Indexing and Query Processing in RDF Quad-Stores

By

Jyoti Leeka

Under the Supervision of

Dr. Srikanta Bedathur

Indraprastha Institute of Information Technology Delhi

December, 2017

©Jyoti Leeka, 2017.

Indexing and Query Processing in RDF Quad-Stores

By

Jyoti Leeka

Submitted

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

to the

Indraprastha Institute of Information Technology Delhi

December, 2017

Certificate

This is to certify that the thesis titled - **"Indexing and Query Processing in RDF Quad-Stores"** being submitted by **Jyoti Leeka** to Indraprastha Institute of Information Technology, Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by her under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

Dr. Srikanta Bedathur

December, 2017

New Delhi

Abstract

RDF data management has received a lot of attention in the past decade due to the widespread growth of Semantic Web and Linked Open Data initiatives. RDF data is expressed in the form of triples (as Subject - Predicate - Object), with SPARQL used for querying it. Many novel database systems such as RDF-3X, TripleBit, etc. – store RDF in its native form or within traditional relational storage – have demonstrated their ability to scale to large volumes of RDF content. However, it is increasingly becoming obvious from the knowledge representation applications of RDF that it is equally important to integrate with RDF triples additional information such as source, time and place of occurrence, uncertainty, etc. Consider an RDF fact (BarackObama, isPresidentOf, UnitedStates). While this fact is useful for finding information regarding president of United States, it does not provide sufficient information for answering many challenging questions like what is the temporal validity of this fact?, where did this fact come from?, etc.

Annotations like confidence, geolocation, time, etc. can be modeled in RDF through a techniques called reification, which is also a W3C recommendations. Reification, retains the triple nature of RDF and associates annotations using blank nodes.

The focus of this thesis is on database aspects of storing and querying RDF graphs containing annotations like confidence, etc. on RDF triples. In this thesis, we start by developing an RDF database, named RQ-RDF-3X for efficiently querying these RDF graphs containing annotations over native RDF triples. Next, we noticed that more than 62% facts in real-world RDF datasets like YAGO, DBpedia, etc. have numerical object values. Suggesting the use of queries containing ORDER-BY clause on traditional graph pattern queries of SPARQL. State-of-the-art RDF processing systems such as Virtuoso, Jena, etc. handle such queries by first collecting the results and then sorting them in-memory based on the user-specified function, making them not very scalable. In order to efficiently retrieve results of top-k queries, i.e. queries returning the top-k results ordered by a user-defined scoring function, we developed a top-k query processing database named Quark-X. In Quark-X we propose indexing and query processing techniques for making top-k querying efficient.

Motivated by the importance of geo-spatial data in critical applications such as emergency response, transportation, agriculture etc. In addition to its widespread use in knowl-

edge bases such as YAGO, WikiData, LinkedGeoData, etc. We developed STREAK, a RDF data management system that is designed to support a wide-range of queries with spatial filters including complex joins, with top-k queries over spatially enriched databases. While developing STREAK we realized that to make effective use of this rich data, it is crucial to efficiently evaluate queries combining topological and spatial operators – e.g., overlap, distance, etc. – with traditional graph pattern queries of SPARQL. While there have been research efforts for efficient processing of spatial data in RDF/SPARQL, very little effort has gone into building systems that can handle both complex SPARQL queries as well as spatial filters.

We describe novel contributions of each of these engines developed below.

RQ-RDF-3X : RQ-RDF-3X presents extensions to triple-store style RDF storage engines to support reification and quads. In RQ-RDF-3X, we support triple annotations by assigning a unique identifier (R) to each (S, P, O) triple. Thus, the fundamental change required is to support an additional field (R) that has triple identifier. The inclusion of this additional field requires the query optimizer of the triple store being extended to be aware of the unique characteristic of the triple identifier (R). Additionally this requires careful re-thinking of existing indexing and query optimization approaches adopted by state-of-the-art triple stores. In order to achieve fast performance in RQ-RDF-3X we propose an efficient set of indices which enables RQ-RDF-3X to efficiently reduce the query processing time by making use of merge joins. The set of indices are stored compactly using an efficient compression scheme. We demonstrate experimentally that RQ-RDF-3X achieves one to two orders of magnitude speed-up over both commercial and academic engines such as Virtuoso, RDF-3X, and Jena-TDB on real-world datasets - YAGO and DBpedia.

Quark-X: Quark-X is an efficient top-k query processing framework for RDF quad stores. The contributions of Quark-X include novel in-memory synopsis indexes for predicates describing numerical objects. This is in the same spirit as building impact-layered indexes in information retrieval but carefully redesigned for use for ranking in reified RDF. Additionally, Quark-X proposes a novel Rank-Hash Join (RHJ) algorithm designed to utilize the synopsis indexes, by selectively performing range scans for facts containing numerical objects early on – this is crucial to the overall performance of SPARQL queries

which involve a large number of joins. We show experimentally that Quark-X achieves one to two magnitude speed-up over baseline databases namely Virtuoso, Jena-TDB, SPARQL-RANK and RDF-3X on YAGO and DBpedia datasets.

STREAK: STREAK is an efficient engine for processing top-k SPARQL queries with spatial filters. Spatial filters are used to evaluate distance relationships between entities in SPARQL queries. STREAK introduces various novel features such as a careful identifier encoding strategy for spatial and non-spatial entities for reducing storage cost and for early pruning, the use of a semantics-aware Quad-tree index that allows for early-termination and a clever use of adaptive query processing with zero plan-switch cost. For experimental evaluations, we focus on top-k distance join queries and demonstrate that STREAK outperforms popular spatial join algorithms as well as state of the art commercial systems such as Virtuoso.

Contents

1 Introduction **17**

1.1 Motivation . . . 17

1.2 Overview . . . 20

1.3 Contributions and Organization . . . 21

2 Background and Preliminaries **27**

2.1 Resource Description Framework . . . 27

2.2 SPARQL . . . 29

2.2.1 Motivational Queries . . . 30

2.3 Datasets . . . 33

2.4 Related Work . . . 34

2.5 RDF-3X . . . 39

2.5.1 Storage . . . 40

2.5.2 Query Processing . . . 40

3 RQ-RDF-3X: An Efficient Quad-Store **43**

3.1 Motivation . . . 43

3.2 Organization . . . 44

3.3 Related Work . . . 44

3.3.1 Reification Support . . . 45

3.3.2 N-Quads . . . 45

3.4 RQ-RDF-3X Framework . . . 46

3.4.1 Storage and Indexing in RQ-RDF-3X . . . 46

3.4.2 Selectivity Estimation 49
3.4.3 Query Translation and Optimization 52
3.5 Evaluation 53
3.5.1 Experimental Setup 53
3.5.2 Compared Systems 54
3.5.3 Benchmark Queries 55
3.5.4 Query Processing Performance 56
3.5.5 Analysis of Results 57
3.6 Discussion & Outlook 61
3.6.1 Outlook 61

4 Quark-X: An Efficient Top-k Processing Framework for RDF Quad Stores 63

4.1 Motivation 63
4.2 Organization 64
4.3 Preliminaries 65
4.3.1 Running Example 65
4.4 Related Work 66
4.5 Indexing for Quantitative Facts 68
4.5.1 Quantifiable Indexes 69
4.5.2 Semantic Encoding of Identifiers 70
4.6 Query Processing 73
4.6.1 S-index Join 74
4.6.2 Non-quantifiable Predicate Joins 75
4.7 Update Handling 80
4.8 Implementation Details 81
4.9 Evaluation Framework 82
4.9.1 Datasets 83
4.9.2 Benchmark Query Workloads 84
4.10 Experimental Results 85
4.10.1 Loading of Data and Database Size 85

4.10.2 Query Performance 87
4.10.3 Impact of Varying k 92
4.11 Discussion & Outlook 94
4.11.1 Outlook 94

5 STREAK: An Efficient Engine for Processing Top-k SPARQL Queries with Spatial Filters 97
5.1 Motivation 97
5.1.1 Challenge 98
5.1.2 Contributions 99
5.1.3 Organization 100
5.2 Preliminaries 100
5.2.1 Running Example 101
5.3 STREAK 102
5.3.1 S-QuadTree Index for Spatial Entities 103
5.3.2 Spatial Join Algorithm in STREAK 108
5.3.3 Adaptive Query Processing for Top-K Spatial Joins 114
5.4 Evaluation Framework 120
5.4.1 Datasets 120
5.4.2 Benchmark Query Workloads 121
5.5 Experimental Results 123
5.5.1 Performance of Spatial Join Processing in STREAK 123
5.5.2 Comparison with Database Engines 125
5.5.3 Comparison with varying k 128
5.6 Related Work 129
5.7 Discussion & Outlook 133
5.7.1 Outlook 133

6 Conclusions and Future Work 135
6.1 Future Work 136

A Queries **139**

A.1 RQ-RDF-3X Benchmark Queries 139

A.1.1 YAGO 139

A.1.2 DBpedia 150

A.2 Quark-X Benchmark Queries 157

A.2.1 YAGO 157

A.2.2 DBpedia 162

A.3 STREAK Benchmark Queries 166

A.3.1 YAGO 166

A.3.2 LGD 170

Bibliography **175**

List of Figures

1-1 LOD Cloud 18

1-2 RDF Statement 18

1-3 SPARQL Query in reified form 19

1-4 SPARQL Query in N-Quads form 20

2-1 SPARQL Query in N-Quad form 28

2-2 SPARQL Query in reified form 28

2-3 Example RDF Graph 29

2-4 Example SPARQL Query 31

2-5 Example Top-k Query 32

2-6 Example Top-k Spatial Distance Join Query 32

2-7 Operator Tree showing SIP 42

3-1 RQ-RDF-3X: Compression Scheme for B+-leaf entries 48

3-2 Running example in RQ-RDF-3X 50

3-3 RQ-RDF-3X: Reified Query Reformulation 50

3-4 Find the predecessors and successors of Governor of New York who have also won a prize 52

3-5 RQ-RDF-3X: Query Processing Time of Benchmark Queries excluding Dictionary build time 58

3-6 RQ-RDF-3X: Total Query Processing Time of Benchmark Queries 59

3-7 RQ-RDF-3X, PostgreSQL, RDF-3X performance in warm cache 60

4-1 Quark-X: Running example RDF dataset containing Quantitative facts . . . 65

4-2 Quark-X: Running example top-k query 65
4-3 Quark-X: Summarized Index Creation 69
4-4 Quark-X Query Processing 75
4-5 Quark-X: Cold-cache Query Processing Performance 88
4-6 Quark-X: Warm-cache Query Processing Performance using operating system's cache only 90
4-7 Quark-X: Warm-cache Query Processing Performance using engine's own cache 91
4-8 Quark-X: Performance over DBpedia for Varying k 92
4-9 Quark-X: Performance over YAGO for Varying k 93

5-1 STREAK: Example RDF knowledge graph 101
5-2 STREAK: Running example query 102
5-3 STREAK: Toy S-QuadTree 104
5-4 STREAK: Selecting optimal set of nodes on S-QuadTree 112
5-5 STREAK: Query Processing Flow-Chart 117
5-6 STREAK: Possible Plan Choices 118
5-7 STREAK: Effect of Sideways Information Passing 124
5-8 STREAK: S-QuadTree Vs. Sync. R-tree for Spatial Join 124
5-9 STREAK: Performance of STREAK Vs. PostgreSQL and Virtuoso 127
5-10 STREAK: Performance of STREAK Vs. PostgreSQL and Virtuoso 127
5-11 STREAK: Performance with Varying k 128
5-12 STREAK: Performance with Varying k 129

List of Tables

2.1 Horizontal Representation 36
2.2 Vertical Table 37

3.1 Summarized Characteristics of Benchmark Queries in RQ-RDF-3X 55
3.2 RQ-RDF-3X: Sizes of Datasets and Databases 56
3.3 Prevalence of Reified Entries in YAGO and DBpedia datasets 56

4.1 Quark-X: Sizes of Datasets and Databases 83
4.2 Quark-X: Characteristics of Benchmark Queries 86
4.3 Quark-X: Data Load Performance of Various Frameworks 86

5.1 STREAK: Dataset Characteristics 121
5.2 STREAK: Characteristics of Benchmark Queries 122
5.3 STREAK: On-Disk Database Size for YAGO and LGD 126

Chapter 1

Introduction

1.1 Motivation

The Resource Description Framework (RDF) data model is the common way of representing semantically linked data on the web and SPARQL is used for querying repositories of RDF data. Recent years have seen a tremendous growth in the size and variety of RDF data with the availability of semantic data from disciplines as varied as network sciences [39], biology [3], public administration [84], knowledge sharing [89], business intelligence [62], etc. Many of these datasets already contain close to billion facts and are growing rapidly. A big challenge posed in front of the data management community is how to access this big RDF data efficiently.

RDF is essentially a graph of entities, where entities and connecting links are identified by Uniform Resource Identifiers (URIs). The entities and connecting links can be searched for by dereferencing the URIs. RDF is one of the main components of Linked Data [18]. Wikipedia defines Linked Data, shown in figure 1-1, as "a term used to describe a recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF". Linked Open Data cloud is used for describing RDF datasets connected to each other through RDF links. Two entities linked in RDF can themselves be drawn from two different datasets of Linked Open Data cloud. This allows, for example, *places* in YAGO [89] dataset to be connected to geographical coordinates in Linked Geo Data (LGD) [11] dataset. Thus, linked data helps to connect

entities across datasets, that were not previously linked.

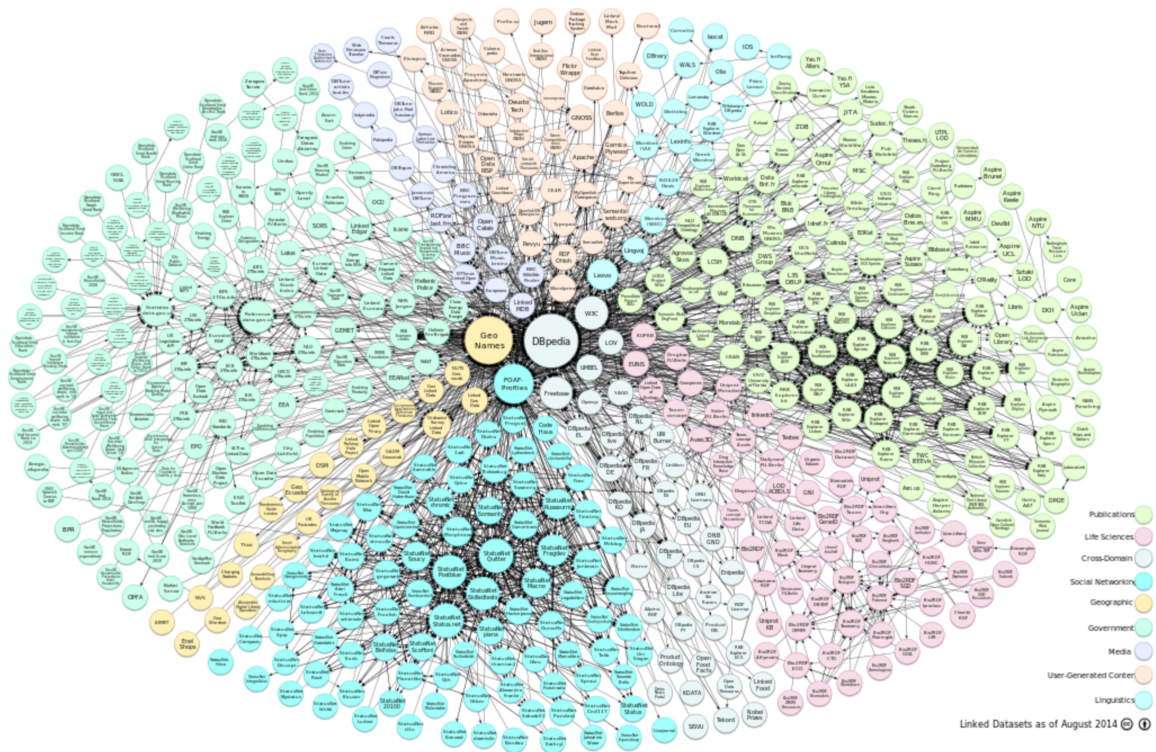


Figure 1-1: LOD Cloud

A fact is represented in RDF using a statement, which is at its heart, a triple representing a relationship between things – denoted by Subject(S) and Object(O) node – connected through a relationship edge denoted by Predicate (P), as shown in figure 1-2. The collection of triples taken together forms a RDF graph – visualized by representing subjects and objects as nodes in the graph with predicates as labeled directed edges connecting subject to its object. While these triples are useful for modeling dyadic relationships between entities, they are inadequate in their simple form for advanced modeling needs such as multi-entity relationships, provenance annotations for facts [21], uncertainty associated with automatically extracted facts [89], etc. The RDF standard offers *reification* [74] as a way to model these by enabling a RDF graph to act as a metadata description of other RDF triples.

Figure 1-2: RDF Statement

Reification [74] allows statements about other statements to be made using RDF's built-in vocabulary. In order to allow different ways of representing a statement, a statement can be reified multiple times. A reified statement is expressed in RDF as four statements. An RDF triple (S,P,O) with statement identifier I is expressed in reified form as:

```
<I>  rdf:type rdf:statement.
<I>  rdf:subject <S>.
<I>  rdf:predicate <P>.
<I>  rdf:object <O>.
```

Additional information e.g., name of the website from which the fact is extracted can be associated with the above RDF statement as:

```
<I>  <hasSource> <source>.
```

At the same time, another relatively simpler model called *N-Quads* [2] is also used quite frequently to associate a URI of RDF graph with a triple. Triples in N-Quads documents have an additional fourth column called context which is used for representing contextual information such as the name of the graph that the triple is part of in the dataset. N-Quads storing name of the graph in the context column are also known as Named Graphs [2].

An example signifying the importance of reification from YAGO [89] is given in Fig. 1-3:

```
_:id_57 rdf:type rdf:statement;
        rdf:subject <Ann_Richards>;
        rdf:predicate <holdsPoliticalPosition>;
        rdf:object <Governor_of_Texas>;
        <hasSuccessor> <George_W._Bush>;
        <hasPredecessor>  <Bill_Clements>;
        <occursSince> "1991-01-15";
        <occursUntil> "1995-01-17";
    <extractionSource> <http://en.wikipedia.org/wiki/Ann_Richards>.
```

Figure 1-3: SPARQL Query in reified form

Next we represent the same information as N-Quads below in Fig. 1-4:

The above RDF segment represents the fact that Ann Richards was the Governor of Texas from 1991-01-15 to 1995-01-17, her predecessor was Bill Clements and she was

```
<Ann_Richards> <holdsPoliticalPosition> <Governor_of_Texas> <id_57>.
<id_57> <hasSuccessor> <George_W._Bush> <id_58>.
<id_57> <hasPredecessor>  <Bill_Clements> <id_59>.
<id_57> <occursSince> "1991-01-15" <id_60>.
<id_57> <occursUntil> "1995-01-17" <id_61>.
<id_57> <extractionSource> <http://en.wikipedia.org/wiki/Ann_Richards> <
   id_62>.
```

Figure 1-4: SPARQL Query in N-Quads form

succeeded by George W. Bush. Note that modeling each of the triples in isolation would not convey the same information.

For querying the above RDF statements to find out the successor of Ann Richards when she was the Governor of Texas, SPARQL query can be written in reified form as:

```
SELECT ?a
WHERE {
    ?r rdf:subject <Ann_Richards>.
    ?r rdf:predicate <holdsPoliticalPosition>.
    ?r rdf:object <Governor_of_Texas>.
    ?r <hasSuccessor> ?a
}
```

1.2 Overview

In this thesis, we focus on efficiently storing RDF data modeled using reification and N-Quads. Reification and N-Quads allow an additional identifier to be associated with an entire RDF statement which can then be used to add further annotations. We structure our discussion around efficient storage of these RDF statements, and cover the following topics:

- SPARQL queries over reified RDF data: We develop an efficient quad-store RQ-RDF-3X that includes an efficient set of indexes and a smart query processor which generates an optimal join ordering for efficiently executing reified SPARQL queries.

- Top-k queries over RDF quads: We propose an efficient top-k query engine for RDF quads, which supports ranking queries over user-defined ranking functions contain-

ing statement-level confidence values in addition to other quantifiable values in the database. We develop novel indexing and query processing techniques for accelerating top-k querying.

- Top-k spatial distance join queries over RDF quads: We consider an important SPARQL query variant over spatially aware RDF databases (containing locations and geometries), namely the *Top-k Spatial Distance Join* (K-SDJ) query where the traditional top-k ranking with arbitrary ranking function on the (numerical) predicates in RDF data is over the results of a spatial distance join between geographical entities in the knowledge base. The resulting system, called STREAK introduces a number of novel features including a specialized spatial index structure suitable for K-SDJ queries over RDF quads, and query processing framework to support K-SDJ queries.

In the rest of the chapter we discuss the contributions and organization of the thesis.

1.3 Contributions and Organization

The thesis is structured around providing efficient ways to store and query graph-shaped RDF data. We make the below mentioned technical contributions:

- RQ-RDF-3X: We propose a generalized storage engine RQ-RDF-3X in Chapter 3, which can efficiently store and query additional information about triples like source, time, space, confidence, etc. A natural question to ask is: how do existing RDF stores store this additional information about triples. The state-of-the-art native stores such as RDF-3X [69] and Hexastore [101] are strictly designed for storing triples, and answering queries over them using SPARQL. As a result, they can incorporate only the standard reified models in their storage model. However, this severely degrades their query processing efficiency due to increased number of joins that need to performed in order to answer even the simplest of queries. The N-Quads model can be stored in these systems only after converting it to a reified form, and is thus equally inefficient to query. Though storage of meta-facts has been addressed by Wilkinson, et al. [103] and Alexander, et al. [6] but these techniques are extremely inefficient due to the

large number of self-joins required. In RQ-RDF-3X, we propose the following key features:

  - We implement the proposed framework within RDF-3X [69], a high performance RDF engine. We attach additional information to triples with fact id — which we assign to every fact. We present a description of the modifications made at the indexing and query processing stage of RDF-3X.
  - We devise an efficient set of indices which enable us to efficiently reduce the query processing time by making use of merge joins. The set of indices are stored compactly using an efficient compression scheme.
  - We propose a set of queries over annotated graphs which we have used for benchmarking the performance of the proposed engine RQ-RDF-3X. Our experimental results over YAGO [89] and DBpedia [43] show that our proposed approach is highly scalable and efficient in comparison to PostgreSQL [78] (which emulates Sesame style storage [27]), RDF-3X, Virtuoso [33] and Jena-TDB [53].

- Quark-X: Quark-X a top-k query engine for RDF quads. A straightforward way of storing RDF data is using the relational model which enables the use of top-k algorithms designed for relational databases used for RDF data. The property table technique [86, 60, 102] and vertically partitioned approach [4, 85] for storing RDF data are two such techniques which can draw advantage from top-k algorithms proposed for relational databases. However, many researchers have shown that these models of storing RDF in relational databases are not efficient in handling complex query patterns seen in SPARQL queries [101, 69, 70]. Simple triple-store model of storing RDF in relational tables is also not effective for top-k querying since: (a) self-joins incurred by top-k algorithms over a large table are bound to be expensive, (b) either of the two access methods viz., the sorted or the random access [51] are not suitable because of unsorted nature of quantifiable values in RDF and the complex pattern matching model of SPARQL queries.

Quark-X overcomes these limitations by introducing a combination of adaptively switching block-wise sorted and random accesses based on their cost estimates along with its semantic encoding of identifiers to improve locality of reference of subjects associated with quantifiable predicates. To the best of our knowledge, these features are not explored until now in relational top-k processing systems as well as RDF quad stores. We describe Quark-X in Chapter 4. Quark-X proposes the following key features:

- The use of compact in-memory synopsis indexes –called *S-indexes*– in addition to on-disk fine-grained indexes –called *Q-indexes*– for quantifiable predicates involved in user-defined ranking functions. This is in the same spirit as building impact-layered indexes in information retrieval [9, 10], but carefully redesigned for use for ranking in reified RDF.
- Intelligent reassignment of identifiers to RDF resources so that entities associated with similar predicates and reification structures are collocated on disk.
- We propose a novel Rank-Hash Join (RHJ) algorithm designed to utilize the synopsis indexes, by selectively performing range scans for quantifiable facts early on – this is crucial to the overall performance of SPARQL queries which involve a large number of joins.
- Processing of data in blocks whenever possible, which enables simultaneous processing of multiple buckets of S-indexes to quickly generate the top-k results or reach early-termination criterion.
- We evaluate Quark-X by comparing it with two state-of-the-art commercial RDF management systems – Jena-TDB [53], and Virtuoso 7.2 [93](a highly optimized RDBMS for storing RDF) as well as two academic RDF systems – SPARQL-RANK [64] and RDF-3X [69]. We also develop a query workload, which represents various usage patterns of SPARQL with ORDER-BY/LIMIT over reified RDF datasets. Our performance results demonstrate that Quark-X is significantly faster than comparable systems in both, cold as well as warm cache settings, while needing a very small memory-footprint during query pro-

cessing.

- STREAK: STREAK is an efficient top-k spatial distance join processing engine. We discuss STREAK in Chapter 5. A large-body of work exists in relational database research for integrating geo-spatial and top-k early termination operators within generic query processing framework. Therefore, a natural question to ask is whether those methods can be applied in a straightforward manner for RDF/SPARQL as well. Unfortunately, as we delineate below, the answer to this question is in the negative, primarily due to the schema-less nature of RDF data and the self-join heavy plans resulting from SPARQL queries.

  Consider applying one of the popular techniques, which build indexes over joining (spatial) tables during an offline/pre-processing phase [75]. However, in RDF data, there is only one large triples table, resulting in a single large spatial index that needs to be used during many self-joins making it extremely inefficient. On the other hand, storing RDF in a property-table form is also impractical due to a large number of property-tables – e.g., in a real-world dataset like BTC (Billion Triples Challenge) [28], there are 484, 586 logical tables [68].

  Next, the unsorted ordering (w.r.t. the scoring function) of spatial attributes during top-k processing prevents the straightforward application of methods that encode spatial entities to speed up spatial-joins [61]. This is because these approaches assume sorted order of spatial attributes, and hence are able to choose efficient merge joins as much as possible. Finally, state-of-the-art query optimization techniques for top-k queries [52] cannot be adopted as-is for spatial top-k since real-world spatial data does not follow uniform distribution assumptions that are used during query optimization.

  STREAK's key contributions are three-fold:

  – Central to the ability of STREAK to support efficient spatial distance joins is a novel schema-aware, in-memory index called S-QuadTree. It not only stores the spatial intersections but also the *soft-schema*, in the form of characteristic

sets, of spatial entities in a compact manner. As we show experimentally, S-QuadTree outperforms current spatial indexes designed for spatial distance join by one to two orders of magnitude due to its ability to filter out entities.

– A new spatial distance join algorithm, which exploits the information stored in S-QuadTree to smartly traverse the index structure to balance the I/O and CPU costs during join processing.

– An adaptive query plan generation algorithm called *Spatial AQP*, that switches the driver-driving plans based on the varying join statistics of spatial attributes in different data blocks as well as the join cost reductions possible through the early-termination feature of top-k algorithms.

We experimentally evaluate our STREAK framework by making use of two large-scale real-world spatially aware RDF datasets – *viz.,* YAGO [47] and LinkedGeo-Data [11]. Due to the lack of well-established benchmark queries for K-SDJ queries in RDF/SPARQL setting, we developed our own benchmark queries over these two datasets which reflect some of the queries from GeoCLEF. We compare our STREAK framework against PostgreSQL with spatial indexing and Virtuoso, a state-of-the-art RDF/SPARQL system. Our experimental results show that STREAK is able to outperform these systems by one to two orders of magnitude in both cold and warm cache settings.

Chapter 2

Background and Preliminaries

RDF data model is used for representing semantic web data. Some classical state-of-the-art solutions [23, 27] also use the relational model for storing semantic web data in relational databases. It is noteworthy that most of the contributions made in this thesis do not depend on the specific data model in use. For instance, the indexing and query processing techniques proposed in this thesis do not depend on the data model used, as they are based only on the characteristics of the underlying data. However, for concreteness we use RDF data model — a World Wide Web Consortium (W3C) recommended data model for describing entities, often referred to as resources in W3C.

2.1 Resource Description Framework

Resource Description Framework (RDF) describes resources in triples, also known as RDF statements or facts as $<$Subject(s), Predicate(p), Object(o)$>$ triples. The entities and the relationship between them could be any of the following:

- **URIs** (U): URIs or Uniform Resource Identifiers are identifiers which are uniquely used for identifying resources across datasets on the Web.
- **Blank Nodes** (B): Blank nodes uniquely identify nodes within a graph. These nodes are not assigned global identifiers aka URIs. Usually blank nodes are used for associating RDF facts with existing RDF facts.

- **String Literals** (L): Objects are denoted by string literals.

Definition 2.1.1. RDF Graph: An RDF Graph $\mathsf{G}= (\mathsf{S}, \mathsf{E}, \mathsf{O})$ is a triplet of starting node (S), ending node (O) and edges (E) where

- Starting nodes are a finite set of URI and Blank Nodes, such that $\mathsf{S} \subset (U \cup B)$
- Ending nodes are a finite set of URI, Blank Node and Literals, such that $\mathsf{O} \subset (U \cup L \cup B)$
- Edges are a finite set of RDF triples (s, p, o) such that $\mathsf{E} \subset (\mathsf{S} \text{ x } U \text{ x } \mathsf{O})$.

Consider the graph shown in Figure 2-3, where URI nodes are represented by circles. The graph shows that Toquart and Salmon Beach are located in Washington. Additionally, floods were reported in Salmon Beach. In future, we can easily add more triples about Salmon Beach to this graph. Such a flexibility, stems from schema-less nature of RDF. The graph can be expressed in RDF as follows in N-Quad form as shown in Fig. 2-1:

```
<Salmon_Beach> <reported> <flood> <id_57>.
<id_57> <reportedBy> <Tribune> <id_58>;
        <Salmon_Beach> <hasGeometry> "POINT(28.3, -80.6)" <id_59>.
```

Figure 2-1: SPARQL Query in N-Quad form

The same graph represented above is shown below in reified form in Fig. 2-2:

```
BASE <http://example.org/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
_:id_57 rdf:type rdf:statement .
_:id_57  rdf:subject <Salmon_Beach> .
_:id_57  rdf:predicate <reported> .
_:id_57  rdf:object <flood> .
_:id_57  <reportedBy> <Tribune> .
<Salmon_Beach> <hasGeometry> "POINT(28.3, -80.6)".
```

Figure 2-2: SPARQL Query in reified form

The above statements represent RDF in Turtle format [15]. In Turtle (subject, predicate, object) terms are separated by white spaces and each RDF statement is terminated

by a '.' . URIs in Turtle can be absolute or relative. An absolute URI is presented as <http://example.org/>. Relative URIs are resolved to the current base URI. Keyword BASE is used for denoting the base URI. Similarly keyword PREFIX denotes the prefix URI, which is concatenated with the local part to get the URI of the RDF resource. For example, the URI of rdf:type is <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>. For ease of exposition, we omit BASE and PREFIX URI's while describing RDF statements henceforth.

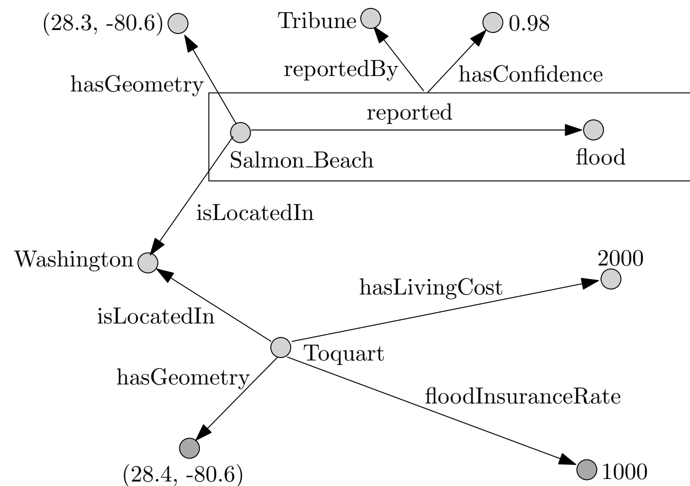


Figure 2-3: Example RDF Graph

2.2 SPARQL

SPARQL [36] is the standard query language for querying RDF. SPARQL queries support conjunctions and disjunctions of triple patterns. Triple patterns are similar to RDF triples barring that the resources may be variable. SPARQL's equivalent in relational world are select-project-join queries. The triple pattern matches that sub-graph of RDF graph, which has terms matching the variables in the SPARQL query.

