Scalable Discovery and Analytics on Web Linked Data

Dissertation by
Ibrahim Hosny Mohamed Abdelaziz

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

May, 2018
The dissertation of Ibrahim Abdelaziz is approved by the examination committee

Committee Chairperson: Panos Kalnis, Professor
Committee Members: Hakan Bagci - Associate Professor, Marco Canini - Assistant Professor, Matthias Renz - Associate Professor
ABSTRACT

Scalable Discovery and Analytics on Web Linked Data
Ibrahim Abdelaziz

Resource Description Framework (RDF) provides a simple way for expressing facts across the web, leading to Web linked data. Several distributed and federated RDF systems have emerged to handle the massive amounts of RDF data available nowadays. Distributed systems are optimized to query massive datasets that appear as a single graph, while federated systems are designed to query hundreds of decentralized and interlinked graphs.

This thesis starts with a comprehensive experimental study of the state-of-the-art RDF systems. It identifies a set of research problems for improving the state-of-the-art, including: supporting the emerging RDF analytics required by many modern applications, querying linked data at scale, and enabling discovery on linked data. Addressing these problems is the focus of this thesis.

First, we propose Spartex; a versatile framework for complex RDF analytics. Spartex extends SPARQL to seamlessly combine generic graph algorithms with SPARQL queries. Spartex implements a generic SPARQL operator as a vertex-centric program that interprets SPARQL queries and executes them efficiently using a built-in optimizer. We demonstrate that Spartex scales to datasets with billions of edges, and is at least as fast as the state-of-the-art specialized RDF engines. For analytical tasks, Spartex is an order of magnitude faster than existing alternatives.

To address the scalability limitation of federated RDF engines, we propose Lusail; a scalable system for querying geo-distributed RDF graphs. Lusail follows a two-tier strategy: (i) locality-aware decomposition of the query into subqueries to
maximize the computations at the endpoints and minimize intermediary results, and
(ii) selectivity-aware execution to reduce network latency and increase parallelism.
Our experiments on billions of triples show that Lusail outperforms existing systems
by orders of magnitude in scalability and response time.

Finally, enabling discovery on linked data is challenging due to the prior knowledge
required to formulate SPARQL queries. To address these challenges; we develop
novel techniques to (i) predict semantically equivalent SPARQL queries from a set
of keywords by leveraging word embeddings, and (ii) generate fine-grained and non-
blocking query plans to get fast and early results.
# TABLE OF CONTENTS

Examination Committee Page 2

Copyright 3

Abstract 4

List of Figures 10

List of Tables 13

1 Introduction 15
   1.1 Limitations of Existing Systems .......................... 18
      1.1.1 Limited Support for Rich RDF Data Analytics ......... 18
      1.1.2 Non-Scalable Querying over Decentralized RDF Graphs . 19
      1.1.3 Unsupported Data Discovery over Linked Data ......... 19
   1.2 Contributions and Thesis Organization .................... 20

2 A Survey and Experimental Comparison of Large Scale SPARQL Engines 23
   2.1 Introduction ............................................. 23
   2.2 Background .............................................. 25
      2.2.1 Single-machine RDF stores .............................. 26
   2.3 Distributed RDF Systems .................................. 28
      2.3.1 MapReduce and Graph Based Systems .................... 29
      2.3.2 Specialized RDF Systems ............................... 32
   2.4 Federated Query Processing ................................. 39
      2.4.1 Index-Based Federated Engines .......................... 40
      2.4.2 Index-Free Federated Engines ........................... 41
   2.5 Experimental Evaluation ................................... 42
      2.5.1 System Setup ........................................... 42
      2.5.2 Startup Overhead ....................................... 45
      2.5.3 Initial Replication Cost ............................... 47
4.4.2 Evaluation of Subqueries .................................................. 115
4.5 Experimental Study .......................................................... 118
  4.5.1 Evaluation Setup ......................................................... 118
  4.5.2 Lusail on a Local Cluster ............................................. 121
  4.5.3 Lusail in a Geo-Distributed Setting ................................. 125
  4.5.4 Analyzing Lusail ......................................................... 127
4.6 Conclusion ................................................................. 133

5 Towards Data Discovery on Web Linked Data .......................... 134
  5.1 Introduction ................................................................. 134
  5.2 System Architecture ....................................................... 137
  5.3 Discovery Support on Lusail ............................................ 138
    5.3.1 Keywords to SPARQL Queries ..................................... 138
    5.3.2 Fine-grained Planner ................................................. 142
    5.3.3 Multi-Query Optimization ........................................ 143
  5.4 Evaluation ................................................................. 144
    5.4.1 Keyword to SPARQL Queries ..................................... 144
    5.4.2 Fast and Early Results ............................................ 146
    5.4.3 Multi-Query Optimizations ........................................ 148
  5.5 Conclusion ................................................................. 148

6 Concluding Remarks ....................................................... 150
  6.1 Summary of Contributions ............................................... 150
  6.2 Future Research Directions ............................................ 152

References ................................................................. 154

Appendices ................................................................. 163
  A.1 Spartex Appendix ....................................................... 164
    A.1.1 Implementation Details ........................................... 164
    A.1.2 Queries ............................................................... 165
  A.2 Lusail Queries .......................................................... 167
    A.2.1 Bio2RDF Endpoints and Queries ................................. 167
    A.2.2 The Drug Query Based on QFed ................................ 168
    A.2.3 LUBM Queries ....................................................... 169
  A.3 Keyword Queries ........................................................ 171
    A.3.1 Keyword Queries over DBpedia .................................. 171
A.3.2 Keyword Queries over Multiple Interlinked Datasets . . . . . . 171
LIST OF FIGURES

1.1 Example RDF graph consisting of interconnected triples like \( \langle \text{Ben}, \text{takes}, \text{Databases} \rangle \). ........................................... 16
1.2 \( Q_a \) finds the advisors of the students who take Databases. ........... 16

2.1 An example RDF graph. Each edge and its associated vertices correspond to an RDF triple; e.g., \( \langle \text{Bill}, \text{worksFor}, \text{CS} \rangle \). .................. 25
2.2 A SPARQL query that finds CS professors with their advisees. ......... 26
2.3 Vertical partitions of the data of Fig. 2.1. ................................. 30
2.4 Trinity.RDF Graph Exploration Execution Plan for \( Q_{prof} \) using the data in Figure 2.2. ............................................. 33
2.5 Adpart-NA Query Execution Plan for the SPARQL query in Figure 2.2. 34
2.6 An example of semantic hash partitioning with forward direction and \( k = 1 \). ................................................................. 37
2.7 A summary graph for the RDF in Figure 2.1. .............................. 37
2.8 Data Scalability ............................................................. 54
2.9 Strong Scalability using LUBM-10240. ..................................... 55
2.10 Cumulative execution time of a workload. ................................. 56
2.11 Sensitivity of FedX to varying endpoints. .................................. 57

3.1 Example query \( Q_s \). Solid lines are part of the original graph while dotted lines represent triples that needs to be computed first. ....... 68
3.2 Spartex Architecture. ......................................................... 71
3.3 Computation iterations for solving \( Q_s \). ..................................... 76
3.4 Query Optimization; LUBM-4000 ............................................. 87
3.5 Data Scalability using LUBM dataset. ...................................... 88
3.6 Machine Scalability (LUBM-10240). ....................................... 88
3.7 Rich RDF Analytics: In the first use case, graph analytics output is utilized in a SPARQL query while in the second case, the output of a SPARQL query is used as input to a graph analytics algorithm. .... 89
3.8 Use case 3: SamplID analytics pipeline. .................................... 90
4.1 A decentralized graph for two universities managed by independent geo-distributed SPARQL endpoints (EP). The red dotted line represents an interlink between endpoints, i.e., a vertex in an endpoint referring to another vertex in another endpoint. Thus, to get the address of the university from which Tim got his PhD, the interlink from EP2 to EP1 must be traversed.

4.2 A SPARQL query, $Q_a$, over a decentralized RDF graph across different universities. This query has to traverse the interlink between EP2 and EP1.

4.3 The Lusail system architecture.

4.4 Locality analysis of data instances in EP1 and EP2 from Figure 4.1 that match $?S$, $?U$, and $?P$ in a pair of triple patterns in $Q_a$.

4.5 A Lusail SPARQL check query to detect whether $?P$ is a global join variable or not. The check query returns zero or only one value.

4.6 Two possible decompositions of $Q_a$, where the GJVs are $?U$ and $?P$. Any pair of predicates, which causes a variable to be a GJV, cannot be in the same subquery.

4.7 Query Evaluation in Lusail.

4.8 QFed Benchmark: Queries with Filter have high selectivity while Big literal queries have bigger intermediate data.

4.9 LUBM Benchmark: FedX and HiBISCuS evaluate queries one triple pattern at a time in a bound join. Lusail decomposes these queries based on the location of data instances.

4.10 LargeRDFBench: Most of the simple queries do not access large intermediate data, unlike the complex and large queries. X corresponds to time out while missing bars correspond to runtime errors.

4.11 Geo-distributed federation: endpoints are deployed in 7 different regions of the Azure cloud. Communication cost affects all systems, but Lusail can execute all queries and outperforms other systems.

4.12 Profiling Lusail by varying query complexity, the number of endpoints, and the data size.

4.13 Evaluating different threshold values for identifying subqueries to delay.

4.14 Lusail optimizations can move computations on intermediate data to the endpoints, retrieve less irrelevant data, and achieve orders of magnitudes speedup compared to FedX. X corresponds to time out.
5.1 A SPARQL query to find US presidents and associated news articles in the linked web data. ................................. 135
5.2 The Lusail discovery system. .................................. 137
5.3 Lusail pipeline for predicting a set of SPARQL queries from a list of keywords. ............................................. 139
5.4 Predicated SPARQL queries to find "news article and party of Barack Obama". .................................................. 141
5.5 Effect of Heterogeneity on Lusail Performance. Endpoints are deployed on machines from the Azure public cloud. ................. 147
5.6 Effect of MQO for different LUBM workload sizes evaluated on 256 university endpoints. ........................................ 148
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of state-of-the-art distributed RDF systems.</td>
<td>28</td>
</tr>
<tr>
<td>2.2</td>
<td>1-hop undirected guarantee partitioning of the RDF in Figure 2.1. subject(s) and object(s) highlighted in blue are owned by W1; the rest belong to W2. Replicated triples are highlighted in yellow.</td>
<td>36</td>
</tr>
<tr>
<td>2.3</td>
<td>Datasets; ( M ): millions. #S, #P, #O denote number of distinct subjects, predicates and objects.</td>
<td>42</td>
</tr>
<tr>
<td>2.4</td>
<td>Partitioning configurations and replication ratio.</td>
<td>44</td>
</tr>
<tr>
<td>2.5</td>
<td>Preprocessing time (Minutes).</td>
<td>45</td>
</tr>
<tr>
<td>2.6</td>
<td>Runtime for LUBM-10240 queries (ms). SM: Single Machine, MR: MapReduce, and SS: Specialized systems. S2X failed to execute all queries; gStore and gStoreD could not preprocess the data within 24 hours.</td>
<td>48</td>
</tr>
<tr>
<td>2.7</td>
<td>Query runtimes (ms) for WatDiv and YAGO2 datasets. SHARD and gStoreD crashed in several queries while DREAM could not finish WatDiv queries within 24 hours.</td>
<td>51</td>
</tr>
<tr>
<td>2.8</td>
<td>Query runtimes for Bio2RDF (ms). gStore, gStoreD, RDF-3X, H-RDF-3X, SHAPE, S2RDF and DREAM could not finish the data preprocessing within 24 hours while S2X failed to execute all queries.</td>
<td>53</td>
</tr>
<tr>
<td>3.1</td>
<td>Dataset statistics in millions (M). Number of triples, unique subjects, objects and predicates.</td>
<td>83</td>
</tr>
<tr>
<td>3.2</td>
<td>Query runtime for LUBM-10240 (seconds)</td>
<td>84</td>
</tr>
<tr>
<td>3.3</td>
<td>Query runtimes for YAGO2 (sec)</td>
<td>85</td>
</tr>
<tr>
<td>3.4</td>
<td>Query runtimes for Bio2RDF (sec)</td>
<td>85</td>
</tr>
<tr>
<td>3.5</td>
<td>Query engine. LUBM-10240; runtime (sec)</td>
<td>91</td>
</tr>
<tr>
<td>3.6</td>
<td>Query engine. YAGO2; runtime (msec)</td>
<td>92</td>
</tr>
<tr>
<td>3.7</td>
<td>Query engine. Bio2RDF; runtime (msec)</td>
<td>92</td>
</tr>
<tr>
<td>4.1</td>
<td>Datasets used in experiments.</td>
<td>119</td>
</tr>
<tr>
<td>4.2</td>
<td>Query runtimes (sec) on real endpoints. ZR: zero results error, RE: runtime exception.</td>
<td>127</td>
</tr>
</tbody>
</table>
5.1 QALD6 Benchmark .................................................. 145
5.2 Using Lusail-KeyWordSearch to solve Keyword Queries over Multiple Interlinked Datasets ........................................ 146
A.1 QALD6 Benchmark .................................................. 171
Chapter 1

Introduction

The Resource Description Framework (RDF) [1] is a versatile data model that provides a simple way to express facts in the semantic web. RDF data are collections of \( \langle \text{subject}, \text{predicate}, \text{object} \rangle \) triples that form directed labeled graphs; an example is shown in Figure 1.1. Due to its versatility, many communities are adopting the RDF model for publishing their data. For example, many public knowledge bases like DBpedia [2], UniProt [3], PubChemRDF\(^1\) and Bio2RDF\(^2\) contain billions of facts in RDF. Traditionally, RDF data are searched for subgraphs that match specific patterns. Such queries are expressed in SPARQL\(^3\) which consist of a set of RDF triple patterns, where some of the columns are variables. For example, query \( Q_a \) in Figure 1.2(a) retrieves all the advisors of students who take the Databases course. The query corresponds to the graph pattern in Figure 1.2(b). The answer is the set of bindings of \(?stud\) and \(?prof\) that render the query graph isomorphic to subgraphs in the data. In Figure 1, the answer of \( Q_a \) is \((?stud, ?prof) \in \{(John, Fred), (Ben, Lisa)\}\).

The emergence of these massive amounts of RDF data offer unique opportunities for data scientists and researchers to make discoveries and breakthroughs using SPARQL. There are three categories of RDF applications; (i) many RDF datasets exist as a single massive RDF graph; e.g. DBPedia, UniProt and PubChemRDF. A growing number of applications utilize SPARQL to query biological datasets for discovering associated diseases [4, 5], finding melanoma drugs [6], determining side ef-

\(^1\)http://pubchem.ncbi.nlm.nih.gov/rdf
\(^2\)http://bio2rdf.org
\(^3\)http://www.w3.org/TR/rdf-sparql-query
Figure 1.1: Example RDF graph consisting of interconnected triples like \((\text{Ben}, \text{takes}, \text{Databases})\).

\[
\text{SELECT} \ ?\text{stud} \ ?\text{prof} \ \text{WHERE}\{ \\
\ ?\text{stud} \ \text{takes} \ \text{Databases} . \\
\ ?\text{stud} \ \text{advisor} \ ?\text{prof} . \\
\} \\
\]

(a) SPARQL (b) Graph

Figure 1.2: \(Q_a\) finds the advisors of the students who take Databases.

effects associated with certain targets [7] and reporting drug-metabolizing enzymes [8]. Some of these datasets are massive such that processing it on a single machine is infeasible. For example, PubChemRDF [9] and UniProt [10] have more than 100 and 34 billion triples, respectively. Therefore, distributed systems that are capable of utilizing a cluster of machines efficiently are required. (ii) The second category of applications [11, 12, 13, 14] goes beyond simple pattern matching, and requires complex pipelines that combine a variety of graph algorithms with SPARQL. These applications require systems that are capable of performing both types of computations; generic graph algorithms; e.g., PageRank, SSSP and Betweenness centrality, and SPARQL querying. Finally, (iii) decentralized RDF graphs nowadays consist of more than 85 billions triples distributed over more than 3400 datasets⁴ in different domains, such as media, government, and life sciences [15]. In life sciences, Bio2RDF [16] has decentralized graphs of about 11 billions triples across 35 datasets. In decentral-

⁴http://stats.lod2.eu/
ized social networks [17, 18, 19], users store their data in their own RDF dataset and each user’s dataset is connected with remote datasets of other users. Applications in this category require systems that can evaluate SPARQL queries efficiently and is able to query hundreds or even thousands of datasets.

Motivated by the need for these applications, three types of systems have emerged. First, to support these massive amounts of data and to overcome the scalability limitations of single machines, many distributed SPARQL query engines [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32] have been introduced. They utilize shared-nothing computing clusters, partition the RDF graph among multiple machines (a.k.a., workers) to handle big datasets, and parallelize query execution to reduce the running time. Answering queries typically involves processing of local data at each worker, interleaved with data exchange among workers. The main challenge of distributed systems is scaling-out, which is affected by various factors including data partitioning, load balancing, indexing, query optimization and communication overhead. Second, to support generic graph computations on top of RDF data; two types of systems have to be utilized; Specialized RDF Engines [33, 34, 30, 25] and Generic Graph Frameworks [35, 36, 37, 38]. The reason is that specialized RDF engines are capable of evaluating SPARQL queries only while graph frameworks can perform generic graph computations efficiently. Third, a federated RDF engine [39, 40, 41, 42] is required to evaluate queries over decentralized RDF graphs. These engines does not assume any control over the data, since the data is accessible through independent, remote SPARQL endpoints.

Despite the plethora of distributed/federated RDF systems and their practical applications, there is limited information about their comparative performance and scalability. These systems exhibit a variety of trade-offs that are not well-understood, due to the lack of any comprehensive quantitative and qualitative evaluation. This thesis starts by providing a comprehensive experimental evaluation of state-of-the-
art RDF systems using a variety of very large real and synthetic datasets and query loads; distributed, analytical and decentralized. Based on this extensive evaluation, we identify a set of research directions for optimizing state-of-the-art RDF systems. These limitations are summarized as follows.

1.1 Limitations of Existing Systems

1.1.1 Limited Support for Rich RDF Data Analytics

A growing number of modern applications [11, 12, 13, 14] require sophisticated analytics on RDF graphs that combine structural queries with generic graph computations. For example, Qu et al. [11] mine biological data to identify new targets for existing drugs. First, they run a set of SPARQL queries against biological databases. On the results they execute graph centrality algorithms to identify key biological entities. The state-of-the-art centralized [43, 34, 44] and distributed [45, 21, 24, 25, 23] RDF data management systems fail to support generic graph algorithms as they focus on SPARQL queries only. SPARQL lacks procedural capabilities; therefore, as noted in the work [46, 47, 48], expressing graph operations using SPARQL results in verbose and complex queries that are hard to formulate and expensive to evaluate. This is evident in some recent works [46, 47, 48] that try to express generic graph operations in SPARQL; these approaches are limited to only a small set of graph operations like clustering and graph diffusion. On the other hand, many vertex-centric graph management systems, such as Pregel [49], PowerGraph [50] and GRACE [36], have been proposed for efficient graph analytics. However, these systems lack the capability of evaluating general SPARQL queries, requiring users to write a separate vertex-centric program for each SPARQL query. Such an approach is counter-productive and requires prior knowledge about the data.
1.1.2 Non-Scalable Querying over Decentralized RDF Graphs

Linked Data refers to a set of best practices used to publish and connect data on the web. RDF is a key ingredient of Linked Data and provides a generic data model for describing and linking entities together. Linked Open Data (LOD) forms a virtually integrated distributed database where documents are stored at autonomous sites, some of which expose their data to users through a SPARQL endpoint. Distributed RDF systems [25, 26, 27, 28, 29, 30, 31, 32] assume full control over the RDF data where the data has to be downloaded first and then replicated and/or partitioned among different servers in the same cluster. With web-scale data, downloading the whole data into a centralized place and partitioning the data is an prohibitively expensive. Moreover, whenever a data update is available, these systems have to update its partitions and rebuild some indices which is a very expensive and time consuming process when handling large-scale data. On the other hand, federated query processing engines receive federated queries which have to be evaluated by querying several endpoints and linking their partial results together. Several approaches [51, 52] have been proposed to optimize query processing over a small number of heterogeneous data sources by utilizing a global schema. In the case of schema similarity and interlinks among sources, these approaches cause unnecessary data retrieval and communications, leading to poor scalability and response time.

1.1.3 Unsupported Data Discovery over Linked Data

Enabling data discovery over linked web requires a more intuitive way for searching RDF data; e.g. keyword-based search. Providing interactive keyword search querying over such an environment is very challenging. Existing systems provide keyword search over a single dataset. These systems require downloading the whole dataset in a centralized place and performing expensive preprocessing to build data indices or graph summaries. This expensive preprocessing phase is infeasible over such a
large scale of interconnected web data; e.g. LOD cloud. Furthermore, keyword-based search is an exploratory process where users may need to change their query multiple times to get the required answer. This means that waiting for long times for an answer is not plausible and query results have to be reported as early as possible.

1.2 Contributions and Thesis Organization

The rest of this thesis is organized as follows:

- **Chapter 2:** This chapter presents a survey [53] of 22 state-of-the-art systems that cover the entire spectrum of distributed RDF data processing and categorize them by several characteristics. Then, 12 representative systems are selected and used to perform extensive experimental evaluation with respect to pre-processing cost, query performance, scalability and workload adaptability, using a variety of synthetic and real large datasets with up to 4.3 billion triples. Our results provide valuable insights for practitioners to understand the trade-offs for their usage scenarios. We also build on a recent experimental study for federated RDF engines [54] by a conducting a scalability experiment where we systematically increase the number of endpoints. Based on our experimental evaluation, we identify a set of research problems towards optimizing the state-of-the-art large-scale RDF systems.

- **Chapter 3:** Existing systems support either declarative SPARQL queries, or generic graph processing, but not both. We bridge this gap by introducing Spartex [55, 56], an RDF analytics framework based on the vertex-centric computation model. It implements a generic SPARQL operator as a vertex-centric program that interprets SPARQL queries and executes them efficiently using a built-in optimizer. In Spartex, user-defined vertex centric programs can be invoked from SPARQL as stored procedures. Spartex allows the execution of
a pipeline of graph algorithms without the need for multiple reads/writes of input data and intermediate results. We present various scenarios where our framework simplifies significantly the implementation of complex RDF data analytics programs. We demonstrate that Spartex scales to datasets with billions of edges, and show that our core SPARQL engine is at least as fast as the state-of-the-art specialized RDF engines. For complex analytical tasks that combine generic graph processing with SPARQL, Spartex is at least an order of magnitude faster than existing alternatives.

- **Chapter 4**: A crucial factor for a good performance in federated query processing is pushing as much computation as possible to the local endpoints. Surprisingly, existing federated SPARQL engines are not effective at this task, since they rely only on schema information to push computation to the endpoints, and schema information is limited in its power. Therefore, these systems often cause unnecessary data retrieval and communication, leading to poor scalability and response time. To address these limitations, we propose Lusail [57, 58, 59], a scalable and efficient federated SPARQL system for querying large geo-distributed RDF graphs through different endpoints. Lusail uses a novel query rewriting algorithm to push computation to the local endpoints by relying on information about the RDF instances and not only the schema. The query rewriting algorithm has the additional advantage of exposing parallelism in query processing, which Lusail exploits through advanced scheduling at query run time. Our experiments using datasets with billions of triples show that Lusail outperforms state-of-the-art systems by orders of magnitude in terms of scalability and response time.

- **Chapter 5**: Enabling data discovery through keyword-based search on top of Lusail faces two challenges: (i) mapping keywords to corresponding SPARQL
queries without relying on a preprocessing phase is very challenging. (ii) keyword queries are exploration-based where users do not know yet what is the correct query formulation. So it is important to report query results as soon as they become ready. We propose data discovery modules [60] on top of Lusail for (i) a keyword search querying interface that does not require any expensive preprocessing phase. (ii) A non-blocking query execution module to produce query results as early as possible, and (iii) a fine-grained query optimizer to handle the heterogeneity existing in real deployments.

This thesis contains published work and work to be submitted. Specifically, the experimental study described in Chapter 2 is published in Very Large Data Bases Conference (VLDB 2017) [53]. Furthermore, the work in Chapter 3 (Spartex) is published in IEEE Transactions on Parallel and Distributed Systems (TPDS) [56]. The work described in Chapters 4 (Lusail) is published in Very Large Data Bases Conference (VLDB 2018) [59] while the work in Chapter 5 is to be submitted [60]. Both Spartex and Lusail have been demonstrated in Very Large Data Bases Conference (VLDB 2015) [55] and ACM International Conference on Management of Data (SIGMOD 2017) [58], respectively.
Chapter 2

A Survey and Experimental Comparison of Large Scale
SPARQL Engines

2.1 Introduction

Despite the plethora of large-scale distributed and federated RDF systems and their practical applications, there is limited information about their comparative performance. Some published surveys provide qualitative comparisons. For example, Sakr et al. [61] present an overview of using the relational model for RDF data. Svoboda et al. [62] classify state-of-the-art indexing approaches for linked data. Kaoudi et al. [63] survey RDF data management systems designed for the cloud. Ozsu [64] provides a broader overview of existing centralized and distributed RDF engines and discusses querying techniques for linked data. Finally, Ma et al. [65] survey techniques that use relational and NoSQL databases to store large RDF data. None of the aforementioned surveys contains any experimental evaluation. Saleem et al. [54] provided an extensive experimental comparison of state-of-the-art federated RDF engines considering query performance and source selection. However, it lacks a scalability study to assess how scalable are existing federated engines as the number of endpoints and accordingly the data size increases.

Motivated by the lack of quantitative comparison, in this chapter we provide a comprehensive experimental evaluation of state-of-the-art distributed RDF systems, using a variety of very large real datasets and query loads. We start with an extensive survey that covers 22 relevant systems. We describe the execution model and the
graph partitioning strategy of each system, discuss the similarities and differences and explain the various trade-offs. We also categorize the systems based on the: (i) underlying implementation framework (e.g., MapReduce, vertex-centric or proprietary distributed processing); (ii) use of generic joins versus specialized graph exploration; (iii) data replication strategy; and (iv) adaptivity to query workload.

Then, we perform extensive experimental evaluation of the following 12 representative systems: S2RDF [31], AdPart [30], DREAM [32], Urika-GD [66], CliqueSquare [29], S2X [67], TriAD [25], SHAPE [24], H-RDF-3X [20], H2RDF+ [23], SHARD [22] and gStoreD [68]. We use all the standard synthetic benchmarks (e.g., LUBM [69]) and a variety of very large real datasets (e.g., Bio2RDF [70]) with up to 4.3 billion triples, to stretch the systems to their limits. We analyse the following metrics: (i) **Startup cost**: it includes the time overhead of data partitioning, indexing, replication and loading to HDFS/memory, as well as the storage overhead of replication; (ii) **Query efficiency**: we utilize query workloads with varying query complexities and selectivities, and report the execution time; (iii) **Scalability** to very large datasets; and (iv) **Adaptability** to different workloads. Finally, we complement Saleem et al. [54] study by conducting a scalability experiment of state-of-the-art federated engines.

Our results suggest that specialized in-memory systems, such as AdPart [30] or TriAD [25] provide the best performance, assuming the data can fit in the cumulative memory of the computing cluster. If this condition is not satisfied, MapReduce based systems (e.g., H2RDF+ [23]), are an acceptable alternative. In contrast, the startup costs of some systems (e.g., S2RDF [31]) or the excessive replication (e.g., DREAM [32]), severely limit their applicability to large datasets. Our results also show that existing federated SPARQL engines cause unnecessary data retrieval and communication, leading to poor scalability and response time. In an attempt to standardize the evaluation of future systems and assist practitioners to select the ap-
Figure 2.1: An example RDF graph. Each edge and its associated vertices correspond to an RDF triple; e.g., \((Bill, worksFor, CS)\).

appropriate solution for their data and applications, we publish online all datasets, our evaluation methodology and links to the systems.

The rest of this chapter is organized as follows: Section 2.2 provides essential background on RDF and an overview of single machine RDF stores. Section 2.3 contains a survey of the 22 state-of-the-art distributed RDF systems while Section 2.4 provides an overview of existing federated RDF engines. Section 2.5 presents the experimental evaluation of the selected 12 distributed systems and a scalability experiment for federated engines. Finally, Section 2.6 concludes our findings.

2.2 Background

RDF [1] is a standard data model and the core component of the W3C Semantic Web. RDF datasets consist of triples \(\langle subject, predicate, object \rangle\), where the predicate (P) represents a relationship between a subject (S) and an object (O). RDF data can be viewed as a long relational table with three columns, or as a directed labeled graph with vertices reflecting the entities and edge labels as the predicates. Figure 2.1 shows an example RDF graph of students and professors in an academic network.

SPARQL [71] is the de-facto query language for RDF data. In its simplest form,
a.k.a. Basic Graph Pattern (BGP), a SPARQL query consists of a set of RDF triple patterns; some of the nodes in a pattern are variables that may appear in multiple patterns. For example, the query in Figure 2.2(a) returns all professors who work for CS with their advisees. The query corresponds to the graph pattern in Figure 2.2(b). The answer is the set of ordered bindings of (?prof, ?stud) that render the query graph isomorphic to subgraphs in the data. Assuming data are stored in a table \( D(s,p,o) \), the query can be answered by first decomposing it into two subqueries: \( q_1 \equiv \sigma_{p=\text{worksFor} \land o=CS}(D) \) and \( q_2 \equiv \sigma_{p=\text{advisor}}(D) \). The subqueries are 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 2.1, we get \( (?\text{prof}, ?\text{stud}) \in \{(\text{James}, \text{Lisa}), (\text{Bill}, \text{John}), (\text{Bill}, \text{Fred}), (\text{Bill}, \text{Lisa})\} \).

