutive edges in the path query will also be replicated for `Prof.James`,

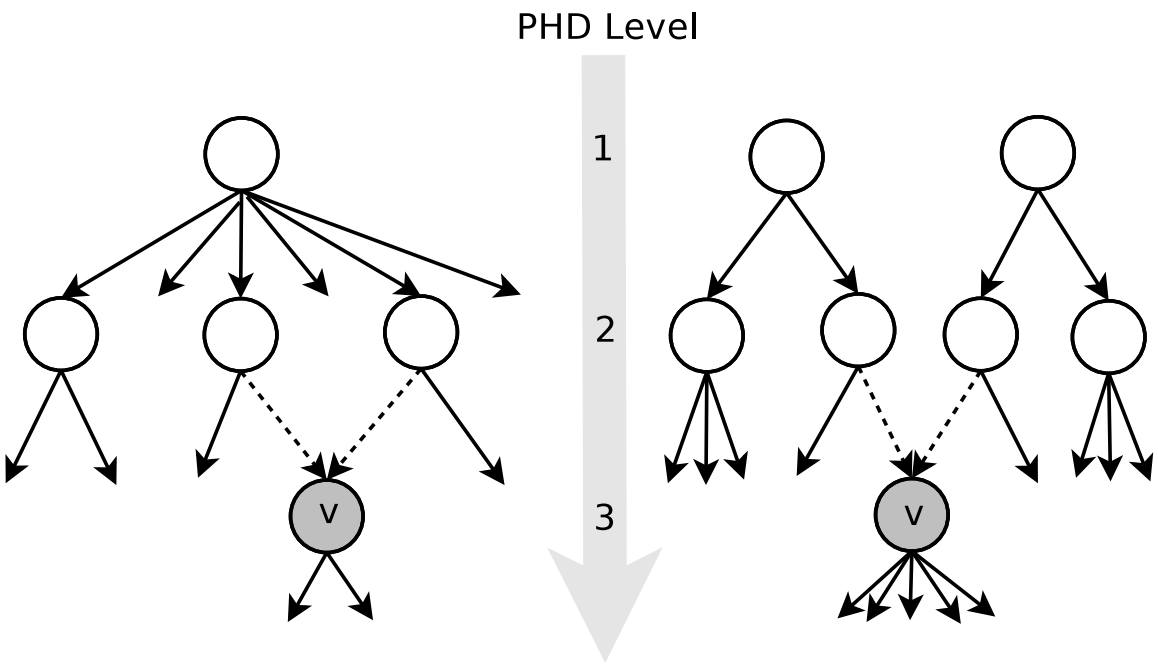

Figure 5.3: Replication patterns in a graph - Heavy edges are replicated

and so forth.

Figure 5.3 illustrates the scenario leading to replication in the PHD algorithm. The algorithm works in iterations of propagation which resemble a tree, where in each iteration, the children of all vertices in the last iteration are obtained and redistributed according to the placement of their parents. If two or more vertices in iteration $i-1$ have the same child vertex in iteration i (when the graph is no longer a tree), the entire subtree rooted by this vertex in all iterations greater than i will be replicated on all the machines of the parent vertices. The dashed lines in Figure 5.3 depict this collapsing scenario. All connections emanating from the shaded vertex in the figure will be replicated on the nodes where its parents were placed.

Propagation is trivial in path queries. The system simply starts from the core vertex and carries over the placement information towards the end of the path. Even in the cases when the core vertex is chosen in the middle of the path, propagation can

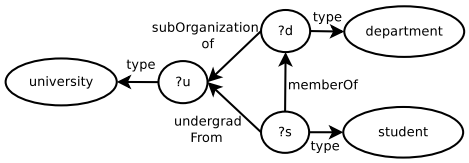

Figure 5.4: Query 2 in LUBM benchmark

advance from the core in both directions towards both ends of the path. Nevertheless, it is not straightforward how to handle propagation in more complicated queries, such as the query in Figure 5.4, which is the second query in the popular LUBM SPARQL benchmark. Let us assume the core vertex has been chosen to be `?u`. After hash-distributing the three adjacent edges to the core, the placement needs to propagate to the remaining three edges. Consider the edge between `?s` and `?d` which is labeled *memberOf*. It is not clear how the propagation should take place on this edge. In other words, triples of this edge can be placed according to either the propagation column `?s` from the triple `<?s> <undergraduateDegreeFrom> <?u>`, or that from the triple `<?d> <subOrganizationOf> <?u>`. Both propagations will give correct results when performing the join, however, the resulting replication may vary drastically between both cases as will be explained shortly. Although the efficiency of query execution is independent of the replication ratio–because of the way replication is managed in PHD-Store as explained in Section 6–the data storage is now a significant challenge. In order to minimize the resulting replication, we propose a method for *walking* the query graph in order to reach a good propagation direction. After obtaining propagation directions, the query graph is decomposed into independent paths, each of which starts from the core vertex. The data are propagated along each one of these independent paths in the same manner discussed above in Algorithm 1. This results in part of the data being replicated to answer this specific query completely in parallel with zero communication as illustrated in Figure 5.3. We call this process the *redistribution* of a query. A *redistributed* query has its data placed on the workers in such a way that

it can be executed again without any communication.

The redistribution method is based on the following observation:

Observation 1. *Collapsing of vertices happens less frequently when propagating from high-degree vertices to lower-degree vertices than the opposite direction.*

The reason for this is that fixing high-degree vertices as cores depicts the natural structure of a graph clustering, where lower-degree vertices are *attached* to high-degree ones, and therefore should follow their placement. An obvious example is students versus universities. If each student vertex (lower degree) is treated as a core, propagation will lead to many students collapsing to one university. If universities (higher degree) are cores, students are grouped in clusters around them, and this can effectively lead to zero collapsing.

It follows from this observation that if high degree vertices were in higher levels of the replication tree, they would result in less replication than if they were in lower levels. This is shown in Figure 5.3, where less dashed lines exist in the lower parts of the tree when a high degree vertex is chosen as core. Therefore, the proposed *graph walk* algorithm tries to push the highest degree vertices to the top of the tree, and then walk the graph from higher to lower degree vertices. The graph walk algorithm is explained formally in Algorithm 2.

The query is handled as an undirected graph, and traversed from the core vertex. We start by assigning a score value for each vertex in the query graph, which is equal to its degree that was calculated by the statistics manager, and then choosing a core vertex for the graph walk. For simplicity, we now choose the highest degree vertex to be the core. There can be other factors in deciding the core vertex, such as the centrality and degree of the vertex in the query graph.

We now discuss the *graph walk* algorithm, which aims at minimizing communi-

Algorithm 2: The *graph walk* algorithm for decomposing the query graph

Input: $Q = \{E\}$; a query consisting of a set of edges, v'; index of the core vertex

Result: Set of paths $\{P\}$

```
1  set adj = getAdjacentEdgesTo(v');
2  set {adj}.parent = φ;
3  addToPendingList(adj);
4  while pendingEdges notEmpty do
5      set e = getHighestDegreeEdge({E});
6      removeEdge(e);
7      appendToPath(e, e.parent);
8      set {adj} = getAdjacentEdgesTo(e);
9      foreach a in {adj} do
10         if a NOT explored then
11             set a.parent ← e;
12             addToPendingList(a);
```

cation by decomposing the query graph into independent paths that traverse higher degree vertices first, and push low degree vertices toward the end of the paths. The algorithm inserts all edges incident to the core in a *pending edges* set. Then gradually keeps exploring new edges by removing the edge with the highest vertex score first from the set, and inserting all its adjacent edges to the pending edges. Note that the direction of traversal of the graph is independent from the actual edge directions of the query. The result is a set of independent paths that start at the core vertex, which is the highest degree vertex in the query. As an example, consider the query in Figure 5.4. The vertices in the query are shaded such that darker vertices have higher scores (i.e. average degrees for simplicity). Being the highest score, the `?u` vertex was chosen as core, and the query is decomposed into a set of paths as in Figure 5.5.