SPARQL queries have the following format:

```
SELECT [projection clause]
WHERE [graph pattern]
FILTER [expressions]
ORDER BY [ranking function]
```

```
LIMIT [number of solutions]
```

The `SELECT` clause includes a set of variables that should be instantiated from the RDF knowledge base (variables in a SPARQL query are denoted by a "?" prefix). A graph pattern in the `WHERE` clause consists of *triple patterns* in the following forms:

1. *s p o*
2. *r* **rdf:subject** s. *r* **rdf:predicate** p. *r* **rdf:object** o,

r stands for *fact id* (or reification id). *s, p, o* and *r* can be either bound to constants, or unbound variables in the query. The predicates starting with prefix rdf are part of the RDF reification vocabulary to explicitly declare the various parts of an RDF statement identified through its identifier *r*. The `FILTER` clause limits to those results whose filter expression evaluates to `TRUE`. The `ORDER BY` clause in the query allows a user-defined ranking function to establish the order of bindings of the projected variables (the `SELECT` clause).

Although SPARQL 1.1 standard enables a large array of possible ranking functions to be used here, in this work we limit ourselves to *convex monotonic* functions involving quantifiable (i.e., numerical) values. Examples of convex monotonic functions are: $(a + b)$,($(a * b)$ for $a, b > 0$) The `LIMIT` clause controls the number of results returned.

Throughout this thesis, we use W3C's recommended standardized SPARQL [95] query language in its actual state, i.e., without any modifications or extensions for representing meta facts. Next we discuss motivation behind different types of SPARQL queries we use in this thesis.

2.2.1 Motivational Queries

In this subsection we begin by discussing motivational queries for RQ-RDF-3X, then we move to Quark-X and finally we talk about STREAK.

First type of SPARQL query which use in this thesis is a query over quads. For exposition Figure 2-4 depicts this query type. This query finds places which have been reported to be flooded. Obviously, the information reported may also be rumours, there is confidence value and source information associated with the reported statement, which are projected

out by the `SELECT` clause. We propose in **Chapter 3: RQ-RDF-3X**, a generalized storage engine which can efficiently answer these queries.

```
SELECT ?place ?newsAgency ?location
WHERE {
  ?reif rdf:subject ?place;
        rdf:predicate :reported;
        rdf:object :flood;
        :reportedBy ?newsAgency.
  ?place :isLocatedIn ?location.
}
```

Figure 2-4: Example SPARQL Query

Next, we focus on efficiently evaluating top-k queries over RDF quads. We were motivated by the following two observations: first, a large fraction of semantic resources such as YAGO and DBpedia contain facts linking to quantifiable values which naturally suggests the use of queries containing ORDER-BY. This is further amplified by the use of reification to attach annotations such as *confidence* and *time*. Secondly, despite the SPARQL recommendation of using ORDER-BY / LIMIT operators, efficient processing of top-k queries with a user-defined ranking function within RDF/SPARQL setting have received limited attention [64, 96].

Figure 2-5 depicts an example top-k query over quads. This query finds places near a flooded place which have low rates of house insurance and living cost. Again, flood news may also be rumours, there is confidence value associated with the *reported* statement which is one of the factors in the ranking function. Note that, nearness between places in the query is ensured by their co-location within the same county (state). We propose in **Chapter 4: Quark-X**, an efficient top-k query processing framework which efficiently answers such queries.

Finally, we observed that many RDF knowledge bases — such as YAGO [47, 46, 16, 65], DBpedia [12], LinkedGeoData [11], GeoKnow [35, 37, 58], etc. – encode not only simple relationships between entities, but also model higher-order information such as uncertainty, spatio-temporal context, strength of relationships, and more. Querying such knowledge bases often involves user-defined ad-hoc ranking with top-k result cut-offs. Thus, unlike our earlier work Quark-X, the challenge we face are two-fold: first, efficiently support sophisticated spatial querying and second, to support top-k early-termination – both

```
SELECT SELECT ?place ?anotherPlace ?newsAgency ?location
  ((f1(?livingCost) + f2(?insuranceRate)) * f3(?conf) as ?rank)
WHERE {
  ?reif rdf:subject ?place;
                rdf:predicate :reported;
                rdf:object :flood.
                :reportedBy ?newsAgency.
                :hasConfidence ?conf.
  ?place :isLocatedIn ?location.

  ?anotherPlace :hasLivingCost ?livingCost.
  ?anotherPlace :floodInsuranceRate ?insuranceRate.
  ?anotherPlace :isLocatedIn ?location.
} ORDER BY DESC(?rank) LIMIT 10
```

Figure 2-5: Example Top-k Query

in combination with the graph pattern query paradigm of SPARQL. Figure 2-6 depicts such a top-k spatial distance join query. This query finds places near a flooded place which have low rates of house insurance and living cost. Again, the flooding information reported may also be rumours, therefore there is confidence value associated with the *reported* statement which is one of the factors in the ranking function. We present in **Chapter 5: STREAK**, a top-k spatial distance join querying engine which competently answers such queries.

```
SELECT ?place ?anotherPlace ?newsAgency ?locPlace ?locAnotherPlace
  ((f1(?livingCost) + f2(?insuranceRate)) * f3(?conf) as ?rank)
WHERE {
  ?reif rdf:subject ?place;
                rdf:predicate :reported;
                rdf:object :flood.
                :reportedBy ?newsAgency.
                :hasConfidence ?conf.
  ?place :hasGeometry ?geoPlace1.
  ?place :isLocatedIn ?locPlace

  ?anotherPlace :hasLivingCost ?livingCost.
  ?anotherPlace :floodInsuranceRate ?insuranceRate.
  ?anotherPlace :isLocatedIn ?locAnotherPlace.
  ?anotherPlace :hasGeometry ?geoPlace2.
  FILTER (distance(?geoPlace1, ?geoPlace2) < 10)
} ORDER BY DESC(?rank) LIMIT 10
```

Figure 2-6: Example Top-k Spatial Distance Join Query

2.3 Datasets

In this sub-section we describe some popular real-world datasets from Linked Data domain which we have used for evaluating the thesis:

- YAGO (Yet Another Great Ontology) [89]: is a high-precision knowledge-base containing facts extracted from English Wikipedia and unified with Wordnet and GeoNames. Entities and binary relationships constitute YAGO's data model. In particular, YAGO has been constructed from category system and infoboxes from Wikipedia and taxonomy of concepts from Wordnet. YAGO contains close to 120 million facts and more than 10 million entities. YAGO uses reification to store additional information about facts.

- DBpedia [59]: DBpedia also is derived from Wikipedia, and contains over 400 million facts containing triples plus provenance information, describing 3.7 million things. It is important to note that DBpedia uses N-Quads for modeling higher order relationships.

  Using Wikipedia search alone it is very difficult to find rivers that flow into Bay of Bengal and are greater than 200 miles, or all German musicians who lived at the time Beethoven was born. However, Wikipedia's structured form DBpedia can be used to answer such expressive queries.

- LGD (Linked Geo Data) [11]: LGD contains freely available collaboratively collected data from Open Street Map (OSM) project. The OSM project contains a large amount of geographical data. OSM data can be classified into three basic data types: nodes, ways and relations. *Nodes* represent points, *ways* represent sequences of points, *relations* constitute nodes and/or ways. Ways with same start and end nodes are used to represent buildings, land use areas, etc. LGD contains approximately 2 billion triples.

2.4 Related Work

We next discuss the storage techniques employed by RDF storage engines. We also introduce a taxonomy to classify the storage techniques into two stages namely, *URI Encoding* and *indexing*. *URI Encoding* stage focuses on storing RDF resources efficiently. In this stage instead of storing URIs directly as string, RDF databases first associate a numerical identifier to each resource and store this identifier instead. After mapping the URIs to identifiers *indexing* stage focuses on efficiently storing SPO triples themselves.

1. **URI Encoding stage**: Uniform Resource Identifiers (URIs) used for uniquely identifying resources (Subject, Predicate, Object) are: (1) typically long in length and thus occupy a lot of storage space [69]; (2) have string data type – which makes comparisons difficult [69]. Therefore, it is inefficient to store URIs as-it-is [69]. In order to overcome this many RDF storage engines map URIs to numeric identifiers using a dictionary. Mapping URIs using dictionary has its own advantages and disadvantages. Based upon how the URI's are processed and stored in a triple store, RDF storage engines can be categorized in the following ways:

    (a) **Mapping URIs to numeric identifiers**: In this URIs are mapped to integers – such that the mapped integers are assigned in an increasing order of insertion of RDF facts. Advantages of this approach are listed below:

    - Numeric identifiers are of fixed length as compared to string URIs, hence processing is faster [69].
    - Numeric identifiers save a lot of space as they are typically shorter in comparison to string URIs [69].
    - Query execution is faster – as numeric comparisons are faster as compared to string comparisons [69].

    However, this approach suffers from the overhead of additional joins which need to be processed in order to map URIs to numeric identifiers and vice versa. Storage engines which employ this technique are: RDF-3X [69], Sesame [26], HexaStore [101], TripleBit [106].

(b) **Mapping URIs to hash values**: In this URI's are mapped to hash values. As is apparent this approach is prone to collisions between URI's which are mapped to the same hash value by the function. Thus unlike numeric identifiers which are assigned in an increasing order of occurrence of resources, hash values are assigned using a hash function and thus are liable to collision. In a manner similar to point 1, this approach suffers from an overhead of processing additional joins – used for mapping hash values back to URIs . The advantages of this approach are same as are outlined for numeric identifiers. Storage engine which employ this technique is: 3store [42]. 3store uses different hash functions for IRIs and literal values – the hash values generated by the hash function aid in distinguishing between the two.

(c) **Hybrid of URIs and numeric identifiers**: Small URIs are kept as-it-is whereas larger URIs are mapped to numeric identifiers. In this the storage medium is used for distinguishing between the system generated numerical identifiers and smaller string resources. This optimization helps avoid the extra join for mapping integers to resources for smaller string resources. Such an approach is adopted in Virtuoso [33].

Note that in this thesis in RQ-RDF-3X we adopt the approach of mapping resources to numerical identifiers. While in Quark-X and STREAK we use semantic encoding of entities for mapping resource to identifiers. At a high level, semantic encoding of identifiers is based on the fact that entities in RDF exhibit a soft schema [68]. Thus in Quark-X and STREAK we assign identifiers in an increasing order to entities exhibiting the same schema. This helps Quark-X and STREAK collocate entities sharing the same schema. This collocation of entities helps perform sequential disk accesses thus improving efficiency.

2. **Indexing**: Based on the manner in which RDF triples are stored in RDF storage engines, we categorize along the following dimensions:

   (a) **Horizontal Representation**: A single table is used for storing RDF facts – here subject, predicate and object act as attributes of the table. In order to improve

Subject	**Predicate**	**Object**
RDF-3X	author	Neumann
GraphLab	author	Yuncheng Low
Neumann	worksAt	MPI
..	..	..

Table 2.1: Horizontal Representation

performance appropriate indexes can be build over the table. Approaches like RDF-3X [69], Hexastore [101] and Virtuoso [33] implement this approach – for increasing performance, indexes are built over single virtual table. Storing multiple permutations of indexes considerably improves performance, but these techniques suffer from the issue of: (i) increased index scan time with increase in the size of the table; (ii) high storage cost as these approaches store same data in multiple redundant permutations. An example illustrating Horizontal Representation approach is shown in Table 2.1.

Note that all three database systems viz RQ-RDF-3X, Quark-X and STREAK proposed in this thesis adopt Horizontal Representation approach of storing RDF facts. Efficient compression technique for indexes proposed in this thesis in section 3.4.1 helps reduce the storage footprint.

(b) **Graph-based Storage**: Querying RDF graphs typically involves finding the result sub-graph in the original graph. Existing RDF engines belonging to this category are described below:

- GRIN [90]: GRIN partitions the RDF graph using Partitioning Around Medoids (PAM) [55] clustering algorithm. In PAM first centroids are randomly chosen and then clustering is done based on the distance of the other vertices from the centroid vertices. After this centroid vertices are recomputed and clustering is again performed until equilibrium is reached.
- DOGMA [25] performs graph partitioning using GGGP graph partitioning algorithm [54]. It partitions graphs in such a way that number of edges crossing in between sub-graphs stored on leaf pages is minimized – this is advantageous for queries whose sub-graphs can be found on a single page. This approach is inefficient for queries where the resultant subgraph spans

Subject/Predicate	**author**	**worksAt**	**publishedIn**	**hasWon**
RDF-3X	Neumann	NULL	VLDB	bestPaperAward
Neumann	NULL	MPI	NULL	NULL
...	...	...	...	...

Table 2.2: Vertical Table

multiple pages, thus requiring all pages to be fetched from the disk. Unlike its predecessor, GRIN, DOGMA operates on disk.

- chameleon-db [8]: In this, sub-graph matching algorithms are run on partitioned RDF graph. For efficiency, chameleon-db periodically repartitions its database depending on the workload. Unlike DOGMA and GRIN which are workload agnostic, chameleon-db is workload aware – such that it adjusts its partitions based on the query workload.

Storage approaches suggested in DOGMA and GRIN work efficiently for a specific query workload or for certain query patterns. chameleon-db overcomes this drawback by re-partitioning based on the query workload – but chameleon-db's performance is unclear(variable) for rapidly changing query patterns.

(c) **Property Table**: Another approach is to aggregate entities sharing the same properties and to store them in one table as shown in Table 2.2. In Table 2.2 Subjects are stored horizontally in first column, and Predicates are stored as column names. For example, for the fact (RDF-3X, author, Neumann), RDF-3X is stored in the first column, predicate author is the column header, and object Neumann is stored at the row and column intersection. This approach eliminates the need for Subject-Subject joins and hence this helps improve efficiency.

This approach suffers from the disadvantage of: (a) Increased space requirement due to the presence of NULL values (b) Queries for which property values are not specified may involve scanning the entire table (c) Storage of multi-valued properties becomes difficult. Some of the present day frameworks which store data in this format or its slight variants are explained below:

- TripleBit [106]: TripleBit's design of property table is slightly different: here subjects and objects are represented in rows and predicates are repre-

sented in columns. Values in the matrices are either 0s or 1s. Each triple is uniquely identified by two 1s in the corresponding column.

- Levandoski and Mokbel [60] propose to cluster properties together and store them in property or n-ary tables, properties which dont belong to clusters – clustered based on co-occurrence – are stored in accordance with column-store approach (described in (d)). Here, properties which coexist together for a large number of subjects are grouped in one cluster. The advantage of this approach being that it efficiently processes star-shaped queries.

- A specialized form of property tables is *clustered property tables*. In clustered property tables correlated predicates are stored in the same table and the remaining predicates (which are not correlated with other predicates) are stored in a separate table. Example frameworks which use this approach are: Jena2 [103] and Sesame [27].

Since exhaustive indexing is typically not done in vertical partitioning, therefore as compared to triples table and graph based approach, this approach suffers from the disadvantage of scanning all triples, for properties which are left unspecified.

(d) **Column-store**: It employs a decomposed storage schema [29]. In this approach a separate two-column table is created for each property – in this way it resembles the column-store storage approach of relational databases. This approach works well for queries where property values are specified. The negative side of this approach is that it degrades the performance of queries where (i). properties are unrestricted (ii). joins need to be performed across tables – as this will slow down result retrieval in comparison to approaches which sequentially access stored information. This approach has also been popularly described in literature as: Vertical Partitioning [92] or horizontal (binary) stores [81] or binary table approach. Frameworks which store RDF data in this form are: RDF data-centric storage [60]; Abadi, et al. [4]; [85]; [5] etc. Some of the

frameworks using this approach are briefly described below:

- Abadi, et.al. in their work [4], [85], [5] suggest a column store approach with property tables sorted by subject. Its advantage being it enables fast processing of merge joins and hence gives efficient performance for queries in which properties and/or subjects are specified. This approach stores multi-valued subjects in consecutive rows of the property table. In order to gain advantage column-oriented database systems are used as underlying storage – the approach thus obtains benefits of compressibility and performance of column-oriented stores.

(e) **Storing Pre-computed Joins**: This category contains frameworks which store precomputed joins – some of these are given below:

- Groppe, et.al. [38] in their work illustrate that two triple patterns can be joined in 6 ways (Subject-Subject, Subject-Predicate, Subject-Object, Predicate-Predicate, PredicateObject, Object-Object). If two simple triple query patterns are joined by a common variable then there are only 4 relevant positions (e.g. for Subject-Subject join – the relevant positions are: Predicate and Object of both the triples), therefore the authors construct and use 16 different indices for each join case. For all 6 join scenarios authors use 96 different indices, which the authors show is practical from their experimental evaluation.

2.5 RDF-3X

Next, we describe RDF-3X, the RDF data management system, that forms the basis for the three systems we propose in this thesis. RDF-3X is used as an underlying framework for the proposed database engines in this thesis namely, RQ-RDF-3X, Quark-X and STREAK. RDF-3X is a RISC-style architectured state-of-the-art RDF query processing engine. In the next sections, we focus on the storage and query processing components of RDF-3X.

2.5.1 Storage

RDF-3X employs a simple yet powerful solution of materializing all sorted permutations of the schema. Materializing all possible permutations as indexes does away with the need for physical design tuning. It achieves this by building materialized clustered indexes over all six possible permutations of SPO. Furthermore it builds count-aggregated variants for all three two-dimensional and all three one-dimensional projections. An important point to note here is RDF-3X stores these indexes in a compressed format using delta compression. As a result the total storage space required to store these materialized indexes is less than size of the raw dataset itself. Next we explain the indexes used by RDF-3X in detail. RDF-3X stores all SPO triples in clustered B+-trees. Individual pages in B+-trees are compressed using delta encoding. For better compression and for efficient range scans triples are sorted and stored lexicographically in B+-trees. An additional optimization RDF-3X employs is to replace strings in SPO triples by integer ids using a mapping dictionary.

2.5.2 Query Processing

Translating SPARQL queries

RDF-3X first compiles a SPARQL query and constructs its query graph representation. Each triple pattern denotes a node in the query graph. This induces a scan of RDF-3X's indexes with literals used for performing selections. Two nodes in the query graph are connected if and only if they share a common query variable.

Plan Generation

RDF-3X uses a bottom-up dynamic programming based query optimizer. The query optimizer tries to draw maximum advantage of RDF-3X's materialized sorted indexes by adopting merge joins as much as possible. It generates plans which preserve interesting orders for successive joins so that it can accelerate queries using its materialized indexes. Only when merge joins are not possible RDF-3X's optimizer switches to hash-based join processing.

Next we discuss RDF-3X's bottom-up dynamic programming algorithm for plan generation. RDF-3X seeds the DP-table with scans of SPO indexes. Index selection is based on two factors namely: (1). Performing efficient range scans by matching the prefix of the index with literals in the triple pattern; (2). Choosing indexes which are suitable for subsequent merge joins later on. This is because the selected plan may have lower overall cost. After seeding the DP table. The optimizer begins by considering all plan choices. It then discards plans based on estimated execution costs to reduce the search space of plan exploration. It does so by generating larger plans by joining optimal solutions of smaller plans. An additional optimization which RDF-3X's query optimizer embeds is to greedily push down the FILTER predicates in the query plan.

Selectivity Estimation Next we describe RDF-3X's selectivity estimation in detail. For finding selectivities of individual triple patterns RDF-3X precomputes exact result cardinalities of SPO triple patterns and stores them in a B+-tree. Thus finding selectivities now requires one or two lookups in B+-trees – the authors of RDF-3X claim this to be insignificant compared to actual query execution cost.

For estimating selectivities of joins between two triple patterns RDF-3X transforms it as a join between one triple pattern and all triples in the database and a final selection. To understand this better consider a join between (c1, c2, ?o1) and (?s2, c3, c4) on ?o1=?s2, here ci's are constants, while variables begin with a '?' e.g. ?o1, ?s2, ?p2, ?o2, etc. Now this join can be re-written in the following manner:

1. Join between (c1, c2, ?o1) and (?s2, ?p2, ?o2) on ?o1=?s2 and
2. (1) Followed by a selection on ?p2=c3 and ?o2=c4

Note that although this form is inefficient for query execution but because it is equivalent to the original form, therefore it can be exploited for selectivity estimation. Note again that we already computed (2)as described above where we found the selectivity of individual triple patterns.

For finding (1) a triple pattern (c1, c2, ?v) needs to be joined with all the other triples (?s2, ?p2, ?o2) in a database on either subject, object or predicate. Assuming a join on

either of them and let ?v1 denote the joining variable. We want to compute:

Cardinality (c1, c2, ?v) $\bowtie_{?v=?v1}$ (?v1, ?p2, ?o2)

= | (c1, c2, ?v) $\bowtie_{?v=?v1}$ (?v1, ?p2, ?o2) |

= $\sum_{x \in (c1,c2,?v)}$ |x, ?p2, ?o2|

RDF-3X precomputes and stores the above join cardinality in B+-trees. Similarly join size pre-computation is performed for triple patterns with one constant and no constant. These materialized join cardinalities helps RDF-3X compute join selectivities between SPO triple patterns in query graph.

Sideways Information Passing For exposition consider a query which wants to retrieve the location of a place whose official language is English, and has been reported to be flooded. Its operator tree is as shown in the Figure 2-7:

$\bowtie_{S=S}$

$\bowtie_{S=S}$

O	P	S
119	20	11
119	20	55
119	20	127
119	20	135
119	20	239

O=English,
P= Language

O	P	S
111	3	11
111	3	55
111	3	127
111	3	135
111	3	239

O=Flood,
P= reported

P	S	O
100	13	48
100	19	57
100	127	58
100	255	59
100	266	61

P=isLocatedIn

Figure 2-7: Operator Tree showing SIP

From Figure 2-7 notice in PSO index during scanning column S there are large gaps in 19 and 127; 127 and 255. Now if we pass this information to the other OPS indexes, then this allows the indexes to skip large parts. This passing of information from one index to the other – for skipping large parts of indexes – is called *Sideways Information Passing*.

Chapter 3

RQ-RDF-3X: An Efficient Quad-Store

3.1 Motivation

Given reification and N-Quad extensions to the basic triple model that is most associated with RDF, a natural question to ask is: how do existing RDF stores cope with these extensions? In this chapter, we will restrict our discussion to native stores alone, although similar observations can be made for relational RDF stores as well. The state-of-the-art native stores such as RDF-3X [69] and Hexastore [101] are strictly designed for storing triples, and answering queries over them using SPARQL. As a result, they can incorporate only the standard reified models in their storage model. However, this severely degrades their query processing efficiency due to increased number of joins that need to performed in order to answer even the simplest of queries. The data in N-Quad can be stored in these systems only after converting it to a reified form, and is thus equally inefficient to query. To understand how reification results in a large number of joins consider a query wanting to retrieve all predecessors of Governors of New York. Note that this is the representation which RDF-3X uses to represent reified queries. This query can be expressed in SPARQL as:

```
select ?a?b?e
where {
        ?r rdf:type rdf:statement.
```

```
?r rdf:subject ?a.
?r rdf:predicate <holdsPoliticalPosition>.
?r rdf:object <Governor_of_New_York>.
?r <hasPredecessor> ?b }
```

Note that the above query needs to perform 5 joins on ?r between (?r rdf:subject ?a), (?r rdf:type rdf:statement), (?r rdf:predicate <holdsPoliticalPosition>), (?r < hasPredecessor> ?b) and (?r rdf:object <Governor_of_New_York>). Therefore, reification results in large number of joins, in comparison to RQ-RDF-3X – described in detail in this Chapter – which replaces them with one equivalent single join between:

```
?r ?a <holdsPoliticalPosition> <Governor_of_New_York> and
?r <hasPredecessor> ?b
```

We arrived at the above representation by assigning a unique identifier to each triple in the database. In the above RDF query, the variable representing this identifier for the triple (?a, holdsPoliticalPosition, Governor_of_New_York) is ?r. Thus, for retrieving the results a single join needs to be performed between (?r ?a <holdsPoliticalPosition> <Governor_of_New_York>) and (?r <hasPredecessor> ?b).

In a nutshell, in comparison to RDF triple-stores like RDF-3X which can query only over reified data, RQ-RDF-3X needs to perform 3x lesser number of joins. This helps improve query performance.

3.2 Organization

The chapter is organized as follows: Section 3.3 outlines related work. Section 3.4 describes proposed approach. Section 3.5 reviews performance and query evaluation.

3.3 Related Work

This section presents an overview of frameworks which support storage of reification and N-Quads.

3.3.1 Reification Support

Some of the prominent frameworks which support reification are: Jena2 [103] and Oracle [6]. Jena2's predecessor Jena1 [103] also supported reification, but its applicability was limited by its ability to reify each fact only once. Jena2 [103] overcame this limitation by storing separate tables for asserted and reified statements. Reified statements in Jena2 are stored in a separate property tables. The property table contains Statement URI, rdf:subject, rdf:predicate and rdf:object as columns. None of the systems mentioned above have reported any performance benchmarks on big-data datasets containing more than 20-30 million triples. Additionally, reified data deteriorates the query processing efficiency of state-of-the-art triple stores like RDF-3X, Hexastore, etc. This is due to the increased number of joins that need to be performed in order to answer even the simplest queries.

Next we highlight the difference in RQ-RDF-3X's compression scheme with respect to RDF-3X, whose architecture is explained in detail in Section 2.5 of Chapter 2. The compression scheme in RDF-3X allows to store only SPO triples efficiently. This is because it uses a header byte to store the lengths of fixed byte encoded SPO triples which differ from each other only in one value. RDF-3X completely uses all bits in its header byte and is thus not able to accommodate size information of R. A possible extension of fixed byte encoding scheme used in RDF-3X is to use two header bytes – which we show experimentally later is not an efficient option because of increased storage space requirement and hence decreased query processing efficiency. Extension of fixed byte encoding of RDF-3X is shown in Fig. 3-1. RQ-RDF-3X on the other hand uses variable byte encoding in addition to fixed byte encoding and is thus able to efficiently store S, P, O and R.

3.3.2 N-Quads

YARS2 [44, 45] stores quads in the form of <subject, predicate, object, context>. They built 6 indices such that the indices cover all the 16 possible access patterns of quads. In a similar manner Kowari [104] also stores RDF statements in six different orders of indexes such that the indices cover all possible access patterns. However in contrast to YARS2, Kowari stores the indices using AVL trees. As both YARS2 and Kowari are optimized

for simple lookups, therefore they do not process complex queries efficiently [69, 101]. Virtuoso[33] is another well-known quad-store.

3.4 RQ-RDF-3X Framework

We will focus on the extensions needed to triple-store style RDF storage engines to support N-quads, by using RDF-3X [69] as the basis for our concrete implementation. Recall that we explained the architecture of RDF-3X in detail in Section 2.5 of Chapter 2.

The fundamental change required is to support an additional field (R) that has the reification identifier or context identifier. This requires a wide-spectrum of modifications within the storage engine ranging from storage and index organization, selectivity estimation as well as query planning and optimization. In the rest of this section, we describe these modifications within the context of RDF-3X.

3.4.1 Storage and Indexing in RQ-RDF-3X

RQ-RDF-3X supports reification by assigning a unique identifier (R) to each Subject (S), Predicate (P), Object (O) triple. This is similar to earlier proposals [103, 6], also aimed at minimizing the explosion in triple count due to naive reification. As a consequence, only one additional fact is required to store the meta-data associated with each fact.

The aggressive indexing strategy of RDF-3X is taken further in RQ-RDF-3X by building clustered B^+-tree indexes over all 24 permutations of (S, P, O, R)-quads, in addition to all permutations of ternary, binary and unary projections of these quads. Refer to Section 2.5.1 regarding further details on RDF-3X's indexing. We store these (S, P, O, R) indexes and their ternary projections using a novel compression strategy for quads, which can also extend easily to very large datasets.

Index Compression in RQ-RDF-3X

For ease of exposition, we represent the four values in any permutation of the (S, P, O, R)-quad as (v_1, v_2, v_3, v_4). Indexes are lexicographically sorted quads stored in clustered

B^+-trees, whose entries are compressed to retain scalability as well as provide economy of space.

Each B^+-leaf entry comprises two parts – a header byte and zero or more value bytes. The header byte is used as four 2-bit entries where each pair of bits encode the number of bytes consumed by the corresponding entry. The bit pattern `11` is used to signify that the value uses 3 or more bytes.

For storing values (v_1 / v_2 / v_3 / v_4) in the quad, we first apply delta encoding with respect to the previous quad. This is because lexicographic ordering causes most neighboring triples to have similar values of v_1 and v_2 with very small differences for v_3 and v_4. Thus we use delta encoding for storing only the changes between quads. When delta value needs 1 or 2 bytes, then we directly encode number of bytes consumed by an entry in the 2-bits reserved for it in the header byte. We then store the value itself using the efficient fixed byte encoding. While if the delta needs 3 or more bytes, we use variable byte encoding in the third byte onwards. In variable-byte encoding an integer is encoded as a sequence of 7-bit bytes, with the remaining bit denoting 1 for last byte and a 0 for all other bytes. We use a combination of fixed and variable byte encoding to use the header byte effectively. Note that using only fixed byte encoding for storing values requires two header bytes, which we show experimentally later is not an efficient option because of increased storage requirement and hence decreased query processing efficiency.

Figure 3-1 illustrates the compression scheme, considering three different possible scenarios. In Figures 3-1(a) and 3-1(b), delta of v_1,v_2 and v_3 are zero, hence only v_4 is stored. Figure 3-1(c) illustrates the condition where the adjacent quads are distinct.

Aggregated Indices

For certain SPARQL queries building indices on just the ternary, binary and unary projections is sufficient. Such indexes are called aggregated indexes henceforth. For example, consider the query

```
select ?a?b?e
where { ?r rdf:subject ?a.
        ?r rdf:predicate ?b.
```

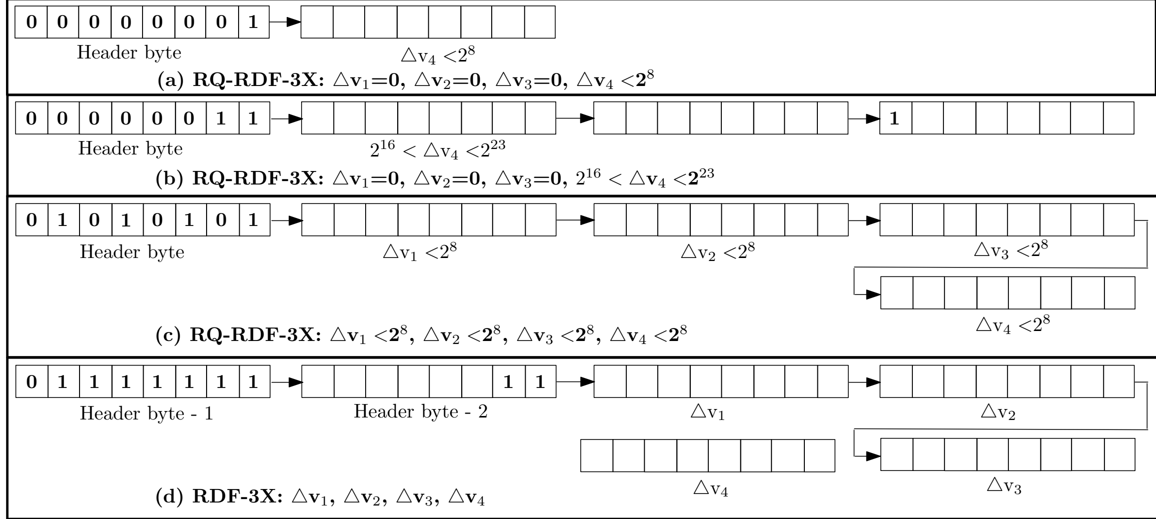


Figure 3-1: (a,b,c)Compression Scheme in RQ-RDF-3X (Leading bit 1 for last byte and 0 for all other bytes) (d) Compression Scheme of RDF-3X for Quads with first two header bytes

```
?r rdf:object ?c.
?r ?d ?e }
```

It finds information about the triple's subject and predicate, along with the metadata which is associated with the triple `?a?b?c`. In order to do so, RQ-RDF-3X transforms the query into 2 sub-queries: (i) `?r ?a ?b ?c` (formed by aggregating the reified query patterns) and (ii) `?r ?d ?e`. As the query does not need to retrieve the bindings of the variables `?c` and `?d` it suffices to store just the ternary projection for `?r ?a ?b ?c` and binary projection for `?r ?d ?e`. In RQ-RDF-3X we have build 24 ternary projection indices (SPR, SPO, SRP, SRO, SOR, SOP, PSR, PSO, PRS, PRO, POR, POS, RSP, RSO, RPS, RPO, ROS, ROP, OSR, OSP, OPR, OPS, ORP, ORS), 12 binary projections indices (SP, SR, SO, PS, PR, PO, RS, RP, RO, OS, OP, OR) and 4 unary projections indices (S, P, R, O) for faster scans for processing queries that do not require all bindings. In these indices, in addition to storing appropriate S,P,R,O values the count of projected values in the dataset are also stored. These counts serve in supporting correct number of duplicates in results and also aid in selectivity estimation as we describe below. We show experimentally in Section 3.5 that the cost of storing these indices is not too high.