### 2.2.1 Single-machine RDF stores

We discuss the storage models of single-machine RDF systems since several distributed systems [32, 24, 20] rely on them for querying RDF data within a single partition.

**Triple Table** uses a single table with three columns corresponding to subject, predicate and object to store RDF data. An index is created per column for faster join evaluation. A query with several predicates corresponds to a set of self-joins on the large triple table. This approach scales poorly due to expensive self joins [72]. RDF-3X [73] and Hexastore [74] reduce this cost by using a set of indices that cover all possible permutations of S, P and O. These indices are stored as clustered B+-trees.
and are compressed using rigorous byte-level techniques. Aggregate indices, selectivity histograms and statistics about frequently accessed paths are used to select the lowest-cost execution plan. Their optimizer use order-preserving merge-joins for most of the operations and hash-joins for the last few ones.

**Property table** is a wider and flattened representation of RDF data [75]. The dimensions of this table are determined by the number of subjects and distinct predicates. Each cell in the table contains the object value of the corresponding subject and predicate. This representation has high storage overhead when the number of unique predicates is large due to its sparse representation. Moreover, it can not represent multi-valued attributes, i.e., a subject connected to different objects using the same predicate. Jena2 [76] solves this problem by introducing two alternative representations, namely, the clustered property table and property-class tables. BitMat [77] proposes an alternative representation of property tables that uses a compressed 3-dimensions bit-matrix. Each dimension of the bit-matrix corresponds to a part of an RDF triple; S, P and O. Each cell represents the existence of an RDF triple defined by the S,P,O positions. Queries are executed on the compressed data without materializing intermediate join results.

**Vertical Partitioning** is another representation for RDF data proposed by SW-Store [78]. The triples table is vertically partitioned into $n$ tables, where $n$ is the number of distinct predicates. A two columns table is created for each predicate where a row is a pair of subject-object values connected through the predicate. Tables are sorted on the subject to render subject lookup and merge joins faster. This approach stores multi-valued attributes as successive rows and does not store NULL values. It provides good performance for queries with bounded predicates, however, it requires scanning multiple tables to reconstruct information related to a single entity.
### 2.3 Distributed RDF Systems

Distributed RDF systems scale to large datasets by partitioning the RDF graph among many compute nodes (workers) and evaluating queries in a distributed fashion. Each SPARQL query is decomposed into multiple subqueries, which are then evaluated independently. Since the data is distributed, nodes may need to exchange intermediate results during query evaluation; therefore queries with large intermediate results incur high communication cost [25, 20]. To achieve acceptable response time, distributed systems attempt to minimize communication cost and maximize parallelism. This is accomplished by efficient data partitioning that maximizes data locality; load balancing that avoids stragglers; efficient join implementations; and the utilization of native RDF indexing techniques to speedup joins and index lookups.

We summarize in Table 2.1 the various features of existing distributed RDF systems. Note that Replication in this table points to systems that explicitly replicate RDF data, excluding HDFS replication.

![Table 2.1: Summary of state-of-the-art distributed RDF systems.](image)

In this survey, we categorize distributed RDF management systems along 2 dimensions based on their execution model: (i) MapReduce and Graph-based systems rely on general purpose frameworks, like Hadoop or Spark, that offer seamless data distribution and parallelization at the cost of flexibility. (ii) Specialized RDF systems are built specifically for SPARQL query evaluation, by utilizing custom physical lay-
outs, native RDF indexing, efficient communication protocols and explicit replication. Within this category, we define three subcategories based on the data partitioning scheme: lightweight, sophisticated and workload-aware partitioning. Systems based on sophisticated partitioning offer faster query execution at the cost of startup time and storage requirements. Workload-aware systems achieve faster query execution by adapting their data partitioning to the entire query workload.

2.3.1 MapReduce and Graph Based Systems

SHARD [22] is a triple store implemented on top of MapReduce [83]. The entire RDF dataset is stored in a single file within HDFS [84], where each line represents all triples of a single subject. The input dataset is hash-partitioned among workers such that each worker is responsible for a distinct set of triples. SHARD does not use indexing; consequently, during query evaluation it scans the entire dataset. SPARQL queries are executed as a sequence of MapReduce iterations. Each iteration is responsible for a single subquery, while the results are continuously joined with subsequent iterations. The final iteration is responsible for filtering the bounded variables and removing redundant results.

HadoopRDF [26] also uses HDFS to store the RDF data as flat files; replication and distribution is left to HDFS. Unlike SHARD, HadoopRDF uses multiple files on HDFS, a file for each predicate which is similar to SW-Store’s [78] vertical partitioning. HadoopRDF also splits each predicate file into multiple smaller files based on explicit and implicit type information. Initially, it divides the \texttt{rdf:type} file into as many files as the number of distinct objects. Then, a set of files are created for each type \texttt{type.object}. For example, \texttt{rdf:type} in Figure 2.3 results in only one file (\texttt{type.Grad}) because predicate \texttt{type} has only one object (\texttt{Grad}). Then, HadoopRDF divides the remaining predicate files into multiple files based on the object type. It splits triples based on the RDF class their object belongs to. For example, triple
$\langle \text{John}, \text{teacherOf}, \text{OS} \rangle$ will be stored in a file named $\text{teacherOf}\_\text{Course}$. To retrieve this implicit type information, it needs to join the predicate file with the type_\* files. HadoopRDF executes queries as a sequence of MapReduce iterations. It contains a query optimizer to select the query plan that minimizes the number of MapReduce iterations and the size of intermediate results.

CliqueSquare [29] exploits the replication of HDFS to maximize the efficiency of parallel joins and to reduce the communication cost. To achieve this, CliqueSquare partitions and stores the data in three different ways, by hashing each triple on the subject, predicate and object. Furthermore, it partitions the data within each machine into smaller files by applying property-based grouping similar to HadoopRDF. The integration of HDFS replication and partitioning enables CliqueSquare to perform all first-level joins (i.e., subject-subject, subject-predicate, etc.) locally in each machine. The CliqueSquare optimizer minimizes the number of joins by generating relatively shallow plans that use multi-way joins. It finds the possible clique decompositions of the query graph, where each clique corresponds to a multi-way join, and then selects the decomposition with the lowest cost. The resulting plan is executed as a sequence of MapReduce iterations.

H2RDF+ [23] is a distributed RDF engine based on MapReduce and Apache HBase [85]. It materializes all six permutations of RDF triples using HBase tables, which are sorted key-value maps. Data partitioning is left to HBase that range-partitions tables based on keys. Maintaining these indices offers several benefits: (i)
all SPARQL triple patterns can be answered efficiently by a single scan on the corresponding index; (ii) merge join can be employed to exploit the precomputed ordering in these indices; and (iii) every join between triple patterns is executed as merge-join. H2RDF+ maintains a set of aggregated statistics to estimate the selectivity of triple patterns, join results and join cost. It uses a greedy algorithm that finds at each execution step the join with the lowest cost. It uses multi-way merge join for sorted data, and sort-merge join for unsorted intermediate results. Simple queries are executed efficiently in a centralized fashion, while complex queries with large intermediate results are evaluated as a sequence of MapReduce jobs. H2RDF+ utilizes lazy materialization to minimize the size of the intermediate results.

S2X [67] exploits the inherited graph structure of RDF to process SPARQL as graph-based computations on top of GraphX. It uses the parallel vertex-centric model to evaluate the BGP matching of SPARQL while other operators, such as OPTIONAL and FILTER, are processed through Spark RDD operators. BGP matching starts by distributing all triple patterns to all graph vertices. Each vertex matches its edge labels with the triple’s predicate. Graph vertices cooperatively validate their triple candidacy with their direct neighbours by exchanging messages. Then, the partial result are collected and incrementally merged. S2X uses two string encoding types: hash and count-based. Hash-based encoding utilizes a 64-bit hash function to encode subjects and objects, while count-based assigns unique numeric values to them. S2X does not have a special RDF partitioner; it uses GraphX 2D hashing, which hashes on the encoded subject then hashes on the encoded object, to partition the input graph among workers.

Sedge [81] proposes similar techniques for SPARQL query execution on top of the vertex-centric processing model. The entire graph is replicated several times and each replica is partitioned differently. Each SPARQL query is executed against the replica that minimizes communication. Sedge does not provide automatic translation
of SPARQL queries to the vertex-centric model; a vertex-centric program has to be written manually for each query, which is counter-productive and requires prior knowledge about the data.

2.3.2 Specialized RDF Systems

2.3.2.1 Lightweight Partitioning

**Trinity.RDF** [21] is a distributed in-memory RDF engine that can handle large datasets. It represents RDF data in a graph form using adjacency lists, stored in the Trinity key-value store [38]. The graph is hash-partitioned on vertex-id; this is equivalent to partitioning the data twice, on subject and object. Trinity.RDF uses graph exploration for query evaluation. In every iteration, a single subquery is explored starting from the valid bindings in all workers. For example, Figure 2.4 shows how Trinity.RDF executes \( Q_{prof} \). Starting with the pattern \( \langle ?prof, worksFor, CS \rangle \), it explores the neighbours of CS connected via \( worksFor \). It finds that the possible binding for \( ?prof \) are James and Bill. In the next iteration, it starts to explore from nodes James and Bill via edge \( advisor \), and generates the bindings for \( \langle ?stud, advisor, ?prof \rangle \). Graph exploration avoids the generation of redundant intermediate results. 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 processing, which is a potential bottleneck.

**TriAD** [25] 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. It creates six in-memory tables on each machine, one for each permutation of subject, predicate, object. The six SPO permutations are arranged into two
groups; subject-key indices (SPO, SOP, PSO), and object-key indices (OSP, OPS, POS). Each of these indices is then hash-partitioned among different machines and sorted within each machine in lexicographic order. This enables TriAD to perform efficient distributed merge-joins over the different SPO indices. Multiple join operators are executed concurrently by all workers, which communicate via asynchronous message passing. At each compute node, TriAD uses multiple-threads to evaluate multiple operators in the query plan in parallel. TriAD shards one (both) relation(s) when evaluating distributed merge (hash) joins, which does not preserve the locality of intermediate results. This causes TriAD to re-shard intermediate results if the sharding column of the previous join is not the current join column. This cost is significant for large intermediate results with multiple attributes.

**AdPart-NA** [86, 30] employs lightweight partitioning that hashes triples on subject. Each worker stores its local set of triples using three in-memory data structures; P-index, PS-index and PO-index. P-index returns the set triples having the given predicate. Similarly, PS and PO indices return the nodes connected to the given predicate-subject or predicate-object, respectively. AdPart-NA exploits the query structure and the hash-based data locality in order to minimize the communication cost during query evaluation. It capitalizes on the subject-based locality and the locality of its intermediate results (pinning) to evaluate joins in parallel without com-
Figure 2.5: Adpart-NA Query Execution Plan for the SPARQL query in Figure 2.2.

Whenever possible, intermediate results are hash-distributed among workers instead of broadcasting to all workers. Figure 2.5 shows how AdPart-NA evaluates $Q_{\text{prof}}$ by a single subject-object join, assuming the following execution order: $q_1$: $\langle ?\text{prof}, \text{worksFor}, \text{CS} \rangle$, $q_2$: $\langle ?\text{stud}, \text{advisor}, ?\text{prof} \rangle$. For such queries, AdPart-NA employs distributed semi-join. Each worker scans its PO index to find all triples matching $q_1$, projects on join column $?\text{prof}$, and exchanges it with the other worker. Once the projected column is received, each worker computes semi-join $q_1 \bowtie_{?\text{prof}} q_2$ using its PO index. Specifically, $w_1$ probes $p = \text{advisor}, o = \text{Bill}$ while $w_2$ probes $p = \text{advisor}, o = \text{James}$. Finally, each worker computes $q_1 \bowtie_{?\text{prof}} q_2$.

DREAM [32] utilizes data replication instead of partitioning by building a single database that is replicated to all workers. It also avoids expensive intermediate data shuffling and only exchanges small auxiliary data. Each machine uses RDF-3X on its assigned data for statistics estimation and query evaluation. DREAM decomposes each query into multiple, usually non-overlapping, subqueries where each subquery is answered by a single worker. Depending on the query complexity, DREAM’s optimizer decides to run it either in a centralized or in a distributed fashion. Although DREAM does not incur any partitioning overhead, it exhibits excessive replication
and costly pre-processing because of the centralized database construction.

$\text{gStoreD}$ [68] is a distributed partitioning-agnostic system. It does not decompose the input query which is sent as-is to all workers. gStoreD starts the query evaluation by computing the partial local matches at each worker. This process depends on a revised version of gStore [44], a single-machine graph-based RDF engine. Then, gStoreD assembles the partial matches to build the cross-partition results. gStoreD allows for two modes of assembly; centralized and distributed. The centralized assembly mode sends the partial results to a centralized site, whereas distributed mode assembles them in multiples sites in parallel.

2.3.2.2 Sophisticated Partitioning

$\text{H-RDF-3X}$ [20] uses METIS [87], a balanced vertex partitioning method, to efficiently assign each graph node to a single partition. Then, H-RDF-3X enforces the so-called $k$-hop guarantee where, for any vertex $v$ assigned to partition $p$, all vertices up to $k$-hops away and the corresponding edges are replicated in $p$. This way any query within radius $k$ can be executed without communication. For example, partitioning the graph in Figure 2.1 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-3X store; duplicate results are expected due to replication. For example, query $Q = \langle ?\text{stud}, \text{advisor}, \text{Bill} \rangle$ returns duplicate $\langle \text{Lisa}, \text{advisor}, \text{Bill} \rangle$ and $\langle \text{Fred}, \text{advisor}, \text{Bill} \rangle$; one from each partition. To solve this problem, H-RDF-3X introduces the notion of triple ownership. For each vertex $v$ assigned to partition $p$, H-RDF-3X stores a new triple $\langle v, \text{is\_owned}, \text{yes} \rangle$ at partition $p$. During query evaluation an extra join is required for filtering out duplicate results. Queries with radius larger than $k$ are executed using expensive MapReduce joins. $k$ must be small (e.g., $k \leq 2$ in [20]) because replication increases exponentially with $k$. 
Table 2.2: 1-hop undirected guarantee partitioning of the RDF in Figure 2.1. subject(s) and object(s) highlighted in blue are owned by W1; the rest belong to W2. Replicated triples are highlighted in yellow.

EAGRE [79] transforms the RDF data into an entity graph by grouping triples based on the subject where each subject is called an entity. Then, it groups entities with similar properties into an entity class. EAGRE generates a compressed entity graph containing only the entity classes and their relationships which is then partitioned using METIS. At each machine, entities belonging to the same class are treated as high dimensional data indexed by a Space Filling Curve. This maintains an order preserving layout of the data which fits well range and order by queries. EAGRE aims at minimizing the I/O costs by a distributed scheduling approach that reduces the total number of data blocks to read for query evaluation. Similar to H-RDF-3X, EAGRE also suffers from the overhead of MapReduce joins for queries that cannot be evaluated locally.

SHAPE [24] uses semantic hash partitioning to group vertices based on URI hierarchy for the sake of increasing data locality. It identifies groups of triples anchored at the same subject or object and tries to place these grouped triples in the same partition. Then, SHAPE applies its semantic hashing technique in two phases: (i) baseline hash partitioning and (ii) $k$-hop expansion which adds to each partition all triples whose shortest distance to any anchor of the partition is at most $k$. Figure 2.6 shows an example for how to expand a baseline partition in the forward direction with $k = 1$. Each resulting partition is managed by a standalone RDF-3X store.
Figure 2.6: An example of semantic hash partitioning with forward direction and $k = 1$.

Figure 2.7: A summary graph for the RDF in Figure 2.1.

Similar to H-RDF-3X, SHAPE suffers from the high overhead of MapReduce 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.

**TriAD-SG** [25] is a variation of TriAD that uses METIS for data partitioning. Edges that cross partitions are replicated, resulting in $1-hop$ guarantee. It defines a summary graph which includes a vertex for each partition; edges connect vertices that share cross-partition edges. Figure 2.7 shows the summary graph of the data in Figure 2.1. Queries in TriAD-SG are evaluated against the summary graph first, in order to prune partitions that do not contribute to query results. Then, they are 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.

**S2RDF** [31] is a SPARQL engine built on top of Spark [88]. It proposes a relational
partitioning technique for RDF data called Extended Vertical partitioning (ExtVP). ExtVP extends the vertical partitioning approach used by HadoopRDF to minimize the size of input data during query evaluation. ExtVP uses semi-join reduction [89] to minimize data skewness and eliminate dangling triples that do not contribute to any join. For every two vertical partitions (see Figure 2.3), ExtVP pre-computes join reductions. The results are materialized as tables in HDFS. Specifically, for two partitions $P_1$ and $P_2$, S2RDF pre-computes: (i) subject-subject: $P_1 \bowtie_{s=s} P_2$, $P_2 \bowtie_{s=s} P_1$, (ii) subject-object: $P_1 \bowtie_{s=o} P_2$, $P_2 \bowtie_{s=o} P_1$, and (iii) object-subject: $P_1 \bowtie_{o=s} P_2$, $P_2 \bowtie_{o=s} P_1$. The objective of this reduction is to use the semi-join reduced tables for joins instead of the base table since the reduced tables are much smaller, i.e., $T_1 \bowtie_{A=B} T_2 = (T_1 \bowtie_{A=B} T_2) \bowtie (T_1 \bowtie_{A=B} T_2)$. S2RDF does not run on Spark directly; it translates SPARQL queries into SQL jobs which are then executed on top of Spark SQL [90]. S2RDF follows a similar approach to Sempala [82] and PigSPARQL [80]. Sempala is a distributed RDF engine that translates SPARQL into SQL which runs on top of Apache Impala [91]. Similarly, PigSPARQL translates SPARQL queries into Pig Latin [92] scripts on Apache Pig.

### 2.3.2.3 Workload-Aware Partitioning

Partout [28] is a workload-aware distributed RDF engine. It relies on a given query workload to extract representative triple patterns and uses them to partition the data into fragments. Partout has two objectives: (i) collocate fragments that are used together in queries; and (ii) achieve load balancing among workers. Partout defines a load score for each fragment and sorts fragments in descending order. For each fragment, it calculates a benefit score for allocating it to each machine. The benefit score takes into account both the machine utilization as well as the fragment locality. Each worker runs RDF-3X on its assigned fragments. Partout uses global statistics to generate an initial query plan, which is then refined by a cost model that
considers data fragments and their locations. The final query plan is executed in parallel by all machines, where each machine sends the results to other hosts in the pipeline.

**WARP** [27] uses a representative query workload to replicate frequently accessed data by extending the $n$-hop guarantee method [20]. Given a user query, WARP determines its center node and radius. If the query is within the $n$-hop guarantee, WARP sends the query to all machines, which evaluate the query in parallel. Otherwise, the query is decomposed into subqueries for which a distributed query evaluation plan is created. Subqueries are evaluated in parallel by all machines and the results are sent to the master which combines them using merge join.

**AdPart** [30] extends AdPart-NA with adaptive workload-awareness, to cope with the dynamism of RDF workloads. It monitors the query workload and incrementally redistributes parts of the data that are frequently accessed by hot patterns. By maintaining these patterns, many future queries are evaluated without communication. The adaptivity of AdPart complements its good 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) are processed in parallel.

### 2.4 Federated Query Processing

Linked Data refers to a set of best practices used to publish and connect data on the web. RDF is a key ingredient of Linked Data and provides a generic data model for describing and linking entities together. Linked Open Data (LOD) forms a virtually integrated distributed database where documents are stored at autonomous sites, some of which expose their data to users through a SPARQL endpoint. Federated query processing engines receive federated queries which have to be evaluated by
querying several endpoints and linking their partial results together.

Given a query $Q$ and a list of SPARQL endpoints, federated engines starts by detecting what are the relevant endpoints to the query. This process is called source selection which is used by the federated engine to decompose the query into multiple subqueries. Each subquery is sent to the relevant endpoints and its results are sent back to the federated engine. Finally, the federated engine joins the individual subqueries’ results together to formulate the final query answer. Two approaches are used to perform the source selection phase: index-based [40, 93, 42] and index-free [39] approaches. Index-based methods preprocess each endpoint to collect information and statistics about the actual data hosted by the endpoint. On the other hand, index-free methods do not assume any knowledge about the data distribution.

2.4.1 Index-Based Federated Engines

DARQ [93] introduces the usage of service descriptions which are used for source selection and query decomposition. Service descriptions provide declarative descriptions about the data available from each endpoint in the form of capabilities. Capabilities of a data source list for each predicate the constraints on the subjects/objects found in this data source. Consequently, DARQ is only able to answer queries with bound predicates. Service descriptions also contain statistical information like average selectivity estimates of subject, predicate and object combinations. Based on the service description, DARQ decomposes the input query into multiple subqueries where each subquery is one or more triple patterns. If the triple pattern is relevant to one data source, it is sent to this data source only. Otherwise, the subquery is sent to all sources. Then, DARQ utilizes the statistics available in the service description to estimate the cardinality of subqueries and devise an effective query execution plan.

SPLENDID [40] uses Vocabulary of Interlinked Datasets (VOID) descriptions which incorporates statistics and information about the endpoints’ data. It stores
information about predicates and types available at each endpoint in the form of an inverted index. Moreover, it contains other useful statistics like triple count, number of distinct predicates and so on. SPLENDID utilizes VOID descriptions for source selection and cardinality estimation. For triple patterns that not covered by VOID descriptions, it uses SPARQL ASK requests. Using the information acquired from the source selection phase, SPLENDID decompose the query into subqueries. Triple patterns are sent independently to all relevant sources. They can be grouped together if they are exclusively relevant to a single source (exclusive group [39]).

ANAPSID [42] is a query engine that adapts to data availability and runtime conditions of SPARQL endpoints to hide delays from users. Its SPARQL operator can detect if a source is blocked and provide results as quickly as data arrives from the sources. ANAPSID uses an index which stores the ontology used to describe the data of each endpoint. Furthermore, it stores endpoints execution timeouts which indicates their capabilities. A query is decomposed into multiple simple subqueries based on the source selection information and the endpoints capabilities. A subquery is sent to an endpoint if its estimated execution time is less than the endpoint timeouts.

2.4.2 Index-Free Federated Engines

FedX [39] is an index-free federated engine which does not assume any prior knowledge about the endpoints and their data. Rather, it utilizes SPARQL ASK queries to find the relevant endpoints for the query triple patterns. It also utilizes its cache to store the results of ASK requests for future usage. Therefore, FedX’s startup cost and the cost of adding a new endpoint is negligible. Based on the ASK request results, FedX decompose the input query into multiple subqueries. If a group of triple patterns can be answered by only one endpoint, it can be processed by this endpoint as one unit, in the form of an exclusive group.

The indices (summaries) collected by index-based methods makes query evaluation
Table 2.3: Datasets; $M$: millions. $\#S$, $\#P$, $\#O$ denote number of distinct subjects, predicates and objects.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Triples ($M$)</th>
<th>$#S$ ($M$)</th>
<th>$#O$ ($M$)</th>
<th>$#S \cap O$ ($M$)</th>
<th>$#P$</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WatDiv</td>
<td>109.23</td>
<td>5.21</td>
<td>17.93</td>
<td>4.72</td>
<td>86</td>
<td>15</td>
</tr>
<tr>
<td>YAGO2</td>
<td>284.30</td>
<td>10.12</td>
<td>52.34</td>
<td>1.77</td>
<td>98</td>
<td>42</td>
</tr>
<tr>
<td>WatDiv-1B</td>
<td>1,092.16</td>
<td>52.12</td>
<td>179.09</td>
<td>46.95</td>
<td>86</td>
<td>149</td>
</tr>
<tr>
<td>LUBM-10240</td>
<td>1,366.71</td>
<td>222.21</td>
<td>165.29</td>
<td>51.00</td>
<td>18</td>
<td>224</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>4,287.59</td>
<td>552.08</td>
<td>1,075.58</td>
<td>491.73</td>
<td>1,714</td>
<td>596</td>
</tr>
</tbody>
</table>

more efficient compared to index-free methods. However, there are a set of limitations/challenges that face these systems: (i) indices have to be constantly updated to ensure result correctness. (ii) The preprocessing (startup) cost increases significantly as the federated database contains larger number of endpoints and large data sizes. (iii) The cost of an endpoint joining the federation is higher than index-free methods. In the later case, adding a new endpoint to the federation is just a matter of knowing the endpoint URL. In contrast, index-based methods have to create summaries for the new endpoint and update its indices.

2.5 Experimental Evaluation

In this section, we evaluate the performance of 12 representative distributed RDF systems using multiple real and synthetic datasets. We also build on a recent experimental study for federated RDF engines [54] by conducting a scalability experiment for the state-of-the-art federated engine. All datasets, workloads, and the detailed results of our experiments are available online [94].

2.5.1 System Setup

Datasets: We used real and synthetic datasets of variable sizes, summarized in Table 2.3. We used the LUBM [69] synthetic data generator to create a dataset of 10,240 universities consisting of 1.36 billion triples. LUBM and its template queries are widely used for testing most distributed RDF engines [21, 23, 24, 25]. We also used WatDiv [95] 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 synthetic dataset: WatDiv with 109 million and WatDiv-1B with 1 billion triples. We also use two real datasets: YAGO2 [96] and Bio2RDF [70]. YAGO2 is derived from Wikipedia, WordNet and GeoNames containing 284 million triples. Bio2RDF provides linked data for life sciences and contains around 4.3 billion triples connecting 24 different biological datasets.

**Hardware Setup:** All systems are deployed on a 12 machine cluster connected by a 10Gbps Ethernet switch. Each machine has a 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs (12 cores each). The cluster runs a 64-bit Linux.

**Hadoop and Spark configuration:** We use Hadoop 1.2.1 and Spark 1.6.2 for MapReduce and Spark-based systems and configure them to best utilize the available resources. For Hadoop, we use 12 mappers per worker (a total of 144 mappers), while we configure Spark with 12 cores per worker to achieve a total of 144 Spark cores. Note that we do not use all 24 cores for Hadoop and Spark workers to allow for other background processes, such as HDFS threads and Spark communicator threads.

**Compared Systems:** We evaluate the performance of 5 systems from the MapReduce and Graph-based category and 7 from Specialized RDF systems. The 12 evaluated systems are: (i) AdPart [30], the current state-of-the-art distributed RDF engine. (ii) TriAD [25], a recent efficient in-memory RDF system that uses lightweight hash partitioning. We also consider TriAD-SG, which uses graph summaries for join-ahead pruning. (iii) Three Hadoop-based systems which use lightweight partitioning: CliqueSquare [29], H2RDF+ [23] and SHARD [22]. (iv) SHAPE [24], a semantic hash partitioning approach for RDF data. (v) H-RDF-3X [20], a system that uses METIS for graph partitioning and applies the $k - hop$ guarantee scheme. We configure SHAPE with full level semantic hash partitioning and enable the type optimization. For H-RDF-3X, we enable the type and high degree vertices optimizations. (vi) S2RDF [31], a SQL-based RDF engine on top of Spark. (vii) S2X [67], an
Table 2.4: Partitioning configurations and replication ratio.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>(a) Partitioning Config.</th>
<th>(b) Initial Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM-10240</td>
<td>H-RDF-3X SHAPE</td>
<td>H-RDF-3X SHAPE DREAM</td>
</tr>
<tr>
<td>WatDiv</td>
<td>2 undirected 2 forward</td>
<td>19.5% 42.9% 1200%</td>
</tr>
<tr>
<td>YAGO2</td>
<td>3 undirected 3 undirected</td>
<td>1090% 0% 1200%</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>2 undirected 2 forward</td>
<td>73.7% 0% 1200%</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

RDF engine on top of GraphX. (viii) DREAM [32] which distributes the query execution among fully-fledged unperturbed data stores. (ix) Urika-GD; a data analytics appliance which provides an RDF triplestore and a SPARQL query engine. Urika-GD provides graph-optimized hardware with 2TB of global shared-memory and 64 Threadstorm processors with 128 hardware threads per processor. (x) gStoreD [68] a partitioning agnostic approach for distributed SPARQL query processing. Finally, as baselines, we also compare to two single-machine engines; (xi) RDF-3X [73] and (xii) gStore [44]. These 12 systems were selected based on: (i) the availability of their source codes. (ii) They were successfully run and provided correct results, and (iii) they either provide the best performance in their category or introduce novel approaches for distributed SPARQL query evaluation.

**Configuration:** We use the source codes provided by each system’s authors and enable all optimizations. H-RDF-3X and SHAPE were configured to partition each dataset such that all queries are processable without communication (Table 2.4(a)). To achieve this, we configure H-RDF-3X with undirected guarantee, instead of forward guarantee. For TriAD-SG, we use the same number of partitions reported in [25] for LUBM-10240 and WatDiv. Determining the number of summary graph partitions requires empirical evaluation of some data workload or a representative sample. Generating a representative sample from real data is tricky, whereas empirical evaluation on the original data is costly. Therefore, we do not evaluate TriAD-SG on Bio2RDF and YAGO2.
2.5.2 Startup Overhead

Our first experiment measures the time it takes all systems for preparing the data prior to answering queries. In Table 2.5, systems are only allowed 24 hours to complete preprocessing. For fair comparison, we include the overhead of loading data into HDFS for Hadoop and Spark-based systems; however, we exclude the string-to-id mapping time for all systems.

