Accelerating SPARQL Queries and Analytics on RDF Data

Dissertation by

Razen Mohammad Al-Harbi

In Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy

King Abdullah University of Science and Technology

Thuwal, Kingdom of Saudi Arabia

October, 2016

EXAMINATION COMMITTEE

The dissertation of Razen Mohammad Al-Harbi is approved by the examination committee.

Committee Chairperson: Panos Kalnis - professor
Committee Members: Khaled Salama - Associate Professor, Marco Canini - Assistant Professor, Michail Vlachos - IBM Zurich Research Laboratory

ABSTRACT

Accelerating SPARQL Queries and Analytics on RDF Data

Razen Mohammad Al-Harbi

The complexity of SPARQL queries and RDF applications poses great challenges on distributed RDF management systems. SPARQL workloads are dynamic and consist of queries with variable complexities. Hence, systems that use static partitioning suffer from communication overhead for workloads that generate excessive communication. Concurrently, RDF applications are becoming more sophisticated, mandating analytical operations that extend beyond SPARQL queries. Being primarily designed and optimized to execute SPARQL queries, which lack procedural capabilities, existing systems are not suitable for rich RDF analytics.

This dissertation tackles the problem of accelerating SPARQL queries and RDF analytics on distributed shared-nothing RDF systems. First, a distributed RDF engine, coined AdPart, is introduced. AdPart uses lightweight hash partitioning for sharding triples using their subject values; rendering its startup overhead very low. The locality-aware query optimizer of AdPart takes full advantage of the partitioning to (i) support the fully parallel processing of join patterns on subjects and (ii) minimize data communication for general queries by applying hash distribution of intermediate results instead of broadcasting, wherever possible. By exploiting hash-based locality, AdPart achieves better or comparable performance to systems that employ sophisticated partitioning schemes.

To cope with workloads dynamism, AdPart is extended to dynamically adapt to workload changes. AdPart monitors the data access patterns and dynamically redistributes and replicates the instances of the most frequent patterns among workers.

Consequently, the communication cost for future queries is drastically reduced or even eliminated. Experiments with synthetic and real data verify that AdPart starts faster than all existing systems and gracefully adapts to the query load.

Finally, to support and accelerate rich RDF analytical tasks, a vertex-centric RDF analytics framework is proposed. The framework, named SPARTex, bridges the gap between RDF and graph processing. To do so, SPARTex: (i) implements a generic SPARQL operator as a vertex-centric program. The operator is coupled with an optimizer that generates efficient execution plans. (ii) It allows SPARQL to invoke vertex-centric programs as stored procedures. Finally, (iii) it provides a unified in-memory data store that allows the persistence of intermediate results. Consequently, SPARTex can efficiently support RDF analytical tasks consisting of complex pipeline of operators.

ACKNOWLEDGEMENTS

All praise is due to Allah (God), the Lord of the Worlds. His blessings and peace be upon our Prophet Mohammad. I am forever in my father's debt for believing in me, may the lord bless his soul. No success would materialize in my life without my mother, Sabah; her prayers, support and uplifting are the main reasons for my determination. I stand speechless in front of the sacrifices, dedication and support endowed by my other half, Rihab. This chapter of my life would not have been completed without her presence and the inspiration I receive from my kids: Mohammad, Majed and Malik. Also, I would like to thank my family for their support. In particular, I thank my sister and brother in-law, Rezan and Majed for their support; without them, I would not have become the person I am now.

I am filled with gratitude to my advisor, Professor Panos Kalnis, for his patience, guidance and endless support. My research journey would have not succeeded without his contributions. I am grateful to Saudi Aramco for supporting my graduate-level studies. Special thanks go to my ENOD/HPCG family for supporting me during the writing of this thesis. I sincerely thank Professor Mootaz Elnozahy and Professor Nikos Mamoulis for their valuable comments and feedback on this thesis.

I am thankful to all members of the InfoCloud group and my friends at KAUST. Special thanks go to Dr. Majed Sahli, Ibrahim Abdelaziz, Dr. Essam Mansour, Dr. Ahmad Showail and Mohammad Alarawi for enriching my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

> Strive always to excel in virtue and truth.
>
> Prophet Mohammad (PBUH)
> 571 — 633 CE

Resource Description Framework (RDF) [1, 2] is a standard data model and the core component of the W3C Semantic Web [3, 4]. The Simple Protocol And RDF Query Language (SPARQL) [5, 6] is the official W3C standard query language for querying and extracting information from RDF data. RDF was originally designed to be a meta-data model for describing web pages. However, it is now a standard model for exchanging data and knowledge among various data sources on the Web.

The decoupling between RDF and its schema allows the schema to freely change without affecting users. Therefore, Social networks, commercial search engines, online shopping and scientific databases are adopting RDF for exchanging data or publishing contents. This wide adoption has lead to an ever increasing volume of publicly available RDF data on the Web. Public knowledge bases and databases, such as Universal Protein Resource (UniProtKB) [7], PubChemRDF [8], DBpedia [9], Bio2RDF [10] and Probase [11] have billions of facts in RDF format. These knowledge bases are usually linked, as in the Linked Open Data (LOD) [12, 13] cloud, and are globally queried using SPARQL [14, 15, 16].

RDF datasets consist of triples of the form ⟨subject, predicate, object⟩, where

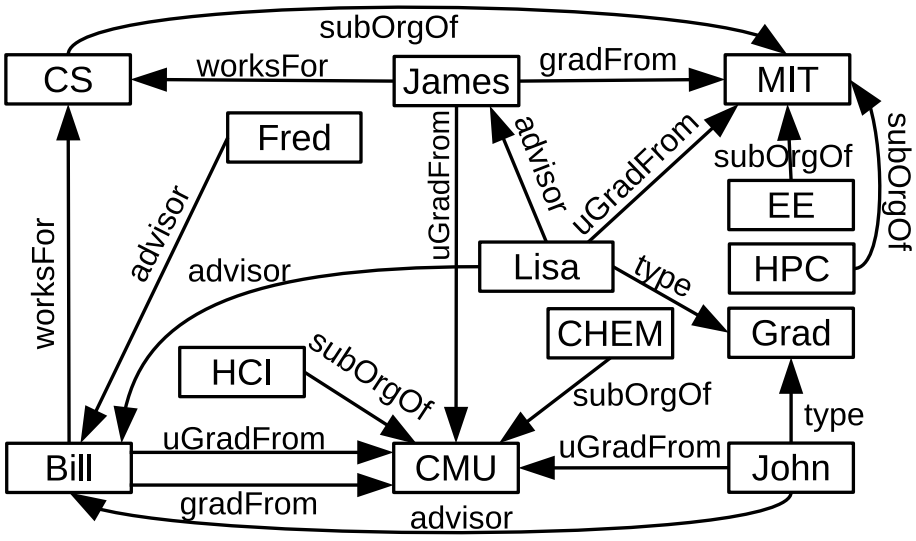


Figure 1.1: Example RDF graph. An edge and its associated vertices correspond to an RDF triple; e.g., ⟨Bill, worksFor, CS⟩.

predicate represents a relationship between two entities: a subject and an object. An RDF dataset can be regarded as a long relational table with three columns. It can also be viewed as a directed labeled graph, where vertices and edge labels correspond to entities and predicates, respectively. Figure 1.1 shows an example RDF graph of students and professors in an academic network. Table 1.1 shows a tabular representation of some triples from Figure 1.1.

Table 1.1: Example RDF data. Each row constitute a triple.

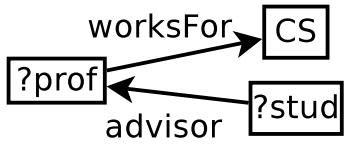
subject	predicate	object
HPC	subOrgOf	MIT
EE	subOrgOf	MIT
CS	subOrgOf	MIT
James	gradFrom	MIT
Lisa	uGradFrom	MIT
HCI	subOrgOf	CMU
CHEM	subOrgOf	CMU
Bill	gradFrom	CMU

In its simplest form[1], a SPARQL query consists of a set of RDF triple patterns; some of the nodes in a pattern are variables which may appear in multiple patterns.

[1]This form is usually referred to as Basic Graph Pattern (BGP).

```
SELECT ?prof ?stud WHERE {
 ?prof    worksFor  CS .
 ?stud    advisor   ?prof .
}
```

(a) SPARQL



(b) Graph

Figure 1.2: A query that finds CS professors with their advisees.

For example, the query in Figure 1.2(a) returns all professors who work for CS with their advisees. The query corresponds to the graph pattern in Figure 1.2(b). The answer is the set of ordered bindings of (?prof, ?stud) that render the query graph isomorphic[2] to subgraphs in the data. Assuming the data is stored in a table $D(s, p, o)$, the query can be answered by first decomposing it into two subqueries, each corresponding to a triple pattern: $q_1 \equiv \sigma_{p=worksFor \wedge o=CS}(D)$ and $q_2 \equiv \sigma_{p=advisor}(D)$. The subqueries can be answered independently by scanning table D; then, their intermediate results are joined on the subject and object attribute: $q_1 \bowtie_{q_1.s=q_2.o} q_2$. By applying the query on the data of Figure 1.3, we get `(?prof, ?stud)` $\in$ {`(James, Lisa),(Bill, John), (Bill, Fred),(Bill, Lisa)`}.

As the volume of RDF data continues soaring, managing, indexing and querying RDF data becomes challenging. Early research efforts on RDF data management resulted in efficient centralized RDF systems; like chameleon-db [17], RDF-3X [19, 20, 21], HexaStore [22], TripleBit [23], BitMat [24] and gStore [25]. However, centralized data management does not scale well for complex queries on web-scale RDF data [26, 27]. To cope with the massive data growth, many distributed [26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38] RDF management systems have been introduced. These systems scale-out to overcome the limitations of single-machine stores. (i) They are capable of handling large datasets by dividing or partitioning the data among multiple machines (workers). (ii) They reduce the total running time by distributing data

[2]When a query has variable predicates, its evaluation becomes a subgraph homomorphism problem [17, 18]

processing and querying workload among the workers. Generally, answering queries involves local execution followed by communication between workers.

Despite the advances in distributed RDF systems, current systems cannot sustain good performance for different types of SPARQL workload [17, 39] and RDF analytical tasks. Specifically, there are two problems that contribute to this limitation. First, all existing systems rely on static partitioning; and assume that one partitioning scheme fits all workloads. However, SPARQL workloads are very diverse [13, 17, 40, 41] and dynamic [17, 42]. A single workload can have queries with different complexities [17]. Hence, there will always be queries that are not favored by the partitioning scheme used by the system. Furthermore, the workload itself is very dynamic [17, 42]; the part of the data that is being queried now may not be queried in the future. Under these conditions, current systems cannot provide good performance for the entire workload. Consequently, the overall system performance will degrade even if a small percentage of the workload is not processed efficiently.

The second problem lies in the lack of generality in current distributed RDF systems. Existing systems are primarily designed to model, store and query RDF data behind a SPARQL end point. Hence, they are not capable of supporting rich RDF analytical tasks [43, 44, 45, 46], where native graph processing is needed. Merely utilizing a generic processing framework [47, 48, 49, 50, 51, 52, 53] for SPARQL query answering [26, 27, 28, 29, 30, 31, 33, 37, 38] or representing RDF data in graph native format [27] is not sufficient for rich RDF analytical tasks. More importantly, many emerging RDF applications [43, 44, 45, 46] require both SPARQL querying[3] and generic processing. Accordingly, researchers and users resort to utilize multiple and usually different systems for analyzing and processing RDF data [44]; mandating expensive data shuffling and formating in a single or among multiple system(s).

The overarching outcome of this thesis is to accelerate and efficiently process

[3]These applications require inferencing capabilities as well; however, inferencing is outside the scope of this thesis.

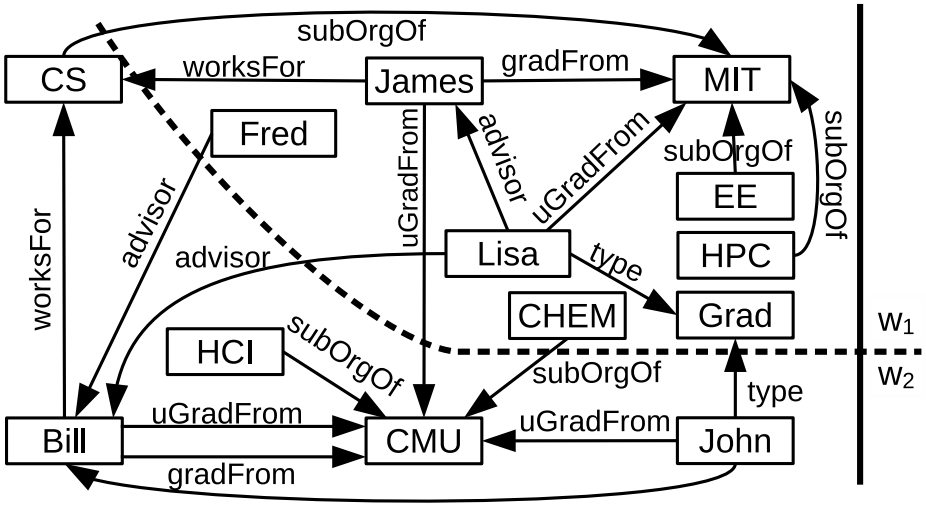


Figure 1.3: The graph in Figure 1.1 is partitioned among two workers: W1 and W2.

SPARQL queries and RDF analytical tasks on distributed shared nothing environments. This chapter is organized as follows. Section 1.1 details the limitations of static partitioning schemes. Section 1.2 discuses the problem of lacking generality in existing distributed RDF data management systems. Section 1.3, describes solutions to these problems, give an overview of this thesis and a list of its contributions.

1.1 Limitations of Static Partitioning Schemes

Distributed RDF management systems scale-out by partitioning RDF data among many compute nodes (workers); hence queries can be evaluated in a distributed fashion. A SPARQL query is decomposed into multiple subqueries that are evaluated at each node independently. Since data is distributed, the nodes may need to exchange intermediate results during query evaluation. Consequently, queries with large intermediate results incur high communication cost, which is detrimental to the query performance [26, 32].

Distributed RDF systems aim at minimizing the number of decomposed subqueries by partitioning the data among workers. The goal is that each node has all the data it needs to evaluate the entire query and there is no need for exchanging intermediate

Table 1.2: A triple is placed at the worker which stores the subject entity.

W1

subject	predicate	object
HPC	subOrgOf	MIT
EE	subOrgOf	MIT
CHEM	subOrgOf	CMU
James	worksFor	CS
James	uGradFrom	CMU
James	gradFrom	MIT
Lisa	uGradFrom	MIT
Lisa	type	Grad
Lisa	advisor	James
Lisa	advisor	Bill
Fred	advisor	Bill

W2

subject	predicate	object
CS	subOrgOf	MIT
HCI	subOrgOf	CMU
Bill	worksFor	CS
Bill	gradFrom	CMU
Bill	uGradFrom	CMU
John	type	Grad
John	uGradFrom	CMU
John	advisor	Bill

results. In such a *parallel* query evaluation, each node contributes a partial result of the query; the final query result is the union of all partial results. To achieve this, some triples may need to be replicated across multiple partitions.

For example, in Figure 1.3, assume the data graph is divided by the dotted line into two partitions and assume that triples follow their subject placement (see Table 1.2). To answer the query in Figure 1.2, nodes have to exchange intermediate results because triples ⟨Lisa, advisor, Bill⟩ and ⟨Fred, advisor, Bill⟩ cross the partition boundary. Replicating these triples to both partitions allows each node to answer the query without communication. Still, even sophisticated partitioning and replication cannot guarantee that arbitrarily complex SPARQL queries can be processed in parallel; thus, expensive *distributed* query evaluation, with intermediate results exchanged between nodes, cannot always be avoided.

Existing RDF systems inherently face three limitations due to static partitioning. (i) Partitioning cost: balanced graph partitioning is an NP-complete problem [54]; thus, existing systems perform heuristic-based partitioning. In systems that use simple hash partitioning heuristics [27, 28, 29, 30, 55, 33, 38], queries have low chances to be evaluated in parallel without communication between nodes. Hence, they incur

excessive communication during query evaluation. On the other hand, systems that use sophisticated partitioning heuristics [26, 31, 32, 36] suffer from high preprocessing cost and sometimes high replication. More importantly, they pay the cost of partitioning the entire data regardless of the anticipated workloads. However, as shown in a recent study [44], only a small fraction of the whole graph is accessed by typical real query workloads. For example, a real workload consisting of more than 1,600 queries executed on DBpedia (459M triples) touches only 0.003% of the whole data.

(ii) Inefficient distributed execution: All existing systems tend to focus on optimizing the execution of queries that are favored by their static partitioning schemes. Systems that rely on simple hash partitioning [27, 28, 29, 30, 55] perform very well for star[4] queries. For example, H2RDF+ leverages its sorted indices to efficiently execute star queries using multi-way merge join. Similarly, systems that employ sophisticated partitioning that rely on replication [26, 31, 32, 36] perform very well for queries that can be executed locally within each partition. However, for both schemes, queries that cannot be solved locally suffer from excessive communication. The effect is very substantial for systems that rely on MapReduce based distributed joins as shown in [32] and validated by this thesis.

(iii) Workload awareness and adaptivity: SPARQL query evaluation exhibits poor data locality, therefore, regardless of the partitioning heuristic used, there will always be queries that cross partition boundaries and require expensive distributed evaluation. Therefore, WARP [34] and Partout [35] consider the workload during data partitioning and achieve significant reduction in the replication ratio, while showing better query performance compared to systems that partition the data blindly. Nonetheless, both these systems assume a representative (i.e., static) query workload and do not adapt to changes. Aluç et al. [17, 39] showed that systems need to continuously adapt their physical layouts based on workloads in order to consistently

[4]Star queries consist of multiple triple patterns that share the same join column.

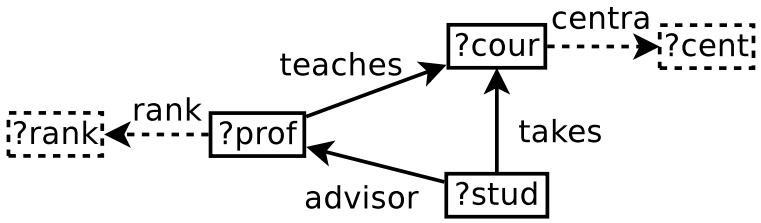


Figure 1.4: Q_s retrieves students who take courses taught by their advisors. It also retrieves professors' ranks and courses' centralities. Solid lines are part of the original graph while dotted lines represent triples that needs to be computed first.

provide good performance. The same concept is also applicable to data partitioning in distributed RDF systems.

Thus, this thesis argues that distributed RDF systems should start fast, evaluate queries efficiently and leverage query workloads to adapt dynamically.

1.2 Problems of Lacking Generality

An emerging new type of RDF analytics; in drug repositioning [45], biological data analysis [43, 46] and RDF data sampling [44], requires the combination of general graph processing with SPARQL structural queries. For example, Qu et al. [45] filter the results of SPARQL queries by a set of graph centrality algorithms to identify key biological entities within the resulting RDF subgraphs. Also, Rietveld et al. [44] use a pipeline of complex operations to analyze RDF data. They compute the degree centrality and PageRank of the corresponding RDF graph using a generic graph engine. The final computation results are written into RDF format. Finally, they run SPARQL queries against the mutated RDF graph enriched by centrality and PageRank information for each node in the graph. Other scenarios may require a variety of graph algorithms, such as reachability queries, or community detection.

All existing RDF management systems cannot support such applications natively. To see why, consider query Q_s in Figure 1.4. While it syntactically represents a normal SPARQL query, it has some special semantics. The query retrieves all students who

take courses taught by their advisors. However, it also retrieves the ranks of the matching professors and the centrality of their taught courses for a possible filtering step, where professors with high PageRank values are considered popular; while core courses have higher centrality as they are taken by all students.

Q_s has two types of triple patterns; the first type comes from the structure of the input graph (solid lines in Figure 1.4) while the other type of triples is derived from the vertex computed values (dotted lines in Figure 1.4). Therefore, PageRank and centrality algorithms need to be applied on the input graph prior to query evaluation. Consequently, to solve queries like Q_s, systems need to have all the following capabilities: (i) they should be able to efficiently execute general analytical algorithms as well as SPARQL queries. In other words, the data abstraction and physical layout should be suitable for both. (ii) They should support the execution of a pipeline of operators, where the output of one can be used by the others if needed without disturbing the original physical layout. For example, the PageRank and centrality results in Q_s should not affect the layout of the original RDF graph. Finally and more importantly, (iii) these pipelines should be triggered declaratively using SPARQL rather than writing special procedural code for sharing results among operators.

Obviously, all specialized [17, 19, 20, 21, 22, 23, 24, 25] RDF systems i.e. systems that are built natively to support SPARQL queries only, are not capable of solving Q_s. These systems use physical layouts and indices that are optimized for SPARQL. Furthermore, to support general analytical algorithms, they need to reinvent the wheel by implementing the whole graph analysis software stack. Therefore, the only way to evaluate queries like Q_s is to express analytical algorithms using SPARQL itself. Yet, because SPARQL is not a procedural language, expressing graph operations results in verbose and complex queries [56] that are hard to formulate and expensive to evaluate. This is evident in some recent works [56, 57, 58, 59] which are limited to a small set of graph operations like clustering and graph diffusion. Accordingly, data

movement and formatting in a single or multiple systems cannot be avoided when evaluating queries like Q_s.

Many RDF systems embark on generic distributed computation platforms, like Trinity [48], GraphLab [51], Hadoop [60], Spark-based GraphX [61, 62, 63, 64] and PigLatin [65] for distributed SPARQL query evaluation; enabling data processing beyond SPARQL. However, all distributed systems lack at least one (the third) or all the capabilities mentioned above. Particularly, Hadoop based systems [26, 28, 29, 30, 31, 33, 37, 38] are not suitable for graph processing [49]. The reason is that MapReduce requires passing the entire graph state from one iteration to the next. Furthermore, these systems model and index RDF data using traditional RDF schemes by creating indices on all permutations of subject, predicate and object or by using vertical partitioning schemes [33, 38, 66].

On the contrary, Trinity.RDF [27], Goodman et. al. [67] and S2X [68] use native graph representation to model RDF data on top of Trinity, GraphLab and GraphX, respectively; making them more suitable for RDF analytical tasks. However, aside from missing the third capability above, their SPARQL operators have other limitations. For example, Trinity.RDF uses graph exploration to minimize communication for SPARQL query evaluation. This approach requires a final sequential join at the master when solving queries with cycles. Similarly, S2x matches all triple patterns of the query in the first iteration. Hence, it generates large and usually unnecessary intermediate results, which significantly affect the performance [17]. The SPARQL operator in Goodman et. al. [67] does not have a query planner [5], a crucial component for efficient query execution.

[5]The authors claim that due simplicity of queries, query planning does not have significant impact on query performance. However, this thesis shows later that subquery ordering makes the difference between sub-second query evaluation and query timeout.

1.3 Contributions and Thesis Organization

The remainder of this dissertation is organized as follows: In Chapter 2, the related work in the area of RDF data partitioning and analytics is surveyed. Subsequently, the specific contributions of this thesis are introduced:

- Chapter 3 introduces AdPart, a distributed in-memory RDF engine. AdPart alleviates the limitations of static partitioning employed by existing distributed RDF systems. It uses lightweight initial partitioning that distributes triples by hashing on their subjects. This partitioning has low cost and does not incur any replication. Thus, the preprocessing time is low effectively addressing the partitioning cost limitation. AdPart exploits hash-based locality awareness to process in parallel (i.e., without data communication) the join patterns on subjects included in a query. In addition, intermediate results can potentially be hash-distributed to single workers instead of being broadcasted everywhere. The locality-aware query optimizer of AdPart considers these properties to generate an evaluation plan that minimizes intermediate results shipped between workers.

- In Chapter 4, AdPart is extended with an adaptive incremental redistribution feature. A hierarchical heat-map of accessed data patterns is maintained by AdPart to monitor the executed workload. Hot patterns are redistributed and potentially replicated in the system in a way that future queries that include them are executed in parallel by all workers without data communication. To control replication, AdPart operates within a budget and employs an eviction policy for the redistributed patterns. By using lightweight hash partitioning, avoiding the upfront cost, and adopting a pay-as-you-go approach, AdPart overcomes the limitations of static partitioning schemes. Accordingly, AdPart executes tens of thousands of queries on large graphs within the time it takes other systems to conduct their initial partitioning.

- Chapter 5, addresses the generality problem by introducing SPARTex, a distributed system for rich RDF data analytics. SPARTex is designed to be implemented on top of a variety of vertex-centric graph processing engines (e.g., Pregel-like). It extends and unify the in-memory graph representation of the underlying vertex-centric system to be used by generic vertex-centric programs as well as SPARQL. Coupled with a novel optimizer, SPARTex implements an efficient SPARQL query operator as a vertex-centric program. To facilitate rich RDF analytics, SPARTex is inspired by the database community, where the coupling of SQL code with a generic programming language (e.g., Java, C++) is common. Therefore, it allows users to write queries that combine declarative SPARQL queries with procedural code (vertex-centric programs) for generic graph processing. Different operators can be pipelined; and can communicate by setting vertices properties in the unified in-memory data store.

This dissertation contains published work and work to be submitted. Specifically, the work described in Chapter 3 and Chapter 4 is published in the Very Large Data Bases Journal (February 2016, VLDBJ) [69]. The work in Chapter 5 is to be submitted to the Very Large Data Bases Conference (VLDB 2017). Both AdPart and SPARTex have been demonstrated at the Very Large Data Bases Conference (VLDB 2015) [70, 71]. Figure 1.5 shows the roadmap.

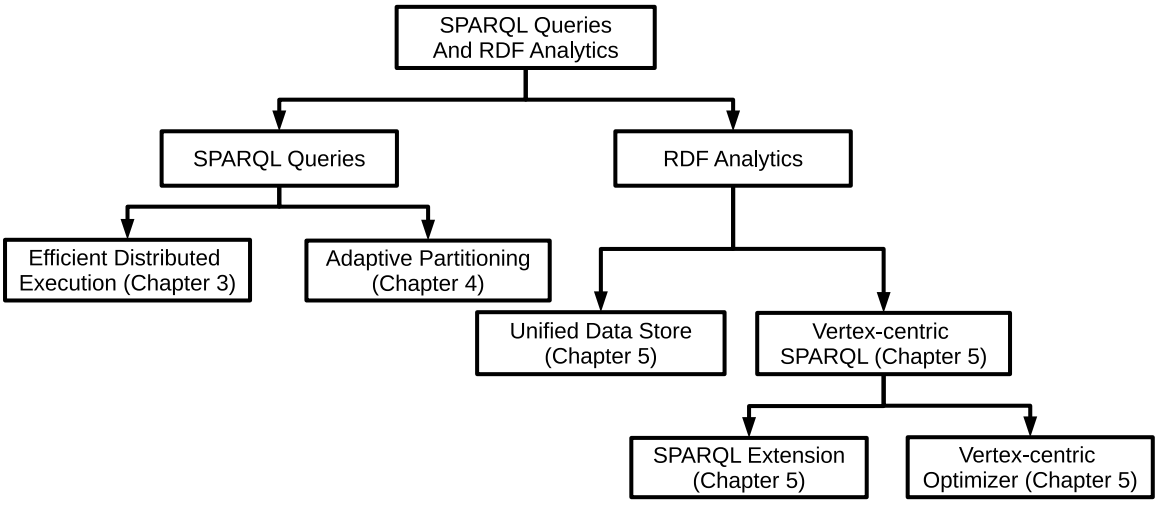


Figure 1.5: The roadmap of this thesis.

Chapter 2

Related Work

> Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.
>
> Albert Einstein
> 1879 — 1955 CE

This chapter sheds the light on recent distributed RDF systems related to the work of this thesis[1]. Specifically, this chapter focuses on the following key aspects of distributed RDF systems: (i) it discusses and classify the partitioning schemes employed by existing systems and (ii) how distributed query evaluation and optimization is carried under these partitioning settings (Section 2.1). Furthermore, (iii) this chapter includes a discussion on existing relational approaches related to data partitioning and query execution (Section 2.1.4). Finally, the chapter is concluded by a discussion on the generality of existing solutions for supporting rich RDF data analytics (Section 2.2).

2.1 Data Partitioning

This section reviews partitioning schemes used in recent distributed RDF systems. Current partitioning schemes can be generally categorized into 3 categories: (i) Lightweight Data Partitioning: this includes systems that use random, hash or range

[1]More details can be found in published surveys [72, 73, 74, 75].

Table 2.1: Summary of state-of-the-art distributed RDF systems

System	Partitioning Strategy	Partitioning Cost	Replication	Workload Awareness	Adaptive
TriAD-SG [32]	Graph-based (METIS) & Horizontal triple Sharding	High	Yes	No	No
H-RDF-3X [26]	Graph-based (METIS)	High	Yes	No	No
Partout [35]	Workload-based horizontal fragmentation	High	No	Yes	No
SHAPE [31]	Semantic Hash	High	Yes	No	No
S2RDF [76]	Extended Vertical Partitioning	High	No	No	No
Wu et al. [36]	End-to-end path partitioning	Moderate	Yes	No	No
TriAD [32]	Hash-based triple Sharding	Low	Yes	No	No
Trinity.RDF [27]	Hash	Low	Yes	No	No
HadoopRDF [33]	Vertical Partitioning/property Files on HDFS	Low	No	No	No
H2RDF+ [30]	H-Base partitioner (range)	Low	No	No	No
Rya [37]	Accumulo partitioner (range)	Low	No	No	No
CliqueSquare [38]	Hybrid (Hash + Vertical Partitioning)	Low	Yes	No	No
SHARD [28]	Hash (one big file)	Low	No	No	No
AdPart	Hash	Low	Yes	Yes	Yes

based partitioning. These systems incur minimal data preprocessing overhead; hence, referred to as lightweight. (ii) Sophisticated Partitioning: this includes systems that employ sophisticated heuristics for data preprocessing prior to data partitioning. These systems usually incur some replication to minimize communication during query evaluation. Finally, (iii) Workload-Aware Data Partitioning: workload-aware systems embark on some prior knowledge about the anticipated workloads which are used during the data partitioning phase. In the next few sections, systems are classified based on these categories. Table 2.1, summarizes the main characteristics of these systems.

2.1.1 Lightweight Data Partitioning

SHARD [28] is a horizontally scalable triple store engine built on top of MapReduce [47] framework. It stores RDF data into flat files using Hadoop Distributed File System (HDFS) [60]. The whole RDF data is kept into one file where each line represents all the triples of a certain subject. The file is then split by HDFS into blocks (usually 64MB in size) which are randomly distributed among machines. Although this storage model is simple, it leverages a set of benefits introduced by HDFS implementation. The RDF data is replicated and stored in a simple, easy to read format. The data is hash partitioned among the workers such that each worker is responsible for a set of triples.

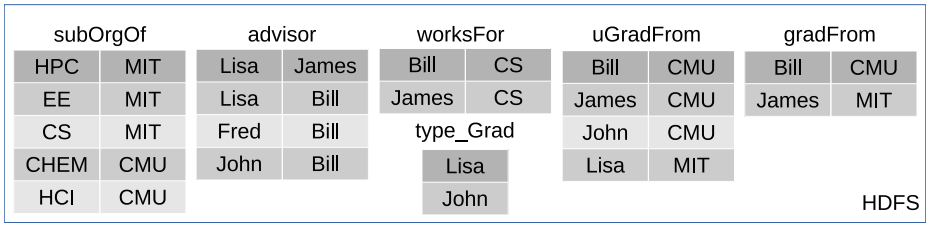


Figure 2.1: HadoopRDF store the data in Figure 1.3 on HDFS.

SHARD does not keep any form of data indices. As a result, while solving a query, it has to scan the whole list of triples even if query touches a very small amount of data. SPARQL queries are solved using a set of MapReduce [47] iterations equals to the number of joins in the query. Each MapReduce iteration is responsible for solving a single subquery and the results of each iteration are continuously joined with next iterations. A final MapReduce iteration is responsible for filtering the bound variables and remove redundant results.

HadoopRDF [33] uses HDFS flat files to store the RDF data. Similar to SHARD, data partitioning in HadoopRDF is left to HDFS; however, it groups triples using smaller files. Specifically, RDF data is split into multiple smaller files using two steps: Predicate Split (PS) and Predicate Object Split (POS). In the first step (PS), the data file is split based on the predicate into multiple smaller files where each file corresponds to a different predicate. This is similar to vertical partitioning strategy used by SW-Store [66]. For example, the RDF data in Figure 1.3 is stored by HadoopRDF on HDFS as shown in Figure 2.1. The POS step works on the explicit type information in the rdf:type file. It divides the rdf:type file into as many files as the number of distinct objects. Then, a set of files are created for each type_object pairs. For example, in Figure 2.1 only one file named type_Grad is created because there is only one object (Grad) associated with the type predicate. Other predicates files are also partitioned based on the object types into multiple files.

Upon receiving a SPARQL query from the user, it is passed to the Input Selector component which selects the files needed to execute the given query. Depending

on the complexity of the query, HadoopRDF uses one or multiple Hadoop [60] jobs to evaluate SPARQL queries. To minimize the overhead incurred by Hadoop jobs, HadoopRDF's cost model uses a heuristic to minimize the number of needed MapReduce jobs for solving a query. If the planner finds different query plans with the same number of jobs, it uses collected summary statistics to select the best join order that leads to the minimum intermediate results. The planner defines the query plan which represents an ordered set of MapReduce jobs, each associated with its input information. The framework then executes these jobs in order, such that the output file of each job is the input of the next one. The last job output is the answer of the given query.

CliqueSquare [38] partitioner exploits the fact that HDFS replicates data blocks to achieve fault-tolerance. Accordingly, it partitions the data by hashing on all three columns, i.e. it hashes triples based on their subject, predicate and object values. This enables CliqueSquare to perform all first-level joins in a query plan (subject-subject, subject-predicate...etc) locally in each compute node. Finally, it applies property-based grouping similar to the approach followed by HadoopRDF [33].