5.4 Notes on replication

We distinguish queries in terms of propagation as follows:

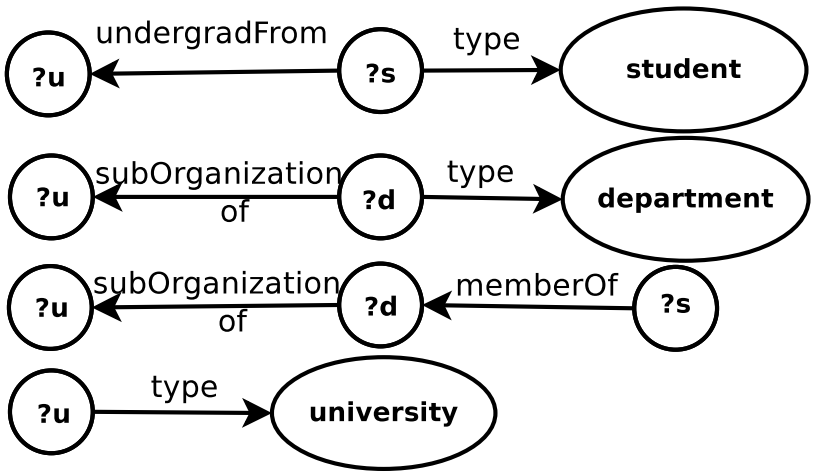

Figure 5.5: Query decomposition into a set of paths

Definition 4. *Generic query. A query where all entities (subjects and objects) are variables is called a generic query.*

Definition 5. *Bound query. A query which has at least one entity binding (a constant subject or object) is called a bound query.*

Generic queries usually result in a significant amount of replication, since their joins have very low selectivities, joining all instances of one entity in the data against all instances of another. In contrast, bound queries usually result in a limited number of result triples in every step of the joins. As a result, performing PHD for a bound query has no overhead, and usually results in negligible replication. It is noted that generic queries are much less common than bound queries, as will be shown in the benchmark. In addition, the graph walk Algorithm 2 provides an effective heuristic that can minimize the replication resulting from generic queries.

A generic query is called the *superset* of a bound query, if the binding exists on the core vertex. For example, the generic query in Figure 5.4 is a superset of the query that finds all departments and students of `Stanford`, since it generalizes the solution for all universities. From this relationship, we note the following lemma:

Lemma 1. *A query q that is a subset of a redistributed generic query Q is also redistributed, and can be answered without communication.*

It is worth mentioning that the PHD way of replicating data is a generalized approach with respect to the replication done by Huang et al. in their N-Hop guarantee paper [4]. In a nutshell, the N-Hop guarantee replicates data such that for any vertex, all data that are N-hops away are replicated on its worker, then they include extra triples for entity ownership, which are joined in the last step to produce only original triples. The PHD replication is an on-demand technique, therefore only entities that are related to each other will be replicated to exist on the same compute node, resulting in significantly less replication. In addition, being query-based, PHD can carry out replication to several hops only to a specific query, without the need to apply the same replication to all entities in the data, avoiding an explosion in data storage, which the N-Hop guarantee technique suffers from. For example, in the experiments done by [4], a 2-hop guarantee takes up to 190% of the original data size, including the ownership triples, given that high degree vertices are not replicated, which results in inefficient execution of queries involving such vertices. Replication goes up to 500% of the original data if high degree vertices are replicated.

As we explained above, executing the PHD is a series of semi-joins that is chosen carefully to minimize the resulting replication. However, this means that this set of joins are not necessarily in the best order possible for a fast execution. As a result, executing the PHD might itself be slower than executing the original query. However, as will be shown in the evaluation section, query execution benefits by at least an order of magnitude after performing the PHD, which in most cases takes less time than the distributed execution of the original query.

Furthermore, we argue that vertices with very high degrees should not act as core vertices in any query. The reason is that when a vertex grows exceptionally

large in degree, it can cause unbalance in the replication, since it will draw too many connections to its cluster. We treat such vertices as outliers, and they are given scores of zero to prevent them from being core vertices in queries. We use the same degree threshold used in [4], which is three times the standard deviation away from the average degree in the data.

Chapter 6

QUERY PROCESSING

This section discusses several aspects related to query processing. In PHD-Store, query processing takes place in two places. First on the master computer, which analyzes and indexes queries, and generates a plan of execution. The master then starts communicating with client machines, which execute the plan and return the results.

6.1 Statistics manager

In addition to statistics needed for classic cost estimation, and as discussed in the previous section, PHD-Store needs some statistics on the data in the query decomposition phase in order to guide the query graph walk. Particularly, the degrees of each of the entities in the data are needed to decide on the directions of the graph walk. It is, however, prohibitively expensive to keep this kind of information for each single vertex in the entire data. RDF data is specifically challenging in this regard, since in most cases there is no schema provided with the data to explain the types of each entity, but rather the type can be *inferred* based on the connections this entity makes in the data. Huang et al. [4] try to use the specific relationship in the data

to categorize vertices and associate them with their respective degrees. They rely on the existence of specific schema-exposing relationship, such as the `type` relationship in the LUBM benchmark data. For example, for type `student`, they calculate the average degree of all such vertices that satisfy the triple `<?x> <type> <student>` and they assign this average to the type `student` itself. Consequently, they need the query to explicity state the type of each entity, as in the case of the query shown in Figure 5.4. While this method is inexpensive, it is limited to the special cases that both the data and queries include such special relationships as `type`, that describe or categorize entities. The method also requires that this relationship is known by name before the run time of the system. In other words, for a query that consists of the triple `<?stud> <advisor> <?prof>`, there is no means for the system to figure out that the vertex `?stud` actually implies the type `student` if it was not stated clearly by a triple. Note that the name is arbitrary since it is a variable.

In order to avoid such limitations, we try to calculate the statistics based only on the edges in the data. For each unique predicate, for example `advisor`, we need to calculate the degrees of all vertices on the right of this predicate, take their average, and mark the predicate's right hand side with this number, and repeat the process for the left of the predicate. When a new query asks for triples with the predicate `advisor`, we can find the degrees for vertices on both sides, without the need to know the type of those vertices. Consider now a query with two edges joining on one column, such as the star-shaped query in Figure 2.1(b), where the predicates are `advisor` and `memberOf`. Note that the system will calculate two, possibly different, numbers for average degrees of the join column `?x`, calculated from the two predicates. This can be the case if the vertices that have the predicate `advisor` are different from those that are related by `memberOf` to other entities. In this case we choose to take the maximum of the calculated degrees for each vertex.

In order to avoid expensive communication in the process of collecting statistics, we choose to keep only approximate statistics. For every predicate, we only calculate the average degrees of its two ends on each machine individually, rather than calculating the global average degree from the entire graph. The average degrees of a certain vertex are then sent to the master node, and the latter computes the average of averages, which is a relatively accurate indication of the actual degree of the vertex. This makes degree calculation significantly less expensive, and avoids heavy communication while providing a reasonable approximation to the actual average degrees.

Using this information, PHD-Store can be flexible about collecting statistics from any kind of data, without the need for a schema or special `type` predicates. At the same time, this maintains enough information to guide the graph walk algorithm from higher to lower degree vertices.

6.2 Data indexing

PHD-Store keeps two indices saved on each of the client nodes: the *data index*, and the *replica index*. The first is the main data index, which keeps the original data assigned to this worker in the beginning, which could be a random distribution of the data, and never changes during the runtime of the system. This index is used to answer queries which are not served by the current distribution of the data. Both the Data Index and the Replica Index contain instances of a local RDF index. A local RDF index is a comprehensive-indexing structure following the same approach of Hexastore [13] and RDF-3X [6], except for variable-predicate indices. Particularly, it contains indexes to return all triples given a predicate, or predicate and subject, or predicate and object. We are currently more interested in constant-predicate queries,