**Lightweight partitioning:** The preprocessing phase of systems under this category require the least time due to their lightweight partitioning overhead. SHARD and H2RDF+ employ random and range-based partitioning, respectively, while CliqueSquare uses a combination of hash and vertical partitioning. S2X depends on GraphX’s default partitioning strategy, however, its encoding modes may affect GraphX’s partitioning results. The hash-based encoding has very long loading time and cannot load all graphs, such as YAGO2. On the other hand, count-based encoding has faster data loading in GraphX but is slightly slower in query runtime. In our experiments, we only report the best query evaluation results for S2X irrespective of the encoding scheme since their performance do not vary significantly. MapReduce-based systems suffer from the overhead of storing their data on HDFS.

<table>
<thead>
<tr>
<th></th>
<th>LUBM-10240</th>
<th>WatDiv</th>
<th>YAGO2</th>
<th>Bio2RDF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Machine Systems</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gStore</td>
<td>&gt;24h</td>
<td>175</td>
<td>362</td>
<td>&gt;24h</td>
</tr>
<tr>
<td>RDF-3X</td>
<td>392</td>
<td>33</td>
<td>91</td>
<td>&gt;24h</td>
</tr>
<tr>
<td><strong>Distributed Systems</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARD</td>
<td>72</td>
<td>9</td>
<td>17</td>
<td>143</td>
</tr>
<tr>
<td>H2RDF+</td>
<td>152</td>
<td>9</td>
<td>22</td>
<td>387</td>
</tr>
<tr>
<td>CliqueSquare</td>
<td>167</td>
<td>10</td>
<td>19</td>
<td>N/A</td>
</tr>
<tr>
<td>S2X (Count)</td>
<td>48</td>
<td>3</td>
<td>9</td>
<td>158</td>
</tr>
<tr>
<td>S2X (Hash)</td>
<td>114</td>
<td>8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>S2RDF (VP)</td>
<td>84</td>
<td>13</td>
<td>25</td>
<td>&gt;24h</td>
</tr>
<tr>
<td>S2RDF (ExtVP)</td>
<td>204</td>
<td>225</td>
<td>1,144</td>
<td>&gt;24h</td>
</tr>
<tr>
<td>AdPart-NA</td>
<td>14</td>
<td>1.2</td>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td>TriAD</td>
<td>72</td>
<td>4</td>
<td>11</td>
<td>75</td>
</tr>
<tr>
<td>TriAD-SG</td>
<td>737</td>
<td>63</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>H-RDF-3X</td>
<td>939</td>
<td>285</td>
<td>199</td>
<td>&gt;24h</td>
</tr>
<tr>
<td>SHAPE</td>
<td>263</td>
<td>79</td>
<td>251</td>
<td>&gt;24h</td>
</tr>
<tr>
<td>gStoreD</td>
<td>&gt;24h</td>
<td>85</td>
<td>254</td>
<td>&gt;24h</td>
</tr>
<tr>
<td>DREAM</td>
<td>427</td>
<td>36</td>
<td>98</td>
<td>&gt;24h</td>
</tr>
</tbody>
</table>

Table 2.5: Preprocessing time (Minutes).
first before they can start their preprocessing phase. Both TriAD and AdPart-NA use hash-based partitioning. However, TriAD is slower than AdPart-NA because it sorts indices, gathers statistics and partitions its data twice (on subject and object columns).

**Sophisticated partitioning:** The overhead of RDF-3X on LUBM-10240 is large compared to several distributed engines while gStore failed to preprocess it within 24 hours. Both systems preprocessed WatDiv-100 and YAGO2 in a reasonable time due to the smaller data size. However, they are still slower than several distributed systems such as AdPart-NA and H2RDF+. Both single-machine systems failed to preprocess Bio2RDF within 24 hours. As Table 2.5 shows, systems that rely on METIS for partitioning (i.e., H-RDF-3X and TriAD-SG) have significant startup cost since METIS does not scale to large RDF graphs [24]. To apply METIS, we had to remove all triples connected to literals; otherwise, METIS will take several days to partition LUBM-10240 and YAGO2. For LUBM-10240, SHAPE incurs less preprocessing time compared to METIS-based systems. However, SHAPE performs worse for WatDiv and YAGO2 due to data imbalance, causing some of its RDF-3X engines to take more time in data indexing. Due to the uniform URI’s of YAGO2 and WatDiv, SHAPE could not utilize its semantic hash partitioning and placed the entire dataset in a single partition. Finally, both SHAPE and H-RDF-3X did not finish partitioning Bio2RDF and were terminated after 24 hours.

S2RDF has two preprocessing modes: VP and ExtVP. VP mode partitions the RDF graph based on the triples predicate, and stores each predicate in HDFS as a compressed columnar storage format (parquet file). ExtVP mode builds on top of VP. For every two VPs, ExtVP pre-computes its join reductions and materializes the results as new partitions (tables for Spark SQL) in HDFS. Hence, ExtVP incurs significantly higher overhead, an order of magnitude slower, compared to its VP version. The overhead incurred by ExtVP depends on several factors including dataset
size, density and number of predicates. For example, the size of YAGO2 is almost five times less than the size of LUBM-10240, however, ExtVP requires almost an order of magnitude extra time to process YAGO. This behaviour is also noticed with ExtVP on WatDiv in comparison to LUBM-10240. This is mainly due to the sparsity and the lower number of predicates in LUBM-10240 compared to YAGO and WatDiv. In particular, ExtVP generates a total of 179 partitions (30,337 HDFS objects) for LUBM-10240 compared to 2,221 partitions (319,662 HDFS objects) for WatDiv and 8,125 partitions ($\approx$ 2 million HDFS objects) for YAGO2. Due to this high overhead, S2RDF failed to preprocess Bio2RDF dataset within 24 hours.

### 2.5.3 Initial Replication Cost

We only report the initial replication for SHAPE, H-RDF-3x and DREAM (Table 2.4(b)) since other systems do not explicitly apply replication. H-RDF-3X results in only a 19.5% replication for LUBM-10240 and 73.7% for YAGO2. Since LUBM is generally sparse and uniformly structured around high degree vertices, METIS results in a small edge cut between the different partitions, which significantly reduces the cross partition replicated vertices. SHAPE, however, incurs 42.9% replication due to its full level semantic hash partitioning and type optimization. METIS fails to reduce the graph edge-cuts for WatDiv because of its dense nature. As a result, H-RDF-3X replicated the whole partitions blindly using $k$-hop guarantee which results in 1090% replication. Since the URI’s of WatDiv and YAGO2 are uniform, the semantic hash partitioning of SHAPE places them in a single partition. Therefore, it incurs no replication and performs as slow as a single-machine RDF-3X store.

DREAM indexes the entire database on a single machine which are then replicated to other workers. DREAM’s preprocessing overhead is reasonable for small datasets (Table 2.5), however; it does not scale for larger graphs where it requires more than a day to build the Bio2RDF database. For our 12 machines cluster, the replication
ratio of DREAM is 1200% for all datasets.

2.5.4 Query Performance

In this section, the performance of each query is averaged over five runs for any system.

<table>
<thead>
<tr>
<th>LUBM-10240</th>
<th>Complex Queries</th>
<th>Simple Queries</th>
<th>Geo- Query/h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1</td>
<td>L2</td>
<td>L3</td>
</tr>
<tr>
<td>SM</td>
<td>RDF-3X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDF-3X</td>
<td>1,812,250</td>
<td>101,750</td>
<td>1,898,500</td>
</tr>
<tr>
<td>H2RDF+</td>
<td>265,430</td>
<td>71,720</td>
<td>264,780</td>
</tr>
<tr>
<td>CliqueSquare</td>
<td>125,020</td>
<td>71,010</td>
<td>80,010</td>
</tr>
<tr>
<td>S2RDF-VP</td>
<td>217,537</td>
<td>28,917</td>
<td>145,761</td>
</tr>
<tr>
<td>S2RDF-ExtVP</td>
<td>46,552</td>
<td>35,802</td>
<td>21,533</td>
</tr>
<tr>
<td>AdPart-NA</td>
<td>2,743</td>
<td>120</td>
<td>320</td>
</tr>
<tr>
<td>TriAD</td>
<td>6,023</td>
<td>1,519</td>
<td>2,387</td>
</tr>
<tr>
<td>TriAD-5G</td>
<td>5,392</td>
<td>1,774</td>
<td>4,636</td>
</tr>
<tr>
<td>Urika-GD</td>
<td>5,835</td>
<td>2,396</td>
<td>1,871</td>
</tr>
<tr>
<td>H-RDF-3X</td>
<td>7,004</td>
<td>2,640</td>
<td>7,957</td>
</tr>
<tr>
<td>H-RDF-3X (in-memory)</td>
<td>6,841</td>
<td>2,597</td>
<td>7,948</td>
</tr>
<tr>
<td>SHAPE</td>
<td>25,319</td>
<td>4,387</td>
<td>25,360</td>
</tr>
<tr>
<td>DREAM</td>
<td>13,031,410</td>
<td>98,263</td>
<td>2,358</td>
</tr>
<tr>
<td>DREAM (cached)</td>
<td>1,843,376</td>
<td>98,263</td>
<td>&lt;1</td>
</tr>
</tbody>
</table>

Table 2.6: Runtime for LUBM-10240 queries (ms). SM: Single Machine, MR: MapReduce, and SS: Specialized systems. S2X failed to execute all queries; gStore and gStoreD could not preprocess the data within 24 hours.

2.5.4.1 LUBM dataset

In this experiment (see Table 2.6), we use the LUBM-10240 dataset and L1-L7 [77] queries. We classify those queries, based on their structure and selectivity into simple and complex. L4 and L5 are simple selective star queries. L6 is also a simple query because it is highly selective. L1, L3 and L7 are complex queries with large intermediate results but very few final results. Finally, L2 is a simple yet non-selective star query that generates large final results.

RDF-3X (single-machine) performs well for simple queries while it is significantly slower for complex queries. SHARD, H2RDF+ and CliqueSquare suffer from the expensive overhead of MapReduce joins; hence, their performance is significantly worse than other systems. For selective simple queries, H2RDF+ avoids the overhead of MapReduce-based joins by solving these queries in a centralized manner which is an order of magnitude faster. The flat plans of CliqueSquare significantly reduce the
join overhead for complex queries and achieve up to an order of magnitude better performance. S2X fails to run all queries as it generates a lot of intermediate results at the vertex level. Compared to MapReduce systems, S2RDF-ExtVP shows significant performance improvement due to its in-memory caching technique as well as the materialized join reduction tables. Note that S2RDF requires loading multiple partitions into memory for each query before execution. For example, it loads 6 partitions (1,200 HDFS object) to process L1 and L3.

SHAPE and H-RDF-3X perform better than MapReduce-based systems because they do not require communication. H-RDF-3X performs better than SHAPE as 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 (1.5 sec on our cluster). Without this overhead, both systems would perform well for simple selective queries. For complex queries, these systems still perform reasonably well as they run in parallel without any communication overhead. For example, for query L7 which requires excessive communication, H-RDF-3X and SHAPE perform better than some of the specialized systems, such as TriAD. With low hop guarantee, the preprocessing cost for SHAPE and H-RDF-3X can be reduced at the cost of worse query performance because of the MapReduce joins. Although SHAPE and H-RDF-3X do not incur any communication, they still suffer from two limitations: (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 and (ii) to filter out duplicate results, H-RDF-3X requires an additional join with the ownership triple pattern. For fairness, we also stored H-RDF-3X databases in a memory-mounted partition; still, it did not affect the performance significantly. As a result, we do not consider in-memory H-RDF-3X in further experiments.

In the specialized systems category, AdPart-NA and TriAD perform the best. They are comparable for queries L4 and L5 due to high selectivity. AdPart-NA
exploits the initial hash distribution and solves L4 and L5 without communication. It also solves L2 without communication but slower due to L2’s low selectivity. For this query, AdPart-NA is more than an order of magnitude faster than TriAD and TriAD-SG. This is because AdPart-NA utilizes hash indexes and right deep tree plan to report the results using a single scan followed by hash lookups. TriAD solves L2 by two distributed index scans (one for each base subquery) followed by a merge join. The merge join utilizes binary search to find the location of the join key, which is only efficient for selective queries. TriAD-SG outperforms TriAD and AdPart-NA in L6 as its pruning technique eliminates communication.

In DREAM, statistics are collected on a query-by-query basis and then cached for future queries. This causes huge performance difference between the first run of DREAM and its subsequent runs. The number of machines that can be utilized during query execution is bounded by the number of join vertices; only a maximum of 3 workers for LUBM queries. This explains its huge overhead when processing complex queries, i.e., L1 and L7. For L3, the query planner detects the empty result during statistics collection phase and terminates the query execution early. DREAM executes simple queries on a single machine by using RDF-3X.

2.5.4.2 WatDiv dataset

The WatDiv benchmark defines 20 query templates\(^1\) classified into four categories: linear (L), star (S), snowflake (F) and complex queries (C). Table 2.7 shows the geometric mean of running 20 queries from each category on WatDiv-100M.

RDF-3X and gStore perform well for most queries, where they are faster than several distributed systems including S2X, CliqueSquare, SHAPE and gStoreD. H2RDF+ and CliqueSquare perform worse than most distributed systems due to the overhead of MapReduce. H2RDF+ performs much better than CliqueSquare.

\(^{1}\)http://db.uwaterloo.ca/watdiv/basic-testing.shtml
Table 2.7: Query runtimes (ms) for WatDiv and YAGO2 datasets. SHARD and gStoreD crashed in several queries while DREAM could not finish WatDiv queries within 24 hours.

<table>
<thead>
<tr>
<th>SM</th>
<th>gStore</th>
<th>RDF-3X</th>
<th>SHARD</th>
<th>H2RDF+</th>
<th>CliqueSquare</th>
<th>S2X</th>
<th>S2RDF-VP</th>
<th>S2RDF-ExtVP</th>
<th>AdPart-NA</th>
<th>TriAD</th>
<th>Urika-GD</th>
<th>H-RDF-3X</th>
<th>SHAPE</th>
<th>gStoreD</th>
<th>DREAM (cached)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-L5</td>
<td>1.51</td>
<td>620</td>
<td>N/A</td>
<td>5.441</td>
<td>29.216</td>
<td>292.964</td>
<td>2,180</td>
<td>1,880</td>
<td>9</td>
<td>4</td>
<td>1,264</td>
<td>1,672</td>
<td>1,870</td>
<td>732</td>
<td>N/A</td>
</tr>
<tr>
<td>S1-S7</td>
<td>1.76</td>
<td>355</td>
<td>N/A</td>
<td>8.679</td>
<td>23.908</td>
<td>241.954</td>
<td>2,764</td>
<td>2,269</td>
<td>15</td>
<td>15</td>
<td>1,330</td>
<td>1,693</td>
<td>1,824</td>
<td>203</td>
<td>N/A</td>
</tr>
<tr>
<td>F1-F5</td>
<td>3.540</td>
<td>1,656</td>
<td>N/A</td>
<td>18.457</td>
<td>40.464</td>
<td>241.954</td>
<td>4,717</td>
<td>3,531</td>
<td>106</td>
<td>45</td>
<td>1,743</td>
<td>1,778</td>
<td>1,846</td>
<td>2,049</td>
<td>N/A</td>
</tr>
<tr>
<td>C1-C3</td>
<td>5.930</td>
<td>1,656</td>
<td>N/A</td>
<td>45.796</td>
<td>55.835</td>
<td>597.393</td>
<td>6,969</td>
<td>5,295</td>
<td>190</td>
<td>170</td>
<td>2,723</td>
<td>1,920</td>
<td>2,273</td>
<td>8,482</td>
<td>N/A</td>
</tr>
</tbody>
</table>

| YAGO2   |        |        |       |        |             |      |          |             |          |      |          |        |       |         |               |
| Y1      | 575    | 358    | N/A   | 12,349 | 73,011      | 1,863,374 | 9,129    | 2,622       | 46       | 15    | 665,514 | 246,081 | 1,568 | 1,530   | 620            |
| Y2      | 11,050 | 3,588  | N/A   | 43,868 | 36,006      | 1,720,424 | 10,161   | 3,032       | 46       | 15    | 1,568   | 246,081 | 1,568 | 1,530   | 620            |
| Y3      | 2,178  | 357    | N/A   | 35,517 | 100,015     | 1,720,424 | 10,161   | 3,032       | 46       | 15    | 1,568   | 246,081 | 1,568 | 1,530   | 620            |
| Y4      | 2,636  | 3,350  | N/A   | 35,517 | 100,015     | 1,720,424 | 10,161   | 3,032       | 46       | 15    | 1,568   | 246,081 | 1,568 | 1,530   | 620            |
| GeoMean |        |        |       |        |             |      |          |             |          |      |          |        |       |         |               |
| Query/h | 698    | 22     | N/A   | 21,430 | 77,755      | 1,817,050 | 9,358    | 3,203       | N/A      | N/A   | N/A      | N/A    | N/A   | N/A     | N/A           |

Even though the flat plans of CliqueSquare reduce the number of MapReduce joins, H2RDF+ uses a more efficient join implementation using traditional RDF indices. Furthermore, H2RDF+ encodes the URIs and literals of RDF data; hence it has lower overhead than CliqueSquare when shuffling intermediate results. S2X performs much worse due to the significant network overhead of shuffling vertices with high memory footprint. The performance of S2RDF, on the other hand, is close to MapReduce-based systems on WatDiv dataset. Unlike the results in LUBM-10240, S2RDF-ExtVP shows only a slight improvement in comparison to S2RDF-VP.

Even though SHAPE and H-RDF-3X use 3-hop undirected guarantee replication, they do not outperform the single-machine RDF-3X. gStoreD is only faster than its centralized version (gStore) for L and S queries. Its parallelization overhead affected its performance on F and C queries. AdPart-NA and TriAD, on the other hand, provide better performance than all systems. TriAD performs better than AdPart-NA for L and F queries as it requires multiple subject-object joins. In contrast to AdPart, TriAD utilizes subject-object locality to answer these joins without communication. For complex queries with large diameters, AdPart-NA performs better as a result of its locality awareness. DREAM results are omitted because the overhead of its statistics calculation is extremely high (more than 24 hours) due to the high number of triple patterns in WatDiv.
2.5.4.3 YAGO dataset

YAGO2 does not provide benchmark queries. Therefore, we use the test queries (Y1-Y4) defined by AdPart to benchmark the performance of different systems (Table 2.7). Similar to WatDiv dataset, H2RDF+ outperforms CliqueSquare and SHARD due to the utilization of HBase indices and its distributed implementation of merge and sort-merge joins. Moreover, S2X is still the slowest system while S2RDF provides better performance compared to MapReduce-based systems. Similar to the result in LUBM-10240, ExtVP shows better performance compared to VP. gStoreD fails to process all YAGO queries due to its parallelization overhead. AdPart-NA solves most of the joins in Y1 and Y2 without communication, which explains the 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; therefore, it performs worse than TriAD.

2.5.4.4 Bio2RDF dataset

Bio2RDF dataset does not have benchmark queries; therefore, we used the test queries (B1-B5) which are extracted from a real query log. The results of this experiment are shown in Table 2.8. Note that S2X failed to execute all Bio2RDF queries because of the huge amount of data it generates at its initial supersteps. Moreover, gStore, gStoreD, RDF-3X, H-RDF-3X, SHAPE, S2RDF and DREAM could not not finish their preprocessing phase within 24 hours. The fastest systems in this experiment are AdPart-NA and TriAD while H2RDF+ and SHARD performed worse than other systems due to the MapReduce overhead. TriAD outperforms all other systems on queries B1, B2 and B3 since these queries require subject-object or object-object joins which contradicts the initial partitioning of AdPart-NA. B4 is a complex query with 2-hop radius which AdPart-NA could efficiently process while TriAD crashed. B5 is a simple star query with only one triple pattern in which both AdPart-NA and TriAD
2.5.5 Scalability

In this section, we select a system from each category\(^2\) and test its data and strong scalability. We use AdPart-NA as a representative for specialized systems, SHAPE for systems that employ sophisticated partitioning, CliqueSquare for MapReduce-based, S2X for graph-based and S2RDF for sophisticated partitioning systems on Spark.

**Data Scalability.** In this experiment, we show how the performance of different distributed systems change as we increase the dataset size. Using LUBM, we generated 7 datasets ranging from 21 millions triples to 1.3 billion triples; LUBM-160, LUBM-320, LUBM-640, LUBM-1280, LUBM-2560, LUBM-5120 and LUBM-10240. Similarly, we used WatDiv data generator to create 5 datasets with sizes ranging from 50 to 800 million triples. Figures 2.8(a) and 2.8(b) show the throughput (queries per hour) of the selected systems for both LUBM and WatDiv datasets. As expected, the throughput of most of the systems; with the exception of S2X, decrease slowly with the dataset size. AdPart-NA achieves the best throughput followed by SHAPE and S2RDF-ExtVP. Due to the scalability of MapReduce and Spark, CliqueSquare and S2RDF scale well with the data size. Similarly, SHAPE scales well as it does not incur communication during query evaluation. On the other hand, S2X do not scale because it generates a huge amount of intermediate results during BGP matching that excessively grow with larger datasets. Moreover, S2X could not evaluate

\(^2\)More scalability experiments are available online in [94].
LUBM-10240 and WatDiv-800M queries.

**Strong Scalability.** In this experiment, we use a workload of 35 queries to demonstrate the strong scalability of LUBM-10240 from 2 up to 16 machines. Strong Scalability for WatDiv-1B dataset is available in [97]. Figure 2.9 shows the time of each system as the number of machines increases. SHAPE achieves the best speedup as it solves queries without any communication overhead among machines. AdPart-NA incurs higher processing and communication overheads on larger clusters; therefore, it becomes less scalable after 4 machines. Finally, S2RDF-ExtVP does not scale well on larger cluster while CliqueSquare scales reasonably well as we increase the number of machines.

### 2.5.6 Workload Adaptivity

In this section, we evaluate the performance of workload-aware systems using SPARQL workloads on LUBM-10240 and WatDiv-1B. While Partout works for small datasets and workloads, it took a significant amount of time to preprocess the above datasets; it could not finish partitioning LUBM-10240 and WatDiv-1B within a day even for small workload sizes. Therefore, we only test the adaptivity of AdPart
against two representative systems: AdPart-NA and S2RDF. AdPart-NA shows the best performance among specialized systems while S2RDF provides the best performance compared to other MapReduce-based systems.

**LUBM-10240 workload:** Using the 14 LUBM benchmark queries, a workload of 10K unique queries with different constants and structures is generated. Then, we shuffle the queries to generate a random workload for our experiments. This workload covers a wide spectrum of query complexities including simple selective queries, star queries as well as queries with complex structures and low selectivity.

**WatDiv-1B workload:** This workload contains a 5K-query from each query category (i.e., L, S, F and C) in WatDiv, resulting in a total of 20K queries. To simulate a change in the workload, queries of the same WatDiv-1B template are run consecutively. Therefore, after every sequence of 5K query executions, the type of queries changes.

Figure 2.10 shows the cumulative time as the execution progresses for both datasets. Without adaptivity, the cumulative time of both AdPart-NA and S2RDF increases sharply. AdPart-NA finished all LUBM-10240 queries under 1.5 hours while S2RDF could only finish 1460 queries within its 24 hours window (Figure 2.10(a)). Figure 2.10(b) also shows a similar behaviour where AdPart-NA finished all of the 20K
queries of WatDiv-1B under 2 hours while S2RDF finished only 5100 queries. AdPart, on the other hand, adapts to the workload by redistributing frequently accessed data to allow future queries to be executed in parallel and without communication. Therefore, its performance becomes almost constant after 2.5K queries in LUBM-10240 allowing the workload to finish under 10 minutes. A similar behaviour is observed in WatDiv-1B, where the cumulative time becomes almost constant after 10K queries and the workload finishes in less than 15 minutes. In comparison to AdPart-NA, AdPart is an order of magnitude faster in the LUBM-10240 workload and 6 times faster in the WatDiv-1B workload.

Figure 2.10: Cumulative execution time of a workload.
2.5.7 Scalability of Federated Systems

In this section, we examine the performance of FedX [98], a federated query processor that was shown to outperform other similar systems [54], as the number of endpoints increases. FedX starts processing a query by sending ASK requests to all the endpoints to determine the relevant ones. The result of this source selection step is cached for future invocations. FedX then optimizes the join order and identifies exclusive groups. It uses block nested loops join to reduce the number of HTTP requests in the bound join. We use the QFed [99] and LUBM [69] benchmarks with two queries, the Drug query from QFed and Q2 from LUBM. Drug finds all medicines that target asthma and obtains information about them. It uses up to 4 datasets (in 4 endpoints). Q2 finds graduate students who are registered in courses delivered by their advisor. Each endpoint in the LUBM benchmark corresponds to a university. Both queries are shown in Appendix A.2. We run each query multiple times so that FedX can cache the results of source selection.

Figure 2.11 shows the response time and the number of remote requests for different numbers of endpoints, excluding the ASK queries made during source selection. We see that both the response time and the number of requests increase by orders of magnitude as the number of endpoints increases. Processing one triple pattern at a time while binding query variables to values from intermediate results causes a huge
number of remote requests, which limits scalability. Even with the optimizations employed by FedEx, remote requests are still a bottleneck. In the bound join phase, the number of remote requests depends on the size of the intermediate data. Bound joins also reduce parallelism since only one join step is processed at a time.

2.5.8 Experiments Summary

2.5.8.1 Experiences with Evaluated Systems

AdPart-NA and TriAD: provide the lowest query runtime and startup overhead among all 12 distributed systems. However, they have huge memory overhead because they store the whole RDF graph in memory. For example, to query LUBM-10240, AdPart-NA and TriAD require 317GB and 232GB of RAM, respectively. AdPart is a good choice that reduces the end-to-end workload runtime by dynamically adapting its data distribution as the workload changes.

DREAM and gStoreD: are both disk-based systems that rely on existing single-machine engines; RDF-3X and gStore, respectively. DREAM replicates the full RDF graph to all workers, and incurs significant overhead for building and indexing the entire database on a single machine. We could not run gStoreD on LUBM-10240 and Bio2RDF as it requires significant time for data preprocessing.

SHAPE and H-RDF-3X: significantly outperform MapReduce-based systems at the cost of higher preprocessing overheads. Semantic hash partitioning in SHAPE has lower startup cost compared to METIS in H-RDF-3X. However, SHAPE only works for LUBM-10240 as it locates YAGO2 and WatDiv datasets in a single machine which results in a poor query performance.

SHARD, H2RDF+ and CliqueSquare: SHARD runs on all dataset but with slower query runtimes as it does not use sophisticated RDF indices or complex query optimizations. On the other hand, H2RDF+ uses specialized RDF indices and better
distributed join algorithms which allow it to significantly outperform SHARD. The optimizer of CliqueSquare uses its flat plans optimization to significantly reduce the join overhead of complex queries.

**S2X and S2RDF:** S2X has lightweight startup cost and huge runtime overhead caused by the size of intermediate results during BGP matching. As a result, it fails to run queries on LUBM-10240 and Bio2RDF. On the other hand, S2RDF has higher preprocessing overhead but shows significant performance improvement compared to MapReduce-based systems. This efficiency is mainly driven by its in-memory caching techniques and the materialization of different join reduction tables prior to query execution.

### 2.5.8.2 Discussion

**Index Size.** We noticed that the dataset size does not reflect the systems’ actual memory/desk usage. Each system has its unique storage requirements which may exceed the original size of the input data. Typically, distributed systems load the RDF graph through complex data structures and sophisticated indexing; e.g. traditional SPO permutations. For example, to index WatDiv-1B (149GB), AdPart and TriAD require 208GB and 165GB, respectively. Details about the index sizes of various systems is available in [97].

**Influence of Query Complexity.** Simple queries; e.g. LUBM (L4, L5 and L6) and WatDiv (star-shaped), require few joins or have a very small number of results. Therefore, several systems within the same category performs comparably where even a centralized system may perform better than several distributed systems. As queries require larger number of joins; e.g. LUBM (L1, L3 and L7) and WatDiv (snowflake and complex), or generate larger amounts of intermediate and final results; e.g. LUBM (L2 and L7), the advantage of distributed systems becomes visible. We
noticed that the optimizations of each system favors certain query types. For example, CliqueSquare works well with complex queries where its flat plans significantly reduce the overhead of distributed joins. Likewise, AdPart works well when queries do not contain object-object joins that contradict its initial partitioning scheme.

**Effect of Programming Language.** We noticed that the programming language used in the evaluated systems does not significantly influence its performance. Rather, how each system parallelizes its tasks, reduces the communication cost and employs less synchronization barriers is more important. Specifically, all specialized systems are built using C++; however, their divergent evaluation techniques result in distinct query performance. For example, while TriAD and AdPart-NA are both specialized in-memory systems, their performance is different because of their employed optimization techniques; e.g. intermediate result pinning vs. sharding and right-deep vs. bushy tree planning. The same applies to Java-based MapReduce systems.
2.6 Conclusion

In this chapter, we provide an experimental evaluation of state-of-the-art distributed RDF systems. First, we classify and present a brief overview about systems in each category. Then using large-scale real and synthetic datasets, we extensively evaluated existing systems, through a wide range of SPARQL queries, considering different performance factors including: startup overhead, incurred replication, query performance and scalability.