3.4.2 Selectivity Estimation

Accurately estimating the selectivity of query patterns is essential for query optimization. In RQ-RDF-3X, the cardinality of simple SPARQL patterns that have one, two or three variables can be accurately determined by using the frequency counts stored in aggregated indexes for appropriate ternary, binary and unary projections respectively. If the query pattern consists only of variables, then the result cardinality is simply the total number of quads in the database.

Join Selectivity Estimation

Before we begin the join selectivity estimation in RQ-RDF-3X, we briefly recap the process in RDF-3X (for triple pattern joins). In RDF-3X, a join between two triple patterns with projected attributes is translated into a join between a pattern with selection and one without selection. The triple pattern with lower cardinality is chosen for selection. Thus, the join selectivity is estimated as the selectivity of one triple pattern with all facts in the database. This is explained in detail with example in Section 2.5.2. When extending this approach in RQ-RDF-3X for quads, we have to deal with four separate cases depending on the number of variables in the query – i.e., one, two, three or four variables.

A join is evaluated in a manner similar to RDF-3X by selecting the query pattern with least cardinality to be joined with the pattern without any selection. If the SPARQL query pattern having lesser cardinality contains only one variable then it may be joined with either subject, predicate, reification or object present in the other query pattern. Let us suppose that the join is performed on subject as illustrated in figure 3-3, then the join selectivity between two SPARQL query patterns is estimated by finding all quads where subject of the first triple pattern is equal to the subject, predicate, reification or object of the second triple pattern. With one the patterns containing 3 constants and the other contains no constant, this boils down to summing the unary projections for all subjects present in the database. However, when executed naively it leads to inaccurate cardinality estimates.

We illustrate this problem with the help of a small example: consider the dataset expressed in the form of (S,P,R,O) shown in figure 3-2:

```
< a, b,  r1, c1 >
< a, b,  r2, c1 >
< a, b,  r3, c1 >
< a, b1, r4, c5 >
< a, d1, r5, f1 >
< a, d2, r6, f2 >
< a, d3, r8, f3 >
```

Figure 3-2: Example Dataset in RQ-RDF-3X

Figure 3-3: Reified Query Reformulation

Consider now the join query pattern:

`(?s, b, ?r, c1)` ⋈ `(?s, b1, ?r1, c5).`

The actual cardinality of the above join is 3, while the procedure described above estimates it as 7.

Recall that in RDF-3X the join selectivity between two SPARQL query patterns is estimated by finding all quads where subject of first triple pattern is equal to the subject, predicate or object of the second triple pattern. Also note that reification identifiers are unique identifiers introduced by the system, which the users are unaware of and hence are always represented by variables in the query. Thus RDF-3X's join selectivity method does not work with reified triples, as the estimated join cardinality will be equal to all facts in the knowledge base. Thus in RQ-RDF-3X in order to estimate the join cardinality accurately for triple patterns containing at least two constants we estimate the join cardinality as the highest cardinality of the query pattern forming the edge.

Our algorithm for selectivity estimation is described step-by-step below:

1. For estimating the selectivity of join between two quad patterns we first transform it as a join between one quad pattern and all quad in the database and a final selection.

2. For joins on reification identifier with two constants, join cardinality is estimated to be the highest cardinality of the query pattern forming the edge.

For finding (1) a triple pattern (c1, c2, ?v) needs to be joined with all the other triples (?s2, ?p2, ?o2) in a database on either subject, object or predicate. Assuming a join on

either of them and let ?v1 denote the joining variable. We want to compute:

Cardinality (c1, c2, ?v) $\bowtie_{?v=?v1}$ (?v1, ?p2, ?o2)

= | (c1, c2, ?v) $\bowtie_{?v=?v1}$ (?v1, ?p2, ?o2) |

= $\sum_{x\in(c1,c2,?v)}$ |x, ?p2, ?o2|

The pseudo-code of the algorithm is given below:

Algorithm 1 Pseudo Code for finding cardinality

Input Patterns to be joined (?a, ?b, ?c, ?d) and (?d, ?e ?f, ?g)
Output Estimated Cardinality

```
1: if ?d=reification then:
2:     if (?a=const && ?b=const) || (?b=const && ?c=const) || (?a=const && ?c=const) then
3:         card ← max(cardinality(?a, ?b, ?c, ?d), cardinality(?d, ?e, ?f, ?g)     ▷ determined by
   using frequency counts stored with aggregated indexes
4:     else
5:         card ← | (c1, ?b, ?c, ?v) ⋈_{?v=?v1} (?v1, ?p2, ?o2,?r2) |
6:         card ← ∑_{x∈(c1,?b,?c,?v)} |x, ?p2, ?o2, ?r2|
7:     end if
8: end ifreturn card
```

To summarize, accurate cardinality estimates can be obtained for such cases by using RDF-3X's selectivity estimation algorithm, barring the join between input patterns containing at most two variables, one of which is reification. In such exceptional cases the estimated cardinality is equal to the highest cardinality amongst joining input patterns. This is explained in detail with the help of an example below.

For the example given above as `(?s,b,?r,c1)` has a cardinality of 3 which is smaller than 7 (estimated cardinality obtained after naive extension); therefore, RQ-RDF-3X correctly chooses 3 as the cardinality estimate, which also happens to be exact. For named graphs the procedure is similar, with an exception in condition (ii) that the higher cardinality pattern should have 3 constants.

The join selectivity is estimated in a similar manner when the query pattern contains 2, 3 or 4 variables. The number of possible ways to join them are 12, 12 and 16 respectively. We store the calculated selectivity in B^+-trees, indexed by constants present in the query pattern. We acknowledge that since we extended RDF-3X's approach of storing calculated selectivities in B+-trees therefore this technique suffers from high storage cost. Note that although this approach incurs high storage cost, as our experiments show, this is offset by

superior query processing performance.

3.4.3 Query Translation and Optimization

RQ-RDF-3X transforms SPARQL queries containing reified triple patterns into tuple calculus form, which is used by the optimizer to determine an optimized execution plan. The first step in this is to identify all the quad patterns in the SPARQL query. For instance, given the following reified query:

```
select ?a ?b ?c ?e ?d
where
{?r rdf:subject ?a.
 ?r rdf:predicate <holdsPoliticalPosition>.
 ?r rdf:object <Governor_of_New_York>.
 ?r <hasPredecessor> ?b.
 ?r <hasSuccessor> ?c.
 ?r1 rdf:subject ?c.
 ?r1 rdf:predicate <hasWonPrize>.
 ?r1 rdf:object ?e.
 ?r1 <extractionSource> ?d}
```

Figure 3-4: Find the predecessors and successors of Governor of New York who have also won a prize

The following quad patterns are identified from the above query:

```
Q1: ?a, <holdsPoliticalPosition>, ?r, <Governor_of_New_York>
Q2: ?r, <hasPredecessor>, ?r2, ?b
Q3: ?r, <hasSuccessor>, ?r3, ?c
Q4: ?c, <hasWonPrize>, ?r1, ?e
Q5: ?r1, <extractionSource>, ?r4, ?d
```

The optimized execution plan is determined in a two stage process, not very different from the approach taken by RDF-3X. In the first stage, all variable bindings that are not used in other parts of the query are identified in order to project them away with the help of aggregated indices. Naturally, for all triple matching query patterns involving just <S,P,O>, reification can be projected out.

In the next stage, the optimizer retains indexes that may produce tuples in order appropriate for successive merge joins; and plans thus generated are pruned by the plan pruning mechanism of RDF-3X. Based on the resulting set of plans for smaller problems (for quad patterns), larger problems are solved using a bottom-up dynamic programming approach.

3.5 Evaluation

In this section, we evaluate the performance of RQ-RDF-3X by considering state-of-the-art solutions for managing reified RDF data over large RDF datasets that contain significant amount of reified content.

3.5.1 Experimental Setup

As already mentioned, RQ-RDF-3X is built on top of RDF-3X system which is written in C/C++ and runs in single-threaded mode. All experiments we report were performed on a server class machine Dell R650 with 2.50GHz Intel Xeon E5-2640 running Ubuntu 12.04 LTS. Additionally the machine had 2.6 TB effective storage running at RAID-5 with disk speed of 7200 rpm. Also the server has 64GB RAM. All numbers are reported after running experiments 5 times and taking their average. To ensure that the OS caches are cleared after every run, we drop all filesystem caches using `echo 3 > /proc/sys/vm/drop_caches`.

We conducted experiments on two large-scale real-world datasets, YAGO [89] and DBpedia (3.7 dump) [43].

It is important to note that neither of these two datasets model reification in the way it is defined in the RDF standard, choosing instead a more compact representation — YAGO [88] uses Turtle as its native format: fact identifiers are stored in a commented line before each fact, and DBpedia uses N-Quads. Table 3.2 provides the raw-size of these datasets, in their native form as well as after translating them to the standard reified form. As these numbers show, reified forms incur 20-25% more space in the raw size. Further, Table 3.3 summarizes the amount of reified content found in these datasets. As these numbers show, a significant fraction of content is reified and in YAGO, these reifications are

quite complex with upto 10 entries taking part in a reification on an average.

3.5.2 Compared Systems

RQ-RDF-3X is compared with four state-of-the-art systems: PostgreSQL (ver. 9.1.9) [78], RDF-3X (v0.3.7) [69], Virtuoso (ver. 06.01.3127) [33] and Jena-TDB (ver. 2.10.0) [53]. We briefly describe how these systems were set up in our evaluations:

PostgreSQL: YAGO and DBpedia were loaded from their reified form by creating two tables: (i) a dictionary mapping each string to a unique integer id, and (ii) a four-column table of (O, R, P, S) ordering consisting of integer ids generated from the facts table of RQ-RDF-3X. Indexes were built on the dictionary as well as all 24 permutations of (O, R, P, S).

Virtuoso and Jena-TDB: Both these systems are quad-stores widely used in the Semantic Web community. They store a triple plus a graph reference per row – i.e., in named graph semantics. We loaded YAGO by treating reification identifiers that identify each triple uniquely as named graph ids. On the other hand, DBpedia can be loaded in named graph format (its natural form) as well as reified form. We loaded DBpedia into Virtuoso in both forms, and into Jena-TDB as named graphs only. Jena-TDB and Virtuoso were widely used at the time of writing, these days Blazegraph is also more widely used.

RDF-3X: Being strictly a triple-store, RDF-3X can be loaded with only fully reified datasets – DBpedia and YAGO were converted appropriately.

Table 3.2 also reports the resulting database sizes for each of the systems we have used. As the table clearly shows, Virtuoso has the most compact database for both named graph and reified forms. On the other hand, PostgreSQL lies on the other extreme with almost 6-10 times larger database than Virtuoso. RQ-RDF-3X has the largest database size – barring PostgreSQL – due to its very aggressive indexing.

Query id	# TP	# Joins	Degree of Joins	Types of Joins	# R-TP
1	4	2	(3,2)	(RS, OS)	1
2	5	3	(3,2,2)	(RS,OS)	3
3	5	2	(3,3)	(RS, OO)	2
4	4	3	(2,2,2)	(RS, OO)	2
5	5	4	(2,2,2,2)	(RS, SS, OS)	2
6	6	3	(3,3,2)	(RS,OO,OS)	2
7	5	4	(2,2,2,2)	(RS, SO)	1

(a) YAGO

Query id	# TP	# Joins	Degree of Joins	Types of Joins	# R-TP
1	6	4	(3,2,2,2)	(RS,OO)	3
2	2	1	(2)	(RS)	1
3	4	3	(2,2,2)	(RS,OO)	2
4	2	1	(2)	(RS)	1
5	4	3	(2,2,2)	(RS,OS)	2
6	4	3	(2,2,2)	(RS,OS)	2

(b) DBpedia

Table 3.1: Summarized Characteristics of Benchmark Queries in RQ-RDF-3X

3.5.3 Benchmark Queries

The benchmark queries of RQ-RDF-3X were influenced by the SPARQL query features that have been found to be quite important by Aluç, et al. [7]. Important features which we consider in this work are: (a) the number of triple patterns (TP), (b) the number of joins, (c) the degree of joins, (d) types of joins. In addition we also consider one additional feature: the number of reified triple patterns in a query (R-TP): It denotes number of triple patterns in the SPARQL query with which additional information is associated. Table 3.1 shows summarized characteristics of benchmark queries. Queries in our benchmark have been constructed keeping in mind their diversity in terms of query features such as number of triple patterns, number of joins, degree of joins, type of joins and number of reified triple patterns in the query.

Note that the system only supports SPARQL queries containing joins between triple patterns. SPARQL queries containing filtering, unions and negations in pattern matches, property paths, optionals and not exists are not supported currently by the system.

	YAGO	DBpedia-RF	DBpedia-NG
Raw RDF data size(in Turtle)	18.5	-	-
Raw RDF data size(in reified form)	22	57	45
PostgreSQL	493	549	-
RDF-3X	61	87	-
RQ-RDF-3X	113	190	-
Virtuoso	45	80	43
Jena-TDB	101	-	68

Table 3.2: Sizes of Datasets and Databases (in GBs)(RF: Reified Form, NG: Named Graphs)

	YAGO	DBpedia
Fraction of facts with reification	31.5%	50%
Average size of reified entries	10	1

Table 3.3: Prevalence of Reified Entries

3.5.4 Query Processing Performance

For YAGO and DBpedia, we constructed a small benchmark query set consisting of queries that use reified context. We provide the query set for DBpedia and YAGO as well as the SQL translations of these queries in Appendix A.1.

In the same manner as RDF-3X[69], for PostgreSQL and RDF-3X we report in Figure 3-5 query execution time (excluding the compilation time). For PostgreSQL, we manually translated the queries into SQL so that they are as efficient as possible. Further, to eliminate the issues our PostgreSQL installation had in the final dictionary lookups, all numbers we report are *before* the final id to string dictionary lookup. When dictionary look-up time is included, the overall runtime of RDF-3X and RQ-RDF-3X increase by less than 10% while PostgreSQL slowed down by an order of magnitude. We make a separate one-on-one comparison with Virtuoso and Jena-TDB, by reporting the total query execution time(compilation and query execution time) in Figure 3-6.

Figure 3-5 and 3-6 summarizes the results of our evaluation of query processing performance. As these results clearly demonstrate RQ-RDF-3X is significantly faster than PostgreSQL, RDF-3X, Virtuoso and Jena-TDB. Specifically, RDF-3X turned out to be consistently slowest when compared to PostgreSQL, RQ-RDF-3X, Virtuoso and Jena-TDB. Although we gave significant advantage to Virtuoso and Jena-TDB–by loading Dbpedia in named graph form which has smaller database size — RQ-RDF-3X outperforms these

systems for all queries except for Query 5 over DBpedia, where Jena-TDB is slightly faster.

We summarize the results of our query evaluation performance in comparison to PostgreSQL and RDF-3X in warm-cache in Figure3-7. Note that similar to cold-cache the results demonstrate the superior performance of RQ-RDF-3X in comparison with PostgreSQL and RDF-3X in warm-cache too. Note that all these systems in warm-cache use operating system caches. Virtuoso and Jena in comparison use their own caches. Therefore for a fair comparison we report results in warm-cache for systems which use caching mechanism similar to RQ-RDF-3X's.

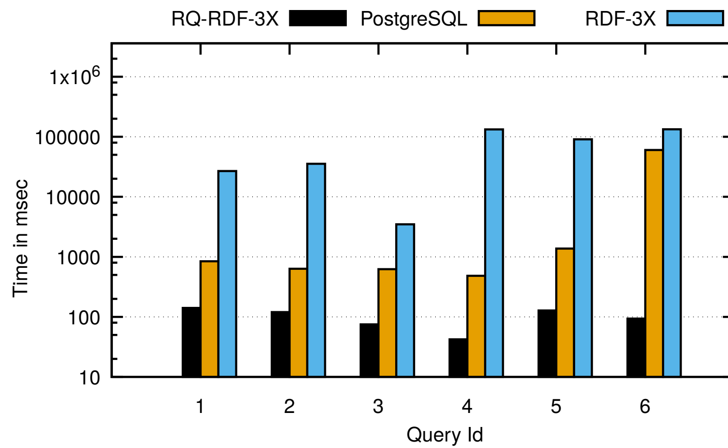
Note that due to logistic issues we did not do experiments on SSD's. We believe that SSD's are not ideal for storing RDF graphs. This is because writes cause the SSD's to wear out easily. This makes SSD's non ideal for RDF because of its pay as you go philosophy.

We acknowledge that results may be better with PostgreSQL 9.2 or later versions because PostgreSQL in these versions started supporting index-only scans.

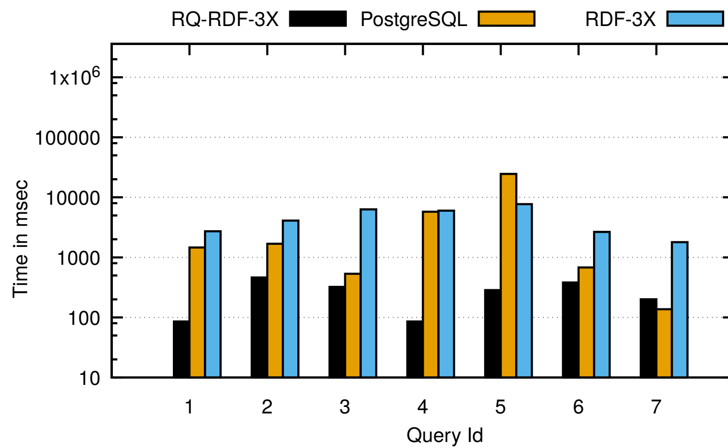
3.5.5 Analysis of Results

RQ-RDF-3X outperforms PostgreSQL, RDF-3X, Virtuoso and Jena-TDB for almost all queries we have considered on both YAGO and DBpedia. The performance advantange of RQ-RDF-3X over RDF-3X is clearly due to the smaller number of star joins that it needs to perform. RDF-3X requires 3 times more joins on DBpedia and 2 times more joins on YAGO than other frameworks namely RQ-RDF-3X, PostgreSQL, Virtuoso and Jena-TDB. Next, PostgreSQL has the obvious disadvantage due to the fact that its indices are non-clustered. For Virtuoso we can not comment on its poor performance as it is closed source and no published work describes its functionality and query optimization techniques. This is clear when we consider the performance of Jena-TDB which has substantially more indices (GOSP, GPOS, GSPO, OSPG, SPOG, POSG, SPO, POS, OSP; G:graph name), and is much closer in performance to RQ-RDF-3X.

Importantly RQ-RDF-3X benefits greatly from its rich indexes, namely: OPSR, OPS, PSO, PRSO, SPOR, POS, ORPS, RSPO, PSOR and OPSR, Particularly all queries in YAGO and DBpedia benchmark greatly benefit from OPSR indexes, note that in addi-

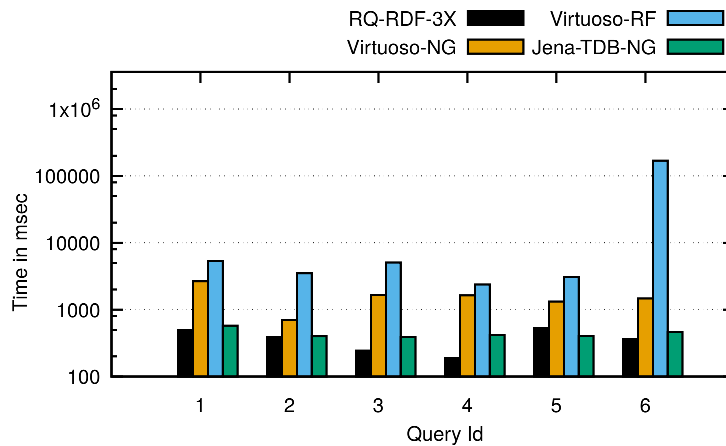


(a) DBPedia

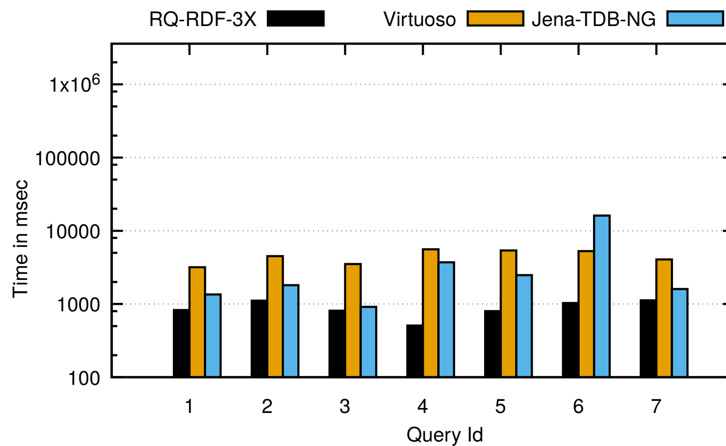


(b) YAGO

Figure 3-5: Query Processing Time of Benchmark Queries (in msec) – without dictionary (NG:Named Graph, RF: Reified Form). Named Graph and Reified Form are W3C recommended ways of storing additional information regarding triples.

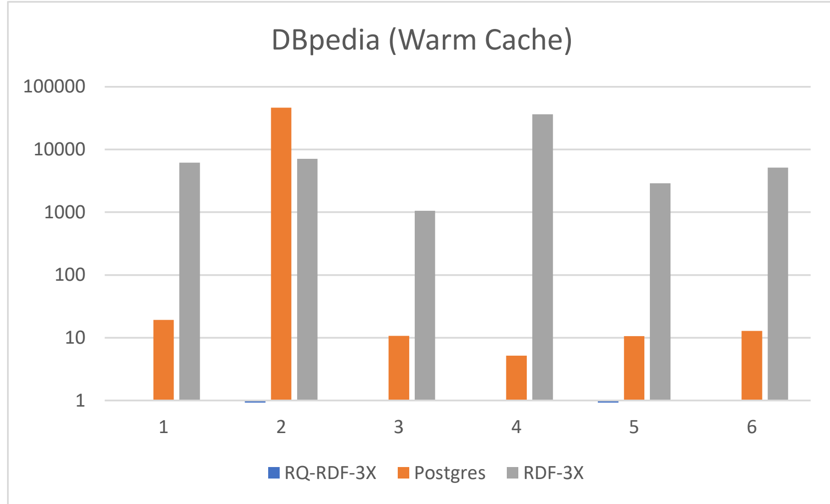


(a) DBPedia

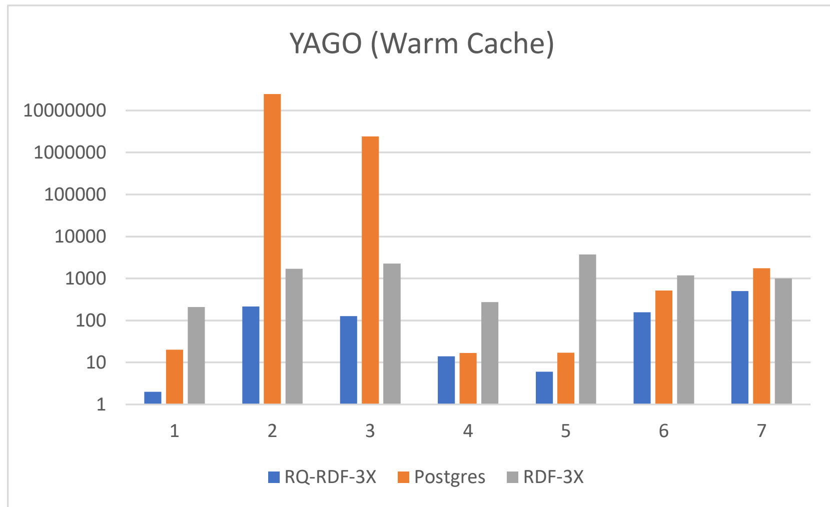


(b) YAGO (Virtuoso-NG consistently outperforms Virtuoso-RF, therefore we show results for Virtuoso-NG only)

Figure 3-6: Total Query Processing Time of Benchmark Queries (in msec) (NG:Named Graph, RF: Reified Form). RF format retains the triple nature, while NG format introduces quads. RF format needs to process many more joins than NG format. RQ-RDF-3X's choice of (1) exhaustive indexing with efficient compression technique, (2) improved query plan generation algorithm helps it outperform the other systems in comparison.



(a) DBPedia



(b) YAGO

Figure 3-7: RQ-RDF-3X, PostgreSQL, RDF-3X performance in warm cache

tion Query 2 also benefits from PRSO index. Query 7 of YAGO and Queries 1, 3 and 5 of DBpedia are accelerated by SPOR index. RSPO index is used for accelerating Query 4. It is noteworthy that reified information connected to triples greatly benefit from bushy plans which the dynamic programming based query optimizer of RQ-RDF-3X generates for all queries of YAGO and Query 1, 3, 5 and 6 of DBpedia. We can not comment on RQ-RDF-3X's poor performance in comparison to Jena-TDB for Query 5 because there is no published work describing its technique and query optimization techniques.

We conclude our experimental evaluation by stating that RQ-RDF-3X efficiently supports SPARQL queries containing reification. Although SPARQL named graph queries are not directly supported by RQ-RDF-3X, they can be easily provided by making simple changes to RQ-RDF-3X's query parser.

3.6 Discussion & Outlook

In this chapter, we presented RQ-RDF-3X, a reification and quad enhanced RDF-3X framework for executing SPARQL queries over massive datasets containing metadata associated with facts. We conducted preliminary experiments over YAGO and DBpedia datasets with a set of queries over reified content. Our comparison with RDF-3X, PostgreSQL as well as Virtuoso and Jena-TDB shows that RQ-RDF-3X is significantly faster than these systems. Notable features of RQ-RDF-3X which cause these performance gains are: (i) exhaustive indexing equipped with a simple yet scalable compression scheme. The proposed efficient set of indexes enable us to efficiently reduce the query processing time by using merge joins. (ii) a smart query processor which generates an optimal join ordering for reified and N-Quad queries. (iii) a selectivity estimator which generates optimal selectivity estimates for reified and N-Quad queries.

3.6.1 Outlook

One may ask whether the techniques presented in this chapter can also be implemented in a distributed setting. One may achieve this by first partitioning the graph and then storing the partitioned sub-graph via RQ-RDF-3X. This is similar in spirit to the solution proposed for

storing triples in a distributed setting, namely, H-RDF-3X [48] and SHARD [80]. However while doing so, the classical partitioning techniques need to be made aware of the annotations attached to triples, such that triples and their annotations are collocated on the same node. Experimentally evaluating such a setup is an interesting open issue.

Chapter 4

Quark-X: An Efficient Top-k Processing Framework for RDF Quad Stores

4.1 Motivation

In this chapter we focus on efficiently evaluating top-k queries over reified RDF data. We were motivated by the following two observations: first, a large fraction of semantic resources such as YAGO and DBpedia contain facts linking to quantifiable values which naturally suggests the use of queries containing ORDER-BY. This is further amplified by the use of reification to attach annotations such as *confidence* and *time*. Secondly, despite the SPARQL recommendation of using ORDER-BY / LIMIT operators, efficient processing of top-k queries with a user-defined ranking function within RDF/SPARQL setting have received limited attention [64, 96].

Take, for instance, the following query from the YAGO dataset: *Find the top ten leaders of countries with the highest rate of inflation and the lowest rate of economic growth, and yet these leaders own luxurious items e.g., jewels, private homes, sports clubs etc..* Since information about such possessions may also be rumours, there is confidence value associated with the ownership statement which is one of the factors in the ranking function. Most RDF processing systems handle such queries by first collecting the results and then sorting them in-memory based on the user-specified function; this approach is not very scalable. On the other hand, commonly used rank-join approaches also can not be effectively ap-

plied due to the extensive combination of joins based on ranking attributes as well as other non-quantifiable predicates in a SPARQL query.

A straightforward way of storing RDF data is using the relational model which enables the use of top-k algorithms designed for relational databases used for RDF data. The property table technique [86, 60, 102] and vertically partitioned approach [4, 85] for storing RDF data are two such techniques which can draw advantage from top-k algorithms proposed for relational databases. However, many researchers have shown that these models of storing RDF in relational databases are not efficient in handling complex query patterns seen in SPARQL queries [101, 69, 70]. Simple triple-store model of storing RDF in relational tables is also not effective for top-k querying since: (a) self-joins incurred by top-k algorithms over a large table are bound to be expensive, (b) either of the two access methods viz., the sorted or the random access [51] are not suitable because of unsorted nature of quantifiable values in RDF and the complex pattern matching model of SPARQL queries (cf. Section 4.5.2).

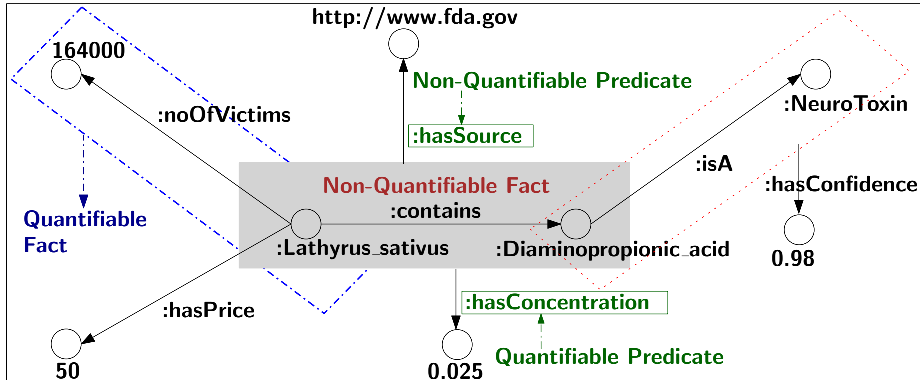
Quark-X overcomes these limitations by introducing a combination of adaptively switching block-wise sorted and random accesses based on their cost estimates along with its semantic encoding of identifiers to improve locality of reference of subjects associated with quantifiable predicates. To the best of our knowledge, these features are not explored until now in relational top-k processing systems as well as RDF quad stores.

4.2 Organization

The rest of this chapter is organized as follows. Section 4.3 introduces preliminaries. Section 4.4 includes related work done in the field of RDF stores, Databases, and Information Retrieval. Quark-X is explained in detail in Section 4.5 and 4.6 – these sections explain Quark-X's indexing and query processing subsystems respectively. Update management is explained in section 4.7. Section 4.8 explains the implementation details of Quark-X. Section 4.9 explains our evaluation framework. Section 4.10 includes our experimental evaluation.

4.3 Preliminaries

4.3.1 Running Example




```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns\#>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
#@ <id1>
:Lathyrus_sativus :contains :Diaminopropionic_acid.
#@ <id2>
:Lathyrus_sativus :noOfVictims "164000"^^xsd:int.
#@ <id3>
:Lathyrus_sativus :hasPrice "50.0"^^xsd:double.
#@ <id4>
<id1> :hasSource <http://www.fda.gov>.
#@ <id5>
:Diaminopropionic_acid :isA :NeuroToxin.
#@ <id6>
<id5> :hasConfidence "0.98"^^xsd:double.
#@ <id7>
<id1> :hasConcentration "0.025"^^xsd:double .
```

Figure 4-1: Example RDF knowledge graph

```
SELECT ?product ?chemical ?source
  ((f1(?countVictims) + f2(?price)) * f3(?conc) * f4(?conf) as ?rank)
WHERE {
  ?product :noOfVictims ?countVictims.
  ?product :hasPrice ?price.
  ?reif rdf:subject ?product; rdf:predicate :contains; rdf:object ?
     chemical.
  ?reif :hasConcentration ?conc.
  ?reif :hasSource <http://www.fda.gov>;
  ?reif1 rdf:subject ?chemical; rdf:predicate :isA; rdf:object ?toxin.
  ?reif1 :hasConfidence ?conf.
} ORDER BY DESC(?rank) LIMIT 2
```

Figure 4-2: Running Example Query

For ease of exposition, we use reified RDF listing shown in Figure 4-1(in a format popularized by YAGO) as a running example throughout this chapter. For illustration, we have used synthetic numbers for estimated affected population. The top-k SPARQL query

over the running example snippet that we use is given in Figure 4-2. This query *finds top-2 food products which contain toxins, ranked based on number of victims, product's cost and toxin's concentration in the product. The toxicity of a chemical compound has associated confidence value which needs to be included in the ranking function.*

For the rest of the chapter, we adopt the following terminology: We call the query patterns containing quantifiable predicates as *quantifiable query patterns*, and the remaining query patterns are called *non-quantifiable query patterns* (abbreviated as *NQP*). Subjects of quantifiable query patterns are called *quantifiable variables*. The example query in Figure 4-2 has (?product <hasPrice> ?price) as a quantifiable query pattern. Also, the facts containing quantifiable values in our knowledge base are called *quantifiable facts*.