CliqueSquare presented a novel approach for optimizing BGP queries in a parallel environment, such as MapReduce. To reduce query response times, it builds flat plans where the number of joins is minimal. A query is represented as a variable graph where each node corresponds to a triple patterns and edges between nodes denote that triple patterns share a join variable. The CliqueSquare algorithm starts with the initial variable graph and keeps finding possible clique decompositions of the graph. Then, each decomposition is reduced until it consists of a single node from which CliqueSquare builds the corresponding logical query plan. CliqueSquare aims at finding the possible flattest plan to decrease response time. Logical query plans are translated into physical MapReduce operators which are then transformed into a MapReduce program. CliqueSquare uses a cost function that estimates the scanning,

joining, incurred I/O and data transfer costs.

H2RDF+ [30] is a highly scalable distributed RDF engine based on MapReduce [47] framework and Apache HBase [53]. H2RDF+ [30] materialize all six permutations of RDF triples using HBase tables which are sorted key-value maps. Data partitioning is left to HBase which range partitions tables based on keys. Maintaining these indices offers several benefits: (i) using a single scan on the corresponding index, all SPARQL triple patterns can be answered efficiently. (ii) Merge join can be employed to exploit the precomputed ordering in these indices. (iii) Every join between triple patterns can be done using merge joins.

H2RDF+ has a set of aggregated statistics used to estimate the selectivity of a triple pattern, join results and join cost. Its online query planner uses a greedy algorithm that decides at each execution step the join with the smallest cost. Two different join algorithms can be executed: multi-way merge join algorithm which is efficient over already sorted data. The other one is sort-merge join which is used to join unsorted intermediate results. Based on the query complexity, H2RDF+ adaptively decides whether to execute the query in a centralized or a distributed fashion. Simple queries are executed efficiently in a centralized fashion; while complex queries with large intermediate results are evaluated using a set of MapReduce jobs. H2RDF+ utilizes lazy materialization to minimize the size of the intermediate results. Similarly, Rya [37] uses a key-value store (Accumulo [77]) for RDF data storage which range-partitions the data based on keys such that the keys in each partition are sorted. However, when solving a SPARQL query, Rya executes the first subquery using range scan on the appropriate index; it then utilizes index lookups when joining with the next subqueries.

Trinity.RDF [27] is a distributed in-memory RDF engine that can handle web scale RDF data. It represents RDF data in its native graph form (i.e., using adjacency lists) and uses a key-value store (Trinity [48]) as the back-end store. The RDF graph

is partitioned using vertex id as hash key. This is equivalent to partitioning the data twice; first using subjects as hash keys and second using objects. Trinity.RDF uses graph exploration for SPARQL query evaluation and relies heavily on its underlying high-end InfiniBand interconnect. In every iteration, a single subquery is explored starting from valid bindings by all workers. This way, generation of redundant intermediate results is avoided. However, because exploration only involves two vertices (source and target), Trinity.RDF cannot prune invalid intermediate results without carrying all their historical bindings. Hence, workers need to ship candidate results to the master to finalize the results, which is a potential bottleneck of the system.

DREAM [78] follows a very different yet simple approach. DREAM does not partition the data, rather it builds one database and replicates it on all workers. Instead of running the query in parallel by all machines, DREAM decomposes the query into multiple (usually non-overlapping) subqueries. Each subquery is answered by one worker which has the entire database. Workers then exchange auxiliary metadata to finalize the query evaluation. While in principle DREAM does not incur any partitioning overhead, its preprocessing phase is very expensive because of the centralized database construction. Furthermore, parallelism in DREAM is limited by the number of query decompositions (usually very small).

All the above systems use lightweight partitioning schemes, which are computationally inexpensive; however, queries with long paths and complex structures incur high communication costs. In addition, systems that evaluate joins using MapReduce suffer from its high overhead [32, 36]. Although AdPart (introduced in this thesis) also uses lightweight hash partitioning, it avoids excessive data shuffling by exploiting hash-based data locality. Furthermore, it adapts incrementally to the workload to further minimize communication.

Table 2.2: Partitioning the graph in Figure 1.1 using 1-hop undirected guarantee. Replicated triples are highlighted.

W1		
subject	predicate	object
HPC	subOrgOf	MIT
EE	subOrgOf	MIT
CHEM	subOrgOf	CMU
James	worksFor	CS
James	uGradFrom	CMU
James	gradFrom	MIT
Lisa	uGradFrom	MIT
Lisa	type	Grad
Lisa	advisor	James
Lisa	advisor	Bill
Fred	advisor	Bill
John	type	Grad
CS	subOrgOf	MIT

W2		
subject	predicate	object
CS	subOrgOf	MIT
HCI	subOrgOf	CMU
Bill	worksFor	CS
Bill	gradFrom	CMU
Bill	uGradFrom	CMU
John	type	Grad
John	uGradFrom	CMU
John	advisor	Bill
CHEM	subOrgOf	CMU
James	uGradFrom	CMU
Lisa	advisor	Bill
James	worksFor	CS
Fred	advisor	Bill

2.1.2 Sophisticated Partitioning Schemes and Replication

Several systems employ general graph partitioning techniques for RDF data, in order to improve data locality.

H-RDF-3X [26] uses METIS [54] to partition the RDF graph by assigning each vertex to a single partition. For example, in Figure 1.1, each vertex is assigned to a single worker. METIS results in a balanced vertex partitioning where each partition has an equal share of vertices. Then, H-RDF-3X enforces the so-called k-hop guarantee where for any vertex v assigned to partition p all k-hop away vertices and their edges are replicated in p. This way any query with radius k or less can be executed without communication. For example, partitioning the graph in Figure 1.1 among two workers using 1-hop undirected guarantee yields the partitions shown in Table 2.2. Each partition is stored and managed by a standalone centralized RDF store like RDF-3X. Under this setting, any query with radius of 1 can be answered without communication. For example, the query in Figure 1.2 can be answered without communication among workers. Note that because of the replication, results duplication may occur.

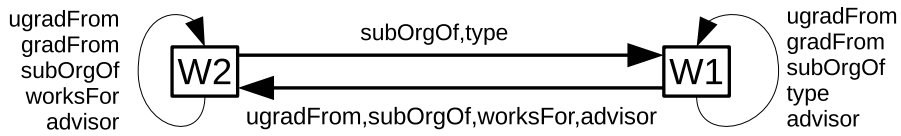


Figure 2.2: The summary graph for the data in Figure 1.1.

For example, the query Q =⟨?stud, advisor, Bill⟩ will return duplicate ⟨Lisa, advisor, Bill⟩ and ⟨Fred, advisor, Bill⟩; one from each partition. To solve this problem, H-RDF-3X introduces the notion of ownership triples that are created during the partitioning phase. For each vertex v assigned to partition p, H-RDF-3X stores a new triple ⟨v, is_owned, yes⟩ at partition p. Then, when evaluating a query, an extra join is carried out for filtering duplicate results. In our example, the query becomes Q =⟨?stud, advisor, Bill⟩ AND ⟨?stud, is_owned, yes⟩. Other queries with radius larger than k are executed using expensive MapReduce joins. Replication increases exponentially with k; thus, k must be small (e.g., $k \leq 2$ in [26]).

Similarly, EAGRE [79] transforms the RDF graph into a compressed entity graph that is partitioned using a MinCut algorithm, such as METIS. EAGRE and H-RDF-3X suffer from the overhead of MapReduce-based joins for queries that cannot be evaluated locally. For such queries, sub-second query evaluation is not feasible [32]. TriAD [32] employs lightweight hash partitioning based on both subjects and objects. Since partitioning information is encoded into the triples, TriAD has full locality awareness of the data and processes large number of concurrent joins without communication. However, because TriAD shards one (both) relations when evaluating distributed merge (hash) joins, the locality of its intermediate results is not preserved. Thus, if the sharding column of the previous join is not the current join column, intermediate results need to be re-sharded. The cost becomes significant for large intermediate results with multiple attributes.

TriAD-SG [32] uses METIS for data partitioning. Edges that cross partitions are replicated, resulting in $1-hop$ guarantee. A summary graph is defined, which includes

a vertex for each partition. Vertices in this graph are connected by the cross-partition edges. For example, Figure 2.2 shows the summary graph of the data in Figure 1.1. A query in TriAD-SG is evaluated against the summary graph first, in order to prune partitions that do not contribute to query results. Then, the query is evaluated on the RDF data residing in the partitions retrieved from the summary graph. Multiple join operators are executed concurrently by all workers, which communicate via an asynchronous message passing protocol.

Sophisticated partitioning techniques, like MinCut, reduce the communication cost significantly by minimizing the edge-cut. However, such techniques are prohibitively expensive and do not scale for large graphs, as shown in [31]. Furthermore, MinCut does not yield good partitioning for dense graphs. Thus, TriAD-SG does not benefit from the summary graph pruning technique in dense RDF graphs because of the high edge-cut. To alleviate METIS overhead, an efficient approach for partitioning large graphs was introduced [80]. However, queries that cross partition boundaries and hence result in poor performance cannot be eliminated.

SHAPE [31] is based on a semantic hash portioning approach for RDF data. It starts by simple hash partitioning and employs the same k-hop strategy as H-RDF-3X [26]. It also relies on URI hierarchy, for grouping vertices to increase data locality. Each resulting partition is manged by a standalone RDF-3X store. Similar to H-RDF-3X, SHAPE suffers from the high overhead of MapReduce-based joins. It also requires an extra join for filtering duplicate results. Furthermore, URI-based grouping results in skewed partitioning if a large percentage of vertices share prefixes. This behavior is noticed in both real as well as synthetic datasets (See Section 3.3).

Wu et al. [36] recently proposed an end-to-end path partitioning scheme, which considers all possible directed paths in the RDF graph. These paths are merged in a bottom-up fashion. While this approach works well for star, chain and directed cyclic queries, other types of queries result in significant communication. For example,

queries with object-object joins or queries that do not associate each query vertex with the type predicate require inter-worker communication. Note that our adaptivity technique (Chapter 4) is orthogonal to and can be combined with end-to-end path partitioning as well as other partitioning heuristics to efficiently evaluate queries that are not favored by the partitioning.

S2RDF [76] is a SPARQL engine built on top of Spark [62]. It proposes a relational partitioning technique for RDF data called Extended Vertical partitioning (ExtVP). ExtVP extends the vertical partitioning approach used by HadoopRDF [33] to minimize the size of input data during query evaluation. To do so, ExtVP uses semi-join reduction [81] to minimize data skewness and eliminate dangling triples that do not contribute to any join. For every two vertical partitions (see Figure 2.1), ExtVP computes join reductions for the two vertical partitions. The results are materialized as tables in HDFS. Specifically, for two partitions P_1 and P_2, S2RDF computes: (i) subject-subject: $P_1 \ltimes_{s=s} P_2$, $P_2 \ltimes_{s=s} P_1$, (ii) subject-object: $P_1 \ltimes_{s=o} P_2$, $P_2 \ltimes_{s=o} P_1$, and (iii) object-subject: $P_1 \ltimes_{o=s} P_2$, $P_2 \ltimes_{o=s} P_1$. The intuition behind this reduction is that a join between any two tables, say T_1 and T_2, can be computed using the semi-join reduced tables which are much smaller than the base tables. For example, $T_1 \bowtie_{A=B} T_2 = (T_1 \ltimes_{A=B} T_2) \bowtie (T_1 \rtimes_{A=B} T_2)$.

Systems that use sophisticated partitioning heuristics focus on minimizing communication irrespective of the workload. Hence, they pay the cost for data partitioning ahead of time even if the workload touches a very small part of the data. Moreover, these systems optimize their partitioning scheme to honor a specific type of queries. Other types of queries are neglected and usually introduce excessive communication that is detrimental to the performance. On the contrary, AdPart incurs a minimal partitioning overhead and dynamically adapts to the workload.

2.1.3 Workload-Aware Data Partitioning

Partout [35] is a workload-aware distributed RDF engine. It relies on a given workload to divide the data between nodes. First, it extracts representative triple patterns from the query load. Then, it uses these patterns to partition the data into fragments and collocates data fragments that are accessed together by queries in the same worker. Similarly, WARP [34] uses a representative query workload to replicate frequently accessed data. However, these system are static and do not adapt to the workload. Partout and WARP can adapt only by applying expensive re-partitioning of the entire data; otherwise, they incur high communication costs for dynamic workloads. On the contrary, AdPart adapts incrementally by replicating only the data accessed by the workload which is expected to be small [44].

Sedge [82] solves the problem of dynamic graph partitioning and demonstrates its partitioning effectiveness using SPARQL queries over RDF. The entire graph is replicated several times and each replica is partitioned differently. Every SPARQL query is translated manually into a Pregel [49] program and is executed against the replica that minimizes communication. Still, this approach incurs excessive replication, as it duplicates the entire data several times. Moreover, its lack of support for ad-hoc queries makes it counter-productive; a user needs to manually write an optimized query evaluation program in Pregel.

2.1.4 Related Solutions

This sections discusses solutions used by RDF engines and relational databases to minimize queries latencies.

Materialized views and Results Caching

Several works attempt to speed up the execution of SPARQL queries by materializing a set of views [83, 84] or a set of path expressions [85]. The selection of views

is based on a representative workload. Recomputing and materializing the views adaptively is a very expensive process that cannot be achieved in an online fashion [17]. On the other hand, AdPart does not generate local materialized views. Instead, it incrementally redistributes the data accessed by hot patterns in a way that preserves data locality and allows queries to be executed with minimal communication.

Other works embark on final result caching. Martin et al. [86] utilize a relational database for storing meta information about cached queries. When a cached query is submitted to the system again, the cached result is returned immediately. However, the proposed solution is very sensitive to slight changes in the queries [87]. For example, the framework cannot match two exactly similar queries with different subqueries orderings or different variables names. Papailiou et al. [87] rectifies this problem by introducing a canonical labeling technique for SPARQL queries. Same queries with different orderings or variables names will result in the same canonical label. The canonical label is used as a key for the cached result. Moreover, this framework uses a generic query abstraction to proactively cache extra results that can be used by queries with the same structure but different constants. While caching is orthogonal and complementary to the work presented in this thesis, there is a major difference. Caching is useful for exactly matching queries, queries whose result is contained within cached results or queries that can be answered by joining the results of multiple cached queries; other queries would require expensive distributed evaluation. On the contrary, the distributed query evaluation proposed by this thesis aim at efficiently evaluating all queries posed to the system. Furthermore, the adaptivity feature allows the efficient execution of exactly matching, isomorphically different or even structurally different queries.

Eventual Indexing And Adaptive Physical Layout

Idreos et al. [88] and Alagiannis et al. [89, 90] introduced the concept of reducing the data-to-query time for relational data. They avoid building indices during data loading; instead, they reorder tuples incrementally during query processing. In AdPart, the concept of eventual indexing is extended to dynamic and adaptive graph partitioning. In this thesis, the graph partitioning problem is very expensive; hence, the potential benefits of minimizing the data-to-query time are substantial.

To address the problem of workload dynamism and diversity, Aluç [17] proposed an approach for adaptively changing the physical layout based on workloads. He introduces a Workload-aware group-by-query (G-by-Q) representation where the content of each database record and the way it is serialized is dynamically determined based on the workload. G-by-Q approach aims at scaling-up SPARQL query evaluation by creating millions of G-by-Q. On the other hand, AdPart tries to minimize replication (see Chapter 4) by finding shared commonalities among redistributed patterns in the workload. In an ideal solution, an adaptive physical layout technique can be used on each compute node. At the same time, a dynamic incremental redistribution technique similar to the one introduced in this thesis can be used to decide data placement [17].

Relational Model

Relevant systems exist that focus on data models other than RDF. Schism [91] deals with data placement for distributed OLTP RDBMS. Using a sample workload, Schism minimizes the number of distributed transactions by populating a graph of co-accessed tuples. Tuples accessed in the same transaction are put in the same server. Similarly, SWORD [92, 93] models the workload as a hypergraph, where each hyperedge corresponds to a transaction and uses METIS to partition this hypergraph and guide data placement decisions. To minimize the overhead of the partitioning phase, SWORD

uses a hypergraph compression technique. To deal with dynamic workloads, SWORD uses an incremental approach for dynamic data repartitioning. Similar to Schism, SWORD aims at minimizing the number of distributed transactions. This is not appropriate for SPARQL because some queries access large parts of the data that would overload a single machine. Instead, AdPart exploits parallelism by executing such a query across all machines in parallel without communication. H-Store [94] is an in-memory distributed RDBMS that uses a data partitioning technique similar to ours. Nevertheless, H-Store assumes that the schema and the query workload are given in advance and assumes no ad-hoc queries. Although, these are valid assumptions for OLTP databases, they are not for RDF data stores.

ElasTras [95, 96] is a distributed system that provides transactional support to partitioned databases. It statically partitions data at the schema level by co-locating tuples that are accessed together in the same partition. ElasTras partitions data based on the primary key of a single root table. Descendant tables, that have the primary key of the root table as a foreign key, are partitioned using the foreign key value. The process repeats recursively on subsequent levels; effectively resulting in a tree schema. Accordingly, data can be organized in row groups where data of each row group is stored in a single partition. While limiting distributed transactions, this approach cannot be applied on RDF for many reasons: (i) The schemaless nature of RDF makes the process of defining root-descendant relationships among Tables[2] infeasible in most of the cases. Furthermore, (ii) ElasTras mandates databases to conform to the tree schema; an assumption that does not apply on graph data like RDF. More importantly, (iii) partitions in ElasTras are created statically and does not change if the access pattern changes. G-Store [97], on the other hand, employs the Key Group abstraction to minimize the overhead of distributed transactions. in G-Store, no data migration is needed, rather, the ownership of keys in a key group

[2]Assuming that each predicate defines a table like in SW-Store [66]

are co-located in a single partition. This approach is not suitable for RDF because a single query can touch large amount of data. In this case, managing the data in a single partition can be overwhelming.

2.2 System Generality

This section analyzes and reviews the generality of existing distributed RDF systems.

2.2.1 SPARQL on Generic Frameworks

Several distributed RDF systems [28, 33, 30, 38] are built on top of MapReduce [47]. While the underlying framework is capable of performing graph analytics, these systems are optimized for solving SPARQL only. Specifically, these systems model the data in a way suitable for join evaluation not for graph analysis. For example, H2RDF+ [30] is based on MapReduce and HBase indices. It is optimized for multi-way merge joins and not for iterative graph computations. The same applies to S2RDF [76], CliqueSquare [38] and HadoopRDF [33] which rely on vertical partitioning to boost join evaluation. PigSPARQL [98] transforms each SPARQL query into a PigLatin [98] program that is executed using MapReduce which is not suitable for iterative graph algorithms [49]. MapReduce requires passing the entire graph state from one iteration to the next.

2.2.2 SPARQL on Graph Frameworks

Many graph management systems have been proposed for efficient graph analytics, including Pregel [49], PowerGraph [50], GRACE [52], and SociaLite [99]. However, these systems lack the capability of evaluating ad-hoc SPARQL queries, which means that a program has to be written for each SPARQL query. Sedge [82] focus on dynamic RDF partitioning. Goodman et al. [100] proposed a vertex-centric program for solving SPARQL queries using GraphLab [51] framework. Both approaches [82, 100] do not

have a query optimizer and therefore users have to select a good query evaluation plan themselves. Such an approach is tedious, counter-productive and requires prior knowledge about the data. Notice that an unoptimized query can take significant time or cause the system to run out of memory or crashes (see Section 5.3.2).

Trinity.RDF [27] is a SPARQL engine built on top of Trinity [48]. However, while it uses a native graph format, it only focus on SPARQL query evaluation. Trinity.RDF cannot support rich RDF analytics as it does not pipeline SPARQL and other operators. To perform analytics users need to use Trinity Specification Language (TSL) for data modeling. Moreover, without the SPARQL extension proposed in this thesis (see Chapter 5), Trinity.RDF cannot declaratively execute analytical tasks.

GraphX [63, 64] is a graph processing system built on top of a Spark [61, 62]; a general purpose distributed data flow framework. GraphX aims at unifying graph-parallel (e.g. vertex-centric) and data-parallel computations (e.g., map-reduce) in a single system. The same data can be viewed as both tables and graphs which allows both types of computations to be applied. As a result of the generality and unification of GraphX, it is not as fast as specialized graph engines [63, 64]. Yet, it allows users to stay within a single framework and remove the burden of moving data between systems and format it accordingly. SPARTex is inspired by the same motivation which tries to unify both SPARQL structural querying and generic graph computations.

2.2.3 Rich RDF Analytics

Deweese et al. [58] proposed an implementation of the peer pressure clustering algorithm using SPARQL. Qi et al. [57] introduced distributed remote clustering algorithms that minimize the communication overhead. Both approaches, however, focus on clustering and cannot support generic graph analytics. Techentin et al. [56] exploit the update capability of SPARQL 1.1 for implementing iterative algorithms. How-

ever, this approach results in lengthy and verbose SPARQL queries that are hard to evaluate and understand. uRiKA [59] allows the invocation of few predefined graph algorithms that were requested by users. These algorithms are tailored and optimized for the uRiKA appliance by expert researchers; users cannot add any new algorithm or modify any existing one. In contrast, SPARTex allows users to implement any algorithm using simple vertex-centric API's.

Blazegraph [101] is a commercial, specialized, high-performance graph database with support for the Blueprints and RDF/SPARQL APIs. Recently, Blazegraph announced the release of their RDF GAS API which enables rich RDF analytical and mining tasks. Since it is a specialized product, the entire graph analytics software stack had to be implemented. SPARTex, on the other hand, takes a different approach by supporting rich RDF analytics on top of distributed graph system. Blazegraph uses the Gather Apply Scatter (GAS) [51] model for implementing data mining algorithms which can be invoked by SPARQL end points as services. However, due to the 1-D partitioning scheme in their distributed version, the solution is limited to a single machine and does not scale-out efficiently [102].

2.2.4 Related Solutions

Gao et al. [103] introduce a system for continuous approximate pattern detection over evolving graphs. Fard et al. [104] propose a vertex-centric approach for graph simulation on massive graphs. Graph simulation provides a practical alternative to subgraph isomorphism, which is an NP-Complete problem, by relaxing its stringent matching conditions. This allows matches to be found in polynomial time. These solutions are approximate, while SPARQL requires exact subgraph pattern matching. Horton+ [105] solves reachability queries over large attributed multi-graphs. It introduces a declarative query language with a compiler and its own query optimizer. Horton+ solves reachability queries which consider only paths with closures

and cannot solve generic SPARQL queries with complex structure or cycles. Moreover, unlike reachability queries, SPARQL allows variable vertices that can match any node in the data. Motivated by the fact that graph data are usually stored in relational databases and users tend to apply SQL as well as graph algorithms on their data, Jindal et al. introduce Vertexica [106] a relational databases system capable of performing graph analytics. Vertexica does not focus on RDF; hence, SPARQL queries are not supported.

Chapter 3

Exploiting Hash-based Locality

> Everything is theoretically
> impossible, until it is done.
>
> Robert A. Heinlein
> 1907 — 1988 CE

AdPart advocates the reliance on workload guided partitioning, where data is incrementally and dynamically repartitioned based on the workload. However, at any moment in time, there might be queries that are not favored by the current distribution. While AdPart will eventually adapt to them, these queries need to be executed efficiently; otherwise, the whole system performance will degrade. Therefore, AdPart introduces an efficient baseline for distributed SPARQL query evaluation. AdPart exploits the hash-based data locality to execute queries comparable or faster than state-of-the-art distributed RDF systems. This chapter, discusses the system architecture of AdPart and its distributed SPARQL query execution approach. Adaptive redistribution is introduced in the next chapter.

3.1 System Architecture

Overview: AdPart employs the typical master-slave paradigm and is deployed on a shared-nothing cluster of machines (see Figure 3.1). This architecture is used by other systems, e.g., Trinity.RDF [27] and TriAD [32]. AdPart uses the standard Message Passing Interface (MPI) [107] for master-worker communication. In a nutshell, the

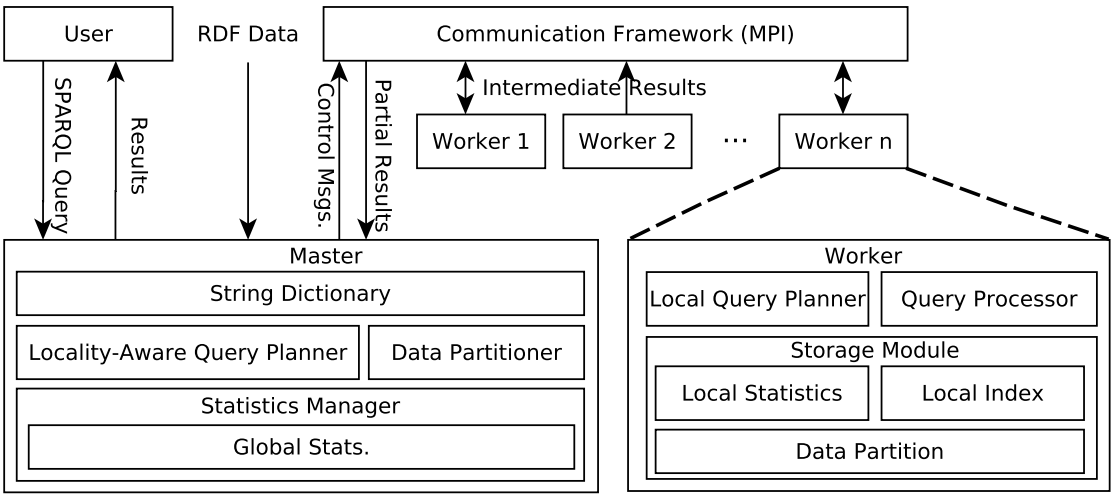


Figure 3.1: System architecture of AdPart

master begins by encoding the data and partitioning it among workers. Each worker loads its triples and collects local statistics. Then, the master aggregates these statistics and becomes ready for answering queries. Each query is submitted to the master, which decides whether the query can be executed in parallel or distributed mode. In parallel mode, the query is evaluated concurrently by all workers without communication. Queries in distributed mode are also evaluated by all workers but require communication.

3.1.1 Master

String Dictionary. RDF data contain long strings in the form of URIs and literals. To avoid the storage, processing, and communication overheads, we follow the common practice [19, 27, 30, 32] and encode RDF strings into numerical IDs and build a bi-directional dictionary.

Data Partitioner. A recent study [40] showed that 60% of the joins in a real workload of SPARQL queries are on the subject column. Hence, AdPart uses lightweight node-based partitioning using subject values. Given W workers, a triple t is assigned to

worker w_i, where i is the result of a hash function applied on $t.subject$. [1] This way all triples that share the same subject go to the same worker. Consequently, any star query joining on subjects can be evaluated without communication cost. AdPart does not hash on objects because they can be literals and common types; this would assign all triples of the same type to one worker, resulting in load imbalance and limited parallelism [26].

Statistics Manager. It maintains statistics about the RDF graph, which are used for global query planning and during adaptivity. Statistics are collected in a distributed manner during bootstrapping.

Locality-Aware Query Planner. Our planner uses the global statistics from the statistics manager and the pattern index from the redistribution controller to decide if a query, in whole or partially, can be processed without communication. Queries that can be fully answered without communication are planned and executed by each worker independently. On the other hand, for queries that require communication, the planner exploits the hash-based data locality and the query structure to find a plan that minimizes communication and the number of distributed joins (Section 3.2).

Failure Recovery. The master does not store any data but can be considered as a single-point of failure because it maintains the dictionaries, global statistics, and PI. A standard failure recovery mechanism (log-based recovery [108]) can be employed by AdPart. Assuming stable storage, the master can recover by loading the dictionaries and global statistics because they are read-only and do not change in the system. The PI can be recovered by reading the query log and reconstructing the heat map. Workers on the other hand store data; hence, in case of a failure, data partitions need to be recovered. Shen et al. [109] proposes a fast failure recovery solution for distributed graph processing systems. The solution is a hybrid of checkpoint-based and log-based recovery schemes. This approach can be used by AdPart to recover

[1] For simplicity, we use: $i = t.subject \mod W$.

worker partitions and reconstruct the replica index. Reliability is outside the scope of this thesis and we leave it for future work.

3.1.2 Worker

Storage Module. Each worker w_i stores its local set of triples D_i in an in-memory data structure, which supports the following search operations, where s, p, and o are subject, predicate, and object, respectively:

1. given p, return set $\{(s, o) \mid \langle s, p, o\rangle \in D_i\}$.
2. given s and p, return set $\{o \mid \langle s, p, o\rangle \in D_i\}$.
3. given o and p, return set $\{s \mid \langle s, p, o\rangle \in D_i\}$.

Since all the above searches require a known predicate, we primarily hash triples in each worker by predicate. The resulting predicate index (simply P-index) immediately supports search by predicate (i.e., the first operation). Furthermore, we use two hash maps to re-partition each bucket of triples having the same predicate, based on their subjects and objects, respectively. These two hash maps support the second and third search operation and they are called predicate-subject index (PS-index) and predicate-object index (PO-index), respectively. Given the number of unique predicates is typically small, our storage scheme avoids unnecessary repetitions of predicate values. Note that when answering a query, if the predicate itself is a variable, then we simply iterate over all predicates. Our indexing scheme is tailored for typical RDF knowledge bases and their workloads. Being orthogonal to the rest of the system, alternative schemes, like indexing all SPO combinations [19, 22], could be used at each worker). Finally, the storage module computes statistics about its local data and shares them with the master after data loading.

Query Processor. Each worker has a query processor that operates in two modes: (i) Distributed Mode for queries that require communication. In this case, the locality-

aware planner of the master devises a global query plan. Each worker gets a copy of this plan and evaluates the query accordingly. Workers solve the query concurrently and exchange intermediate results (Section 3.2.1). (ii) Parallel Mode for queries that can be answered without communication. In this case, the master broadcasts the query to all workers. Each worker has all the data needed for query evaluation; therefore it generates a local query plan using its local statistics and executes the query without communication.

Local Query Planner. Queries executed in parallel mode are planned by workers autonomously. For example, star queries joining on the subject are processed in parallel due to the initial partitioning.

3.2 Query Evaluation

A basic SPARQL query consists of multiple subquery triple patterns: $q_1, q_2, \ldots, q_n$. Each subquery includes variables or constants, some of which are used to bind the patterns together, forming the entire query graph (e.g., see Figure 1.2(b)). A query with n subqueries requires the evaluation of $n - 1$ joins. Since data are memory resident and hash-indexed, we favor hash joins as they prove to be competitive to more sophisticated join methods [110]. Our query planner devises an ordering of these subqueries and generates a left-deep join tree, where the right operand of each join is a base subquery (not an intermediate result). We do not use bushy tree plans to avoid building indices for intermediate results.

3.2.1 Distributed Query Evaluation

In AdPart, triples are hash partitioned among many workers based on subject values. Consequently, subject star queries (i.e., all subqueries join on the subject column) can be evaluated locally in parallel without communication. However, for other types of queries, workers may have to communicate intermediate results during join evaluation.

Table 3.1: Matching result of q_1 on workers w_1 and w_2.

w_1

?prof
James

w_2

?prof
Bill

Table 3.2: The final query results $q_1 \bowtie q_2$ on both workers.

w_1

?prof	?stud
James	Lisa

w_2

?prof	?stud
Bill	Lisa
Bill	John
Bill	Fred

For example, consider the query in Figure 1.2 and the partitioned data graph in Figure 1.3. The query consists of two subqueries q_1 and q_2, where:

- q_1: ⟨?prof, worksFor, CS⟩
- q_2: ⟨?stud, advisor, ?prof⟩

The query is evaluated by a single subject-object join. However, neither of the workers has all the data needed for evaluating the entire query; thus, workers need to communicate. For such queries, AdPart employs the Distributed Semi-Join (DSJ) algorithm. Each worker scans the PO-index to find all triples matching q_1. The results on workers w_1 and w_2 are shown in Table 3.1. Then, each worker creates a projection on the join column $?prof$ and exchanges it with the other worker. Once the projected column is received, each worker computes the semi-join $q_1 \rtimes_{?prof} q_2$ using its PO-index. Specifically, w_1 probes $p = \texttt{advisor}, o = \texttt{Bill}$ while w_2 probes $p = \texttt{advisor}, o = \texttt{James}$ to their PO-index. Note that workers also need to evaluate semi-joins using their local projected column. Then, the semi-join results are shipped to the sender. In this case, w_1 sends ⟨Lisa, advisor, Bill⟩ and ⟨Fred, advisor, Bill⟩ to w_2; no candidate triples are sent from w_2 because `James` has no advisees on w_2. Finally, each worker computes the final join $q_1 \bowtie_{?prof} q_2$. The final query results at both workers are shown in Table 3.2.

Table 3.3: The final query results $q_2 \bowtie q_1$ on both workers.

w_1

?prof	?stud
James	Lisa
Bill	Lisa
Bill	Fred

w_2

?prof	?stud
Bill	John

Hash-based data locality

Observation **1.** DSJ can benefit from subject hash locality to minimize communication. If the join column of the right operand is subject, the projected column of the left operand is hash distributed by all workers, instead of being broadcast to every worker.

In our example, since the join column of q_2 is the object column ($?prof$), each worker sends the entire join column to the other worker. However, based on Observation 1, communication can be minimized if the join order is reversed (i.e., $q_2 \bowtie q_1$). In this case, each worker scans the P-index to find triples matching q_2 and creates a projection on $?prof$. Then, because $?prof$ is the subject of q_1, both workers exploit the subject hash-based locality by partitioning the projection column and communicating each partition to the respective worker, as opposed to broadcasting the entire projection column to all workers. Consequently, w_1 sends `Bill` to only w_2 because of Bill's hash value. The final query results are shown in Table 3.3. Notice that the final results are the same for both query plans; however, the results reported by each worker are different.

Pinned subject

Observation **2.** Under the subject hash partitioning, combining right-deep tree planning and the DSJ algorithm, causes the intermediate and final results to be local to the subject of the first executed subquery pattern p_1. We refer to this subject as pinned_subject.