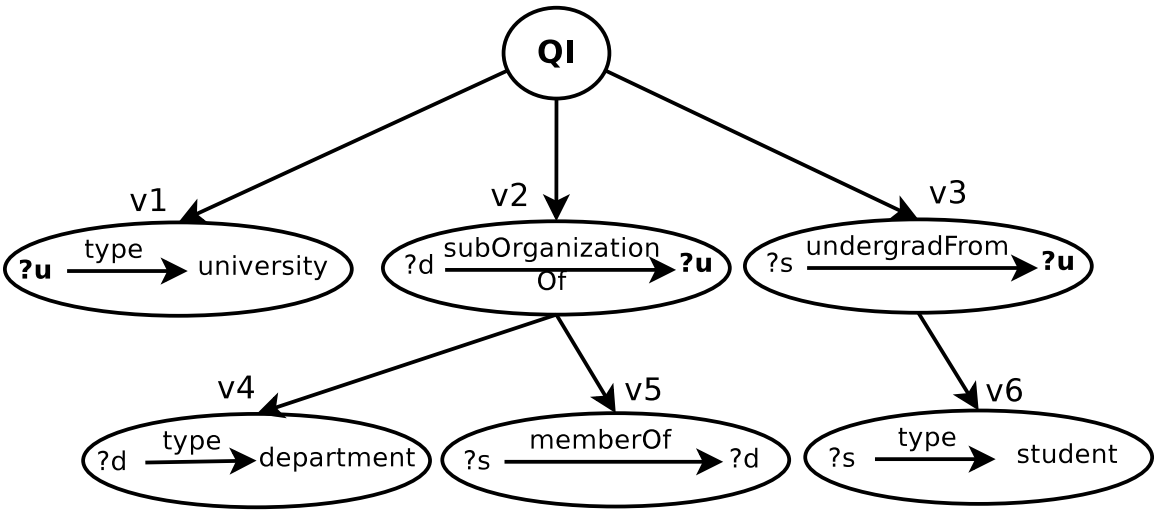

Figure 6.1: A Path Forest out of Query2

therefore we choose not to index variable predicates. The indices, however, can be easily extended to full RDF-3X instances that support variable predicates as well.

Currently, PHD-Store operates in read-only mode, so the data in this index is never changed, and queries are redistributed into the second index, which is called the *Replica Index*. This index is empty at the beginning of the runtime, and gets slowly populated each time the system distributes the data for a new query.

Recall the query in Figure 5.4, which is the second query in the popular LUBM SPARQL benchmark. The query returns all triples representing a university, a department in that university, and a student who is a member of this department, and who has an undergraduate degree from the university. The query was decomposed into a set of independent paths as in Figure 5.5. This set of *distributed* paths is added to the index in the form of a *Path Forest*. We define the Path Forest structure as follows:

Definition 6. *Path forest is a forest that is built out of the trees of all redistributed queries. Each vertex v in the forest represents an edge E_v from a redistributed query. An edge e between two vertices v_1 and v_2 in the forest exists to indicate that there is*

a decomposed path $E_{v_1} \rightarrow E_{v_2}$.

Traversing the Path Forest from the root to any leaf vertex shows a path which has been decomposed from a redistributed query using the graph walk algorithm. Figure 6.1 shows a Path Forest out of the decomposition of Query2. The replica index is essentially a Path Forest that stores all replicated triples resulting from redistributing a query.

Each of the vertices in the replica index Path Forest contains an instance of a local index, which indexes a propagation result for the path starting from the root and ending at this specific vertex. For example, v_2 will contain the result set of the sub-query `<?d> <subOrganizationOf> <?u>` on the local machine, while v_4 will contain the result set of the semi-join of the previous sub-query with the sub-query `<?d> <type> <department>`.

6.3 Query index

PHD-Store uses a special data structure to keep track of recent queries on the master node, called a *query index (QI)*. This data structure is responsible for: 1) analyzing each incoming query to check its *importance* and decide whether it should be redistributed, 2) storing metadata about the query, and 3) checking if an incoming query can be solved without communication by current distribution of data. The QI's structure is a Path Forest that is constructed from all recent queries by decomposing each query into a set of paths, and adding those paths to extend the QI forest. It keeps track of processed queries in the form of paths, in order to allow queries that share paths to make use of the same distribution of data.

PHD-Store allows defining a threshold for the replication ratio that the system should not exceed. While this can result in more communication, it allows systems

with limited physical memory resources to process bigger amounts of data. As a result, PHD-Store does not keep the replication data of all queries in its running history, but chooses to rank queries according to their *importance* so that it can remove the data generated by less interesting queries when it exceeds this threshold. Theoretically, this ranking can be customized according to the requirements of the compute cluster. Several factors can affect this ranking, such as frequency of query execution, freshness (last execution of a query), result size, or execution time (bigger queries have more priority to be indexed than smaller and faster queries).

In our implementation, we used a simple timestamp-based measure to decide on the freshness of the queries and use it as a measure for ranking. This is similar to the Most Recently Used cache policy (MRU), and can be represented as a window of recent queries which keeps expanding as long as the replication budget allows. It will start truncating old queries when the budget exceeds the threshold, so that older queries are not favored any more by the system. This *aging* measure is derived directly from the main requirements of PHD-Store, since it allows the system to be adaptive about the query load. After a number of queries have been processed, the master node makes decisions about which queries to keep in the replica indices, and which to remove to reuse their space. This leverages the capability of changing the index structure according to the recent number of queries, while keeping the required resources fixed.

When a new query is introduced to the system, it is associated with an importance value that is updated everytime this query is posed to the system. If the query is *hot* enough, the process of redistributing it starts by decomposition as explained in the previous section, and its paths are added to the QI path forest. The first level in the QI forest contains all core-adjacent edges from queries that have been redistributed before. Note that a core vertex may be an inner vertex in another query, and therefore

would exist once in the first level of the forest, and once in a deeper level. A new path is added gradually by checking the QI forest level by level, starting with its level 0, which contains all core-adjacent edges. Note that a vertex in the QI forest is actually an edge in a query. The first level is searched for an edge which matches the first edge in the new path, which have the same parent as the first edge's parent, which is the empty set ϕ. If such an edge is found, the same process is repeated for the second level. Otherwise, a new node is created in the forest, and the rest of the path is added automatically as a new branch in the forest.

The main responsibility of the QI is checking whether (part of) a query has been redistributed for and can therefore be solved efficiently without communication. In a nutshell, if the query has been redistributed before, meaning that it exists in the QI path forest. The Path Forest structure makes this a simple task after decomposing the new query into paths. Each path now can be traced along the forest, making sure that all its vertices (which are edges in the query) exist in the QI forest. The nodes of the QI are updated with the new timestamps. If all the paths are indexed, then the QI signals the query planner that the query is fully processable without communication. Otherwise, the query's *importance* is updated accordingly, and is answered using distributed joins if it has not yet become one of the hot queries for redistribution.

The QI forest maintains timestamps for each node in the QI forest to indicate the freshness of the path starting from the root going down until this specific node. When an indexed query is answered by the system, its timestamps need to be updated. This is done starting from the root, but not necessarily down to the leaves if the query's decomposed paths are shorter than the indexed paths. Consequently:

Lemma 2. *A node in the QI forest will always have the same or older timestamp as its parent. It follows that leaf nodes have the oldest timestamps in the whole QI*

forest.

We keep the leaf timestamps in a priority queue so that we can find the oldest vertex quickly. Typically, vertices along a path will have the same timestamp, which is calculated when this path is first inserted to the QI forest. If part of this path is queried again, the timestamps of its vertices (always from the root down to some depth) are updated. This makes deletion of the oldest path very efficient, by consulting the leaves data structure to get the oldest vertex, then traversing the QI forest bottom up from that vertex to find the highest ancestor node with the same old timestamp, and deleting the path starting from that ancestor down to the vertex. Because local indexing happens on the vertices of the QI forest themselves, deleting a path does not need to traverse the entire dataset on the worker node, which can involve scanning a large amount of data, rather simply it deletes the index allocated on the vertices along the deleted path.

Currently, PHD-Store deals with query indexing in the granularity of a whole query. In other words, if only some of the paths are indexed, then it treats the query as an entirely new query. The concept, however, should intuitively extend to an edge granularity such that the nodes of the QI forest store also the importance of their edges, and PHD is carried out on hot paths instead of hot queries. Then a new query which has only some of its paths indexed can be answered more efficiently by performing parallel joins without communication for these indexed paths, and then completing the query using distributed joins.

Chapter 7

EXPERIMENTAL EVALUATION

We run all our experiments on a cluster of 21 Shuttle SH55J2 cube machines. Machine 1 acts the master of the system and the remaining 20 are the workers. Each machine has an Intel(R) Core(TM) i5 660 CPU at 3.33 GHz with 16 GB of RAM and 2 X 2TB (7200 RPM, 6.0Gb/s) hard disks in RAID-0 configuration. The machines run 64-bit 2.6.38-8Linux Kernel (Ubuntu 11.04 Natty distribution) and are connected via a 1Gbps Juniper EX4200 switch.