4.4 Related Work

In this section, we briefly discuss and contrast our work with the related work from RDF and relational databases as well as from information retrieval. We will limit ourselves to the discussion on top-k query processing, and direct the interested reader to W3C recommendations on supporting annotations and reification in RDF (cf. Section 4.3 in [1]), and their usage in real-world semantic datasets such as YAGO [47].

RDF/SPARQL: Although modern RDF systems such as Virtuoso and Jena have fairly sophisticated SPARQL query processing, their approach to top-k queries is to collect all the results of a query, sort them or use an in-memory priority queue to compute top-k answers. This approach is expensive as the query engine needs to process all solutions, even though only k of them are requested by the user.

The other approach involves *early termination* in an explicit manner. We are aware of only a few such approaches in the context of SPARQL – albeit only over triple-stores – which we discuss next. The SPARQL-RANK framework proposed by Magliacane et al. [64] makes use of different index permutations used in native triple-stores for fast random access during top-k processing, and applies early-termination criterion. They propose an algorithm, which requires the left-most index used in the join plan to be sorted based on the ranking function, and then it randomly probes the right-side index. Thus, when the

right-side index is large, the performance of rank join suffers. Also, the requirement for the left-side index to be sorted based on ranking function makes it unsuitable for arbitrary user-defined ranking functions.

In another framework introduced by Wang et al. [100], quantitative entities in the RDF dataset are separated out into an *MS-tree* index. In the first step of query processing, candidate entities are located using the MS-tree index that are then used as *seeds* for performing breadth-first (BFS) traversals over the graph to find matching sub-graphs. If the query requires only a few highly correlated predicates, the algorithm may end up storing many unnecessary nodes in the queue, making the retrieval of the first entity possible only after several iterations. On the other hand, our approach does not require unrelated predicates and entities to be stored together. However, we did not empirically compare our work with this work as it is still unclear how to apply their BFS based candidate generation phase on reified databases.

Relational Databases: Hash Rank-Join (HRJN) [52] and Nested Loops Rank Join (NRJN) [49] represent the state-of-the-art relational rank-join algorithms. HRJN [52] is based on ripple join algorithm [41]. It maintains two hash tables in-memory for storing the input tuples seen so far, the stored input tuples are used for finding join results. These results are in-turn fed to a priority queue, which outputs them in the order specified by the ranking function. NRJN is similar to HRJN except that unlike HRJN it does not store input tuples, but rather follows a nested-loop strategy. However for RDF data, SPARQL-RANK showed experimentally that it outperformed HRJN [64]. The performance gain was attributed to the unsorted nature of numerical attributes present in indexes build by RDF engines. We show in a later section that we outperform SPARQL-RANK by a large margin by targeting precisely the numerical attributes once again. Hence, we did not compare Quark-X with algorithms like HRJN for relational databases.

Information Retrieval (IR): *Block-max* index structure proposed by Ding et al. [32] is one of the effective approaches proposed in the IR community for retrieving top-k documents efficiently. The block-max index stores documents sorted by document ids in a block-partitioned inverted index. In contrast, the identifiers in our approach are sorted by scores. Our approach is somewhat similar to impact-layered indexes, where the posting list

is divided into layers such that the higher layer posting list has a lower score than the layer below it. Additionally, top-k ranking algorithms proposed in the IR community are different from those proposed in the RDF and relational databases community – in IR, bindings of just one variable needs to be retrieved, whereas in relational database as well as RDF setting, bindings of multiple variables need to be retrieved, leading to unsorted orders.

4.5 Indexing for Quantitative Facts

Quark-X pursues the exhaustive indexing approach for RDF databases made popular by RDF-3X [69], and additionally develops an indexing framework for efficiently answering top-k queries. This section presents the details of the indexing framework in Quark-X for top-k processing. It consists of:

1. two indexes, an in-memory synopsis index called S-index and a bucket-ordered quantifiable Q-index, on *quantifiable facts* in the database,

2. a semantic re-encoding strategy of identifiers which utilizes the information gathered during indexing to remap the ids involved in quantifiable facts.

The index creation process, and the semantic re-encoding of identifiers is illustrated in Figure 4-3.

Similar to most state-of-the-art RDF engines, Quark-X encodes typically long URIs and strings in RDF as fixed-length numerical identifiers and maintains these mappings in a dictionary structure. The size of the dictionary is further reduced by identifying frequently occurring common maximal prefixes among URIs in the database which are encoded first into integers. These prefixes are represented using their encodings in URIs they occur in, and these prefix encoded strings are stored in the dictionary. For example, if the URI prefix http://yago-knowledge.org/resource is mapped to integer 1, then the URI http://yago-knowledge.org/resource/contains is prefix-encoded to 1/contains and then stored in the mapping dictionary.

4.5.1 Quantifiable Indexes

Quark-X introduces two special indexes for quantifiable facts in the database, designed to help in efficient early pruning of results and in preserving interesting orders.

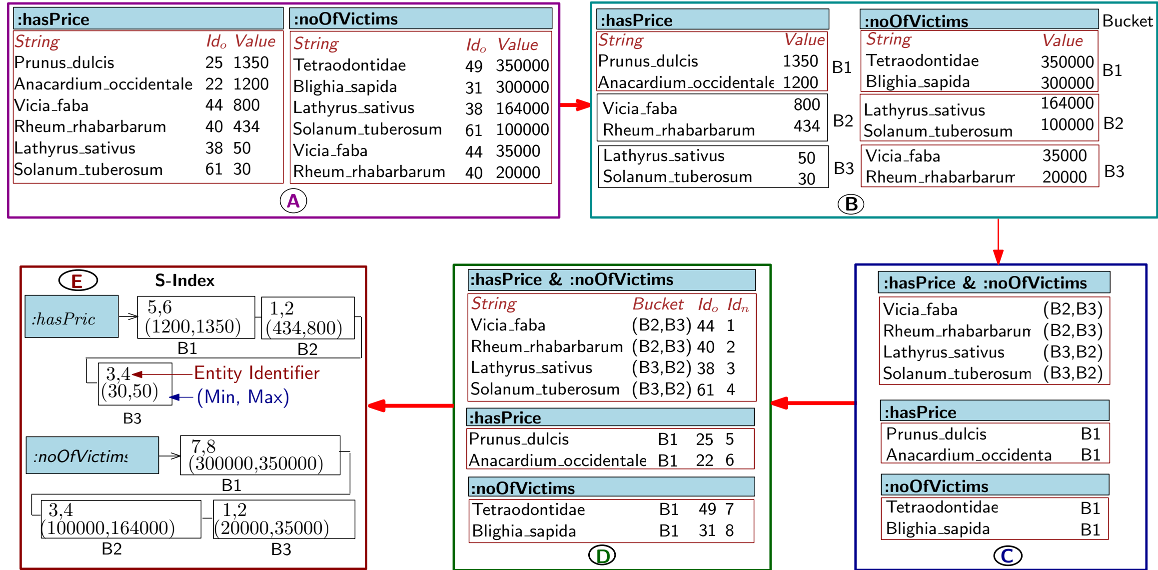


Figure 4-3: Summarized Index Creation (ⒶRaw Numerical Facts Ⓑ Bucket Creation (B*i*=Bucket) Ⓒ Characteristic set-Bucket Mapping Ⓓ Assign new collocated ids (Id_o: old identifier, Id_n: new identifier) Ⓔ S-Index) [1]

S-index The S-index is an in-memory synopsis index which stores for each quantifiable predicate a statistical metadata summarizing all the entities and their associated quantifiable values in a compact form. In particular, histogram-based information is maintained for each quantifiable predicate to describe the value distribution. For simplicity, we employ an equi-depth histogram over the range of values for each quantifiable predicate. We can easily employ other forms of histograms as required. Associated with each bucket of the histogram, we maintain:

- the lower and upper values, $min[b]$ and $max[b]$, defining the range of quantifiable values covered by the bucket b,
- the set of *subject ids* associated with the quantifiable value falling within this bucket range.

Although it is possible to store the set of subject ids as a Bloom filter, we chose not to do so as it can not be used to maintain *sorted orders* within each bucket, which, as we show later, can be used to further speed up top-k processing.

We illustrate the process of constructing the S-index in Figure 4-3 for our running example data from Figure 4-1. The S-index constructed at this stage is called a *temporary S-index*. In this example, the predicate *:hasPrice* has values in the range of [30, 1350], and entities associated with the predicate *:hasPrice* have been divided into 3 buckets. Similarly, the predicate *:noOfVictims* too has 3 buckets. For brevity, the buckets for the other quantifiable predicates, viz., *:hasConfidence* and *:hasConcentration*, are not shown. The semantic encoding strategy we explain next necessitates re-mapping of subject ids in the S-index as shown in step E.

Q-Index Since S-index buckets only store upper- and lower-bounds of scores in the bucket, we maintain an additional B+-tree index called *Q-index* on *predicate id*, *bucket number*, *subject id* and the corresponding *quantifiable value*. This index is used for finding the exact numerical values associated with a subject and a (quantifiable) predicate. Once the candidate buckets are determined using S-index, it is quite straightforward to use Q-index to retrieve quantifiable values in the order of their subject ids.

4.5.2 Semantic Encoding of Identifiers

In traditional RDF engines the URIs and strings in the database are encoded typically in their order of appearance in the database or hashing. Both these techniques have been shown to be suboptimal for compressibility and, more importantly, for efficient join processing [91]. In top-k ranking join queries, the problem is further exacerbated by the requirement that identifier assignment should not only help the classical equi-join processing, but also preserve the ordering over quantifiable values.

To address this, during *S-index* construction the RDF terms are re-encoded and remapped in the dictionary as well as the S-index buckets. These new encodings are derived through the *soft-schema* present in the RDF data which stems from the fact that multiple RDF statements are used to describe the "properties" of a subject. To illustrate this, consider the following query patterns from query given in Figure 4-2 – (?product :noOfVictims ?countVictims), (?product :hasPrice ?price) – both the patterns describe a food product, its price and the number of victims through the predicates *:noOfVictims*

and *:hasPrice*. These two predicates are also strongly correlated in the database since this pairing of predicates holds for only food products. Many entities can likewise be uniquely identified by the predicates connected to them. This observation has been used earlier for improving the cardinality estimates of RDF queries [68], where they name the set of predicates connected to an entity as its *characteristic set*.

The *semantic encoding scheme* employed by Quark-X can be understood by using a subset of the example knowledge base shown in Figure 4-1. In that knowledge base, subjects like *Rheum_rhabarbarum*, *Vicia_faba*, etc. are described by two quantifiable predicates – *:noOfVictims* and *:hasPrice*. On the other hand, subjects like *Prunus_dulcis*, *Blighia_sapida* etc. have information pertaining to just one quantifiable predicate, either *:noOfVictims* or *:hasPrice*. Resulting in the following 3 characteristic sets: *(:noOfVictims and :hasPrice)*, *(:noOfVictims)* and *(:hasPrice)*. Next we present the pseudo code of semantic encoding scheme aka characteristic set generation algorithm of Quark-X in Algorithm 2. The algorithm receives as input SPO index and outputs the computed characteristic sets as well as the mapping between characteristic sets and subjects connected to characteristic set. Note that the characteristic sets are precomputed by making a pass on the data.

Algorithm 2 Pseudo Code for computing Characteristic Set

Input SPO index
Output characteristicSets, mapCharacteristicSetSubjects

```
 1: first=true
 2: characteristicSets = emptyset
 3: mapCharacteristicSetSubjects = emptyset   ▷ map between characteristicSet and subjects in it
    FOR row.s, row.p, row.o IN SPO
 4: predicateSet = emptyset
 5: subjectSet = emptyset          ▷ collect predicates connected to Subject in predicateSet
 6: while (row.s == previousRow.s) or (first == true) do
 7:     predicateSet.add (row.p)
                                   ▷ add to characteristic set the newly found predicate set
 8:     characteristicSets.add(predicateSet)
 9:     mapCharacteristicSetSubjects.add[predicateSet].add(row.s)
10: end while
```

Using the temporary S-index, the subjects that have quantifiable predicates belonging to a characteristic set are assigned their corresponding buckets (illustrated in steps A through

C in Figure 4-3). While doing so, subjects falling within each characteristic set are ordered according to their predicate values, instead of subject values. Next, ids are assigned to the sorted subjects in a manner so that subjects belonging to the same characteristic set and bucket are allocated consecutive ids. This is shown in Figure 4-3, step D. Then the ids present in the *temporary S-index* are re-mapped using the mappings generated at the end of step D. The resulting semantically encoded S-index is shown in the step E of Figure 4-3. Next we present the pseudo code of summarized index creation in Algorithm 3.

Algorithm 3 Pseudo Code for Summarized Index Creation

```
    Input PSO index, characteristicSets, mapCharacteristicSetSubjects, bucketSize
    Output SIndex
 1: map_sub ← empty                          ▷ map of subject TO predicate, object, bucket
 2: bucket ← 1
 3: FOR row.p, row.s, row.o IN PSO
 4:     list ← empty                         ▷ List containing bucket, predicate and object
 5:     FOR iter IN bucketSize
 6:         list.add(bucket, row.p, row.o)
 7:     map_sub.add(row.s,list)              ▷ create a map of subjects instead
 8:     bucket++
 9:     if(row.p != previousRow.p)
10:         bucket ← 1
                                             ▷ Assigning identifiers to Subjects
11: Identifier ← 0
12: SIndex ← empty                           ▷ list of predicate, bucket, subject, object
13: FOR characteristicSetSubject IN mapCharacteristicSetSubjects:
14:     FOR subject IN characteristicSetSubject:
15:         Subject.id ← identifier
16:         identifier++
17:     FOR predicates, subjects IN characteristicSetSubjects:
18:         FOR predicate IN predicates:
19:             FOR s IN subjects:
20:                 SIndex.add(predicate, map_sub[s][0], map_sub[s][1], map_subj[s][2])
21: Sort (SIndex)      ▷ Sort in lexicographic order Predicate, Bucket, Subject, Object
```

[1] Example data manually extracted from the following two books:
1. K.R. Natarajan. India's poison peas. Chemistry, 49(6), 1976. (obtained from the FDA Poisonous Plant Database).
2. David R. Briggs. Naturally-occurring toxicants in some nutritionally significant plant foods and fish.

4.6 Query Processing

Now, we turn our attention to the top-k query processing in Quark-X and describe how our quantifiable indexes are utilized to evaluate queries efficiently. Since S-index is an in-memory synopsis index which summarizes the quantifiable value distribution, we aim to use it greedily for join-ahead pruning early on in the plan. The query compiler simply extracts the quantifiable predicates from the query before generating a cost-based query plan on non-quantitative patterns. The S-index based join-ahead pruning over quantifiable query patterns is added subsequently so as to generate candidate ids while maintaining interesting orders.

Note that S-index buckets are processed bucket at a time in the order of quantifiable value, and the same sequence is retained in subsequent (non-quantifiable) joins as well. We call this as *S-index filtering of non-quantitative index*, and it plays an important role in the query evaluation pipeline we present next.

The query processing in Quark-X proceeds in three stages: first, the *S-index join* is performed using early-termination over S-index synopsis to generate candidate ids with lower and upper bounds on the associated quantifiable values. For exposition, *S-Index* join between predicates *:noOfVictims* and *:hasPrice* is shown in Figure 4-4 $E1'$ this join gives values (3,4). Next, these candidate ids are used for join-ahead pruning over non-quantifiable query patterns in *NQP-join* or non-quantifiable query pattern joins. This join ahead pruning is shown in Figure 4-4 $E2'$, where PSOR is the index over the non-quantifiable query pattern *?reif ?product ?contains ?chemical*. The join after respective pruning of indexes is shown in Figure 4-4 E', with its succinct view shown in Figure 4-4 E. Finally, the *SQ-index join* is performed where the final list of top-k results with their complete order-by scores is generated. Figure 4-4 F and E shows the results of join from E being passed to Figure 4-4 F (aka Q-Index) for retrieving values of top-k tuples. We describe each of these stages next.

The entire query processing flow is illustrated for our running example in Figure 4-4. In the rest of this section, we regularly reference various named parts of this figure for the ease of exposition.

4.6.1 S-index Join

Next we explain S-Index Join which we perform using **neighborhood expansion** [105]. Assume that the scoring function is monotonically non-increasing, and denote it by *f(p,q,$\cdots$)* over a set of predicates *p,q,.....* Let $< B_p^1, B_p^2, ..., B_p^m >$ and $< B_q^1, B_q^2, ..., B_q^n >$ be the S-index buckets for predicates *p* and *q* respectively. As required by the ranking function, these buckets are accessed in the decreasing order of their scores, and the candidate list of join entities is generated as follows: first process the pair of first buckets from each S-index, that is, B_p^1 and B_q^1. If there is a common identifier among the entity inverted lists of these two buckets, then this pair is stored as $(B_{i_1}, B_{j_1}\ )$. The next best pair of buckets are found by considering the combinations containing at most one next bucket from each predicate – that is, (B_p^1, B_q^2) and B_p^2, B_q^1 . These are pushed to a priority queue in order to extract the best pair which has, at least, one common entity. The process is continued by expanding the bucket pairs that are considered until k candidate results are retrieved and there is no combination of buckets in the priority queue that has a maximum score (or minimum score for non-decreasing score function) more than the k-th result. Note that this in itself is not an entirely novel technique – similar ideas have been explored in distributed top-k processing [67], scan depth estimation [72], and others for enabling early termination [51]. We use S-index in a more effective way by combining the quantifiable value sorted candidate ids generated by the index with non-quantifiable joins. Recall that Quark-X employs semantic encoding of identifiers (cf. Section 4.5.2), which ensures that identifiers belonging to a characteristic set to be clustered. This works synergistically with S-index to provide increased sequential scans on other indexes.

In our running example from Section 4.3.1, the S-index join on quantifiable predicates :hasPrice and :noOfVictims, retrieves the following bucket pairs: $L1' = (B2, B3)$ with join results $(1, 2)$ and $\max$-score bound 28×10^6 and $L2' = (B3, B2)$ with join result $(3, 4)$ with $\max$-score bound 8.2×10^6. The left-bottom block of Figure 4-4 with label E_1' depicts the S-index join result $L2'$. Observe that the identifiers in $L2'$ – *viz.*, $(3, 4)$ – are also present in the PSOR index (PSOR stands for an index sorted lexicographically in the order of **P**redicate, **S**ubject, **O**bject, **R**eification ids) for the predicate :contains which is a

non-quantifiable part of the query, as illustrated in the grey block E_2' of Figure 4-4. Such a collocation significantly speeds up the query processing by preferring range-scans over random probes.

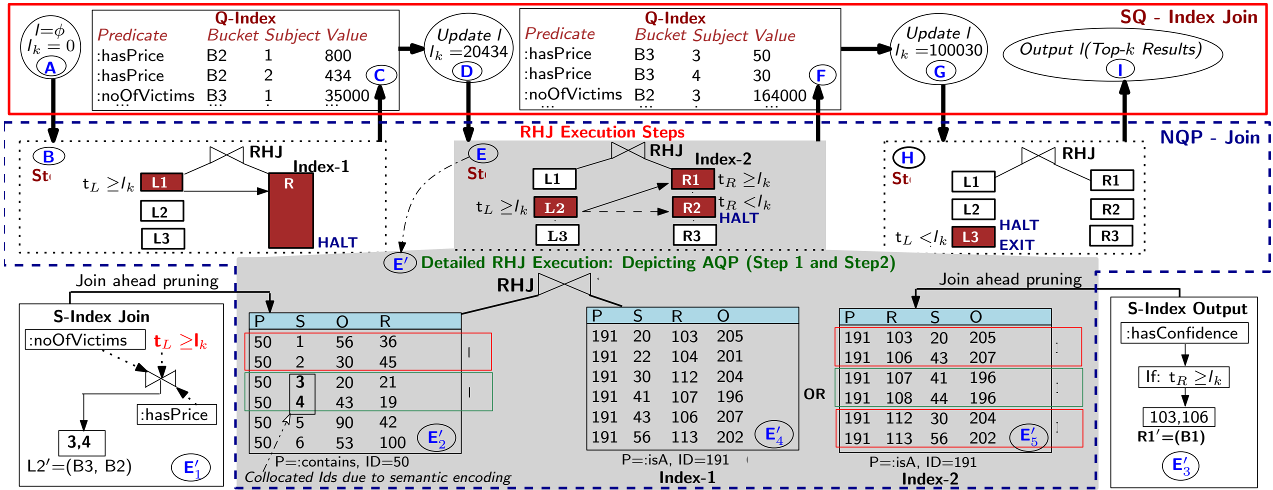


Figure 4-4: Quark-X Query Processing (AQP: Adaptive Query Processing; Reading order from label A to I)

4.6.2 Non-quantifiable Predicate Joins

We now turn our attention to the join processing between non-quantifiable query predicates using the S-index join results for join-ahead pruning. Note that S-index induced score computation takes place on *quantifiable variables* – i.e., subjects with quantifiable predicates in the query. Therefore, by treating ordering over quantifiable variables as an *interesting order* [52], we can generate rank-aware plans which can exploit the S-index results.

Generation of such plans is quite different from the classical top-k ranking systems used in RDF commercial stores such as Virtuoso and open-source systems such as Jena. These systems end up materializing all results of a join before generating the final ranked list. To remedy this, we propose a novel rank-join operator called *Rank-Hash Join (RHJ)* which we describe next.

Algorithm 4 RHJ Algorithm

```
 1: left-bucket-no=0
 2: right-bucket-no=0
 3: scanLeft=true                          ▷ To decide whether to scan from left or right side
 4: while true do
 5:     if scanLeft then                                                ▷ Scan from left side
 6:         if not buildHashTableLeft(left-bucket-no) then                       ▷ left empty
 7:             if checkLeftHaltCond() then                        ▷ Halt condition satisfied
 8:                 return false                                              ▷ Halt and exit
 9:             else
10:                 left-bucket-no ← left-bucket-no+1
11:                 scanLeft ← true
12:             end if
13:         else
14:             right-bucket-no ← 0
15:             scanLeft ← false
16:         end if
17:     else                                                           ▷ Scan from right side
18:         if not probeRight() then                        ▷ No tuples retrieved from right
19:             if checkRightHaltCond() then
20:                 left-bucket-no ← left-bucket-no+1
21:                 scanLeft ← true
22:             else
23:                 right-bucket-no ← right-bucket-no+1
24:                 scanLeft ← false
25:             end if
26:         else
27:             hashJoinLeftRight()                ▷ join & pass results to other operators
28:             right-bucket-no ← right-bucket-no+1
29:             scanLeft ← false
30:         end if
31:     end if
32: end while
```

Rank-Hash Join (RHJ) algorithm

The key idea of our Rank-Hash Join is to use the results from the S-index join step to adaptively decide if the right-side index of the join has to be *probed* or *to be scanned* fully. It does so in a manner similar to a classical hash-join, but differing in not requiring the entire result of the left hand side of the join to be in hash table. Instead, RHJ needs to maintain only the ids from a single bucket in the S-index in the hash table. Since we process all elements in the hash-table completely before pulling the next bucket from the underlying S-index join, it is guaranteed not to miss any results.

The RHJ algorithm builds hash-table on the *left-side* using tuples filtered through from the S-index join stage. For retrieving tuples from *right-side* of the join, the following two index choices exist: (1) index on the sorted order of joining variables, which we term as **index-1**, that can enable efficient disk skips by using the ids retrieved from left hand side of the join in a *sideways information passing* optimization [70], and (2) index on the sorted order of quantifiable variables, which we term as **index-2**, over which we can limit the number of pages required to be scanned by utilizing the entity ids from the underlying S-index.

These indices are adaptively selected, triggered by a condition based on the cost calculated using a cost model on these index alternatives which estimates the number of disk pages required to be scanned.

The cost model for index-1 Due to its sorted order, the number of pages required to be fetched from this index in the worst case is equal to the number of tuples retrieved from left sub-plan. Hence, the number of tuples satisfying the left sub-plan is taken to be the estimate of the cost of this candidate plan. Specifically,

$$C_1 = N \times \prod_{i \in qp} S_i \times S_{nqp},$$

where, S_{nqp} is the selectivity of the non-quantifiable predicate in the left sub-plan; S_i denotes the selectivity of *i*-th quantifiable predicate in left sub-plan; N is the number of elements in a bucket (note that we use equi-depth buckets), and qp are the quantitative predicates in the query.

The cost model for index-2 As a result of semantic encoding, the ids within a bucket are stored consecutively on a disk page (or in adjacent disk pages) which can be retrieved with only one seek. Therefore, in the worst case the number of pages required to be fetched from this index is equal to number of buckets whose score is above l_k where l_k is the score of the k-th best scoring element seen so far, i.e.,

$$C_2 = \text{ number of buckets having score greater than } l_k.$$

After estimating the cost of each candidate plan, the query re-optimizer chooses the plan with smallest cost amongst the two choices.

Note that, due to *S-index filtering of non-quantitative index*, our algorithm incurs zero-cost for switching indexes (plans) at *"materialization" points* [31], i.e. decision points where plans are changed. In RHJ, materialization points are points at which next bucket is retrieved from leftmost index. For example in our running example, points at which buckets L1 (left-middle with label B), L2 (middle with label E), L3 (right-middle with label H) are retrieved from the leftmost index are materialization points. We believe, there is only one prior work by Ilyas et al. [50] in DBMSs, which also uses adaptive query processing(AQP) for efficient top-k retrieval. However, unlike our proposal, the state-saving techniques proposed in [50] wastes a significant amount of already done work while switching plans (with state-of-the-art rank join algorithms like HRJN, NRJN, etc).

Our novel RHJ algorithm borrows ideas from classical hash join and adaptive query processing (AQP) research and applies it in the context of RDF/SPARQL processing, to the best of our knowledge, for the first time.

RHJ Stepwise Description: Now we go through the workings of the RHJ algorithm illustrating it step-by-step based on our running example from Section 4.3.1. We refer to the steps illustrated in the schematic diagram in Figure 4-4, with its pseudo code shown in Algorithm 4. In the rest of this section, we use l to denote the list with top-k results, with l_k denoting the score of the k-th result. For ease of exposition, our description is ordered using the labels used in Figure 4-4 starting from label A at the top-left of the figure until label I at the top-right of the figure.

A: Initialize the algorithm with $l = \emptyset$ and $l_k = 0$.

B: Left and right-side are joined using *classical hash join* where:

- *Left side:* S-index join pulls the first bucket $L1' = (B2, B3)$, which is used as a filter over P=:contains index, which helps create candidate result block $L1$.
- *Right side:* Uses index-1 as illustrated in the grey portion of the figure with label E'_4. The **index-1** (PSRO ordered index) is preferred here over the **index-2** (PRSO-ordered) alternative based on the cost considerations described above. In particular, we observe that the ids 56 and 30 in column *O*, are present only in bucket $R3$ of **index-2**

which necessitates retrieval of all pages from $R1$ and $R2$ from disk, making it extremely inefficient. In comparison, **index-1** is used, it can utilize the sideways information passing optimization since this index has the joining variable in sorted order. This results in only 2 disk seeks in the worst-case (assuming tuples corresponding to 56 and 30 are stored on separate disk pages).

C: Find quantifiable values from *Q-index* using candidate buckets retrieved from *S-index* in B.

D: The retrieved quantifiable values are aggregated using ranking function in top-middle block with label D. The resulting k tuples with highest score are then stored in *l*. The kth maximum scoring element l_k is passed to NQP-join stage – it helps in performing early-termination check by verifying that, *there is no combination of buckets that has a maximum score (for descending) more than the k-th result*.

E: Left and right are again joined using *classical hash join* in middle block with label E.

• Left Side: *S-index Join* (left-bottom block with label E'_1) pulls next bucket $L2' = (B3, B2)$ (shown in left-bottom with label E'_1), which is in-turn used as filter over P=*:contains* index (grey block with label E'_2), which helps create block $L2$

• Right-side: Uses index-2 as illustrated in the grey block with label E'_5. Note, here index-scan is speed-up by pulling buckets from S-index (shown in right-bottom block with label E'_3).

We explain next the reason for choosing **index-2** (PRSO). From grey block with label E'_2, we observe block $L2$ requires 2 disk seeks for the two identifiers 20 and 43 when **index-1** (PSRO) – shown with label E'_4 in grey block – is used. However, using l_k (score of kth result), we observe that only bucket $R1$ (shown with label E'_5) is needed from the right side index. Thus, it is beneficial to only scan $R1$ (requiring 1 disk seek) of **index-2** instead of performing a full scan on the **index-1**.

F: Find quantifiable values from *Q-index* for the new set of candidate buckets retrieved from *S-index* in E.

G: The retrieved quantifiable values are again aggregated using ranking function. The resulting k tuples with highest score are then stored in *l*. The kth maximum scoring element l_k is passed to NQP-join stage – which helps in performing early-termination

check (described in D).

H: When pulling next bucket from S-index Join, we find early-termination condition is satisfied (as score of k-th element is greater than maximum score (t_L) of yet to be retrieved buckets i.e. $l_k > t_L$), hence the algorithm terminates.

I: Outputs top-k scores stored in list l.

SQ - Index Join

The S-Index join stage passes bindings of entity ids along with their corresponding bucket numbers to NQP-Join stage, for quantifiable variables appearing in sorted order in index scans. For finding the explicit numerical values of other quantifiable variables which appear in unsorted order in index scans, we first find their corresponding bucket numbers using S-Index.

Using the bucket number retrieved either using the approach described above or using S-Indexes, we find the exact numerical values using Q-Index. The obtained numerical values are stored in a list l, and the list is in-turn used to find the score of the last element l_k with respect to the ranking function.

4.7 Update Handling

Quark-X handles updates similar to the other state-of-the-art RDF management system like RDF-3X and similarly assumes that updates are mostly insertions, are far fewer compared to queries and can be batched together. During batch updates, Quark-X creates differential indexes for S-index, Q-index, and permutations of SPOR which are stored in main memory and are merged with the main index at suitable intervals. For recovering in case of failure, differential indexes are additionally stored in log files on disk.

During query processing, these indexes, and the main Quark-X indexes undergo merge-join; however, being small, these incur little overhead. While re-assigned ids in main indexes helped us in clustering together semantically similar ids on disk, thereby reducing disk seeks, the differential indexes reside entirely in main memory and can avoid disk seek altogether. Id-assignment is therefore not done in these indexes and is deferred until they

are merged with the main index at which point all ids are reassigned afresh.

4.8 Implementation Details

In this work, we assumed system architecture of a quad-store that provides with a mapping dictionary between ids and strings. In line with this, we have used RQ-RDF-3X [56] as a baseline framework for our implementation. **RQ-RDF-3X** is a quad store with exhaustive clustered B+-indexing of all permutations of SPOR. RQ-RDF-3X stores all the 24 permutations over quads, many of which are superfluous to begin with, thus, we dropped the following: 1) permutations where R appeared in the third position (e.g. SPRO) were removed because R is always unique, therefore, a sorted order of O for SPR does not provide additional help during joins; 2) all permutations where R appeared in the first place (except RSPO) were removed since sorted order of R, always unknown, does not give any advantage for speeding-up range scans. But when R is a joining variable, its sorted ordering helps perform joins efficiently; however, only one index (RSPO) is sufficient for this purpose.