We highlight the following limitations and possible future research directions for optimizing existing distributed RDF systems: (i) there is an emerging type of applications [11, 12, 13, 14] that require the combination of SPARQL subgraph pattern matching and generic graph algorithms. However, there is a gap between generic graph systems and existing RDF systems which leaves these applications without a direct support. (ii) By relying only on schema information to decompose queries, federated engines cause unnecessary data retrieval and communication, leading to poor scalability and response time. (iii) All existing systems focus exclusively on supporting BGP SPARQL queries. Hence, they do not support evaluating other widely used generic operators such as FILTER, LIMIT and OPTIONAL. (iv) Several systems are highly optimized for specific datasets, e.g., LUBM benchmark, and do not perform well for other real datasets. (v) Most distributed systems do not support updates, adding new triples would require several systems to rebuild their indices, which is very expensive for large-scale graphs. (vi) No partitioning algorithm suits all RDF graphs and all query types. The complexity of sophisticated partitioning schemes does not allow distributed RDF systems to process very large graphs in a timely manner. At the same time, they do not always guarantee better performance compared to lightweight partitioning. We believe that focusing on the problem of matching partitioning strategies to RDF graphs, instead of using a single partitioning strategy for all, is a promising research direction.
Chapter 3

Combining Vertex-centric Graph Processing with SPARQL for Large-scale RDF Data Analytics

3.1 Introduction

The emerging trend for RDF analytics [11, 12, 13, 14] goes beyond simple pattern matching, and requires complex pipelines that combine a variety of graph algorithms with SPARQL. For example, Qu et al. [11] mine biological data to identify new targets for existing drugs. First, they run a set of SPARQL queries against biological databases. On the results they execute graph centrality algorithms to identify key biological entities. Rietveld et al. [12] focus on advanced graph sampling. First, they compute the degree centrality and PageRank of the input graph using a generic graph engine, and generate an RDF dataset enriched by centrality and PageRank information for each vertex. Then, they run SPARQL queries against the intermediate RDF graph.

The state-of-the-art centralized [43, 34, 44] and distributed [25, 45, 24, 23, 21] RDF data management systems fail to support generic graph processing as they focus only on SPARQL, which lacks procedural capabilities. Expressing graph algorithms using SPARQL results in verbose and complex queries that are hard to formulate and expensive to evaluate [46, 47, 48]. On the other hand, vertex-centric graph processing systems, such as Pregel [49], PowerGraph [50] and GRACE [36], support graph analytics, but lack built-in SPARQL engines, requiring users to hard-code and manually optimize any pattern matching query.
To address the above problems we propose Spartex, a distributed framework for rich RDF data analytics. Spartex allows users to write queries that combine declarative SPARQL with procedural code for generic graph processing. The advantages of Spartex include: (i) increased productivity: Our framework simplifies the implementation of complex RDF analysis pipelines, by allowing seamless integration of SPARQL and user defined procedures within the same programming environment, and effortless information exchange between them. (ii) Speed: By eliminating the need to move data among independent processing systems, complex analytical tasks are executed at least an order of magnitude faster in our framework. Even for pure SPARQL queries, Spartex is as fast as dedicated RDF engines, due to its sophisticated query optimizer. (iii) Versatility: Spartex can be implemented on a variety of vertex-centric graph engines, including Graphlab, Giraph, Pregel, Spark GraphX, and others. (iv) Scalability: By taking advantage of the underlying graph engine, Spartex can scale to very large graphs and many compute nodes. (v) Community support: There is a thriving community of developers for graph analysis algorithms (e.g., clustering, centrality, mining, machine learning, etc) using the vertex-centric paradigm; the resulting libraries can be used in Spartex.

We realize Spartex in three steps. First, we define an extension of SPARQL. Then, we implement a generic SPARQL query engine as a vertex-centric program. Finally, we extend the in-memory graph representation of the underlying vertex-centric engine. We next give overviews of each step:

SPARQL Extension: Spartex is inspired by the database community, where the coupling of SQL code with a generic programming language (e.g., Java, C++) is common. Spartex defines a Graph Analytics extension of SPARQL that allows generic user-defined procedures (UDPs) to be intermixed with SPARQL. A UDP can be any program implemented in the vertex-centric model (e.g., PageRank, Shortest-Paths, Centrality). UDps communicate with SPARQL by updating the RDF graph. Spartex
also allows filters that limit the scope of the UDP, where the filter condition is a separate SPARQL query.

**SPARQL Engine:** We implement an efficient SPARQL query engine as a vertex-centric program, allowing UDPs and SPARQL queries to run on top of the same vertex-centric framework. Our SPARQL operator leverages the message-passing nature of the vertex-centric frameworks for join evaluation. Given a SPARQL query $Q$, our SPARQL operator has two stages: (i) a cost-based optimizer picks a trail on $Q$ to minimize the number of messages. (ii) Vertices exchange messages with their neighbors along the selected trail for several rounds, until the query is solved.

**In-memory Data Store:** The underlying vertex-centric frameworks typically store the input as a generic graph in memory. Spartex extends this with a per-vertex data store that is tailored to RDF data. Our data store efficiently filters the neighbors of vertices by specific predicates. The data store also allows updates on the RDF graph. Consequently, Spartex introduces novel ways to implement RDF analytics that were not feasible before: (i) Results of generic graph algorithms are represented as new triples; therefore, original RDF data and vertex computed values can be combined in a SPARQL query. (ii) 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. (iii) The entire analysis is executed within our framework; there is no need to use different systems or materialize and reformat intermediate results. In summary, our contributions are:

- We propose Spartex, an RDF analytics framework that allows SPARQL queries to be combined with generic graph algorithms. We demonstrate various scenarios where our framework simplifies the implementation of sophisticated RDF analytics programs.
• At the semantic level, Spartex extends SPARQL (Section 3.2), allowing generic
UDPs to be intermixed with SPARQL queries. At the systems level, Spartex
implements a complete SPARQL query engine (Section 4.5) with a cost-based
optimizer and an in-memory miniature data store.

• We evaluate Spartex using real and synthetic datasets with billions of triples
on a cluster of 288 CPU cores (Section 3.5). We show that our SPARQL engine
is at least as fast as the state-of-the-art dedicated RDF systems. For complex
analytical tasks that combine generic graph processing with SPARQL, we show
that Spartex is at least 10 times faster that existing alternatives.

3.2 Graph Analytics Extension

Our SPARQL extension for graph analytics enables two important functionalities:
(i) users can write their own algorithms in a procedural language and invoke them
from within SPARQL as user-defined procedures. (ii) Users can mutate the original
RDF graph by adding/deleting triples. In particular, the output of UDPs can be
materialized as new triples in the RDF graph. These triples can then be used in a
SPARQL query or as input to another graph algorithm.

3.2.1 New Spartex Constructs

Below we introduce the new Spartex constructs that enable the invocation of UDPs
and information exchange between UDPs and SPARQL.

3.2.1.1 Calling UDPs

The first construct allows users to call an already implemented user-defined procedure
(UDP), as follows:

\[
\text{CALL proc(list[params]) AS list[predicates]}
\]
The above construct calls UDP proc. list[params] is the set of input parameters, while list[predicates] is the set of vertex predicates that proc will add to the RDF graph; this is effectively the output of the UDP. As an example, the following code computes and stores the PageRank value of each vertex in an RDF dataset:

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

Prefix defines the URI of the stored procedure. The input of PageRank is the maximum number of iterations. For each vertex \( v \), the algorithm adds to the RDF graph a new triple \( \langle v, sptx:pRank, rank \rangle \) where \( sptx:pRank \) is the property name and \( rank \) is the PageRank value of \( v \). The resulting triples can be used later within a SPARQL query or as input to other algorithms (see Section 3.2.2).

### 3.2.1.2 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. We introduce 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 }
```

The Basic Graph Pattern (BGP) used with data filters is a star query around a common vertex\(^1\). 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 UDPs through keyword using. For example, to exclude the triples with predicate rdf:type from the PageRank computation, we define the following filter and use it in PageRank:

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
FILTER_EDGE AS no_type WHERE {
```

\(^1\)More sophisticated filtering can be achieved by combining FILTER and ADD TRIPLE (see Section 3.2.2.2)
Users can specify multiple filters to define the subset of the graph the UDP is applied on. Notice that our data filter construct is different from SPARQL filters. The later are used to filter the results of a SPARQL query, whereas our data filter is coupled with the invocation of a graph algorithm to temporary exclude part of the RDF graph before running the algorithm.

3.2.1.3 Graph Updates

We mentioned how a UDP can update the RDF graph. However, users may want to explicitly insert or delete some triples. This can be done as follows:

\[
\text{ADD TRIPLE} \{ \text{list[triple patterns]} \} \text{ WHERE } \{ \text{BGP} \}
\]

\[
\text{DROP TRIPLE} \{ \text{list[triple patterns]} \} \text{ WHERE } \{ \text{BGP} \}
\]

For example, the following code drops all triples with predicate \textit{sptx:pRank} and object (i.e., rank value) less than a threshold:

\[
\text{DROP TRIPLE} \{ ?x \textit{sptx:pRank} ?\text{rank} \} \text{ WHERE} \{
    ?x \textit{sptx:pRank} ?\text{rank} .
    \text{FILTER(?\text{rank} < \text{threshold})}
\}
\]

3.2.2 Case Studies of RDF Analytics

We present three case studies that demonstrate how our SPARQL extensions simplify the implementation of complex RDF analytics applications.

3.2.2.1 Using Graph Analytics Output in SPARQL

Consider query \(Q_s\) in Figure 3.1; it returns the set of students who take courses taught by their advisors. Assume we want to restrict the query results to only popu-
lar professors and important courses. For the sake of the example, let PageRank and centrality indicate professor popularity and course importance, respectively. PageRank and centrality values do not exist in the original data, so they will be computed by UDPs; the corresponding triples are depicted with dotted lines in the figure. \( Q_s \) can be expressed as:

\[
\text{PREFIX sptx: <http://www.spartex.com/analytics/>}
\]
\[
\text{CALL com.sptx.algo.centrality() AS sptx:centrality}
\]
\[
\text{CALL com.sptx.algo.PageRank(max_iter) AS sptx:pRank}
\]
\[
\text{SELECT ?s WHERE {}
}\]
\[
\quad ?p \text{ teaches } ?c .
\]
\[
\quad ?s \text{ takes } ?c .
\]
\[
\quad ?s \text{ advisor } ?p .
\]
\[
\quad ?p \text{ sptx:pRank } ?rank .
\]
\[
\quad ?c \text{ sptx:centrality } ?cent .
\]
\[
\quad \text{FILTER (?rank > val1 && ?cent > val2)}
\]
\[
}\]

Spartex executes the centrality and PageRank algorithms; their output for each vertex is added as new triples to the RDF graph. Then, Spartex executes the subgraph pattern matching part of the query, expressed in SPARQL; only vertices that satisfy the filter constraints are retrieved.