We have implemented both the partitioner as well as the query engine of [4]. The partitioner was implemented using Hadoop to ensure scalability using the same algorithms defined in [4]. To make the comparison between PHD-Store and [4] fair, we implemented the query engine of [4] in memory and did not use RDF-3X as it runs from disk. We used the same indexing methodology defined by RDF-3X which is also used in PHD-Store. In the experiments, we will refer to our system as PHD, and the two configurations of [4] as 1-Hop and 2-Hop. In the 1-Hop configuration, data that are 1-hop away from any vertex are replicated to its machine, and this is extended to data that are 2-hops away for the second configuration. In some of the experiments, we also compare against a simple distributed semi-join implementation, which shows the time of query without any kind of indexing and using original data

on all the workers. We refer to this as the Semi-Join.

7.1 Data and queries

To evaluate the performance of our system, we used the very well known benchmarks: Lehigh University Benchmark (LUBM), which is capable of generating a synthetic RDF dataset in the academic domain. We generated a dataset consisting of 2000 universities that resulted in almost 50GB of data in N-Triples format and around 267 million triples. The LUBM benchmark has 14 queries out of which we select 11 queries that have at least one join. These queries are Q1, Q2, Q3, Q4, Q5, Q7, Q8, Q9, Q11, Q12 and Q13. Moreover, we added two queries S1 and S2 (see Appendix A), which are meant to test the systems against more complicated query patterns. S1 is a chain query that is several hops long, and cannot be processed without communication even under the 2-hop configuration of [4]. S2 is another special query that cannot be solved without communication regardless of the N-hop guarantee, because of the high degree vertex optimization.

From the above mentioned queries we create 90 similar queries that have the same pattern but with different constants. These queries are classified into two categories simple and complex. Simple queries are the ones that require few index lookups and little data exchange which make semi-join execution of these queries fast. Complex queries are the ones that need to exchange significantly large data when being executed in a distributes semi-join fashion or the ones that are non-processable by the 2-hop guarantee configuration of [4]. 20 of the 90 queries were classified as complex and the remaining 70 were classified as simple. We generated simple and complex query workloads randomly selected from the pools of simple and complex queries, respectively. Each workload consists of 1000 queries. Also, we generate a 1000 mixed

	PHD	1-hop	2-hops
pre-processing (hrs)	0	9.2	13.25
Data Loading and indexing (sec)	51.91	71.05	175.95
Collecting Statistics (sec)	5.70	0	0

Table 7.1: LUBM: Data to query Time

workload randomly selected from the pool of all queries both simple and complex.

7.2 Data-to-query time

In this experiment we will show how PHD-Store significantly reduces the pre-processing time incurred in [4] while achieving better or comparable results when answering queries. One of the key contributions of PHD-Store, is that data need not localize on a single machine, or a particular compute cluster to perform pre-processing. Instead, PHD-Store gathers approximate statistics about the data distribution that helps guide the graph walk algorithm as discussed in Chapter 6. This is mainly due to the fact that PHD-Store does not run communication-intensive algorithms such as graph minimal cuts.

Partitioning the graph using the N-Hop guarantee algorithm accounts for most of the pre-processing time for [4]. In contrast, PHD-Store runs a light-weight statistics calculation process, which takes up most of the pre-processing time as shown in this experiment. In addition, [4] takes more time to load and index the data compared to PHD-Store. This is mainly caused by the severe replication that they introduce in the N-Hop guarantees process. As a result, PHD-Store manages to answer each of the query batches (simple, complex and mixed) in a very short total time compared to [4]. Table 7.1 shows the drastic difference in preprocessing and querying time between PHD-Store and [4]. The next section will discuss in details the execution times for

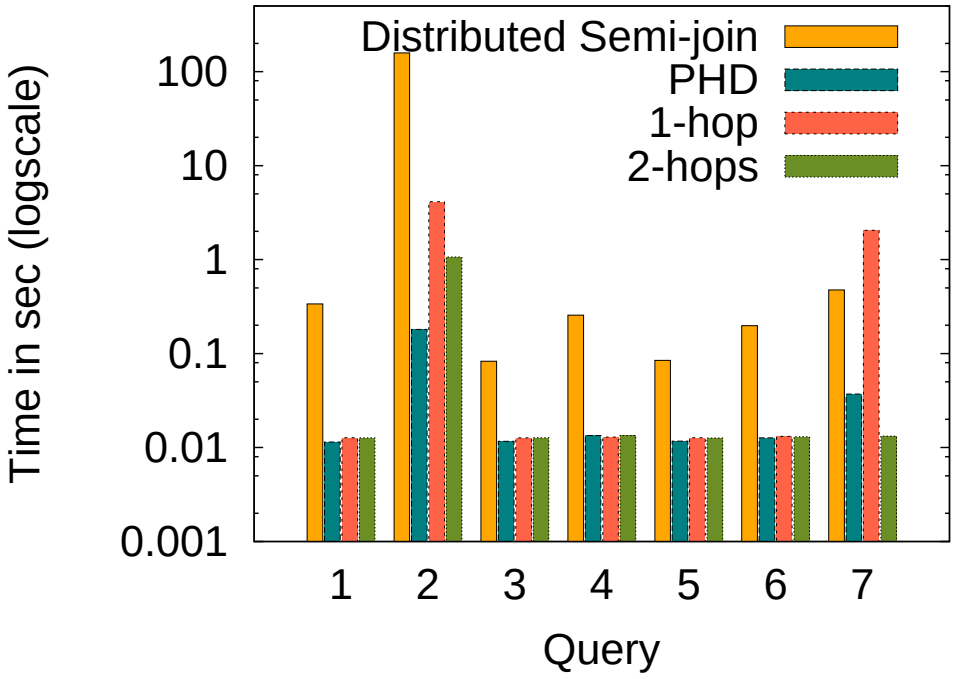

Figure 7.1: Execution time for LUBM queries 1-7

single queries as well as for query loads defined above.

7.3 Performance evaluation

In this experiment, we show the effectiveness of the PHD algorithm when executing each of the benchmarked queries. We note that these times do not include the redistribution time itself, but only the query execution after indexing in all systems.

Figures 7.1 and 7.2 show the execution time for the benchmarked queries running on four systems: Semi-Join, PHD-Store, 1-Hop and 2-Hop. In most of the queries, PHD-Store achieves better or comparable performance to both configurations of [4], without incurring the cost of pre-processing (note the logarithmic scale on the y-axis). In [4], it was reported that changing the hop guarantee from 1-hop to 2-hop has a significant effect on queries (2, 7 and 8) as this change would make these

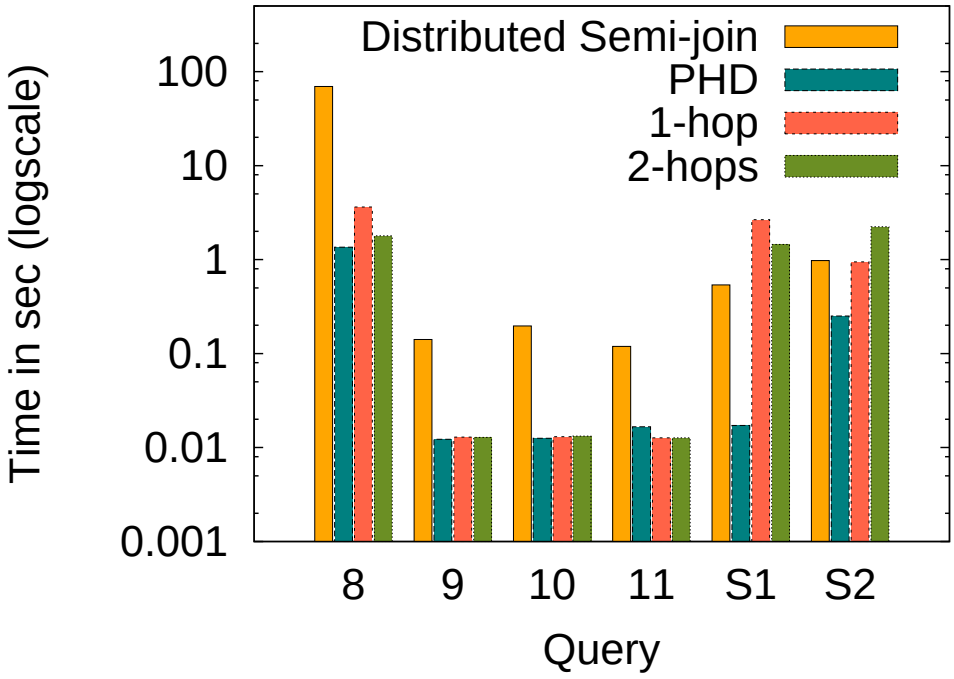

Figure 7.2: Execution time for LUBM queries 8-13.

queries processable without communication. However, in our implementation of [4], only query 7 has been significantly affected by this change. This is because of the way we implement the distributed join using MPI. In contrast, distributed join in [4] is implemented using Hadoop, which needs about 20 seconds to start, and in addition writes to disk in multiple iterations, causing total join time to be higher. The communication overhead shows clearly in query 7 since the join column that is shipped between machines is significantly large. In Q7, PHD-Store performs worse than the 2-hop configuration of [4] because Q7 gets distributed with a constant (university0) as the core. Accordingly only one machine (to which university0 is hashed) will answer the query. In contrast, multiple machines will participate in answering this query without communication in the 2-hop configuration of [4].