After explaining RQ-RDF-3X, now we explain how processing data in *blocks* improved query execution time of Quark-X. At the beginning of query processing, in the S-Index join stage, the candidate set of buckets required to be retrieved is equal to the set of all plausible buckets. Accordingly, the S-Index aggregates buckets until a fraction of k entities are retrieved and these aggregated buckets are then passed to the NQP-Join stage. As the query execution proceeds, a stage comes when it is possible to restrict the candidate set of buckets based on their score and the score of the k-th result. At this stage, we divide these candidate set of buckets into fractions. It is evident that upon incremental processing, the threshold score would get tighter, which helps in early termination. We also access data in blocks during the SQ-Index join stage – where we keep materializing the results obtained from NQP-Join until retrieving k results. SQ-Index join then processes these materialized results together. After k results have been retrieved, SQ-Index join processes together aggregate buckets passed by S-Index join stage to NQP-Join. Thus block-wise access helps in amortizing cost of index scans over a range of results.

Next we explain how aggregating buckets until a fraction of k entities are retrieved helps in accelerating query performance with the help of an example. Consider the running query which aims at retrieving top-10 low pricing products. For explanation we begin by choosing 10 low priced products from S-Index of prices. Since S-Index in our running example contains 2 elements in each bucket. Therefore we need to aggregate 5 buckets. These products are then looked up in non-quantitative index to find the chemicals these products contain. Now note that not all products have information pertaining to chemicals in the knowledge base. Therefore many of these products are pruned. Choosing 10 products helped us perform sequential scan on the non-quantitative index to retrieve chemicals contained in them.

Next consider we retrieved two products which also had information pertaining to chemicals from the previous step. Therefore we opt for retrieving 8 products next from S-Index *price*. Moving in this fashion for performing fast sequential scans we keep aggregating S-Indexes until k fraction of entities are retrieved.

4.9 Evaluation Framework

Quark-X is implemented in C++, compiled with g++-4.8 with -O3 optimization flag. All experiments were conducted on a Dell R620 server with Intel Xeon E5-2640 processor @ 2.5GHz, 64GB main-memory, RAID-5 hard-disk with 3TB effective size. In our experimental evaluation, we report cold-cache timings after dropping filesystem caches using: `echo 3>/proc/sys/vm/drop _caches`, and warm-cache numbers by repeatedly running the query processor with the same query 5 times, and taking the average of last 3 runs.

We evaluate against two research prototypes – RDF-3X and SPARQL-RANK, and two state-of-the-art commercial systems – Jena-TDB-2.13.0 and Virtuoso 7.2.

Among the research prototypical competitors we use, the inability of RDF-3X to compute only the top-k results, and the higher number of random seeks incurred by the query processing algorithm of SPARQL-RANK make them overall much slower in comparison to Quark-X. Note that the overheads due to random seeks are further exacerbated in the

	YAGO	DBpedia-RF
Size of input files (in ttl)	46 GB	74 GB
# of quads	$473,271,482$	$668,867,020$
# of numerical quads[2] containing only confidence	$236,635,830$	$334,433,512$
# of numerical quads without confidence	$569,558$	$197,530$

Table 4.1: Sizes of Datasets and Databases (RF: Reified Form)

straight forward extension of these algorithms to quads due to the additional joins with metadata like confidence values. On the other hand, Quark-X outperforms the commercial systems which natively support quads by smartly encoding the ids using soft-schema embedded in the data to improve locality of reference in its index scans.

4.9.1 Datasets

Despite the abundance of a number of performance benchmarks for RDF/SPARQL query processing [40, 20, 83], our evaluation could not use them since all of them are designed primarily for triple-stores, with no queries using reification and named-graphs in their query set. Therefore, we decided to work with a suite of top-k queries which we designed over two large real-world datasets which extensively use named-graphs or reification. The first dataset we use is DBpedia 3.7 [19], which contains facts extracted from Wikipedia, with provenance expressed in the form of named graph. The second dataset, YAGO [47], contains facts extracted from Wikipedia and combined with GeoNames and WordNet. It encodes additional information –e.g. confidence score, time, spatial location, and provenance– with each fact using fact ids encoded as a comment before each triple in Turtle format.

To simulate the situation where a confidence score (a real-number between 0 and 1.0) is associated with each fact, we assigned confidence values using an exponential distribution to all facts in the original dataset, and then we translated them into quads. Thus, YAGO contains a total of $237,180,265$ numerical quads – with $236,610,707$ quads containing only the confidence predicate and $569,558$ containing the remaining quantifiable predicates. Similarly for DBpedia, out of $668,867,020$ quads, we have $415,131,628$ numerical

[2]quads containing quantifiable predicates

quads with $396,144,979$ containing only confidence values that we assigned. Table 4.1 summarizes key statistics of the dataset and the size of the resulting database in Quark-X.

4.9.2 Benchmark Query Workloads

Our benchmark query set consists of a set of 11 top-k ranking queries each for DBpedia and YAGO. These queries are designed keeping in mind the following SPARQL query features that have been found to be quite important [7]:

- **Shape:** Based on the overall shape of the SPARQL query, we classify them as either *Star* or *Complex*. Queries that belong to *Star* class have only one non-quantifiable triple pattern, connected to other quantifiable triple patterns. On the other hand, queries which contain more than one non-quantifiable triple pattern connected to other quantifiable triple patterns belong to *Complex* class. Although this is a high-level classification, we found them to be sufficiently useful to highlight the differences in the query performance.

- **Number of triple patterns(# TP):** helps differentiate between simple queries involving no join and complex queries involving many joins.

- **Number of quantifiable triple patterns(# Quant TP):** denotes number of quantifiable filters which can be applied in a query [107].

- **Number of non-quantifiable triple patterns(# Non Quant TP):** The number of non-quantifiable triple patterns in the query highlights the efficiency of a top-k for processing non-quantifiable patterns alongside quantifiable query patterns.

- **Count of joins(# Joins):** represents number of RDF terms with same subject or object.

- **Join type:** type of joins. Different indexes may give different performance on different types of joins, this parameter captures this complexity e.g., SS denotes Subject-Subject join, OS denotes Object-Subject join, etc.

- **Join degree:** highest degree of joins. Denotes number of triple patterns in a query whose subject or object have same join vertex.

Additionally it includes **statistical** features, which includes:

- **Result cardinalities:** represents number of result tuples of a query.
- **Filtered result selectivity:** represents percentage of rows which are expected to be returned from a query.

Table 4.2 gives details some of the important features of all 11 queries for each dataset we have considered in this evaluation. As shown, the queries are designed so as to provide a broad coverage of all the key features. We point to Appendix A.2 for query listing in SPARQL. It is worth mentioning that although our framework is aimed at convex monotonic ranking functions, for simplicity we use linear ranking functions in evaluation.

4.10 Experimental Results

In this section, we present the results of our performance evaluation of Quark-X against the baseline systems we have considered. We also discuss the impact of individual components of Quark-X on the overall performance of top-k queries. Unless stated explicitly otherwise, the results correspond to the setting $k = 50$.

4.10.1 Loading of Data and Database Size

We start by discussing the loading time – summarized in Table 4.3 – of various frameworks. Among the systems compared, SPARQL-RANK and Jena-TDB took the longest time to load. SPARQL-RANK operates on an older version of Jena (ARQ 2.8.9) and it took more than 2 weeks to load DBpedia, whereas Jena-TDB-2.13.0 needed about 5 days. Note that Quark-X is faster than RDF-3X this is because RDF-3X uses reified representation therefore Quark-X's load times are better than RDF-3X.

The size of the database created by Quark-X was smaller than that by SPARQL-RANK and comparable to the one created by Jena-TDB. However, it was larger than that of RDF-3X since Quark-X uses RQ-RDF-3X as the underlying framework which creates many

Query id	Shape	# TP	# QuantTP	# Non-QuantTP	Join Vertex Count	Join Vertex Degree	Join Vertex Type	Res.Card.	Filtered Result Selectivity
1	Star	4	3	1	2	(3,2)	(SS,RS)	$111,425$	4.7 x 10^{-4}
2	Star	5	4	1	2	(4,2)	(SS,RS)	$108,911$	4.6 x 10^{-4}
3	Star	4	3	1	3	(2,2,2)	(SS,SO,RS)	298	1.3 x 10^{-6}
4	Star	4	3	1	3	(2,2,2)	(SS,SO,RS)	396	1.67 x 10^{-6}
5	Complex	5	2	3	4	(2,2,2,2)	(SO,RS)	700	2.9 x 10^{-6}
6	Star	3	2	1	2	(2,2)	(SO,RS)	$3,902$	0.16 x 10^{-4}
7	Star	3	2	1	2	(2,2)	(SO,RS)	$15,380$	0.65 x 10^{-4}
8	Complex	5	2	3	3	(3,2,2)	(SS,SO,RS)	$44,063$	1.9 x 10^{-4}
9	Complex	6	2	4	3	(4,2,2)	(SS,SO,RS)	$12,909$	5.4 x 10^{-5}
10	Star	3	2	1	2	(2,2)	(SO,RS)	42186	1.78 x 10^{-4}
11	Complex	4	2	2	3	(2,2,2)	(SS,RS,RS)	$2,639$	0.1 x 10^{-4}

(a) YAGO

Query id	Shape	# TP	# QuantTP	# Non-QuantTP	Join Vertex Count	Join Vertex Degree	Join Vertex Type	Res.Card.	Filtered Result Selectivity
1	Complex	4	3	2	3	(3,2,2)	(SS,SO,RS)	7763	0.23 x 10^{-4}
2	Star	4	3	1	2	(3,2)	(SS,RS)	26631	0.79 x 10^{-4}
3	Complex	5	3	3	3	(4,2,2)	(SS,SO,RS)	313	0.93 x 10^{-6}
4	Star	4	3	1	1	(4)	(SS)	861	0.29 x 10^{-2}
5	Star	9	8	1	2	(8,2)	(SS,RS)	167	0.5 x 10^{-6}
6	Complex	5	2	3	4	(2,2,2,2)	(SS,SO,RS)	21	0.06 x 10^{-6}
7	Complex	4	2	2	3	(2,2)	(RS,RS)	69282	2.07 x 10^{-4}
8	Complex	3	1	2	1	(3)	(SS)	85968	0.25
9	Complex	4	2	2	2	(3,2)	(SS,SO)	293	0.14 x 10^{-2}
10	Complex	5	3	2	3	(3,2,2)	(SS,SO,RS)	293	0.87 x 10^{-6}
11	Simple	3	1	2	1	(3)	SS	182	0.0019

(b) DBpedia

Table 4.2: Quark-X: Characteristics of Benchmark Queries

Framework	DBpedia		YAGO	
	Time	Size	Time	Size
Quark-X	13.28 hours	249 GB	7.59 hours	175 GB
Virtuoso	2.37 hours	66 GB	1.53 hours	43 GB
Jena-TDB	5 days	296 GB	3 days	132 GB
RDF-3X	14 hours	156 GB	8.75 hours	96 GB
SPARQL-RANK	18 days	326 GB	10 days	221 GB

Table 4.3: Quark-X: Data Load Performance of Various Frameworks

more clustered indexes than RDF-3X and has to build an additional Q-index. Apart from the Q-Index, Quark-X also creates S-Indexes, but that has a comparatively smaller memory footprint. The size of S-index for the two datasets YAGO and DBpedia is 904 MB and 1.7 GB respectively, about 2% of the size of the raw data, despite the fact that more than 50 percent of facts in two large real-world datasets which we have used for experimentation

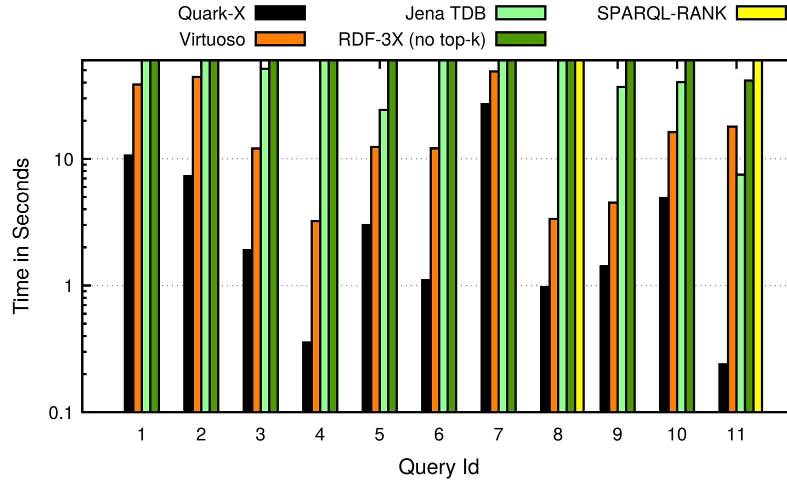
(YAGO and DBpedia) are quantitative. Further, the cost of construction of Q-Index can be amortized by removing the quantitative facts stored in POS and PSO indexes of the underlying RDF store (RQ-RDF-3X in our case), as this information is already present in S and Q-Indexes.

From the results in Table 4.3, the performance of Virtuoso is noticeably better with respect to loading time (using buffer size of 48 GB) and size of database created. This is not very surprising because unlike RQ-RDF-3X (and RDF-3X) which takes an exhaustive indexing approach, Virtuoso builds only two default indexes (PSOG and POSG), plus 3 distinct projections (SP,OP,GS) [22]. We would like to emphasize that the ideas introduced in this chapter can be applied to other RDF quad-stores. Choosing RQ-RDF-3X is merely to demonstrate the effectiveness of our approach. Note that Quark-X uses only a small amount of storage (about 6% of the size of raw data for both S- and Q-indexes together), rest of the overhead is due to the underlying engine RQ-RDF-3X.

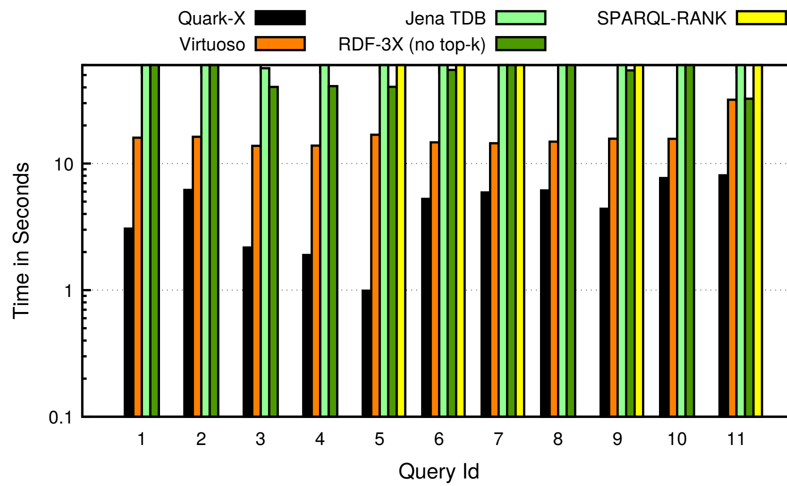
4.10.2 Query Performance

Now we turn our attention to the query processing performance in answering top-k queries, by first presenting the cold-cache performance followed by the warm-cache performance. In our discussion, we use aggregated *speedup* values computed as the geometric mean of individual query speedups for each system considered. Thus, $\text{speedup}_{(X,Y)} = \left(\prod_{i=1}^{n} \frac{Y_{Q_i}}{X_{Q_i}}\right)^{\frac{1}{n}}$, where X_{Q_i} denotes the time taken by the system X for evaluating the query Q_i and n is the total number of queries in the benchmark. Workload-average benchmarks like TPC frequently use geometric mean, since it normalizes the values being averaged against outliers.

Our primary comparison is against the columnar-store Virtuoso, which has been shown to have superior performance in comparison to other RDF storage engines [22]. It is embellished with many optimizations, of which vectorization and cache-consciousness are the most relevant to our experiments. In contrast, the current implementation of Quark-X runs in single threaded mode and does not have cache-conscious features as well as vectorized execution modes. We believe that the use of these optimizations will significantly help in further improving the performance of Quark-X.



(a) DBpedia



(b) YAGO

Figure 4-5: Cold-cache Query Processing Performance

Cold-cache Performance

The performance of each system on all benchmark queries is summarized in Figure 4-5, which plots in logarithmic scale the average time taken, in seconds, after running the query in cold-cache 5 times. Note that we set the time-out for queries as 30 minutes, hence, the Y-axis is limited to $1,800$.

The first observation we can make from these numbers is that Quark-X outperforms all other systems under consideration by a large margin. We also observe that SPARQL-RANK is many orders of magnitude slower than even RDF-3X which does not do any top-k processing at all, and instead materializes all the results of the query. As we already discussed in Section 4.4, this is primarily due to the query processing algorithm of

SPARQL-RANK which ends up using many random accesses over indexes. It is worth mentioning that SPARQL-RANK returned incorrect results for all queries except Q_8, Q_{11} of DBpedia, and Q_5, Q_6, Q_7, Q_9, Q_{11} of YAGO, hence the results for the incorrect queries have not been shown in Figure 4-5. For queries returning correct results, Quark-X outperformed SPARQL-RANK over both DBpedia and YAGO, with all benchmark queries timing out on SPARQL-RANK. In subsequent experiments, we will not report the results over SPARQL-RANK.

Quark-X outperforms RDF-3X for all queries by 1–2 orders of magnitude. The reason for poor performance of RDF-3X is: its inability to retrieve just the top-k results.

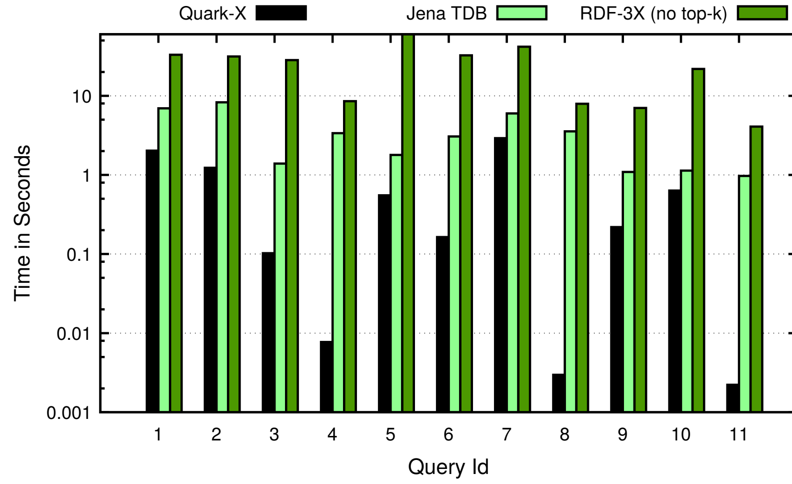
We can also see that Virtuoso is by far the closest in performance to Quark-X cold-cache setting. Quark-X outperforms it by a speedup factor of 3.9 over YAGO and 7 for DBpedia. Of all the queries, Quark-X is significantly faster for Query Q_{11} over DBpedia by almost two orders of magnitude over Virtuoso, demonstrating the power of S-index. S-indexes help Quark-X skip over large portions during query evaluation.

Warm-cache Performance

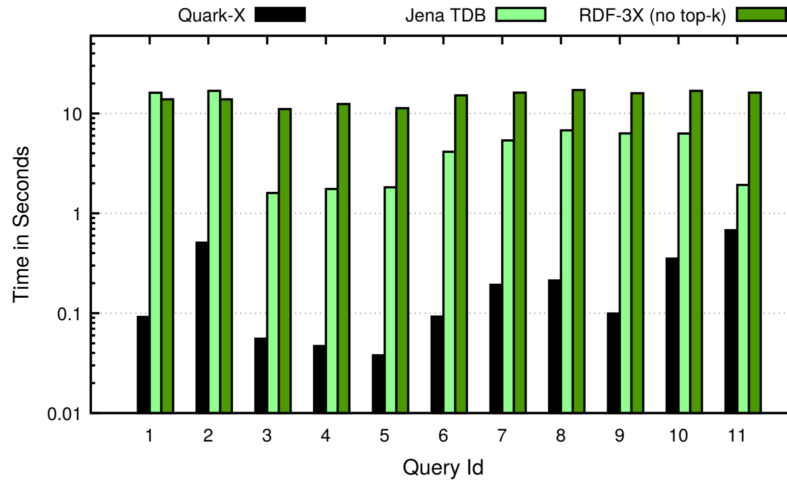
We now turn our attention to warm-cache query processing performance of systems under comparison. Jena and RDF-3X make use of only the operating system cache, whereas Virtuoso explicitly manages its own (somewhat large) cache. Therefore, in order to have a fair comparison, we present warm-cache results in two parts:

Mode 1: Figure 4-6 reports the results of comparison amongst Quark-X, Jena, and RDF-3X where all the systems use only O.S. caches. It can be seen that Quark-X outperforms Jena and RDF-3X significantly. Specifically, over DBpedia, Quark-X outperforms Jena by a speedup factor of about 22 and RDF-3X by a factor of 206.

Mode 2: In this mode, we compared Quark-X and Virtuoso, both using their own internal caches. We suitably modified Quark-X also to cache the working set of the database, similar to Virtuoso. Since Quark-X does not yet use vectorization, the vector size of Virtuoso is set to 1. The results of this comparison are shown in Figure 4-7, where we find that Quark-X continues to outperform Virtuoso – by a speedup factor of 24.2 for YAGO and 13.6 for DBpedia. Note that we do not report numbers for Virtuoso for Q_2 and Q_4 of



(a) DBpedia

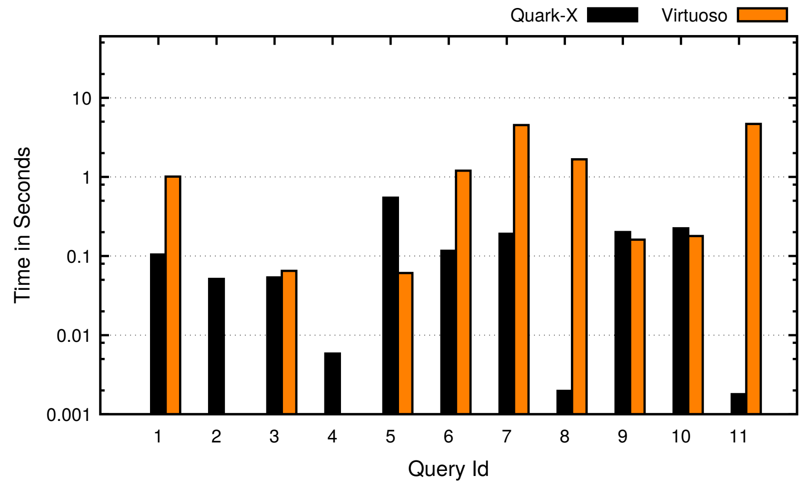


(b) YAGO

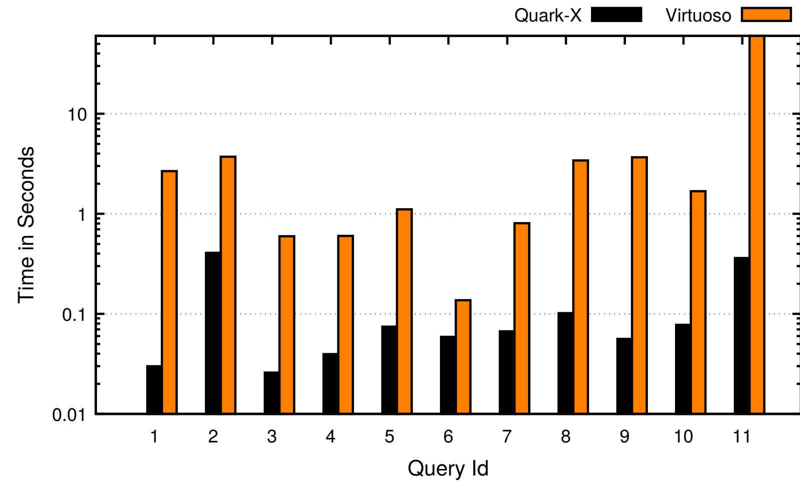
Figure 4-6: Warm-cache Query Processing Performance (Mode1)

DBpedia, because, somewhat surprisingly, it returned incorrect results.

Quark-X is slower than Virtuoso for queries Q_5, Q_9 and Q_{10} over DBpedia. On further analysis, we discovered that unlike YAGO, DBpedia is highly unstructured leading to a huge number of characteristic sets, many of which occur only once. Specifically, DBpedia contains 197,530 characteristic sets and YAGO contains 86 characteristic sets of quantifiable predicates. Large number of characteristic sets leads to fragmentation in id space – which naturally leads to increased random accesses. For a given set of quantitative predicates of Q_5, Q_9 and Q_{10} the *Non-Quantifiable Predicate Join (NQP-Join)* stage (cf. Section 4.6), of Quark-X's query processing engine has to look through many different characteristic set space – which is bound to cause a significant overhead. We can mitigate this by generating



(a) DBpedia



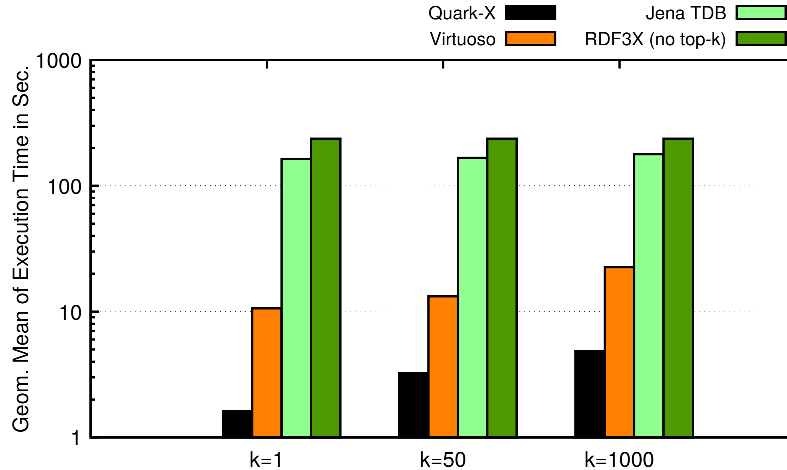
(b) YAGO

Figure 4-7: Warm-cache Query Processing Performance (Mode2)

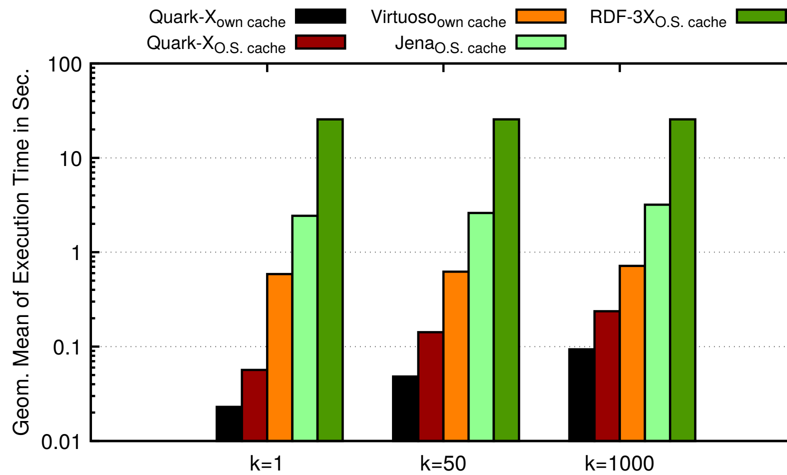
characteristic sets with approximate overlap.

Finally, Quark-X is more than an order of magnitude superior to all the other systems even when it retrieves all the required results e.g. for Q_6 of DBpedia. The good performance of Quark-X in comparison to Virtuoso and Jena is attributed to its efficient use of S-Indexes and reduced memory and index access due to increased data-locality induced by its novel semantically encoded identifiers.

We also observe that the consistent poor performance of Jena in both cold as well as warm cache is due to its exclusive use of index-based joins leading to random access patterns during query processing.



(a) Cold-cache



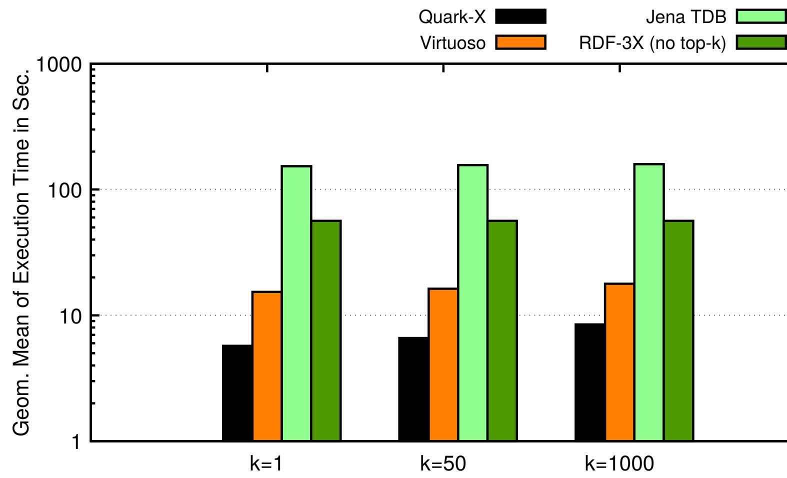
(b) Warm-cache

Figure 4-8: Quark-X: Performance over DBpedia for Varying k

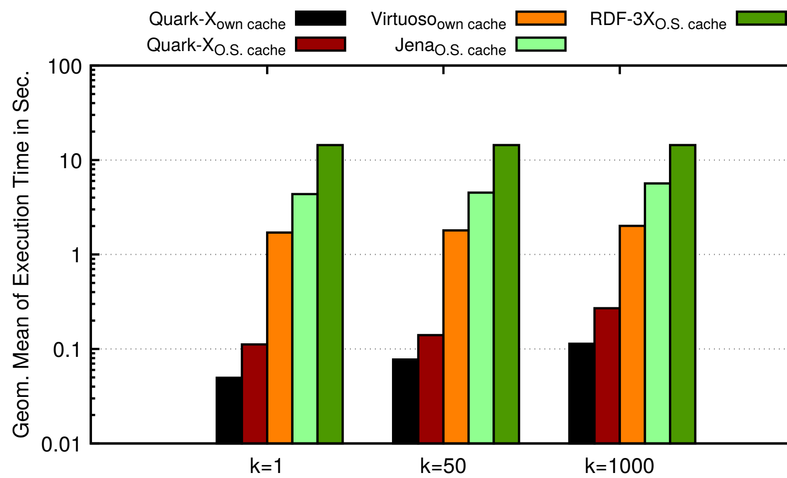
4.10.3 Impact of Varying k

Next, we consider the impact of varying k. Figure 4-8 and Figure 4-9 shows the geometric mean of all queries for each value of $k = \{1, 50, 1000\}$ in both cold- and warm-cache (the subscript denotes, whether the system uses its own cache or O.S. cache) for DBpedia and YAGO respectively.

One of the immediate effects that can be observed from these plots is that on increasing the value of k, systems which translate URIs and strings in RDF data into integer ids and back during final result generation are significantly affected. If the dictionary used for this translation is inefficient, for large values of k, the system can spend significant amount of time in its dictionary lookups during result generation. Due to this reason even



(a) Cold-cache



(b) Warm-cache

Figure 4-9: Quark-X: Performance over YAGO for Varying k

though Virtuoso and Jena evaluate and output all results still they show slight variation with increasing value of k. Quark-X, in contrast, does not evaluate all results, thus, the variation of Quark-X's query processing performance is much better. Additionally, it is noteworthy that Quark-X's performance in warm cache using its own cache is better than its performance using O.S. caches. This performance improvement is attributed to the elimination of decompression cost in Quark-X when it uses its own cache, as decompression is known to be beneficial for cold cache but is an unnecessary overhead for warm cache (when caching is performed by the O.S.).

Upon incrementing k, the use of block-wise processing in Quark-X also comes into play as follows: we process k fraction of buckets at a time until all k results are obtained. As we increase k, if all top results are obtained from the initial fraction of buckets, then we can see significant performance speedups. Only in specific queries which include many non-quantifiable facts, this feature does not play a crucial role.

4.11 Discussion & Outlook

This chapter presented Quark-X, an efficient top-k query processing framework for RDF quad stores. The salient features of Quark-X include its indexes, viz., (1) S-Index and Q-Index, which provide a so far unique way to perform early-termination and join-ahead pruning ahead of the join operator using quantifiable facts, (2) the Rank-Hash Join query execution algorithm to adaptively choose the best index for joins at runtime, and (3) the Semantic Encoding strategy used for increasing data locality. Through our experiments on two large real-world datasets, Quark-X was shown to outperform existing commercial and academic engines by 1-2 orders of magnitude.