### 3.2.2.2 Using SPARQL Output in Graph Analytics

The results of SPARQL can be used in subsequent general graph algorithms. Consider again \( Q_s \) and assume we want to find the shortest path between popular professors who teach important courses and every other vertex in the graph. This can be done...
by executing the Single Source Shortest Path (SSSP) algorithm starting from those 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 TRIPLE
{?p sptx:popular "T" .} WHERE {
  ?p teaches ?c .
  ?s takes ?c .
  FILTER (?rank > val1 && ?cent > val2)
}
FILTER_VERTEX AS start WHERE {
}
CALL algo:SSSP() USING start AS sptx:sssp
```

First, we identify the qualifying professors and add in the corresponding vertices \(?p\), a Boolean flag \(T\) in the form of a triple \(\langle ?p, sptx:popular, "T" \rangle\). We create a filter to retrieve those professors who are connected with the Boolean flag, and use the filter in the SSSP procedure call, so the algorithm starts from vertices that match the filter.

### 3.2.2.3 Sampling RDF Graphs

SamplD [12] is a pipeline for sampling RDF graphs. Given an RDF graph, SamplD applies a set of graph operations using Apache Giraph [100] and PIG [101]. It transforms the input into a directed unlabeled graph and computes degree centrality and PageRank in the transformed graph. Each triple is assigned a score and triples with the highest scores are selected as the graph sample. SamplD pipeline requires the movement of the input and intermediate result thought multiple programming platforms. We demonstrate how we implement the same pipeline entirely within Spartex:
The entire pipeline can be reduced to three simple UDP calls. Our approach avoids data movement and increases productivity by requiring from the user to learn only one programming environment.

### 3.3 System Architecture

Our Spartex prototype is built on top of GPS [102], an open-source Pregel clone. However, Spartex supports a variety of distributed vertex-centric bulk synchronous graph processing frameworks, such as Pregel [49], GraphLab [37] and Trinity [38]. In such frameworks, users define a generic compute function to be executed on each vertex independently. Vertices interact with each other through messages. 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, e.g., HDFS.

An overview of Spartex is depicted in Figure 3.2. Spartex follows a master-slave architecture. Users write vertex-centric programs for generic graph processing, compile and add them to the classpath. Spartex treats these programs as user-defined stored procedures. In addition, Spartex provides an efficient vertex-centric SPARQL operator (Section 4.5). 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 we describe each component of Spartex.
3.3.1 Master

The master is not assigned any portion of the input graph; rather it orchestrates the workers’ activity. Users submit their program to the master, which parses it and generates an execution plan for the entire pipeline of the generic algorithms and SPARQL queries. Then the master asks the workers to execute each step of the plan. Below, we define each of the master’s components:

**Query Manager.** It is responsible for parsing, optimizing and executing user programs. The *Query Parser* separates the procedural constructs from the declarative SPARQL patterns. It checks the existence of the called UDPs and the consistency of their parameters. Then, our *SPARQL optimizer* enumerates possible execution plans for the SPARQL part, estimates their costs using global statistics and generates an optimized plan. The *Pipeline Executer* receives a global pipelined execution plan consisting of the consolidated procedural and optimized SPARQL plan. It dictates which vertex-centric program to run for how many iterations or until the program converges. After each step, the executer initializes the next vertex-centric process to be executed, until the plan is completed.

**Statistics Manager.** The master gathers global statistics during the RDF graph
loading phase. First, each worker loads and indexes its assigned vertices and their edges. Next, vertices report statistics about their neighbours and predicates, as well as the correlation between different predicates (refer to Section 3.4.2.2 for details). Finally, each worker synchronizes its collected statistics with the master. The master aggregates all statistics in a global structure.

**UDP Catalogue.** It contains meta-data about user-defined procedures that are registered with Spartex.

### 3.3.2 Worker

Vertex-centric frameworks divide the input graph into partitions. A vertex \( v \) with its outgoing edges is assigned to worker \( W \) based on a hashing scheme \( W(v \mod k) \), where \( k \) is the number of partitions. Spartex slightly modifies the default partitioning scheme such that both the incoming and outgoing edges of each vertex \( v \) are assigned to the same worker; this is equivalent to partitioning on both subject and object vertices.

In Figure 1.1, assume that vertices Ben and Databases are assigned ids of 0 and 1, respectively and we have only two workers \((k = 2)\). This means that vertex Ben along with its edges (type, student), (advisor, Lisa) and (takes, Databases) will be assigned to worker \( W_0 \) \((= 0 \mod 2)\). Similarly, vertex Databases along with all its incoming and outgoing edges will be assigned to worker \( W_1 \) \((=1 \mod 2)\). Spartex works with any disjoint vertex partitioning scheme, i.e., schemes that assign each graph vertex to a single partition with all its incoming and outgoing edges. Furthermore, it can work with any a sophisticated vertex partitioning scheme that produces higher quality partitions and reduces the communication cost; the choice of partitioning scheme is orthogonal to our work.

**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. Spartex models the data in a uniform way while providing different data access methods. Specifically, our framework supports: (i) label-based neighbour access used for SPARQL query evaluation; and (ii) label-oblivious neighbour access used by most generic graph algorithms. 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 Index.** As workers parse their assigned partitions, they create a per-vertex (miniature) data store tailored to RDF data, which consists of the following: (i) Predicate-Object (PO) index: given an edge predicate \( p \), PO index returns a list of all outgoing neighbors (objects). (ii) Predicate-Subject (PS) index: given an edge predicate \( p \), PS index returns a list of all incoming neighbors (subjects). For label-oblivious access, our API allows to access these indices without providing a predicate. In this case, it returns all incoming or outgoing neighbours of the vertex regardless of the predicate label. Notice that edge labels are assigned to the same partition on which their corresponding vertices exist. Therefore, label-based and label-oblivious data access in Spartex is independent of the partitioning method. It is just an abstraction that allows different graph algorithms to access the data as needed.

**Miniature Properties Store.** Newly added triples are added to this store as vertex properties. Since these triples are updated frequently and to avoid imposing overhead on the PS and PO indices, we employ a separate store per vertex. 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 algorithm 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, etc). It is used by the worker to switch between different UDPs at runtime when directed by the master.

3.4 SPARQL Query Engine

In this section, we present our vertex-centric SPARQL operator. Consider query $Q_s$ defined by the solid lines in query $Q_s$ (Figure 3.1). $Q_s$ consists of 3 triple patterns: $q_1 : (\text{?p, teaches, ?c})$, $q_2 : (\text{?s, advisor, ?p})$ and $q_3 : (\text{?s, takes, ?c})$. In the relational model, $Q_s$ is answered\(^2\) by scanning the data to find the matches of each triple pattern. Then, the intermediate results are joined to formulate the final answers. However, relational approaches are not suitable for Spartex because data, computation and communication are all vertex-centric. Instead, we employ graph exploration and use inter-vertex message passing for query evaluation.

3.4.1 Query Evaluation

For a query graph $Q$, we select 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 $(v_e, p, v_t, \text{direction})$, 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 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 $Q_s$ that starts from $\text{?p}$ is $(\bar{q}_1, \bar{q}_2, \bar{q}_3)=((\text{?p, teaches, ?c, out}), (\text{?c, takes, ?s, in}), (\text{?s, advisor, ?p, out}))$. Obviously, there are many potential trails that can start from any of the query vertices. Query planning is discussed in Section 3.4.2.

A query is evaluated in $n + 1$ iterations, where $n$ is the number of exploration edges in the trail selected by the query optimizer (see Section 3.4.2). At each iter-

---

\(^2\)In this example, a bushy execution plan is assumed.
Input: ExplorationEdge \( \bar{e} \), MessageList \( ml \), Iteration \( i \)

1. \( eVertex \leftarrow \bar{e}.expVertex; \)
2. \( tVertex \leftarrow \bar{e}.termVertex; eDirection \leftarrow \bar{e}.direction; \)
3. \( ePredicate \leftarrow \bar{e}.predicate; \)
4. \( vertexSubQueries \leftarrow \text{getVertexSubqueries}(eVertex); \)
5. if Matches\( (\text{vertexSubQueries}, eVertex) \) then
   
   - \( \text{neighbors} \leftarrow \text{Empty}; \)
   - if \( eDirection \) is Outgoing then
     
     - \( \text{neighbors} \leftarrow \text{PO}[ePredicate]; \)
   - else
     
     - \( \text{neighbors} \leftarrow \text{PS}[ePredicate]; \)
   - if \( i = 1 \) then
     
     - \( \text{msg} \leftarrow \text{Empty}; \)
     - \( \text{msg}[eVertex] \leftarrow \text{vertexID}; \)
     - \( \text{sendMessageToAll}(\text{msg}, \text{neighbors}); \)
   - else
     
     - if isQueryVertexVisited\( (\bar{e}.termVertex) \) then
       
       - foreach \( \text{msg} \) in \( ml \) do
         
         - if \( \text{msg}[tVertex] \in \text{neighbors} \) then
           
           - \( \text{msg}[eVertex] \leftarrow \text{vertexID}; \)
           - \( \text{sendMessage}(\text{msg}, \text{msg}[tVertex]); \)
         - else
           
           - foreach \( \text{msg} \) in \( ml \) do
             
             - \( \text{msg}[eVertex] \leftarrow \text{vertexID}; \)
             - \( \text{sendMessageToAll}(\text{msg}, \text{neighbors}); \)
         - else
           
           - foreach \( \text{msg} \) in \( ml \) do
             
             - \( \text{msg}[eVertex] \leftarrow \text{vertexID}; \)
             - \( \text{sendMessageToAll}(\text{msg}, \text{neighbors}); \)
     - voteToHalt();

Algorithm 1: ExploreEdge

In every iteration, each vertex executes ExploreEdge (Algorithm 1), whose inputs are the exploration edge \( \bar{e}_i \), the messages received from the previous iteration and the iteration count. As an example, query \( \bar{Q}_s \) is evaluated using the previous trail and the data graph in Figure 1.1 as follows:

Iteration 1 (\( \bar{e}_1 \)): Initially, all vertices are active and each vertex executes Ex-
exploreEdge with $\bar{q}_1$ and empty message list as inputs. Each vertex checks 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). Based on the exploration edge direction, all matching vertices retrieve their neighbours connected by predicate $\bar{q}_1.p$ (lines 7-10). Each vertex creates a message containing its id and sends it to the retrieved neighbours. Finally, all vertices vote to halt; vertices become active if and only if they receive a message in the next iteration. In $Q_s$, the exploration vertex $v_e = ?p$. A matching vertex for $?p$ needs to be a subject and an object for predicates teaches and advisor, respectively. From the data graph of Figure 1.1, Fred and James are matches of $?p$, while Tim is not, 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 predicate teaches (line 7-8). A message is formulated from each matching vertex of $?p$ and is sent to its neighbors connected via predicate teaches (lines 11-14). Figure 3.3 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 vertex checks if it matches the exploration vertex $\bar{q}_2.e_v = ?c$; both do. Each of them uses 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...

**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 Ben’s 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, we keep iteration 4 for the sake of explanation.

**Discussion:** Our exploration approach utilizes 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 [21], which resembles semi-join. Trinity.RDF requires a final centralized join, especially for cyclic queries [23]. In contrast, Spartex messages 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: in the third iteration of our example, Ben discards its message because it can validate that Fred (a visited node) is not in his neighbors list; we call this optimization *pre-join*. Although carrying the historical bindings seems to incur high communication overhead, the maximum number of variables per query is usually small [103]. The second advantage is the search space pruning that happens 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.

3.4.2 Query Planning

Query evaluation performance is highly influenced by the trail followed during execution. In this section, we describe our 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.

3.4.2.1 Query Optimization

The space of possible trails depends on the query graph structure and the fact that each edge has to be visited once. 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, 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: The number of edges that the CPP will duplicate is positively correlated to the number of odd degree vertices. Therefore, we introduce a query
coarsening optimization that minimizes the number of odd degree vertices. 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 their neighbor. For example, query $Q_s^c$ is the coarsened version of $Q_s$; vertices that match $?c$ and $?p$ will validate if they have the rank and centrality predicates, respectively. This applies to all types of star queries, which are coarsened into a single vertex. Hence, star queries are solved in a single iteration without communication.

After coarsening, Algorithm 2 enumerates all possible trails and select the trail with the minimum estimated cost. We use the CPP to calculate how many edges need to be duplicated to make the graph Eulerian. CPP will return zero if the graph is already Eulerian. The maximum trail length is set to the number of edges in the graph plus the number of duplicate edges (line 3). Then, the algorithm searches for an exploration trail from each vertex in the query graph (lines 5-6) using procedure $FindTrail$ (lines 8-21). We employ a branch and bound strategy to prune the search space of possible trails using the plan cost as upper-bound (plan cost estimation is discussed in Section 3.4.2.2). Initially, the plan cost is set to infinity. A branch is pruned in three cases: (i) if a valid exploration plan with less cost (so far) is found; i.e., all edges are visited, the cost bound and the best found plan are updated (lines 12-16). (ii) Since the cost is monotonically increasing, if the current exploration plan cost exceeded the bounded cost, the algorithm terminates (line 10). (iii) The algorithm terminates when the length of the exploration plan exceeds the maximum bounded length (line 11).

### 3.4.2.2 Cost Estimation

The number of exchanged messages during query evaluation depends on the order of exploration. Our optimizer tries to minimize the size of the intermediate results by
exploring the most selective edges first. With the absence of a schema, selectivity estimation in SPARQL is a challenging task [43]. We propose a selectivity estimation method that captures the correlation between pairwise predicates. During data loading, each vertex collects the correlation information and sends it to the master, which aggregates 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, predicate `teaches` in Figure 3.1 appears three times and has \((sc, oc) = (3, 2)\). Similarly, `type` has 11 unique subjects and 5 unique objects.

**Predicate 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 \( v \). For example, \( PPD(\text{advisor}, \text{takes}, \text{out}, \text{out}) \) returns \(\{4, (1, 1)\}\) because there are 4 vertices that have outgoing edges labeled `advisor` and `takes`. On average, each vertex
Algorithm 3: Exploration Trail Cost

has one edge labeled advisor and one edge labeled takes. From the exploration point of view, it means that there are 4 vertices that exist when transitioning between advisor and takes. These vertices will get one message from the previous iteration and send one message out.

Neumann et al. [104] provide better cardinality estimation by counting the characteristic sets of the data. Each set stores a group of correlated predicates along with their statistics. In Spartex, the query execution plan follows a trail over predicates; therefore, our pairwise predicate correlations provide accurate estimates when moving between triple patterns. For star-shaped queries, characteristic sets would provide better estimation as they capture correlation among multiple predicates. However, this has more computation overhead due to the limited granularity (vertex-based) of vertex-centric frameworks which require more communication/synchronization in order to collect the characteristic sets.
Algorithm 3 calculates the cost of the exploration trail \((\bar{q}_1, \bar{q}_2,\ldots, \bar{q}_n)\) using \(n\) iterations. The plan cost is initially zero (line 1). For each exploration edge, Algorithm 3 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, we use \(PPD\) 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 predicates. Each matching vertex sends a number of messages equal to its average degree on 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 sent in this iteration (line 17). (iii) If \(ev\) is already explored and the termination vertex is not explored yet, then we simply forward the messages received through the exploration predicate \(\bar{q}_i.p\). This serves as an upper bound on the number of messages to be sent (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).

### 3.5 Experimental Evaluation

The current version of Spartex is implemented on top of GPS [102]; an open-source Pregel clone. We deploy Spartex and other systems on a cluster of 12 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs with 12 cores each. The
Table 3.1: Dataset statistics in millions (M). Number of triples, unique subjects, objects and predicates.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Triples (M)</th>
<th>#S (M)</th>
<th>#O (M)</th>
<th>#S∩O (M)</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEGG</td>
<td>89.18</td>
<td>8.63</td>
<td>35.68</td>
<td>8.50</td>
<td>140</td>
</tr>
<tr>
<td>LinkedGeoData</td>
<td>274.67</td>
<td>51.92</td>
<td>121.10</td>
<td>41.47</td>
<td>18,272</td>
</tr>
<tr>
<td>YAGO2</td>
<td>295.85</td>
<td>10.12</td>
<td>52.34</td>
<td>1.77</td>
<td>98</td>
</tr>
<tr>
<td>LUBM-10240</td>
<td>1,366.71</td>
<td>222.21</td>
<td>165.29</td>
<td>51.00</td>
<td>18</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>4,287.59</td>
<td>552.08</td>
<td>1,075.58</td>
<td>491.73</td>
<td>1,714</td>
</tr>
</tbody>
</table>

machines run 64-bit 3.2.0-38 Linux Kernel and are connected by a 10Gbps Ethernet switch.

We use real and synthetic datasets containing from 89M to 4.2B triples; Table 3.1 shows the details. KEGG\(^3\) is a real dataset that integrates biological, chemical and genomic information. LinkedGeoData\(^4\) is a spatial knowledge base derived from OpenStreetMap. YAGO2\(^5\) is a real dataset that combines facts from Wikipedia, WordNet and GeoNames. Bio2RDF\(^6\) is another real dataset that interconnects 24 different biological datasets. We also use the synthetic LUBM\(^7\) data generator to generate various synthetic datasets with up to 1.36B triples (corresponds to simulated data from 10,240 universities). We use the same YAGO2 and Bio2RDF queries used by AdPart [45], which span a wide range of structures and complexity classes. For LUBM, we use the queries defined by Atre et al. [77] and used by most RDF engines [25, 45, 24, 23, 21]; additionally, we define two more complex queries (see Appendix A.1.2).

### 3.5.1 SPARQL Query Performance

In this section, we evaluate the SPARQL query performance of Spartex against state-of-the-art RDF engines that are built on top of generic frameworks\(^8\), namely:

---

\(^3\)http://www.genome.jp/kegg/
\(^4\)http://linkedgeoadata.org/
\(^5\)http://yago-knowledge.org/
\(^6\)http://bio2rdf.org/
\(^7\)http://swat.cse.lehigh.edu/projects/lubm/
\(^8\)In Section 3.5.5, we also compare the performance of Spartex against state-of-the-art native distributed RDF engines.
Table 3.2: Query runtime for LUBM-10240 (seconds)

<table>
<thead>
<tr>
<th>LUBM-10240</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
<th>L6</th>
<th>L7</th>
<th>P</th>
<th>D</th>
<th>GMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartex</td>
<td>4.48</td>
<td>6.78</td>
<td>5.45</td>
<td>3.40</td>
<td>3.24</td>
<td>2.38</td>
<td>7.02</td>
<td>7.89</td>
<td>6.21</td>
<td>4.85</td>
</tr>
<tr>
<td>S2RDF</td>
<td>46.55</td>
<td>35.80</td>
<td>21.53</td>
<td>9.22</td>
<td>2.72</td>
<td>4.55</td>
<td>47.34</td>
<td>48.10</td>
<td>55.89</td>
<td>20.04</td>
</tr>
<tr>
<td>CliqueSquare</td>
<td>125.02</td>
<td>71.01</td>
<td>80.01</td>
<td>90.01</td>
<td>24.00</td>
<td>37.01</td>
<td>224.04</td>
<td>161.02</td>
<td>160.02</td>
<td>88.35</td>
</tr>
<tr>
<td>H2RDF+</td>
<td>285.43</td>
<td>71.72</td>
<td>264.78</td>
<td>24.12</td>
<td>4.76</td>
<td>22.91</td>
<td>180.32</td>
<td>1142.10</td>
<td>568.58</td>
<td>105.86</td>
</tr>
<tr>
<td>SHARD</td>
<td>413.72</td>
<td>187.31</td>
<td>N/A</td>
<td>358.20</td>
<td>116.62</td>
<td>209.80</td>
<td>469.34</td>
<td>596.08</td>
<td>544.94</td>
<td>317.60</td>
</tr>
</tbody>
</table>

S2RDF [31], CliqueSquare [29], H2RDF+ [23] and SHARD [105]. We also tried to compare against HadoopRDF [26] but it crashed in several datasets and queries.

**LUBM Dataset:** LUBM queries can be classified into *simple* (L4, L5 and L6), which are very selective and touch the same small number of triples regardless of the dataset size; and *complex* (L1, L2, L3, L7, P and D), which consist of non-selective joins and generate a lot of intermediate results. Table 3.2 shows the result for the LUBM-10240 dataset. If a system fails to execute a query within 1 hour or crashes, we mark it N/A. H2RDF+ performs better than SHARD in all queries due to the utilization of HBase indices and its distributed implementation of sort-merge joins. The flat plans of CliqueSquare reduce the overhead of joins for complex queries. Therefore, it outperforms both SHARD and H2RDF+ in the non-selective complex queries: L1, P and D. However, for selective simple queries, H2RDF+ is faster because these queries are solved in a centralized manner; hence avoiding the overhead of MapReduce based joins. Compared to MapReduce systems, S2RDF shows a significant performance improvement due to its in-memory caching technique as well as the materialized join reduction tables. However, S2RDF requires loading multiple partitions into memory for processing each single query. Spartex utilizes the efficient inter-vertex communication of vertex-centric frameworks. 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 results. In contrast, S2RDF, H2RDF+, CliqueSquare and SHARD require multiple distributed joins. As a result, the geometric mean of Spartex is up to *two orders of magnitude* better than the competitors.
Table 3.3: Query runtimes for YAGO2 (sec)

<table>
<thead>
<tr>
<th>YAGO2</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
<th>Y4</th>
<th>GMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartex</td>
<td>2.8</td>
<td>3.5</td>
<td>2.0</td>
<td>2.7</td>
<td>2.7</td>
</tr>
<tr>
<td>S2RDF</td>
<td>2.822</td>
<td>3.032</td>
<td>3.393</td>
<td>3.628</td>
<td>3.203</td>
</tr>
<tr>
<td>CliqueSquare</td>
<td>139.0</td>
<td>73.0</td>
<td>36.0</td>
<td>100.0</td>
<td>77.7</td>
</tr>
<tr>
<td>H2RDF+</td>
<td>19.9</td>
<td>12.3</td>
<td>43.9</td>
<td>35.5</td>
<td>21.4</td>
</tr>
<tr>
<td>SHARD</td>
<td>238.9</td>
<td>238.9</td>
<td>N/A</td>
<td>N/A</td>
<td>238.9</td>
</tr>
</tbody>
</table>

Table 3.4: Query runtimes for Bio2RDF (sec)

<table>
<thead>
<tr>
<th>Bio2RDF</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>GMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartex</td>
<td>2.0</td>
<td>3.2</td>
<td>3.4</td>
<td>7.9</td>
<td>2.3</td>
<td>3.3</td>
</tr>
<tr>
<td>H2RDF+</td>
<td>5.6</td>
<td>12.7</td>
<td>322.3</td>
<td>7.9</td>
<td>4.3</td>
<td>15.0</td>
</tr>
<tr>
<td>SHARD</td>
<td>239.3</td>
<td>309.4</td>
<td>512.8</td>
<td>787.1</td>
<td>112.3</td>
<td>320.0</td>
</tr>
</tbody>
</table>

**YAGO2 Dataset:** Table 3.3 shows the results for the YAGO2 dataset. Y1 and Y2 are selective queries that result in small number of results. Y3 and Y4 are more complex and consist of non-selective object-object joins. H2RDF+ again outperforms CliqueSquare and SHARD. The flat plans do not improve the performance of CliqueSquare compared to H2RDF+. The reason is that, while the flat plans reduce the number of MapReduce-based joins, H2RDF+ implements a more efficient 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. On the other hand, S2RDF performs significantly better compared to CliqueSquare, H2RDF+ and SHARD, because it incurs heavy preprocessing of the RDF data to precompute and materialize join results. Spartex outperforms other systems by up to two orders of magnitude. The utilization of direct inter-vertex communication for join evaluation allows Spartex to evaluate queries in a more efficient way. Furthermore, the coarsening strategy helps in solving the queries in fewer iterations by evaluating multiple joins concurrently.

**Bio2RDF dataset:** Table 3.4 shows the results for Bio2RDF. S2RDF and CliqueSquare are excluded as they failed to process this dataset. Similar to its behavior in the other datasets, SHARD still performs worse than all other systems due to the MapReduce overhead and lack of efficient indices. Spartex outperforms both H2RDF+ and SHARD in all queries by up to an order of magnitude.
3.5.2 Optimizer

To evaluate the effectiveness of our query optimizer and cost model, we need to show that the selected plan is efficient. We use LUBM and consider only the complex queries; L1, L3, L7, P and D. These queries are solved in several iterations and generate large intermediate and/or final results\(^9\). In this experiment, we execute all possible plans for each query. Queries L1, L3 and L7 have the same structure; each has 6 possible trails. Queries P and D have 36 and 176 possible trails, respectively.

Figure 3.4(a) shows the estimated versus the actual number of messages transferred between vertices during the execution of the selected plan for each query. Our approach estimates accurately the total number of messages for most queries. 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. Our cost function is monotonically increasing, hence, it can not capture this sudden drop of intermediate results. Nonetheless, the number of messages for the last iteration is the same for all plans; therefore, our optimizer selects an efficient plan.

Figure 3.4(b) shows the fastest and slowest execution times for each query and compares them with the execution time of the plan selected by our optimizer. For all complex queries, our optimizer selects a plan that is either optimal, or performs close to the optimal. Note that for P, there were 19 plans that never finish because of the huge number of generated messages, which cause network contention.

3.5.3 Scalability

To evaluate the scalability of Spartex, we conduct two experiments: (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.

**Data Scalability:** Using the LUBM data generator, we generate five datasets:

\(^9\)L6 is very selective and L2, L4, L5 are solved within a single iteration without communication
LUBM-500, LUBM-1000, LUBM-2000, LUBM-4000 and LUBM-8000. Since simple queries touch almost the same amount of data regardless of the data size, Spartex provides almost a steady performance for these queries as shown in Figure 3.5(a). In Figure 3.5(b), we show that Spartex scales well for the complex LUBM queries. 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.

**Machine Scalability:** In this experiment, we vary the number of workers while the dataset is fixed to LUBM-10240. Figure 3.6 shows the scalability results for Spartex as the number of workers increases. Spartex achieves almost ideal scalability up to 144 workers for both simple and complex queries. After that, query response times are dominated by the communication cost which grows as we increase the number of workers.

### 3.5.4 Rich RDF Analytics

To demonstrate the effectiveness of Spartex on rich RDF analytics, we evaluate the three case studies described in Section 3.2.2. Since no other system can fully support these case studies, we compare against combinations of SPARQL engines and graph processing systems. Specifically, we combine S2RDF [31] and H2RDF+ [23] SPARQL...
engines with four different graph analytics systems; GPS [102], GraphX [106], PowerGraph [50] and PEGASUS [107].

Figure 3.7 shows the wall time of the first two use cases (see Section 3.2.2) for the LUBM-4000 dataset. Some graph engines take more time than others to load the graph. For example, PowerGraph and GPS require around 10 minutes for graph loading and indexing compared to only 2 minutes for GraphX. For fair comparison, we exclude the graph loading time for all systems. All combinations in this experiment require the data to be moved between multiple systems and be formatted accordingly.

In the first use case, graph analytics are executed prior to query evaluation. PowerGraph, GPS and PEGASUS are used to evaluate PageRank and degree centrality
algorithms and the output is stored in HDFS. The output of GraphX is materialized as in-memory RDDs which are then merged with the original graph and given as input to S2RDF. PEGASUS performed worse than all other graph engines confirming that MapReduce approaches do not perform well for graph analytics. For H2RDF+ based approaches, the results are converted to RDF format and passed to H2RDF+ along with the original RDF graph. Then, H2RDF+ partitions the input data, builds its RDF indices, and evaluates the SPARQL query. Similarly, S2RDF+ preprocesses the RDF data (i.e., original graph and computed PageRank and centrality values) to build its indices. S2RDF requires heavy preprocessing to pre-compute join reductions for each two vertical partitions in the data. The results are then materialized as tables in HDFS. While both GraphX and S2RDF are based on Spark, S2RDF does not run directly on Spark; instead, it translates SPARQL queries into SQL jobs which are then executed on top of Spark SQL. In general, the cost of data formatting and indexing in all combinations is substantial, accounting for more than 80% of the
Spartex performs better than all above-mentioned systems, because all processing is done within the same framework, without additional data formatting or indexing. Note that, for pure SPARQL queries, Spartex performs significantly better than H2RDF+ and S2RDF. The same applies on the second use case; however, since the SSSP algorithm is not available in PEGASUS, we only compare to GPS+H2RDF+, PowerGraph+H2RDF+ and GraphX+S2RDF. Notice that the SPARQL query evaluation part is minimal compared to the total time. Therefore, using a more efficient specialized RDF engine (e.g. AdPart or TriAD), will not help because the total runtime is dominated by the formatting and indexing phases. The same applies when using a more efficient graph analytics engine like PowerGraph.

The third use case, SamplID [12] (see Section 3.2.2.3), starts by loading the RDF data into HDFS and transforming it into an unlabeled directed graph, which is then processed by Apache Pig and Giraph to evaluate PageRank and degree centrality. The results are stored in RDF format with each triple assigned a score that denotes its importance. Figure 3.8 shows the time of each phase of the SamplID pipeline for two real datasets; KEGG and LinkedGeoData. The rewriting (RDF to unlabeled graph) and the reverse (unlabeled graph to RDF) phases consume most of the time. Spartex does not incur such overhead. As a result, Spartex provides a single system for the
Table 3.5: Query engine. LUBM-10240; runtime (sec)

<table>
<thead>
<tr>
<th>LUBM-10240</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
<th>L6</th>
<th>L7</th>
<th>P</th>
<th>D</th>
<th>GMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartex-Native</td>
<td>2.881</td>
<td>0.406</td>
<td>2.953</td>
<td>0.001</td>
<td>0.001</td>
<td>0.010</td>
<td>2.386</td>
<td>3.408</td>
<td>4.768</td>
<td>0.222</td>
</tr>
<tr>
<td>AdPart-NA</td>
<td><strong>2.743</strong></td>
<td><strong>0.120</strong></td>
<td><strong>0.320</strong></td>
<td>0.001</td>
<td>0.001</td>
<td>0.040</td>
<td>3.203</td>
<td>5.724</td>
<td>4.793</td>
<td><strong>0.193</strong></td>
</tr>
<tr>
<td>TriAD</td>
<td>6.023</td>
<td>1.519</td>
<td>2.387</td>
<td>0.006</td>
<td>0.004</td>
<td>0.114</td>
<td>17.586</td>
<td>19.839</td>
<td>65.628</td>
<td>1.035</td>
</tr>
<tr>
<td>TriAD-SG (100K)</td>
<td>5.392</td>
<td>1.774</td>
<td>4.636</td>
<td>0.009</td>
<td>0.005</td>
<td>0.010</td>
<td>21.567</td>
<td>44.135</td>
<td>144.256</td>
<td>1.119</td>
</tr>
<tr>
<td>SHAPE</td>
<td>25.319</td>
<td>4.387</td>
<td>25.960</td>
<td>1.603</td>
<td>1.574</td>
<td>1.567</td>
<td>15.026</td>
<td>N/A</td>
<td>N/A</td>
<td>5.575</td>
</tr>
</tbody>
</table>

entire SamplD pipeline with almost one order of magnitude better performance.

### 3.5.5 Native Spartex Query Engine

In this section, we compare Spartex against state-of-the-art native distributed RDF systems. For fair comparison, we implemented our SPARQL query engine only (i.e., not the entire Spartex) outside any vertex-centric framework, using C++ and MPI. We call this version *Spartex-Native*. We compare Spartex-Native against the following systems: (i) AdPart [45] is the current state-of-the-art distributed in-memory RDF system. It uses workload adaptive graph partitioning and implements locality-aware query optimizations. Since our work is orthogonal to graph partitioning, we compare against the non-adaptive version, called AdPart-NA. (ii) TriAD[^10] [25] is a distributed in-memory RDF engine that uses sophisticated graph partitioning and asynchronous message passing. We also compare against TriAD-SG, a version of TriAD that uses graph summaries for join-ahead pruning. (iii) SHAPE [24] is a distributed engine that uses RDF-3X for storage and relies on static replication. We configure SHAPE with all possible optimizations and make sure that all queries can be solved without communication between nodes.

Table 3.5 shows the runtime in seconds for the LUBM-10240 dataset. For the simple queries, all systems except SHAPE, perform similarly. SHAPE is much slower because replicas are stored together with the original triples, resulting in significant overhead. For complex queries, Spartex-Native is significantly faster than TriAD,

[^10]: Trinity.RDF [21] is not publicly available; therefore we compare to TriAD, which reported [25] performance superior to Trinity.RDF.
Table 3.6: Query engine. YAGO2; runtime (msec)

<table>
<thead>
<tr>
<th>YAGO2</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
<th>Y4</th>
<th>GMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartex-Native</td>
<td>38</td>
<td>126</td>
<td>35</td>
<td>33</td>
<td>49</td>
</tr>
<tr>
<td>AdPart-NA</td>
<td>19</td>
<td>46</td>
<td>570</td>
<td>77</td>
<td>79</td>
</tr>
<tr>
<td>TriAD</td>
<td>16</td>
<td>1,568</td>
<td>220</td>
<td>18</td>
<td>100</td>
</tr>
<tr>
<td>SHAPE</td>
<td>1,824</td>
<td>665,514</td>
<td>1,823</td>
<td>1,871</td>
<td>8,022</td>
</tr>
</tbody>
</table>

Table 3.7: Query engine. Bio2RDF; runtime (msec)

<table>
<thead>
<tr>
<th>Bio2RDF</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>GMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartex-Native</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>26</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>AdPart-NA</td>
<td>17</td>
<td>16</td>
<td>32</td>
<td>89</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>TriAD</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>N/A</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

TriAD-SG and SHAPE because of the better execution plans and the various optimizations. For example, queries L1 and L7 are cyclic, allowing Spartex-Native, TriAD and TriAD-SG to execute multiple joins concurrently; however, Spartex-Native minimizes communication. Compared to AdPart-NA, Spartex-Native is slightly better for some complex queries and slightly worse for others. For example, query L3 returns empty results; AdPart-NA evaluates the join that generates the empty set earlier than Spartex-Native.

Table 3.6 shows the runtime in msec for YAGO2. Queries Y1 and Y2 are simple, whereas Y3 and Y4 are complex. Compared to AdPart-NA and TriAD, Spartex-Native is slightly slower for some queries, but significantly faster for others. In terms of geometric mean (GMean) over all queries, Spartex-Native is clearly better. The bad performance of SHAPE is explained by the fact that the 2-hop forward partitioning placed all data in a single partition; therefore, although 12 machines are available, one is overloaded. TriAD-SG is not listed because we do not know the optimal number of summary graph partitions, a process that requires empirical evaluation [25]. Similar results are shown in Table 3.7 for Bio2RDF. Again, Spartex-Native is the fastest for the majority of the queries, and achieves the lowest geometric mean. TriAD-SG is excluded for the same reason as above. SHAPE is also excluded because, with 2-hop forward partitioning, it failed to preprocess the Bio2RDF dataset within reasonable time.
In summary, our core SPARQL query engine is at least as fast as the state-of-the-art distributed RDF systems. This shows the effectiveness of our query optimizer and the efficiency of our SPARQL query engine. It also demonstrates that our SPARQL operator can be used as an efficient stand-alone distributed RDF engine.
3.6 Conclusion

Spartex bridges the gap between specialized RDF stores and generic graph engines. Our proposed SPARQL extension allows the invocation of user defined vertex-centric programs. Declarative SPARQL queries as well as procedural graph algorithms can be pipelined without moving and reformatting data between different systems. By coupling our SPARQL operator with a cost-based optimizer, complex SPARQL queries are evaluated efficiently. Spartex significantly outperforms approaches that employ combinations of existing systems when performing rich RDF analytics. For pure SPARQL queries, Spartex is highly scalable and as fast as the best specialized existing RDF engines. For future work, we will consider the opportunity for cross-algorithms optimizations that may be possible when combining SPARQL and graph analytics in the same query. For example, most of the execution time is spent on analytical algorithms. Therefore, multiple query optimization techniques can be employed to evaluate multiple algorithms concurrently. Notice that analytical queries may consist of a pipeline of operators. Hence, pipelines of multiple queries need to be aligned in a way that minimizes the overall execution time.
Chapter 4

Lusail: A System for Querying Linked Data at Scale

4.1 Introduction

The Resource Description Framework (RDF) is extensively used to represent structured data on the Web. RDF uses a simple graph data model in which the data are in the form of triples (subject, predicate, object). A key feature is the ability to link two entities from two different RDF datasets which are maintained by two independent authorities, as shown in Figure 4.1. Through such links, large decentralized graphs are created among a large number of geo-distributed RDF stores where each RDF store can be queried through its own SPARQL endpoint. The Linked Open Data (LOD) Cloud is one such decentralized RDF graph, and it has more than 150 billion triples in around 10000 datasets\(^1\) in different domains, such as media, government, and life sciences [15].

Users can retrieve data from an individual dataset by issuing SPARQL queries against its SPARQL endpoint. However, it is often very useful to issue SPARQL queries that integrate data from multiple RDF datasets, which would require federated query processing. For example, Figure 4.2 shows a query \((Q_a)\) on data from the LUBM benchmark [69] at two endpoints. \(Q_a\) returns all students who are taking courses with their advisors along with the URI and location of the advisors’ alma mater. \(Q_a\) has three answers: (Kim, Joy, CMU, ”CCCC”), (Kim, Tim, MIT, ”XXX”) and (Lee, Ben, MIT, ”XXX”). One cannot simply evaluate \(Q_a\) independently at each endpoint.

\(^1\)http://stats.lod2.eu/
Figure 4.1: A decentralized graph for two universities managed by independent geo-distributed SPARQL endpoints (EP). The red dotted line represents an interlink between endpoints, i.e., a vertex in an endpoint referring to another vertex in another endpoint. Thus, to get the address of the university from which Tim got his PhD, the interlink from EP2 to EP1 must be traversed.