Q13 has the exact same execution for both configurations of [4] because of the existence of the high degree node `?u` which is of type University. Regardless of the

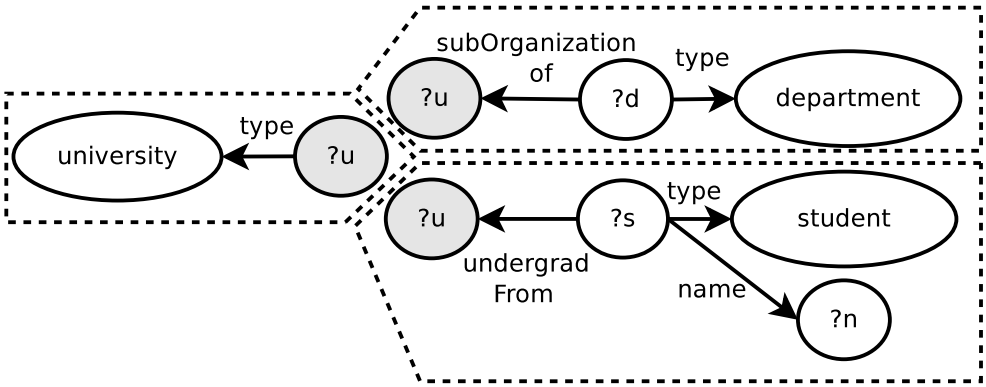

Figure 7.3: Q13 decomposed into 3 sub queries

hop-guarantee, both configurations would decompose the query into the three sub-queries shown in Figure 7.3. Each sub-query is then executed individually without communication, and the results are joined on `?u`. The 1-hop version performs better than the 2-hop version due to the significant difference in replication which makes the execution of each sub-query faster. Both configurations incur the same communication overhead when executing the final distributed semi-join, as the sizes of the intermediate results are exactly the same.

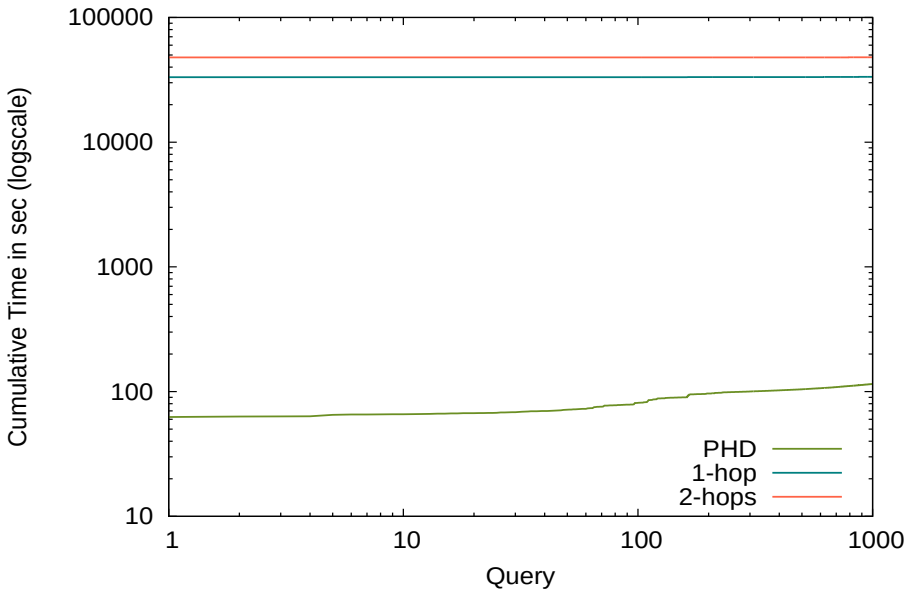

Figure 7.4: Cumulative execution time for LUBM simple load

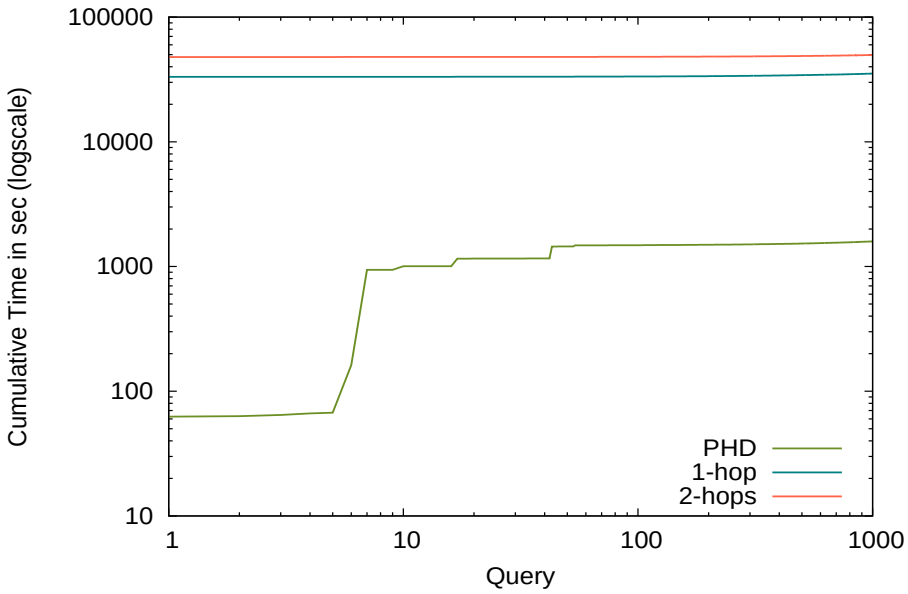

Figure 7.5: Cumulative execution time for LUBM complex load

Figures 7.4, 7.5 and 7.6 show the time it takes each of the aforementioned system to execute the simple, complex and mixed workloads respectively. It is clear from these figures that PHD-Store executes all of these batches of queries in a very short turn-around time compared to both configurations of [4]. PHD-Store starts answering queries immediately after loading the data and incurs some overhead at the beginning of the workload to distribute for new hot queries, which is shown by the jumps on the PHD-Store trend. However, the system adapts and achieves better or comparable performance compared to [4] at the end of each of the three workloads. Without considering the preprocessing time, the 2-hop version of [4] performs better than all other systems in the simple workload, since all the queries in this query load are processable without communication. On the other hand, the 1-Hop version takes significantly more time than the other systems due to the existence of some queries that are beyond 1-hop and hence need more time to execute (note the logarithmic y-axis scale). PHD-Store takes a reasonable time to execute the simple work load given the fact that the system would have to execute each unique query in the load at least