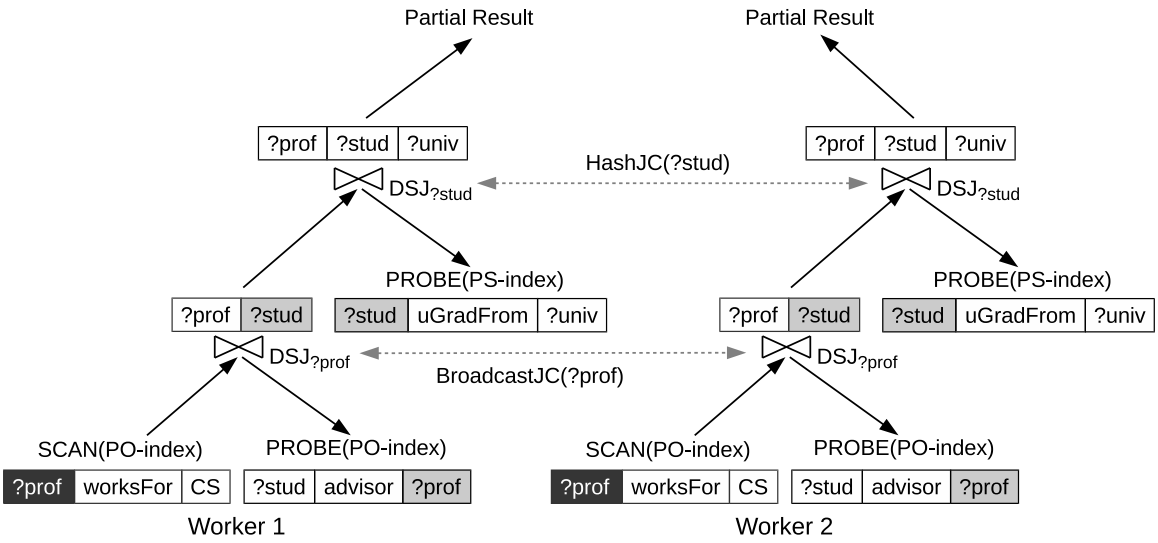


Figure 3.2: Executing query Q_{prof} in the following order: q_1, q_2, q_3

In our example, executing q_1 first causes $?prof$ to be the pinned_subject because it is the subject of q_1. Hence, the intermediate and final results are local (pinned) to the bindings of $?prof$, James and Bill in w_1 and w_2, respectively. Changing the order by executing q_2 first made $?stud$ to be the pinned_subject. Accordingly, the results are pinned at the bindings of $?stud$.

AdPart leverages Observations 1 and 2 to minimize communication and synchronization overhead. To see this, consider Q_{prof} which extends the query in Figure 1.2 with one more triple pattern, namely q_3: ⟨?stud, uGradFrom, ?univ⟩. Assume Q_{prof} is executed in the following order: q_1, q_2, q_3. The query execution plan is pictorially shown in Figure 3.2. The results of the first join (i.e., $q_1 \bowtie q_2$) are shown in Table 3.2 ($?prof$ is the pinned_subject). The query continues by joining the results of $(q_1 \bowtie q_2)$ with q_3 on $?stud$, the subject of q_3. Both workers project the intermediate results on $?stud$ and hash distribute the bindings of $?stud$ (Observation 1). Then, all workers evaluate semi-joins with q_3 and return the candidate triples to the other workers where the final query results are formulated.

Notice that the execution order q_1, q_2, q_3 requires communication for evaluating both joins. A better ordering is q_2, q_1, q_3. The execution plan is shown in Figure 3.3.

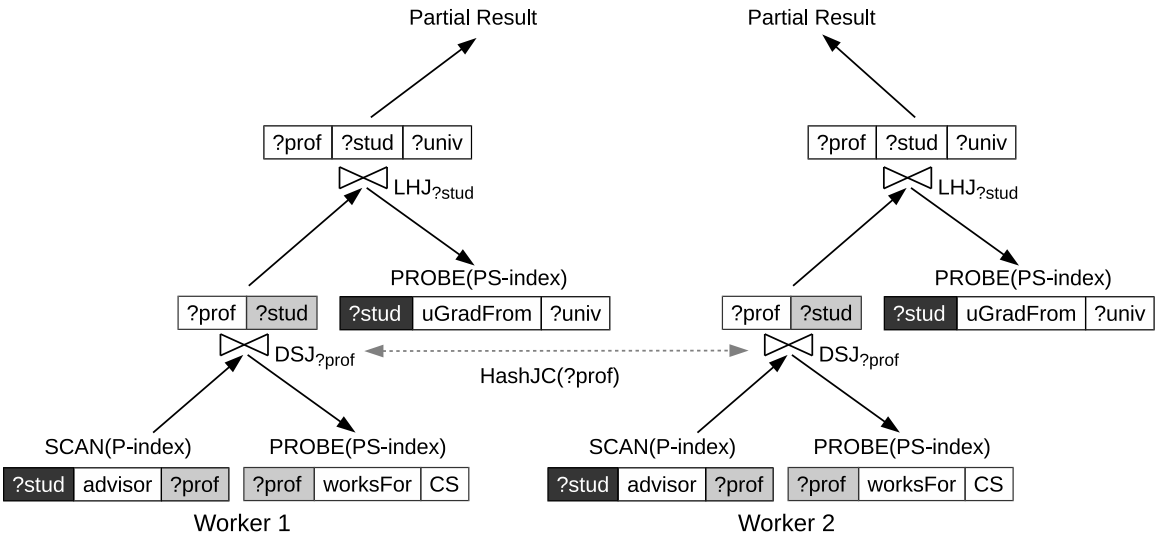


Figure 3.3: Executing query Q_{prof} in the following order: q_2, q_1, q_3

Table 3.4: Communication cost for different join types

Subject Pinning	**SS**	**SO/OO**	**OS**
Pinned	No Communication	Broadcast	Direct Communication
Unpinned	Direct Communication	Broadcast	Direct Communication

The first join (i.e., $q_2 \bowtie q_1$) already proved to incur less communication by avoiding broadcasting the entire projection column. The result of this join is pinned at ?*stud* as shown in Table 3.3. Since the join column of q_3 (?*stud*) is the *pinned_subject*, joining ($q_2 \bowtie q_1$) with q_3 can be processed by each worker without communication using Local Hash Join (LHJ). Therefore, the ordering of the subqueries affects the amount of communication incurred during query execution.

The four cases of a join

Formally, joining two subqueries, say p_i (possibly an intermediate pattern) and p_j, has four possible scenarios: the first three assume that p_i and p_j join on columns c_1 and c_2, respectively. (i) If $c_2 = subject$ AND $c_2 = pinned\_subject$, then the join is processed in parallel without communication. (ii) If $c_2 = subject$ AND $c_2 \neq pinned\_subject$, then the join is evaluated using DSJ, but the projected join column of p_i is hash distributed.

(iii) If $c_2 \neq$ *subject*, then the join is executed using DSJ and the projected join column of p_i is sent from all workers to all other workers. Finally, (iv) if p_i and p_j join on multiple columns, we opt to join on the subject column of p_j, if it is a join attribute. This allows the join column of p_i to be hash distributed as in *(ii)*. If the subject column of p_j is not a join attribute, the projection column is broadcast to all workers, as in *(iii)*. Verifying on the other columns is carried out during the join finalization. Table 3.4 summarizes these scenarios.

Based on the above four scenarios, we introduce our Locality-Aware Distributed Query Execution algorithm (see Algorithm 1). The algorithm receives an ordering of the subquery patterns. For each join iteration, if the second subquery joins on the pinned subject, the join is executed without communication (line 7). Otherwise, the join is evaluated by the DSJ algorithm (lines 8-28). In the first iteration, p_1 is a base subquery pattern; however, for the subsequent iterations, p_1 is a pattern of intermediate results. If p_1 is the first subquery to be matched, each worker finds the local matching of p_1 (line 10) and projects on the join column c_1 (line 13). If the join column of q_2 is subject, then each worker hash distributes the projected column (line 15); or sends it to all other workers otherwise (line 17). To avoid the overhead of synchronization, communication is carried out using non-blocking MPI routines. All workers perform semi-join on the received data (line 22) and send the results back to w (line 23). Finally, each worker finalizes the join (line 27) and formulates the final result (line 28). Lines 22 and 27 are implemented as local hash-joins using the local

index in each worker. The result of a DSJ iteration becomes p_1 in the next iteration.

Input: Query Q with n ordered subqueries $\{q_1, q_2, \ldots q_n\}$

Result: Answer of Q

```
1  p1 ← q1;
2  pinned_subject ← p1.subject;
3  for i ← 2 to n do
4      p2 ← qi;
5      [c1, c2]← getJoinColumns(p1, p2);
6      if c2 == pinned_subject AND c2 is subject then
7          p1 ← JoinWithoutCommunication (p1, p2, c1, c2);
8      else
9          if p1 NOT intermediate pattern then
10             RS1 ← answerSubquery(p1);
11         else
12             RS1 is the result of the previous join
13         RS1[c1] ← πc1(RS1); // projection on c1
14         if c2 is subject then
15             Hash RS1[c1] among workers;
16         else
17             Send RS1[c1] to all workers;
18         Let RS2 ← answerSubquery(p2);
19         foreach worker w, w : 1 → N do
20             // RS1w[c1] is the RS1[c1] received from w
21             // CRS2w are candidate triples of RS2 that join with RS1w[c1]
22             CRS2w ← RS1w[c1] ⋊RS1w[c1].c1=RS2.c2 RS2;
23             Send CRS2w to worker w;
24         foreach worker w, w : 1 → N do
25             // RS2w is the CRS2w received from worker w
26             // RESw is the result of joining with worker w
27             RESw ← RS1 ⋈RS1.c1=RS2w.c2 RS2w;
28         p1 ← RES1 ∪ RES2 ∪ .... ∪ RESN;
```

Algorithm 1: Locality-Aware Distributed Execution

Table 3.5: Triples matching ⟨?s, advisor, ?p⟩ and ⟨?s, uGradFrom, ?u⟩ on two workers.

Worker 1

advisor	?s	?p
	Fred	Bill
	Lisa	Bill
	Lisa	James

uGradFrom	?s	?u
	Lisa	MIT
	James	CMU

Worker 2

advisor	?s	?p
	John	Bill

uGradFrom	?s	?u
	Bill	CMU
	John	CMU

Algorithm 1 can solve star queries that join on the subject in parallel mode. Traditionally, the planning is done by the master using global statistics. We argue that allowing each worker to plan the query execution autonomously would result in a better performance. For example, using the data graph in Figure 1.3, Table 3.5 shows triples that match the following star query:

- q_1: ⟨?s, advisor, ?p⟩
- q_2: ⟨?s, uGradFrom, ?u⟩

Any global plan (i.e., $q_1 \bowtie q_2$ or $q_2 \bowtie q_1$) would require a total of four index lookups to solve the join. However, w_1 and w_2 can evaluate the join using 2 and 1 index lookup(s), respectively. Therefore, to solve such queries, the master sends the query to all workers; each worker utilizes its local statistics to formulate the execution plan, evaluates the query locally without communication, and sends the final result to the master.

3.2.2 Locality-Aware Query Optimization

Our locality-aware planner leverages the query structure and hash-based data distribution during query plan generation to minimize communication. Accordingly, the planner uses a cost-based optimizer, based on Dynamic Programming (DP), for finding the best subquery ordering. The same approach is used by other systems

[32, 19, 27]. Each state S in DP is identified by a connected subgraph ϱ of the query graph. A state can be reached by different orderings on ϱ. Thus, we maintain in each state the ordering that results in the least estimated communication cost ($S.cost$). We also keep estimated cardinalities of the variables in the query. Furthermore, instead of maintaining the cardinality of the state, we keep the cumulative cardinality of all intermediate results that led to this state. Although the cardinality of the state will be the same regardless of the ordering, different orderings result in different cumulative cardinalities.

We initialize a state S for each subquery pattern (subgraph of size 1) p_i. $S.cost$ is initially zero because a query with a single pattern can be answered without communication. Then, we expand the subgraph by joining with another pattern p_j, leading to a new state S' such that:

$$S'.cost = min(S'.cost,\ S.cost + cost(S, p_j))$$

If we reach a state using different orderings with the same cost, we keep the one with the least cumulative cardinality. This happens for subqueries that join on the $pinned\_subject$. To minimize the DP table size, we maintain a global minimum cost ($minC$) of all found plans. Because our cost function is monotonically increasing, any branch that results in a cost $> minC$ is pruned. Moreover, because of Observation 1, we start the DP process by considering subqueries connected to the subject with the highest number of outgoing edges; this increases the chances for converging to the optimal plan faster. The space complexity of the DP table is $O(s)$ where s is the number of connected subgraphs in the query graph. Since each state can be extended by multiple edges, the number of updates applied to the DP table (i.e., the time complexity) is $O(sE)$, where E is the number of edges in the query graph.

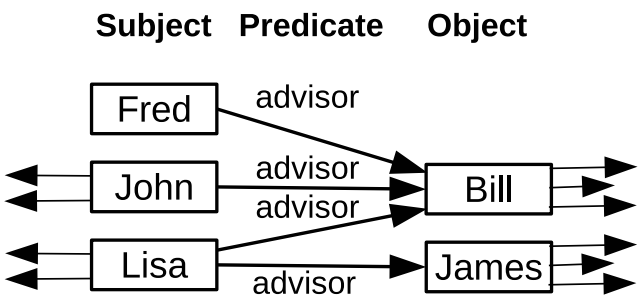


Figure 3.4: Statistics calculation for p=*advisor*, based on Figure 1.3.

3.2.3 Cost Estimation

We first describe the statistics used for cost calculation. Recall that AdPart collects and aggregates statistics from workers for global query planning and during the adaptivity process. Keeping statistics about each vertex in the RDF data graph is too expensive. Therefore, we focus on predicates rather than vertices; this way the storage complexity of statistics is linear to the number of unique predicates, which is typically small compared to the data size. For each unique predicate p, we calculate the following: (i) The cardinality of p, denoted as $|p|$, is the number of triples in the data graph that have p as predicate. (ii) $|p.s|$ and $|p.o|$ are the numbers of unique subjects and objects using predicate p, respectively. (iii) The subject score of p, denoted as $\overline{p_S}$, is the average degree of all vertices s, such that $\langle s, p, ?x\rangle \in D$. (iv) The object score of p, denoted as $\overline{p_O}$, is the average degree of all vertices o, such that $\langle ?x, p, o\rangle \in D$. (v) Predicates Per Subject $P_{ps} = |p|/|p.s|$ is the average number of triples with predicate p per unique subject. (vi) Predicates Per Object $P_{po} = |p|/|p.o|$ is the average number of triples with predicate p per unique object.

For example, Figure 3.4 illustrates the computed statistics for predicate advisor using the data graph of Figure 1.3. Since advisor appears four times with three unique subjects and two unique objects, $|p| = 4$, $|p.s| = 3$ and $|p.o| = 2$. The subject score $\overline{p_S}$ is $(1+3+4)/3 = 2.67$ because advisor appears with four unique subjects: Fred, John and Lisa, whose degrees (i.e., in-degree plus out-degree) are 1, 3 and 4, respectively.

Similarly, $\overline{p_O} = (6 + 4)/2 = 5$. Finally, the number of predicates per subject P_{ps} is $4/3 = 1.3$ because Lisa is associated with two instances of the predicate (i.e., two advisors).

We set the initial communication cost of DP states to zero. Cardinalities of subqueries with variable subjects and objects are already captured in the master's global statistics. Hence, we set the cumulative cardinalities of the initial states to the cardinalities of the subqueries and set the size of the subject and object bindings to $|p.s|$ and $|p.o|$. Furthermore, the master consults the workers to update the cardinalities of subquery patterns that are attached to constants or have unbounded predicates. This is done locally at each worker by simple lookups to its PS- and PO- indices to update the cardinalities of variables bindings accordingly.

We estimate the cost of expanding a state S with a subquery p_j, where c_j and P are the join column and the predicate of p_j, respectively. If the join does not incur communication, the cost of the new state S' is zero. Otherwise, the expansion is carried out through DSJ and we incur two phases of communication: (i) transmitting the projected join column and (ii) replying with the candidate triples. Estimating the communication in the first phase depends on the cardinality of the join column bindings in S, denoted as $B(c_j)$. In the second phase, communication depends on the selectivity of the semi-join and the number of variables ν in p_j (constants are not communicated). Moreover, if c_j is the subject column of p_j, we hash distribute the projected column. Otherwise, the column needs to be sent to all workers. The cost of expanding S with p_j is:

$$cost(S, p_j) = \begin{cases} 0, \text{if } c_j \text{ is subject \& } c_j = pinned\_subject \\ \\ S.B(c_j) + (\nu \cdot S.B(c_j) \cdot P_{ps}), \\ \text{if } c_j \text{ is subject \& } c_j \neq pinned\_subject \\ \\ (S.B(c_j) \cdot N) + (\nu \cdot N \cdot S.B(c_j) \cdot P_{po}), \\ \text{if } c_j \text{ is not subject} \end{cases}$$

Next, we need to re-estimate the cardinalities of all variables $\overline{v} \in p_j$. Let $|p.\overline{v}|$ denote $|p.s|$ or $|p.o|$ if $\overline{v}$ is subject or object, respectively. Similarly, let $P_{p\overline{v}}$ denote $|P_{ps}|$ if $\overline{v}$ is subject or $|P_{po}|$ if $\overline{v}$ is object. We re-estimate the cardinality of $\overline{v}$ in the new state S' as:

$$S'.B(\overline{v}) = \begin{cases} min(S.B(\overline{v}), |P|), & \text{if } \nu = 1 \\ min(S.B(\overline{v}), |p.\overline{v}|), & \text{if } \overline{v} = c_j \text{ \& } \nu > 1 \\ min(S.B(\overline{v}), S.B(\overline{v}) \cdot P_{p\overline{v}}, |p.\overline{v}|), & \text{if } \overline{v} \neq c_j \text{ \& } \nu > 1 \end{cases}$$

We use cumulative cardinality when we reach the same state by two different orderings. Thus, we also re-estimate the cumulative state cardinality $|S'|$. If P_{pc_j} denotes $|P_{ps}|$ or $|P_{po}|$ depending on the position of c_j, $|S'| = |S| \cdot (1 + P_{pc_j})$. Note that we use an upper bound estimation for cardinalities. A special case of the last equation is when a subquery has a constant; then, we assume that each tuple in the previous state connects to this constant by setting P_{pc_j}=1. Note that a more accurate cardinality estimation like the one used in Trinity.RDF [27] is orthogonal to our optimizer.

Table 3.6: Datasets Statistics in millions (M)

Dataset	Triples (M)	#S (M)	#O (M)	#S∩O (M)	#P	Indegree (Avg/StDev)	Outdegree (Avg/StDev)
LUBM-10240	1,366.71	222.21	165.29	51.00	18	16.54/26000.00	12.30/5.97
WatDiv	109.23	5.21	17.93	4.72	86	22.49/960.44	42.20/89.25
WatDiv-1B	1,092.16	52.12	179.09	46.95	86	23.69/2783.40	41.91/89.05
YAGO2	284.30	10.12	52.34	1.77	98	5.43/2962.93	28.09/35.89
Bio2RDF	4,287.59	552.08	1,075.58	491.73	1,714	8.64/21110.00	16.83/195.44

3.3 Experimental Evaluation

In this section, we evaluate the non-adaptive version of AdPart, coined AdPart-NA against existing systems. The adaptive version of AdPart will be evaluated in the next chapter. In Section 3.3.1, we provide the details of the data, the hardware setup, and the competitors to our approach. In Section 3.3.2, we demonstrate the low startup and initial replication overhead of AdPart-NA compared to all other systems. Then, in Section 3.3.3, we apply queries with different complexities on different datasets to show that AdPart-NA leverages the subject-based hash locality to achieve better or similar performance compared to other systems. The results show that the baseline query evaluation i.e. for queries that are not favored be the current distribution, are answered efficiently by AdPart-NA without jeopardizing the overall system performance.

3.3.1 Setup and Competitors

Datasets: We conducted our experiments using real and synthetic datasets of variable sizes. Table 3.6 describes these datasets, where #S, #P, and #O denote respectively the numbers of unique subjects, predicates, and objects. We use the synthetic LUBM[2] data generator to create a dataset of 10,240 universities consisting of 1.36 billion triples. LUBM and its template queries are used for testing most distributed RDF engines [27, 30, 31, 32]. However, LUBM queries are intended for semantic inferencing and their complexities lie in semantics not structure. Therefore, we also use

[2]http://swat.cse.lehigh.edu/projects/lubm/

WatDiv[3] which is a recent benchmark that provides a wide spectrum of queries with varying structural characteristics and selectivity classes. We used two versions of this dataset: WatDiv (109 million) and WatDiv-1B (1 billion) triples. As both LUBM and WatDiv are synthetic, we also use two real datasets. YAGO2[4] is a real dataset derived from Wikipedia, WordNet and GeoNames containing 300 million triples. Bio2RDF[5] dataset provides linked data for life sciences and contains 4.64 billion triples connecting 24 different biological datasets. The details of all queries and workloads used in this thesis are available in the Appendix A.1.

Hardware Setup: We implemented AdPart in C++ and used a Message Passing Interface library (MPICH2) for synchronization and communication. Unless otherwise stated, we deploy AdPart and its competitors on a cluster of 12 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs (12 cores each). The machines run 64-bit 3.2.0-38 Linux Kernel and are connected by a 10Gbps Ethernet switch.

Competitors: We compare AdPart-NA against TriAD [32], a recent in-memory RDF system, which is shown to have the fastest query response times to date. We compare to TriAD and TriAD-SG; the former uses lightweight hash partitioning while the later uses graph summaries for join-ahead pruning. We also compare against three Hadoop-based systems which use lightweight partitioning: CliqueSquare[38], SHARD [28] and H2RDF+ [30]. Furthermore, we compare to two systems that rely on static replication by prefetching and use RDF-3X as underlying data store: SHAPE [31] and H-RDF-3X [26]. We configure SHAPE with full level semantic hash partitioning and enable the type optimization (see [31] for details). For H-RDF-3X, we enable the type and high degree vertices optimizations (see [26] for details). Finally, we compare to DREAM [78], a distributed system that does not partition the data, rather it distributes the

[3]http://db.uwaterloo.ca/watdiv/
[4]http://yago-knowledge.org/
[5]http://download.bio2rdf.org/release/2/

Table 3.7: Partitioning Configurations

	LUBM-10240	WatDiv	Bio2RDF	YAGO2
SHAPE	2 forward	3 undirected	2 undirected	2 forward
H-RDF-3X	2 undirected	3 undirected	2 undirected	2 undirected

query execution among fully-fledged unpartitioned data stores. We compare with distributed systems only, because they outperform state-of-the-art centralized RDF systems.

3.3.2 Startup Time and Initial Replication

Our first experiment measures the time it takes all systems for preparing the data prior to answering queries. We exclude the string-to-id mapping time for all systems. For fair comparison, SHAPE and H-RDF-3X were configured to partition each dataset such that all its queries are processable without communication. Table 3.7 shows the details of these partitioning configurations. Using 2-hop forward guarantee for H-RDF-3X (which minimizes its replication [26]), we cannot guarantee that all queries can be answered without communication. This is mainly due to the high degree vertices optimization. For TriAD-SG, we used the same number of partitions reported in [32] for partitioning LUBM-10240 and WatDiv. Determining a suitable number of summary graph partitions requires empirical evaluation of some workload on the data or a representative sample. While generating a representative sample from these real data might be tricky, empirical evaluation on the original big data is costly [32]. Therefore, for fair comparison, we do not evaluate TriAD-SG on Bio2RDF and YAGO2.

As Table 3.8 shows, systems that rely on METIS for partitioning (i.e., H-RDF-3X and TriAD-SG) have significant startup cost. This is because METIS does not scale to large RDF graphs. To apply METIS, we had to remove all triples connected to literals; otherwise, METIS takes several days to partition LUBM-10240 and YAGO2.

Table 3.8: Preprocessing time (minutes)

	LUBM-10240	WatDiv	Bio2RDF	YAGO2
AdPart	**14**	**1.2**	**29**	**4**
TriAD	72	4	75	11
SHARD	72	9	143	17
H2RDF+	152	9	387	22
CliqueSquare	167	10	N/A	19
SHAPE	263	79	>24h	251
DREAM	392	33	>24h	91
TriAD-SG	737	63	N/A	N/A
H-RDF-3X	939	285	>24h	199

For LUBM-10240, SHAPE incurs less preprocessing time compared to METIS-based systems. However, for WatDiv and YAGO2, SHAPE performs worse because of data imbalance, causing some of the RDF-3X engines to take more time in building the databases. Partitioning YAGO2 and WatDiv using 2-hop forward and 3-hop undirected, respectively, placed all the data in a single partition. The reason is that all these datasets have uniform URI's, hence SHAPE could not utilize its semantic hash partitioning. SHAPE and H-RDF-3X did not finish partitioning Bio2RDF and were terminated after 24 hours.

SHARD and H2RDF+ employ lightweight partitioning, random and range-based, respectively. CliqueSquare uses a combination of hash and vertical partitioning. Therefore, they require less time compared to other systems. However, since they are Hadoop-based, they suffer from the overhead of storing the data first on Hadoop File System (HDFS) before building their data stores. TriAD and AdPart-NA use lightweight hash partitioning and avoid the upfront cost of sophisticated partitioning schemes. As Table 3.8 shows, both systems start 4X up to two orders of magnitude faster than other systems. TriAD takes more time because it hash partitions the data twice (on the subject and object columns). Furthermore, TriAD requires extra time for sorting its indices and computing statistics. Finally, while DREAM does not partition the data among workers, it incurs a significant overhead building and

Table 3.9: Initial replication

	LUBM-10240	**WatDiv**	**YAGO2**
SHAPE	42.9%	(1 worker) 0%	(1 worker) 0%
H-RDF-3X	19.5%	1090%	73.7%

indexing the entire database on a single machine. While not reported in this experiment, there is another significant overhead for copying the database among all nodes because of the shared nothing environment. This approach works reasonably well for small datasets, however; it does not scale for large datasets like Bio2RDF, requiring more than a day for building the database.

Initial replication: We report only the initial replication of SHAPE and H-RDF-3x, since AdPart-NA, TriAD, SHARD and H2RDF+ do not incur any initial replication (the replication caused by AdPart's adaptivity is evaluated in the next chapter). For LUBM-10240, H-RDF-3X results in the least replication (see Table 3.9) as LUBM is uniformly structured around universities (high degree vertices). Because of the high degree optimization, all entities of type university and their edges are removed before partitioning the graph using METIS. The resulting partitions are fully disconnected with zero edge cut. The extra replication is mainly because of the ownership triples used for duplicate elimination (see [26] for details). With full level semantic hash partitioning and type optimization, SHAPE incurs almost double the replication of H-RDF-3X. For WatDiv, METIS produces very bad partitioning because of the dense nature of the data. Consequently, partitioning the whole data blindly using k-hop guarantee would result in excessive replication because of the high edge-cut. H-RDF-3X [26] replicated the data almost 11 times, i.e., each partition has almost the whole original graph. Because of the URI's uniformity of WatDiv and YAGO2, SHAPE places the data on a single partition. Therefore, it incurs no replication but performs as good as a single machine RDF-3X store.

Table 3.10: Query runtimes for LUBM-10240 (ms)

LUBM-10240	L1	L2	L3	L4	L5	L6	L7	Geo-Mean
AdPart-NA	**2,743**	**120**	**320**	**1**	**1**	40	**3,203**	**75**
TriAD	6,023	1,519	2,387	6	4	114	17,586	369
TriAD-SG	5,392	1,774	4,636	9	5	**10**	21,567	333
SHAPE	25,319	4,387	25,360	1,603	1,574	1,567	15,026	5,575
H-RDF-3X	7,004	2,640	7,957	1,635	1,586	1,965	7,175	3,412
H-RDF-3X (in-memory)	6,841	2,597	7,948	1,596	1,594	1,926	7,551	3,397
CliqueSquare	125,020	71,010	80,010	90,010	24,000	37,010	224,040	74,488
H2RDF+	285,430	71,720	264,780	24,120	4,760	22,910	180,320	59,275
SHARD	413,720	187,310	aborted	358,200	116,620	209,800	469,340	261,362
DREAM	13,031,410	98,263	2,358	18	14	10,755	4,700,381	12,110
DREAM (cached stats)	1,843,376	98,263	<1	18	14	468	83,053	911

3.3.3 Query Performance

In this section, we compare AdPart-NA on individual queries against state-of-the-art distributed RDF systems. We demonstrate that even with simple partitioning scheme AdPart-NA is competitive to systems that employ sophisticated partitioning techniques. This shows that the subject-based hash partitioning and the distributed evaluation techniques proposed in Section 3.2 are very effective.

LUBM dataset: In the first experiment (Table 3.10), we compare the performance of all systems using the LUBM-10240 dataset and queries L1-L7 defined in [24]. Queries L1-L7 can be classified based on their structure and selectivities into simple and complex. L4 and L5 are simple selective star queries whereas L2 is a simple yet non-selective star query that generates large final results. L6 is a simple query because it is highly selective. L1, L3 and L7 are complex queries with large intermediate results but very few final results.

SHARD CliqueSquare and H2RDF+ suffer from the expensive overhead of MapReduce-based joins; hence, their performance is significantly worse than all other systems. The flat plans of CliqueSquare significantly reduce the joins overhead for complex queries. However, for selective simple queries, H2RDF+ avoids the overhead of MapReduce based joins by solving these queries in a centralized manner. Hence, it achieves an order of magnitude better performance for these queries. SHAPE and H-RDF-3X perform better than MapReduce-based systems because they do not require commu-

nication. H-RDF-3X performs better than SHAPE because it has less replication. However, as both SHAPE and H-RDF-3X use MapReduce for dispatching queries to workers, they still suffer from the non-negligible overhead of MapReduce (around 1.5 seconds on our cluster). Without this overhead, both systems would perform well for simple selective queries. Even for complex queries, these systems still perform reasonably well because queries run in parallel without any communication overhead. For example, for query L7 which requires excessive communication, H-RDF-3X and SHAPE perform better than TriAD and TriAD-SG. Note that this performance comes at a high preprocessing cost. Obviously, with a low hop guarantee, the preprocessing cost for SHAPE and H-RDF-3X is reduced but the query performance becomes worse because of the MapReduce-based joins [32]. AdPart-NA outperform SHAPE and H-RDF-3X for three reasons: (i) managing the original and replicated data in the same set of indices results in large and duplicate intermediate results, rendering the cost of join evaluation higher. (ii) To filter out duplicate results, H-RDF-3X requires and additional join with the ownership triple pattern. (iii) TriAD and AdPart are designed as in-memory engines while RDF-3X is disk-based. For fairness, we also stored H-RDF-3X databases in a memory-mounted partition; still, it did not affect the performance significantly.

DREAM relies on the underlying engine (RDF-3X) for gathering some statistics that will be used by DREAM's query planner to decide how a query is decomposed. Statistics are collected by evaluating the selectivities of many joins in each decomposed subgraph. DREAM requires the execution of many permutations of these joins in order to collect exact statistics. Statistics are cached for future queries on a query-by-query basis, which explains the huge performance difference between the two versions of DREAM. In a nutshell, the exact statistics calculation in DREAM is impractical. A minimal change in the query structure will mandate statistics recalculation. Moreover, DREAM suffers from limited parallelism because of the query

decomposition approach. The number of machines that can be utilized during query execution is bounded by the number of join vertices. In the case of LUBM queries, the maximum number of join vertices is 3. On the other hand, all other systems can utilize all machines during query execution; which explains the performance superiority of systems that execute the query in parallel by all workers like H-RDF-3X SHAPE, AdPart-NA and TriAD. The overhead of DREAM's limited parallelism is significant in complex queries that generate large intermediate results i.e. queries L1 and L7.

In-memory RDF engines, AdPart-NA and TriAD, perform equally for queries L4 and L5 due to their high selectivities and star-shapes. AdPart-NA exploits the initial hash distribution and solves these queries without communication. Similarly, L2 consists of a single subject-subject join; however, it is highly non-selective. TriAD solves the query by two distributed index scans (one for each base subquery) followed by a merge join. The merge join utilizes binary search for finding the beginning of the sorted runs from the left and right relations. These searches perform well for selective queries but not for L2. AdPart-NA performs better than TriAD-SG by avoiding unnecessary scans. In other words, utilizing its hash indexes and the right deep tree planning, AdPart-NA requires a single scan followed by hash lookups. As a result, AdPart-NA is faster than TriAD by more than an order of magnitude. The pruning technique of TriAD-SG eliminates the communication required for solving L6. Therefore, it outperforms TriAD and AdPart-NA. DREAM execute all these queries in a centralized manner by directing the queries to a single machine RDF-3X.

Although AdPart-NA and TriAD have a similar partitioning scheme (with the difference in TriAD's full locality awareness on both subjects and objects), AdPart-NA achieves better performance than TriAD and TriAD-SG for all complex queries, L1, L3 and L7. There are three reasons for this: (i) When executing distributed merge/hash joins, TriAD needs to shard one/both relations among workers. On

Table 3.11: Query runtimes for WatDiv (ms)

WatDiv-100	Machines	L1-L5	S1-S7	F1-F5	C1-C3
AdPart-NA	**5**	9	**7**	160	**111**
TriAD	5	**4**	15	**45**	170
SHAPE	12	1,870	1,824	1,836	2,723
H-RDF-3X	12	1,662	1,693	1,778	1,929
H2RDF+	12	5,441	8,679	18,457	65,786
CliqueSquare	12	29,216	23,908	40,464	55,835

the contrary, AdPart-NA only exchanges the unique values from the projected join column. The effect becomes more prominent in TriAD when concurrent joins at the lower level of the execution plan generate large and unnecessary intermediate results. These results need to be asynchronously sharded before executing joins at higher levels. (ii) AdPart-NA exploits the subject-based locality and the locality of the intermediate results (pinning strategy) during planning to decide which join operators can run without communication regardless of their location in the execution tree. On the other hand, in TriAD, once a relation is sharded the locality of the intermediate results is destroyed which mandates further shardings at higher join operators. Finally, (iii) as in L2, if the join being executed is not selective, merge join performs worse than the hash joins used by AdPart-NA. The pruning technique of TriAD-SG was effective in reducing the overall slaves query execution time. However, the cost for summary graph processing in L3 and L7 was high; hence, the high query execution times compared to TriAD.

For L3, AdPart-NA is 7x to 14x faster than Triad and TriAD-SG, respectively. AdPart-NA evaluates the join that gives an empty intermediate result early, which avoids subsequent useless joins. However, the first few joins cannot be eliminated during query planning time. For DREAM, the planner detects that there is a join with empty result during statistics collection and elects to terminate the query execution. Once DREAM caches the query statistics, it will decide that the query has empty results and terminate.