4.11.1 Outlook

There is ample room for future research with some of the prominent research directions discussed below:

- In our experiments we chose bucket size to be proportional to the block size of the

storage device used such as hard disk, Solid State Disks, etc. This is done to enable join-ahead pruning of SPRO indexes using S-Index at block-wise granularity. For future work, an interesting direction is to generalize the bucket size for memory-resident DBMS architectures which operate at a finer granularity. Note that S-Index enabled join-ahead pruning is applicable in this scenario without modification.

Also when different storage devices are used it may be advisable to replicate S-Index at different granularities, thus introducing additional redundancy. Experimentally evaluating such a setup where more than one partitioning strategy is employed is an interesting open issue.

- The techniques presented in this chapter can also be used for top-k querying on temporal data, albeit with careful use of S-indexes. Unlike numerical attributes, temporal attributes are expressed as (*time begin, time end*) intervals which requires careful bound checking during query processing. Thus investigating Quark-X's performance on temporal datasets is an exciting research direction.

- Present work opens up fascinating avenues for future research in query optimization for top-k queries in RDF quad stores. In Quark-X we used a heuristic based approach for plan generation, wherein numerical predicates were pushed deep in the query plan. Naturally the heuristic based approach may not always produce the optimal plan. Hence the need to develop sophisticated plan generation techniques for top-k queries.

- Finally, one may ask whether bucketed approach presented in this work can also be implemented in a distributed setting e.g., in a shared nothing architecture. Note our S-indexes can be used for join-ahead pruning in a distributed environment too. Investigations in this setting will open new research directions.

Chapter 5

STREAK: An Efficient Engine for Processing Top-k SPARQL Queries with Spatial Filters

5.1 Motivation

Motivated by its use in critical applications such as emergency response, transportation, agriculture etc., geo-spatial data are part of many large-scale semantic web resources. Efforts to standardize the representation of geo-spatial data and relationships between spatial objects within RDF/SPARQL has resulted in GeoSPARQL standard [77], adopted by many RDF data repositories such as LinkedGeoData [11], GeoKnow [58], etc. Even the large-scale open-domain knowledge-bases such as YAGO [47, 65], WikiData [94], and DBPedia [59] contain significant amounts of spatial data. To make effective use of this rich data, it is crucial to efficiently evaluate queries combining topological and spatial operators –e.g., overlap, distance, etc.– with traditional graph pattern queries of SPARQL.

Furthermore, modern knowledge bases contain not only simple (binary) relationships between entities, but also model higher-order information such as uncertainty, context, strength of relationships, and more. Querying such knowledge bases results in complex workloads involving user-defined ad-hoc ranking, with top-k result cut-offs to help in rea-

soning under uncertainty and to reduce the resulting overload. Thus the geo-spatial support is not just limited to efficient implementation of spatial operators that works well with SPARQL graph pattern queries, but also should integrate with top-k early termination too.

We present STREAK, an efficient RDF data management system that can model higher-order facts, support the modeling of geo-spatial objects and queries over them, enable efficient combined querying of graph pattern queries with spatial operators, expressed in the form of a FILTER, along with top-k requirements over arbitrary user-defined ranking functions, expressed in the ORDER BY — LIMIT clauses. In this chapter, we focus our attention on a recently proposed *top-k spatial distance join* (K-SDJ) [79], although other spatial operators are also possible within STREAK. Following are a few applications of such distance join queries within the context of RDF databases from the GeoCLEF 2006-07 [66]:

(i) *Find top wine producing regions around rivers in Europe*

(ii) *List car bombings near Madrid and rank them by estimated casualties*

(iii) *Find places near a flooded place which have low rates of house insurance*.

Similarly, distance join queries over graph structured data can also be formulated in many diverse scenarios – e.g., in a job search system one can ask for *top jobs with highest wage which match the skills and are near the location of a given candidate*, which can be expressed as a SPARQL query with a spatial distance FILTER and top-k ranking on highest wage.

5.1.1 Challenge

A large-body of work exists in relational database research for integrating geo-spatial and top-k early termination operators within generic query processing framework. Therefore, a natural question to ask is whether those methods can be applied in a straightforward manner for RDF/SPARQL as well. Unfortunately, as we delineate below, the answer to this question is in the negative, primarily due to the schema-less nature of RDF data and the self-join heavy plans resulting from SPARQL queries.

Consider applying one of the popular techniques, which build indexes over joining (spatial) tables during an offline/pre-processing phase [76]. However, in RDF data, there

is only one large triples table, resulting in a single large spatial index that needs to be used during many self-joins making it extremely inefficient. On the other hand, storing RDF in a property-table form is also impractical due to a large number of property-tables – e.g., in a real-world dataset like BTC (Billion Triples Challenge) [28], there are $484,586$ logical tables [68].

Next, the unsorted ordering (w.r.t. the scoring function) of spatial attributes during top-k processing prevents the straightforward application of methods that encode spatial entities to speed up spatial-joins [61]. This is because these approaches assume sorted order of spatial attributes, and hence are able to choose efficient merge joins as much as possible. Finally, state-of-the-art query optimization techniques for top-k queries [52] cannot be adopted as-is for spatial top-k since real-world spatial data does not follow uniform distribution assumptions that are used during query optimization.

5.1.2 Contributions

In this chapter, we present the design of the STREAK system, and make following key contributions:

1. We present a novel soft-schema aware, spatial index called S-QuadTree. It not only partitions the 2-d space to speed spatial operator evaluation, it compactly stores the inherent soft-schema captured using *characteristic sets*. Experimental evaluations show that the use of S-QuadTree results up to *three orders of magnitude* improvements over using state-of-the-art spatial indexes for many benchmark queries.

2. We introduce a new spatial join algorithm, that utilizes the soft-schema stored in S-QuadTree to select optimal set of nodes in S-QuadTree. The spatial objects enclosed/overlapping the 2-d partitions of these nodes are used for pruning the search space during spatial joins.

3. We propose an adaptive query plan generation algorithm called *Adaptive Processing for Spatial filters or APS*, that switches query plan with very low overhead. STREAK achieves this via its cost model to ensure that entire query is processed with least cost.

We implement these features within QUARK-X [57], a top-k SPARQL processing sys-

tem based on RDF-3X [69, 70], resulting in a holistic system capable of efficiently handling spatial joins. For experimental evaluation, we make use of two large real-world datasets — (a) YAGO [47, 65], and (b) LinkedGeoData [11]. Benchmark queries were focused on K-SDJ queries, and were a combination of a subset of queries from GeoCLEF [66] challenge and queries we constructed using query features found important in literature [7, 57, 61]. We compare the performance of STREAK with the system obtained by replacing our S-QuadTree based spatial join by synchronous R-tree traversal spatial join algorithm [24]; and test end-to-end system performance against PostgreSQL [78] which has in-built spatial support and Virtuoso [93], a state-of-the-art RDF/SPARQL processing system. Our experimental results show that STREAK is able to outperform these systems for most benchmark queries by $1-2$ orders of magnitude in both cold and warm cache settings.

5.1.3 Organization

The rest of the chapter is organized as follows: Section 5.2 briefly introduces the SPARQL query structure, and how spatial distance joins can be expressed. It also outlines a running example query that we use throughout the chapter. Next, in Section 5.3 we describe the key features of STREAK including S-QuadTree, node selection algorithm, and APS algorithm. We describe the datasets and queries used in our evaluation in Section 5.4. Detailed experimental results are given in Section 5.5 followed by an overview of related work in Section 5.6.

5.2 Preliminaries

RDF consists of triples, where each triple represents relationship between *subject(s)* and *object(o)*, with name of the relationship being *predicate(p)*. SPARQL is the standard query language for RDF [95] and we consider SPARQL queries having the following structure:

```
SELECT [projection clause]
WHERE [graph pattern]
FILTER [spatial distance function]
ORDER BY [ranking function]
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
#@ <id1>
:Mosel :grapeVariety :Albalonga.
#@ <id2>
:Mosel :soilType :porus_slate.
#@ <id3>
:Mosel :hasProduction "4500000000"^^xsd:double.
#@ <id4>
:Mosel :hasGeometry "POINT(28.6,77.2)".
#@ <id4>
:Moselle :pollutedBy :pesticide.
#@ <id5>
<id4> :includes ?pest.
#@ <id6>
?pest :concentration ?c.
#@ <id7>
:Moselle :hasMouth :Rhine.
#@ <id8>
:Moselle :source :Vosges_mountains.
#@ <id8>
:Moselle :hasGeometry "LINESTRING((28.3,77.5),(28.4,77.6))".
```

Figure 5-1: Example RDF knowledge graph: Describes pollution in wine growing regions of Europe.

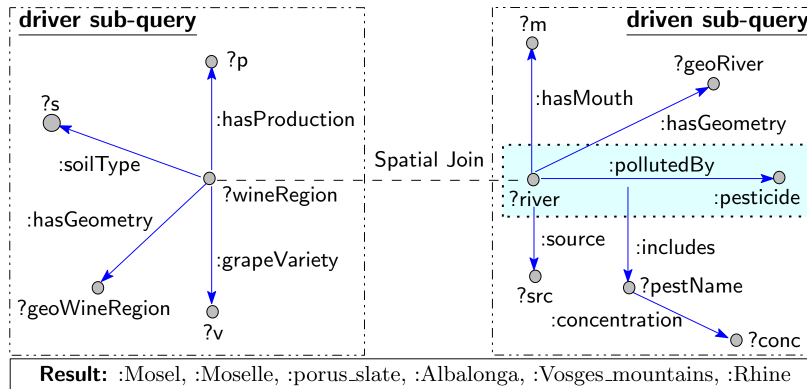
```
LIMIT [top-K results]
```

The `SELECT` clause retrieves results consisting of variables (denoted by “?" prefix) and their bindings. Graph patterns in WHERE clause (also called triple patterns) are expressed as: ⟨?s ?p ?o⟩, ⟨?r rdf:subject ?s. ?r rdf:predicate ?p. ?r rdf:object ?o⟩, where ?r represents fact or reification identifier. Keyword prefix rdf is part of the RDF vocabulary and helps declare parts of an RDF statement identified through its identifier *?r*.

Next, the FILTER clause restricts solutions to those that satisfy the spatial predicates. In this chapter, we focus only on the use of DISTANCE predicate for distance joins. It should be noted that the techniques discussed in this chapter are equally applicable to all spatial predicates defined in GeoSPARQL standard by the Open Geospatial Consortium [14]. The ORDER BY clause helps establish the order of results using a user-defined ranking function. The LIMIT clause controls the number of results returned. Together, ORDER BY and LIMIT form the top-k result retrieval with ad-hoc user-defined ranking function.

5.2.1 Running Example

As a running example, we use a reified RDF listing of the knowledge graph shown in Figure 5-1. It shows a snippet which contains information about pollution in wine growing regions of Europe. Figure 5-2 illustrates a typical top-k SPARQL query on this knowledge graph. This query produces the details of the top-1 wine producing region situated near a




```
SELECT ?wineRegion ?river ?s ?v ?src ?m
  ((f1(?p) * f2(?c)) as ?rank)
WHERE {
  ?wineRegion :grapeVariety ?v.
  ?wineRegion :soilType ?s.
  ?wineRegion :hasProduction ?p.
  ?wineRegion :hasGeometry ?geoWineRegion.
  ?reif rdf:subject ?river.
  ?reif rdf:predicate :pollutedBy.
  ?reif rdf:object :pesticide.
  ?reif :includes ?pestName.
  ?pest :concentration ?c.
  ?river :hasMouth ?m.
  ?river :source ?src.
  ?river :hasGeometry ?geoRiver.
  FILTER(distance(?geoWineRegion,?geoRiver)<"10")
} ORDER BY DESC(?rank) LIMIT 1
```

Figure 5-2: Running Example Query: Finds the high-producing wine growing regions of a given soil type and grape variety (driver sub-query) and rivers with high pollution levels and specified mouth and source (driven sub-query), which are no more than 10kms apart (spatial join). The results are ranked based on product of amount of wine production and the river pollution concentration (top-k).

river, ranked using a function combining its production and pollution levels.

5.3 STREAK

In this section, we describe in detail our STREAK system for processing top-k spatial join queries. STREAK follows exhaustive indexing approach introduced by RDF-3X [69], and extended to support reified RDF stores [56, 57]. This results in indexes over permutations of *subject*, *predicate*, *object* and *reification id* assigned to each statement in the knowledge graph. STREAK also maintains an index containing block-level summary of various

numerical attributes useful for top-k processing of user-specified scoring function.

Apart from these indexes, STREAK introduces a novel variant of quad-tree, a popular in-memory spatial index structure, called S-QuadTree. It stores all the spatial entities in the knowledge graph, and embeds the associated soft-schema information. The structure of S-QuadTree index along with spatial entity identifier encoding are explained in Section 5.3.1.

Next, in Section 5.3.2, we present the details of the spatial join algorithm which leverages S-QuadTree in a careful manner to balance the CPU and IO costs while processing the spatial filter specified in the query. We adopt the terminology from the adaptive query processing literature [31] and term one side of the spatial join (see Figure 5-2) as **driver sub-query**, whose tuples are joined with remaining sub-query, coined **driven sub-query**. The plan of the driven sub-query is chosen based on runtime statistics of driver sub-query. As part of the spatial adaptive query processing, explained in Section 5.3.3, STREAK chooses the driver and driven sub-query from the original query.

5.3.1 S-QuadTree Index for Spatial Entities

Underpinning STREAK is the S-QuadTree index that is designed with compactness and efficiency in mind, to drive a dynamic query execution strategy that changes plans during run-time. STREAK combines Z-order locality preserving layout for identifier encoding, quad-trees [34] for spatial partitioning and indexing, and characteristics sets for capturing the soft-schema information related to spatial objects in the corresponding partition. The intuition behind combining all this information in a single index is motivated by the schema-less, single table philosophy of the RDF. The resulting index, called the S-QuadTree index, enables a more deeply integrated query processing strategy that helps in early pruning as well. We illustrate the structure of S-QuadTree for our running example query in Figure 5-3.

We note that R-Trees are a popular choice for indexing spatial data. However, with the design of our indexes and our adaptive query processing strategy, we observed that STREAK outperforms R-Trees in this RDF setting (validated in the experimental section).

Identifier assignment and encoding: Quad-trees recursively partition the spatial re-

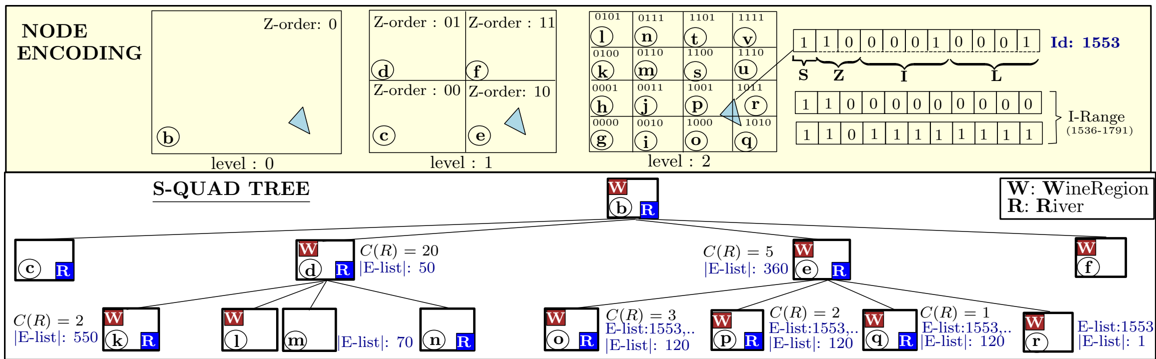


Figure 5-3: Node encoding: The triangle is assigned the identifier based on *(z-order = 2, local id = 1, level = 1)*. **S-QuadTree: where each node captures E-List, I-Range, Characteristic Sets for River (R) and Wine (W), Cardinalities for Spatial CS for river C(R), and MBR (not shown)**

gion into four quadrants and offer a light-weight hierarchical indexing that can be easily maintained in-memory. For each spatial entity in the dataset, we assign a unique identifier that inherently captures its spatial characteristics within the hierarchical index. For both points and polygons, the identifier value corresponds to the deepest node in the quad-tree in which the object completely lies inside. For some polygons, this containing node could be the root of the tree as well. In any case, we encode the identifier value compactly in the following format: **(S, Z, I, L)**.

- **S:** is the most-significant bit (MSB) of the identifier. This indicates whether the identifier corresponds to a spatial or non-spatial entity. This allows spatial entities to be clustered together in storage.

- **Z:** is the Z-order of the deepest-level node that completely encloses the spatial object. This ensures that in the *ID* space, those spatial objects with the same *Z-order prefix* are clustered together for efficient retrieval later. Thus, together S and Z help in preserving locality.

- **I:** is the local identifier (incrementally assigned) of the object that distinguishes from other objects within the same node. In the case of skewed datasets, when the number of spatial objects causes an overflow in the local identifier bits, then, such identifiers are assigned to the parent node. The algorithms specified in this chapter work even in the above case, without any loss of generality.

- **L:** is the level / height of the node within the quad-trees. We limit the maximum number of levels in the quad-tree to 10 and therefore $|L|$ is fixed to 4 bits, in this chapter. We found that there is little benefit in partitioning a node to have more than a million (4^{10}) quadrants. The number of bits occupied by Z is 2^L, and the rest of the bits capture the local identifiers.

Naively storing the identifiers leads to a significant increase in storage space. Our representation not only ensures that the quad-tree remains lightweight, but also enables co-location of objects in space using S and Z bits as explained earlier. By design, we impose equivalent hierarchies for both quad-trees and Z-curve encoding to capture the identifiers in the above form.

For example, the triangle in Figure 5-3 is completely enclosed by node e, therefore this polygon is assigned the *z-order 2*.

I-Range and E-list: During the indexing process, after generating the identifiers for spatial objects, we capture two types of information in each and every non-empty node of the S-QuadTree that will help to aggressively prune results during query execution.

1. *I-Range:* Range (min and max) of object identifiers which completely lie within the node or any of its children. Note that the *range* of these objects can be identified from the Z-order of the node.

2. *E-List:* Spatial objects overlapping the node but not fully contained within the node. The *E-list* objects are stored within each node of the S-QuadTree as a list of *Explicitly Encoded Spatial Objects*. These objects overlap more that one child node (necessarily) and therefore are assigned the Z-order of the parent node. To identify the exact child nodes with which these objects overlap, they are explicitly stored as *E-list*. Storing only the *E-list* objects helps save processing time – as prefix range comparisons are faster. Typically, on the dataset we have seen, only a small number of *Explicitly Encoded Spatial Objects* need to be stored for each node of the quad-tree.

In Figure 5-3, nodes o, p, q and r do not completely enclose the triangle, therefore this object (ID: 1553) is explicitly stored in their nodes. Hence, object 1553 is called an *E-list*

object in these nodes. However, the same object lies completely inside node e, and hence this object lies in node's *I-Range* (1536 — 1791).

I-Range and *E-list* offers two complementary ways to prune intermediate results during processing. The role of *I-Range* and *E-list*, and the trade-offs for both during adaptive query execution are clarified in the subsequent sections.

Characteristic Sets(CS): For completeness of the chapter, we again revisit Characteristic Sets which were described in Chapter 4. RDF data usually has a *soft-schema*, stemming from the fact that multiple RDF statements are used to describe the 'properties' of a subject, and these statements repeat often for similar subjects. To illustrate this, consider the following query patterns for the query given in Figure 5-2 — (?wineRegion :grapeVariety ?v), (?wineRegion :soilType ?s), (?wineRegion :production ?p). These statements describe a wine producing region's soil, variety of grapes grown and soil type, which it imposes through the predicates :grapeVariety, :soilType and :production. These three predicates are also strongly correlated in the database since this pairing of predicates holds for only wine producing regions. Other entities can likewise be uniquely identified by the predicates connected to them. This observation has been used earlier for improving the cardinality estimates of RDF queries [68], where they name the set of predicates connected to an entity as its characteristic set. In general Neumann et al. [68] define for each entity *s* its characteristic set S for an RDF dataset R as:

$$S(s) = \{p | \exists o : (s, p, o) \in R\}$$

They define the set of characteristic sets as:

$$S = \{S(s) | \exists p, o : (s, p, o) \in R\}$$

In order to leverage the soft-schema for efficient spatial joins, we store in each node of quad-tree, three types of characteristic sets of all objects enclosed by that node. Note that we don't store the CS *as is*, but for space efficiency, encode them with Bloom filters – more accurately in counting Bloom filters. Although using Bloom filters incurs false positives it is important to note that this will not affect the correctness of results. This only results

in overestimation of query plan cost. Note that this may cause bad plans to be chosen we leave the study of sensitivity of plan selection to Bloom filter configurations as future work. Also once the query plan is chosen false positives cause potentially more disk blocks to be read.

- **Self:** The CS that identifies a spatial object. For example, for the object *:Mosel*, its CS is *Vineyard*.
- **Incoming:** The CS that are incoming towards the spatial entity. With (:Hochmosel-brücke, :isLocatedIn, :Mosel), the incoming entity is *:Hochmoselbrücke*. And the CS that identifies *:Hochmoselbrücke* is *Bridge*.
- **Outgoing:** The CS that are outgoing from the spatial entity. For (:Mosel, :isIn, :Germany), the CS corresponding to *:Germany* is *Country*.

Nodes *d* and *e* exhibited in bottom center of the Figure 5-3 intersect spatial objects described by characteristic sets **W** and **R**, and node *f* exhibited in bottom right of Figure 5-3 intersect spatial objects described by characteristic set **W** only. In Figure 5-3 that **R** and **W** are the characteristic sets of river and wine, respectively. Hence, when performing spatial join between the *driver* sub-query described by **W** and the *driven* sub-query described by **R**, the spatial join algorithm needs to visit only the children of the nodes *d* and *e*. Thus, storing the Characteristic Set(CS) of spatial objects within the nodes of the quad-tree helps perform focused traversal of the quad-tree, and reducing comparisons.

Cardinalities of Spatial CS: For each characteristic set and node, we find the objects of characteristic set that intersect with the node's spatial extent. We refer to these characteristic sets as *Spatial CS*. We store the cardinalities of *Spatial CS* in nodes of S-QuadTree. These stored cardinalities help us estimate the cardinalities of characteristic sets which would satisfy the spatial join constraint.

MBR: To improve performance STREAK stores in each node only the Minimum Bounding Rectangle (MBR) of spatial objects associated with that node of S-QuadTree, this helps improve performance by avoiding comparisons with unoccupied regions of nodes.

Note that this Chapter adds specific set of encoding strategies and indexes, and they could interfere with the use of indexes and plan selection that were described in Chapter

4. For instance, spatial encoding may conflict with semantic encoding if implemented together in the same system. Therefore the proposed encoding strategies do not work well with each other and must be implemented independently.

Finally, S-QuadTree follows a space-oriented partitioning approach, therefore updates are additions to the nodes of the quad-tree and its children, without affecting the nodes of the S-QuadTree with which the inserted/updated spatial object does not intersect.

5.3.2 Spatial Join Algorithm in STREAK

We next discuss how STREAK employs its S-QuadTree for efficiently implementing a spatial-join during query processing. Broadly speaking, it adopts the popular block-wise query processing technique during which it retrieves bindings from the driver sub-query in a block-wise manner. For every such block, it uses *sideways information passing*(as also described in Chapter 4 to find candidate blocks from the driven plan based on their spatial proximity, with which the actual spatial-join is finally performed. The S-QuadTree is used to efficiently compute a smaller set of candidate blocks that leads to reduction in the number of unnecessary joins. The overall process is executed in three phases which are described below. For explanation we continue with our running example of Figure 5-3, with driver sub-query as **W***ine producing region*, and driven sub-query as **R***iver*.

1. Obtain candidate nodes: In this stage block-wise bindings retrieved from the driver sub-query are used to *filter out* spatial regions, essentially S-QuadTree nodes, that do not contain results of the spatial join. This is done by traversing the S-QuadTree from its root to its leaf nodes by following only nodes that satisfy both the bindings from the driver sub-query as well as characteristic sets of the driven sub-query; we denote this set of nodes by $\mathcal{V}$. For example, at level 1, we observe that nodes c and f do not satisfy the join as either **W** or **R** is missing. However, nodes d and e participate in the join between **W** and **R**, thus STREAK traverses only these nodes.

We use $\mathcal{V}_l$ to denote the nodes in $\mathcal{V}$ in level l. Due to the nature of the explicitly encoded objects associated with a node, all the spatial objects that are results of the join are contained in $\mathcal{V}_l$ for any level l, and in particular, in the leaf-nodes of $\mathcal{V}$ (which we denote

by $\mathcal{V}_L$).

Algorithm 5 Filtering driven sub-query

```
1:  Input: Driver-SubQuery, Driven-SubQuery, S-Quadtree
2:  Output: Filtered-SpatialObj
3:  S = Spatial_Entities(Driver-SubQuery)
4:  (I-Range, E-list) = Nearby_SpatialEntities(S, S-Quadtree)
5:  for row in Driven-SubQuery do
6:      if row.SpatialEntity in I-Range then
7:          Filtered-SpatialObj.add(row.SpatialEntity)
8:      end if
9:      if row.SpatialEntity in E-list then
10:         Filtered-SpatialObj.add(row.SpatialEntity)
11:     end if
12: end for
13: return Filtered-SpatialObj
```

2. Filtering driven sub-query: The *I-Range* and *E-list* associated with a node in the S-QuadTree, say a, store the spatial objects associated with a and can be used for filtering the driven sub-query using *sideways information processing*. Next we explain how filtering is done: After obtaining satisfying spatial entities from driver sub-query, the spatial entities are looked up in SQuad-tree to find nearby spatial entities which satisfy the spatial join constraint. As spatial entities are stored with *I-Range* and *E-list* associated with every node of SQuad-tree, therefore we use these spatial entities as filters over the driven sub-query. The algorithm for performing filtering of driven sub-query is explained formally in Algorithm 5. Inputs to the algorithm are Driver-SubQuery, Driven-SubQuery and S-QuadTree. Driver-SubQuery and Driven-SubQuery are shown in Figure 5-2. Function Spatial_Entities(Driver-SubQuery) finds spatial entities in driver sub-query. Function Nearby_SpatialEntities(S, S-Quadtree) finds I-Range and E-lists of spatial entities satisfying the spatial predicate, and stores them as tuple in (I-Range, E-list). Lines 5 to 12 iterate over all rows/tuples in Driven-Subquery to find spatial entities – denoted by row.SpatialEntity – which are also present in either I-Range or E-list found in line 4. Data structure Filtered-SpatialObj is a list. If row.SpatialEntity is in I-Range or E-list, then it is added to Filtered-SpatialObj. List Filtered-SpatialObj is returned in line 13.

Note that, there may be local density variations in different nodes of the S-QuadTree. Therefore, it is important to carefully choose the nodes in $\mathcal{V}$ to be used for filtering. Going

forward, we first discuss the process of filtering and associated CPU and IO costs using *I-Range* and *E-list*, which help skip entries which are retrieved from index scans of driven sub-query. Then we describe our algorithm to choose appropriate nodes in $\mathcal{V}$ that can be used for efficient filtering. Recall from Figure 5-3 that *I-Range* of a node is the set of identifiers which completely lie within that node e.g. *I-Range* of node '*d*' is from 1536 to 1791. An RDF fact containing a spatial entity is skipped if it does not lie within *I-range* or *E-list*. Thus *I-Range* and *E-list* can be used for skipping irrelevant entries in the indexes. Note that skipping entries using *I-range* is faster — to understand this consider the case when the current block does not contain values in between *I-range*, then a skip can be made to the next block by skipping all irrelevant entries in between. However, skipping using a large *E-list* can be expensive, as each RDF fact needs to be potentially checked for skip. We observed that this causes high CPU overhead, as it disrupts the high cache locality of sequential scans [70]. Therefore, we associate a CPU cost with a that is proportional to the cardinality of $E\text{-}list(a)$.

Take for example, Figure 5-3 in which we assume that $\mathcal{V}$ is the set of all nodes containing both **W** and **R**. We observe that the size of *E-list* at child node k is greater than at node d. However, notice that node d satisfies the join with estimated cardinality of ***C(R)*** being 20 whereas, its children node k has estimated cardinality of ***C(R)*** as 2. Therefore, with respect to fetching of blocks, it is better to filter using the *I-range* and *E-list* of k compared to those of d. We capture this by defining the IO cost of a node a as the cardinality of the characteristic set stored at a and represent it as $|CS(a)|$.

Observe that the spatial objects associated with a are also associated with the parent of a – so it may not be prudent to filter using both a and its parent. Furthermore, the IO cost of a node decreases as the level increases whereas CPU cost increases. Therefore, it is necessary to compute an optimal set of nodes for filtering, say $\mathcal{V}^* \subseteq \mathcal{V}$ such that (a) the nodes in $\mathcal{V}^*$ collectively cover all spatial objects associated with nodes in $\mathcal{V}$, and (b) have minimal IO and CPU footprints. We should mention that nodes in $\mathcal{V}^*$ may come from different levels, e.g., a few possibilities of $\mathcal{V}^*$ in Figure 5-3 are {b}, {d, e}, {d, o, p, q}, {e, k}. To compute $\mathcal{V}^*$, first we associate a $cost(a)$ with every node $a \in \mathcal{V}$ based on the above discussion.

$$cost(a) = \alpha_{IO} \overbrace{|CS(a)|}^{\text{IO cost}} + \alpha_{CPU} \overbrace{|E\text{-}list(a)|}^{\text{filtering}}$$

Next, for any node a, we define $\mathcal{V}^*(a)$ as the optimal set of nodes from $\mathcal{V}$ in the subtree rooted at a. For computing the cost of $\mathcal{V}^*$, we also need to take into consideration the inherent (CPU) cost of combining the large *E-list*s of nodes in $\mathcal{V}^*$. Obviously, this cost is 0 if there is none or only one *E-list* in consideration and is proportional to the size of the combined list (we keep the lists sorted) if there are two or more lists; we use α_{merge} as the proportionality constant and denote $\alpha_{merge}|E\text{-}list(a)|$ by $\xi(a)$.

Algorithm 6 Computing optimal nodes for filtering

1: **Input**: candidate nodes $\mathcal{V}$, parameters α_{IO}, α_{CPU}, α_{merge}
2: **Output**: optimal set of nodes $\mathcal{V}^*$
3: **for** each node a, starting from leaves, in a bottom-up fashion **do**
4:     **if** a is a leaf node **then**
5:         Compute $\mathcal{V}^*(a)$, $\sigma^*(a)$ and $\xi^*(a)$ using eq. 5.1
6:     **else**
7:         Compute $\mathcal{V}^*(a)$, $\sigma^*(a)$ and $\xi^*(a)$ using eq. 5.2
8:     **end if**
9: **end for**
10: **return** $\mathcal{V}^*(r)$

Algorithm 6 explains how to compute the optimal set of nodes by recursively computing the values $\mathcal{V}^*(a)$ for every node a, starting from the leaf nodes, based on the following theorem. The theorem uses two additional recursive functions $\sigma^*(a)$ and $\xi^*(a)$. The former represents the cost of $\mathcal{V}^*(a)$ and the latter represents $\sum_{j \in \mathcal{V}^*(a)} \xi(j)$. In the following theorem, r denotes the root of the S-QuadTree, $\gamma(a)$ denotes the children of a that also belong to $\mathcal{V}^*$ and $\mu(a)$ stands for $\sum_{j \in \gamma(a)} \xi^*(j)$ if $|\gamma(a)| > 1$, and is 0 otherwise.

Theorem 5.3.1. $\mathcal{V}^* = \mathcal{V}^*(r)$ can be recursively computed using recurrences 5.1 and 5.2. Furthermore, complexity of computing this is linear in the number of nodes of the S-QuadTree.