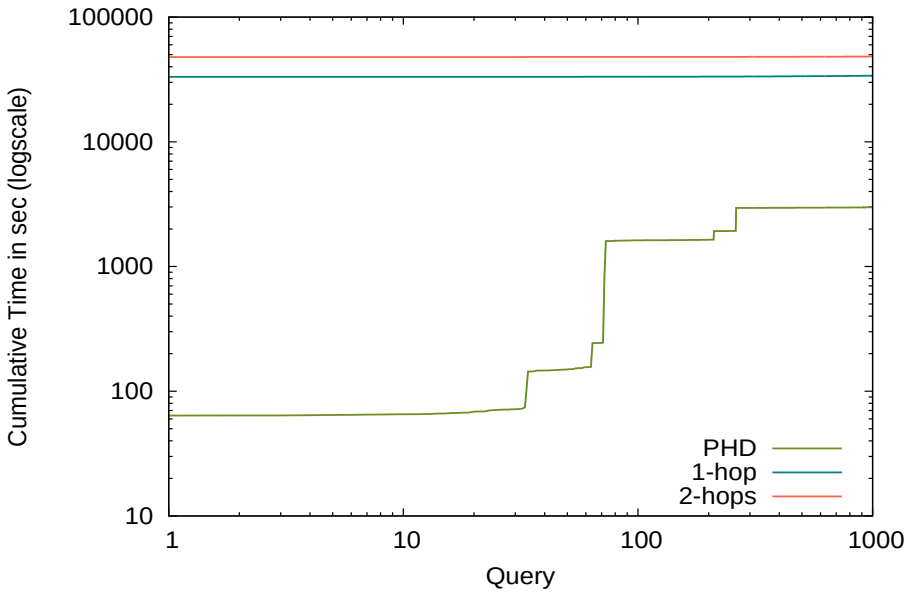

Figure 7.6: Cumulative execution time for LUBM mixed load

once in distributed semi-join fashion, before it decides to redistribute for it using the PHD algorithm. Only then the query becomes processable without communication.

In the other two workloads, particularly the mixed, PHD-Store initially spends some time distributing for new hot queries but then converges after executing approximately the 100th query. Note that although the mixed load consists of 90 queries, there are only few spikes (redistributions) because later queries benefit from previous redistributions.

Figure 7.7 shows the amount of data communicated as more queries are posed to the system. As the system answers more queries and adapts, less data need to be communicated in order to solve frequent queries. The reduction in the number of triples transferred shown in the figure is the main source of performance gain in PHD-Store. Note the logarithmic scale on the y-axis.

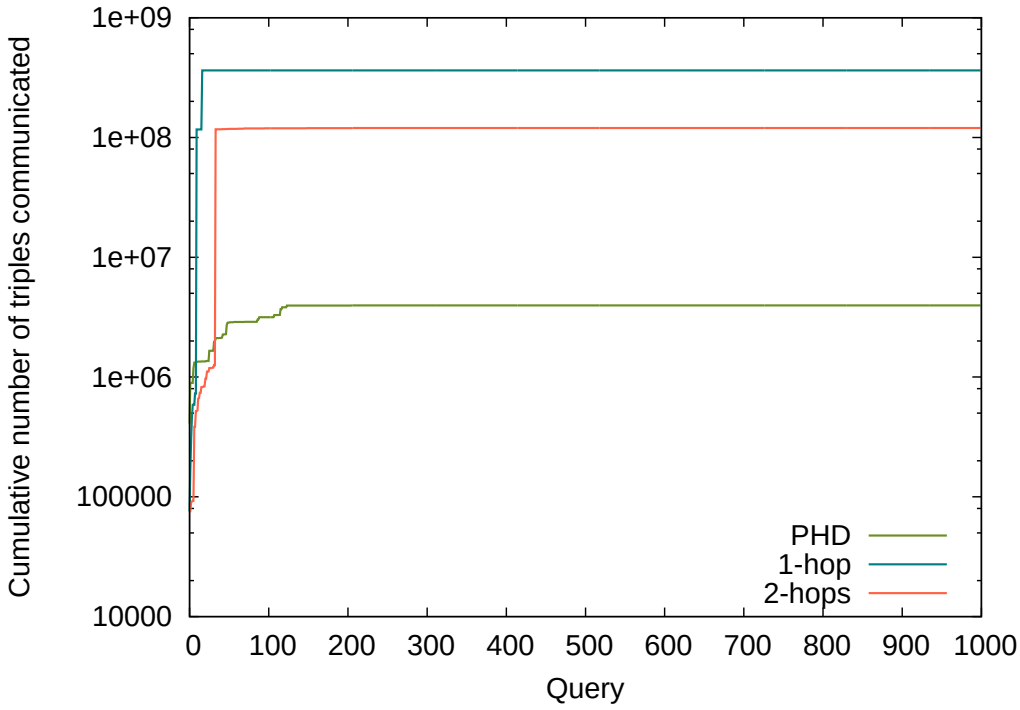

Figure 7.7: Cumulative data communicated (in triples) for mixed workload

7.4 PHD distribution cost

In the current implementation of the system, the system needs to go offline to execute the PHD distribution for a query. During this time, we argue that a separate thread can utilize the main data indices on the workers to answer queries in a traditional distributed semi-join manner, or even use the replica indices if changes can be committed inside a transaction. In this experiment, we show the time needed for the system to redistribute each of the workload queries according to the PHD algorithm, in comparison with the original time it took to solve without using any indexing. Figure 7.8 shows that the time the system needs to redistribute a query is comparable to the time it needs to execute in semi-join. However, for some queries, the PHD cost is higher due to the order of the joins dictated by the PHD algorithm. Nonetheless, the overhead can be tolerable considering the cost of pre-processing and performance

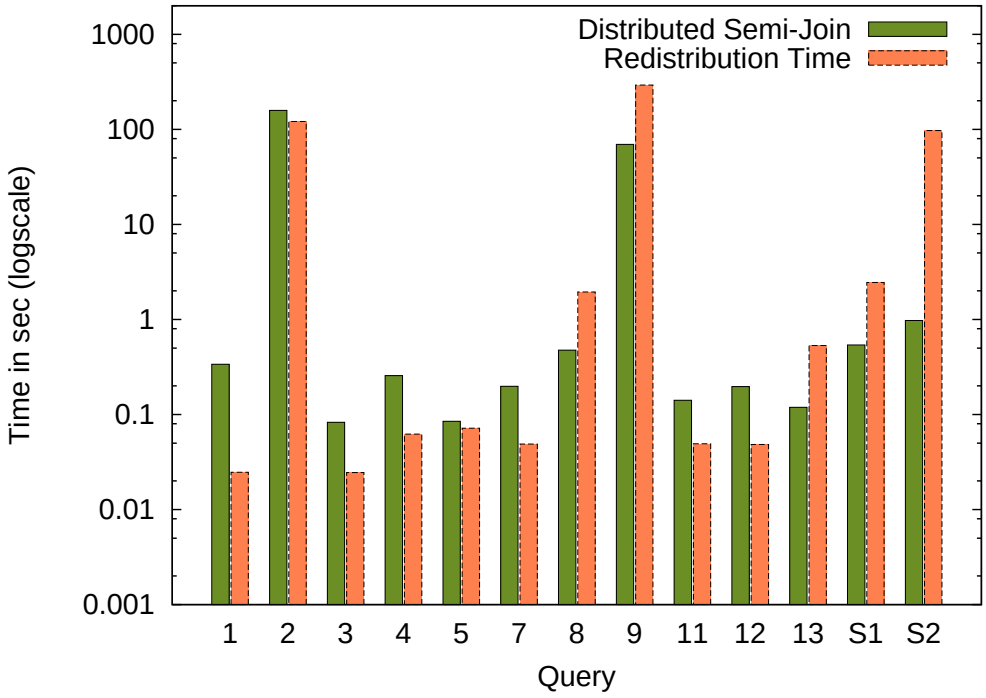

Figure 7.8: PHD redistribution time for all LUBM queries

gain after redistribution. Note the logarithmic scale on the y-axis in the figure.