```
    ?S rdf:type ub:graduateStudent .
    ?P rdf:type ub:associateProfessor .
    ?C rdf:type ub:graduateCourse .
}
```

Figure 4.2: A SPARQL query, $Q_a$, over a decentralized RDF graph across different universities. This query has to traverse the interlink between EP2 and EP1.

and concatenate their results as this will miss the results about Tim since EP2 does not have the address of MIT. Instead, we need a federated query processor that can automatically identify the endpoints that can answer each triple pattern, detect interlinks between endpoints and automatically traverse them to compute the query answer. A federated query processor would decompose $Q_a$ into subqueries, send each subquery to the relevant endpoint, and compute the answer to $Q_a$ from the results of the subqueries. Computing such an answer typically requires the federated query processor to join the results obtained from the endpoints. For example, in $Q_a$ this join would combine the address of MIT from EP1 with the information about Tim from EP2.

Conceptually, a query like $Q_a$ can be processed by sending each of its triple patterns to all the endpoints, retrieving all matching triples from the endpoints, and
joining all of these triples at the federated query processor to compute the query answer. This strategy is clearly inefficient since it sends triple patterns to endpoints even if they have no answers for them, retrieves triples that may not be relevant, and joins triples at the federated query processor even if they could be joined at the endpoints. Thus, this strategy would result in an unnecessarily large number of requests to the endpoints and unnecessarily large amounts of data retrieved from the endpoints and transferred over the network to the federated query processor. To avoid these unnecessary overheads, it is important for federated query processors to push as much processing as possible to the endpoints.

Existing SPARQL federated query processing systems rely on schema information to push processing to the endpoints. For example, they use SPARQL ASK queries to check whether or not a triple pattern has an answer at an endpoint [39]. If a group of triple patterns can be answered exclusively by one endpoint, then it is possible to send this group to the endpoint as one unit, known as an exclusive group. Relying solely on schema information is not effective since RDF sources often utilize similar ontologies (e.g., EP1 and EP2 in Figure 4.2 have the same predicates), thus a triple pattern could be answerable by multiple endpoints and therefore cannot be part of an exclusive group. In this case, the triple pattern is sent to all the endpoints that can answer it and the values in the retrieved triples are bound to other triple patterns; the triple patterns with bound values are sent to the endpoints to retrieve further triples. This is known as a bound join operation, and effectively amounts to the query being processed one triple pattern at a time.

This strategy retrieves unnecessary data from the endpoints, since it retrieves all data matching a triple pattern even if this data is not useful for the rest of the query. Moreover, this process limits the available parallelism since only one join step can be processed at a time, and the federated query processor has to wait for the results of this join step before issuing the next join. To quantify the inefficiency of this
approach, we note that our experiments on FedEx [39], a federated SPARQL system that uses this approach and that was shown to outperform similar systems [54], show that increasing the number of endpoints from 1 to 4 can lead to 6 orders of magnitude increase in the number of requests sent to the endpoints, and 3 orders of magnitude increase in the running time (details in Section 2.5.7).

This chapter addresses the limited ability of existing systems to push query processing to the local endpoints. We present Lusail, a scalable and efficient system for federated SPARQL query processing over decentralized RDF graphs. Lusail is the first system to decompose the federated query based on instance information not just schema information. That is, Lusail decomposes the query based on knowledge of the locations of the actual RDF triples matching triple patterns in the query. This knowledge helps us identify, for example, that the instances matching the variable \(?S\) in \(\langle ?S, \text{ub:advisor}, ?P \rangle\) and \(\langle ?S, \text{ub:takesCourse}, ?C \rangle\) in \(Q_a\) are always located in the same endpoint, so these triple patterns can be joined locally at the endpoint even though schema information tells us that both endpoints can answer both triple patterns. In contrast, the instances matching the variable \(?U\) in \(\langle ?P, \text{ub:PhDDegreeFrom}, ?U \rangle\) and \(\langle ?U, \text{ub:address}, ?A \rangle\) are sometimes located in different endpoints, so these triple patterns cannot be joined locally.

Lusail processes queries in a two-phase strategy: (i) Locality-Aware DEcomposition (LADE) of the query into subqueries to maximize the computation at the endpoints and minimize intermediate results, and (ii) Selectivity-Aware and Parallel Execution (SAPE) to reduce network latency and increase parallelism. Unlike prior approaches, the decomposition of LADE is based not only on schema information but also on instance information, i.e., the location of triples satisfying the triple patterns in the query. SAPE decides the order of executing the subqueries generated by LADE based on their result sizes and degree of parallelism.

In summary, our contributions are as follows:
• We investigate the scalability of the state-of-the-art method, FedEx [39], while varying data sizes and number of endpoints.

• A locality-aware decomposition method that dramatically reduces the number of remote requests and allows for better utilization of the endpoints. We also provide a proof of correctness. (Section 4.3)

• A cost model that uses lightweight runtime statistics to decide the order of submitting subqueries and the execution plan for joining the results of these subqueries in a non-blocking fashion. This leads to a parallel execution that balances between remote requests and local computations. (Section 4.4)

• Our experiments on real data and synthetic benchmarks with billions of triples show that Lusail outperforms state-of-the-art systems by up to three orders of magnitude and scales up to 256 endpoints compared to 4 endpoints in existing systems. (Section 4.5)

We present the architecture of Lusail in Section 4.2 and conclude in Section 4.6.

4.2 The Lusail Architecture

Figure 4.3 shows the architecture of Lusail. Lusail analyzes each query to identify the relevant endpoints and its correct decomposition that achieves high parallelism and minimal communication cost. After that, Lusail sends the subqueries to the relevant endpoints, joins their results, and sends the query answer back to the user.

Locality-Aware Decomposition (LADE): Query decomposition starts by identifying the relevant endpoints (source selection). Like similar systems, we use a set of SPARQL ASK queries, one for each triple pattern. Furthermore, LADE takes the additional step of checking, for each pair of triple patterns with a common (or join) variable, whether the pair can be evaluated as one unit by the relevant endpoints.
To do so, LADE utilizes the knowledge of the locations of the actual RDF triple instances matching a query variable. The result of this check determines a group of triple patterns, i.e., a subquery, that can be sent together to an endpoint. Based on this analysis, LADE decomposes the query into a set of independent subqueries. Lusail caches the results of both the source selection phase and the check queries that determine the triple patterns which cannot be executed locally at an endpoint.

**Selectivity-Aware Planning and Parallel Execution (SAPE):** SAPE takes as input the set of subqueries produced by LADE and schedules them for execution. This set of independent subqueries can be submitted concurrently for execution at each of the relevant endpoints, and Lusail can use one thread per endpoint to collect their results. SAPE uses cardinality estimates for the different triple patterns to delay subqueries that are expected to return large results. The results of these subqueries will then need to be joined by SAPE using a parallel join, where the join order is determined based on the actual sizes of the subquery results. SAPE achieves a high degree of parallelism while minimizing the communication cost by (i) obtaining results from different endpoints simultaneously, and (ii) utilizing different threads in joining
the results.

**Elastic Request Handler (ERH):** Lusail utilizes multiple threads for evaluating the ASK queries from LADE or the subqueries from SAPE at the endpoints. ERH manages the allocation of threads from one or more machines to these tasks, where the number of available threads is determined by the number of physical cores.

### 4.3 Locality-Aware Decomposition

To push as much processing to the endpoints as possible, LADE maximizes the number of triple patterns in a given query that can be sent together to each endpoint. In a decentralized RDF graph, data instances matching a pair of triples may not be located in the same endpoint, e.g., the triples having \(?U\) as a common variable in Figure 4.1. Thus, putting this pair in the same subquery may miss results. LADE starts by analyzing which triples cannot be in the same subquery, and identifying the common variables in these triples as *global join variables (GJV)*. Then, it decomposes the conjunction of triple patterns into subqueries. We assume no prior knowledge of the data sources, such as schema, data distribution, or statistics. LADE relies solely on a set of check queries written in SPARQL.

In this section, we only discuss how Lusail evaluates conjunctive SPARQL queries. However, Lusail also supports queries with joins on variable predicates as well as UNION, FILTER, LIMIT, and OPTIONAL statements (see Appendix A.2). For example, Lusail determines where to add the FILTER and OPTIONAL clauses during query decomposition and during the global join evaluation.

#### 4.3.1 Detecting Global Join Variables

A global join variable \(v\) is a variable that appears in at least two different triple patterns such that these triple patterns, when taken together, cannot be solved by a single endpoint. A global join between data coming from two or more endpoints
Figure 4.4: Locality analysis of data instances in EP1 and EP2 from Figure 4.1 that match \(?S\), \(?U\), and \(?P\) in a pair of triple patterns in \(Q_a\).

will be needed. Given two triple patterns, \(TP_i\) and \(TP_j\), in a subquery, a GJV may appear in the triple patterns as: (i) object in \(TP_i\) and subject in \(TP_j\), (ii) object in both patterns, or (iii) subject in both. Let \(v_i\) and \(v_j\) be the sets of instances of \(v\) that satisfy \(TP_i\) and \(TP_j\), respectively.

\(Q_a\) (Figure 4.2) has four variables appearing in more than one triple pattern, namely \(?S\), \(?U\), \(?P\), and \(?C\). Figure 4.4 shows our analysis for the first three variables. In EP1 and EP2, all instances matching \(?S\) in \(⟨?S, \text{ub:advisor}, ?P⟩\) are co-located with all instances matching \(?S\) in \(⟨?S, \text{ub:takesCourse}, ?C⟩\). Thus, \(?S\) is not a GJV and hence the two corresponding triple patterns can be sent together in a single subquery to each relevant endpoint. However, for the triples involving \(?U\), \(⟨?P, \text{ub:PhDDegreeFrom}, ?U⟩\) and \(⟨?U, \text{ub:address}, ?A⟩\), we notice that in EP2 there is a professor, Tim, who got his PhD from another university. Thus, to get the address of that university, we need to perform a join between data fetched from EP1 and EP2. Therefore, \(?U\) is a GJV.

We now describe how LADE detects GJVs by determining the actual location of data instances depending on the roles they play, i.e., object or subject. We first discuss how to merge two triple patterns and then generalize to more than two (see Algorithm 4). Two triple patterns \(TP_i\) and \(TP_j\) are put together in a single subquery under two conditions: (i) both triple patterns have the same list of relevant endpoints,
Figure 4.5: A Lusail SPARQL check query to detect whether \(?P\) is a global join variable or not. The check query returns zero or only one value.

and (ii) each relevant endpoint can fully answer both triple patterns without missing any result, i.e, all instances that match \(v\) in \(TP_i\) and \(TP_j\) are in the same endpoint.

**Object and Subject.** Consider the variable \(?U\) in \(Q\)a (Figure 4.2). It appears as an object in \(TP_i\): \(?P\) ub:PhDDegreeFrom \(?U\) and as a subject in \(TP_j\): \(?U\) ub:address \(?A\). Checking the location of the data instances \(v_i\) and \(v_j\) that match \(?U\) in each endpoint has two cases: (i) remote instances, where \(v_i\) and \(v_j\) are located in different endpoints, i.e., all or some professors received their PhD from another university (in a different endpoint); e.g., EP2 in Figure 4.4, and (ii) local instances, where all \(v_i\) and \(v_j\) are located in the same endpoint, i.e., all professors teaching in a university \(A\) received their PhD from \(A\) (in the same endpoint), e.g., EP1 in Figure 4.4.

We check the relative complement (i.e., set difference) of \(v_i\) and \(v_j\) in all relevant endpoints by sending a SPARQL query to each endpoint. If one or more of these endpoints has instances in \(v_i\) but not in \(v_j\), then \(v\) is a GJV. At each endpoint, we check for each data instance appearing as an object in \(TP_i\) whether this instance appears locally as a subject in \(TP_j\). Once a common variable is found to be a GJV, the triple patterns cannot be combined in the same subquery even for those endpoints that return an empty result for the difference in the instances, e.g., the pair of triple patterns where \(?U\) is common (Figure 4.4). This allows us to have simple plans and better parallel execution.

Set difference \((-\)) is implemented using \textit{FILTER NOT EXISTS} (Figure 4.5) where \(TP_i\): \(\langle ?S, \text{Predicate}_i, ?P \rangle\), and \(TP_j\): \(\langle ?P, \text{Predicate}_j, ?C \rangle\). If there is a triple pattern
setting a type for $v$ ($\langle ?P, \text{rdf:type}, T \rangle$), we use it to limit the check to only the relevant values of $v$. Since Lusail needs to only know whether the result is an empty set, we use $\text{LIMIT 1}$.

**Objects/Subjects Only.** If a variable appears only as *object*, respectively *subject*, in both triple patterns $TP_i$ and $TP_j$, Lusail checks in each relevant endpoint that $v_i - v_j$ and $v_j - v_i$ are both empty. As shown in Figure 4.4, the variable $?S$ appears as subject in both $\langle ?S, \text{ub:advisor}, ?P \rangle$ and $\langle ?S, \text{ub:takesCourse}, ?C \rangle$. Having two empty sets in the same endpoint means that (i) any graduate student $?S$ having an advisor $?P$ should take a course $?C$ and (ii) any graduate student $?S$ taking a course $?C$ should have an advisor $?P$, all located in the same endpoint.

Algorithm 4 receives a query and a list of relevant endpoints and outputs a set of GJVs ($V$) along with the triple patterns that caused each variable to be a GJV. It assumes that source selection is already done using ASK requests or the Lusail cache. The algorithm starts by retrieving the set of query variables and triple patterns. Each variable is associated with its subject and object patterns (line 2). The algorithm iterates over the variables to detect GJVs.

If the variable joins triple patterns from different sources, then it is a GJV (lines 8-11). There is no need to check the other conditions. Otherwise, Algorithm 4 formulates a set of check queries as discussed above. For the *object only* and *subject only* cases, Algorithm 4 formulates check queries for all possible pairwise combinations of the triple patterns associated with the variable (lines 13-14). For the *object and subject* case, the check query is a combination of object triples and subject triples (lines 15-16).

The algorithm uses the elastic request handler (Figure 4.3) to execute check queries. It initializes the handler with the size of the thread pool and the set of endpoints (line 18). Then, it iterates over all check queries and executes each at the relevant endpoints (lines 19-23). If the query returns any results, then the cor-
**Algorithm 4: Detecting Global Join Variables**

The responding variable is a GJV (lines 22-23). The algorithm returns the set of GJVs along with the triple patterns that caused each variable to be a GJV.

Let $|V|$ be the number of variables appearing in more than one triple pattern in the query and $|T|$ be the number of triples. Since check queries are formed for pairs of triples, the maximum number of check queries, $C_Q$, is bound by $O(|V| \times |T|^2)$. Assuming $N$ relevant endpoints, LADE creates a maximum of $N \times C_Q$ requests. Since the number of triple patterns in real-world SPARQL queries is usually small [103], the number of GJVs is also small. Therefore, $N \times C_Q$ will be typically small. In addition, the check queries are lightweight and have minimal overhead (see Section 4.5.4).

### 4.3.2 Query Decomposition

Algorithm 5 decomposes a query $Q$ into multiple subqueries to be sent to different endpoints. If $Q$ has no GJVs, the algorithm returns $Q$ (line 3). Otherwise, LADE uses the set of GJVs and the source selection information to decompose $Q$. It iterates
Algorithm 5: Query Decomposition

over all join variables in any order using the current join variable as a root. It tries to find the best decomposition that leads to a set of subqueries with minimal execution cost (cost estimation is discussed in Section 4.4). The algorithm has two phases: branching (lines 9-30) and merging (lines 32).

In the branching phase, we build a query tree with the current join variable as its root (line 9). An initial set of subqueries is created at the root, one subquery per child (lines 13-20) and each subquery is expanded through depth first traversal (lines 21-30). A triple pattern is added (lines 24-25) if both the subquery and the
triple pattern have the same relevant sources, and the addition of the pattern does not
cause a query variable to be a GJV. If one of the conditions is invalid, a new subquery
is created from the current triple pattern and added to the set of subqueries (lines
27-28). In both cases, the edge destination node is added to the nodes stack and
marked as visited (lines 29-30).

The merging phase (line 32) starts once all triple patterns are assigned to one of
the subqueries (line 31). The function \textit{mergeSubQ} (line 32) loops through the set
of subqueries and merges a pair of subqueries if they have common variables, the same
relevant sources, and no pair of triple patterns from both subqueries has a common
variable that is global. If the estimated cost (line 33) of the current decomposition is
less than other decompositions, the algorithm updates the minimum cost and selects
the current decomposition as the current best. The algorithm continues to check
other possible decompositions using the remaining join variables.

The algorithm returns the best subquery decomposition (line 37). For simplicity,
the pseudo-code of the algorithm assumes a connected query graph. Lusail also
supports queries with multiple disconnected subgraphs, in which case it executes
each subquery independently and creates a special join variable that connects these
subqueries, if possible.

Figure 4.6 shows two possible decompositions for \textit{Q}_a (Figure 4.2), which has two
GJVs, namely \texttt{?U} and \texttt{?P}. The generated set of subqueries may change depending
on the order in which variables are selected during query decomposition (line 5 in
Algorithm 5). However, all decompositions produce the same result set and do not
miss any triple (details in Section 4.3.4), but some decompositions may generate
more intermediate results and thus cost more. To avoid a costly decomposition,
LADE enumerates all possible decompositions and chooses at compile time the best
decomposition expected to minimize the intermediate results.

The outer loop (line 5 in Algorithm 5) iterates over the set of GJVs to generate all
Figure 4.6: Two possible decompositions of $Q_a$, where the GJVs are $?U$ and $?P$. Any pair of predicates, which causes a variable to be a GJV, cannot be in the same subquery.

possible query decompositions. Each iteration performs a depth first traversal, whose complexity is $O(|V| + |T|)$ where $|V|$ is the total number of query variables and $|T|$ is the number of triple patterns. Since the number of iterations is small (i.e. number of GJVs), the algorithm complexity is still bound by $O(|V| + |T|)$.

4.3.3 Generic SPARQL Queries

So far, we only discussed how Lusail evaluates conjunctive SPARQL queries. However, Lusail also supports queries with joins on variable predicates, UNION, FILTER, LIMIT, and OPTIONAL statements, see the listed queries in Appendix A.2. In a nutshell, Lusail determines where to add clauses, such as FILTER, LIMIT, and OPTIONAL, in the subqueries or during the global join evaluation. For example, FILTER statements with a single variable are pushed with relevant subqueries and thus handled by the endpoints. In case of multi-variable filters, if both variables exist exclusively in the same subquery, then they are handled by the endpoints. Otherwise, Lusail considers the filter clause during the join evaluation phase. Queries with optional statement are handled similarly. If the optional pattern exist within a single subquery and has nothing to do with a global join variable, then the endpoints evaluate it. Otherwise, Lusail takes care of the optional results when finalizing the query answer. Limit and Union statements have naive implementation in Lusail. For
queries with a limit clause, SAPE stops the evaluation of the last subquery once the required number of results is reported. Queries with union are split into multiple queries which are evaluated independently and their answers are concatenated after removing the duplicates. Finally, if the query has variable predicate then there are two cases; (i) if no predicate join is required or the predicate join exists exclusively within a single subquery, then the triple pattern is pushed to the relevant endpoints for evaluation and no special handling is required. Otherwise, (ii) If there is a predicate join across several subqueries, then SAPE handles it during the join evaluation phase.

4.3.4 Result Completeness

Missing Results. The optimization introduced by LADE assigns triple patterns to different subqueries based on the concept of locality. Results would be missed in two cases. Case 1: A subquery contains a set of triple patterns where a GJV is considered to be local. This can only happen if the subquery contains triple patterns that access predicates through interlinks, e.g., a subquery that contains ⟨?P, ub:PhDDegreeFrom, ?U⟩ and ⟨?U, ub:address, ?A⟩ will cause Q_a to miss the result (Kim, Tim, MIT, ”XXX”) when the subquery is submitted to EP2. However, such a case cannot happen since LADE puts triple patterns into the same subquery only if the data instances matching them are located in the same endpoint. Lemma 1 formalizes this argument.

Case 2: A subject or object may be present in more than one endpoint, e.g., EP1 has ⟨a1, p, b⟩, ⟨b, q, c1⟩ and EP2 has ⟨a2, p, b⟩, ⟨b, q, c2⟩. Having the pair of triple patterns, ⟨?x, p, ?y⟩ and ⟨?y, q, ?z⟩, in the same subquery does not miss the local triples matching the query. Lusail first detects ?y as a local join variable and then performs the join between the results of the same subquery from different endpoints at the Lusail server (see Section 4.4.2).

Lemma 1. Any local join variable detected by LADE is a true local join variable.
**Proof:** Let $v$ be the join variable and $TP(v) = \{tp_1, tp_2, ... tp_k\}$ be the set of triple patterns in which $v$ appears. $v$ can appear in $TP(v)$ as subject only, object only, or subject and object.

**Subject only:** In this case, $\forall_{tp_i \in TP(v)} tp_i. subj = v$. Let $B_i$ and $B_j$ be the set of bindings of $v$ from triples $tp_i$ and $tp_j$, respectively. LADE decides that $v$ is a local join variable iff: $\forall_{0 < l < t} \forall_{0 < i < k, i \neq j} B_i(ep_l) - B_j(ep_l) = \phi$ and $B_j(ep_l) - B_i(ep_l) = \phi$ where $k = |TP(v)|$ and $t$ is the number of relevant endpoints. At each relevant endpoint, $B_i - B_j = \phi$ means that each endpoint can fully evaluate $tp_i \bowtie tp_j$ locally. This means that $v$ is a true local join variable and there is no need to join $tp_i$ and $tp_j$ across endpoints. The same applies for $B_j - B_i$.

**Object only:** In this case, $\forall_{tp_i \in TP(v)} tp_i.obj = v$. The same analysis of the subject only case applies.

**Subject/Object:** Let $TPS(v) = \{tps_1, ... tps_s\}$ and $TPO = \{tpo_1, ... tpo_o\}$ be the set of triples in which $v$ appears as subject and object, respectively. $\forall_{tps_i \in TPS(v)} tps_i. subj = v$ and $\forall_{tpo_j \in TPO(v)} tpo_j. obj = v$. Let $B_i$ and $B_j$ be the set of bindings of $v$ using triple $tps_i$ and $tpo_j$, respectively. LADE decides that $v$ is a local join variable iff: $\forall_{0 < l < t} \forall_{0 < i < s, 0 < j < o} B_i(ep_l) - B_j(ep_l) = \phi$. At each relevant endpoint, $B_i - B_j = \phi$ means that each endpoint can fully evaluate $tps_i \bowtie tpo_j$ locally. It also means that $v$ is a true local join variable and there is no need to join $tps_i$ and $tpo_j$ across endpoints. □

**Extraneous computations.** In some cases, LADE may detect a join variable as being global while the triple patterns sharing this variable could be solved together locally at the endpoints. For example, the variable ?P in ⟨?S, ub:advisor, ?P⟩ and ⟨?P, ub:teacherOf, ?C⟩. As shown in EP1 (Figure 4.4), there is an advisor (Ann) who has joined MIT but who is not a teacher of any course yet. ?P will be considered as a GJV based on our checks. However, it is clearly safe to send both triple patterns in the same subquery since there is no need to access data in remote endpoints. Adding
more checks to avoid such cases would be too expensive since it will require accessing all other relevant endpoints. Such cases may lead to query plans with unnecessary GJVs, i.e., more remote requests and more join computations at global level rather than at the endpoints. Lemma 2 shows that assuming that a join variable is global, while it is not, does not affect the correctness of the results.

**Lemma 2.** Any local join variable $v$ can be selected as a global join variable without affecting the result correctness.

**Proof:** Let $TP(v) = \{tp_1, ..., tp_k\}$ be the set of triple patterns in which $v$ appeared. If $v$ is a local join variable, each relevant endpoint can evaluate $TP(v)$ as a single subquery. The set of bindings of the local join variable $v$ is simply the union of all bindings from all relevant endpoints, i.e. $B_l(v, TP(v)) = \bigcup_{0 < i < t} B_l(v, TP(v), ep_i)$ where $t$ is the number of relevant endpoints. Assume now that $v$ is considered a global join variable. In this case, each endpoint will evaluate each triple pattern independently and the results are joined at the global level. Let $B_g(v, tp_j) = \bigcup_{0 < i < t} B_g(v, tp_j, ep_i)$ be the set of bindings of the global variable $v$ for triple pattern $tp_j$. Then, the global bindings of $v$ is $B_g(v, TP(v)) = B_g(v, tp_1) \bowtie B_g(v, tp_2) ... \bowtie B_g(v, tp_k)$. Since $v$ should be a local join variable, then the join between different endpoints is always empty. This means that $B_g(v, TP(v)) = \bigcup_{0 < i < t} B_g(v, tp_1, ep_i) \bowtie B_g(v, tp_2, ep_i) ... \bowtie B_g(v, tp_k, ep_i)$ which is equivalent to evaluating all triples in $TP(v)$ as a single subquery and taking the union across the relevant endpoints. Consequently, $B_g(v, TP(v)) = B_l(v, TP(v))$. □

### 4.4 Selectivity-Aware Execution

The Selectivity-Aware Planning and parallel Execution (SAPE) algorithm is responsible for choosing: (i) a good execution order for the subqueries that would balance between the communication cost and the degree of parallelism and (ii) a good join order for the subquery results. An overview of SAPE is shown in Figure 4.7. SAPE
estimates the cardinality of the different subqueries and accordingly delays subqueries expected to return large results. Non-delayed subqueries are evaluated concurrently while the delayed ones are evaluated serially using bound joins. The objective of SAPE is to maximize the degree of parallelism while minimizing the communication cost in terms of number of requests to endpoints and size of subquery results.

4.4.1 Subquery Ordering and Cost Model

LADE outputs a set of independent subqueries that can be submitted concurrently for execution at each of the relevant endpoints. The results of these subqueries will then need to be joined at the global level. There are two extreme approaches to execute these subqueries.

The simplest approach is to simultaneously submit the subqueries to the relevant endpoints and wait for their results to start the join. For example, the subqueries of Figure 4.6 would be executed concurrently and after receiving all their results, a join phase would start. Notice that the subquery \( \langle ?U, \text{address}, ?A \rangle \) is so generic that executing it independently will retrieve all entities with addresses regardless of whether these entities match \(?U\) in the remaining subqueries (see Figure 4.1). These subqueries, which touch most of the endpoints or retrieve large amounts of intermediate results, affect query evaluation time by overwhelming the network, the
endpoints, and Lusail with irrelevant data. Examples include: (i) generic subqueries that are relevant to the majority of the endpoints, e.g., common RDF predicates such as owl:sameAs, rdf:type, rdfs:label, and rdfs:seeAlso. (ii) Simple subqueries that have one triple pattern with two or three variables, e.g., ⟨?s, ?p, ?o⟩ or ⟨?s, owl:sameAs, ?o⟩, and (iii) optional subqueries.

At the other extreme, we can submit the most selective subquery first and use the actual bindings of the variables obtained to submit the next most selective subquery with its variables bound to the values retrieved by the first subquery, and then submit the next most selective subquery in a similar fashion, and so on. While limiting the amount of intermediate results to be retrieved from the endpoints, this approach offers no parallelism beyond submitting the same subquery to multiple endpoints.

Our objective is to balance between the degree of parallelism, i.e., the number of subqueries submitted concurrently, and the communication cost, which is dominated by the size of intermediate results. Our only constraint is that Lusail should avoid collecting expensive statistics during pre-processing or at runtime. Therefore, Lusail uses only lightweight per-triple statistics during query evaluation. To fulfill our objective, we detect the subqueries expected to return substantially fewer results if some of their variables are bound to the results already obtained. The idea is to cluster subqueries based on their estimated cardinality and the number of endpoints they access while taking into account the variability in these values. To this end, we introduce the concept of delayed subqueries, which are evaluated using the actual bindings of the variables that have been already obtained. We thus follow a two-phase subquery evaluation: (i) concurrently submit non-delayed subqueries to the endpoints, and (ii) use the variable bindings obtained from the first phase to evaluate the delayed subqueries.

We introduce a cost model to determine delayed and non-delayed subqueries. SAPE assumes that subquery cardinalities follow a normal distribution, so most sub-
queries return results whose sizes are within one standard deviation of the mean. SAPE calculates the mean \( \mu \) and standard deviation \( \sigma \) values for all the cardinalities and all the numbers of relevant endpoints per subquery. Outliers, e.g., subqueries returning extremely large results (very low selectivity) or accessing a large number of endpoints compared to other subqueries, misleadingly increase the standard deviation. This may lead SAPE to consider some subqueries that are better to be delayed as non-delayed. We, therefore, apply the Chauvenet’s criterion \[108\] for detecting and rejecting outliers before computing \( \mu \) and \( \sigma \). Any subquery \( sq_i \) with cardinality \( C(sq_i) > \mu_C + \sigma_C \) is delayed, as shown in Figure 4.7. We apply the same concept for the number of relevant endpoints per subquery. With this heuristic, only subqueries (including outliers) whose results are expected to be significantly larger than the majority of subqueries will be delayed.

The cardinality of a subquery is estimated based on the cardinality of its triple patterns, which is collected during the query analysis phase using a simple SELECT COUNT query, one per triple pattern. Whenever a filter clause is available for a subject and/or object, it is pushed with the statistics query to obtain better cardinality estimates. Note that cardinality statistics per predicate are usually collected by RDF engines for their runtime query optimization \[109, 33, 30\] and it may be possible to use them to provide the required estimates. We leave this as future work.

We need to estimate the cardinality of the variables in the projection list of each subquery. The cardinality of a variable \( v \) in a subquery \( sq_i \), denoted \( C(sq_i, v) \), represents the number of bindings of \( v \). If two triple patterns \( TP_i \) and \( TP_j \) join on a variable \( v \), then the number of bindings of \( v \) at endpoint \( ep_k \) after the join will be:

\[
C(sq_i, v, ep_k) = \min(C(TP_i, ep_k), C(TP_j, ep_k))
\]

Therefore, we use the minimum cardinality of the predicates in which \( v \) is a common variable as an upper bound of the cardinality of \( v \) per endpoint. Thus, the total cardinality of \( v \) in the subquery \( sq_i \) is the sum of its cardinalities in all the relevant
endpoints $ep$, estimated as:

$$C(sq_i, v) = \sum_{ep \in srcs(sq_i)} C(sq_i, v, ep)$$

The cardinality of a subquery $sq_i$, denoted as $C(sq_i)$, is the maximum cardinality of the subquery projected variables.

While the proposed cost model is simple, it provides accurate cardinality estimates. To measure estimation accuracy, we compared the estimated vs. actual cardinality of subqueries with more than one triple pattern using the q-error metric [110]. Let $a$ be the actual cardinality and $e$ be an estimate of $a$. The q-error is defined as $\max(e/a, a/e)$. Using LargeRDFBench queries [111], the median q-error of Lusail in our experiments is 1.09, close to the optimal value of 1.

### 4.4.2 Evaluation of Subqueries

Different orders of delayed subquery evaluation can result in different computation and communication costs. Our query planner tries to find an order of subqueries that has the minimum cost. Given a set of non-delayed subqueries, SAPE evaluates them concurrently and builds a hashmap that contains the bindings of each variable. As a result, SAPE knows the exact number of bindings of each subquery variable. Then, we refine the cardinality of the delayed subqueries based on the cardinality of variables they can join with. The first delayed subquery to be evaluated is the one with the lowest cardinality.

Once the first subquery is selected, it is evaluated at the corresponding endpoints and its results are used to update the bindings hashmap. SAPE continues to select the next subquery to be evaluated until all subqueries are executed. When executing a subquery with its variables bound to values from the bindings hashmap, SAPE groups values from the hashmap into blocks and submits a subquery for each block (as opposed to a subquery for each value).
Algorithm 6: Subqueries Evaluation

Algorithm 6 describes our selectivity-aware evaluation technique for subqueries. The input is a set of independent subqueries with their delay decisions. Each subquery contains its triple patterns, the relevant endpoints (sources), the projection variables, and whether the subquery is optional. The algorithm initializes the request handler which creates a thread per relevant endpoint (line 1). If there is only one subquery, i.e., the query is disjoint, the algorithm evaluates the whole query at all relevant endpoints independently (line 3). Then, it aggregates the results obtained from relevant endpoints, joins the partial results from different endpoints, if necessary, and returns the final query answer (line 4).

If the query is not disjoint, SAPE iterates over all input subqueries and evaluates each subquery at its relevant endpoints (lines 6-19). In the first phase, non-delayed subqueries are evaluated and their results are collected concurrently (lines 6-7). This step is non-blocking, i.e., each thread is assigned all relevant subqueries at the same time.

Whenever possible, the results of non-delayed subqueries are joined together. This reduces the number of found bindings used in delayed subqueries. In the second phase,
SAPE evaluates the delayed subqueries using the found bindings from the first phase (lines 10-18). SAPE selects the next delayed subquery to be the one with the smallest estimated cardinality (line 11). SAPE formulates a set of modified subqueries from the subquery itself using the found bindings (line 12). It appends a data block to the subquery using the SPARQL VALUES construct, which allows multiple values to be specified in the data block. If the subquery contains triple patterns of the form ⟨?s, ?p, ?o⟩, the source selection process is repeated using the found bindings to reduce the number of relevant endpoints (line 13). Without this refinement, such subqueries are relevant to all endpoints.

We empirically verified that the source selection refinement step on irrelevant endpoints using ASK queries costs significantly less than evaluating the delayed subquery with the found bindings. Finally, the bound subqueries are evaluated and their results are merged (lines 15-16). SAPE updates the set of found bindings using the current subquery results (line 17). After that, the evaluated subquery is removed from the delayed subqueries list (line 18). SAPE continues to evaluate the other subqueries until no more delayed subqueries are left.

**Join Evaluation.** Each endpoint thread maintains a set of relevant subqueries and their corresponding results. This information is encapsulated in the request handler object which is then passed to the threads performing the joins (line 19). Each subquery corresponds to a relation (R) for which we know the true cardinality and is partitioned among a set of threads. The join evaluation algorithm has four main steps: (i) For each subquery, it collects aggregate statistics (relation size and number of partitions) from all threads. (ii) It then uses a cost-based query optimizer based on the Dynamic Programming (DP) enumeration algorithm [112]. The DP algorithm starts with a join tree of size 1, i.e., a single relation, where the join cost is initially zero. It then builds larger join trees by considering the rest of the relations, pruning expensive partial plans as early as possible. At each DP step, SAPE joins
the current subplan with another relation \((R)\) leading to a new state \(S'\) with cost: 
\[
Cost(S') = \min(Cost(S'), Cost(S) + \text{JoinCost}(S, R)).
\]
Since the expanded state \(S'\) can be reached using different orders, we associate each state with the minimum cost found. Using an in-memory hash join algorithm, joining the subplan at state \(S\) with another relation \(R\) has two phases; hashing and probing. Assuming that \(S\) is the smaller relation, the join cost is estimated as follows:
\[
\text{JoinCost}(S, R) = \frac{1}{S.\text{threads}} |S| + \frac{1}{R.\text{threads}} C(R, v)
\]
All threads with the smaller relation build a hash table for their part of \(S\). The threads that maintain \(R\) evaluate the join by probing these hash tables with the found bindings of the join variables. (iii) Given the devised join order, SAPE joins the different subqueries together to produce the query answer. (iv) Finally, SAPE aggregates the joined results from the individual threads and returns the result.

4.5 Experimental Study

4.5.1 Evaluation Setup

**Systems Compared.** We evaluate Lusail\(^2\) against one index-free system, FedEx [39], and two index-based systems, SPLENDID [40] and HiBISCuS [41]. [54] has shown that FedEx outperformed other systems on the majority of queries and datasets. HiBISCuS [41] is an add-on to improve performance; we use it on top of FedEx. SPLENDID showed competitive performance to FedEx on several queries in [54] and LargeRDFBench\(^3\). Similarly to Lusail, both FedEx and SPLENDID support multi-threaded.

**Computing Infrastructure.** We used two settings for our experiments: two local

\(^2\)https://github.com/Lusail/lusail

\(^3\)https://github.com/AKSW/LargeRDFBench
clustering, 84-cores and 480-cores, and the public cloud. The 84-cores cluster is a Linux cluster of 21 machines, each with 4 cores and 16GB RAM, connected by 1Gbps Ethernet. The 480-cores cluster is a Linux cluster of 20 machines, each with 24 cores and 148GB RAM, connected by 10Gbps Ethernet. We use the 84-cores cluster in all experiments except those that need 256 endpoints for the LUBM dataset. For the public cloud, we use 18 virtual machines on the Azure cloud to form a real federation.

Datasets. We use several real and synthetic datasets. Table 4.1 shows their statistics. QFed [99] is a federated benchmark of four different real datasets. Although the total number of triples used in QFed is only 1.2 million, there are interlinks between the four datasets, which makes federated query evaluation challenging. LargeRDFBench is a recent federated benchmark of 13 different real datasets with more than 1 billion triples in total. We also used the synthetic LUBM benchmark to generate data for 256 universities, each with around 138K triples. It includes links between the different universities through students and professors.

Queries. QFed [99] has different categories of queries. Each query has a label $C$ followed by the number of entities for each class, and a label $P$ followed by the number of predicates linking different datasets. LUBM comes with its benchmark queries. We

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Endpoint</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>QFed</td>
<td>DailyMed</td>
<td>164,276</td>
</tr>
<tr>
<td></td>
<td>Diseasome</td>
<td>91,182</td>
</tr>
<tr>
<td></td>
<td>DrugBank</td>
<td>766,920</td>
</tr>
<tr>
<td></td>
<td>Sider</td>
<td>193,249</td>
</tr>
<tr>
<td></td>
<td><strong>Total Triples</strong></td>
<td><strong>1,215,627</strong></td>
</tr>
<tr>
<td>LargeRDFBench</td>
<td>LinkedTCGA-M</td>
<td>415,030,327</td>
</tr>
<tr>
<td></td>
<td>LinkedTCGA-E</td>
<td>344,576,146</td>
</tr>
<tr>
<td></td>
<td>LinkedTCGA-A</td>
<td>35,329,868</td>
</tr>
<tr>
<td></td>
<td>ChEBI</td>
<td>4,772,706</td>
</tr>
<tr>
<td></td>
<td>DBPedia-Subset</td>
<td>42,849,609</td>
</tr>
<tr>
<td></td>
<td>DrugBank</td>
<td>517,023</td>
</tr>
<tr>
<td></td>
<td>Geo Names</td>
<td>107,950,085</td>
</tr>
<tr>
<td></td>
<td>Jamendo</td>
<td>1,049,647</td>
</tr>
<tr>
<td></td>
<td>KEGG</td>
<td>1,090,830</td>
</tr>
<tr>
<td></td>
<td>Linked MDB</td>
<td>6,147,996</td>
</tr>
<tr>
<td></td>
<td>New York Times</td>
<td>335,198</td>
</tr>
<tr>
<td></td>
<td>Semantic Web Dog Food</td>
<td>103,595</td>
</tr>
<tr>
<td></td>
<td>Affymetrix</td>
<td>44,207,146</td>
</tr>
<tr>
<td></td>
<td><strong>Total Triples</strong></td>
<td><strong>1,003,960,176</strong></td>
</tr>
<tr>
<td>LUBM</td>
<td>256 Universities</td>
<td>35,306,161</td>
</tr>
</tbody>
</table>

Table 4.1: Datasets used in experiments.
only used the queries that access multiple endpoints. Queries $Q_1$, $Q_2$, and $Q_3$ in our experiments correspond to $Q_2$, $Q_9$, and $Q_{13}$ in the benchmark while $Q_4$ is a variation of $Q_9$; it retrieves extra information from remote universities. LargeRDFBench has three categories: simple $S$, complex $C$, and large (big) $B$. LargeRDFBench subsumes the FedBench benchmark [113]. The complex category contains 10 queries with a high number of triple patterns and advanced SPARQL clauses. The large category has 8 queries with large intermediate results.

**Endpoints.** We use Jena Fuseki 1.1.1 as the SPARQL engine at the endpoints for LUBM and QFed. Since Jena runs out of memory while indexing the endpoints of LargeRDFBench, we used a Virtuoso 7.1 instance for each of the 13 endpoints in LargeRDFBench. The standard, unmodified installation of each SPARQL engine is run at the endpoints and used by all federated systems in our experiments.

**Data Preprocessing Cost.** Index-based systems such as SPLENDID and HiBIS-CuS require a preprocessing phase that generates summaries about the data schemas and collects statistics that are used during query optimization. In real applications, endpoints might not allow collecting these statistics. Moreover, it is a time consuming process dominated by the dataset size. For example, SPLENDID needs 25 and 3,513 seconds to pre-process QFed and LargeRDFBench, respectively. In contrast, Lusail and FedX do not require any preprocessing. Hence, index-free methods are preferred in a large scale and dynamic environment, since endpoints can join and leave the federation at no cost.

In the rest of this section, we present the results of our evaluation on a local cluster and on a geo-distributed settings, in Sections 4.5.2 and 4.5.3, respectively. We analyze the different costs of Lusail’s query processing and its sensitivity to the threshold for delayed queries in Section 4.5.4.

In all subsequent experiments, all systems are allowed to cache the results of source selection. Each query is run three times and we report the average of the last two.
Figure 4.8: QFed Benchmark: Queries with Filter have high selectivity while Big literal queries have bigger intermediate data.

We set a time limit of one hour per query before aborting.

4.5.2 Lusail on a Local Cluster

We compare Lusail to FedX, HiBISCuS, and SPLENDID. They are all deployed on one machine of the 84-cores cluster. The endpoints are also deployed on the same cluster.

QFed Dataset. Figure 4.8 shows the query performance of Lusail compared to FedX and HiBISCuS. SPLENDID timed out in all QFed queries except $C2P2$ which is answered in 56 seconds. Lusail achieves better performance than FedX and HiBISCuS for all queries. Queries with filter, namely $C2P2BF$, $C2P2BOF$, $C2P2F$ and $C2P2OF$, have high selectivity, i.e., less intermediate data. Hence, most of these queries are answered within a few seconds. Lusail is up to six times faster than other systems for these queries. Using big literal object ($C2P2B$, $C2P2BO$) increases the volume of communicated data. Hence, FedX and HiBISCuS timed out after one hour in $C2P2BO$, while FedX took significant time to evaluate $C2P2B$, on which HiBISCuS timed out. This is due to the large size of communicated data and the number of remote requests. Lusail successfully answers both queries in less than 2 seconds.
LUBM Dataset. This experiment utilizes up to four university datasets\(^4\) from the LUBM benchmark, each in a different endpoint. Figures 4.9(a) and 4.9(b) show the results using two and four endpoints, respectively. The datasets at the endpoints have the same schema. Therefore, FedX and HiBISCuS cannot create exclusive groups. Instead, a subquery is created per triple pattern and is sent to all endpoints. Bound joins are then formulated using all the results retrieved from the different endpoints. This leads to a huge number of remote requests. Lusail utilizes the schema as well as the location of data instances accessed by the query to formulate the subqueries. Thus, Lusail discovered that both $Q1$ and $Q2$ are disjoint queries and their final results can be formulated by sending the whole query to each endpoint independently.

$Q3$ and $Q4$ need to join data from different endpoints. $Q3$ finds graduate students who received their undergraduate degree from university0. This limits the size of intermediate data and the number of endpoints. FedX and HiBISCuS do not utilize such filtering so they sent the query to all endpoints. Lusail decomposed the query into two subqueries: the first subquery (students who obtained an undergraduate degree from university0) is sent to the relevant endpoint. The second subquery contains

\(^4\)The other systems do not scale beyond four endpoints while Lusail scales to 256 endpoints (Figures 4.12(b) and 4.12(c)).
only \( \langle ?x, \text{rdf:type, ub:GraduateStudent} \rangle \), which is relevant to all endpoints. Hence, Lusail decided to delay its evaluation and managed to outperform the other systems on four endpoints. Lusail decomposed \( Q_4 \) into two subqueries, with the second subquery delayed until the results of the first subquery are ready. The figures illustrate that Lusail is up to three orders of magnitude faster than FedX and HiBISCuS for queries \( Q1, Q2, \) and \( Q4 \). FedX and HiBISCuS ran out of memory for \( Q1 \) on four endpoints. SPLENDID managed to run only \( Q3 \) on four endpoints and took 52 seconds, significantly slower than all other systems, so it is not included in the figures.

**LargeRDFBench Dataset.** Figure 4.10 shows the response times of the different systems on the LargeRDFBench queries. The performance of Lusail and FedX is comparable for most of the simple queries. The preprocessing performed by the index-based systems, HiBISCuS and SPLENDID, sometimes results in better performance on the simple queries, but not always. For example, HiBISCuS is much slower than Lusail for \( S13 \) and \( S14 \), and SPLENDID has the worst performance in \( S6, S7, S9, \) and \( S14 \). Lusail is the fastest system for \( S13 \) and \( S14 \) since these two queries return relatively large intermediate results. However, Lusail is generally not faster than the index-based systems on the simple queries since they do not generate
large intermediate results and they access datasets with different schemas, so Lusail’s optimizations do not improve performance.

The complex and large (or big) queries have a larger number of triple patterns per query, on average, and access a larger amount of intermediate data. Lusail achieves significantly better performance than other systems for most of the complex queries (Figure 4.10). $C5$ contains two disjoint subgraphs joined by a filter variable, a query not supported by Lusail’s competitors. Both FedX and HiBISCuS could not finish on $C1$ and $C9$ within an hour. SPLENDID evaluated only 5 out of the 10 complex queries. $C2$ is a selective query returning 4 results, which explains why all systems have comparable performance. FedX achieved the best performance for $C4$ followed by Lusail, while HiBISCuS could not evaluate the query within one hour. $C4$ contains a LIMIT clause of 50 results. Lusail’s current implementation uses a simple approach for the LIMIT clause. It computes all the final results and returns only the top 50 results. FedX cuts short the query execution once the first 50 results are obtained, so it outperformed Lusail on $C4$. SPLENDID achieved the best performance only on $C6$, and other systems have comparable performance on this query.

Lusail is superior for all large queries. These queries generate large intermediate results, which explains the high response time of Lusail. Similar to $C5$, $B5$ and $B6$ contain two disjoint subgraphs joined by a filter variable, which is not supported by systems other than Lusail. For the remaining queries, FedX and HiBISCuS timed out on two queries and returned no results on another two. SPLENDID succeeded only on $B2$ and timed out on the rest.

**Summary.** Lusail is the only system that successfully executes all queries of LargeRDFBench, often showing orders of magnitude better performance than other systems. In contrast, the other systems time out or fail to execute on some queries, in addition to their performance being highly variable and unpredictable.
4.5.3 Lusail in a Geo-Distributed Setting

In this section, we evaluate Lusail by simulating a real scenario on the cloud as well as using real endpoints.

Using the MS Azure cloud. We create a real geo-distributed setting by deploying SPARQL endpoints in 7 regions of the Azure cloud in the US and Europe. We used 17 D4 Azure VMs (8 Cores, 28 GB memory), 13 for the LargeRDFBench endpoints and four for the LUBM and QFed endpoints, interchangeably. Lusail and its competitors are deployed on one D5.V2 instance (16 Cores, 56 GB memory) in Central US. None of the 17 VMs is located in Central US.

Figure 4.11: Geo-distributed federation: endpoints are deployed in 7 different regions of the Azure cloud. Communication cost affects all systems, but Lusail can execute all queries and outperforms other systems.
The communication cost imposed a clear overhead. For QFed, neither FedX nor HiBISCuS were able to evaluate most of the queries. FedX finished only C2P2BF in 23 seconds, compared to 1.9 seconds for Lusail, while HiBISCuS finished only C2P2 in 4,477 seconds, compared to 9.5 seconds for Lusail. Figures 4.11(a) and 4.11(b) show the query response times of both complex and large queries on LargeRDFBench. We omit the simple queries since they exhibit the same behavior. The high communication overhead affected the runtime of all systems. For complex queries, FedX timed out on two queries and gave runtime errors on two others. HiBISCuS timed out on three queries but did reasonably well in the rest. SPLENDID was able to evaluate only five out of the ten complex queries. Lusail outperformed all other systems in all complex queries, in some cases by up to two orders of magnitude (C1 and C9). Large queries show the same behavior. Lusail is the only system that returns results, with no time out or runtime errors.

Figure 4.11(c) shows results on two endpoints of the LUBM dataset. Lusail’s query response times increased slightly compared to the local cluster (Figure 4.9(a)). All queries finished in around 1 second. In contrast, both FedX and HiBISCuS required more than 1,000 seconds; an order of magnitude degradation compared to the local cluster. This shows their sensitivity to the communication overhead since they tend to communicate large volumes of data. With four endpoints, FedX and HiBISCuS were able to evaluate only Q3 and ran out of memory or timed out on the rest.

**Real Endpoints.** In this experiment, we use Lusail and FedX to query real independently deployed endpoints. Specifically, we use the Bio2RDF endpoints and a subset of the LargeRDFBench endpoints. We extracted five representative queries from the Bio2RDF query log: R1, R2, R3, R4, and R5 (queries shown in the appendix). For LargeRDFBench, we evaluated six queries: S3, S4, S7, S10, S14, and C9. We use a single machine of the 84-cores cluster to run Lusail and FedX. We show

---

5 http://bio2rdf.org/
6 http://manager.costfed.aksw.org/costfed-web
Table 4.2: Query runtimes (sec) on real endpoints. ZR: zero results error, RE: runtime exception.

<table>
<thead>
<tr>
<th></th>
<th>Bio2RDF</th>
<th>LargeRDFBench</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>Lusail</td>
<td>12.3</td>
<td>8.1</td>
</tr>
<tr>
<td>FedX</td>
<td>128.1</td>
<td>721.5</td>
</tr>
</tbody>
</table>

the results in Table 4.2. For $S_3$ and $S_4$, which are simple and selective queries, FedX outperforms Lusail, as it does when running on a local cluster (Figure 4.10). FedX was unable to execute four of the other queries, and is one or two orders of magnitude slower than Lusail on the queries that it does execute. This demonstrates that Lusail is capable of answering queries accessing real independently deployed endpoints with good performance.

4.5.4 Analyzing Lusail

4.5.4.1 Profiling Lusail

Lusail has three phases: source selection, query analysis using LADE, and query execution using SAPE. In this experiment, we profile these phases while varying the query complexity and data size. We use LargeRDFBench queries with different complexities, simple ($S_{10}$), complex ($C_4$), and large ($B_1$). Lusail is deployed on a single machine of the 84-cores cluster. The results are shown in Figure 4.12(a).

Source selection and query analysis require a small amount of time compared to query execution, especially for $C_4$ and $B_1$. As expected, the total response time is dominated by the query execution phase. Lusail’s query analysis phase is lightweight, requiring less time than the source selection phase in $S_{10}$ and $C_4$. $B_1$ requires performing a union operation between two sets of triple patterns and retrieves its data from the endpoints with the largest data sizes. Hence, the query analysis phase takes slightly more time than the source selection phase. In all cases, query analysis does not add significant overhead.

The cost of query processing in Lusail also depends on the number of endpoints
Figure 4.12: Profiling Lusail by varying query complexity, the number of endpoints, and the data size.
and the sizes of the datasets. Therefore, we profiled Lusail while varying the number of endpoints, which also increases the data size. LUBM allows us to increase both endpoints and data size in a systematic way by adding more universities. We deployed 256 university endpoints on the 480-cores cluster. Lusail is deployed on one machine in the same cluster.

Figures 4.12(b) and 4.12(c) show the time required for each phase of Q3 and Q4, respectively. Both queries join data from different endpoints to produce the final result. Lusail’s query analysis is lightweight, especially for Q3 since it has only two triple patterns. For Q3, Lusail detects the GJVs using the source selection information, i.e., it does not need to communicate with the endpoints. Source selection time is substantial for these queries and increases slightly as the number of endpoints increases. Query execution time is the dominant factor as the number of endpoints increases. The figures show the total query response time with and without caching the results of ASK and check queries. The cache helps, especially for the more complex Q4 and when the number of endpoints is large.

4.5.4.2 Delayed Subqueries

This experiment evaluates different threshold values for identifying subqueries to delay, namely $\mu$, $\mu + \sigma$, and $\mu + 2\sigma$, in addition to delaying only subqueries with outlier estimated cardinalities. We used the Chauvenet criterion [108] for outlier detection. In this experiment, we use our LargeRDFBench deployment in Microsoft Azure. Figure 4.13 reports the total time for evaluating the queries of each category in LargeRDFBench. For simple and complex queries, $\mu + 2\sigma$ and Outliers allowed most subqueries to be evaluated concurrently and delayed only a few of them. Hence, these thresholds missed the opportunity to delay some subqueries that could reduce the communication cost and the cost of joining the fetched data. Thus, $\mu + 2\sigma$ and Outliers performed significantly worse than $\mu$ and $\mu + \sigma$, as seen in the figure. For
large queries, delaying too many subqueries limits parallelism. Thus, $\mu$ performed significantly worse than others since too few subqueries were evaluated concurrently while the rest were delayed. As shown, $\mu + \sigma$ consistently performs well in all the three categories and hence we use it in our system.

4.5.4.3 Delayed Subqueries Detection Accuracy

We analyzed LADE to measure how many true local subqueries were mistakenly identified as global subqueries. For the LargeRDFBench benchmark, we found that the fraction of local subqueries identified as global subqueries is only 33%. Some of these queries are detected as global because of very generic RDF predicates that are present in most endpoints and hard to avoid, for example, \texttt{owl:sameAs}, \texttt{purl:title}, and \texttt{rdfs:label}. If we exclude these predicates, then this fraction drops to 16%. For LUBM queries, none of the local subqueries was mistakenly identified as a global subquery.

4.5.4.4 Effect of LADE and SAPE

This experiment measures the gain obtained through LADE and SAPE compared to FedX as a baseline. FedX and Lusail are each deployed on a single machine of the 84-cores cluster. We only report results for two queries from each benchmark. However,
Figure 4.14: Lusail optimizations can move computations on intermediate data to the endpoints, retrieve less irrelevant data, and achieve orders of magnitudes speedup compared to FedX. X corresponds to time out.
we observed similar behavior in most of the queries with medium and high complexity. Figures 4.14(a), 4.14(b), and 4.14(c) show the peak memory usage, communication volume (intermediate data to be shipped), and total response time of each query, respectively. FedEx takes a significant amount of time for query execution due to its static query decomposition and bound join evaluation. It could not process three queries out of six within the time limit of one hour (C2P2O, C9 and B1). In these three queries, FedEx did not consume a lot of memory or send a lot of data over the network, but rather it overwhelmed the endpoints with lots of requests and wasted the whole hour waiting for them to finish.

LADE decomposition shifts the computation on the intermediate data from Lusail to the endpoints. Therefore, for the queries that FedEx was able to complete, Lusail with LADE alone (without SAPE) consumes significantly less memory and communication compared to FedEx. Lusail outperforms FedEx by up to three orders of magnitude in terms of query response time. Using the SAPE execution in addition to the LADE decomposition further improved the query response time of Lusail. When enabling SAPE, the delayed queries affect the memory and communication costs, sometimes positively and sometimes negatively. However, communication is performed in parallel using multiple threads. Thus, the net effect is that SAPE with LADE always improve on the query response time compared to LADE alone.
4.6 Conclusion

Lusail optimizes federated SPARQL query processing through a locality-aware decomposition (LADE) at compile time followed by selectivity-aware and parallel query execution (SAPE) at run time. The LADE decomposition is based not only on the schema but also on the actual location of data instances satisfying the query triple patterns. This decomposition increases parallelism and minimizes the retrieval of unnecessary data. SAPE query execution orders queries at run time by delaying subqueries expected to return large results, and chooses join orders that achieve a high degree of parallelism. Lusail outperforms state-of-the-art systems by orders of magnitude and scales to more than 250 endpoints with data sizes up to billions of triples. As future work, we plan to investigate keyword search as a means for querying federated RDF systems, and to develop methods for returning fast and early results during federated query execution. Both extensions aim to facilitate interactive data discovery and exploration on linked web data.
Chapter 5

Towards Data Discovery on Web Linked Data

5.1 Introduction

Linked data is leading to the emergence of a web of data through URIs and RDF links that interlink various data across the web. The amount of linked data available on the web is increasing in an unprecedented way. Currently, linked web data consists of more than 150 billion triples in more than 3,400 datasets\(^1\) in domains, such as media, government, and life sciences. In life sciences, examples include NeuroCommons\(^2\) that develops knowledge base of annotations to the biomedical abstracts using RDF and Bio2RDF\(^3\) that has about 12B triples across 35 datasets. Moreover, there is a growing public government data such as data.gov\(^4\). Yet another recent example is Wikidata that acts as a structured RDF data for Wikipedia\(^5\). A growing list of SPARQL endpoints is maintained by W3C\(^6\). This linked web data could be queried using SPARQL to discover useful patterns, infer new facts, and integrate pieces of information into meaningful wholes. Deriving such insights has led to the emergence of data discovery as an important component for linked data applications.

The RDF model allows interlinking entities from different datasets, which are independently maintained and accessed via a SPARQL endpoint. The linked web data represents a massive geo-distributed graph. These geographically dispersed endpoints

\(^1\)http://stats.lod2.eu/
\(^2\)http://sciencecommons.org/projects/data/
\(^3\)http://bio2rdf.org/
\(^4\)http://semantic.data.gov/sparql
\(^5\)https://www.wikidata.org
\(^6\)https://www.w3.org/wiki/SparqlEndpoints
SELECT ?pres ?party ?page WHERE {
}

Figure 5.1: A SPARQL query to find US presidents and associated news articles in the linked web data.

are deployed on machines of different computing capacities, use different RDF engines and may employ different optimization techniques, and handle data of different sizes. This heterogeneity makes enabling discovery on linked web data a very challenging problem. The problem is further compounded by the need to have prior knowledge to be able to formulate SPARQL queries, e.g., URIs and ontology schema as shown in Figure 5.1. These two challenges demand the following: (i) ability to query linked data using simple keywords (ii) a heterogeneity-aware federated RDF engine for querying large-scale graphs across geo-distributed endpoints, and (iii) support for early and fast results over heterogeneous settings.

Forming structured SPARQL queries requires prior knowledge about the schema of each individual dataset as well as the exact URIs of the entities and predicates. For example, to form the query in Figure 5.1, users need to (i) be aware of the exact structure of the predicates and related entities, e.g., <http://dbpedia.org/ontology/party> and <http://dbpedia.org/resource/United_States>, (ii) identify how different predicates and URIs of the same endpoint are connected together to form a subgraph, e.g., the two triple patterns in lines 2-3 are connected through the same subject, and (iii) identify how endpoints are interlinked, e.g., using owl:sameAs in line 4 to interlink DBPedia and NYTimes endpoints. Thus, forming well-structured SPARQL queries is quite challenging.

Keyword-based querying provides a more intuitive way for data discovery. Existing approaches [114, 115, 116] requires downloading the whole data into a centralized
place to create data summaries [115, 116] or graph embeddings [114]. Downloading the huge amount of linked data into a centralized storage is not feasible; it would require a lot of disk space or data owners may now allow it in the first place. Furthermore, some of the utilized preprocessing techniques; e.g. graph embedding [117, 118], are very computationally demanding [118] and can not be easily applied to massive amounts of data.

The query execution paradigm followed by federated RDF systems, such as Lusail [59, 58] and FedX [98], is coarse-grained; it does not consider the delay in fetching a partial result from a specific endpoint. These systems optimize for reducing the number of HTTP requests and communication cost. Therefore, they suffer a significant performance degradation with full heterogeneity. Moreover, these systems use blocking algorithms, i.e., the query results are reported at once at the end of the query evaluation. Thus, the user has to wait till the end of the process to get any result from these systems. This may take a long time due to different reasons, such as network delay, complex queries, or busy endpoints. Users usually prefer to get query results progressively. This has been addressed in different database systems, such as [119, 120]. However, it is challenging for federated RDF query processing as there is neither prior knowledge nor statistics about the datasets.

We developed novel techniques on top of Lusail [59, 58] to (i) predict semantically equivalent SPARQL queries from a set of keywords and (ii) execute these queries based on fine-grained and non-blocking query plans to get fast and early results. Our main idea for predicting a SPARQL query from a set of keywords is to find the top-K URIs representing entities, in RDF Triples, that semantically match the keywords. We introduce a model to estimate the semantic affinity between the given keywords and the set of entities and predicates associated with them. Our semantic affinity model leverages a new semantic similarity metric based on word embeddings [121]. This model is used to detect for each entity the predicates with high semantic affinity
to the given keywords. We build a star query for each entity and its set of elected predicates. Finally, these star queries are assembled to build one or more SPARQL queries.

In this chapter, we show our initial efforts to support data discovery on linked web data and the progressive production of results. Section 5.2 outlines the overall architectural of the Lusail discovery system. In Section 5.3, we highlight the main ideas of our techniques for predicting SPARQL queries and progressive query execution. Section 5.4 shows our initial experimental evaluation while Section 5.5 concludes.

5.2 System Architecture

The architecture for our linked data discovery system is shown in Figure 5.2. Lusail gets as input a list of keywords or a SPARQL query. The list of keywords is analyzed by our query predictor to generate semantically related SPARQL queries. The key idea is to map these keywords to a set of top-K URIs that represent entities, i.e., subjects or objects, in the linked data graph. For each entity, the SPARQL query predictor fetches the predicates associated with the entity. Then, Lusail leverages word embeddings to filter this list of predicates and build SPARQL queries that are semantically related to the given set of keywords.
Lusail’s query decomposer breaks down a SPARQL query into subqueries based on their locality information. The Query Executer takes as an input a set of subqueries. It has two main modes, namely coarse-grained and fine-grained execution. In coarse-grained mode, Lusail optimizes the execution to get the whole result at once. Therefore, the coarse-grained planner works at the level of each subquery to calculate its cardinality in order to maximize the degree of parallelism while minimizing the communication cost. This mode is good for batch or offline processing of analytical queries, i.e., no need to progressive result generation, when there is little degree of heterogeneity, e.g., machines in the same data center. In contrast, the fine-grained planner optimizes the execution plan to report partial results as early as possible. Therefore, the granularity of the planning phase is an HTTP request, i.e., a subquery sent to a specific endpoint. Lusail first estimates the time to evaluate each request and decides to delay those requests expected to take significantly long time to be evaluated. In a nutshell, it favors those joinable requests which when evaluated first will generate the earliest possible query answer.

5.3 Discovery Support on Lusail

5.3.1 Keywords to SPARQL Queries

The main constraint to work at the scale of linked web data, i.e., not a single local RDF dataset, is to avoid expensive or infeasible preprocessing phases to build graph summaries, indexes, or training a graph embeddings model [114, 115, 116]. As shown in Figure 5.3, our key idea for predicting SPARQL queries from a list of keywords is to identify star-shaped subgraphs, i.e., a set of RDF triples with different properties/predicates sharing the same entity; i.e., subject or object. Therefore, our main goal is to create star queries by identifying entities and predicates that are semantically related to the given keywords. These star queries are then assembled together to create a set of possible SPARQL queries. We show in Algorithm 7 the steps required
Input: Keyword Query (KQ), Set of endpoints (Endpoints)
Result: List of possible SPARQL queries (Qs)

1. \( Qs \leftarrow \emptyset \);
2. \( KQ' \leftarrow \text{preprocess} (KQ) \);
3. \( KWS \leftarrow \text{annotate} (KQ') \);
4. \( \text{StarQueries} \leftarrow \emptyset \);
5. foreach \( kw \) in \( KWS \) do
   6. if \( \text{isNamedEntity}(kw) == \text{false} \) then
      continue;
   7. foreach \( endpt \) in \( \text{Endpoints} \) do
      8. \( \text{entURIs} \leftarrow \text{getRelevantEntityURIs}(kw) \);
      9. \( \text{sortedEntURIs} \leftarrow \text{SortBySemanticSim}(kw, \text{entURIs}) \);
     10. \( \text{topEntURIs} \leftarrow \text{getTopURIs}(\text{sortedEntURIs}, k) \);
      11. foreach \( URI \) in \( \text{topEntURIs} \) do
          12. \( \text{predURIs} \leftarrow \text{getPredURIs}(URI) \);
          13. \( \text{sortedPredURIs} \leftarrow \text{SortBySemanticSim}(KWs, \text{predURIs}) \);
          14. \( \text{topPredURIs} \leftarrow \text{getTopURIs}(\text{sortedPredURIs}, k) \);
          15. \( \text{SQ} \leftarrow \text{formStarQuery}(URI, \text{topPredURIs}) \);
          16. \( \text{SQ.numMatchedKeywords} \leftarrow \text{calcMatchedKeywordsByURIs}(\text{SQ}, \text{KWs}) \);
          17. if \( \text{SQ.numMatchedKeywords} == \text{KWs.size}() \) then
              18. \( Qs.\text{insert}(\text{SQ}); \)
          else
              19. \( \text{StarQueries}[\text{endpt}].\text{addStarQuery}(\text{SQ}, kw) \);
      20. \( Qs \leftarrow \text{mergeStarQueries}(\text{StarQueries}) \);
     21. return \( Qs \);

Algorithm 7: Keyword to SPARQL Querying

Figure 5.3: Lusail pipeline for predicting a set of SPARQL queries from a list of keywords.

to map keywords into a set of possible SPARQL queries \( Qs \) which is initially empty (line 1). Our current implementation handles only entity-based keyword queries.

**Keyword Tagging:** Initially, We start by tagging the input keywords and decomposing them into subsets using different NLP techniques, such as part of speech tagging and named entity recognition (lines 2-3). For example, a list of keywords like "news article and party of Barack Obama" will be decomposed into \{\{news article\}, party, \{Barack Obama\}\} and tagged as \{noun, noun, entity\}, respectively. To do so, we rely on an off-the-shelf NLP tools \[122, 123, 124\] for Part Of Speech (POS) tagging and Named Entity Recognition (NER). Our current implementation
uses Stanford POS and NER tagger [125] for this task.

**Subqueries Builder:** In the beginning, a light SPARQL query is formulated for each entity and sent to all available endpoints to get $K$ URIs, where each URI represents a subject or object with a syntactic match, i.e., contains the entity (lines 5-9). The fetched URIs are ranked based on their semantic affinity to the given entity (line 10). We elect from the fetched URIs the ones exceeding a certain threshold of semantic affinity (line 11). Similarly, we retrieve the list of predicates associated with each elected URI (line 12). Accordingly, each entity URI with its surrounding predicates that has high semantic affinity to the given keywords form a star query (line 16). If the formed star query matches all keywords, we add it to the list of possible matched queries (line 19). Otherwise, we insert it into an intermediate data structure that collect all formed star queries from all endpoints using various keywords.

**Query Graph Assembly:** The result from the first phase (lines 5-21) is a set of star queries that match one or more input entities. The second phase of the mapping algorithm tries to merge these individual star queries to form larger queries that match all keywords (line 22). To do so, we implement a recursive approach that start with connecting two star queries together by adding a triple pattern that connects entities from both star queries. If the connected query produced an answer, then it is expanded further by considering more star queries and so on. Checking whether a connected query has an answer or not is done by consulting Lusail federated query evaluation with a ”limit 1” constraint.

**Semantic Affinity Model:** We developed a semantic affinity model based on word embeddings (WEs) [121]. Our model uses WEs with 300 dimensions built with GloVe on 840B (https://nlp.stanford.edu/projects/glove/) tokens from web crawl data. WEs determine the semantic similarity between a pair of words only.
QID: 1
select * where {
}

QID: 2
select * where {
}

Figure 5.4: Predicated SPARQL queries to find ”news article and party of Barack Obama”.

Our semantic affinity model extracts a set of words from the fetched URIs. The model estimates the semantic similarity between two sets of words by combining their WEs. We implemented different methods to combine WEs, such as mean, max, min and addition.

As an example, the subject entity http://dbpedia.org/resource/Barack_Obama in dbpedia.org has several predicates, such as ”dbpedia:party” and ”dbpedia:successor”. The predicate ”dbpedia:successor” has a semantic affinity of 0.3 and 0.6 with the keywords ”news article” and ”party”, respectively while ”dbpedia:party” has a similarity of 1.0 to the keyword ”party”. This entity also exists in NYTimes endpoint as object and is connected to predicates ”nyt:topicPage”, ”nyt:associated_article_count”, and ”nyt:first_use”. Our semantic affinity model identifies that ”news article” keyword and these predicates are semantically similar with ratios, 0.81, 0.8, and 0.6 respectively. Based on this information, a star query is constructed for each entity and identified predicates. Finally, our method assembles the star queries to suggest the possible SPARQL queries. Figure 5.4 shows two generated SPARQL queries for the above list of keywords where the threshold used in our semantic affinity model was 0.75.
5.3.2 Fine-grained Planner

To account for the different heterogeneity factors in real-life endpoints deployments, our fine-grained planner does the following:

**Fine-grained Cost Estimation:** Instead of estimating the cost of executing a subquery as a whole, our planner estimates the cost of each request (a subquery to be evaluated at one endpoint). This is achieved by collecting lightweight per-endpoint statistics that encode (i) the communication latency which also implicitly captures the efficiency of the underlying endpoint RDF engine. (ii) Accuracy of per-triple statistics: Lusail’s cost model is based on a lightweight per-triple statistics. The retrieved statistics for a triple pattern with two constants is a fairly accurate approximation. As the number of variables in a triple pattern increases, per-triple statistics approach fails significantly. Our main goal is to have a minimal cost estimation overhead while providing an accurate statistics estimation.

Our fine-grained cost model modified Lusail’s cost function (see Section 4.4.1) as follows. The cardinality of a variable $v$ in a subquery $sq_i$ is denoted as $C(sq_i, v)$ and represents the number of bindings of $v$. If two triple patterns $TP_i$ and $TP_j$ join on a variable $v$, then the number of bindings of $v$ at endpoint $ep_k$ after the join will be:

$$C(sq_i, v, ep_k) = \min(\delta_{TP_i} \cdot C(TP_i, ep_k), \delta_{TP_j} \cdot C(TP_j, ep_k))$$

where $\delta_{TP_i}$ and $\delta_{TP_j}$ denote the statistics accuracy of the two triple patterns $TP_i$ and $TP_j$, respectively. We use the minimum cardinality of the predicates, in which $v$ is a common variable, as an upper bound of the cardinality of $v$ per endpoint. Thus, the estimated time for retrieving the bindings of $v$ in the subquery $sq_i$ is the sum of its cardinalities in all the relevant endpoints $ep$ multiplied by latency of each endpoint. This is estimated as:

$$T(sq_i, v) = \sum_{ep \in \text{arcs}(sq_i)} \lambda(ep) \cdot C(sq_i, v, ep)$$
where $\lambda(ep)$ is the latency of the endpoint $ep$ estimated at the source selection phase. Therefore, we estimate subquery $sq_i$ evaluation time, denoted as $T(sq_i)$, as the maximum retrieval time of the subquery projected variables.

**Early-Results Aware Query Execution:** Unlike the coarse-grained planner which decides whether to delay a subquery or not based on its estimated cardinality, our planner estimates the time required to evaluate each request and delays the requests expected to take significantly long time for evaluation. Therefore, our non-blocking query execution module generates query plans that report results as soon as they become available. Non-delayed requests are evaluated by the corresponding threads independently while a delayed request is paired with a request at the same level such that it keeps pulling from the other thread result buffer and evaluates its subquery using bound joins. During evaluation, the thread keeps receiving the results in an iterative manner.

### 5.3.3 Multi-Query Optimization

Lusail generates a set of possible queries for the given keywords. To evaluate these queries, Lusail allows **multi-query optimization** (MQO) on which the system evaluates all possible matched queries together as a single workload. For a batch of $N$ queries, Lusail analyzes each query individually and decomposes it into a set of subqueries (see Section 4.3). Then, Lusail detects the common subqueries among all sets of subqueries. In different queries, if two or more subqueries share the same predicates and the same number of variables, then they form a common subquery. It maintains the list of common subqueries across the $N$ queries. Then, Lusail estimates the cost of these subqueries and decides delayed and non-delayed subqueries. Notice that, the delay decision tend to be more accurate as the number of subqueries increases which ensures larger population and better estimation of the mean and standard deviation.

In MQO mode, Lusail first evaluates the common non-delayed subqueries. Then,
it iterates over all queries to produce the final result. There are three possibilities for the query evaluation: (i) when the query contains only non-delayed subqueries, Lusail joins their computed results and reports the final answer. (ii) When the query contains both delayed and non-delayed subqueries, Lusail uses its adaptive technique (APEX) that utilizes bound joins for evaluating delayed subqueries. Finally, (iii) if all the subqueries are delayed, Lusail detects the subquery with the minimal cost, change its status to non-delayed and utilizes APEX to find the query results.

5.4 Evaluation

In this section, we evaluate our proposed data discovery modules. First, we evaluate how accurate and efficient our keyword to SPARQL module. Then, we evaluate the performance of reporting fast and early results compared to blocking query execution modules. Finally, we test the effectiveness of our multi-query optimization technique.

5.4.1 Keyword to SPARQL Queries

To the best of our knowledge, we are unaware of any system that supports keyword search over such a large-scale of decentralized linked data with multiple interconnected datasets (endpoints). However, there are multiple systems [126, 127, 114, 115] that allow users to evaluate keyword queries over a single endpoint; e.g. DBpedia. Therefore, we evaluate the performance of our approach by conducting two experiments, one utilizing a well known benchmark (QALD) to query a single dataset and the other for keyword queries that integrate information from multiple datasets.

**QALD Benchmark:** Question Answering over Linked Data (QALD) is a standard evaluation benchmark for question answering over Linked Data. QALD provides multilingual question answering over DBpedia (http://dbpedia.org/sparql/) RDF data. It provides a set of questions available in different languages (we focus on English only). Each question is associated with a set of keywords, corresponding
Table 5.1: QALD6 Benchmark

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lusail-KeywordSearch</td>
<td>0.54</td>
<td>0.88</td>
<td>0.58</td>
<td>182 sec</td>
</tr>
<tr>
<td>HAWK</td>
<td>0.2</td>
<td>0.78</td>
<td>0.17</td>
<td>612 sec</td>
</tr>
</tbody>
</table>

SPARQL queries and their reference answers. There are two sets of questions; training and testing. Since Lusail does not require any kind of training or bootstrapping, we focus only on the test set. In particular, we used 39 entity search test questions (see Appendix A.3.1). We compare the performance of Lusail against HAWK [126]. Hawk is a recent question answering system over linked data. It is a hybrid question answering framework that combines entity search over linked data and textual data from the Web. The search input is a question expressed in natural language, which passes through an eight-step pipeline including entity annotation, dependency parsing, semantic annotation and pruning.

Table 5.1 shows the performance of Lusail vs. HAWK using the same DBpedia dataset provided by its authors. We show the accuracy as well as the runtime taken by each system for solving the 39 test questions (shown in Appendix A.3.1). We report the precision, recall and f-score metrics for each system which is calculated by comparing the answers provided by each system to the golden answers provided by the benchmark. HAWK generates several possible SPARQL queries for each keyword query considering different variations of the detected nouns/entities. It returns a lot of false positives (non-relevant answers) which explains its high recall; however HAWK achieves low precision and f-score. Our results show that Lusail-KeywordSearch provides a better performance than HAWK in terms of both accuracy and runtime. Neither Lusail-KeywordSearch nor HAWK could answer all the 39 queries which describes the low f-score values in Table 5.1. In particular, Lusail-KeywordSearch could generate correct results for 22 queries while HAWK could only answer 17 queries.

**Question Answering over Multiple Interlinked Datasets:** Since there is no
available benchmarks that provide keyword queries that integrate data from multiple endpoints, we generate a set of queries based on LargeRDFBench benchmark. To do so, we translated a set of queries from QFed and LargeRDFBench to their keyword counterparts. The queries are: Q1: what are the news articles about Barack Obama and his party, Q2: list all drugs that affect humans and other mammals, Q3: Which are targets of Nafcillin?, Q4: What is the chemical formula of Amcinonide?, and Q5: What is the associated genes of breast cancer?. These queries retrieve data from DBpedia, NYTimes, Sider, DailyMed, Diseasome and DrugBank. We show in Table 5.2 the performance of Lusail for each one of the aforementioned queries. We run Q1 on the 13 LargeRDFBench endpoints while the rest of the queries are evaluated against the 4 QFed endpoints. Due to the large amount of data in LargeRDFBench (~1 billion triples), Lusail required more time to generate the SPARQL queries for Q1. However, it needs only few seconds for Q2-Q5, either for keywprd to sparql or for executing the generated queries as a workload. Finally, Lusail managed relevant queries to all the keyword queries. We show in Appendix A.3.2 the generated SPARQL queries for each keyword query.

5.4.2 Fast and Early Results

A linked data query integrates partial results from endpoints that differs in their hardware specs, dataset sizes, networks speeds, and RDF engines with varying performance. Therefore, some endpoints can be significantly faster than others. To show how fast early results can be returned, we report here some of our experimental
results.

We used 14 geo-distributed VMs over 7 different Azure data centers in the USA and Europe, with a total of 256 endpoints. These instances host the 13 LargeRDFBench endpoints while Lusail is deployed on a D5.V2 instance (16 cores, 56 GB memory) in Central USA, while none of the instances is located in Central USA. LargeRDFBench is a recent federated benchmark of 13 interlinked real datasets pertaining to different domains including DBpedia (wikipedia infoboxes), LinkedMDB (movie database), GeoNames (geographical data), Drugbank and Cancer Genome Atlas. We also use different RDF engines, such as Virtuoso, Stardog, Jena, Blazegraph, and Allegro as the SPARQL engines.

We used queries from LargeRDFBench and the above settings, i.e., different machine specs, different deployment regions and utilizing different RDF engines. Figure 5.5 shows the performance of Lusail with coarse-grained mode in full homogeneous and heterogeneous settings, i.e., first and fourth column, plus the Lusail performance with fine-grained mode, where the second column reports the full time and the third column reports the time of producing the first 1000 results. As the figure shows, our Heterogeneity-Aware and Parallel Execution (HAPE) mode provides better performance compared to the coarse-grained mode of Lusail (SAPE). Furthermore, it also
enables Lusail to report the 1000 results significantly faster which allows for more efficient exploratory based discovery modules.

5.4.3 Multi-Query Optimizations

We conduct an experiment to evaluate the effectiveness of our MQO add-on feature. We use the LUBM workload in [30]. This workload consists of 10K unique queries which are derived from the 14 LUBM benchmarks by changing their structures and constants. We generate workloads of different sizes by randomly choosing queries from the pool of the 10K unique queries. We generated workloads of 10, 20, 40, 80 and 160 queries. Figure 5.6 shows the effect of the MQO of Lusail compared to evaluating queries independently. As the workload size increases, the possibility to find similar substructures between queries increases. Therefore, utilizing the shared computation among queries results in performance gains that range from 34% to 46%.

5.5 Conclusion

Lusail discovery support allows users with no prior knowledge to explore linked web data and extract information using a list of keywords, instead of writing a structured
SPARQL query. Lusail introduces a semantic affinity model that leverages word embeddings to estimate the semantic similarity between the keywords and few samples of the linked data to predict the semantically equivalent SPARQL queries. One main advantage of our semantic affinity prediction is that it does not need any prior knowledge of the datasets, i.e., no need to perform graph summarization nor building graph embedding models. Lusail also provide a fine-grained execution module that masks the high skew among the endpoints in terms of time performance, and reports the results progressively.
Chapter 6

Concluding Remarks

The versatility of the Resource Description Framework (RDF) has allowed many web services to publish very large datasets that are impractical to process on a single machine. Therefore, many distributed and federated SPARQL engines have been proposed. This thesis aims at advancing the state-of-the-art techniques to enable scalable data discovery and analytics over linked Web data. We conclude in the next section with a summary of our contributions. Then, we present an outlook on possible future research directions.

6.1 Summary of Contributions

This thesis started by presenting the first experimental study of 22 state-of-the-art distributed RDF systems. We first provide a survey that cover the entire spectrum of distributed/federated RDF data processing and categorize them by several characteristics. Then, we perform extensive experimental evaluation using large-scale synthetic and real datasets. In an attempt to standardize the evaluation of future systems, we publish online all datasets, our evaluation methodology and links to the different systems. Based on our experimental evaluation, we identify a set of research problems towards optimizing the state-of-the-art RDF systems.

First, we proposed Spartex; an RDF analytics framework that bridges the gap between specialized RDF stores and generic graph engines. Spartex introduces an extension to SPARQL that allows the invocation of user-defined vertex-centric pro-
grams as stored procedures. Therefore, it provides a unified query interface for both types of computations to be executed declaratively. Graph algorithms and SPARQL operators can be pipelined without the need to load and offload the data for each algorithm. Spartex is also coupled with a novel vertex-centric SPARQL operator for solving ad-hoc SPARQL queries efficiently. We demonstrate through various scenarios that Spartex simplifies significantly the implementation of rich RDF analytics programs.

Second, we proposed Lusail; a scalable system for querying linked data. Lusail introduces a locality-aware decomposition method that dramatically reduces the number of remote requests and allows for better utilization of the endpoints. It proposes a novel cost model which is used to order subqueries at runtime based on their selectivity. Lusail delays subqueries expected to return large results, and chooses the subqueries join order that achieves a high degree of parallelism. This leads to a parallel execution that adaptively balances between remote requests and local computations. Our experimental evaluation shows that Lusail outperforms state-of-the-art systems by orders of magnitude and scales to more than 250 endpoints with data sizes up to billions of triples.

Finally, we provided a set of module on top and Lusail towards supporting data discovery and analytics on decentralized linked data. Enabling discovery on linked web data is a very challenging problem due to the prior knowledge required to formulate SPARQL queries and the heterogeneous settings in which query execution should be optimized. To addresses the above challenges, we developed novel techniques to (i) predict semantically equivalent SPARQL queries from a set of keywords by leveraging word embeddings, and (ii) execute these queries based on fine-grained and non-blocking query plans to get fast and early results in heterogeneous settings. Our experiments show that these modules are capable of providing promising performance in terms of query accuracy and overall runtimes without the need for infeasible
expensive preprocessing.

6.2 Future Research Directions

There are several future directions for Spartex. One direction is to consider the opportunity for cross-algorithms optimizations that may be possible when combining SPARQL and graph analytics in the same query. For example, most of the execution time is spent on analytical algorithms. Therefore, multiple query optimization techniques can be employed to evaluate multiple algorithms concurrently. Notice that analytical queries may consist of a pipeline of operators. Hence, pipelines of multiple queries need to be aligned in a way that minimizes the overall execution time.

The limited navigational capabilities of SPARQL led the W3C to introduce property paths [71] in SPARQL 1.1. Property paths are regular expressions which retrieve pair of graph nodes connected to each others given certain path constraints. Arenas et al. [128] studied the semantics and complexities of property paths and showed that its implementation is very challenging. Therefore, another future direction is to extend Spartex query evaluation techniques to evaluate property path queries. We believe that evaluating these queries at the vertex granularity may provide good performance that does not result in an explosion in the intermediate data size.

Our work towards enabling data discovery can be extended to enable generic keyword queries that does not necessarily contain named entities. Also, we are currently using off-the-shelf natural language processing tools for annotating input keywords. These tools provide an acceptable performance; however they are not designed for processing short and sometimes incomplete sentences or even a set of keywords. Therefore, we believe that a future research direction is to revisit these tools to adjust it as necessary to work better for keyword search interfaces.

There are also other future directions that are based on our experimental evaluation of the current state-of-the-art distributed RDF systems. First, emerging hard-
ware architectures nowadays are equipped with many CPU cores within each single machine, which is under utilized by current RDF systems. Distributed RDF systems need to fully utilize the large number of available cores, enhance cache efficiency through smart data layouts and minimize their memory footprint by avoiding duplicate information within the same worker. Second, most distributed systems do not support updates, adding new triples would require several systems to rebuild their indices, which is very expensive for large-scale graphs. Third, no partitioning algorithm suits all RDF graphs and all query types. The complexity of sophisticated partitioning schemes does not allow distributed RDF systems to process very large graphs in a timely manner. At the same time, they do not always guarantee better performance compared to lightweight partitioning. We believe that focusing on the problem of matching partitioning strategies to RDF graphs, instead of using a single partitioning strategy for all, is a promising research direction.
REFERENCES


“SPARQL Query Language for RDF,” https://www.w3.org/TR/rdf-sparql-query/.


[98] 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 Proc. of ISWC, 2011.


[125] “Stanford Named Entity Recognition (NER) and Information Extraction (IE),” https://nlp.stanford.edu/ner/.


APPENDICES
A.1 Spartex Appendix

A.1.1 Implementation Details

We show below the compute function for PageRank in GPS, and hence Spartex, for generic graphs. The inputs are the set of incoming messages to the current vertex (if any) and the current iteration (superstep) number. In the first iteration, all vertices are assigned rank value equal to $1/N$, where $N$ is the number of vertices (line 5-7).

Then, each vertex sends its value to all its neighbours. In the following iterations, each vertex aggregates the incoming messages, calculates the new PageRank value (lines 11 -16), sends it to all neighbours (lines 18-20) and updates the current vertex state (line 22). After reaching the maximum number of iterations, vertices vote to halt.