For leaf node a,

$$\mathcal{V}^*(a), \sigma^*(a), \xi^*(a) = \begin{cases} \{\}, 0, 0 & : a \notin \mathcal{V} \\ \{a\}, cost(a), \xi(a) & : a \in \mathcal{V} \end{cases} \tag{5.1}$$

For non-leaf node a,

$$\begin{array}{l} \mathcal{V}^*(a), \\ \sigma^*(a), \\ \xi^*(a) \end{array} = \begin{cases} \{\}, 0, 0 \quad : \text{if } a \notin \mathcal{V} \\ \left.\begin{array}{l} \{a\} \\ cost(a) \\ \xi(a) \end{array}\right\} \text{if } cost(a) < \mu(a) + \sum_{j \in \gamma(a)} \sigma^*(j) \\ \left.\begin{array}{l} \bigcup_{j \in \gamma(a)} \mathcal{V}^*(j) \\ \sum_{j \in \gamma(a)} \sigma^*(j) + \mu(a) \\ \sum_{j \in \gamma(a)} \xi^*(j) \end{array}\right\} \text{otherwise} \end{cases} \tag{5.2}$$

Proof. The cases for nodes not in $\mathcal{V}$ are obvious and included for completeness. We will discuss the other cases with the help of Figure 5-4. When a is a leaf-node in $\mathcal{V}$, e.g., the left child of E, the optimal $\mathcal{V}^*(a)$ must be $\{a\}$ since it is the only option. $\sigma^*(a)$ and $\xi^*(a)$ are accordingly assigned.

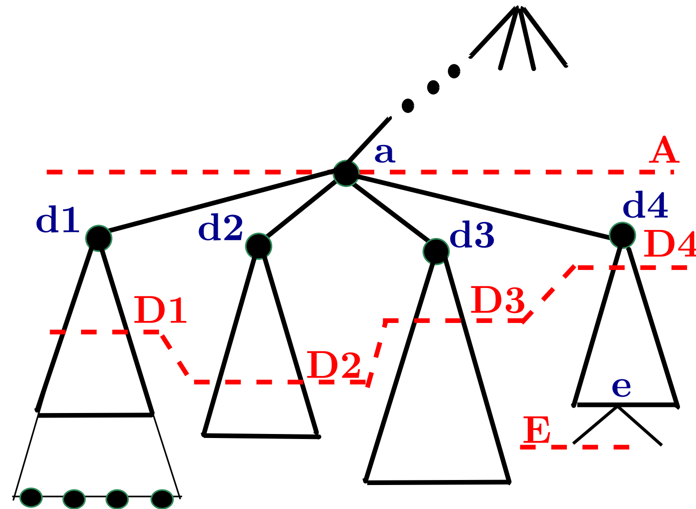


Figure 5-4: Selecting optimal $\mathcal{V}^*$ in S-QuadTree

For the other cases, consider any internal node in $\mathcal{V}$, e.g., a in Figure 5-4, with four children. As the Figure shows, there are two possibilities for choosing $\mathcal{V}^*(a)$. Either it contains a (second case of recurrence 5.2), in which case it should not contain any other node since every spatial object associated with any node in the subtree of a is already associated with a. Otherwise, if $a \notin \mathcal{V}^*(a)$ (third case of recurrence 5.2), then we should select the best possible sets in each of d_i. The latter are nothing but D_i (note that some of them could be empty) since we have to cover all spatial objects associated with nodes in $\mathcal{V}_L$ and, furthermore, any set of nodes in d_i other than D_i has a cost larger than that of D_i. The cost of selecting a in the first case would be simply $cost(a)$. The cost in the second case involves (i) the cost for selecting the D_is as well as (ii) that of merging their $E\text{-}list$s. The cost for case (ii) is 0 if there is at most one $E\text{-}list$ and otherwise, is proportional to the total number of objects in all those $E\text{-}list$s. $\mu(a)$ computes this exact quantity; notice that $\mu(a) = 0$ if all but one D_i are empty and one D_i has only one node.

Since $|\gamma(a)| \leq 4$, $\mu(a)$ can be computed in constant time and so can be $\sigma^*(a)$, $\xi^*(a)$, $\mathcal{V}^*(a)$ (assuming an efficient implementation of union operation for sets). Therefore, $\mathcal{V}^* = \mathcal{V}^*(r)$ can be computed in linear time in the number of nodes in the S-QuadTree. $\square$

Algorithm 6 gives pseudo-code of our dynamic programming algorithm that essentially uses the recurrences in Theorem 5.3.1 to compute $\mathcal{V}^*$ in an efficient manner. These optimal sets of nodes are selected for each block of values retrieved from the *driver* sub-query to be subsequently used for filtering.

Performing the Spatial Join: Next in order to perform spatial join STREAK retrieves spatial objects from the driven sub-query. Naturally *driven* sub-query needs an optimal plan for efficient execution. It determines the optimal plan adaptively using a technique coined *APS* (explained next in Section 5.3.3). It then performs join between spatial objects retrieved from the driver and driven sub-queries using S-QuadTree starting at $\mathcal{V}^*$ as described below. At each level, objects retrieved from the driver and driven sub-queries — coined *driver object* and *driven object* — can overlap with the MBR of none, one or several nodes:

- **Overlap with no MBR:** If (i) driver object and driven objects do not overlap with

a common MBR, and (ii) driven object is not contained in the node overlapped by driver object, then safely filter both driver and driven objects. This is because driver and driven objects do not intersect with each other and hence can be excluded from further consideration.

- **Overlap with one or more MBRs:** If (i) driver object and driven object overlaps with a common MBR, and (ii) driven object is contained in the node overlapped by driver object, then the algorithm recursively checks the children of the node till the level where the diagonal length of the node's MBR is equal to the query distance.

In brief the algorithms keeps finding whether the driver and driven objects intersect or not till it reaches a level – where the diagonal length of the node's MBR is equal to the query distance.

Refinement Step: The spatial join gives pairs of candidate spatial objects, however recall that STREAK approximated the spatial objects by minimum bounding rectangles (MBRs), hence the distances measured need to be validated. Moving forward we pass these candidate spatial objects to *refinement step* which is performed at the end of every block-wise query execution cycle. During *refinement step*, STREAK validates the distance join constraint using object's exact representation.

5.3.3 Adaptive Query Processing for Top-K Spatial Joins

We begin by first discussing the data access choices available to STREAK. A naive way would be to retrieve all tuples from the driver and driven sub-queries and join them. This would naturally will be inefficient for complex queries on large datasets. Another alternative could be to join the driver and driven sub-queries in a tuple-at-a-time manner. This would again be inefficient for complex RDF queries containing many joins because of (1) increased number of function calls required (2) pruning performed by each join operator. The third alternative is to use popular block-wise query execution, which amortizes access costs over a number of tuples [109]. It is to be noted that AQP used in Quark-X 4 used to only replace indexes in the driven sub-query with the operation tree remaining as-it-is,

whereas AQP used in STREAK changes the operation tree of the driven sub-query completely.

The complete block-wise query execution pipeline of STREAK is shown in Figure 5-5. STREAK begins by accessing driver sub-query in block-wise manner, by retrieving block blk. It then joins this block with the driven sub-query to find the complete score of tuples based on the given ranking function. These scores are used to compute the threshold θ, which is the score of kth result tuple, and the algorithm stops after k results have been found with score above θ (for descending). Observe that the driven sub-query can joined either block-wise (by retrieving block blk') or using full-scans. These plans, termed as *N-Plan* and *S-Plan* respectively, are adaptively chosen at run-time based on cost-estimates (described later).

Block-wise query processing brings in two more advantages during this phase. First, operating at block-wise granularity offers a balance between low-overhead of selecting different plans for different portions of input relations (*blocks*), according to ranking functions, and reasonable accuracy of cost estimation. Second, the block-wise bounds of numerical attributes and characteristic set information stored in the nodes of the S-QuadTree enable us to perform early termination and avoiding self-joins, respectively. The primary function of our adaptive query plan generation algorithm, called **Adaptive Processing for Spatial filters or APS** (pronounced "apps"), is thus to identify and route blocks of tuples through customized plans based on statistical properties, relevant to the query execution strategies.

The traditional approach to query optimization involves choosing an optimal plan based on statistics, and using that selected plan during query execution. This, however, would end up processing and producing (1) far more intermediate results than is necessary (2) the entire input cardinality – which is expensive. Previous top-k query optimization work [52] attempted to solve top-k plan generation problem based on a strong assumption that each tuple in driver sub-query is equally likely to join with tuples in driven sub-query. However, this does not frequently hold in tightly coupled triple patterns found in SPARQL queries. Instead, we argue that an adaptive query processing strategy (AQP) can be a better strategy for these workloads, as for each block retrieved it will help terminate early and determine the plan based on the selectivity of the block.

Challenges

Between an optimal plan selection of top-k spatial join queries with AQP and its corresponding efficient execution, lies several challenges that need to be overcome.

1. In the case of top-k queries, where spatial join essentially comprises a self-join between two parts of a dataset, cost estimation is difficult.

2. Different regions of the data have distinct statistical characteristics and a single plan for all regions performs poorly.

3. Early out feature of top-k queries requires estimation of input cardinalities of these index scans thus making it even harder to choose an optimal plan, and in general, poses many challenges in costing rank-join operators.

4. Given the verbose nature of RDF, finding optimal join order of SPARQL queries is challenging because the number of possible query plans is in the order of factorial of number of index scans required.

We will see in this section how the design of our indexes naturally enable us to devise a more effective, adaptive execution strategy for query processing.

Plan Selection:

We address the above mentioned challenges in STREAK, by first breaking down the plan selection problem into that of finding the least-cost for the **driver plan** and the **driven plan**. We use an adaptive cost-based optimization approach to select the query plan which adaptively chooses the driven plan during query execution. Similar to commercial cost based optimizers like SQL-Server we push filters deep in the query plan. STREAK allows us to do spatial filtering using its S-Quad tree and numerical filtering using the early termination condition of top-k queries. Amongst these two we choose the most selective filter. How S-Quadtree and numerical predicates help in performing filtering is described next.

The need for adaptively choosing customized plans for driven sub-query can be well motivated using our running example, which retrieves the top-most wine producing regions

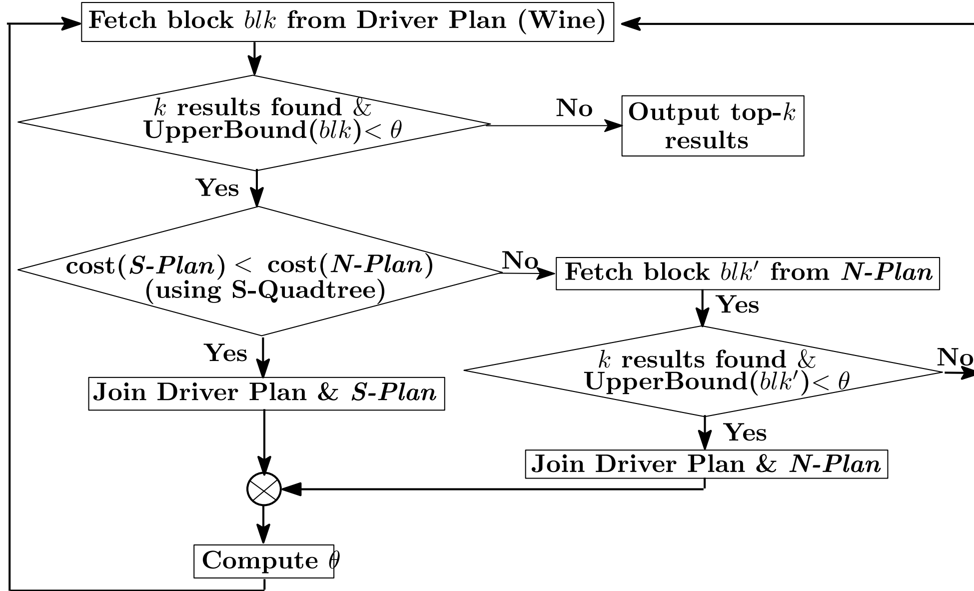


Figure 5-5: Query Processing Flow-Chart

that are located within $d\ kms$ of a river. This involves a spatial join between regions containing vineyards (*driver* sub-query) and rivers (*driven* sub-query). In our example, the wine growing regions are the Gobi desert in China, which has no river flowing across it, and Baden in Germany that has its vineyards directly overlooking the Rhine river. An efficient implementation of top-k spatial distance join operator should therefore perform spatial join with Gobi desert first, as the spatial join will help in filtering Gobi desert. We call the driven plan generated by pushing spatial filtering using S-Quad tree deep in query execution, as **S-Plan**. This is shown in Figure 5-6(B), with driver and driver sub-query shown in Figure 5-6(C) and (D), respectively. Note that S-Plan significantly reduces query execution time when the spatial join operator is highly selective, this is because STREAK S-Quadtree helps perform early filtering.

Secondly, the spatial-join operator should check for early-termination first, when blocks containing spatial regions like Baden are retrieved, as Baden is not the top-most wine growing region. Hence pushing numerical predicates deep in the query plan will help filter such tuples. We call such a driven plan generated by pushing filtering using numerical predicates deep in the query plan as an **N-Plan**, as shown in Figure 5-6(A), with its driver and driven plans shown in Figure 5-6(C) and (E) respectively.

For top-k spatial join workloads both scenarios are highly common.

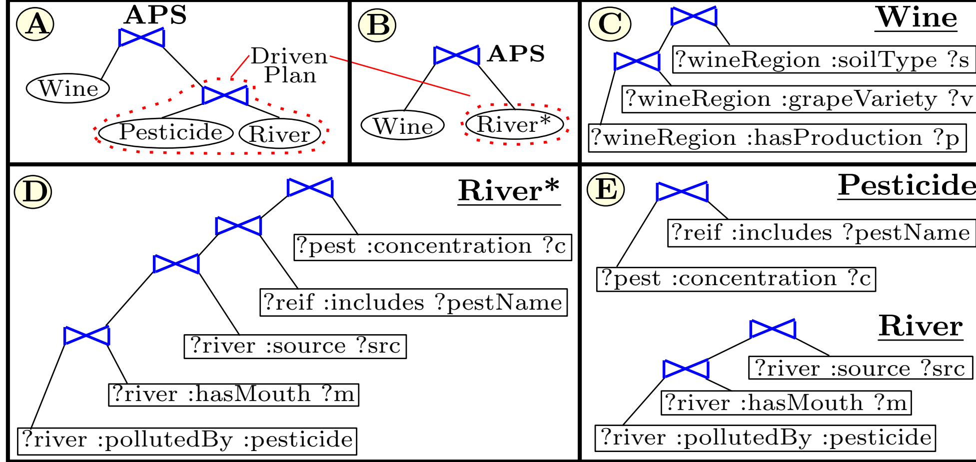


Figure 5-6: Possible Plan Choices: (A) N-Plan, with rivers filtered first by numerical predicate (pesticide concentration), (B) S-Plan, with a spatial join between vineyards and rivers done first, (C) driver plan, (D) Plans supporting S-Plan heuristics, and (E) Plans supporting N-Plan heuristics

STREAK then computes the cost of the two plans and routes the tuples through least cost plan. STREAK's block-wise query execution enables it to switch among the two plans at runtime – with zero cost of switching plans at materialization points — points where next blocks are retrieved and executed in driver sub-query. At these points STREAK performs the spatial join between the newly retrieved blocks from the driver and driven plan respectively, which means that the work done by earlier blocks does not go wasted. This helps STREAK overcome the drawbacks of earlier top-k adaptive query processing approaches that wasted substantial amounts of work at materialization points [52] and used limited AQP by switching only indexes at run-time [57]. In contrast, STREAK switches entire customized plans at runtime.

Cost Model

We now discuss STREAK's cost-model for choosing amongst driven plans. The cost function of our spatial join operator takes into account the cardinality of the spatial join. Greater the cardinality of the spatial join, the greater we expect the number of candidate results to be and thus higher the query execution time. We calculate the cardinality of the join to be a product of the cardinality of spatial characteristic sets stored in the nodes of the S-QuadTree, and the number of tuples retrieved from the driver plan (as discussed in sub-

section 5.3.2).

STREAK uses a simple cost model to estimate the cost of its customized N-Plan and S-Plan. Recall that the N-Plan pushes numerical predicates deep in the query plan and uses block-wise evaluation to achieve early-termination, while the S-Plan pushes spatial join evaluation deep in the query plan and uses full-scans for evaluating the join. For the rest of the section, we denote the time of execution of driven plan R as $T(R)$, and the result cardinality as $C(R)$. Let R and R_i denote full and block-wise executions of S-Plan and N-Plan, respectively. R_i corresponds to R when either (a) all blocks are retrieved, or (b) early-termination is achieved. Putting R and R_i together, we have the following equation where x is the estimated number of blocks required to be retrieved before early-terminating is achieved.

To estimate time cost,

$$T(R) = \begin{cases} x.T(R_i) & : x < R/R_i \\ T(R) & : x \geq R/R_i \end{cases} \tag{5.3}$$

We estimate R_i's result cardinality as $C(R_i) = x.C(R)/nb$, where 'nb' is the number of blocks of numerical predicate. The intuition of the equation is as follows: when number of blocks is 1, then $C(R_i)$ is $C(R)$ i.e. it is equivalent to all intermediate results without any early pruning. We assume that all buckets contribute to $C(R)$'s results equally. So when only a portion of them are selected, $C(R_i)$ is proportional to the fraction of blocks that are selected for early termination, which is characterized by x/nb. To estimate 'x', we find all blocks whose upper bound is greater than the threshold.

We next route the tuples through the plan with the least cost, and thus at runtime seamlessly keep alternating between the two customized plans for each new block of tuples retrieved from the driver sub-query. It may seem at first glance that the cost of *N-Plan* will be lower than *S-Plan* as the cardinality of *N-Plan* is always less than equal to the cardinality of *S-Plan*. However, we found this to not be the case because repeated scans, owing to block-wise query processing, increases the cost of *N-Plan*. We later show experimentally in Section 5.5, that our proposed *APS* algorithm does in fact help select the least cost customized plan, and accelerates query processing times by one to two order(s) in magnitude.

5.4 Evaluation Framework

STREAK is implemented in C++, compiled with g++-4.8 with -O3 optimization flag. All experiments were conducted on a Dell R620 server with an Intel Xeon E5-2640 processor running at 2.5GHz, with 64GB main-memory, and a RAID-5 hard-disk with an effective size of 3TB. For all the experiments, the OS can use the remaining memory to buffer disk pages. In our experimental evaluation, we report cold-cache timings after clearing all filesystem caches and by repeatedly running the query processor with the same query 5 times, and taking the average of the final 3 runs. For warm cache numbers, we repeat the same procedure but without dropping caches.

We perform a direct comparison of STREAK with other full-fledged systems by ensuring the use of spatial indexes for the workloads used in the chapter. In Virtuoso, we employ a two dimensional R-tree implementation for indexing the spatial data and on postgres we have built the “gist” (generalized search tree) index for supporting spatial data. Postgres supports varied geometry types like POINTS, POLYGONS and LINESTRINGS, while Virtuoso only supports POINT geometries [61]. The number of results returned by Postgres and Virtuoso is constrained by using LIMIT ‘k’ in the query.

5.4.1 Datasets

We used two widely used, real-world datasets: YAGO3 Core [65] and Linked Geo Data(LGD) [11] for evaluating the performance of STREAK in comparison with other systems. LGD contains freely available collaboratively collected data from Open Street Map project. YAGO contains facts extracted from Wikipedia, GeoNames and WordNet. Table 5.1 shows the characteristics of the two datasets. We observe that YAGO and LGD datasets contain 85 million and 324 million quads respectively. The size of the quad-tree used for storing YAGO and LGD datasets, is 37 MB and 119 MB respectively, about 0.04% and 2% of size of raw data. This is despite the fact that 50 percent of facts in LGD contain description about spatial objects, consisting of rich suite of geometries such as POINTS, POLYGONS and LINESTRINGS.

YAGO [65] has supplementary information like time and confidence scores attached to

Table 5.1: Dataset Characteristics

Dataset	quads	points	lineStrings	polygons
YAGO	$85M$	$453K$	0	0
LGD	$30M$	$590K$	$2.6M$	$264K$

facts, representing the confidence of the extraction algorithm in extracting facts. We extend this to create a more realistic scenario, where a confidence score is attached to facts using exponential and uniform distributions. We find that results with uniform distribution follow a similar trend, so for sake of brevity we report the results only for exponential distribution.

5.4.2 Benchmark Query Workloads

Numerous performance benchmarks have been created in the recent years for processing RDF / SPARQL queries. However, none of them cater to queries involving reification (or named graphs), spatial and top-k evaluation. Therefore, we design a set of benchmark queries containing query features that have been found important in literature [7, 61, 57], namely — **structural** and **statistical** features. Note that the features used for benchmarking STREAK are same as Quark-X described in Section 4.9.2 with the addition of the following spatial feature:

Type: type of spatial extent e.g. polygon, line string, point, etc. This feature is interesting because it tests the ability of the spatial database to deal with different types spatial objects with varying complexity.

Table 5.2 shows the features of the 8 benchmark queries which we have created for both LGD and YAGO datasets. While YAGO consists of only point objects, LGD consists of different types of spatial objects and hence, the benchmark for LGD is more diverse comprising spatial joins of different object types. For both data sets, it is noteworthy that these queries provide a comprehensive coverage of all key features. For simplicity, we present results for linear ranking function, although our system STREAK is aimed at convex monotonic functions.

Query id	Shape	Type	Left Card.	Right Card.	# TP	# Quant TP	# Non Quant TP	Join Vertex Count	Join Vertex Type
1	Complex	Point Polygon	3,485	13,090	6	2	4	4	(SS, RS)
2	Complex	Point Point	3,485	1,189	6	2	4	4	(SS, RS)
3	Complex	Point Point	3,485	524	7	2	3	4	(SS, RS)
4	Complex	Point Point	25,617	524	9	2	7	4	(SS, RS)
5	Complex	Point Polygon	13,090	524	9	2	7	4	(SS, RS)
6	Complex	Point LineString	13,090	2,601,040	6	2	4	4	(SS, RS)
7	Complex	LineString Point	3,485	2,601,040	6	2	4	4	(SS, RS)
8	Complex	Polygon LineString	13,090	2,601,040	7	2	5	4	(SS, RS)

(a) LGD

Query id	Shape	Type	Left Card.	Right Card.	# TP	# Quant TP	# Non Quant TP	Join Vertex Count	Join Vertex Type
1	Star	Point Point	24,749	294,969	6	2	4	2	(SS)
2	Star	Point Point	106	294,969	8	3	5	2	(SS)
3	Star	Point Point	27	294,969	7	2	5	2	(SS)
4	Star	Point Point	266	294,969	8	3	5	2	(SS)
5	Complex	Point Point	164,073	468,041	8	2	6	6	(OS, RS)
6	Complex	Point Point	3,740	294,969	7	2	5	3	(OS, SS, RS)
7	Complex	Point Point	105	598,063	6	2	4	3	(SS, RS)
8	Complex	Point Point	64,295	64,295	7	3	4	5	(OS, RS, SS)

(b) YAGO

Table 5.2: Characteristics of Benchmark Queries

5.5 Experimental Results

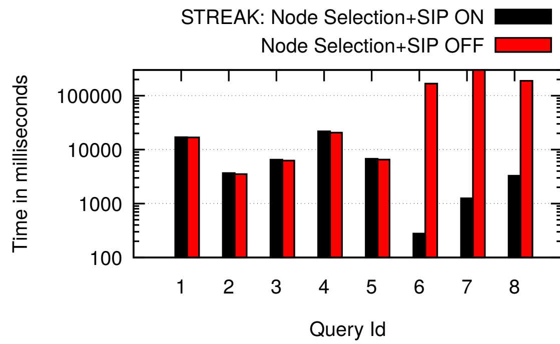
In this section, we present a comprehensive experimental evaluation of STREAK along the following dimensions: (A) the performance of spatial filter processing. Within this, we study (i) the impact of sideways information passing and (ii) the performance of using the S-QuadTree based join over synchronous R-tree traversal based join. (B) end-to-end holistic performance combining all features of STREAK (C) the advantages of early-out top-k processing in STREAK. Unless stated otherwise the results we present in this section correspond to a default setting of $k = 100$.

5.5.1 Performance of Spatial Join Processing in STREAK

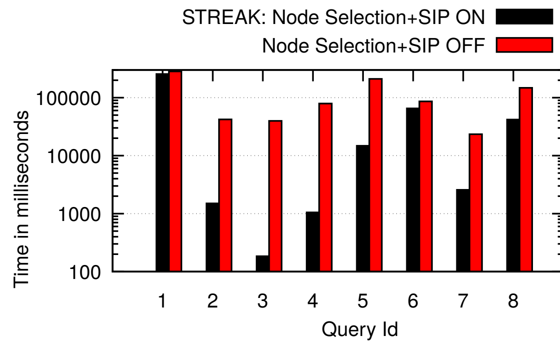
Impact of Sideways Information Passing and Optimal Node-Selection Algorithm

The node selection algorithm determines the optimal set of nodes of the S-QuadTree in order to determine the *I-Range* and *E-list* objects, which subsequently helps filter and narrow down the *driven* sub-query in sideways information passing style. Note that we use S-QuadTree by keeping it fully memory-resident (although it is serialized to disk). Therefore we study the impact of node selection algorithm only under warm-cache setting. Figure 5-7 shows the performance with and without the use of sideways information passing (SIP), powered by the node selection algorithm over S-QuadTree. Specifically, these plots show the time, in milliseconds plotted along Y-axis in log-scale, taken for completing all the benchmark queries over YAGO and LGD datasets (X-axis).

Looking at individual queries in Fig. 5-7, we observe considerable improvement in query execution performance for all queries over YAGO dataset as well as Q_6 & Q_8 of LGD when SIP and node selection algorithms are enabled. This is due to the fact that these queries are highly selective at the spatial join, making the use *I-Range* and *E-list* objects for filtering the intermediate results from the driven plan highly effective. It can reduce the query processing time by upto *3 orders of magnitude*. On the other hand, the queries Q_1 to Q_5 in LGD have low selectivity, with very little impact of SIP in skipping irrelevant entries from the driven plan.

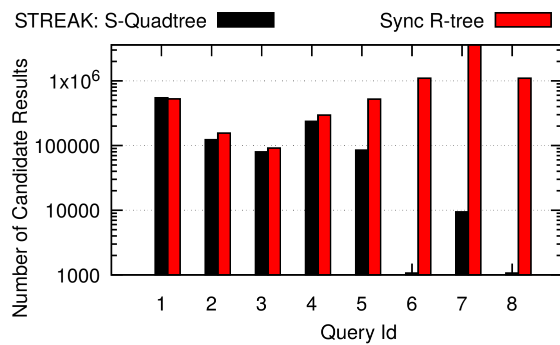


(a) LGD

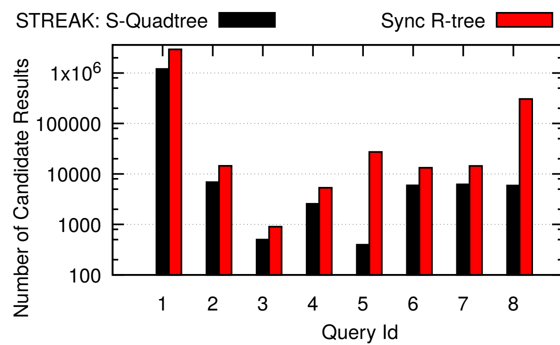


(b) YAGO

Figure 5-7: Effect of Sideways Information Passing



(a) LGD



(b) YAGO

Figure 5-8: S-QuadTree Vs. Sync. R-tree for Spatial Join

Comparison of Spatial Join Algorithms

In the next set of experiments, we compare the performance of spatial join processing using S-QuadTree as opposed to using synchronous R-tree traversal based spatial join algorithm. Towards this, we incorporated the state of the art implementation of synchronous R-tree traversal join algorithm released by Sowell et al. [87] into the STREAK system, and provided a run-time switch that can choose synchronous R-tree in place of S-QuadTree.

Figure 5-8 plots the number of candidate results generated by the two algorithms (along Y-axis in log-scale) for all the benchmark queries. From these results it is immediately apparent that S-QuadTree generates smaller number of candidates, sometimes *2 orders of magnitude* less –e.g., as in LGD Q_6 and YAGO Q_5– than sync. R-tree based method. This can be attributed to the following advantages of S-QuadTree:

- The use of *characteristic set* information within the nodes of the S-QuadTree helps to reduce the number intermediate results significantly in comparison to standard spatial indexes such as R-trees.
- Encoded identifiers are able to achieve a limited granularity owing to the fixed size of identifiers. These identifiers when decoded back suffer from a loss of precision, which in-turn affects the filtering efficiency of spatial join algorithms such as Synchronous R-tree traversal. However, STREAK owing to its use of *I-Range* and *E-list* objects is able to overcome this situation and is thus able to filter efficiently.
- While both spatial join methods lack a well defined way of filtering the results in the driven plan as it is, the use of sideways information passing with S-QuadTree significantly reduces the cardinality of intermediate results.

5.5.2 Comparison with Database Engines

We now turn our attention to the comparison of end-to-end system performance of STREAK with two state of the art publicly available database engines. Specifically, we compare with (i) *PostgreSQL* [78]: a relational database system that has an excellent in-built support for spatial indexing via its generalized search tree (GiST) framework; and (ii) *Virtuoso* [93]: a state of the art commercial RDF processing engine that has a number of performance

Table 5.3: **On-Disk Database Size for YAGO and LGD**

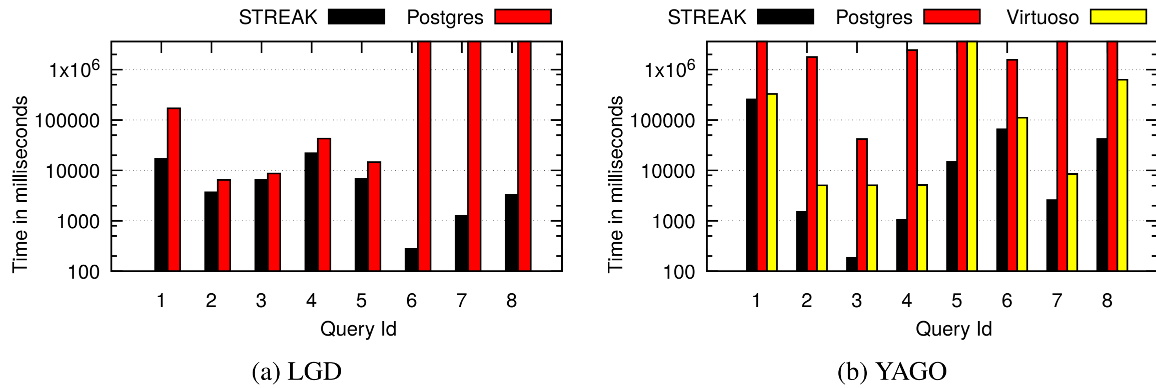
Database Engines	**YAGO** Size (GB)	**LGD** Size (GB)
STREAK	14	**6.4**
PostgreSQL	54	22
Virtuoso	**11**	—NA—

optimizations such as vectorization as well as cache-conscious processing specifically useful for RDF/SPARQL processing. Note that our current implementation of STREAK has neither the vectorized processing nor the cache-conscious features, although these are part of our future work plans.

Before we discuss the query processing performance, we briefly present the effective size of databases created by the three engines summarized in Table 5.3. As we already mentioned in Section 5.4, Virtuoso could not be used for LGD dataset since it lacks support for varied geometry types such POINTS, POLYGONS and LINESTRINGS. At 11 GB of database size, Virtuoso, which employs highly compressed encoding, is the most compact database for YAGO. While STREAK also has a comparable database size (14 GB), PostgreSQL on the other hand has a significantly larger database size –almost **5**× larger. Similar observations also hold for LGD between STREAK and PostgreSQL database sizes.

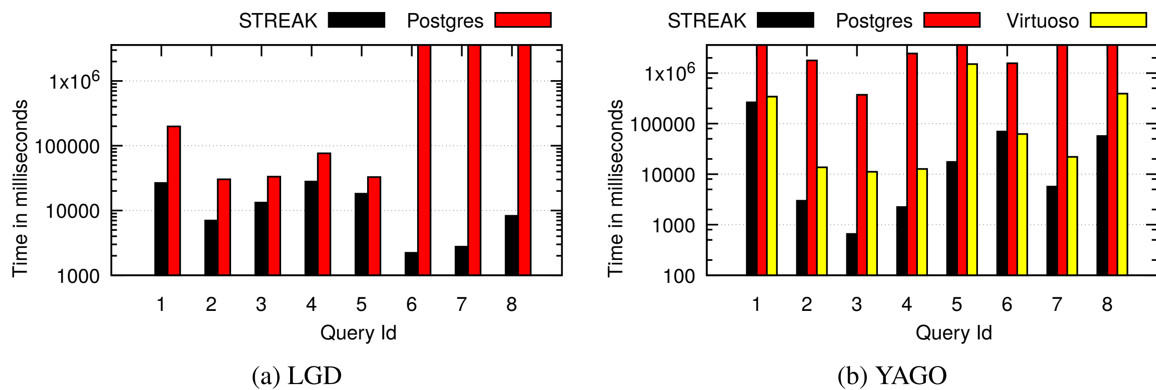
We summarize the comparative query processing performance of the three engines in Figure 5-9 and Figure 5-10 for warm cache and cold cache, respectively. The Y-axis in these plots corresponds to the end-to-end wall-clock time in milliseconds for each query, and is plotted in log-scale.