7.5 Replication evaluation

In order to show the viability of our graph walk heuristic, we distribute every query individually using two approaches when walking the query graph. The first approach is our heuristic discussed in Chapter 5 where we walk the query graph from high score vertices to low score vertices. The second approach tries the opposite direction, walking the query graph from low to high score vertices. Figure 7.9 shows the effect of following our heuristic on replication. It is clear that walking the graph from high to low vertices results in orders of magnitude less replication (note the logarithmic scale on the y-axis). In S2, when following the high to low strategy we distribute the query with the vertex `?u` as a core which is considered to be a generic query distribution

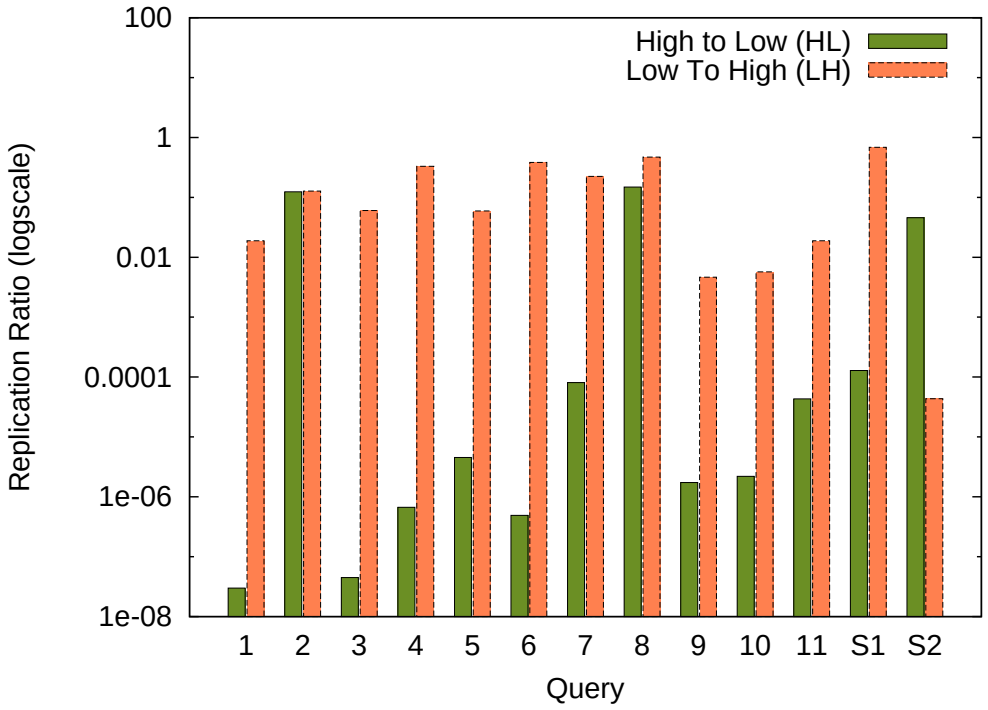

Figure 7.9: PHD replication ratio comparing High-To-Low versus Low-To-High graph walks

and hence many future queries can benefit from such a distribution. On the other hand, walking the graph using the low to high strategy results in distributing the query with a constant as a core. This distribution would result in less replication. However, such a replication will only be useful for future queries that have the name of university0 as a core.

Figure 7.10 shows the net replication ratio after finishing the simple, complex and mixed workloads for all the benchmarked systems. The replication ratios for both configurations of [4] do not change and are not affected by the workloads as partitioning happens in the pre-processing step. PHD-Store (HL) incurs very minimal replication when executing the simple workload because many queries in the load share the same core. Accordingly, many queries will share the same distribution and hence less replication. In the complex queries load, PHD-Store results in a higher replication ratio when compared to the simple load as the queries being distributed are

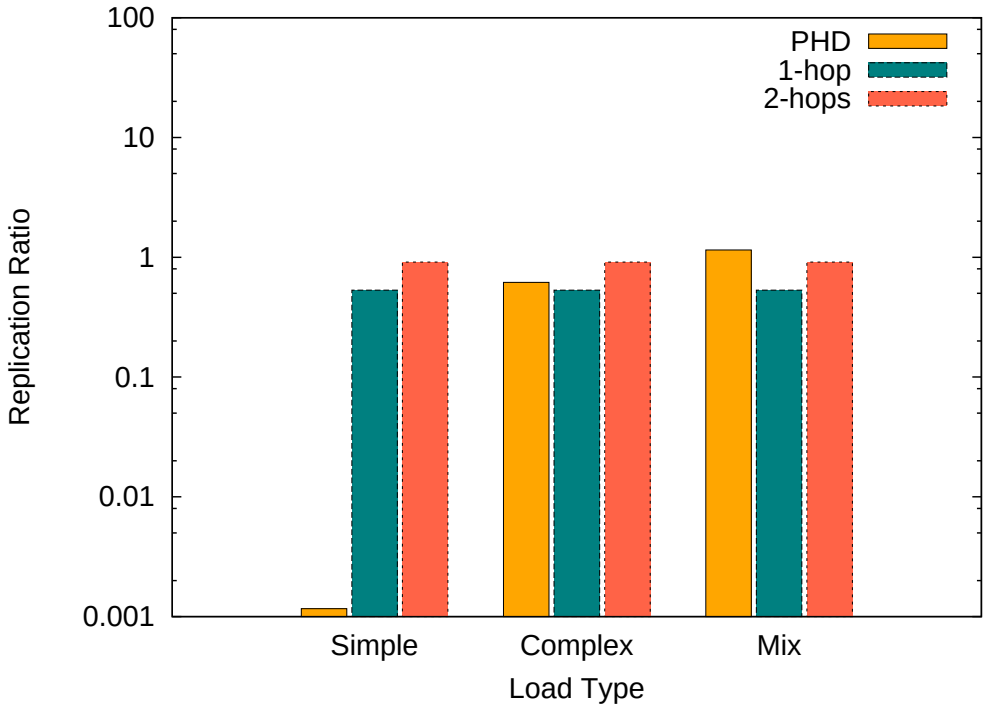

Figure 7.10: Replication ratio for 1000-query loads

all generic. Also, the load contains some long chain queries. These long chain queries cannot be answered by the 2-hop guarantee and would have resulted in replication explosion for [4] to include them in the guarantee. The same argument applies to the mixed load.

7.6 Load Balance

In this experiment we will show that the PHD algorithm will not cause load imbalance between the workers after redistribution. We used the Gini coefficient or the Gini index as a measure of the load dispersion between workers. In other words, it is used as a measure of inequality between workers' loads. It can range from zero to one and the closer the coefficient is to zero the better and vice versa. Figure 7.11 shows the Gini coefficient after each query execution in the workloads for all systems.

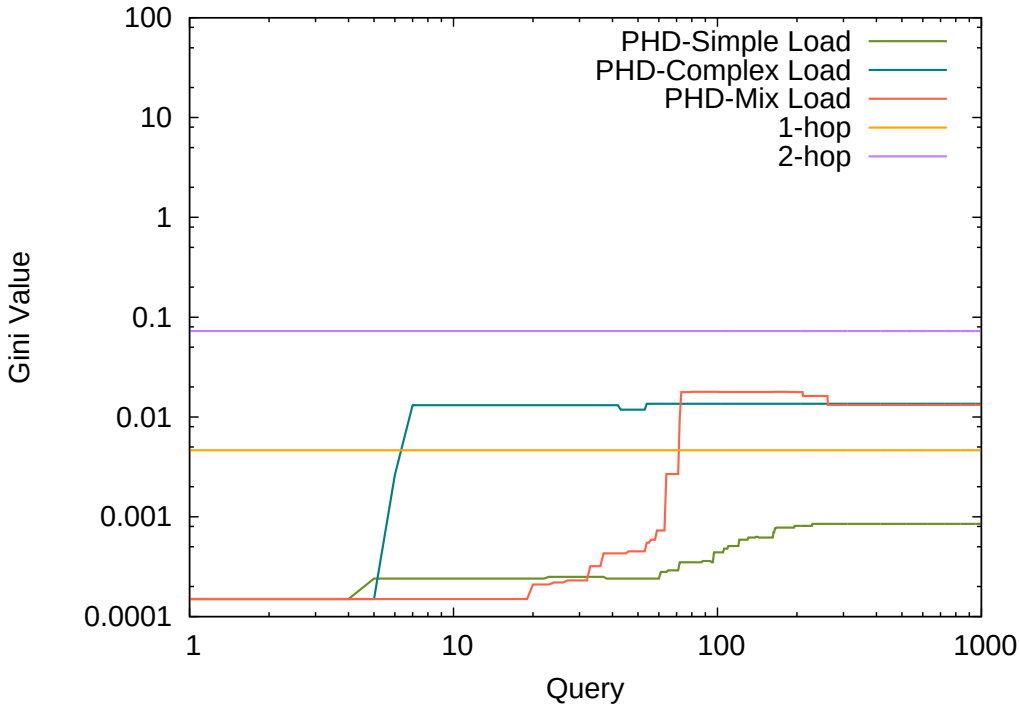

Figure 7.11: Gini coefficient changes as workloads execute

The Gini coefficient of both configurations of [4] will be constant throughout the load execution as the triple placement happens as a pre-processing step. However, since PHD-Store adapts to query loads the Gini coefficient will change each time a query gets redistributed using the PHD algorithm. The figures show that even though we are using a simple hash function to distribute the core vertices around which data are clustered, the amount of dispersion is still maintained to be low. And in all cases the differences in number of triples between workers are better when compared to the 2-hops guarantee configuration of [4].