WatDiv dataset: The WatDiv benchmark defines 20 query templates[6] classified into four categories: linear (L), star (S), snowflake (F) and complex queries (C). Similar to TriAD, we generated 20 queries using the WatDiv query generator for each query category C, F, L and S. We deployed AdPart-NA on five machines to match the setting of TriAD in [32]. Table 3.11 shows the performance of AdPart-NA compared to other systems. For each query class, we show the geometric mean of each system. H2RDF+ and CliqueSquare[7] perform worse than all other systems due to the MapReduce overhead. H2RDF+ performs much better than CliqueSquare. The reason is that, while the flat plans reduce the number of MapReduce-based joins, H2RDF+ uses a more efficient implementation of the join operator using traditional RDF indices. Furthermore, unlike CliqueSquare, H2RDF+ encodes the URIs and literals of RDF data; hence it incurs minimal overhead when shuffling intermediate results. SHAPE and H-RDF-3X, under 3-hop undirected guarantee, do not perform better than a single-machine RDF-3X. SHAPE placed all the data in one machine while H-RDF-3X replicated almost all the data everywhere. AdPart-NA and TriAD, on the other hand, provide significantly better performance than MapReduce-based systems. TriAD performs better than AdPart-NA for L and F queries as these queries require multiple subject-object joins. TriAD can utilize the subject-object locality to answer these joins without communication whereas AdPart needs to communicate. Note that utilizing subject-object locality as in TriAD is orthogonal to our work. For complex queries with large diameters AdPart-NA performs better as a result of its locality awareness. The overhead for statistics calculation in DREAM was extremely high because of the high number of triple patterns in WatDiv benchmark queries. For example, a complex query from the WatDiv templates would take more than 24 hours to execute. Therefore, DREAM results were omitted.

[6] `http://db.uwaterloo.ca/watdiv/basic-testing.shtml`

[7] CliqueSquare crashed while executing 5 and 12 queries from the star and snowflake templates, respectively.

Table 3.12: Query runtimes for YAGO2 (ms)

YAGO2	**Y1**	**Y2**	**Y3**	**Y4**	**Geo-Mean**
AdPart-NA	19	**46**	570	77	**79**
TriAD	**16**	1,568	**220**	**18**	100
SHAPE	1,824	665,514	1,823	1,871	8,022
H-RDF-3X	1,690	246,081	1,933	1,720	6,098
H2RDF+	10,962	12,349	43,868	35,517	21,430
CliqueSquare	139,021	73,011	36,006	100,015	77,755
SHARD	238,861	238,861	aborted	aborted	238,861

YAGO dataset: YAGO2 does not provide benchmark queries, therefore we created a set of representative test queries (Y1-Y4). We show in Table 3.12 the performance of AdPart-NA against all other systems. Similar, to the WatDiv dataset, H2RDF+ outperforms CliqueSquare and SHARD due to the utilization of HBase indices and its distributed implementation of merge and sort-merge joins. AdPart-NA solves most of the joins in Y1 and Y2 without communication; three out of four in Y1 and four out of six in Y2. This explains the comparable to superior performance of AdPart-NA compared to TriAD for Y1 and Y2, respectively. On the other hand, Y3 requires an object-object join on which AdPart-NA needs to broadcast the join column. As a results, TriAD performed better than AdPart-NA.

Bio2RDF dataset: Similar to YAGO2, the Bio2RDF dataset does not have benchmark queries; therefore, we defined five queries (B1-B5) with different structures and complexities. B1 requires an object-object join which contradicts our initial partitioning. B2 and B3 are star queries with different number of triple patterns that require subject-object joins. Therefore, it is expected that TriAD would perform better than AdPart-NA (see Table 3.13). B4 is a complex query with 2-hop radius. AdPart-NA incur communication and utilize subject-based locality during sharding. TriAD, on the other hand, crashed during query evaluation, hence marked as N/A. B5 is a simple star query with only one triple pattern in which all in-memory systems provide the same performance. H2RDF+ and SHARD perform worse than other systems due to

Table 3.13: Query runtimes for Bio2RDF (ms)

Bio2RDF	B1	B2	B3	B4	B5	Geo-Mean
AdPart-NA	17	16	32	89	**1**	15
TriAD	**4**	**4**	**5**	N/A	2	**4**
DREAM	208	102	742	aborted	82	188
DREAM (cached stats)	16	15	142	aborted	12	25
H2RDF+	5,580	12,710	322,300	7,960	4,280	15,076
SHARD	239,350	309,440	512,850	787,100	112,280	320,027

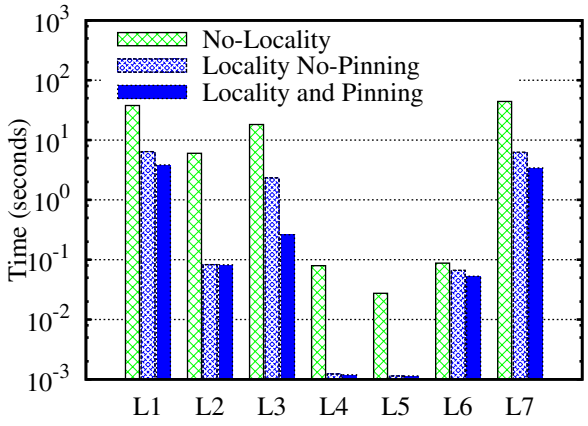
the MapReduce overhead. Overall, TriAD outperforms[8] all systems; however, as we will show in the next chapter, when AdPart adapts, it performs significantly better than all other systems.

Impact of Locality Awareness

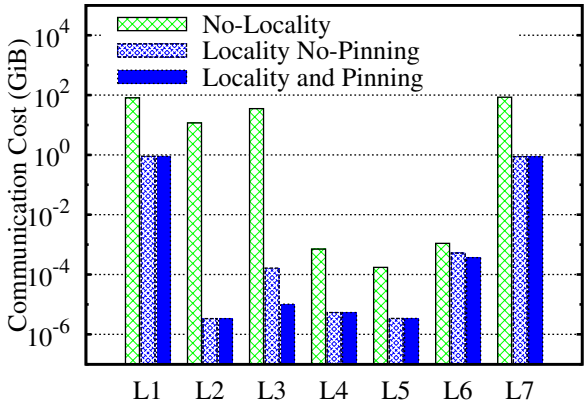
In this experiment, we show the effect of locality aware planning on the distributed query evaluation of AdPart-NA (non-adaptive). We define three configurations of AdPart-NA: (i) We disable the *pinned_subject* optimization and hash locality awareness. (ii) We disable the *pinned_subject* optimization while maintaining the hash locality awareness; in other words, workers can still know the locality of subject vertices but joins on the pinned subjects are synchronized. Finally, (iii) we enable all optimizations. We run the LUBM (L1-L7) queries on the LUBM-10240 dataset on all configurations. The query response times and the communication costs are shown in Figures 3.5(a) and 3.5(b), respectively.

Disabling hash locality resulted in excessive communication which drastically affected the query response times. Enabling the hash locality affected all queries except L6 because of its high selectivity. The performance gain for other queries ranges from 6X up to 2 orders of magnitude. In the third configuration, the pinned subject optimization does not affect the amount of communication because of the hash locality awareness. In other words, since the joining subject is local, AdPart does not communicate intermediate results. However, performance is affected by the synchronization

[8]Failed runs are not included when calculating the geometric mean.



(a) Execution Time



(b) Communication Cost

Figure 3.5: Impact of locality awareness on LUBM-10240.

overhead. The performance gain ranges from 26% in case of L6 to more than 90% for L3. Queries like L2, L4 and L5 are not affected by this optimization because they are star queries joining on the subject. The same behavior is also noticed in the WatDiv-1B dataset.

3.4 Discussion

Our experimental evaluation shows that, by employing simple hash-based partitioning, AdPart starts significantly faster than all existing systems. Moreover, by exploiting subject hash-locality, AdPart achieves a competitive performance to state-of-the-art systems that use sophisticated partitioning. AdPart does not only perform well for star queries that join on subjects i.e. queries that consist of subject-subject joins only, it also solve other types of joins very efficiently by minimizing communication and synchronization overhead.

While the baseline approach of AdPart is competitive, there are other type of joins for which AdPart incurs some overhead. In particular, object-object joins cannot be solved without data broadcast. While this type of joins is not common [40], queries with such joins can be frequent i.e. the same query is repetitive. In the next chapter, we introduce the adaptivity feature of AdPart that allows it to incrementally adapts its data partitioning to honor such type of joins.

Chapter 4

Workload Adaptivity

> We cannot solve our problems with the same thinking we used when we created them.
>
> Albert Einstein
> 1879 — 1955 CE

Studies show that even minimal communication results in significant performance degradation [26, 31, 32]. Thus, data should be redistributed to minimize, if not eliminate, communication and synchronization overheads. AdPart redistributes only the parts of data needed for the current workload and adapts as the workload changes. AdPart monitors the submitted queries in the form of a heat map to detect hot patterns. Once such a pattern is detected, AdPart redistributes and potentially replicates the data accessed by the pattern among workers. Consequently, AdPart adapts to the query load and can answer more queries in parallel mode. The incremental redistribution model of AdPart is a combination of hash partitioning and k-hop replication, guided by the query load rather than the data itself. Specifically, given a hot pattern Q (hot pattern detection is discussed in Section 4.5), AdPart selects a special vertex in the pattern called the *core* vertex (Section 4.2). The system groups the data accessed by the pattern around the bindings of this core vertex. To do so, the system transforms the pattern into a redistribution tree rooted at the core (Section 4.3). Then, starting from the core vertex, first hop triples are hash distributed based on the core bindings. Next, triples that match the second level subqueries are collocated

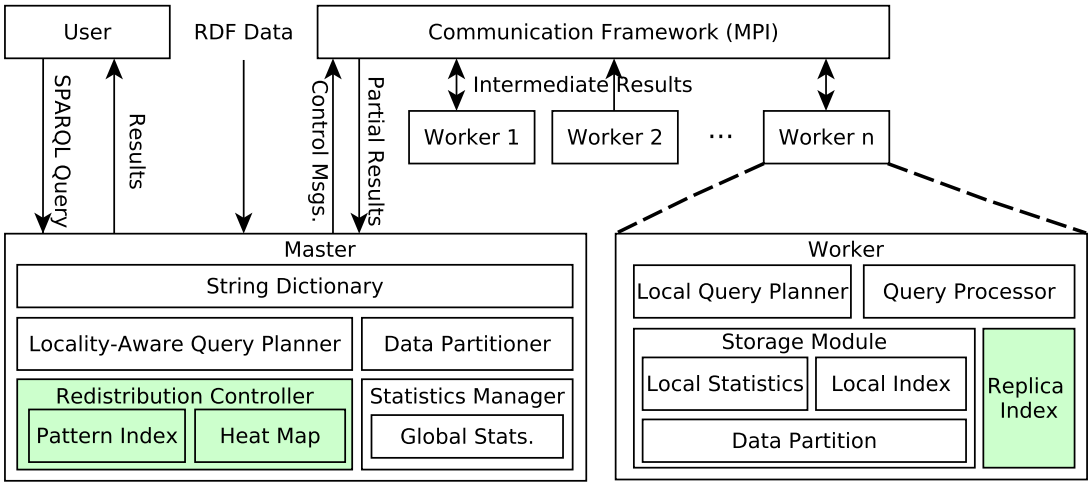


Figure 4.1: System architecture of AdPart

and so on (Section 4.4). AdPart utilizes redistributed patterns to answer queries in parallel without communication.

4.1 Revised System Architecture

To facilitate the adaptivity feature of AdPart, its architecture is slightly modified by adding the modules highlighted in Figure 4.1.

4.1.1 Master

Redistribution Controller. It monitors the workload in the form of heat maps (Section 4.5.1) and triggers the adaptive Incremental ReDistribution (IRD) (Section 4.4) process for hot patterns. Data accessed by hot patterns are redistributed and potentially replicated among workers. A redistributed hot pattern can be answered by all workers in parallel without communication. Replicated hot patterns are indexed in a structure called Pattern Index (PI) (Section 4.6.1). Patterns in the PI can be combined for evaluating future queries without communication. Further, the controller implements replica replacement policy to keep replication within a threshold.

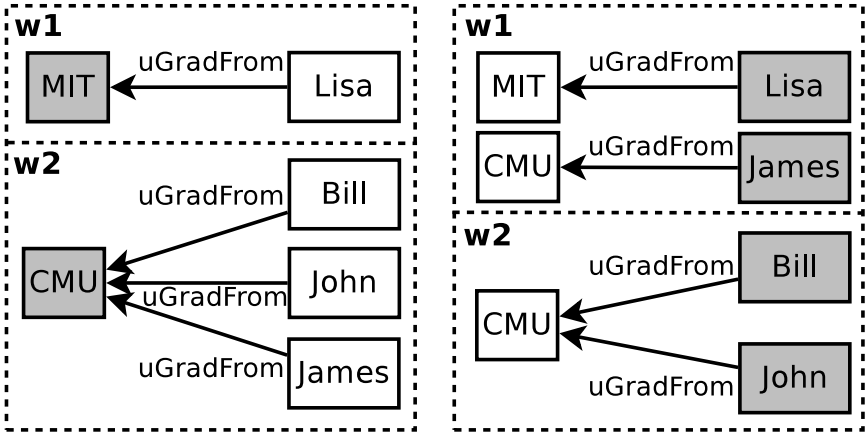
4.1.2 Worker

Replica Index. Each worker has an in-memory replica index (Section 4.6.2) that stores and indexes replicated data as a result of the adaptivity. This index initially contains no data and is updated dynamically by the incremental redistribution (IRD) process.

4.2 Core Vertex Selection

For a hot pattern, the choice of the core vertex has a significant impact on the amount of replicated data as well as on query execution performance. For example, consider query $Q_1 = \langle$?stud, uGradFrom, ?univ$\rangle$. Assume there are two workers, w_1 and w_2, and refer to the graph of Figure 1.3; MIT and CMU are the bindings of ?*univ*, whereas Lisa, John, James and Bill bind to ?*stud*. Assume that ?*univ* is the core, then triples matching Q_1 will be hashed on the bindings of ?*univ* as shown in Figure 4.2(a). Note that every binding of ?*stud* appears in one worker only. Now assume that ?*stud* is the core and triples are hashed using the bindings of ?*stud*. This causes binding ?*univ*=CMU to exist on both workers (see Figure 4.2(b)). The problem becomes more pronounced when the query has more triple patterns. Consider $Q_2 = Q_1$ `AND` $\langle$?prof, gradFrom, ?univ$\rangle$ and assume that ?*stud* is chosen as core. Because CMU exists on both workers, all its graduates (i.e., triples matching $\langle$?prof, gradFrom, CMU$\rangle$ will also be replicated. Replication grows exponentially with the number of triple patterns [26, 31].

Intuitively, if random walks start from two random vertices (e.g., students), the probability of reaching the same well-connected vertex (e.g., university) within a few hops is higher compared to other nodes. In order to minimize replication, we must avoid reaching the same vertex when starting from the core. Hence, it is reasonable to select a well-connected vertex as the core. Although, well-connected vertices can be identified by complex data mining algorithms in the literature, for the sake of minimizing the computational cost, we employ a simple approach. We assume that



(a) Core is ?*univ* (b) Core is ?*stud*

Figure 4.2: Effect of choice of core on replication. In (a) there is no replication. In (b) CMU is both workers.

connectivity is proportional to degree centrality (i.e., in-degree plus out-degree edges). Recall from Section 3.2.3 that we maintain statistics $\overline{p_S}$ and $\overline{p_O}$ for each predicate $p \in P$, where P is the set of all predicates in the data. Let P_s and P_o be the set of all $\overline{p_S}$ and $\overline{p_O}$, respectively. We filter out predicates with extremely high scores and consider them outliers[1].

Outliers are detected using Chauvenet's criterion [111] on P_s then P_o. If a predicate p is detected as an outlier, we set: $\overline{p_S} = \overline{p_O} = -\infty$; otherwise we use $\overline{p_S}$ and $\overline{p_O}$ as computed in Section 3.2.3. Now, we can compute a score for each vertex in the query as follows:

Definition **1** (Vertex score)**.** For a query vertex v, let $E_{out}(v)$ be the set of outgoing edges and $E_{in}(v)$ be the set of incoming edges. Also, let A be the set of all $\overline{p_S}$ for the $E_{out}(v)$ edges and all $\overline{p_O}$ for $E_{in}(v)$ edges. The vertex score $\overline{v}$ is defined as: $\overline{v} = \max(A)$.

Figure 4.3 shows an example for vertex score assignment. For vertex ?*prof*,

[1]In many RDF datasets, vertex degrees follow a power-law distribution, where few ones have extremely high degrees. For example, vertices that appear as objects in triples with rdf:type have very high degree centrality. Treating such vertices as cores results in imbalanced partitions and prevents the system from taking full advantage of parallelism [26].

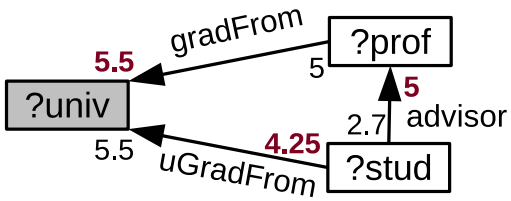


Figure 4.3: Example of vertex score: numbers correspond to $\overline{p_S}$ and $\overline{p_O}$ values. Assigned vertex scores $\overline{v}$ are shown in bold.

$E_{in}(?prof) = \{\texttt{advisor}\}$ and $E_{out}(?prof) = \{\texttt{gradFrom}\}$. Both predicates (i.e., advisor and gradFrom) contribute a score of 5 to $?prof$. Therefore, $\overline{?prof} = 5$.

Definition **2** (Core vertex)**.** Given a query graph $G = (V, E)$ such that V and E are the set of vertices and edges, respectively. Let $f(v)$ be a scoring function that assigns a score to each $v \in V$. We define the core vertex of Q as v' such that $f(v') = \max_{v \in V} f(v)$.

In Figure 4.3, $?univ$ has the highest score, hence, it is the core vertex for this pattern.

4.3 Generating the Redistribution Tree

Let Q be a hot pattern that AdPart decides to redistribute and let D_Q be the data accessed by this pattern. Our goal is to redistribute (partition) D_Q among all workers such that Q can be evaluated without communication. Unlike previous work that performs static MinCut-based partitioning [54], we eliminate the edge cuts by replicating edges that cross partitions. Since the balanced partitioning is an NP-complete problem, we introduce a heuristic for partitioning D_Q with two objectives in mind: (i) the redistribution of D_Q should benefit Q as well as other patterns. (ii) Because replication is necessary for eliminating communication, redistributing D_Q should result in minimal replication.

To address the first objective, we transform the pattern Q into a tree T by breaking cycles and duplicating some vertices in the cycles. The reason is that cycles constrain

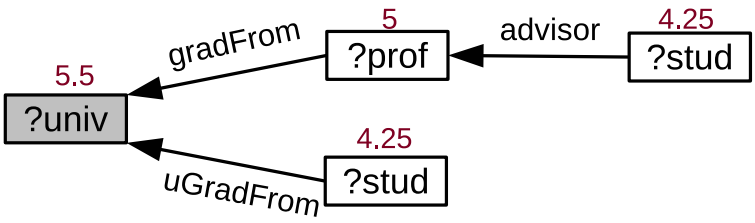


Figure 4.4: The query in Figure 4.3 transformed into a tree using Algorithm 2. Numbers near vertices define their scores. The shaded vertex is the core.

the data grouped around the core to be also cyclic. For example, the query pattern in Figure 4.3 retrieves students who share the same alma mater with their advisors. Grouping the data around universities without removing the cycle is not useful for retrieving professors and their advisees who do not share the same university. Consequently, the pattern in Figure 4.3 can be transformed into a tree by breaking the cycle and duplicating the ?*stud* vertex as shown in Figure 4.4. We refer to the result of the transformation as redistribution tree.

Our goal is to construct the redistribution tree that minimizes the expected amount of replication. In Section 4.2, we explained why starting from the vertex with the highest score has the potential to minimize replication. Intuitively, the same idea applies recursively to each level of the redistribution i.e., every child node in the tree has a lower score than its parent. Obviously, this cannot be always achieved; for example in a path pattern where a lower score vertex comes between two high score vertices. Therefore, we use a greedy algorithm for transforming a hot pattern Q into a redistribution tree T. Specifically, using the scoring function discussed in the previous section, we first transform Q into a vertex weighted, undirected graph G, where each node has a score and the directions of edges in Q are disregarded. The vertex with the highest score is selected as the core vertex. Then, G is transformed into the redistribution tree using Algorithm 2.

Algorithm 2 is a modified version of the Breadth-First-Search (BFS) algorithm, which has the following differences: (i) unlike BFS trees which span all vertices in

```
Input: G = {V, E}; a vertex-weighted, undirected graph, the core vertex v'
Result: The redistribution tree T
1  Let edges be a priority queue of pending edges
2  Let verts be a set of pending vertices
3  Let core_edges be all incident edges to v'
4  visited[v'] = true;
5  T.root=v';
6  foreach e in core_edges do
7      edges.push(v', e.nbr, e.pred);
8      verts.insert(e.nbr);
9      T.add(v', e.pred, e.nbr);
10 while edges notEmpty do
11     (parent, vertex, predicate) ← edges.pop();
12     visited[vertex] = true;
13     verts.remove(vertex);
14     foreach e in vertex.edges do
15         if e.nbr NOT visited then
16             if e.nbr ∉ verts then
17                 edges.push(vertex, e.nbr, e.pred);
18                 verts.insert(e.nbr);
19                 T.add(vertex, e.pred, e.nbr);
20             else
21                 T.add(vertex, e.pred, duplicate(e.nbr));
```

Algorithm 2: Pattern Transformation

the graph, our tree spans all edges in the graph. Each of the edges in the query graph should appear exactly once in the tree while vertices may be duplicated. (ii) During traversal, vertices with high scores are identified and explored first (using a priority queue). Since our traversal needs to span all edges, elements in the priority queue are stored as edges of the form (parent, vertex, predicate). These elements are ordered based on the vertex score first then on the edge label (predicate). Since the exploration does not follow the traditional BFS ordering, we maintain a pointer to the parent so edges can be inserted properly in the tree. As an example, consider the query in Figure 4.3. Having the highest score, ?univ is chosen as core, and the query is transformed into the tree shown in Figure 4.4. Note that the nodes have weights (scores) and the directions of edges have been moved back.

Table 4.1: Triples from Figure 1.3 matching patterns in Figure 4.4.

Worker 1		Worker 2	
t_1	⟨Lisa, uGradFrom, MIT⟩	t_3	⟨Bill, uGradFrom, CMU⟩
		t_4	⟨James, uGradFrom, CMU⟩
		t_5	⟨John, uGradFrom, CMU⟩
t_2	⟨James, gradFrom, MIT⟩	t_6	⟨Bill, gradFrom, CMU⟩
t_7	⟨Lisa, advisor, James⟩	t_8	⟨Fred, advisor, Bill⟩
		t_9	⟨John, advisor, Bill⟩
		t_{10}	⟨Lisa, advisor, Bill⟩

4.4 Incremental Redistribution

Incremental ReDistribution (IRD) aims at redistributing data accessed by hot patterns among all workers in a way that eliminates communication while achieving high parallelism. Given a redistribution tree, AdPart distributes the data along paths from the root to leaves using depth first traversal. The algorithm has two phases. First, it distributes triples containing the core vertex to workers using hash function $\mathcal{H}(\cdot)$. Let t be such a triple and let $t.core$ be its core vertex (the core can be either the subject or the object of t). Let $w_1, \ldots, w_N$ be the workers. t will be hash-distributed to worker w_j, where $j = \mathcal{H}(t.core) \mod N$. Note that if $t.core$ is a subject, t will not be replicated by IRD because of the initial subject-based hash partitioning.

In Figure 4.4, consider the first-hop triple patterns ⟨?prof, uGradFrom, ?univ⟩ and ⟨?stud, gradFrom, ?univ⟩. The core ?$univ$ determines the placement of t_1-t_6 (see Table 4.1). Assuming two workers, t_1 and t_2 are hash-distributed to w_1 (because of MIT), whereas t_3-t_6 are hash-distributed to w_2 (because of CMU). The objects of triples t_1-t_6 are called their source columns.

Definition **3** (Source column)**.** The source column of a triple (subject or object) determines its placement.

The second phase of IRD places triples of the remaining levels of the tree in the workers that contain their parent triples, through a series of distributed semi-joins.

The column at the opposite end of the source column of the previous step becomes the propagating column, i.e., ?$prof$ in our previous example.

Definition **4** (Propagating column)**.** The propagating column of a triple is its object (resp. subject) if the source column of the triple is its subject (resp. object).

At the second level of the redistribution tree in Figure 4.4, the only subquery pattern is ⟨?stud, advisor, ?prof⟩. The propagating column ?$prof$ from the previous level becomes the source column for the current pattern. Triples $t_{7...10}$ in Table 4.1 match the sub-query and are joined with triples $t_{1...6}$. Accordingly, t_7 is placed in worker w_1, whereas t_8, t_9 and t_{10} are sent to w_2.

Input: $P = \{E\}$; a path of consecutive edges, $\mathcal{C}$ is the core vertex.
Result: Data replicated along path P
`// hash-distributing the first (core-adjacent) edge`
1 **if** e_0 *is not replicated* **then**
2 $coreData$ = `getTriplesOfSubQuery(`e_0`);`
3 **foreach** t *in* $coreData$ **do**
4 $m = B(\mathcal{C}) \mod N$; `//` N `is the number of workers`
5 `sendToWorker(`t, m`);`
`// then collocate triples from other levels`
6 **foreach** $i : 1 \rightarrow |E|$ **do**
7 **if** e_i *is not replicated* **then**
8 $candidTriples$ = `DSJ(`e_0, e_i`);`
9 `IndexCandidateTriples(`$candidTriples$`);`
10 $e_0 = e_i$;

Algorithm 3: Incremental Redistribution

The IRD process is formally described in Algorithm 3. For brevity, we describe the algorithm on a path input since we follow depth-first traversal. The algorithm runs in parallel on all workers. Lines 1-5 hash distribute triples that contain the core vertex $\mathcal{C}$, if necessary.[2]. Then, triples of the remaining levels are localized (replicated) in the workers that contain their parent. Replication is avoided for each triple which is already in the worker. This is carried out through a series of DSJ (lines 6-10).

[2]Recall if a core vertex is a subject, we do not redistribute.

We maintain candidate triples at each level rather than final join results. Managing replicas in raw triple format allows us to utilize the RDF indices when answering queries using replicated data.

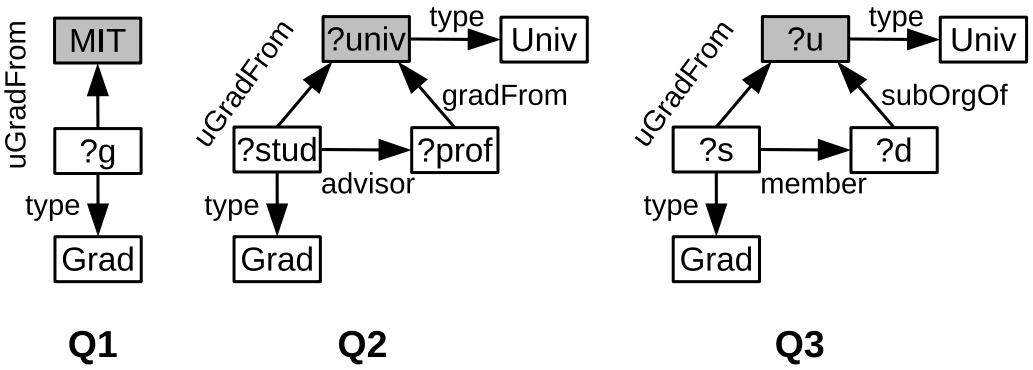
4.5 Queryload Monitoring

To effectively monitor workloads, systems face the following challenges: (i) the same query pattern may occur with different constants, subquery orderings, and variable names. Therefore, queries in the workload need to be deterministically transformed into a representation that unifies similar queries. (ii) This representation needs to be updated incrementally with minimal overhead. Finally, (iii) monitoring should be done at the level of patterns not whole queries. This allows the system to identify common hot patterns among queries.

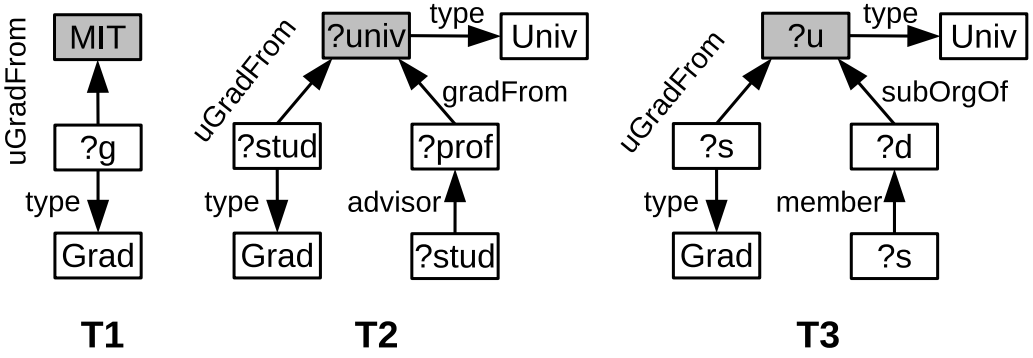
4.5.1 Heat map

We introduce a hierarchical heat map representation to monitor workloads. The heat map is maintained by the redistribution controller. Each query Q is first decomposed into a redistribution tree T using Algorithm 2 (see Section 4.3), with the core vertex as root. To detect overlap among queries, we transform T to a tree template $\mathcal{T}$ in which all the constants are replaced with variables. To avoid losing information about constant bindings in the workload, we store the constants and their frequencies as meta-data in the template vertices. After that, $\mathcal{T}$ is inserted in the heat map which is a prefix-tree like structure that includes and combines the tree templates of all queries. Insertion proceeds by traversing the heat map from the root and matching edges in $\mathcal{T}$. If the edge does not exist, we insert a new edge in the heat map and set the edge count to 1; otherwise, we increment the edge count. Furthermore, we update the meta-data of vertices in the heat map with the meta-data in $\mathcal{T}$'s vertices.

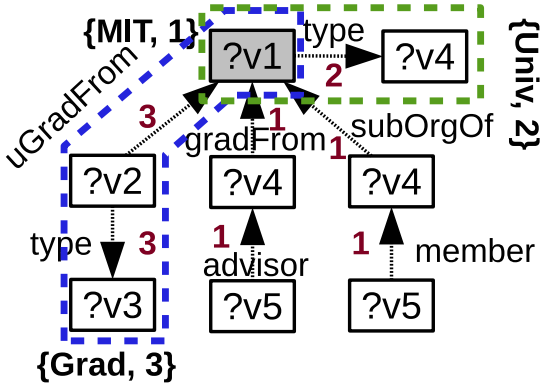
For example, consider queries Q_1, Q_2 and Q_3 and their decompositions T_1, T_2 and



(a) The queries to be answered.



(b) Queries decomposition



(c) Heat map after update

Figure 4.5: Updating the heat map. Selected areas indicate hot patterns.

T_3, respectively in Figure 4.5(a) and (b). Assume that each of the queries is executed once. The state of the heat map after executing these queries is shown in Figure 4.5(c). Every inserted edge updates the edge count and the vertex meta-data in the heat map. For example, edge $\langle ?v_2$, uGradFrom, $?v_1\rangle$ has edge count 3 because it appears in all $\mathcal{T}$'s. Furthermore, $\{MIT, 1\}$ is added to the meta-data of v_1.

We now describe the implementation details of the heat map. We use a dual tree representation for storing the heat map, where a tree node corresponds to an entire triple pattern. An edge denotes the existence of a common variable between any combination of subjects and objects in the connected triples. Note that this representation results in a tree forest. Whenever no confusion arises, we simply refer to both representations as heat map. The root node of the heat map is a dummy node that is connected to all core-adjacent edges from all patterns seen before. Figure 4.6 shows the dual representation of the heat map in Figure 4.5(c).

To update the heat map given a query Q, the tree template $\mathcal{T}$ is also transformed into its dual representation. This typically results in multiple independent trees. The heat map is updated using the dual of $\mathcal{T}$ level by level in a depth first manner. Algorithm 4 shows how the heat map is updated with a new query tree. Initially, a search process is started from the heat map root for each node in the first level of the query tree (line 1-2). The algorithm calls a procedure which takes as input both the heat map node and the query node (lines 3-16). The find function (line 6) is used to match the query node in the current level of the heat map. Recall that triple patterns in the heat map and $\mathcal{T}$ have variable subjects and objects. Therefore, a heat map node matches the query node if they share the same predicate and direction. If no match is found, a new node is inserted in the heat map as a child of the current node (lines 7-9) with frequency 1. Otherwise, the count of the matched heat map node is incremented (lines 10-11). In both cases, we update the metadata (i.e., the occurrences of the target vertices and their frequencies) of the heat map node (line

12). Then, the procedure is recursively called for each child of the query node (lines 13-14). The find function is implemented using hash lookup based on the predicate and direction of the triple pattern. Hence, the complexity of updating the heat map is $O(|E|)$, where E is the number of edges in the query graph.

Input: HeatMap dual representation $\mathcal{T}_{hm}$, query tree dual representation $\mathcal{T}_q$

Result: $\mathcal{T}_{hm}$ updated

```
1  foreach QueryNode N_q → T_q.root.childs do
2      updateFreq (T_hm.root, N_q);
3  Procedure updateFreq(HeatNode N_hm, QueryNode N_q)
5      newParent ← NULL;
6      newParent ← findNode (N_hm.children, N_q);
7      if newParent is NULL then
8          newParent ← N_hm.insert (N_q);
9          newParent.count ← 1;
10     else
11         newParent.count ++;
12     updateMetaData (newParent, N_q);
13     foreach QueryChild C_q → N_q.children do
14         updateFreq (newParent,C_q);
16     return;
```

Algorithm 4: Update Heat Map

4.5.2 Hot pattern detection

The redistribution controller monitors queries by updating the heat map using Algorithm 4. Currently, we use a hardwired frequency threshold[3] for identifying hot patterns. Recall that while updating the heat map, we also update the frequency (count) of its nodes. A pattern in the heat map is considered to be hot if the update

[3]Auto-tuning the frequency threshold is a subject of our future work.