```java
public void compute(Iterable<DoubleWritable> incomingMessages, int superstepNo) {
    int numVertices = ((IntSumGlobalObject) getGlobalObjectsMap().getGlobalObject(
        GlobalObjectsMap.NUM_TOTAL_VERTICES))
        .getValue().getValue();
    if (superstepNo == 1) {
        setValue(new DoubleWritable((double) 1 / (double) numVertices));
        sendMessages(getNeighborIds(), getValue());
        return;
    }
    double sum = 0.0;
    for (DoubleWritable messageValue : incomingMessages) {
        sum += messageValue.getValue();
    }
    double currentState = 0.85 * sum/getNeighborIds().length + 0.15 / (double) numVertices;
    int[] neighborIds = getNeighborIds();
    DoubleWritable messageValue = new DoubleWritable(currentState);
    sendMessages(neighborIds, messageValue);
    setValue(new DoubleWritable(currentState));
    if (superstepNo == numMaxIterations) {
```
The same compute function can be executed in Spartex on the RDF data without any changes. To allow for this flexibility, we modified the underlying data-access module of GPS. The main changes are: (i) prior to calling the compute function on a vertex, Spartex checks whether the vertex should be skipped, by evaluating the given user filters. (ii) To materialize the results, Spartex stores the newly computed vertex value in its miniature properties store right after evaluating the compute function. (iii) For label-based and label-oblivious neighbour access, we changed the data access procedures accordingly. For example, the function getNeighborIds() in lines 7 and 18 is modified to return the set of incoming and outgoing neighbours regardless of the predicate. We also enriched GPS data-access module to allow for retrieving only incoming (or outgoing) neighbours with or without label constraints. Finally, (iv) before returning the set of edges/neighbours, Spartex applies any edge filters.

A.1.2 Queries

We use the standard 7 LUBM queries defined in [77] and used by most distributed RDF systems [45, 25, 21, 23]. Additionally, we define two more complex queries to test the systems rigorously. The two queries are defined below. For YAGO and Bio2RDF, we use the same queries used by AdPart [45].