Chapter 8

CONCLUDING REMARKS

In this document, we have presented out work on RDF data storage and processing, called PHD-Store. PHD-Store minimizes the data-to-query time by answering queries on raw graph data without any indexing, avoiding the conventional expensive pre-processing step. The system recognizes important queries, and creates replicas of the data in such a way to optimize execution of those queries and similar ones in the future. This way, PHD-Store is adaptive in the face of workload fluctuations, and keeps modifying its replicated data to suit the most recent set of queries. PHD-Store depends on replication to optimize query execution, and includes a heuristic to reduce the amount of replication needed for each query by following a clustering-like algorithm for replication. We compare our performance against a state-of-the-art approach that depends also on replication, and we show the drastic difference in performance, favoring the no pre-processing approach of PHD-Store.

There are a number of aspects to enhance in PHD-Store for future work:

- Analyzing incoming queries to decide if they can be answered in part using the current distribution of the data. So far the system can only decide whether a query can be fully answered in parallel using the current distribution. The same point also applies to deciding whether a new query is a subset of an already

redistributed query, and therefore breaking the new query down to two smaller queries to utilize the distribution.

- Including a full-fledged instance of a local index system such as RDF-3X for a better support of query operators. For example, the system currently does not support queries joining on the predicate column. We claim that such queries are very infrequent, and the system can still answer them without using the index.

- Using a more sophisticated measure for deciding on important queries than the currently used MRU policy. Important factors such as the size of the query-induced replication and the frequency of the query have to be included in the decision making.

We also argue that the Propagating Hash Distribution algorithm presented in this work also applies to general purpose graph data. We are also looking forward to exploring working on other popular graph operators aside from join, such as graph random walks, breadth- and depth-first search, and so forth.

Bibliography/References

[1] "The large hadron collider," http://lhc.web.cern.ch/lhc/.

[2] "The large synoptic survey telescope," http://www.lsst.org/lsst/science.

[3] "Rdf primer," http://www.w3.org/TR/rdf-primer/.

[4] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proceedings of the VLDB Endowment*, vol. 4, 2011. [Online]. Available: http://cs-www.cs.yale.edu/homes/dna/papers/sw-graph-scale.pdf

[5] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Comput. Surv.*, vol. 16, no. 2, Jun. 1984. [Online]. Available: http://doi.acm.org/10.1145/356924.356928

[6] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment*, no. 1, pp. 647–659, 2008.

[7] J. Broekstra, A. Kampman, and F. V. Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema." Springer, 2002, pp. 54–68.

[8] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle, "The rdfsuite: Managing voluminous rdf description bases," 2000, pp. 1–13.

[9] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds, "Efficient rdf storage and retrieval in jena2," in *EXPLOITING HYPERLINKS 349*, 2003, pp. 35–43.

[10] D. J. Abadi, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," 2007.

[11] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "Yars2: A federated repository for querying graph structured data from the web," in *of Lecture Notes in Computer Science*. Springer, 2007, pp. 211–224.

[12] B. Liu and B. Hu, "HPRD: a high performance RDF database," *International Journal of Parallel, Emergent and Distributed Systems*, 2010.

[13] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceedings of the VLDB Endowment*, no. 1, pp. 1008–1019, 2008.

[14] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store," in *Programming Support Innovations for Emerging Distributed Applications*, ser. PSI EtA '10, 2010.

[15] O. Erling, "Towards web scale rdf," *4th International Workshop on Scalable Semantic Web Knowledge Base Systems SSWS2008*, vol. 295, 2008. [Online]. Available: http://www.openlinksw.com/weblog/oerling/2008webscale_rdf.pdf

[16] G. Karypis and V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0," http://www.cs.umn.edu/~metis, University of Minnesota, Minneapolis, MN, 2009.

[17] "Apache hadoop," http://hadoop.apache.org/.

[18] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning,"

Proc. VLDB Endow., pp. 48–57, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920853

[19] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of the 2012 international conference on Management of Data*, ser. SIGMOD '12. ACM, 2012, pp. 517–528. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213895

[20] 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 *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10, 2010.

[21] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR*, 2007, pp. 68–78.

[22] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *Proceedings of the 2012 international conference on Management of Data*, ser. SIGMOD '12. ACM, 2012, pp. 241–252.

[23] "Lubm sparql benchmark," http://swat.cse.lehigh.edu/projects/lubm/.

APPENDICES

Appendix A

QUERIES

List of queries taken from LUBM benchmark, and added to the final workloads.

A.1 Q1

```
SELECT ?x WHERE {
        ?x <type> <GraduateStudent> .
        ?x <takesCourse> <http://www.Department0.University0.
           edu/GraduateCourse0> .
}
```

A.2 Q2

```
SELECT ?x WHERE {
        ?x <type> <GraduateStudent> .
        ?y <type> <University> .
        ?z <type> <Department> .
        ?x <memberOf> ?z .
```

```
        ?z <subOrganizationOf> ?y .
        ?x <undergraduateDegreeFrom> ?y .
}
```

A.3 Q3

```
SELECT ?x WHERE {
        ?x <type> <Publication> .
        ?x <publicationAuthor> <http://www.Department0.
            University0.edu/AssistantProfessor0> .
}
```

A.4 Q4

```
SELECT ?x WHERE {
        ?x <type> <Professor> .
        ?x <worksFor> <http://www.Department0.University0.edu
           > .
        ?x <name> ?y1 .
        ?x <emailAddress> ?y2 .
        ?x <telephone> ?y3 .
}
```

A.5 Q5

```
SELECT ?x WHERE {
        ?x <type> <Person> .
```

```
        ?x <memberOf> <http://www.Department0.University0.edu
           > .
}
```

A.6 Q7

```
SELECT ?x WHERE {
        ?x <type> <Student> .
        ?y <type> <Course> .
        <http://www.Department0.University0.edu/
           AssociateProfessor0> <teacherOf> ?y .
        ?x <takesCourse> ?y .
}
```

A.7 Q8

```
SELECT ?x WHERE {
        ?y <subOrganizationOf> <http://www.University0.edu> .
        ?y <type> <Department> .
        ?x <memberOf> ?y .
        ?x <type> <Student> .
        ?x <emailAddress> ?z .
}
```

A.8 Q9

```
SELECT ?x WHERE {
```

```
        ?x <advisor> ?y .
        ?y <type> <Faculty> .
        ?y <teacherOf> ?z .
        ?z <type> <Course> .
        ?x <takesCourse> ?z .
        ?x <type> <Student> .
}
```

A.9 Q11

```
SELECT ?x WHERE {
        ?x <type> <ResearchGroup> .
        ?x <subOrganizationOf> <http://www.University0.edu> .
}
```

A.10 Q12

```
SELECT ?x WHERE {
        ?x <headOf> ?y .
        ?y <type> <Department> .
        ?x <worksFor> ?y .
        ?y <subOrganizationOf> <http://www.University0.edu> .
}
```

A.11 Q13

```
SELECT ?x WHERE {
```

```
        ?x <type> <Person> .
        ?x <degreeFrom> <http://www.University0.edu> .
}
```

A.12 S1

```
SELECT ?x WHERE {
        ?u <type> <University> .
        ?p <type> <Professor> .
        ?s <type> <GraduateStudent> .
        ?c <type> <Course> .
        ?d <type> <Department> .
        ?p <teacherOf> ?c .
        ?s <takesCourse> ?c .
        ?p <headOf> ?d .
        ?d <subOrganizationOf> ?u .
        ?s <name> ?n .
}
```

A.13 S2

```
SELECT ?x WHERE {
        ?x <type> <GraduateStudent> .
        ?y <type> <University> .
        ?z <type> <Department> .
        ?z <subOrganizationOf> ?y .
        ?x <undergraduateDegreeFrom> ?y .
}
```