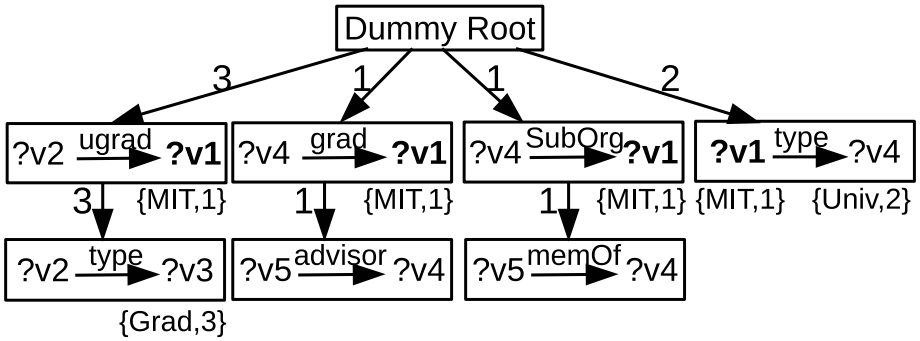


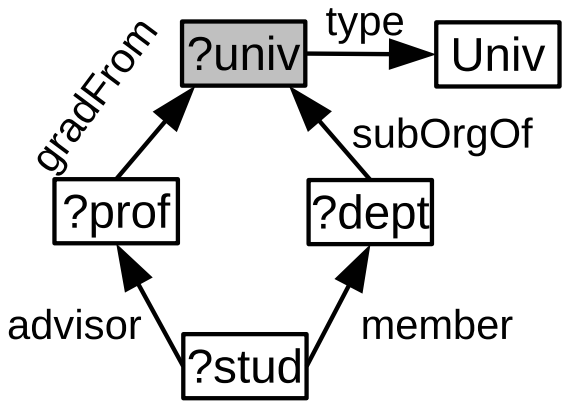
Figure 4.6: Dual Tree Representation of the heat map shown in Figure 4.5(c).

process makes its frequency greater than the threshold. As the heat map update process is carried out in a top-down fashion, we guarantee that a lower node in the heat map cannot have a frequency greater than its ancestors. Once a hot pattern is detected, the redistribution controller triggers the IRD process for that pattern. Recall that patterns in the heat map are templates in which all vertices are variables. To avoid excessive replication, some variables are replaced by dominating constants stored in the heat map. For example, assume the selected part of the heat map in Figure 4.5(c) is identified as hot. We replace vertex $?v_3$ with the constant Grad because it is the dominant value. On the other hand, $?v_1$ is not replaced by MIT because MIT does not dominate other values in query instances that include the hot pattern. We use the Boyer-Moore majority vote algorithm [112] for deciding the dominating constant.

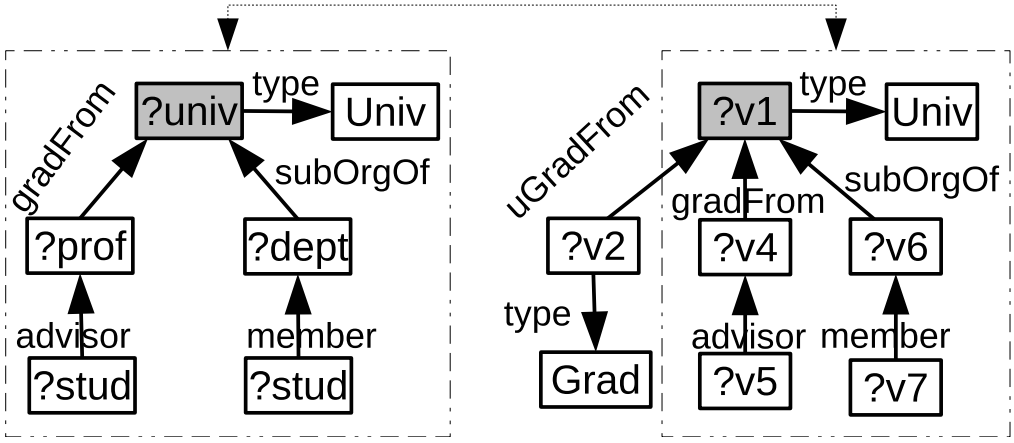
4.6 Pattern and Replica Index

4.6.1 Pattern index

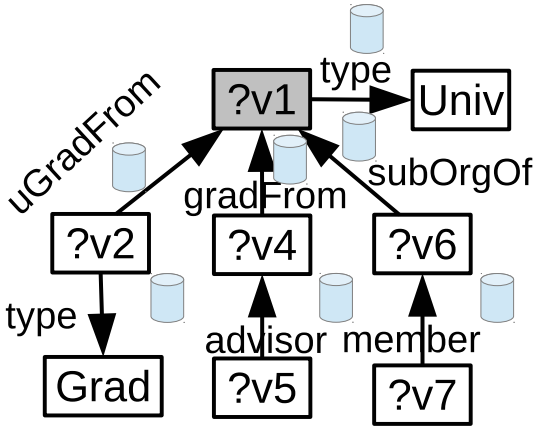
The pattern index is created and maintained by the replication controller at the master. It has the same structure as the heat map, but it only stores redistributed patterns. For example, Figure 4.7(b)(right) shows the pattern index state after redistributing all patterns in the heat map (Figure 4.5(c)). The pattern index is used



(a) Query.



(b) Redistribution Tree (left) and Pattern Index (right)



(c) Replica Index

Figure 4.7: A query and the pattern index that allows execution without communication.

by the query planner to check if a query can be executed without communication. When a new query Q is posed, the planner transforms Q into a tree T. If the root of T is also a root in the pattern index and all of T's edges exist in the pattern index, then Q can be answered in parallel mode; otherwise, Q is answered in distributed fashion. For example, the query in Figure 4.7(a) can be answered in parallel because its redistribution tree (Figure 4.7(b)(left)) is contained in the pattern index. Edges in the pattern index are time-stamped at every access to facilitate our eviction policy.

4.6.2 Replica index

The replica index at each worker is identical to the pattern index at the master and is also updated by the IRD process. However, each edge in the replica index is associated with a storage module similar to the one that stores the original data. Each module stores only the replicated data of the specified triple pattern. In other words, we do not add the replicated data to the main indices nor keep all replicated data in a single index. There are four reasons for this segregation. (i) As more patterns are redistributed, updating a single index becomes a bottleneck. (ii) Because of replication, using one index mandates filtering duplicate results. (iii) If data is coupled in a single index, intermediate join results will be larger, which will affect performance. Finally, (iv) this hierarchical representation allows us to evict any part of the replicated data quickly without affecting the overall system performance. Notice that we do not replicate data associated with triple patterns whose subjects are core vertices. Such data are accessed from the main index directly because of the initial subject-based hash partitioning. Figure 4.7(c) shows the replica index that has the same structure as the pattern index in Figure 4.7(b)(right). The storage module associated with ⟨?v7, member, ?v6⟩ stores replicated triples that match the triple pattern. Moreover, these triples qualify for the join with the triple pattern of the parent edge.

Searching and updating the pattern and replica indices is carried in the same way as for the heat map (see Algorithm 4). However, the findNode function (line 6) is changed to account for triple patterns with bounded subject/objects. Such triple patterns can have at most two matches: (i) an exact match, where all constants are matched; or (ii) a superset match, where both subject and object in the matching pattern are variables. If a triple pattern has two matches, the findNode function proceed with the superset matching branch because it will potentially benefit more queries in the future. This process is also implemented using hash lookups and hence has a complexity of $O(E)$, where E is the number of triple patterns in the query.

4.6.3 Conflicting Replication and Eviction

Conflicts may arise when a subquery appears at different levels in the pattern index. This may cause some triples to be replicated by the hot patterns that include them. This is not a correctness issue for AdPart as conflicting triples (if any) are stored separately using different storage modules. This approach avoids the burden of any housekeeping and existence of duplicates at the cost of memory consumption. Therefore, AdPart employs an LRU eviction policy that keeps the system within a given replication budget at each worker.

Recall that, each time an edge in the pattern index is accessed, its timestamp is updated. The search process in the pattern index is carried out in a top-down fashion. This means that the leaf nodes of the tree have the oldest timestamps. We store the leaves in a priority queue organized by timestamp. When eviction is required, the least recently used leaf and its matching replica index are deleted. Then, the parent of the evicted leaf is updated accordingly.

4.7 Experimental Evaluation

In this section, we evaluate the adaptivity feature of AdPart by comparing it against AdPart-NA and TriAD because they showed the best performance in the previous chapter. The hardware setup, datasets and queries are the same as the ones reported in Section 3.3. In Section 4.7.1, we conduct a detailed study of the effect and cost of AdPart's adaptivity feature. Then, in Section 4.7.2, we show the impact of adaptivity on the execution times of individual queries when compared to other systems. Finally, in Section 4.7.3, we study the data and machine scalability of AdPart. The results show that our system adapts incrementally to workload changes with minimal overhead without resorting to full data repartitioning. When the system adapts, it executes queries several orders of magnitude faster than other systems.

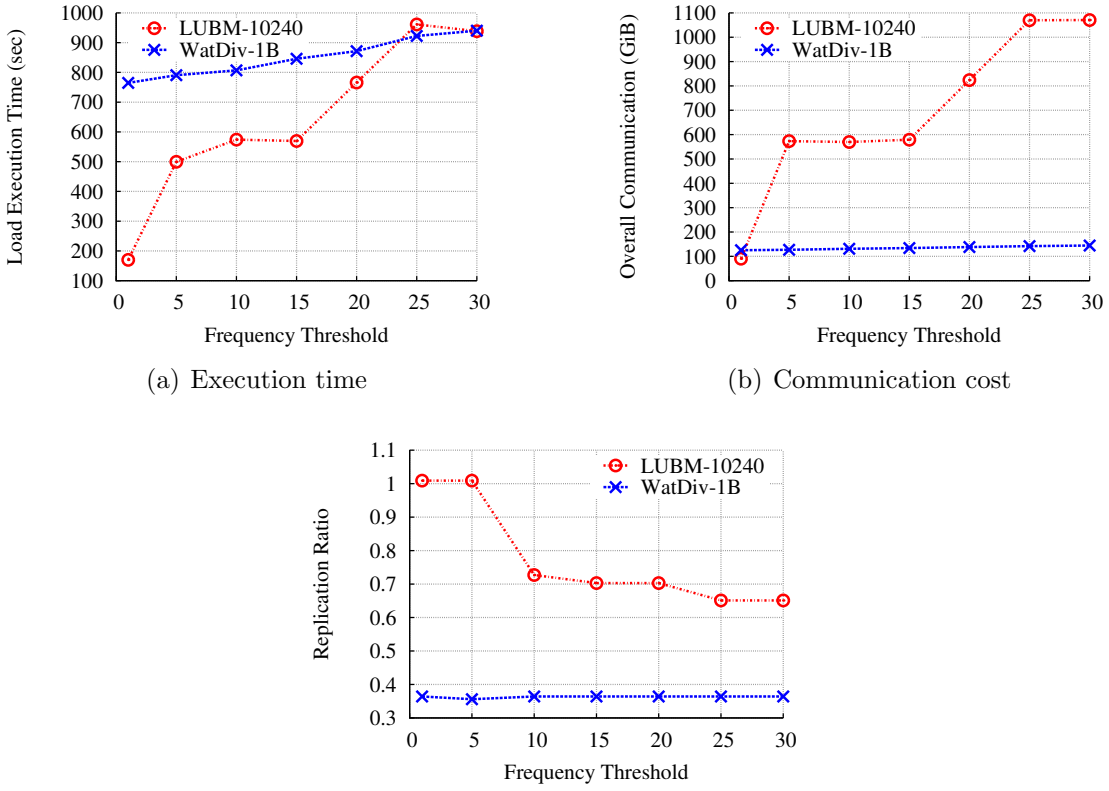
4.7.1 Workload Adaptivity by AdPart

In this section, we evaluate AdPart's adaptivity. For this purpose, we define different workloads on two billion-scale datasets that have different characteristics, namely, LUBM-10240 and WatDiv-1B.

WatDiv-1B workload: We used the benchmark query generator to create a 5K-query workload from each query category (i.e., L, S, F and C), resulting in a total of 20K queries. Also, we generate a random workload by shuffling the 20K queries.

LUBM-10240 workload: As AdPart and the other systems do not support inferencing, we used all 14 queries in the LUBM benchmark without reasoning[4]. From these queries, we generated 10K unique queries that have different constants and structures. We shuffled the 10K queries to generate a random workload which we used throughout this section. This workload covers a wide spectrum of query complexities including simple selective queries, star queries as well as queries with complex structures and

[4]Only query patterns are used. Classes and properties are fixed so queries return large intermediate results.



(a) Execution time

(b) Communication cost

(c) Replication cost

Figure 4.8: Frequency threshold sensitivity analysis.

low selectivities.

Frequency Threshold Sensitivity Analysis

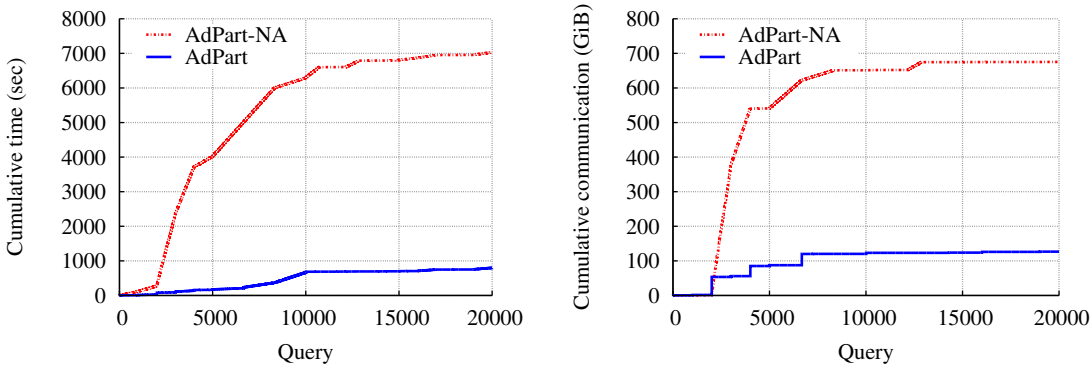
The frequency threshold controls the triggering of the IRD process. Consequently, it influences the execution time and the amount of communication and replication in the system. In this experiment, we conduct an empirical sensitivity analysis to select the frequency threshold value based on the two aforementioned query workloads. We execute each workload while varying the frequency threshold values from 1 to 30. Note that our frequency monitoring is not on a query-by-query basis as our heat map monitors the frequency of the subquery pattern in a hierarchical manner (see Section 4.5). The workload execution times, the communication costs and the resulting replication ratios are shown in Figures 4.8(a), 4.8(b) and 4.8(c), respectively.

We observe that LUBM-10240 is very sensitive to slight changes in the frequency threshold because of the complexity of its queries. As the frequency threshold increases, the redistribution of hot patterns is delayed causing more queries to be executed with communication. Consequently, the amount of communication and synchronization overhead in the system increases, affecting the overall execution time, while the replication ratio is low because fewer patterns are redistributed.

On the other hand, WatDiv-1B is not as sensitive to this range of frequency thresholds because most of its queries are solved in subseconds using our locality-aware DSJ, without excessive communication. Nevertheless, as the frequency threshold increases, the synchronization overhead affects the overall execution time. Furthermore, due to our fine-grained query monitoring, AdPart captures the commonalities between the WatDiv-1B query templates for frequency thresholds 5 to 30. Hence, for all these thresholds the replication ratio remains almost the same. However, the system converges faster for lower threshold values, reducing the overall execution time. In all subsequent experiments, we use a frequency threshold of 10; this results in a good balance between time and replication. We plan to study the auto-tuning of this parameter in the future.

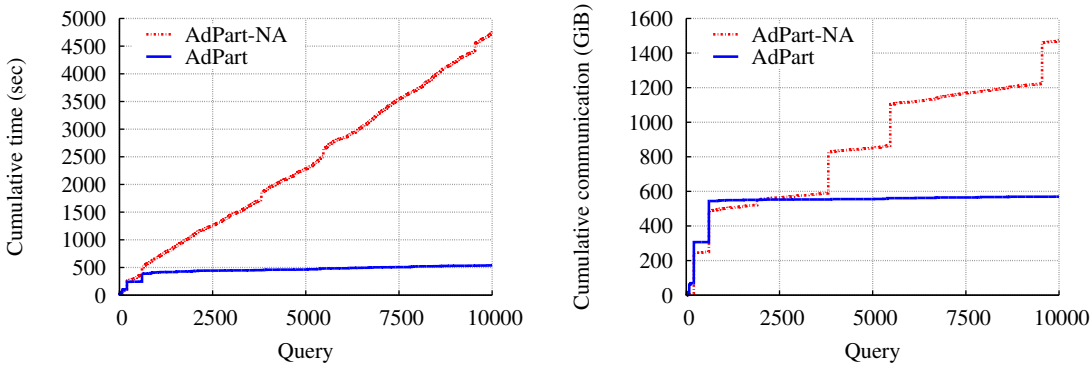
Workload Execution Cost

To simulate a change in the workload, queries of the same WatDiv-1B template are run consecutively while enforcing a replication threshold of 20%. Figure 4.9(a) shows the cumulative time as the execution progresses with and without the adaptivity feature. After every sequence of 5K query executions, the type of queries changes. Without adaptivity (i.e., AdPart-NA), the cumulative time increases sharply as long as complex queries are executed (e.g., from query 2K to query 10K). On the other hand, AdPart adapts to the workload change with little overhead causing the cumulative time to drop significantly by almost 6 times. Figure 4.9(b) shows the cumulative



(a) Execution time (b) Communication cost

Figure 4.9: AdPart adapting to workload (WatDiv-1B).



(a) Execution time (b) Communication cost

Figure 4.10: AdPart adapting to workload (LUBM-10240).

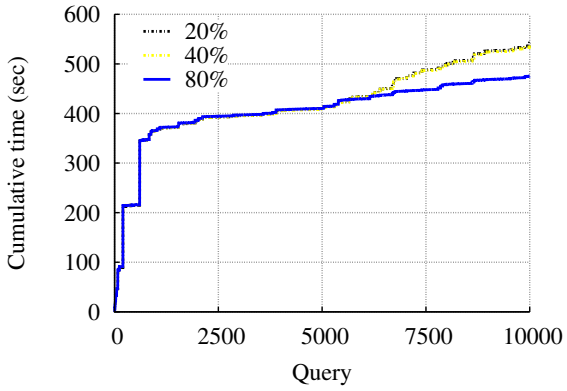
communication costs of both AdPart and AdPart-NA. As we can see, the communication cost exhibits the same pattern as that of the runtime cost (Figure 4.9(a)), which proves that communication and synchronization overheads are detrimental to the total query response time. The overall communication cost of AdPart is more than 7X lower compared to that of AdPart-NA. Once AdPart starts adapting, most of future queries are solved with minimum or no communication. The same behavior is observed for the LUBM-10240 workload (see Figures 4.10(a) and 4.10(b)).

Partitioning based on a representative workload: We tried to use Partout [35] to partition the LUBM-10240 and WatDiv-1B datasets based on a representative workload. However, it could not finish within reasonable time (<3 days) even for small work-

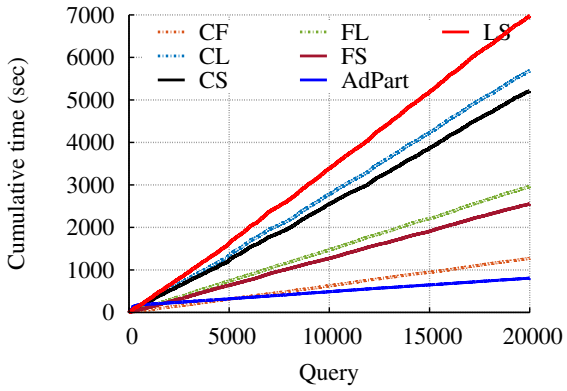
loads. Thus, in this experiment, we simulate two scenarios for workload-based data partitioning using AdPart. First, we assume the availability of a representative workload and measure how the training workload size affects performance. Second, we assume the data is partitioned using a workload that does not fully represent future queries. In both scenarios, there are two phases: training and testing. In the training phase, the adaptivity feature is enabled and the system can perform data redistribution for detected hot patterns. In the test phase, the adaptivity feature is disabled.

In the first scenario, we use a random workload of 10K LUBM queries where the first N% queries are used for training. The remaining queries are used for testing. Figure 4.11(a) shows how AdPart's performance changes as the size of the training window increases from 20% to 80%. With larger window sizes, more hot patterns are detected and redistributed in the training phase. Consequently, more queries in the test phase are solved without communication. Notice that, even with 20% queries, AdPart could detect most of the hot patterns in the workload and adapt accordingly. As a result, there is no significant difference between the total workload execution time when using 80% and only 20% training queries. This concludes that when a representative workload is available, systems that perform static workload-based partitioning like, Partout and WARP, can perform reasonably well for all workload queries.

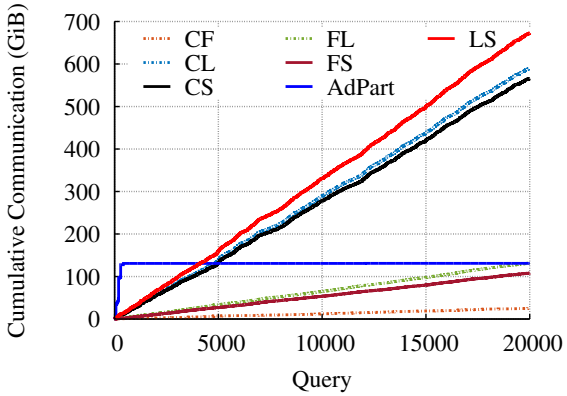
We further investigate another scenario where future queries are not well represented by the partitioning workload. The test set includes query patterns from the training query set as well as new queries that were not seen before. To do so, we train AdPart using different combinations of the workload categories defined by WatDiv-1B (C, F, S, and L). Each combination is made of two categories (10K queries); effectively producing six combinations, mainly CF, CL, CS, FL, FS, and LS. The test set includes 20K random queries made up from the four query categories. This way,



(a) Representative workload



(b) Workload changes (time)



(c) Workload changes (Communication)

Figure 4.11: Comparison with workload-based partitioning.

some of the queries in the test workload would run in parallel while others (not in the representative workload) would require communication.

Figures 4.11(b) and 4.11(c) show the cumulative execution time and communication, respectively, for the test workloads (i.e., excluding the training time). For example, we train the system with the adaptivity feature enabled using 10K queries from two categories, like CF. Then, we test the system using 20K random queries while adaptivity is disabled. Obviously, the performance of the test workload highly depends on the complexity of the queries used in the training phase. For example, the complex (C) and snowflake (F) queries are the most expensive queries in the benchmark. Therefore, when the system is trained using the CF training workload, it performs much better than when trained using the LS workload. CF workload requires less communication because the L and S queries (not in the training workload) do not require excessive data exchange. Nonetheless, the CF execution time keeps increasing due to the existence of communication and synchronization overheads. In the same figures, we show the performance of AdPart without training, but the adaptivity is enabled all the time. Allowing the system to adapt incrementally and dynamically (without training) resulted in better performance when compared to all cases. AdPart incurs more communication at the beginning because of the IRD process; it then converges to almost constant communication.

Next, we test AdPart's performance using a real scenario workload where a certain percentage of the queries is repeated while other new queries are taken into account. We use three workloads, each workload contains 10K LUBM random queries out of which a certain percentage is repeated. Figure 4.12 shows AdPart's performance while varying the amount of repeated queries between 20%, 40% and 80%. As the results suggest, the more the repeated queries, the less the workload execution time. Since AdPart monitors the query patterns and not the individual queries, it could capture most of the patterns in the workload even with only 20% of its queries repeated.

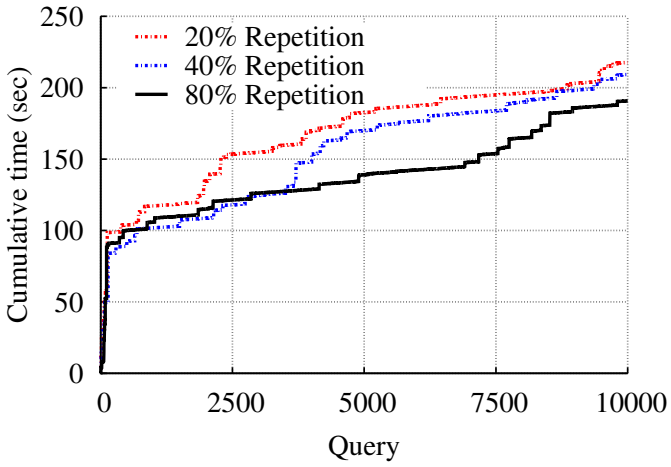
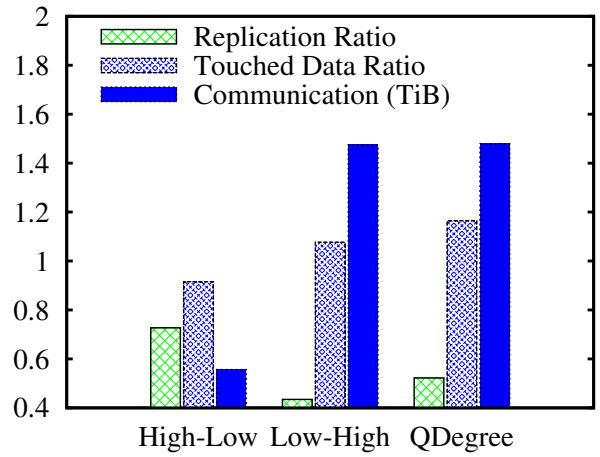


Figure 4.12: Effect of query repetition.

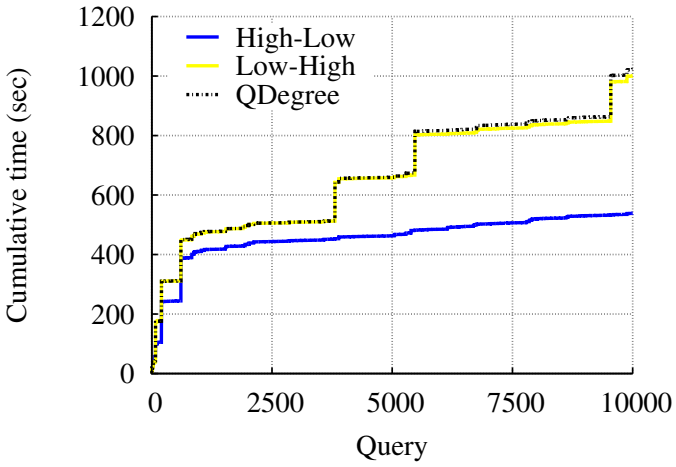
Redistribution Tree Generation

In this experiment, we evaluate our query transformation heuristic (Section 4.2) against two alternative approaches. Recall that when transforming a hot query pattern into a redistribution tree, we select the vertex with the highest score to be the tree root. Then, the query is traversed from high score vertices to lower score ones. We now compare our heuristic (referred to High-Low hereafter) to two different heuristics: (i) in Low-High, the vertex with the least vertex score is selected as core; then the query pattern is traversed by exploring vertices with lower scores first. The (ii) QDegree approach uses a different vertex scoring function where the score of a vertex in the hot query pattern is its out-degree. The pattern is then traversed from high score vertices to lower score ones. Note that the latter approach aims at minimizing the replication in a greedy manner by fully exploiting the initial hash partitioning. Recall that data that binds to triple patterns whose subject is a core are not replicated.

We evaluated all these heuristics by running the LUBM-10240 workload. In Figure 4.13(a), we show the resulting replication, the communication cost and the amount of data touched by the IRD process. Low-High and QDegree resulted in slightly less replication compared to High-Low. The reason is that both heuristics benefit from



(a) Replication and Communication cost



(b) Execution time

Figure 4.13: Effect of hot pattern transformation.

the initial hash partitioning by selecting cores with larger number of outgoing edges. However, the amount of data touched by IRD (i.e., data in the main and replica indices) in Low-High and QDegree is significantly higher. This affects adaptivity's performance because IRD is carried out using a series of DSJ iterations. Furthermore, as the data touched by the process is actually used for evaluating parallel queries, the performance of parallel queries is eventually affected.

Consequently, the cumulative workload execution time using High-Low is 1.9X faster than the other heuristics as shown in Figure 4.13(b). Since QDegree and

Table 4.2: Load Balancing in AdPart

Dataset	Percentage of triples				Replication Ratio
	Max	Min	Average	StDev (σ)	
LUBM-10240	1.43%	1.35%	1.39%	0.02	0.73
WatDiv-1B	1.58%	1.20%	1.33%	0.07	0.36

Low-High touch and communicate almost the same amount of data, their cumulative execution times are also the same. Besides, note that QDegree does not use any statistical information from the data and only relies on the structure of the hot query pattern. Therefore, a redistributed pattern would not benefit other future queries with a slightly different structure. We repeated the experiment on WatDiv-1B and all heuristics resulted in almost the same communication cost, wall time, and touched data. This time, QDegree resulted in the least replication because its exploits best the initial subject-based hash partitioning.

Replication and Load Balance

In this experiment, we evaluate the load balance of AdPart from two different perspectives: (i) data balance, i.e., how balanced is the initial partitioning as well as the replication that results from the IRD process; (ii) work balance, i.e., how the evaluation cost is balanced among all workers in the system, during the execution of the workload. In Table 4.2, we report some statistics that characterize the data load balance in AdPart. Particularly, we report the average and standard deviation (σ) of the percentage of triples stored at each worker. As shown in the table, AdPart achieves very good data balance for both workloads because of the initial subject-based hash partitioning as well as the hashing used during the IRD process. Also, we report how the average partition size changes during the workload execution. Using the 10K queries LUBM workload, Figure 4.14 shows how the partition size increases as more queries are executed. Initially, each partition contains around 19M triples.

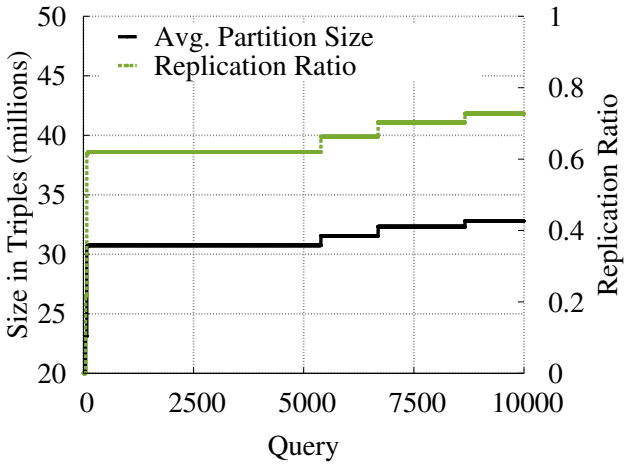
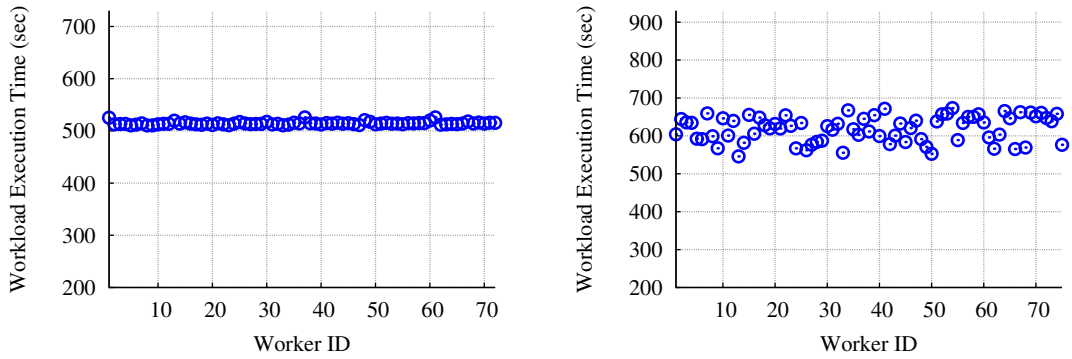


Figure 4.14: Evolution of partition size.



(a) LUBM-10240 (b) WatDiv-1B

Figure 4.15: Workload balance.

This corresponds to a 0% replication ratio as AdPart loads only the original dataset. As the system adapts, the size of each partition slightly increases till reaching an average size of around 33M triples; which counts for a 72% replication ratio after executing the whole 10K workload queries. Work is also well balanced among workers; i.e., the amount of work contributed by each worker is almost the same as shown in Figures 4.15(a) and 4.15(b) for the LUBM-10240 and WatDiv-1B, respectively.

Table 4.3: Query runtimes for LUBM-10240 (ms)

LUBM-10240	L1	L2	L3	L4	L5	L6	L7	Geo-Mean
AdPart	**317**	**120**	**6**	**1**	**1**	**4**	**220**	**15**
AdPart-NA	2,743	**120**	320	**1**	**1**	40	3,203	75
TriAD	6,023	1,519	2,387	6	4	114	17,586	369
TriAD-SG	5,392	1,774	4,636	9	5	10	21,567	333

4.7.2 Query Performance

In the previous sections, we showed how adaptivity could reduce the execution time of the entire workload. In this section, we demonstrate the effectiveness of the adaptivity in reducing the individual query execution wall time by comparing AdPart against AdPart-NA and TriAD. Tables 4.3, 4.4, 4.5 and 4.6 show the performance of AdPart for LUBM-10240, WatDiv, YAGO2 and Bio2RDF datasets, respectively.

Table 4.4: Query runtimes for WatDiv (ms)

WatDiv-100	Machines	L1-L5	S1-S7	F1-F5	C1-C3
AdPart	5	**2**	**2**	**7**	**22**
AdPart-NA	5	9	7	160	111
TriAD	5	4	15	45	170

Table 4.5: Query runtimes for YAGO2 (ms)

YAGO2	Y1	Y2	Y3	Y4	Geo-Mean
AdPart	**3**	**19**	**11**	**2**	**6**
AdPart-NA	19	46	570	77	79
TriAD	16	1,568	220	18	100

For all datasets, once AdPart adapts to the workload, it executes all queries much faster than all other systems. More importantly, queries that require object-object joins (Y3 in YAGO2 and B1 in Bio2RDF), which cannot be executed efficiently by AdPart-NA, are executed faster by AdPart. For star queries that join on subjects (L2, L4 and L5 in LUBM-10240), both AdPart-NA and AdPart perform equally because of the initial subject-based hash partitioning. In other words, these queries are solved in parallel without communication without the need for adaptivity.