```sql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

```
?x ub:memberOf ?z .
?x rdf:type ub:GraduateStudent .
?y rdf:type ub:University .
?x ub:undergraduateDegreeFrom ?y .
?x ub:advisor ?t .
?t ub:worksFor ?z .}
A.2 Lusail Queries

A.2.1 Bio2RDF Endpoints and Queries


PREFIX dbank: <http://bio2rdf.org/drugbank_vocabulary:>
PREFIX mgi:<http://bio2rdf.org/mgi_vocabulary>
PREFIX hgnc:<http://bio2rdf.org/hgnc_vocabulary>
PREFIX pgkb:<http://bio2rdf.org/pharmgkb_vocabulary>
PREFIX omim:<http://bio2rdf.org/omim_vocabulary>
PREFIX sider: <http://bio2rdf.org/sider_vocabulary:>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX kegg: <http://bio2rdf.org/kegg_vocabulary:>

R1: SELECT ?drug ?hgnc ?model WHERE {
  ?model mgi:allele ?all .
  FILTER(?drug = <http://bio2rdf.org/drugbank:DB00619>)
}

R2: SELECT ?gene ?ref ?cterm where{
}


  ?pheno rdf:type omim:Phenotype .
}
A.2.2 The Drug Query Based on QFed

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX dosage: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/dosageforms/>
PREFIX sider: <http://www4.wiwiss.fu-berlin.de/sider/resource/sider/>
PREFIX dailymed: <http://www4.wiwiss.fu-berlin.de/dailymed/resource/dailymed/>

SELECT * WHERE {
  ?drug drugbank:brandName ?brand_name .
}
OPTIONAL {
  ?siderdrug owl:sameAs ?drug .
}

A.2.3 LUBM Queries

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

Q1: SELECT ?X ?Y ?Z WHERE{
  ?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:University .
  ?Z rdf:type ub:Department .
  ?X ub:undergraduateDegreeFrom ?Y .
}

Q2: SELECT ?X ?Y ?Z WHERE{
  ?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:AssociateProfessor .
  ?Z rdf:type ub:GraduateCourse .
}

Q3: SELECT ?X WHERE{
  ?X rdf:type ub:GraduateStudent .
  ?X ub:undergraduateDegreeFrom <www.University0.edu> .
}

  ?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:AssociateProfessor .
  ?Z rdf:type ub:GraduateCourse .
}
}
A.3 Keyword Queries

A.3.1 Keyword Queries over DBpedia

We show below the list of 39 QALD6 entity-based keyword queries which we used to evaluate Lusail and its competitors.

<table>
<thead>
<tr>
<th>QID</th>
<th>keyword queries</th>
<th>NL question sentences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>doctoral supervisor, Albert Einstein</td>
<td>Who was the doctoral supervisor of Albert Einstein?</td>
</tr>
<tr>
<td>3</td>
<td>write, song, Hotel California</td>
<td>Who wrote the song Hotel California?</td>
</tr>
<tr>
<td>8</td>
<td>how many movies, directed, Park Chan-wook</td>
<td>How many movies did Park Chan-wook direct?</td>
</tr>
<tr>
<td>9</td>
<td>developers, DBpedia</td>
<td>Who are the developers of DBpedia?</td>
</tr>
<tr>
<td>11</td>
<td>school name, Obama’s wife, study</td>
<td>What is the name of the school where Obama’s wife studied?</td>
</tr>
<tr>
<td>13</td>
<td>capital, Cameroon</td>
<td>What is the capital of Cameroon?</td>
</tr>
<tr>
<td>14</td>
<td>Boston Tea Party; take place</td>
<td>When did the Boston Tea Party take place?</td>
</tr>
<tr>
<td>15</td>
<td>play, Gus Fring, Breaking Bad</td>
<td>Who played Gus Fring in Breaking Bad?</td>
</tr>
<tr>
<td>16</td>
<td>wrote, Harry Potter</td>
<td>Who wrote Harry Potter?</td>
</tr>
<tr>
<td>17</td>
<td>actors, Big Bang Theory</td>
<td>Which actors play in Big Bang Theory?</td>
</tr>
<tr>
<td>21</td>
<td>president, Eritrea</td>
<td>Who is the president of Eritrea?</td>
</tr>
<tr>
<td>23</td>
<td>create, Family Guy</td>
<td>Who created Family Guy?</td>
</tr>
<tr>
<td>24</td>
<td>How many people, Poland</td>
<td>How many people live in Poland?</td>
</tr>
<tr>
<td>26</td>
<td>voice, Bart Simpson</td>
<td>Who does the voice of Bart Simpson?</td>
</tr>
<tr>
<td>27</td>
<td>compose, soundtrack, Cameron’s Titanic</td>
<td>Who composed the soundtrack for Cameron’s Titanic?</td>
</tr>
<tr>
<td>28</td>
<td>Boris Becker, end career</td>
<td>When did Boris Becker end his active career?</td>
</tr>
<tr>
<td>30</td>
<td>country, Sitemore</td>
<td>What country is Sitemore from?</td>
</tr>
<tr>
<td>31</td>
<td>country, born, Bill Gates</td>
<td>Which country was Bill Gates born in?</td>
</tr>
<tr>
<td>33</td>
<td>city, Nikos Kazantzakis, die</td>
<td>In which city did Nikos Kazantzakis die?</td>
</tr>
<tr>
<td>35</td>
<td>film, Stanley Kubrick, direct</td>
<td>Which films did Stanley Kubrick direct?</td>
</tr>
<tr>
<td>37</td>
<td>How many seats, stadium of FC Porto</td>
<td>How many seats does the home stadium of FC Porto have?</td>
</tr>
<tr>
<td>40</td>
<td>city, headquarters of the United Nations</td>
<td>In which city are the headquarters of the United Nations?</td>
</tr>
<tr>
<td>41</td>
<td>city, born, president of Montenegro</td>
<td>In which city was the president of Montenegro born?</td>
</tr>
<tr>
<td>43</td>
<td>mayor, Paris</td>
<td>Who is the mayor of Paris?</td>
</tr>
<tr>
<td>47</td>
<td>date, princess Diana, died</td>
<td>When did princess Diana die?</td>
</tr>
<tr>
<td>54</td>
<td>where, Syngman Rhee, buried</td>
<td>Where is Syngman Rhee buried?</td>
</tr>
<tr>
<td>56</td>
<td>king, the Netherlands</td>
<td>Who is the king of the Netherlands?</td>
</tr>
<tr>
<td>61</td>
<td>where, Red Sox, play</td>
<td>Where do the Red Sox play?</td>
</tr>
<tr>
<td>62</td>
<td>time zone, Rome</td>
<td>In which time zone is Rome?</td>
</tr>
<tr>
<td>64</td>
<td>Lego Movie, cost</td>
<td>How much did the Lego Movie cost?</td>
</tr>
<tr>
<td>68</td>
<td>writer, Game of Thrones</td>
<td>Who wrote the Game of Thrones theme?</td>
</tr>
<tr>
<td>75</td>
<td>how, Michael Jackson, die</td>
<td>How did Michael Jackson die?</td>
</tr>
<tr>
<td>83</td>
<td>movies, Jesse Eisenberg</td>
<td>What movies does Jesse Eisenberg play in?</td>
</tr>
<tr>
<td>85</td>
<td>museums, London</td>
<td>Show me all museums in London.</td>
</tr>
<tr>
<td>92</td>
<td>city, live, Sylvester Stallone</td>
<td>In which city does Sylvester Stallone live?</td>
</tr>
<tr>
<td>93</td>
<td>person, inspiring, Vincent van Gogh</td>
<td>Who was Vincent van Gogh inspired by?</td>
</tr>
<tr>
<td>96</td>
<td>languages, Pakistan</td>
<td>What languages do they speak in Pakistan?</td>
</tr>
<tr>
<td>99</td>
<td>famous for, Elon Musk</td>
<td>What is Elon Musk famous for?</td>
</tr>
</tbody>
</table>

A.3.2 Keyword Queries over Multiple Interlinked Datasets

PREFIX dbpedia: <http://dbpedia.org/resource/>

PREFIX nyt: <http://data.nytimes.com/>

PREFIX dbank_drg: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>

PREFIX dbank_drgb: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>

PREFIX dailymed_drg: <http://www4.wiwiss.fu-berlin.de/dailymed/resource/drugs/>

PREFIX dailymed_res: <http://www4.wiwiss.fu-berlin.de/dailymed/resource/dailymed/>

PREFIX sider_drg: <http://www4.wiwiss.fu-berlin.de/sider/resource/drugs/>

PREFIX diseasome: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/>
Q1: What are the news articles about Barack Obama and his party?

```lang-sparql
PREFIX diseasome_dis: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseases/>

SELECT * WHERE {
  dbpedia:Barack_Obama <http://www.w3.org/2002/07/owl#sameAs> nyt:474521894807706853 .
  dbpedia:Barack_Obama <http://dbpedia.org/ontology/activeYearsStartDate> ?o38 .
  nyt:474521894807706853 <http://www.w3.org/2002/07/owl#sameAs> ?o16 .
}
```

Q2: List all drugs that affect humans and other mammals

```lang-sparql
PREFIX dbank_drg: <http://www.bioserv.org/resource/dbank_drg/>

SELECT * WHERE {
  dbank_drg:DB00333 dbank_drgb:possibleDiseaseTarget ?o438 .
  dbank_drg:DB00333 dbank_drgb:halfLife ?o441 .
  dbank_drg:DB00333 dbank_drgb:drugType ?o442 .
}
```

```lang-sparql
PREFIX dbank_drg: <http://www.bioserv.org/resource/dbank_drg/>

SELECT * WHERE {
  dbank_drg:DB02439 dbank_drgb:drugType ?o437 .
  dbank_drg:DB02439 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?o438 .
  dbank_drg:DB02439 dbank_drgb:molecularWeightAverage ?o440 .
  dbank_drg:DB02439 dbank_drgb:primaryAccessionNo ?o441 .
  dbank_drg:DB02439 dbank_drgb:dpdDrugIdNumber ?o442 .
  dbank_drg:DB02439 dbank_drgb:genericName ?o443 .
}
```

Q3: Which are targets of Nafcillin?
Q4: What is the chemical formula of Amcinonide?

```
select * where {
  dailymed_drg:2897 dailymed_res:activeIngredient ?o0 .
  dailymed_drg:2897 http://www.w3.org/1999/02/22-rdf-syntax-ns#type ?o1 .
  dailymed_drg:2897 <http://www.w3.org/2002/07/owl#sameAs> ?SelfVar .
  ?SelfVar <http://www.w3.org/2000/01/rdf-schema#seeAlso> ?o11 .
  filter (str(?SelfVar) = "http://www4.wiwiss.fu-berlin.de/sider/resource/drugs/2133")
}
```
select * where {
  diseasome_dis:173 diseasome:associatedGene ?o0 .
}

select * where {
  diseasome_dis:1669 diseasome:associatedGene ?o0 .
  diseasome_dis:1669 http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?o3 .
}

select * where {
  diseasome_dis:1671 diseasome:associatedGene ?o0 .
}

select * where {
  diseasome_dis:1666 diseasome:associatedGene ?o0 .
$$\text{diseasome\_dis:1666 diseasome\_diseaseSubtypeOf } \text{?o2}.$$ $$\text{diseasome\_dis:1666 } \text{<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> } \text{?o3}.$$ $$\text{diseasome\_dis:1666 diseasome\_name } \text{?o4}.$$$$\text{\}}$$