Table 4.6: Query runtimes for Bio2RDF (ms)

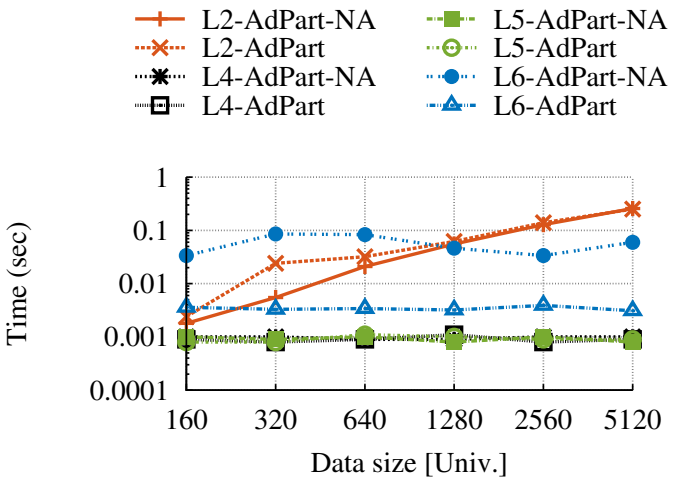
Bio2RDF	B1	B2	B3	B4	B5	Geo-Mean
AdPart	**3**	**2**	**2**	**3**	**1**	**2**
AdPart-NA	17	16	32	89	1	15
TriAD	4	4	5	90	2	7

For L3 in LUBM-10240, AdPart can detect queries with empty results during planning. As each worker makes its local parallel query plan, it detects the zero cardinality of the subquery in the replica index and terminates. This explains the several orders of magnitude gain in query response time.

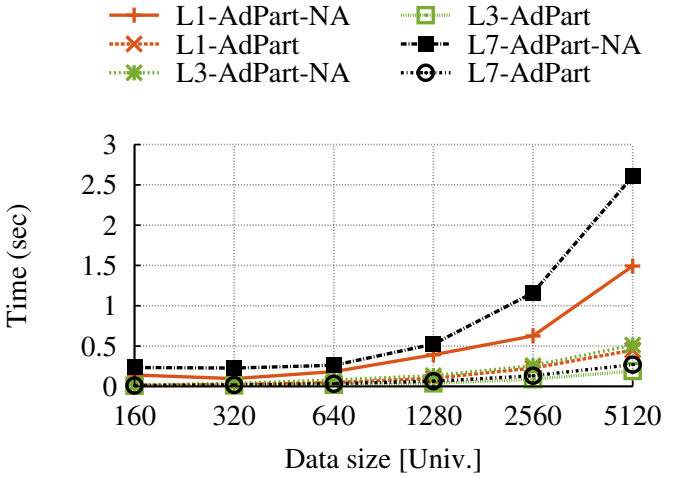
4.7.3 Scalability

Data Scalability. We use the LUBM benchmark data generator to generate six datasets: LUBM-160, LUBM-320, LUBM-640, LUBM-1280, LUBM-2560 and LUBM-5120. We keep the number of workers fixed to 72 (6 workers per machine). Figures 4.16(a) and 4.16(b) show the data scalability of AdPart and AdPart-NA for simple and complex queries respectively. L4, L5, L6 are simple queries that are very selective and touch the same amount of data regardless of the data size. This describes the steady performance of both AdPart and AdPart-NA for these queries. Because L2 is not selective and returns massive final results, it is inevitable for its scalability to degrade as data size increases. Figure 4.16(b) shows the scalability of AdPart for complex queries. Queries L1 and L7 generate large number of intermediate results causing high communication cost, which explains their poor scalability of AdPart-NA. Nevertheless, as AdPart adapts to the workload, many queries are evaluated in parallel mode much faster.

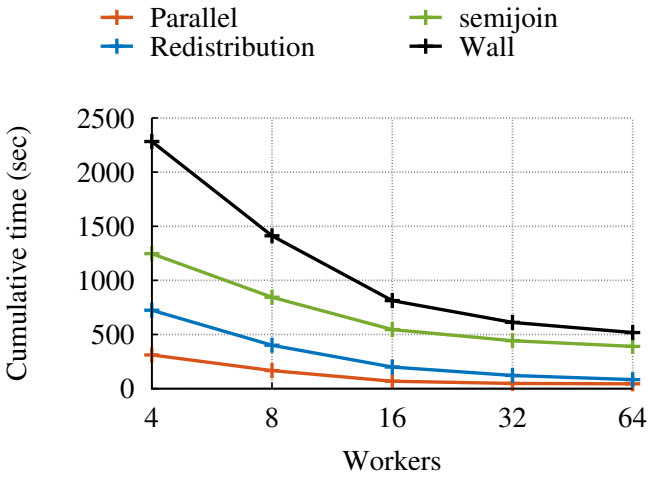
Strong Scalability. In this experiment, we use the 10K workload of LUBM-10240 to demonstrate the strong scalability of AdPart. We fix the workload while increasing the number of workers. Figure 4.16(c) shows the wall time for executing the workload.



(a) Data scalability (simple)



(b) Data scalability (complex)



(c) Strong Scalability

Figure 4.16: AdPart scalability using LUBM dataset.

The time is split into the three constituents of AdPart execution, i.e, distributed execution (semijoin), redistribution (adaptivity) and parallel queries. All components of AdPart scale very well for up to 32 workers, afterwards the overhead of the semijoin communication starts dominating. Note that solving complex queries, like L1, L2, and L7 in parallel mode scale almost optimally. On the other hand, selective queries that touch very few data or are executed by a single worker do not scale. For future work, we will investigate the possibility of exploiting subjects and objects locality to further scale the distributed semijoin to more workers.

Chapter 5

RDF Analytics Framework

> Everything should be made as
> simple as possible, but not simpler.
>
> Albert Einstein
> 1879 — 1955 CE

This chapter introduces SPARTex and RDF analytics framework. Rich RDF analytics is realized in SPARTex by introducing the following features:

SPARQL Extension. SPARTex defines a Graph Analytics extension of SPARQL (GAS-parql) that allows generic User-Defined Procedures (UDPs) to be intermixed and executed in a pipeline together with SPARQL queries. A UDP can be any program implemented in the vertex-centric model (e.g., PageRank, Shortest-Paths, Centrality). UDPs communicate with SPARQL at the granularity of a vertex, by setting vertex properties, which is equivalent to updating the RDF graph. SPARTex also allows filters that limit the scope of the UDP input, where the filter is nothing but a separate SPARQL query. In other words, a UDP can operate on a subset of the input graph that results from evaluating a SPARQL query.

SPARQL Engine. a SPARQL query engine is implemented as a vertex-centric program, allowing UDPs and SPARQL queries to run on top of the same vertex-centric framework. The SPARQL operator leverages the message-passing nature of the vertex-centric frameworks for join evaluation. Given a SPARQL query Q, the op-

erator has two stages: (i) a cost-based optimizer picks a trail[1] on Q to minimize the number of messages generated when the trail is followed on the data graph. (ii) Once a trail is picked, vertices exchange messages with their neighbors along the picked trail. This is done usually using multi-rounds of message-passing among vertices.

In-memory Data Store. The underlying vertex-centric frameworks typically store the data as a generic graph in memory. SPARTex extends this with a per-vertex data store that is tailored to RDF data. It efficiently filters the neighbors of vertices by specific predicates. The data store also allows updates on the RDF graph by attaching properties to vertices without changing the original data layout and indices.

With these features running on a unified framework, SPARTex introduces a new and rich type of RDF analytics that were not feasible before: (i) Graph algorithms and SPARQL can be executed efficiently on the same framework. Hence, there is no need to use different systems or materialize and reformat intermediate results. (ii) Original RDF data and vertex computed values can be combined as a single subgraph pattern in SPARQL. Triple patterns in the body of SPARQL queries can be from the structure of the input data or derived from new values computed per vertex (e.g. see query Q_s in Figure 1.4). (iii) Generic graph algorithms and SPARQL queries can be pipelined so that the output of one operator is the input to another. For example, the Single Source Shortest Path algorithm can start from the vertices that match a specific SPARQL pattern. Different operators can share intermediate results using the in-memory data store. All the aforementioned analytical tasks can be triggered declaratively using GASparql.

5.1 System Architecture

SPARTex is designed to be built on top of distributed vertex-centric bulk synchronous

[1]It is sufficient to consider a trail to be a walk or a path on the query graph such that each edge is visited at least once. More rigorous definition is given in Section 5.3

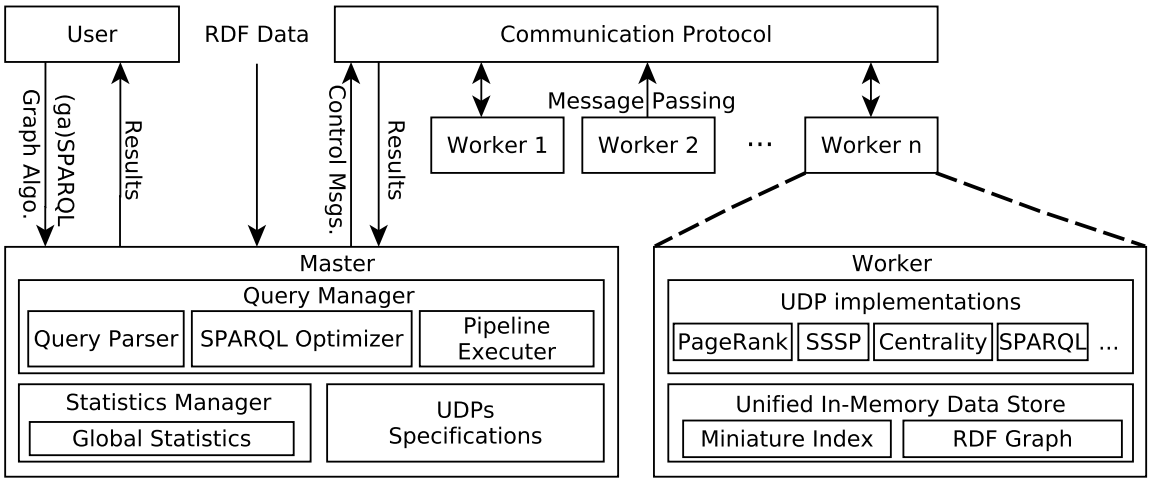


Figure 5.1: SPARTex Architecture.

graph processing frameworks, such as Pregel [49], Trinity [48] and GraphLab [51], that can process very large graphs. Briefly, in these vertex-centric systems, users define a generic compute function that will be executed on each vertex independently. Vertices interact with each others through message passing. A typical vertex-centric program consists of a number of iterations. In each iteration, a vertex can perform computation, change its state, and send messages to its neighbors. Typically, vertex-centric frameworks are coupled with a distributed file system to persist data such as the input graph. SPARTex is built on top of GPS [113], an open-source Pregel clone.

An overview of SPARTex is depicted in Figure 5.1. SPARTex follows the master-slave architecture. Users can write vertex-centric programs for any graph algorithm. Programs are compiled and added to the classpath. SPARTex treats these programs as user-defined stored procedures that can be invoked using GASparql (Section 5.2). The master keeps an entry for each registered UDP which includes the class name and expected input parameters. In addition, SPARTex provides an efficient vertex-centric SPARQL operator (Section 5.3). Having both SPARQL and graph algorithms within the same framework, SPARTex allows both operations to be executed in a pipelined fashion. In the rest of this section, each component of SPARTex is briefly described.

5.1.1 Master

The master is not assigned any portion of the input graph; rather it orchestrates workers activity. Users submit SPARQL queries which may or may not have UDPs to the master. The master parses the query and generate an execution plan for the whole pipeline (i.e. for generic algorithms and pattern matching). The master enforces the plan using a compute function that is executed before starting any iteration. Finally, SPARTex returns the results back to the user. Next, each of the master components is defined.

Query Manager. The query manager is responsible for parsing, optimizing and executing incoming queries. The Query Parser parses the input query and separates the procedural constructs from the declarative patterns of SPARQL. The query manager checks the existence of the called procedures and the consistency of their parameters by consulting the set of UDPs specifications. Then, the pattern matching part is optimized using a SPARQL optimizer. It enumerates possible execution plans and estimate their costs using global statistics maintained in the statistics manager. Finally, the query manager consolidates the procedural part and the optimized pattern matching query plan into a global pipelined execution plan. The Pipeline Executer gets as input a set of required steps for query evaluation. It dictates which vertex-centric program to run for how many iterations or until the program converges if the number of iteration cannot be known a priori. When executing multiple vertex-centric programs, the pipeline executer directs workers to change the UDP to be executed in the next n iterations. After that, all vertices return to active state and the executer sends the next UDP to be executed and so on.

Statistics Manager. The master gathers some global statistics during the RDF graph loading phase which spans the first three compute iterations. In the first iteration, each worker loads/indexes its assigned vertices and their edges. In the second iter-

ation, vertices report statistics about their neighbors and predicates. For example, each vertex counts the number occurrences for each of its attached predicates. It also counts the number of occurrences of subjects and objects attached with these predicates. It also capture more advanced statistics about the correlation between different predicates (more details about the collected statistics are shown in Section 5.3.2) At the end of this iteration, each worker synchronize its collected statistics with the master. In the third iteration, the master retrieves all the statistics collected from all workers, integrates them and store it in a global structure that is kept at the master.

UDPs Specifications. This UDP specification structure contains meta-data about programs that are available in the framework. When a UDP is registered in SPARTex, an entry of it is recorded in this structure and kept at the master.

5.1.2 Worker

Vertex-centric frameworks divide the data graph into partitions where each partition contains a set of data vertices and edges. A vertex v with its outgoing edges are assigned by default to a machine M based on the result of a simple hashing scheme $M_(v \ mod \ k)$, where k is the number of partitions. The default partitioning scheme is modified such that each vertex has both its incoming and outgoing edges[2].

Unified In-Memory Data Store. Generic graph algorithms and SPARQL access data differently. While SPARQL needs to access both incoming and outgoing edges using predicate labels, algorithms like PageRank need to access the outgoing edges only regardless of their labels. Therefore, rich RDF analytics requires modeling the data in a uniform way while providing different data access methods. Specifically, the framework needs to support: (i) label-based neighbor access used for SPARQL query evaluation. (ii) Label-oblivious neighbor access used for algorithms that access the

[2]This is equivalent to partitioning on both subject and object vertices.

RDF data regardless of the edge labels; and (iii) adding, deleting and updating vertex properties. Since computation is done at the vertex granularity, a set of miniature data indices per vertex are created. These indices are accessed through a set of API calls.

Miniature RDF Data Index. it consists of the following two indices: (i) Predicate-Object (PO) Index: given an edge predicate p, returns a list of all the outgoing neighbors (objects). (ii) Predicate-Subject (PS) Index: given an edge predicate p, returns a list of all the incoming neighbors (subjects).

Miniature Properties Store. Each vertex maintains an in-memory key-value store where different algorithms can delete, add or update a vertex property. This way the result of one algorithms can be read by others; enabling pipelined execution of graph algorithms.

UDP implementations. It contains the same meta-data stored at the UDP specifications structure in the master. However, it also contains the actual implementation of the registered vertex-centric programs (e.g. PageRank, SSSP). It is used by the worker to switch between different UDPs at runtime when directed by the master. The worker receives a message from the master that includes the configuration of the next UDP and act accordingly.

5.2 Graph Analytics SPARQL Extension

In this section, Graph Analytics SPARQL (GASparql) extension is introduced. GASparql gives users 3 capabilities: (i) users can write their own algorithms in a procedural language and invoke it from within SPARQL as a user-defined stored procedure. (ii) Users can materialize the computation results as vertex properties. These properties can then be used as input to SPARQL or another graph algorithm; and (iii) users can mutate the original RDF graph by adding/deleting properties as needed.

5.2.1 GASparql Constructs

In this section, new constructs of GASparql are described and illustrated with examples for their usage.

Defining/Calling UDPs

The first construct of GASparql allows users to call/define an already implemented stored procedure. To do so, a user can write the following:

```
CALL proc(list[params]) AS list[properties]
```

The above code calls the procedure proc by specifying its full qualified class name. list[params] is the set of parameters that the procedure expects while list[properties] is the set of vertex properties that proc will add to the RDF data.

As an example, recall that Q_s in Figure 1.4 requires evaluating PageRank algorithm and materializing its results. To do so, a user can write the following:

```
PREFIX  sptx: <http://www.spartex.com/analytics/>
CALL com.sptx.algo.PageRank(max_iter) AS sptx:pRank
```

The PageRank algorithm expects as input the maximum number of iterations. The output is a sptx:pRank value per vertex. The new vertex property (sptx:pRank) is added to the Properties key-value store, where the key is sptx:pRank and the value is the PageRank of the vertex. Notice that an entry in the properties store is actually a triple, where the subject is the vertex itself, the predicate is the property name and the object is the property value. The new added triples can be used later within a SPARQL query (see Section 5.2.2). Similarly for Q_s, the centrality algorithm can be invoked and its result is materialized per vertex. Notice that storing newly computed vertex properties does not require any change to the data layout or indices.

Data Filters

In the previous example, the PageRank algorithm runs on the entire RDF graph. However, there are cases where an algorithm should run only on a subset of the graph. Hence, GASparql introduces a filtering constructs based on the vertices and edges of the RDF graph. Invoked procedures are optionally associated with one or more filters:

```
FILTER_VERTEX AS filter_name WHERE { BGP }
FILTER_EDGE AS filter_name WHERE { BGP }
```

All triple patterns of the Basic Graph Pattern (BGP) in the WHERE clause must have a common vertex. In other words, the BGP is a star query around a specific vertex[3]. For FILTER_VERTEX, vertices that do not match the BGP are filtered out. Similarly, all edges that do not satisfy the BGP pattern of FILTER_EDGE are filtered out. Filters are passed to the stored procedures through the keyword using. Filtering constraints are associated with procedure calls and acts as filtering layer on top of the unified data store. Only data that satisfy the filtering constraints are retrieved.

For example, the objects of triples with rdf:type predicates have lots of incoming edges; hence, would have extremely high PageRank values. Excluding these triples when running the PageRank algorithm can be done as follows:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
FILTER_EDGE AS no_type WHERE {
  ?s ?p ?o .
  FILTER(!sameterm(?p, rdf:type))
}
CALL com.sptx.algo.PageRank(max_iter) USING no_type AS sptx:pRank
```

[3]More sophisticated filtering can be achieved by combining the FILTER and ADD PROPERTY (Section 5.2.1) constructs (e.g. see Section 5.2.2)

Managing Vertex Properties

So far, vertex properties are set or deleted by the stored procedures. However, users may want to deliberately set or delete some vertex properties. Therefore, GASparql introduces two constructs for explicit vertex properties addition and deletion.

```
ADD PROPERTY {list[property patterns]} WHERE {BGP}
DROP PROPERTY {list[property patterns]} WHERE {BGP}
```

For example, to drop the sptx:pRank property that is less than a specific threshold, a user can write:

```
DROP PROPERTY {?x sptx:pRank ?val} WHERE{
     ?x sptx:pRank ?rank .
     FILTER(?rank < threshold)
}
```

5.2.2 RDF Analytics Applications

In this section, three RDF analytical applications that make use of the proposed extension are discussed.

Using Graph Analytics Output in SPARQL

Consider Q_s in Figure 1.4, it returns the set of students who take courses taught by their advisors. Assume the query results needs to be restricted to only popular professors and core courses, where PageRank and centrality indicate professor popularity and course importance, respectively. Q_s can be expressed as:

```
PREFIX  sptx: <http://www.spartex.com/analytics/>
CALL com.sptx.algo.centrality() AS sptx:centrality
CALL com.sptx.algo.PageRank(max_iter) AS sptx:pRank
SELECT ?s WHERE {
    ?p    teaches  ?c .
    ?s    takes    ?c .
    ?s    advisor  ?p .
    ?p    sptx:pRank        ?rank .
    ?c    sptx:centrality   ?cent .
```

```
    FILTER (?rank > val1 && ?cent > val2)
}
```

SPARTex starts by executing the centrality and PageRank algorithms. The sptx:pRank and sptx:centrality properties are set for all vertices. Then, SPARTex plans and executes the subgraph pattern matching part of the query; only vertices the satisfy the filter constraints are retrieved.

Using SPARQL Output in Graph Analytics

In some cases, a general graph algorithm is supposed to operate on a specific part of the data. This can be achieved by using the result of SPARQL as an input to the subsequent general graph algorithm. For example, consider Q_s in the previous example; and suppose that the shortest path between popular professors (i.e. vertices matching $?p$) and every other vertex is to be found. This can be done by executing the Single Source Shortest Path (SSSP) algorithm starting from these professors as follows:

```
PREFIX  sptx: <http://www.spartex.com/analytics/>
CALL com.sptx.algo.centrality()  AS sptx:centrality
CALL com.sptx.algo.PageRank(max_iter) AS sptx:pRank
ADD PROPERTY {?p sptx:popular "T" . } WHERE {
    ?p    teaches  ?c .
    ?s    takes    ?c .
    ?s    advisor  ?p .
    ?p    sptx:pRank        ?rank .
    ?c    sptx:centrality   ?cent .
    FILTER (?rank > val1 && ?cent > val2)
}
FILTER_VERTEX AS start WHERE {
  ?p sptx:popular "T" .
}
CALL algo:SSSP() USING start AS sptx:sssp
```

Using the add property construct, popular professors are identified by setting their sptx:popular property as `true`. Then, a vertex filter is created to exclude all vertices

not satisfying this property. Finally, the filter is associated with the SSSP procedure call so the algorithm only starts from vertices that match the defined filter.

Sampling RDF Graphs

SamplD [44] is a pipeline of graph processing steps for sampling RDF graphs. Given an input graph, SamplD applies a set of graph operations using Apache PIG [114] and Giraph [115]. It transforms the RDF graph into a directed unlabeled graph and analyzes the rewritten graph using degree centrality and PageRank algorithms. Then, each triple is assigned a score and triples with the highest scores are selected to form a smaller sample of the input graph.

SamplD pipeline steps require circulating the graph and using multiple programming platforms. The code below shows how SamplD pipeline can be implemented using SPARTex only and its extension; GASparql.

```
CALL com.sptx.algo.centrality() AS sptx:centrality
CALL com.sptx.algo.PageRank(max_iter) AS sptx:pRank
CALL com.sptx.algo.SamplDRankTriples()
```

SPARTex starts by invoking both the degree centrality and PageRank algorithms and materializing their results in-memory. Then, it executes a vertex-centric program (SamplDRankTriples) that consists of two iterations. In the first iteration, objects send their PageRank and centrality values to their subjects. Then, in the last iteration, subjects receive the object values; compare them to their values and output the triples with their scores.

5.3 SPARQL Query Engine

This section presents the vertex-centric SPARQL operator incorporated in SPARTex. Consider query $\overline{Q_s}$ defined by the solid lines in query Q_s (See Figure 1.4). $\overline{Q_s}$ consists of 3 triple patterns: $q_1 : \langle$?p, teaches, ?c$\rangle$, $q_2 : \langle$?s, advisor, ?p$\rangle$ and $q_3 : \langle$?s, takes, ?c$\rangle$.

In the relational model, $\overline{Q_s}$ is answered[4] by scanning the data to find the matches of each triple pattern. Then, the intermediate results are joined to formulate the final results. However, relational approaches are not suitable for SPARTex due to its vertex-centric nature. Specifically, data, computation and communication are all vertex-centric. Employing a relational approach for query evaluation defeats the purpose of SPARTex. Therefore, SPARTex embarks on a network-based approach (graph exploration) that uses inter-vertex message passing for query evaluation.

5.3.1 Query Evaluation

Formally, given a query graph Q, the cost-based optimizer selects a trail on Q that traverses each edge at least once. A trail consists of a set of ordered exploration edges $\{\bar{q}_1, \ldots, \bar{q}_n\}$. An exploration edge $\bar{q}_i$ is defined as (v_e, p, v_t, *direction*), where v_e and v_t are vertices in the query graph and p is the edge label. The direction is either outgoing or incoming relative to $\bar{q}_i.v_e$ in the query graph. v_e and v_t are referred to as exploration vertex and termination vertex, respectively. The termination vertex of $\bar{q}_i$ is the exploration vertex of $\bar{q}_{i+1}$. For example, a possible trail in $\overline{Q_s}$ that starts from ?p is $\{\bar{q}_1, \bar{q}_2, \bar{q}_3\}$={(`?p`, *teaches*, `?c`, *out*), (`?c`, *takes*, `?s`, *in*), (`?s`, *advisor*, `?p`, *out*)}. Obviously, there are many potential trails that can start from any of the query vertices. Query planning is discussed in Section 5.3.2.

A query is evaluated using $n + 1$ iterations, where n is the number exploration edges in the trail. Each edge is explored in an iteration; the final iteration is needed for reporting the query results. In every iteration, each vertex executes ExploreEdge in Algorithm 5. The inputs for the algorithm are the exploration edge $\bar{q}_i$, the messages received from the previous iteration, and the current iteration number. As a running example, query $\overline{Q_s}$ is evaluated using the previous trail and the data graph in Figure 5.2.

[4]In this example, a bushy execution plan is assumed.

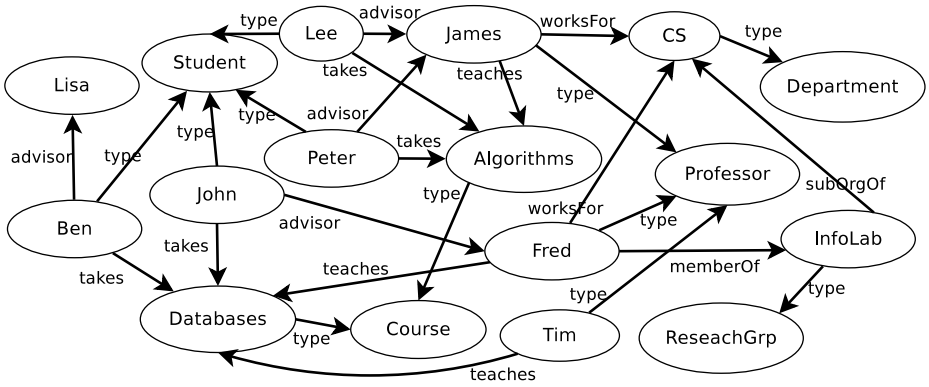


Figure 5.2: An example RDF graph in an academic domain.

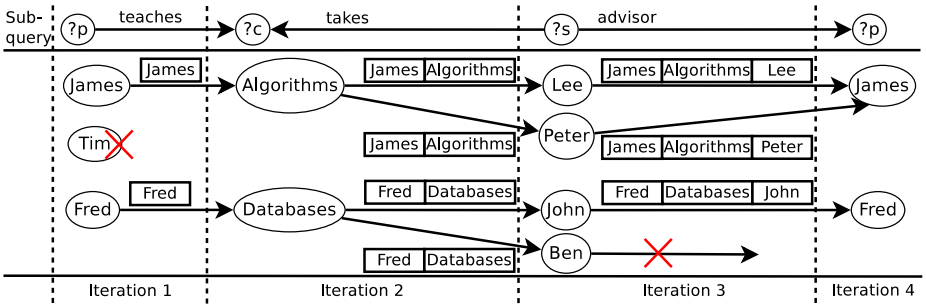


Figure 5.3: Computation iterations for solving Q_s.

Iteration 1 ($\bar{q}_1$). In the first iteration, all vertices are active and each vertex executes ExploreEdge with $\bar{q}_1$, empty message list, and the iteration number as inputs. Each vertex check if it matches the exploration vertex $\bar{q}_1.v_e$. A vertex can be a match to v_e if it has all the subqueries attached to v_e in the query graph (lines 4-5). Then, based on the exploration edge direction, all matching vertices retrieves their neighbors connected by predicate $\bar{q}_1.p$ (lines 7-10). Each vertex creates a message containing its id and send it to the retrieved neighbors. Finally, all vertices vote to halt; vertices become active if and only if they receive a message in the next iteration. In $\overline{Q_s}$, the exploration vertex v_e =?p. A matching vertex for ?p needs to be a subject and an object for the predicates teaches and advisor, respectively. Therefore, using the data graph of Figure 1.3, Fred and James are matches of ?p while Tim is not a match as he does not advise any students and will vote to halt. The direction of $\bar{q}_1$ is out, hence vertices use the PO index to get the list of neighbors(objects) connected via

```
Input: ExplorationEdge ē, MessageList ml, Iteration i
1  eVertex ← ē.expVertex;
2  tVertex ← ē.termVertex; eDirection ← ē.direction;
3  ePredicate ← ē.predicate;
4  vertexSubQueries ← getVertexSubqueries (eVertex);
5  if Matches (vertexSubQueries, eVertex) then
6      neighbors ← Empty;
7      if eDirection is Outgoing then
8          neighbors ← PO[ePredicate];
9      else
10         neighbors ← PS[ePredicate];
11     if i = 1 then
12         msg ← Empty;
13         msg[eVertex] ← vertexID;
14         sendMessageToAll(msg, neighbors);
15     else
16         if isQueryVertexVisited (ē.termVertex) then
17             foreach msg in ml do
18                 if msg[tVertex] ∈ neighbors then
19                     msg[eVertex] ← = vertexID;
20                     sendMessage(msg, msg[tVertex]);
21         else
22             foreach msg in ml do
23                 msg[eVertex] ← = vertexID;
24                 sendMessageToAll(msg, neighbors);
25 voteToHalt ();
```

Algorithm 5: ExploreEdge

predicate teaches (line 7-8). A message is formulated from each matching vertex of ?p and is sent to its neighbors connected via the predicate teaches (lines 11-14). Figure 5.3 depicts the steps.

Iteration 2 ($\bar{q}_2$). Vertices Databases and Algorithms received messages from Fred and James respectively. Hence, they are the only active vertices. Each of these two vertices checks if it matches the exploration vertex $\bar{q}_2.e_v$ =?c. Therefore, the matches of ?c are Databases and Algorithms. Then, each vertex use its PS indices to retrieve its neighbors connected via predicate takes. Each vertex appends its id to the received message and send the updated message to its list of neighbors (lines 21-24). Specif-

ically, Algorithms sends the message [James, Algorithms] to its neighbors Lee and Peter, whereas Databases sends [Fred, Databases] to John and Ben.

Iteration 3 ($\bar{q}_3$). Vertices Lee, Peter, John and Ben check if they match $\bar{q}_3.e_v = ?s$. Matching vertices use their PO indices to get their list of neighbors connected via predicate advisor. Since the termination vertex $\bar{q}_3.e_v = ?p$ has been visited before, messages are forwarded if and only if the $?p$ value in the message is also in the neighbors list. Notice that the message received by Ben has Fred as the $?p$ value, which is not in his neighbors list. Therefore, the message is truncated because it is not a valid result (lines 16-20).

Iteration 4. All vertices that received messages in this iteration have the final answer of $\overline{Q_s}$. This iteration can be omitted because the terminal vertex of the last iteration has already been visited. Hence, the results can be returned at the end of iteration 3. However, iteration 4 is kept for the sake of clarity.

Discussion. The exploration approach discussed in this work takes several advantages of the underlying vertex-centric framework for query evaluation. First, implicit join evaluation is achieved by inter-vertex message passing. This approach is different from the exploration approach discussed in Trinity.RDF which is more like semi-join. Trinity.RDF can only reduce the size of the intermediate relations but require a final centralized join. This is necessary especially for cyclic queries [30]. On the other hand, in SPARTex messages exchanged between vertices carry the intermediate results. Hence, no final join is needed as the final results are built and validated incrementally. Moreover, the bindings can reduce the size of the intermediate results significantly when queries have cycles. For example, in the third iteration of the previous example, Ben discards its message because it can validate that Fred (a visited node) is not in his neighbors list. This optimization is referred to as pre-join. Although, carrying the historical bindings seems to incur high communication overhead, the maximum number of query variables is usually small. For example, in

a real query workload acquired from Bio2RDF[10], the maximum number of variables per query is ten. The second advantage is the search space pruning that happen because of vertex activation/halting. In an exploration iteration, only active vertices apply the compute function; inactive vertices do nothing. Hence, this activation mechanism prunes the search space by eliminating vertices that would not contribute to the query results.

5.3.2 Query Planning

Query evaluation performance is highly influenced by the trail followed during execution. This section describes the cost-based optimizer which for a given query generates all possible query execution plans, estimates their costs and selects the plan with the minimum cost.

Query Optimization

The space of possible trails depends on the query graph structure and the fact that each edge has to be visited once. Specifically, a trail can be defined if and only if exactly zero or two vertices have odd degree. In the former case, the graph is called Eulerian graph; while in the latter is called traversable. The difference is that trails in Eulerian graphs start and end at the same vertex. For example, $\overline{Q_s}$ is Eulerian and has two trails (cycles) that start and end at vertex $?p$ ($?p{-}?c{-}?s{-}?p$ and $?p{-}?s{-}?c{-}?p$). The same applies to vertices $?c$ or $?s$. On the other hand, in a traversable graph, trails have to start from one of the odd degree vertices and end at the other odd degree vertex.

However, trails cannot be found for arbitrary queries that are neither Eulerian nor traversable. To solve this problem, the condition of visiting each edge once is relaxed by allowing the exploration of some edges more than once. This resembles the classical Chinese Postman Problem (CPP). Given a query graph, CPP finds a minimum length

closed walk that traverses each edge at least once. For a non-Eulerian graph, CPP duplicates some edges to make it Eulerian; allowing for a larger space of possible trails.

Query Coarsening. Obviously, the number of edges that the CPP will duplicate is highly correlated with the number of odd degree vertices. The higher the number of odd degree vertices, the higher the number of duplicate edges. This because once a trail passes through a vertex, it needs to exist from the vertex through another edge[5]. So, for each incident edge there has to be another unvisited edge to go through on the way out; which does not apply for odd vertices. For example, leaves i.e. vertices that have a single neighbor, are odd vertices because each has a single edge. Once the trail passes through such a vertex, it has to exit through the same vertex. Hence, that edge will be duplicated in the trail. Therefore, SPARTex introduces a query coarsening optimization that minimizes the number of odd degree vertices before making the graph Eulerian. Recall that, each vertex has direct access to its properties and incoming/outgoing neighbors. Therefore, all leaf vertices that have a single neighbor can be safely merged (coarsened) with its neighbor. For example, query $\overline{Q_s}$ is the coarsened version of Q_s.

Proposition **1.** A coarsened version of the query graph has at most the same number of odd degree vertices as the original query graph.

Proof: Let Q be a query graph with n vertices, L leaves, and O odd-degree vertices. Let Q' be the coarsened version of Q with $n - L$ vertices and O' odd-degree vertices. We show that that $O' \leq O$.

For each leaf $l \in L$, removing l has two cases: (i) l is connected to an odd-degree internal vertex. Removing l makes the degree of the internal vertex even. Then, $O' = O - 2$. (ii) l is connected to an even-degree internal vertex. Removing l

[5]This is true for all situations unless the vertex is the last vertex in the trail.

introduces an internal vertex of odd-degree. However, l itself is an odd degree vertex, then $O' = O$. In both cases, $O' \leq O$ holds. □

In this example, vertices that match $?c$ and $?p$ will validate if they have the rank and centrality properties, respectively. Notice that this optimization would coarsen any star query into a single vertex. Hence, any star query can be solved in a single iteration without communication.

Input: Query graph $Q = (V, E)$
Result: Exploration trail with minimum estimated cost

```
1  maxLength ← 0; minCost ← Infinity; bestPlan ← NULL;
2  if isEulerian (Q) then
3  |  maxLength ← Q.numEdges
4  else
5  |  x_e[|E|] ← CPP(Q); // Number of times each edge is duplicated
6  |  numDupEdges ← SUM(x_e);
7  |  maxLength ← Q.numEdges + numDupEdges;
8  cost ← 0; listVisitedEdges ← Empty;
9  foreach vertex v ∈ Q.vertices do
10 |  FindTrail(v, cost, visitedEdges);
11 return bestPlan;
12 Procedure FindTrail(Vertex v, cost, visitedEdges, x_e)
14 |  if cost > minCost then return;
15 |  ;
16 |  if |visitedEdges| > maxLength then return;
17 |  ;
18 |  if allVisited (Q, visitedEdges) then
19 |  |  if cost < minCost then
20 |  |  |  minCost ← cost;
21 |  |  |  bestPlan ← visitedEdges;
22 |  |  |  return;
23 |  foreach Edge e_i ∈ v.edges do
24 |  |  if x_e[e_i] < 0 then
25 |  |  |  return;
26 |  |  newCost ← cost+ getCost (visitedEdges, e_i);
27 |  |  newVisitedEdges ← visitedEdges.add(e_i);
28 |  |  x_e[e_i] ← x_e[e_i] − 1;
29 |  |  FindTrail (e_i.trmVrtx, newCost, newVisitedEdges, x_e);
30 |  return;
```

Algorithm 6: Query Optimizer

After coarsening the query, the planner uses Algorithm 6 to enumerates all possible

trails and selects the trail with the minimum estimated cost. The planner uses the CPP to get how many times each edge needs to be duplicated to make the graph Eulerian. CPP [116] will return an array x_e of integers; where each entry corresponds to an edge (line 5). A value of zero means that the edge is not duplicated; and will only appear once. The total number of duplicate edges is the sum of all entries in x_e (line 6). The max trail length is set to the number of edges in the graph plus the number of duplicate edges (lines 3 and 7). Then, it starts looking for an exploration trail from each vertex in the query graph (lines 9-10) using the procedure FindTrail (lines 12-28). The planner employs a branch and bound strategy to prune the search space of the possible trails using the plan cost as an upper-bound (plan cost estimation is discussed in Section 30). Initially, the exploration plan cost is set to infinity. A branch is pruned in four cases: (i) if a valid exploration plan with a minimum cost (so far) is found; i.e., all edges are visited, the cost bound and the best found plan are updated (lines 16-20). (ii) Since the cost is monotonically increasing, if the current exploration plan cost exceeded the bounded cost, the algorithm terminates (line 14). (iii) To avoid redundant computations, the algorithm terminates when the length of the exploration plan exceeds the maximum bounded length (line 15). (iii) Finally, if an edge is duplicated more than what induced by the CPP algorithm, the branch is pruned (lines 22-23).

Cost Estimation

The number of exchanged messages during query evaluation depends on the order of the exploration edges. Therefore, the optimizer tries to minimize the size of the intermediate results by exploring the most selective edges first. However, with the absence of a schema, selectivity estimation in SPARQL is a challenging task [20, 117]. Therefore, SPARTex uses a selectivity estimation method that captures the correlation among pairwise predicates. While loading the data, each vertex collects

the correlation information between its triples and sends it to the master, which aggregates the statistics. The master maintains the following statistics:

Predicate Counts $PC(p_i)$: for a predicate p_i, PC returns a pair (sc, oc), where sc and oc are the number of unique subjects and objects, respectively, attached to p_i in the data graph. For example, the predicate teaches in Figure 1.4 appeared three times and it has $(sc, oc) = (3, 2)$. Similarly, type has 11 unique subjects and 5 unique objects.

Predicates Pairwise Degrees $PPD(p_i, p_j, d_i, d_j)$: given a pair of predicates $(p_i,\ p_j)$ with their directions (d_i, d_j), $PPD(p_i, p_j, d_i, d_j)$ returns two values: (i) *count* is the number of vertices that have both predicates with their respective directions. (ii) (ad_i, ad_j) is an estimate of the average number of predicates p_i and p_j for each vertex $v \in U$. For example, $PPD(advisor, takes, out, out)$ returns {4, (1, 1)} because there are 4 vertices that have outgoing edges labeled *advisor* and *takes*. On average, each vertex has one edge labeled *advisor* and one edge labeled *takes*. From the exploration point of view, it means that there 4 vertices that exist when transitioning between advisor and takes. These vertices would get one message from the previous iteration and sends one messages out.

Algorithm 7 shows how to calculate the cost of the exploration trail $\{\bar{q}_1,\ \bar{q}_2,..., \bar{q}_n\}$ using n computation iterations. The plan cost is initially zero (line 1). For each exploration edge, Algorithm 7 increments the total plan cost with the expected number of messages to be transferred during exploration (line 2-5).

The subquery cost can be one of the following: (i) in the first iteration, the number of messages sent can be estimated from the number of matches of the current exploration vertex. The number of matches is estimated by considering all pairwise combinations of the predicates attached to the vertex. For each pair, PPD is used to get the unique number of nodes with this pair. The estimated number of matches is the minimum count of vertices in the graph that are attached to the pairwise predi-

```
   Input: Exploration Trail T {q̄_1, q̄_2,..., q̄_n}
   Result: Estimated Trail cost T_cost
1  T_cost ← 0; iterNo ← 0;
2  foreach ExplorationEdge q̄_i → T.edges do
3      q̄_i.cost ← ExplorationEdgeCost (q̄_i,iterNo);
4      T_cost ← T_cost + q̄_i.cost;
5      iterNo++;
6  Procedure ExplorationEdgeCost(q̄_i, iterNo)
8      iterCost ← 0;
9      if iterationNo = 1 then
10         iterCost ← estimateMatches (q̄_i.ev) * PC[q̄.pred ];
11         coarsenedCost ← 1;
12         foreach q̄_j → q̄_i.ev.coarsenedEdges do
13             coarsenedCost *= estimateMatches (q̄_j.tv)
14         iterCost ← iterCost * coarsenedCost;
15         return iterCost;
16     if explored (q̄.ev) is True then
17         if explored (q̄.tv) is True then return q̄_{i−1}.cost;
18         else return q̄_{i−1}.cost * avgDegree (q_{i−1}.pred, q_i.pred) ;
19     else
20         coarsenedCost ← 1;
21         foreach ē_j → ē_i.ev.coarsenedEdges do
22             coarsenedCost *= estimateMatches (ē_j.tv)
23         if explored (ē.tv) is True then
24             return ē_{i−1}.cost * coarsenedCost;
25         else return ē_{i−1}.cost * avgDegree (q_{i−1}.pred, q_i.pred) * coarsenedCost ;
27     return;
```

Algorithm 7: Explration Trail Cost

cates. Each matching vertex sends a number of messages equal to its average degree on the predicate $\bar{q}_i.p$ (line 10). If ev has a set of coarsened subqueries attached to it, the number of messages is multiplied by the number of bindings to the coarsened leaf vertex (lines 11-14). (ii) If both ev and tv are already explored, then the same number of received messages is is sent in this iteration (line 17). (iii) If ev is already explored and the termination vertex is not explored yet, then the messages received through the exploration predicate $\bar{q}_i.p$ are simply forwarded. This serves as an upper bound on the number of messages to be sent in this case (line 18). (iv) When exploration and termination vertices were not explored before, the number of messages sent is

based on messages received, average degree of the exploration predicate and the number of bindings of the coarsened leaves (line 25). (v) If the termination vertex was visited before, the messages received are forwarded to the termination vertex after considering the coarsened leaves (lines 23-24).

5.4 Experimental Evaluation

In this section, SPARTex is evaluated using the same experimental setup, datasets and queries discussed in Section 3.3. The experiments answer the following questions: (i) How well does SPARTex perform rich RDF analytics compared to combinations of existing systems? (ii) How efficient are the execution plans picked by SPARTex's optimizer? (iii) Using multiple real and synthetic benchmarks, how does SPARTex's SPARQL operator compare to existing specialized RDF systems? (iv) Finally, how well does SPARTex scale?

Implementation. The current version of SPARTex is implemented on top of GPS [113]; an open-source Pregel clone. Furthermore, to demonstrate that the proposed SPARQL operator can be used in native engines, another implementation of the SPARQL operator; coined Spartex-Native, is introduced. Spartex-Native is a modified version of AdPart-NA which differs as follows: (i) Spartex-Native partitions the data on subjects and objects to enable the query coarsening optimization. This means that Spartex-Native has subject-object hash locality awareness. (ii) Spartex-Native does not use distributed semi-join as in AdPart; instead it uses distributed hash joins by following trails.

5.4.1 Rich RDF Analytics

In this section, the three use cases described in Section 5.2.2 are implemented using SPARTex. Since no other system can fully support these use cases, combinations of SPARQL engines and graph processing systems are used. Specifically, H2RDF+ [30] is

Table 5.1: Datasets Statistics in millions (M)

Dataset	Triples (M)	#S (M)	#O (M)	#S∩O (M)	#P
KEGG	89.18	8.63	35.68	8.50	140
LinkedGeoData	274.67	51.92	121.10	41.47	18272

used as SPARQL engine with two different analytics systems. The first combination is H2RDF+ with PEGASUS [118], a graph mining library on top of MapReduce. The second combination uses H2RDF+ with GPS [113], an open source Pregel clone. Figure 5.4 shows the wall time of the first two use cases using LUBM-4000 dataset. In the first use case, the graph analytics are executed prior to query evaluation. GPS and PEGASUS are used to evaluate PageRank and degree centrality algorithms and the output is stored in HDFS. Notice that PEGASUS performed worse than GPS confirming that MapReduce approaches do not perform well for graph analytics. Then, the computation results are formatted as RDF triples and given to H2RDF+ along with the original RDF graph. H2RDF+ partition the input data and build its RDF indices. Finally, H2RDF+ is used to evaluate the SPARQL query and prints the results. Notice that both combinations required the data to be moved between multiple systems and formatted accordingly. SPARTex performs better than both combinations because it maintains the computation results of the analytics part in its in-memory store. These results are then utilized by the SPARQL operator. Therefore, no data formatting or indexing is required. The cost of data formatting and indexing is very substantial accounting for more than 80% of the processing time. Finally, when evaluating SPARQL queries, SPARTex performs significantly better than H2RDF+. The same applies on the second use case; however, since the SSSP algorithm is not available in PEGASUS, SPARTex is compared to H2RDF+GPS only.

Figure 5.5 shows the time of each phase of SamplD pipeline (see Section 5.2.2) for two real datasets; KEGG and LinkedGeoData (See table 5.1). KEGG[6] is a real dataset that integrates biological, chemical and gnomic information while LinkedGeo-

[6]http://www.genome.jp/kegg/

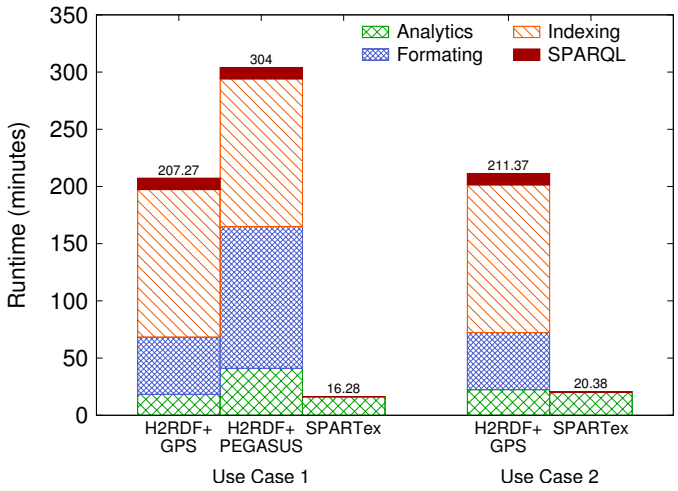


Figure 5.4: Rich RDF Analytics: Combining SPARQL with graph algorithms

Data[7] is a spatial knowledge base derived from the OpenStreetMap data. Both the rewriting (RDF to unlabeled graph) and the round trip (unlabeled graph to RDF) phases are consuming most of the time. SPARTex on the other hand, does not incur any rewriting phase for RDF as the data can be accessed with or without edge labels. This allows SPARTex to save almost 70% of the time spent by the SamplD scripts. Furthermore, SPARTex loads the RDF data once and keep it in memory. So there is no need for intermediate data reading/writing from/to the disk. As a result, SPARTex provides a single system for the whole SamplD pipeline with almost one order of magnitude better performance.

5.4.2 Query Optimizations

In this experiment, the query optimizer and its cost model are evaluated. The experiment shows that the plan selected is an efficient one. It also demonstrates the accuracy of the estimated cost (number of messages) compared to the actual cost. Using LUBM benchmark, only the complex queries; L1, L3, L7, P and D are considered[8]. These queries are solved in several iterations and generate large intermediate

[7] http://linkedgeodata.org/

[8] P and D are two additional complex query patterns that are defined to test the systems rigorously.

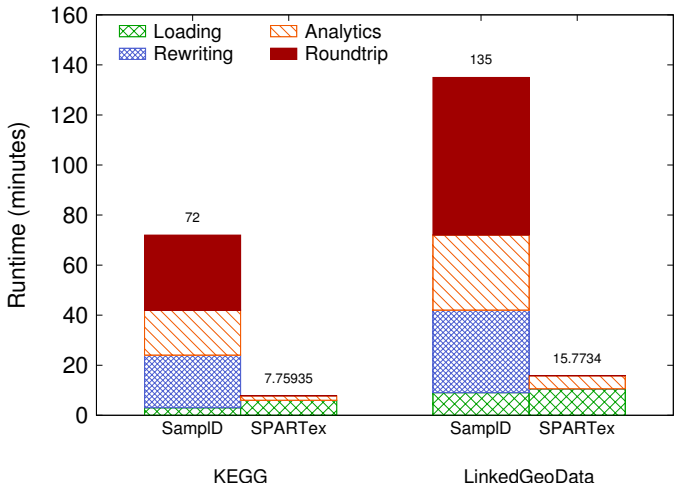
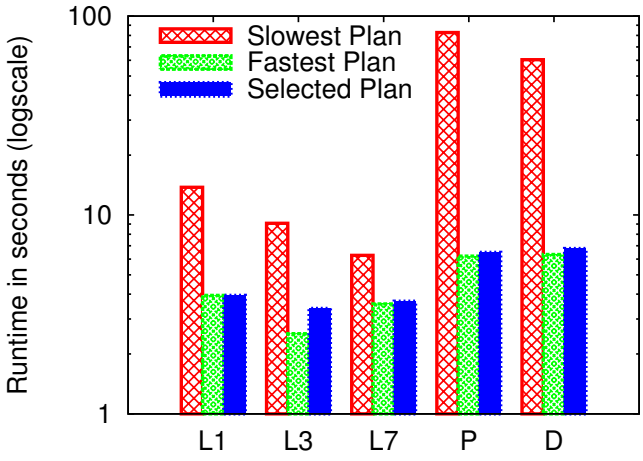


Figure 5.5: SamplD Analytics Pipeline.

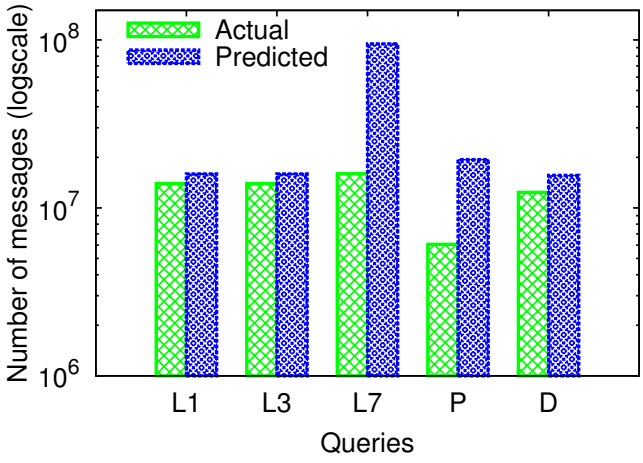
and/or final results. The rest of the queries are very selective (L6) or solved within a single iteration (L2, L4 and L5) and do not require communication.

In this experiment, all possible plans for each LUBM query are executed. Queries L1, L3 and L7 have the same structure; therefore, the number of possible trails is the same for all of them (6 trails). Queries P and D have 36 and 176 possible plans, respectively. Using LUBM-4000, Figure 5.6(a) shows the fastest and slowest execution times for each query. It also shows the execution time for the plan selected by the optimizer. For all complex queries the optimizer selects a plan that is either optimal in the search space or has performance very close to the fastest execution plan. Note that for P there were 19 plans that never finish because of the huge number of generated messages during query execution, which cause network contention.

Figure 5.6(b) shows the estimated vs. the actual number of messages transferred between vertices during the execution of the selected plan for each LUBM query. As shown, the optimizer estimates the total number of messages for almost all queries with a very high accuracy. Query L7 generates a huge number of intermediate results at the first few iterations; however, many intermediate results are dropped at the final iteration because of the cycle. The cost function is monotonically increasing, hence, it can not capture this sudden drop of intermediate results. Nonetheless, the number



(a)



(b) Message count

Figure 5.6: Query Optimization.

Table 5.2: Query runtime for LUBM-10240 (seconds)

LUBM-10240	L1	L2	L3	L4	L5	L6	L7	P	D	Geo-Mean
SPARTex	**4.48**	**6.78**	**5.45**	**3.40**	**3.24**	**2.38**	**7.02**	**7.89**	**6.21**	**4.85**
CliqueSquare	125.02	71.01	80.01	90.01	24.00	37.01	224.04	161.02	160.02	88.35
H2RDF+	285.43	71.72	264.78	24.12	4.76	22.91	180.32	1142.10	568.58	105.860
SHARD	413.72	187.31	N/A	358.20	116.62	209.80	469.34	596.08	544.94	317.606
Spartex-Native	2.881	0.406	2.953	**0.001**	**0.001**	**0.010**	**2.386**	**3.408**	**4.768**	0.222
AdPart-NA	**2.743**	**0.120**	**0.320**	0.001	0.001	0.040	3.203	5.724	4.793	**0.193**
TriAD	6.023	1.519	2.387	0.006	0.004	0.114	17.586	19.839	65.628	1.035
TriAD-SG (100K)	5.392	1.774	4.636	0.009	0.005	0.010	21.567	44.135	144.256	1.119
SHAPE	25.319	4.387	25.360	1.603	1.574	1.567	15.026	N/A	N/A	5.575

of messages for the last iteration is the same for all plans because of the pre-join condition; therefore, the optimizer succeeds in selecting the most efficient plan.

5.4.3 Query Performance

LUBM Dataset. Recall that the LUBM queries defined in [24] can be classified into two types, simple and complex. Simple queries are very selective, touch small number of triples, and generate small intermediate and final results. Complex queries consist of non-selective joins and result in large intermediate results. L1, L2, L3, L7, P and D are complex and the rest of the queries are simple.

Table 5.2 shows the performance of SPARTex against state-of-the-art distributed RDF stores. If a system fails to solve a query within a reasonable time (1 hour) or crashes, it is marked as N/A. RDF engines are categorized into two groups based on their underlying framework: (i) systems built on top of generic frameworks: SPARTex, CliqueSquare, SHARD and H2RDF+; and (ii) Native RDF systems: AdPart-NA, SHAPE, TriAD and Spartex-Native. SHAPE, which uses RDF-3X for storage, is considered native because data is partitioned such that each RDF-3X engine returns a partial final result without the need for any communication.

SPARTex utilizes the efficient inter-vertex communication of vertex-centric frameworks. In other words, the framework is actually contributing to the join by delivering messages directly to the vertices. Vertices that do not receive messages are automatically pruned. For L1, L3 and L7, SPARTex performs multiple joins concurrently

because of the coarsening strategy. Then, only two distributed joins (2 iterations) are required for evaluating the final query results. This is possible because of the pre-join optimization. On the other hand, for SHARD and H2RDF+, multiple distributed joins are necessary for query evaluation. As a result, the geometric mean of SPARTex is one and two orders of magnitude better than CliqueSquare, H2RDF+ and SHARD, respectively.

Similarly and without any data preprocessing, Spartex-Native is significantly faster than SHAPE, TriAD and TriAD-SG for complex queries. SHAPE performs worse because of replication and the way replication is managed. All triples (including replicas) are stored together resulting in a large search space during join evaluation. As simple queries touch small amount of data, Spartex-Native, AdPart-NA and TriAD achieve comparable performance for queries L4 and L5. L2 is a non-selective star query on which the hash-join technique, employed by SPARTex and AdPart-NA, has a better performance than the merge-join employed by TriAD and TriAD-SG. Query L3 returns empty results on which AdPart-NA evaluated the join that produce the empty results earlier than SPARTex and TriAD. Queries L1 and L7 are cyclic queries. Hence, Spartex-Native minimizes the amount of communicated data using the pre-join technique.

Queries P and D consist of 9 and 8 triple patterns which require 8 and 7 joins, respectively. For both queries, AdPart-NA executes 3 joins without communication and synchronization overheads because of the subject pinning technique. The right operand of the remaining joins has subject as the join attribute; hence, AdPart-NA shards the intermediate results among workers. TriAD, on the other hand, needs to shard both relations in order to perform hash/merge joins. The query coarsening technique of Spartex-Native results in eliminating three joins. Moreover, since both queries have multiple cycles, the pre-join is applied multiple times reducing the amount of messages significantly. Accordingly, Spartex-Native achieves the best per-

Table 5.3: Query runtimes for YAGO2 (ms)

YAGO2	**Y1**	**Y2**	**Y3**	**Y4**	**Geo-Mean**
SPARTex	**2,803**	**3,544**	**2,002**	**2,719**	**2,712**
CliqueSquare	139,021	73,011	36,006	100,015	77,755
H2RDF+	10,962	12,349	43,868	35,517	21,430
SHARD	238,861	238,861	aborted	aborted	238,861
Spartex-Native	38	126	**35**	33	**49**
AdPart-NA	19	**46**	570	77	79
TriAD	**16**	1,568	220	**18**	100
SHAPE	1,824	665,514	1,823	1,871	8,022

formance in these two complex queries compared to other specialized RDF engines.

YAGO2 Dataset. Table 5.3 shows the performance of SPARTex using YAGO2 dataset. TriAD-SG is not listed because the optimal number of summary graph partitions is not known, a process that requires empirical evaluation [32]. Recall that Y1 and Y2 are simple queries while Y3 and Y4 are complex. H2RDF+ outperforms CliqueSquare and SHARD due to the utilization of HBase indexes and its distributed implementation of merge and sort-merge joins. The flat plans did not improve the performance CliqueSquare compared to H2RDF+. The reason is that, while the flat plans reduce the number of MapReduce-based joins, H2RDF+ uses a more efficient implementation of the join operator using the traditional RDF indices. Furthermore, unlike CliqueSquare, H2RDF+ encodes the URIs and literals of RDF data; hence it incurs minimal overhead when shuffling intermediate results. SPARTex outperforms SHARD, H2RDF+ and CliqueSquare with up to two orders of magnitude better performance. The utilization of the direct inter-vertex communication for join evaluation helps SPARTex to evaluate queries in a more efficient way. Furthermore, the coarsening strategy helps in solving the queries in less number of iterations by evaluating multiple joins concurrently.

Spartex-Native continues to provide good performance compared to other specialized RDF engines. For SHAPE, 2-hop forward partitioning placed all the data in a single partition which makes it perform as good as a single machine RDF-3X engine.

Table 5.4: Query runtimes for Bio2RDF (ms)

Bio2RDF	**B1**	**B2**	**B3**	**B4**	**B5**	**Geo-Mean**
SPARTex	**2,063**	**3,229**	**3,356**	**7,950**	**2,281**	**3,323**
H2RDF+	5,580	12,710	322,300	7,960	4,280	15,076
SHARD	239,350	309,440	512,850	787,100	112,280	320,027
Spartex-Native	**1**	**1**	8	**26**	**2**	**3**
AdPart-NA	17	16	32	89	1	15
TriAD	4	4	**5**	N/A	2	4

AdPart-NA solves most of the joins in Y1 and Y2 without communication which explains its superior performance compared to Spartex-Native and TriAD. Spartex-Native utilizes the pre-join and coarsening techniques to reduce the number of iterations and communicate less data. Accordingly, Spartex-Native requires three and five iterations to solve Y1 and Y2 respectively. This makes Spartex-Native achieves almost comparable performance to TriAD in Y1 and significantly better in Y2. On the other hand, Y3 require object-object joins which contradicts the subject-based hash partitioning of AdPart-NA causing it to perform worse than Spartex-Native and TriAD which have subject-object locality awareness. In Y3, the coarsening technique of Spartex-Native caused two join operations to be performed without communication which results in an order of magnitude better performance compared to TriAD and AdPart-NA. Y4 produces small number of results compared to Y3 on which TriAD performs better than Spartex-Native. Notice that the geometric mean of Spartex-Native is still better than that of both TriAD and AdPart-NA.

Bio2RDF dataset. Similar to YAGO dataset, TriAD-SG is not listed here as the number of partitions to use for creating the summary graph is not known which requires empirical evaluation of some workload on the data or a representative sample. Also, SHAPE and CliqueSquare failed to process this dataset using 2-hop forward partitioning within a reasonable time.

Similar to its behavior in the other datasets, SHARD still performs worse than all other systems due to the MapReduce overhead and lack of using efficient indices.

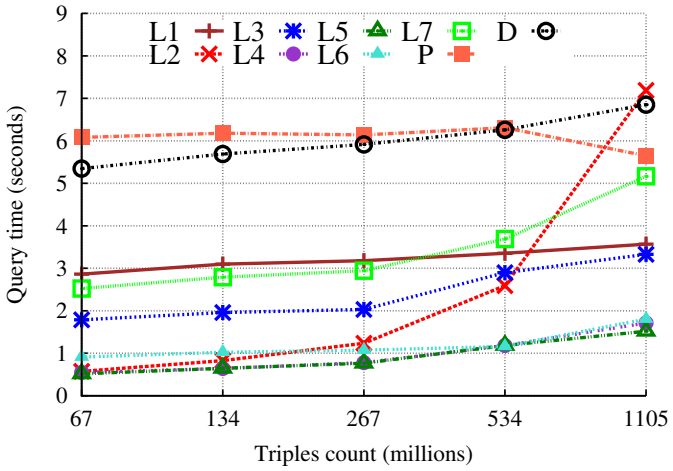


Figure 5.7: Data Scalability using LUBM dataset.

H2RDF+ performs better than SHARD due to the utilization of its HBase indices and the efficient join implementations. SPARTex continues to outperform both H2RDF+ and SHARD in all queries; some times by orders of magnitude speedup. Our native implementation; Spartex-Native, still provides comparable or better performance than TriAD and AdPart-NA with a better geometric mean value.

5.4.4 Scalability

Vertex-centric frameworks scale very well for many graph algorithms [119]. To evaluate the scalability of SPARTex, two experiments are conducted: (i) varying the size of the data while fixing the number of workers (cores) and (ii) varying the number of workers while the data size is fixed. In this experiment, the GPS-based implementation is used.

Data Scalability. Five LUBM datasets were generated using the LUBM benchmark data generator: LUBM-500, LUBM-1000, LUBM-2000, LUBM-4000 and LUBM-8000. The SPARQL operator in SPARTex reduces the size of the intermediate results significantly by exploring subqueries from the results of previous subqueries. Carrying the historical bindings allows it to reduce the number of messages significantly for

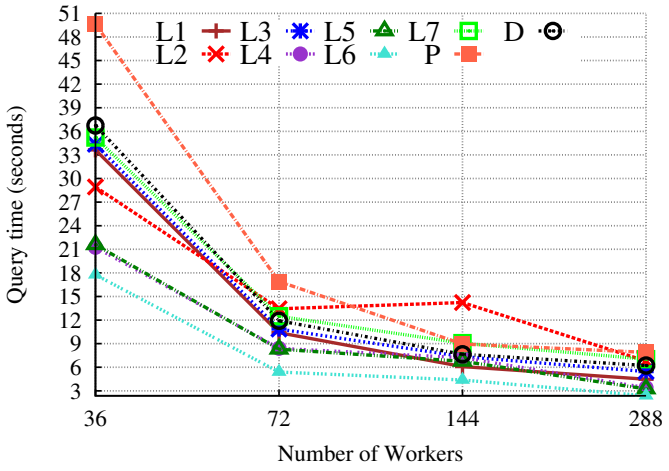


Figure 5.8: Strong Scalability (LUBM-10240).

cyclic queries. Moreover, no final join is needed as the results are built and validated incrementally. The SPARQL operator also reduces the number of computation iterations by query coarsening. All these factors explain the good scalability of SPARTex as the data size grows for complex LUBM queries (see Figure 5.7). Notice that L2 is a reporting query that generates a proportional amount of results to the data size. Therefore, the response time of this query increases as the data grows. Since simple queries touch almost the same amount of data regardless of the data size, SPARTex provides almost steady performance for these queries.

Strong Scalability. In this experiment, the number of workers is varied while the dataset is fixed to LUBM-10240. Figure 5.8 shows the scalability test results for both simple and complex queries. The response times of both simple and complex queries decrease as the number of workers increases. This means that SPARTex can benefit from the parallelism of distributed environments. However, after certain number of workers, query response times are dominated by the communication cost, which increases as the number of workers increases. In all queries, the query response times decrease drastically when the number of workers is increased from 36 to 72. The response times still decrease after that point; but not with the same rate.

Chapter 6

Concluding Remarks

> There is a point in every contest when sitting on the sidelines is not an option
>
> Dean Smith
> 1931 — 2015 CE

This chapter concludes this dissertation with a summary of our contributions and an outlook on possible future research directions.

6.1 Summary of Contributions

The wide adoption of the RDF data format has lead to an ever increasing volume of publicly available RDF data on the Web. Concurrently, the diversity, complexity and dynamism of RDF queries and applications have increased significantly. This dissertation introduces techniques for accelerating SPARQL queries and RDF analytics on distributed shared-nothing RDF systems.

First, AdPart, an adaptive distributed RDF engine, is introduced. AdPart starts significantly fast by employing lightweight partitioning that hashes triples on the subjects. AdPart exploits query structures and the hash-based data locality in order to minimize the communication cost during query evaluation. Specifically, star queries joining on subjects are processed in parallel by all workers. Furthermore, whenever possible, intermediate results are hash-distributed among workers instead of broad-

casting to all workers. By exploiting hash-based locality, AdPart achieves better or comparable performance to systems that employ sophisticated partitioning schemes.

To cope with the dynamism of RDF workloads, AdPart is extended with the adaptive workload-awareness feature. AdPart monitors the query workload and incrementally redistributes parts of the data that are frequently accessed by hot patterns. By maintaining and indexing these patterns, many future queries are evaluated without communication. The adaptivity feature of AdPart complements its excellent performance on queries that can benefit from its hash-based data locality. Frequent query patterns that are not favored by the initial partitioning (e.g., star joins on an object) can be processed in parallel due to adaptivity. The experimental results verify that AdPart achieves better partitioning and replicates less data than its competitors. More importantly, AdPart scales to very large RDF graphs and consistently provides superior performance by adapting to dynamically changing workloads.

Finally, to support and accelerate rich RDF analytical tasks, SPARTex is proposed. SPARTex is a vertex-centric RDF analytics framework that can be implemented on top of any generic vertex-centric graph processing framework (e.g., Pregel). Any graph algorithm implemented in the vertex-centric framework (e.g., PageRank, Shortest-Paths, etc.), can be used in SPARTex. Additionally, SPARTex implements a generic SPARQL operator as a vertex-centric program. The operator interprets general SPARQL queries and includes a query optimizer to generate efficient execution plans. Users can write programs that combine generic graph processing algorithms with SPARQL queries in a pipelined fashion, and can share variables at the granularity of a vertex. Various scenarios, where SPARTex simplifies significantly the implementation of rich RDF data analytics programs, were presented. As a SPARQL engine, SPARTex is at least as fast as the state-of-the-art specialized distributed RDF engines. Moreover, analytical tasks in SPARTex is at least one order of magnitude faster than existing alternatives.

6.2 Future Research Directions

This thesis advances the state-of-the-art techniques for handling dynamic, complex and diverse RDF queries and analytics. However, there are few research directions that can be investigated in the future:

6.2.1 Exploiting Current Hardware Architecture

Nowadays with the advent of multi-core processors and co-processors, single-machines can be thought of as distributed infrastructures. A single machine consists of multiple sockets; each with its own chip and memory controller. As the clock rate of new processors cannot practically keep up with Moore's Law, manufacturers are moving towards increasing the number of processing cores per chip. Furthermore, because memory controllers are distributed, applications are susceptible to the Non-Uniform Memory Access (NUMA) effect. Current RDF data management systems fail at harnessing the full power of this emerging architecture.

Specifically, all single-machine and distributed RDF stores do not fully utilize all the cores available per worker. RDF-3X [19] uses multi-threading for concurrent data scans and for pipelined execution plans using Sideways Information Passing (SIP). Similarly, TriAD [32] uses multi-threading to asynchronously execute multiple branches of the bushy execution plan. However, this type of parallelism is bounded by the number of base subqueries that need to be scanned and the shape of the execution tree. AdPart solves this problem by running a single thread per core; each core is responsible for a data partition. The down side of this approach is that although cores on the same node share the same memory space, workers running on them communicate using MPI. Although MPI is smart and uses memory copy, the overhead is still significant compared to direct memory access. Recently, $Turbo_{HOM++}$ [18] proposed a NUMA aware solution for SPARQL query evaluation. However, their approach is not generic and cannot be applied on systems that use relational joins.

Consequently, a very interesting research problem is to adapt a scale-out in-memory system like AdPart or TriAD to work as a scale-up shared memory solution. The new system needs to be NUMA-aware and capable of evaluating a single SPARQL query using all available cores. To achieve this goal, few design decision needs to be made. First, how would the data be partitioned among the NUMA nodes? Second, what would be the indexing and execution strategies? Regarding data partitioning, the system needs to employ a light-weight partitioning strategy. However, it cannot be by simple hashing. The reason is that, although hashing guarantees vertex locality i.e. a vertex with all its edges are stored on the same partition, it destroys global locality. In other words, two neighboring vertices will hash to two different partitions because of their different hash values. Almost all existing systems that use hash partitioning rely on hashing for string-to-id data encoding. Vertices are stored at partition $p = v.id \ \% \ num\_partitions$. Obviously, smart data encoding would result in better data locality. A recent effort [120] has introduced a locality-based encoding that can be used for global data locality. Then, data is partitioned among all cores. However, partition to NUMA node assignment needs to be done carefully. The reason is that NUMA effect happens when accessing data across NUMA nodes. At the same time, the edge-cut between multiple partitions is not the same. Therefore, the system can create a summary graph of the partitions where each node is a partition and each edge is a cross partition edge. The summary graph can be partitioned among the NUMA nodes. This would place highly correlated partitions on the same NUMA node. As a start, the system can utilize the Morsel Driven approach [121] for data indexing and query evaluation. Query evaluation is carried out concurrently by all cores.

6.2.2 Extending SPARQL Beyond BGP

Currently, AdPart and SPARTex support and are optimized for BGP SPARQL queries. However, there are many other new features that are emerging in SPARQL standard.

Particularly, property paths [122] are of very high importance and challenge [123]. Trail execution plans in SPARTex can be extended to evaluate property path queries based on the existence semantics [123]. In other words, SPARTex can check the existence of property paths without counting them. The execution trail will have some known routes that are based on explicit predicates; while other parts of the trail contain regular expressions allowing many matches. Therefore, generating an efficient plan that generates minimal communication would be very challenging.

6.2.3 Supporting Multi-query Optimization

Multi-query optimization is a classical problem which has been extensively studied in the relational model. However, few efforts [124] have focused on multi SPARQL query evaluation. Since SPARQL multi-query optimization is a NP-Hard problem [124], AdPart and SPARTex can employ heuristics based solution. However, the objective for this optimization is different between the systems. In AdPart, distributed execution incurs communication which needs to be amortized among multiple concurrent queries. Therefore, instead of relying on finding Maximal Common Edge Subgraphs (NP-Hard) [124], AdPart can utilize its query transformation approach and the query index search function to find the common structure among queries. These structures are evaluated once and shared among queries. On the other hand, in SPARTex most of the execution time is spent on analytical algorithms. Therefore, the gain from sharing the execution of generic graph algorithms is higher than sharing the execution of common parts in SPARQL. Consequently, SPARTex can be extended to support multi-query optimization for analytical queries. Notice that analytical queries may consists of a pipeline of operators. Hence, pipelines of multiple queries need to be aligned in a way that minimizes the over all execution time.

REFERENCES

[1] G. Schreiber and Y. Raimond, "RDF Primer," http://www.w3.org/TR/rdf-primer/, 2014.

[2] R. Cyganiak, D. Wood, and M. Lanthaler, "Resource Description Framework (RDF)," https://www.w3.org/TR/rdf-concepts/, 2014.

[3] Berners-Lee, Tim and Hendler, James and Lassila, Ora, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[4] Shadbolt, Nigel and Berners-Lee, Tim and Hall, Wendy, "The Semantic Web Revisited," *IEEE Intelligent Systems*, vol. 21, no. 3, pp. 96–101, 2006.

[5] E. Prudhommeaux and A. Seaborne, "SPARQL Query Language for RDF," https://www.w3.org/TR/rdf-sparql-query/, 2014.

[6] T. W. S. W. Group, "SPARQL 1.1 Overview," https://www.w3.org/TR/sparql11-overview/, 2014.

[7] The UniProt Consortium, "Activities at the Universal Protein Resource (UniProt)," *Nucleic Acids Research*, vol. 42, no. D1, pp. D191–D198, 2014.

[8] "PubChemRDF Release Notes," http://pubchem.ncbi.nlm.nih.gov/rdf/.

[9] Auer, Sören and Bizer, Christian and Kobilarov, Georgi and Lehmann, Jens and Cyganiak, Richard and Ives, Zachary, "DBpedia: A Nucleus for a Web of Open Data," in *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference*, ser. ISWC'07/ASWC'07, 2007.

[10] Callahan, Alison and Cruz-Toledo, Jose and Ansell, Peter and Dumontier, Michel, "Bio2RDF Release 2: Improved Coverage, Interoperability and Provenance of Life Science Linked Data." in *ESWC*, vol. 7882, 2013, pp. 200–212.

[11] Wu, Wentao and Li, Hongsong and Wang, Haixun and Zhu, Kenny Q., "Probase: A Probabilistic Taxonomy for Text Understanding," in *SIGMOD*, 2012.

[12] Bizer, Christian, "The Web of Linked Data: A Global Public Dataspace on the Web: WebDB 2010 Keynote," in *Procceedings of the 13th International Workshop on the Web and Databases*, 2010, pp. 1:1–1:1.

[13] Bizer, C. and Heath, T. and Berners-Lee, T., "Linked data - the story so far," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 3, p. 122, 2009.

[14] Hartig, Olaf and Bizer, Christian and Freytag, Johann-Christoph, "Executing SPARQL Queries over the Web of Linked Data," in *Proceedings of the 8th International Semantic Web Conference*, ser. ISWC, 2009, pp. 293–309.

[15] O. Hartig and M. T. Özsu, "Linked Data query processing," in *IEEE International Conference on Data Engineering (ICDE)*, 2014, pp. 1286–1289.

[16] Schwarte, Andreas and Haase, Peter and Hose, Katja and Schenkel, Ralf and Schmidt, Michael, "Fedx: Optimization techniques for federated query processing on linked data," in *Proceedings of the 10th International Semantic Web Conference*, ser. ISWC, 2011, pp. 601–616.

[17] Aluç, Gúneş, "Workload Matters: A Robust Approach to Physical RDF Database Design," *PhD thesis. University of Waterloo*, 2015.

[18] Kim, Jinha and Shin, Hyungyu and Han, Wook-Shin and Hong, Sungpack and Chafi, Hassan, "Taming Subgraph Isomorphism for RDF Query Processing," *PVLDB*, vol. 8, no. 11, pp. 1238–1249, 2015.

[19] Neumann, Thomas and Weikum, Gerhard, "RDF-3X: A RISC-style Engine for RDF," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.

[20] ——, "The RDF-3X Engine for Scalable Management of RDF Data," *VLDBJ*, vol. 19, no. 1, pp. 91–113, 2010.

[21] ——, "x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases," *PVLDB*, vol. 3, no. 1-2, pp. 256–263, 2010.

[22] Weiss, Cathrin and Karras, Panagiotis and Bernstein, Abraham, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[23] Yuan, Pingpeng and Liu, Pu and Wu, Buwen and Jin, Hai and Zhang, Wenya and Liu, Ling, "TripleBit: A Fast and Compact System for Large Scale RDF Data," *PVLDB*, vol. 6, no. 7, pp. 517–528, 2013.

[24] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data," in *WWW*, 2010.

[25] Zou, Lei and Mo, Jinghui and Chen, Lei and Özsu, M. Tamer and Zhao, Dongyan, "gStore: Answering SPARQL Queries via Subgraph Matching," *PVLDB*, vol. 4, no. 8, pp. 482–493, 2011.

[26] J. Huang, D. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, no. 11, 2011.

[27] Zeng, Kai and Yang, Jiacheng and Wang, Haixun and Shao, Bin and Wang, Zhongyuan, "A distributed graph engine for web scale RDF data," *PVLDB*, vol. 6, no. 4, 2013.

[28] Rohloff, Kurt and Schantz, Richard E., "High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store," in *Programming Support Innovations for Emerging Distributed Applications*, 2010.

[29] Papailiou, Nikolaos and Konstantinou, Ioannis and Tsoumakos, Dimitrios and Koziris, Nectarios, "H2rdf: Adaptive query processing on rdf data in the cloud." in *Proc. of Companion on World Wide Web*, 2012.

[30] Papailiou, Nikolaos and Konstantinou, Ioannis and Tsoumakos, Dimitrios and Karras, Panagiotis and Koziris, Nectarios, "H2RDF+: High-performance distributed joins over large-scale RDF graphs," in *Big Data, 2013 IEEE International Conference on*, 2013.

[31] Lee, Kisung and Liu, Ling, "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *PVLDB*, vol. 6, no. 14, 2013.

[32] Gurajada, Sairam and Seufert, Stephan and Miliaraki, Iris and Theobald, Martin, "TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing," in *SIGMOD*, 2014.

[33] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *TKDE*, vol. 23, no. 9, 2011.

[34] Hose, K. and Schenkel, R., "WARP: Workload-aware replication and partitioning for RDF," in *IEEE International Conference on Data Engineering (ICDE Workshops)*, 2013.

[35] Luis Galarraga and Katja Hose and Ralf Schenkel, "Partout: A Distributed Engine for Efficient RDF Processing," *CoRR*, vol. abs/1212.5636, 2012.

[36] Wu, Buwen and Zhou, Yongluan and Yuan, Pingpeng and Liu, Ling and Jin, Hai, "Scalable SPARQL Querying using Path Partitioning," in *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[37] Punnoose, Roshan and Crainiceanu, Adina and Rapp, David, "Rya: A Scalable RDF Triple Store for the Clouds," in *Cloud-I*, 2012.

[38] Goasdoué, François and Kaoudi, Zoi and Manolescu, Ioana and Quiané-Ruiz, Jorge-Arnulfo and Zampetakis, Stamatis, "CliqueSquare: Flat plans for massively parallel RDF queries," in *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[39] G. Aluç, M. T. Özsu, and K. Daudjee, "Workload Matters: Why RDF Databases Need a New Design," *PVLDB*, vol. 7, no. 10, 2014.

[40] Gallego, Mario Arias and Fernández, Javier D and Martínez-Prieto, Miguel A and de la Fuente, Pablo, "An empirical study of real-world SPARQL queries," in *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India*, 2011.

[41] Duan, Songyun and Kementsietsidis, Anastasios and Srinivas, Kavitha and Udrea, Octavian, "Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets," in *SIGMOD*, 2011.

[42] Markus Kirchberg and Ryan K. L. Ko and Bu-Sung Lee, "From Linked Data to Relevant Data – Time is the Essence," *CoRR*, vol. abs/1103.5046, 2011.

[43] Zhang, Yuji and Tao, Cui and He, Yongqun and Kanjamala, Pradip and Liu, Hongfang, "Network-based analysis of vaccine-related associations reveals consistent knowledge with the vaccine ontology." *J. Biomedical Semantics*, vol. 4, p. 33, 2013.

[44] Rietveld, Laurens and Hoekstra, Rinke and Schlobach, Stefan and Guéret, Christophe, "Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling," in *ISWC*, 2014.

[45] Qu, Xiaoyan and Gudivada, Ranga and Jegga, Anil and Neumann, Eric and Aronow, Bruce , "Inferring novel disease indications for known drugs by semantically linking drug action and disease mechanism relationships," *BMC bioinformatics*, vol. 10, no. Suppl 5, p. S4, 2009.

[46] Tao, Cui and Wu, Puqiang and Zhang, Yuji, "Linked Vaccine Adverse Event Data Representation from VAERS for Biomedical Informatics Research," in

Trends and Applications in Knowledge Discovery and Data Mining, 2014, pp. 652–661.

[47] Dean, Jeffrey and Ghemawat, Sanjay, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.

[48] Shao, Bin and Wang, Haixun and Li, Yatao, "Trinity: a distributed graph engine on a memory cloud," in *SIGMOD*, 2013.

[49] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a System for Large-scale Graph Processing," in *SIGMOD*, 2010.

[50] Gonzalez, Joseph E. and Low, Yucheng and Gu, Haijie and Bickson, Danny and Guestrin, Carlos, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *OSDI*, 2012.

[51] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola and J. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[52] G. Wang, W. Xie, A. Demers and J. Gehrke, "Asynchronous Large-Scale Graph Processing Made Easy," in *CIDR*, 2013.

[53] "HBase," HBase.http://hbase.apache.org.

[54] Karypis, George and Kumar, Vipin, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[55] Harth, Andreas and Umbrich, Jrgen and Hogan, Aidan and Decker, Stefan, "YARS2: A Federated Repository for Querying Graph Structured Data from the Web," in *ISWC/ASWC*, vol. 4825, 2007.

[56] R. W. Techentin, B. K. Gilbert, A. Lugowski, K. Deweese, J. R. Gilbert, E. Dull, M. Hinchey, and S. P. Reinhardt, "Implementing Iterative Algorithms with SPARQL." in *EDBT/ICDT Workshops*, 2014, pp. 216–223.

[57] Qi, Letao and Lin, Harris and Honavar, Vasant, "Clustering remote RDF data using SPARQL update queries," in *IEEE International Conference on Data Engineering (ICDE Workshops)*. IEEE, 2013.

[58] Deweese, Kevin and Gilbert, John R and Lugowski, Adam and Reinhardt, Steve and Kepner, Jeremy and Gilbert, John R, "Graph Clustering in SPARQL," in *SIAM Workshop on Network Science*, vol. 34. ACM, 2013, pp. 930–941.

[59] Mizell, David and Maschhoff, Kristyn J and Reinhardt, Steven P, "Extending SPARQL with graph functions," in *IEEE Big Data*, 2014.

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

[61] Zaharia, Matei and Chowdhury, Mosharaf and Das, Tathagata and Dave, Ankur and Ma, Justin and McCauley, Murphy and Franklin, Michael J and Shenker, Scott and Stoica, Ion, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX*, 2012.

[62] Zaharia, Matei and Chowdhury, Mosharaf and Franklin, Michael J and Shenker, Scott and Stoica, Ion, "Spark: Cluster Computing with Working Sets." *HotCloud*, vol. 10, 2010.

[63] Xin, Reynold S. and Gonzalez, Joseph E. and Franklin, Michael J. and Stoica, Ion, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13, 2013, pp. 2:1–2:6.

[64] Gonzalez, Joseph E and Xin, Reynold S and Dave, Ankur and Crankshaw, Daniel and Franklin, Michael J and Stoica, Ion, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[65] Olston, Christopher and Reed, Benjamin and Srivastava, Utkarsh and Kumar, Ravi and Tomkins, Andrew, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*. ACM, 2008, pp. 1099–1110.

[66] Abadi, Daniel J. and Marcus, Adam and Madden, Samuel R. and Hollenbach, Kate, "SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management," *VLDB J.*, vol. 18, no. 2, pp. 385–406, 2009.

[67] Goodman, Eric L. and Grunwald, Dirk, "Using Vertex-centric Programming Platforms to Implement SPARQL Queries on Large Graphs," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '14, 2014.

[68] Schätzle, Alexander and Przyjaciel-Zablocki, Martin and Berberich, Thorsten and Lausen, Georg, "S2X: Graph-Parallel Querying of RDF with GraphX," in *Proceedings of the 1st Workshop on Big Graphs Online Querying (Big-O(Q))*, 2015.

[69] Harbi, Razen and Abdelaziz, Ibrahim and Kalnis, Panos and Mamoulis, Nikos and Ebrahim, Yasser and Sahli, Majed, "Accelerating SPARQL Queries by

Exploiting Hash-based Locality and Adaptive Partitioning," *VLDBJ*, vol. 25, no. 3, pp. 355–380, 2016.

[70] Harbi, Razen and Abdelaziz, Ibrahim and Kalnis, Panos and Mamoulis, Nikos, "Evaluating SPARQL Queries on Massive RDF Datasets," *PVLDB*, vol. 8, no. 12, pp. 1848–1851, 2015.

[71] Abdelaziz, Ibrahim and Harbi, Razen and Salihoglu, Semih and Kalnis, Panos and Mamoulis, Nikos, "SPARTex: A Vertex-centric Framework for RDF Data Analytics," *PVLDB*, vol. 8, no. 12, pp. 1880–1883, 2015.

[72] Sakr, Sherif and Al-Naymat, Ghazi, "Relational Processing of RDF Queries: A Survey," *SIGMOD Rec.*, vol. 38, no. 4, pp. 23–28, Jun. 2010.

[73] Kaoudi, Zoi and Manolescu, Ioana, "RDF in the Clouds: A Survey," *VLDBJ*, vol. 24, no. 1, pp. 67–91, 2015.

[74] Olivier Curé and Hubert Naacke and Mohamed Amine Baazizi and Bernd Amann, "On the Evaluation of RDF Distribution Algorithms Implemented over Apache Spark," *CoRR*, vol. abs/1507.02321, 2015.

[75] M. Tamer Özsu, "A Survey of RDF Data Management Systems," *CoRR*, vol. abs/1601.00707, 2016.

[76] Alexander Schätzle and Martin Przyjaciel-Zablocki and Simon Skilevic and Georg Lausen, "S2RDF: RDF Querying with SPARQL on Spark," *PVLDB*, vol. 9, no. 10, pp. 804–815, 2016.

[77] "Accumulo," https://accumulo.apache.org/.

[78] Hammoud, Mohammad and Rabbou, Dania Abed and Nouri, Reza and Beheshti, Seyed-Mehdi-Reza and Sakr, Sherif, "DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication," *PVLDB*, vol. 8, no. 6, pp. 654–665, 2015.

[79] Xiaofei Zhang and Lei Chen and Yongxin Tong and Min Wang, "EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud," in *ICDE*, 2013.

[80] Lu Wang and Yanghua Xiao and Bin Shao and Haixun Wang, "How to partition a billion-node graph," in *ICDE*, 2014.

[81] Bernstein, Philip A. and Chiu, Dah-Ming W., "Using Semi-Joins to Solve Relational Queries," *J. ACM*, vol. 28, no. 1, pp. 25–40, 1981.

[82] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *SIGMOD*, 2012.

[83] Z. Chong, H. Chen, Z. Zhang, H. Shu, G. Qi, and A. Zhou, "RDF pattern matching using sortable views," in *CIKM*, 2012.

[84] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, "View selection in Semantic Web databases," *PVLDB*, vol. 5, no. 2, 2011.

[85] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis, "Optimizing query shortcuts in RDF databases," in *ESWC*, 2011.

[86] Martin, Michael and Unbehauen, Jörg and Auer, Sören, "Improving the Performance of Semantic Web Applications with SPARQL Query Caching," in *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part II*, ser. ESWC, 2010.

[87] Papailiou, Nikolaos and Tsoumakos, Dimitrios and Karras, Panagiotis and Koziris, Nectarios, "Graph-Aware, Workload-Adaptive SPARQL Query Caching," in *SIGMOD*, 2015.

[88] S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in *CIDR*, 2007.

[89] Alagiannis, Ioannis and Borovica, Renata and Branco, Miguel and Idreos, Stratos and Ailamaki, Anastasia, "NoDB in Action: Adaptive Query Processing on Raw Data," *PVLDB*, vol. 5, no. 12, pp. 1942–1945, 2012.

[90] Alagiannis, Ioannis and Borovica-Gajic, Renata and Branco, Miguel and Idreos, Stratos and Ailamaki, Anastasia, "NoDB: Efficient Query Execution on Raw Data Files," *Commun. ACM*, vol. 58, no. 12, pp. 112–121, 2015.

[91] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1-2, 2010.

[92] Quamar, Abdul and Kumar, K. Ashwin and Deshpande, Amol, "SWORD: Scalable Workload-aware Data Placement for Transactional Workloads," ser. EDBT, 2013.

[93] Kumar, K. Ashwin and Quamar, Abdul and Deshpande, Amol and Khuller, Samir, "SWORD: Workload-aware Data Placement and Replica Selection for Cloud Data Management Systems," *VLDBJ*, vol. 23, no. 6, pp. 845–870, 2014.

[94] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an Architectural Era: (It's Time for a Complete Rewrite)," in *PVLDB*, 2007.

[95] Sudipto Das and Amr El Abbadi and Divyakant Agrawal, "ElasTraS: An Elastic Transactional Data Store in the Cloud," in *HotCloud*, 2009.

[96] Sudipto Das and Divyakant Agrawal and Amr El Abbadi, "ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud," *ACM Trans. Database Syst.*, vol. 38, no. 1, p. 5, 2013.

[97] ——, "G-Store: a scalable data store for transactional multi key access in the cloud," in *SoCC*, 2010.

[98] Schätzle, Alexander and Przyjaciel-Zablocki, Martin and Lausen, Georg, "PigSPARQL: Mapping sparql to pig latin," in *SWIM*. ACM, 2011, p. 4.

[99] Seo, Jiwon and Park, Jongsoo and Shin, Jaeho and Lam, Monica S, "Distributed SociaLite: A datalog-based language for large-scale graph analysis," *PVLDB*, vol. 6, no. 14, pp. 1906–1917, 2013.

[100] Goodman, Eric L. and Grunwald, Dirk, "Using Vertex-centric Programming Platforms to Implement SPARQL Queries on Large Graphs," in *IA3*, 2014, pp. 25–32.

[101] "Blazegraph," https://www.blazegraph.com/.

[102] "Blazegraph GAS," https://wiki.blazegraph.com/wiki/index.php/RDF_GAS_API.

[103] J. Gao, C. Zhou, J. Zhou and J. Yu, "Continuous Pattern Detection over Billion-Edge Graph Using Distributed Framework," in *ICDE*, 2014.

[104] Fard, Arash and Nisar, M.Usman and Ramaswamy, Lakshmish and Miller, John A. and Saltz, Matthew, "A distributed vertex-centric approach for pattern matching in massive graphs," in *IEEE BigData*, 2013.

[105] M. Sarwat, S. Elnikety, Y. He and M. Mokbel, "Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs," *PVLDB*, vol. 6, no. 14, pp. 1918–1929, 2013.

[106] Jindal, Alekh and Rawlani, Praynaa and Wu, Eugene and Madden, Samuel and Deshpande, Amol and Stonebraker, Mike, "Vertexica: Your Relational Friend for Graph Analytics!" *PVLDB*, vol. 7, no. 13, pp. 1669–1672, 2014.

[107] Forum, Message P, "MPI: A Message-Passing Interface Standard," Knoxville, TN, USA, Tech. Rep., 1994.

[108] Elnozahy, E. N. (Mootaz) and Alvisi, Lorenzo and Wang, Yi-Min and Johnson, David B., "A Survey of Rollback-recovery Protocols in Message-passing Systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[109] Shen, Yanyan and Chen, Gang and Jagadish, H. V. and Lu, Wei and Ooi, Beng Chin and Tudor, Bogdan Marius, "Fast Failure Recovery in Distributed Graph Processing Systems," *PVLDB*, vol. 8, no. 4, 2014.

[110] Blanas, Spyros and Li, Yinan and Patel, Jignesh M., "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *SIGMOD*, 2011.

[111] Bol'shev, L. and Ubaidullaeva, M., "Chauvenet's Test in the Classical Theory of Errors," *Theory of Probability & Its Applications*, vol. 19, no. 4, pp. 683–692, 1975.

[112] R. S. Boyer and J. Strother Moore, "MJRTY: A Fast Majority Vote Algorithm," in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, R. S. Boyer, Ed. London: Kluwer, 1991, pp. 105–118.

[113] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," in *SSDBM*, 2013.

[114] "Apache Pig," https://pig.apache.org/.

[115] "Apache Giraph," https://giraph.apache.org/.

[116] J. Edmonds and E. L. Johnson, "Matching, euler tours and the chinese postman," *Mathematical Programming*, vol. 5, no. 1, pp. 88–124, 1973.

[117] Neumann, Thomas and Moerkotte, Guido, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2011.

[118] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *ICDM*. IEEE, 2009, pp. 229–238.

[119] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams and P. Kalnis, "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing," in *EuroSys*, 2013, pp. 169–182.

[120] Jacopo Urbani and Sourav Dutta and Sairam Gurajada and Gerhard Weikum, "KOGNAC: Efficient Encoding of Large Knowledge Graphs," *CoRR*, vol. abs/1604.04795, 2016.

[121] Leis, Viktor and Boncz, Peter and Kemper, Alfons and Neumann, Thomas, "Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age," in *SIGMOD*, 2014.

[122] S. H. Eric Prudhommeaux and A. Seaborne, "SPARQL 1.1 Query Language," http://www.w3.org/TR/sparql11-query/, 2014.

[123] Arenas, Marcelo and Conca, Sebastián and Pérez, Jorge, "Counting Beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard," in *WWW*, 2012.

[124] Le, Wangchao and Kementsietsidis, Anastasios and Duan, Songyun and Li, Feifei, "Scalable multi-query optimization for SPARQL," in *ICDE*, 2012.

APPENDICES

A.1 LUBM Benchmark Queries

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#}>
PREFIX ub:   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

Q1: SELECT ?X WHERE {
    ?X rdf:type ub:GraduateStudent .
    ?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> .
}

Q2: SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:GraduateStudent .
    ?Y rdf:type ub:University .
    ?Z rdf:type ub:Department .
    ?X ub:memberOf ?Z .
    ?Z ub:subOrganizationOf ?Y .
    ?X ub:undergraduateDegreeFrom ?Y .
}

Q3: SELECT ?X WHERE {
    ?X rdf:type ub:Publication .
    ?X ub:publicationAuthor <http://www.Department0.University0.edu/AssistantProfessor0> .
}

Q4: SELECT ?X, ?Y1, ?Y2, ?Y3 WHERE {
    ?X rdf:type ub:AssociateProfessor .
    ?X ub:worksFor <http://www.Department0.University0.edu}> .
    ?X ub:name ?Y1 .
    ?X ub:emailAddress ?Y2 .
    ?X ub:telephone ?Y3 .
}

Q5: SELECT ?X WHERE {
```

```
    ?X rdf:type ub:UndergraduateStudent .
    ?X ub:memberOf <http://www.Department0.University0.edu}> .
}

Q6: SELECT ?X WHERE {
        ?X rdf:type ub:UndergraduateStudent .
}
Q7: SELECT ?X, ?Y WHERE {
    ?X rdf:type ub:UndergraduateStudent .
    ?Y rdf:type ub:Course .
    ?X ub:takesCourse ?Y .
    <http://www.Department0.University0.edu/AssociateProfessor0> ub:teacherOf ?Y .
}

Q8: SELECT ?X, ?Y, ?Z WHERE {
    ?X rdf:type ub:UndergraduateStudent .
    ?Y rdf:type ub:Department .
    ?X ub:memberOf ?Y .
    ?Y ub:subOrganizationOf <http://www.University0.edu}> .
    ?X ub:emailAddress ?Z .
}

Q9: SELECT ?X, ?Y, ?Z WHERE {
    ?X rdf:type ub:GraduateStudent} .
    ?Y rdf:type ub:AssociateProfessor .
    ?Z rdf:type ub:GraduateCourse .
    ?X ub:advisor ?Y .
    ?Y ub:teacherOf ?Z .
    ?X ub:takesCourse ?Z .
}

Q10: SELECT ?X  WHERE {
    ?X rdf:type ub:TeachingAssistant} .
    ?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> .
}

Q11: SELECT ?X WHERE {
    ?X rdf:type ub:ResearchGroup .
    ?X ub:subOrganizationOf ?Z .
    ?Z ub:subOrganizationOf <http://www.University0.edu> .
}
```

```
Q12: SELECT ?X, ?Y WHERE {
    ?Y rdf:type ub:Department .
    ?X ub:headOf ?Y.
    ?Y ub:subOrganizationOf <http://www.University0.edu> .
}

Q13: SELECT ?X WHERE {
    ?X rdf:type ub:GraduateStudent .
    ?X ub:undergraduateDegreeFrom <http://www.University0.edu> .
}

Q14: SELECT ?X WHERE {
        ?X rdf:type ub:GraduateStudent .
}

P:   SELECT ?y ?z WHERE {
        ?z ub:subOrganizationOf ?y .
        ?x ub:advisor ?t .
        ?z rdf:type ub:Department .
        ?x ub:memberOf ?z .
        ?x rdf:type ub:GraduateStudent .
        ?t ub:worksFor ?z .
        ?y rdf:type ub:University .
        ?x ub:undergraduateDegreeFrom ?y .
        ?t ub:mastersDegreeFrom ?y .
}

D:   SELECT ?y ?z WHERE {
        ?z ub:subOrganizationOf ?y .
        ?z rdf:type ub:Department .
        ?x ub:memberOf ?z .
        ?x rdf:type ub:GraduateStudent .
        ?x ub:undergraduateDegreeFrom ?y .
        ?y rdf:type ub:University .
        ?x ub:advisor ?t .
        ?t ub:worksFor ?z .
}
```

A.2 LUBM Workload

A workload of 20,000 queries is generated from LUBM benchmark queries shown in A.1. For queries that do not have constants (Q2 and Q9), different query patterns are generated by removing some triples and mutating the node types. For example, in Q2, 18 different patterns are generated by alternating student type between UndergraduateStudent and GraduateStudent (see Table A.1). Similarly, other query patterns are generated by removing different combinations of the query triple patterns. No variations are generated from Q6 and Q14 as they have only one triple pattern (*rdf:type*) with a single constant. For the rest of the queries, 1000 different patterns are generated from each query by varying the values of the query constants. For example, in Q1, different query patterns are generated by varying the values of both student type (UndergraduateStudent or GraduateStudent) and graduate courses.

Table A.1: LUBM Workload

	Patterns	**Changes**
Q1	1000	Constants
Q2	18	Structure/Constants
Q3	1000	Constants
Q4	1000	Constants
Q5	1000	Constants
Q6	1	No Changes
Q7	1000	Constants
Q8	1000	Constants
Q9	30	Structure/Constants
Q10	1000	Constants
Q11	1000	Constants
Q12	1000	Constants
Q13	1000	Constants
Q14	1	No Changes

A.3 YAGO2 Queries

```
PREFIX y: <http://yago-knowledge.org/resource/>
```

```
Y1: SELECT ?GivenName ?FamilyName WHERE {
    ?p y:hasGivenName ?GivenName .
    ?p y:hasFamilyName ?FamilyName .
    ?p y:wasBornIn ?city .
    ?p y:hasAcademicAdvisor ?a .
    ?a y:wasBornIn ?city .
}

Y2: SELECT ?GivenName ?FamilyName WHERE {
    ?p y:hasGivenName ?GivenName .
    ?p y:hasFamilyName ?FamilyName .
    ?p y:wasBornIn ?city .
    ?p y:hasAcademicAdvisor ?a .
    ?a y:wasBornIn ?city .
    ?p y:isMarriedTo ?p2 .
    ?p2 y:wasBornIn ?city .
}

Y3: SELECT ?name1 ?name2 WHERE {
    ?a1 y:hasPreferredName ?name1 .
    ?a2 y:hasPreferredName ?name2 .
    ?a1 y:actedIn ?movie .
    ?a2 y:actedIn ?movie .
}

Y4: SELECT ?name1 ?name2 WHERE {
    ?p1 y:hasPreferredName ?name1 .
    ?p2 y:hasPreferredName ?name2 .
    ?p1 y:isMarriedTo ?p2 .
    ?p1 y:wasBornIn ?city .
    ?p2 y:wasBornIn ?city .
}
```

A.4 Bio2RDF

```
PREFIX pharmkb: <http://bio2rdf.org/pharmgkb_vocabulary>
PREFIX irefindex: <http://bio2rdf.org/irefindex_vocabulary>
PREFIX pubmd: <http://bio2rdf.org/pubmed_vocabulary>
PREFIX pubmdrc: <http://bio2rdf.org/pubmed_resource>
PREFIX omim:<http://bio2rdf.org/omim_vocabulary>
PREFIX drug:<http://bio2rdf.org/drugbank>
```

```
PREFIX uniprot:<http://bio2rdf.org/uniprot>

B1: SELECT ?o WHERE {
    pubmdrc:1374967_INVESTIGATOR_1 pubmd:last_name ?o .
    pubmdrc:1374967_AUTHOR_1 pubmd:last_name ?o .
}

B2: SELECT ?articleToMesh WHERE {
    <http://bio2rdf.org/pubmed:126183> pubmd:mesh_heading ?articleToMesh .
    ?articleToMesh pubmd:mesh_descriptor_name ?mesh .
}

B3: SELECT ?phenotype WHERE {
    ?phenotype rdf:type omim:Phenotype .
    ?phenotype rdfs:label ?label .
    ?gene omim:phenotype ?phenotype .
}

B4: SELECT ?pharmgkbid WHERE {
    ?pharmgkbid pharmkb:xref drug:DB00126 .
    ?pharmgkbid pharmkb:xref ?pccid .
    ?DDIassociation pharmkb:chemical ?pccid .
    ?DDIassociation pharmkb:event ?DDIevent .
    ?DDIassociation pharmkb:chemical ?drug2 .
    ?DDIassociation pharmkb:p-value ?pvalue .
}

B5: SELECT ?interaction WHERE {
    ?interaction irefindex:interactor_a uniprot:O17680 .
}
```

A.5 Software

Table A.2 shows references to the systems discussed in this dissertation. Particularly, the table shows the web page URL for each project as well as an approximate number of lines of codes (LOC) written for each system. For AdPart, utility code for RDF data and SPARQL parsing was excluded. Similarly, for SPARTex, the LOC does not include GPS code. However, for completeness, the LOC for the three use case

Table A.2: Software Details

	URL	LOC
AdPart	`https://cloud.kaust.edu.sa/Pages/adpart.aspx`	≈ 8K
SPARTex	`https://cloud.kaust.edu.sa/Pages/spartex.aspx`	≈ 9K

discussed in Chapter 5 is included.