From these results, we can observe that STREAK outperforms both PostgreSQL and Virtuoso on all queries, by a significant margin. For a number of queries –*viz.*, $Q_6 - Q_8$ of LGD and Q_5, Q_7, Q_8 of YAGO – PostgreSQL could not complete within its allotted time. Virtuoso failed to complete for Q_5 of YAGO as well. The poor performance of PostgreSQL can be attributed to its preference to choose nested loop joins, due to its relatively weak cost models for RDF. For Virtuoso, however, we cannot comment on its poor performance as it is closed source and no published work describes its functionality and query optimization techniques.



(a) LGD (b) YAGO

Figure 5-9: Performance of STREAK Vs. PostgreSQL and Virtuoso in Warm Cache



(a) LGD (b) YAGO

Figure 5-10: Performance of STREAK Vs. PostgreSQL and Virtuoso in Cold Cache

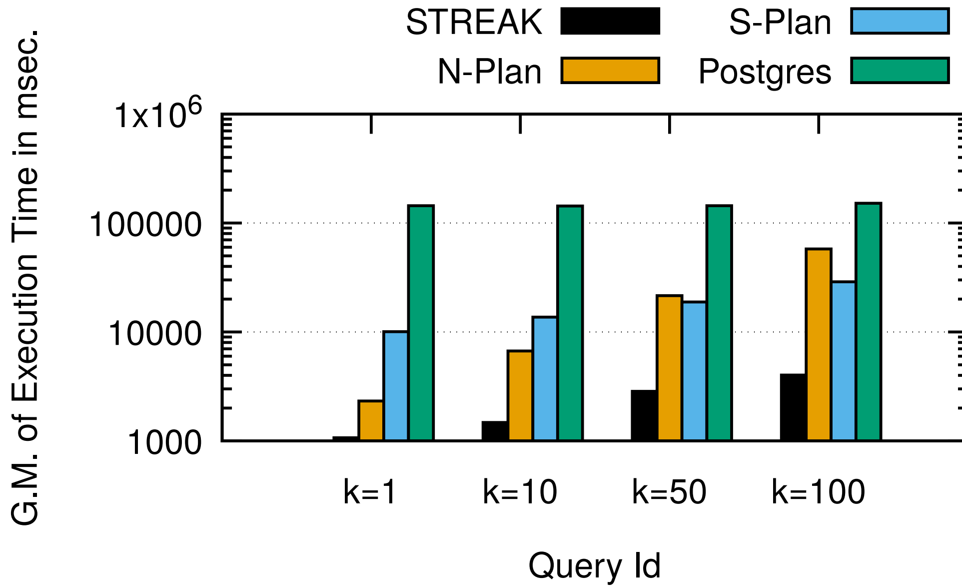
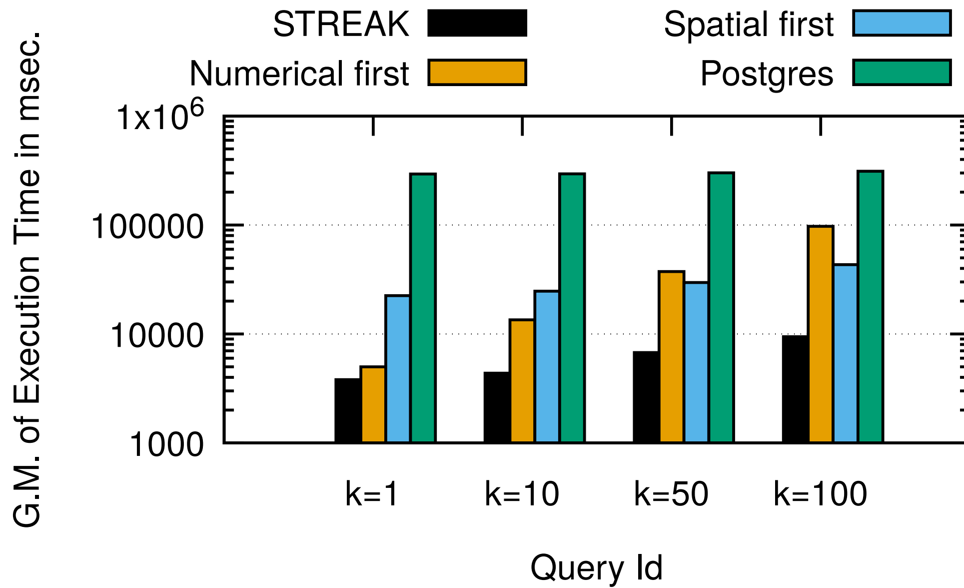


Figure 5-11: Performance with Varying k

5.5.3 Comparison with varying k

Finally, we briefly present the performance evaluation when k value is varied. Figure 5-11 and Figure 5-12 plots the geometric mean of runtime of all queries in the benchmark for $k = \{1, 10, 50, 100\}$ in warm-cache and cold-cache scenario for LGD, respectively. One can observe that STREAK outperforms PostgreSQL by a large margin for all values k. PostgreSQL is unaffected by the variation in the value of k since it simply performs the full evaluation followed by sort in all cases.

It is noteable to see the performance of our APS algorithm in comparison to fixed plans namely, *N-Plan* and *S-Plan*. For k = 1, 10 the *N-Plan* outperforms the *S-Plan*; while for k = 50, 100 the *S-Plan* outperforms the *N-Plan*. This is because of the need for scanning fewer blocks for lower values of k, thus making *N-Plan* a cheap option. While many blocks need to be scanned for larger values of k, thus making *S-Plan* a better option. It is noteworthy that STREAK shows outstanding performance in comparison to both *S-Plan* and *N-Plan*, owing to better plan selection and switching at run-time.



LGD on Cold Cache
Figure 5-12: Performance with Varying k

5.6 Related Work

In this section, we provide a full overview of techniques that overlap with the core aspects of STREAK. These primarily relate to RDF spatial joins, top-k joins, spatial top-k joins and adaptive query processing.

Spatial Joins

We now look at the work done specifically on spatial joins in both relational and RDF settings.

Parliament [14] is a spatial RDF engine built on top of Jena [53]. Parliament supports GeoSPARQL features and uses R-tree as the spatial index. Additional techniques to enhance spatial queries, as adopted by Geo-store [97] and in RDF engine proposed by Liagouris et al. [61] involve encoding spatial data by employing Hilbert space-filling curves for preserving spatial locality.

Another system, S-Store [98], extends state-of-the-art RDF store gStore [108], by indexing data based on their structure. Subsequently, queries are accelerated by pruning based on both spatial and semantic constraints in the query. Yet another spatiotemporal storage system is gst-Store [99], which is an extension of gStore [108]. It processes spatial queries using its tree-style index ST-tree, with a top-down search algorithm.

However, the above systems, while capable of handling spatial filters in the query, are not directly amenable for either top-k processing or adaptive querying. They miss out features to early-prune intermediate results and to process unsorted orders on spatial attributes (imposed by top-k query processing), which are critical for optimizing the workloads seen here. STREAK in stark contrast provides efficient ways of supporting top-k spatial joins using its customized S-QuadTree and query processing algorithms. This helps STREAK outperform state-of-the-art systems like Virtuoso by a large margin as shown experimentally in sub-section 5.5.2.

Relational approaches such as Synchronous R-tree traversal [24] and TOUCH [71] apply spatial-join algorithm on hierarchical tree-like data structure. TOUCH is an in-memory based technique, which uses R-tree traversal for evaluating spatial joins. STREAK differs from TOUCH in using identifier encoding and semantic information embedded within the datasets for early-pruning, and by utilizing statistics stored within the spatial index to choose the query plan at runtime for improving performance. Sync. R-tree traversal builds R-tree indexes over the datasets participating in the spatial join. Starting from the root of the trees, the two datasets are then synchronously traversed to check for intersections, with the join happening at the leaf nodes. Inner node overlap and dead space are two shortcomings in this technique, which are addressed with STREAK's S-QuadTree that uses space-oriented partitioning. We compare STREAK with sync. R-tree traversal in Section 5.5.

on-indexed approaches such as Plane-Sweep algorithm [30], and Partition-based Spatial Merge Join(PBSM) [73] apply spatial join algorithms on non-indexed inputs. PBSM is a disk-based algorithm, which employs grid-based partitioning of the dataset, with spatial objects being replicated in overlapping cells of the grid. To evaluate the query, the plane-sweep algorithm is used, which sorts the dataset in one dimension and sweeps a line across them in the sorted dimension. PBSM suffers from data skew which increases the number of comparisons required, and plane-sweep loses locality along the second dimension. STREAK on the other hand avoids these problem by (1) encoding identifiers and minimaly replicating spatial objects in nodes of the S-QuadTree. Replication helps in storing spatial objects at the finest granularity, and hence helps improve the filtering capability of spatial join algorithms. Note that STREAK's smart use of encoded identifiers helps achieve this at

the cost of minimal memory footprint (2) speeding-up comparisons using the encoded identifiers with binary search, as opposed to finding intersections amongst Minimum Bounding Rectangles. The effectiveness of STREAK against the above approaches were captured in Section 5.5.

Top-k Joins

Hash-based Rank Join (HRJN) [52] and Nested Loop Rank Join (NRJN) [49] are two widely used top-k join algorithms in the relational world. HRJN accesses objects from left (or right) side of join and joins them with tuples seen so-far from right (or left) side of join. All join results are fed to a priority queue, which outputs the results to the users if they are above threshold, thus producing results incrementally. Nested Loop Rank Join (NRJN) algorithm is similar to HRJN, except that it follows a nested loop join strategy instead of buffering join inputs. We do not compare STREAK with a general top-k join algorithms, HRJN and NRJN, since such top-k operators for such spatial workloads have been shown to perform poorly when compared to a block-based algorithm [79] and instead compare with the state-of-the-art spatial top-k algorithm.

Quark-X [57] is a recently proposed top-k query processing engine, which uses summarized numerical indexes for retrieving and ranking RDF facts. Quark-X showed experimentally that it outperformed the other top-k query processing systems by a large margin. However, Quark-X is suited only for 1-dimensional data, as its techniques can not be applied for multi-dimensional data (sorted order is a requisite for Quark-X). Hence, its approach leads to many unnecessary comparison operations in the non-sorted dimension. STREAK in contrast reduces the number of comparisons required drastically, by using its schema-aware S-QuadTree.

Spatial Top-k Joins

Returning the top-k results on spatial queries, where the final results are ranked by using a distance metric has been already studied by Ljosa, et al. [63]. However, such ranking mechanism often have to be restricted to specific, pre-defined aggregation functions. In comparison, STREAK attempts to solve a more challenging problem with the spatial constraint specified in the join predicate.

To the best of our knowledge, the work closely related to STREAK is done by *Qi et*

al., [79]. In this approach *Qi et al.,* retrieve blocks of data ordered by object scores, from input datasets, which are then spatially joined. Their approach: (1) requires data to be sorted based on the ranking function, making it unsuitable for arbitrary user-defined ranking function. STREAK, in contrast, does not impose any such limitation on the data. (2) only supports spatial joins and would require non-trivial modifications for supporting complex patterns seen in SPARQL queries [101, 69, 70]. STREAK in comparison evaluates complex queries efficiently. (3) uses *only* block-wise retrieval for joining. Block-wise retrieval is expected to be fast only if the results are found in first few block accesses. However, such an approach suffers high overheads when blocks deep in the sorted collections need to be retrieved. STREAK circumvents this drawback by adaptively switching between its customized plans at run-time (described in sub-section 5.3.3). (4) incur additional index creation cost at runtime, as they build spatial indexes at run-time. STREAK, on the other hand, builds its spatial index S-QuadTree during pre-processing stage, thus incurring zero index creation overhead during query processing.

Adaptive Query Processing

Eddies [13] and Content Based Retrieval (CBR) [17] are two state-of-the-art approaches which explored AQP for switching plans during query execution. Both these approaches use random tuples for profiling, relying on a machine-learning based solution to model routing predictions. Our work directly contrasts ML-based approaches by using statistics for each *block* of spatial data, which helps in determining the selectivity of spatial join operator. By using fine-grained block-level statistics, we mitigate the errors that are typically associated with a model-based approach, especially when there are many joins involved. Additionally, unlike CBR, which has high routing and learning overhead, our spatial AQP algorithm incurs a very small routing overhead, owing to cost estimate calculations for only the customized plans. We validated this argument and found the overhead of AQP to be very small — just 5-10% of the overall overhead. Finally, routing predictions of Eddy and CBR depend on the model used and the size of the training datasets, while our statistics based approach does not require modeling and is relatively unaffected by the training dataset size.

5.7 Discussion & Outlook

This chapter introduces STREAK, an efficient top-k query processing engine with spatial filters. The novel contributions of this work are its soft-schema aware S-QuadTree spatial index, new spatial join algorithm which prunes the search space using its optimal node selection algorithm, and adaptive query plan generation algorithm called APS which adaptively switches query plan at runtime with very low overhead. We show experimentally that STREAK outperforms popular spatial join algorithms as well state-of-the-art end-to-end engines like PostgreSQL and Virtuoso.

5.7.1 Outlook

The present work opens up several exciting directions for future research, some of which are discussed below:

- As part of future work, an exciting direction could be to design a top-k RDF reasoner with spatial filters. Growing volume of RDF data coupled with the need for early termination in ranking systems entail efficiency and scalability challenges, when developing top-k reasoning systems. Therefore investigating the design of top-k RDF reasoner with spatial filters is an attractive direction for future research.

- Investigating the performance of S-QuadTree in distributed networks for join-ahead pruning and for performing the spatial join is another interesting direction for future research.

- Another appealing research direction – critical for many real world applications – is to study top-k query processing on moving objects. Although STREAK supports bulk updates, its comprehensive evaluation with frequent in-place updates is an interesting open issue.

Chapter 6

Conclusions and Future Work

In this thesis we presented efficient frameworks for executing SPARQL queries over RDF datasets containing higher order relationships associated with facts. We comprehensively tested the developed frameworks and showed their superior performance against state-of-the-art systems over real world datasets .

RQ-RDF-3X [56] : RQ-RDF-3X presents extensions to triple-store style RDF storage engines to support reification and quads. In RQ-RDF-3X, we support meta-knowledge about triples by assigning a unique identifier (R) to each (S, P, O) triple. Thus, the fundamental change required is to support an additional field (R) that has triple identifier. The inclusion of this additional field requires more joins, thus complicating query optimization and increasing query execution cost. This requires careful re-thinking of existing indexing and query optimization approaches adopted by state-of-the-art triple stores. We demonstrate experimentally that RQ-RDF-3X achieves one to two orders of magnitude speed-up over both commercial and academic engines such as Virtuoso [93], RDF-3X [69], and Jena-TDB [53] on real-world datasets - YAGO and DBpedia.

QUARK-X [57]: QUARK-X is an efficient top-k query processing framework for RDF quad stores. The contributions of QUARK-X include novel in-memory synopsis indexes for quantifiable predicates. This is in the same spirit as building impact-layered indexes in information retrieval but carefully redesigned for use for ranking in reified RDF. Additionally, QUARK-X proposes a novel Rank-Hash Join (RHJ) algorithm designed to utilize the synopsis indexes, by selectively performing range scans for quantifiable facts early on –

this is crucial to the overall performance of SPARQL queries which involve a large number of joins. We show experimentally that QUARK-X achieves one to two magnitude speed-up over baseline solutions on YAGO and DBpedia datasets.

STREAK: STREAK is an efficient engine for processing top-k SPARQL queries with spatial filters. STREAK introduces various novel features such as a careful identifier encoding strategy for spatial and non-spatial entities for reducing storage cost and for early pruning, the use of a semantics-aware quad-tree [82] index that allows for early-termination and a clever use of adaptive query processing with zero plan-switch cost. It is designed to support a wide-range of queries with spatial filters including complex joins, top-k, and higher-order relationships over spatially enriched databases. For experimental evaluations, we focused on top-k distance join queries and demonstrated that STREAK consistently outperforms popular spatial join algorithms as well as state of the art commercial systems such as Virtuoso.

In summary, in this thesis we argue with empirical support that while RDF triple stores have been optimized for traditional RDF datasets, it is becoming more and more apparent that we need to support many sophisticated applications that use reification / quads. Signifying scope for further improving the RDF data management. These applications include knowledge representation applications that were the original driving force of RDF. Furthermore modern knowledge extraction systems such as HighLife, NELL, OpenIE etc., not only generate a triple but also associate confidence/context. And in some cases, the relationships themselves have higher arity. Therefore, the storage model and query optimizations we suggest in this thesis are of practical value in supporting applications built on top of such knowledge bases.

6.1 Future Work

The problems addressed in this work are only three among many that arise in the context of RDF data management. There are many opportunities for future research. We describe succinctly some promising and interesting directions below.

Query Optimizer: QUARK-X uses a heuristic-based model for plan generation, in

which quantifiable predicates are pushed deep in the query plan. However a heuristic based query optimizer may not always choose the optimal plan. Therefore the need for the query optimizer to be significantly improved for top-k queries.

RDF graph reasoner: As part of the future work it will be challenging to develop a RDF-graph reasoner which supports inference over RDF quads. Our graph reasoner would also have support for a plethora of queries such as top-k, spatial and temporal queries.

Decentralized processing: Some applications require decentralized processing, we believe that it is possible to extend our indexing techniques into a decentralized setup.

Appendix A

Queries

A.1 RQ-RDF-3X Benchmark Queries

A.1.1 YAGO

Query 1 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?c
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/isConnectedTo>.
  ?r rdf:object <http://yago-knowledge.org/resource/Shenyang>.
  ?r <http://yago-knowledge.org/resource/byTransport> ?c.
  ?c <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://yago-
     knowledge.org/resource/wikicategory_Airlines_established_in_1992>.
  ?c <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://yago-
     knowledge.org/resource/wikicategory_Airlines_of_the_People'
     s_Republic_of_China>
}
```

Query 1 (SQL):

```
        select
        d1.stringvalue,
```

```
d2.stringvalue
from orpsnumericyago2 n1,
         orpsnumericyago2 n2,
         orpsnumericyago2 n3,
         orpsnumericyago2 n4,
         dictionaryFull d1,
         dictionaryFull d2
where
         n1.predicate=(select id from dictionaryFull where
            stringvalue='"http://yago-knowledge.org/resource/
            isConnectedTo"') and
         n1.object=(select id from dictionaryFull where
            stringvalue='"http://yago-knowledge.org/resource/
            Shenyang"') and
         n1.reification=n2.subject and
         n2.predicate=(select id from dictionaryFull where
            stringvalue='"http://yago-knowledge.org/resource/
            byTransport"') and
         n2.object=n3.subject and
         n3.predicate=(select id from dictionaryFull where
            stringvalue='"http://www.w3.org/1999/02/22-rdf-
            syntax-ns#type"') and
         n3.object=(select id from dictionaryFull where
            stringvalue='"http://yago-knowledge.org/resource/
            wikicategory_Airlines_established_in_1992"') and
         n2.object=n4.subject and
         n4.predicate=(select id from dictionaryFull where
            stringvalue='"http://www.w3.org/1999/02/22-rdf-
            syntax-ns#type"') and
         n4.object=(select id from dictionaryFull where
            stringvalue='"http://yago-knowledge.org/resource/
            wikicategory_Airlines_of_the_People''
            s_Republic_of_China"') and
         n1.subject=d1.id and
         n2.object=d2.id;
```

Query 2 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?c?e?d
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/
     holdsPoliticalPosition>.
  ?r rdf:object <http://yago-knowledge.org/resource/Governor_of_New_York
     >.
  ?r <http://yago-knowledge.org/resource/hasPredecessor> ?b.
  ?r <http://yago-knowledge.org/resource/hasSuccessor> ?c.
  ?r1 rdf:subject ?c.
  ?r1 rdf:predicate <http://yago-knowledge.org/resource/hasWonPrize>.
  ?r1 rdf:object ?e.
  ?r1 <http://yago-knowledge.org/resource/extractionSource> ?d
}
```

Query 2 (SQL):

```
        select
        d1.stringvalue,
        d2.stringvalue,
        d3.stringvalue,
        d4.stringvalue,
        d5.stringvalue
        from
        orpsnumericyago2 n1,
        orpsnumericyago2 n2,
        orpsnumericyago2 n3,
        orpsnumericyago2 n4,
        orpsnumericyago2  n5,
        dictionaryFull d1,
        dictionaryFull d2,
        dictionaryFull d3,
        dictionaryFull d4,
        dictionaryFull d5
```

```
where
n1.predicate=(select  id from  dictionaryFull where stringvalue=
   '"http://yago-knowledge.org/resource/holdsPoliticalPosition"
   ') and
n1.object=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/Governor_of_New_York"')
   and
n1.reification=n2.subject and
n2.predicate=( select id from dictionaryFull where stringvalue='
   "http://yago-knowledge.org/resource/hasPredecessor"') and
n1.reification=n3.subject and
n3.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/hasSuccessor"') and
n3.object=n4.subject and
n4.predicate= (select id from dictionaryFull  where stringvalue=
   '"http://yago-knowledge.org/resource/hasWonPrize"') and
n4.reification=n5.subject and
n5.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/extractionSource"') and
n1.subject=d1.id and
n2.object=d2.id and
n3.object=d3.id and
n4.object=d4.id and
n5.object=d5.id;
```

Query 3 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?c?d
WHERE {
  ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://yago-
     knowledge.org/resource/wikicategory_U.S.
     _Presidents_surviving_assassination_attempts>.
  ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://yago-
     knowledge.org/resource/
```

```
     wikicategory_Politicians_with_physical_disabilities>.
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/hasWonPrize>.
  ?r rdf:object ?b. ?r <http://yago-knowledge.org/resource/occursSince>
     ?c.
  ?r <http://yago-knowledge.org/resource/extractionSource> ?d
}
```

Query 3 (SQL):

```
        select
        d1.stringvalue,
        d3.stringvalue,
        d4.stringvalue,
        d5.stringvalue
        from
        orpsnumericyago2 n1,
        orpsnumericyago2 n2,
        orpsnumericyago2 n3,
        orpsnumericyago2 n4,
        orpsnumericyago2 n5,
        dictionaryFull d1,
        dictionaryFull d3,
        dictionaryFull d4,
        dictionaryFull d5
        where
        n1.predicate=(select id from dictionaryFull where stringvalue='"
           http://www.w3.org/1999/02/22-rdf-syntax-ns#type"') and
        n1.object=(select id from dictionaryFull where stringvalue='"
           http://yago-knowledge.org/resource/wikicategory_U.S.
           _Presidents_surviving_assassination_attempts"') and
        n1.subject=n2.subject and
        n2.predicate=(select id from dictionaryFull where stringvalue='"
           http://www.w3.org/1999/02/22-rdf-syntax-ns#type"') and
        n2.object=(select id from dictionaryFull where stringvalue='"
           http://yago-knowledge.org/resource/
           wikicategory_Politicians_with_physical_disabilities"') and
```

```
n1.subject=n3.subject and
n3.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/hasWonPrize"') and
n3.reification=n4.subject and
n4.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/occursSince"') and
n3.reification=n5.subject and
n5.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/extractionSource"') and
n1.subject=d1.id and
n3.object=d3.id and
n4.object=d4.id and
n5.object=d5.id;
```

Query 4 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?a1?c
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/playsFor>.
  ?r rdf:object <http://yago-knowledge.org/resource/Spain_women'
     s_national_football_team>.
  ?r <http://yago-knowledge.org/resource/occursSince> ?c.
  ?r1 rdf:subject ?a1.
  ?r1 rdf:predicate <http://yago-knowledge.org/resource/playsFor>.
  ?r1 rdf:object <http://yago-knowledge.org/resource/Spain_women'
     s_national_football_team>.
  ?r1 <http://yago-knowledge.org/resource/occursSince> ?c
}
```

Query 4 (SQL):

```
select
d1.stringvalue,
d3.stringvalue,
d2.stringvalue
```

```
from
orpsnumericyago2 n1,
orpsnumericyago2 n2,
orpsnumericyago2 n3,
orpsnumericyago2 n4,
dictionaryFull d1,
dictionaryFull d3,
dictionaryFull d2
where
n1.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/playsFor"') and
n1.object=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/Spain_women''
   s_national_football_team"') and
n2.subject=n1.reification and
n2.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/occursSince"') and
n3.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/playsFor"') and
n3.object=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/Spain_women''
   s_national_football_team"') and
n4.subject=n3.reification and
n4.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/occursSince"') and
n2.object=n4.object and
n1.subject=d1.id and
n3.subject=d3.id and
n2.object=d2.id;
```

Query 5 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?c?d
WHERE {
  ?r rdf:subject ?a.
```

```
  ?r rdf:predicate <http://yago-knowledge.org/resource/isConnectedTo>.
  ?r rdf:object <http://yago-knowledge.org/resource/
     Kansas_City_International_Airport>.
  ?r <http://yago-knowledge.org/resource/byTransport> ?c.
  ?a <http://yago-knowledge.org/resource/linksTo> ?b.
  ?r1 rdf:subject ?b.
  ?r1 rdf:predicate <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>.
  ?r1 rdf:object <http://yago-knowledge.org/resource/
     wikicategory_Terrorist_attacks_on_airports>.
  ?r1 <http://yago-knowledge.org/resource/extractionSource> ?d
}
```

Query 5 (SQL):

```
select
d1.stringvalue,
d2.stringvalue,
d5.stringvalue
from
orpsnumericyago2 n1,
orpsnumericyago2 n2,
orpsnumericyago2 n3,
orpsnumericyago2 n4,
orpsnumericyago2 n5,
dictionaryFull d1,
dictionaryFull d2,
dictionaryFull d5
where
n1.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/isConnectedTo"') and
n1.object=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/
   Kansas_City_International_Airport"') and
n1.reification=n2.subject and
n2.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/byTransport"') and
n1.subject=n3.subject and
```

```
        n3.predicate=(select id from dictionaryFull where stringvalue='"
           http://yago-knowledge.org/resource/linksTo"') and
        n3.object=n4.subject and
        n4.predicate=(select id from dictionaryFull where stringvalue='"
           http://www.w3.org/1999/02/22-rdf-syntax-ns#type"') and
        n4.object=(select id from dictionaryFull where stringvalue='"
           http://yago-knowledge.org/resource/
           wikicategory_Terrorist_attacks_on_airports"') and
        n4.reification=n5.subject and
        n5.predicate=(select id from dictionaryFull where stringvalue='"
           http://yago-knowledge.org/resource/extractionSource"') and
        n1.subject=d1.id and
        n2.object=d2.id and
        n5.object=d5.id;
```

Query 6 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?d
WHERE {
  ?a <http://yago-knowledge.org/resource/hasGender> <http://yago-
     knowledge.org/resource/male>.
  ?a <http://yago-knowledge.org/resource/playsFor> <http://yago-
     knowledge.org/resource/Birmingham_Excelsior_F.C.>.
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/wasBornOnDate>.
  ?r rdf:object ?c.
  ?r <http://yago-knowledge.org/resource/happenedIn> ?e.
  ?e <http://yago-knowledge.org/resource/hasMotto> "Forward".
  ?r <http://yago-knowledge.org/resource/extractionSource> ?d
}
```

Query 6 (SQL):

```
        select
        d1.stringvalue,
        d6.stringvalue
```

```
from
dictionaryFull d5,
orpsnumericyago2 n1,
orpsnumericyago2 n2,
orpsnumericyago2 n3,
orpsnumericyago2 n4,
orpsnumericyago2 n5,
orpsnumericyago2 n6,
dictionaryFull d1,
dictionaryFull d6
where
n1.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/hasGender"') and
n1.object=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/male"') and
n1.subject=n2.subject and
n2.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/playsFor"') and
n2.object=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/Birmingham_Excelsior_F.C.
   "') and
n1.subject=n3.subject and
n3.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/wasBornOnDate"') and
n3.reification=n4.subject and
n4.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/happenedIn"') and
n4.object=n5.subject and
n5.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/hasMotto"') and
n5.object=d5.id and
d5.stringvalue='"Forward"' and
n3.reification=n6.subject and
n6.predicate=(select id from dictionaryFull where stringvalue='"
   http://yago-knowledge.org/resource/extractionSource"') and
n1.subject=d1.id and
```

```
        n6.object=d6.id;
```

Query 7 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?b?c
WHERE {
  ?r rdf:subject <http://yago-knowledge.org/resource/Al-Qaeda>.
  ?r rdf:predicate <http://yago-knowledge.org/resource/participatedIn>.
  ?r rdf:object ?b.
  ?r <http://yago-knowledge.org/resource/extractionSource> ?c.
  ?b <http://yago-knowledge.org/resource/happenedIn> ?d.
  ?d <http://yago-knowledge.org/resource/isLocatedIn> ?e.
  ?e <http://yago-knowledge.org/resource/isLocatedIn> <http://yago-
     knowledge.org/resource/Asia>
}
```

Query 7 (SQL):

```
        select
        d1.stringvalue,
        d2.stringvalue
        from
        orpsnumericyago2 n1,
        orpsnumericyago2 n2,
        orpsnumericyago2 n3,
        orpsnumericyago2 n4,
        orpsnumericyago2 n5,
        dictionaryFull d1,
        dictionaryFull d2
        where
        n1.subject=(select id from dictionaryFull where stringvalue='"
           http://yago-knowledge.org/resource/Al-Qaeda"') and
        n1.predicate=(select id from dictionaryFull where stringvalue='"
           http://yago-knowledge.org/resource/participatedIn"') and
        n1.reification=n2.subject and
        n2.predicate=(select id from dictionaryFull where stringvalue='"
```

```
    http://yago-knowledge.org/resource/extractionSource"') and
  n1.object=n3.subject and
  n3.predicate=(select id from dictionaryFull where stringvalue='"
    http://yago-knowledge.org/resource/happenedIn"') and
  n3.object=n4.subject and n4.predicate=(select id from
    dictionaryFull where stringvalue='"http://yago-knowledge.org
    /resource/isLocatedIn"') and
  n4.object=n5.subject and
  n5.predicate=(select id from dictionaryFull where stringvalue='"
    http://yago-knowledge.org/resource/isLocatedIn"') and
  n5.object=(select id from dictionaryFull where stringvalue='"
    http://yago-knowledge.org/resource/Asia"') and
  n1.object=d1.id and
  n2.object=d2.id;
```

A.1.2 DBpedia

Query 1 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?d
WHERE {
  ?r rdf:subject <http://dbpedia.org/resource/Family_of_Barack_Obama>.
  ?r rdf:predicate <http://dbpedia.org/ontology/wikiPageExternalLink>.
  ?r rdf:object ?a. ?r <context> ?c.
  ?r1 rdf:subject <http://dbpedia.org/resource/Family_of_Barack_Obama>.
  ?r1 rdf:predicate <http://dbpedia.org/property/members>.
  ?r1 rdf:object ?b.
  ?r1 <context> ?c.
  ?r2 rdf:subject <http://dbpedia.org/resource/Family_of_Barack_Obama>.
  ?r2 rdf:predicate <http://xmlns.com/foaf/0.1/depiction>.
  ?r2 rdf:object ?d.
  ?r2 <context> ?c
}
```

Query 1 (SQL):

```
  select
```

```
dict1.stringVal,
dict3.stringVal,
dict5.stringVal
from
dbpedianumeric d1,
dbpedianumeric d2,
dbpedianumeric d3,
dbpedianumeric d4,
dbpedianumeric d5,
dbpedianumeric d6,
dictionary dict1,
dictionary dict3,
dictionary dict5
where
d1.subject=(select id from dictionary where md5(stringVal)=md5('
   "http://dbpedia.org/resource/Family_of_Barack_Obama"')) and
d1.predicate=(select id from dictionary where md5(stringVal)=md5
   ('"http://dbpedia.org/ontology/wikiPageExternalLink"')) and
d1.reification=d2.subject and
d2.predicate=(select id from dictionary where md5(stringVal)=md5
   ('"context"')) and d2.object=d4.object and
d3.subject=(select id from dictionary where md5(stringVal)=md5('
   "http://dbpedia.org/resource/Family_of_Barack_Obama"')) and
d3.predicate=(select id from dictionary where md5(stringVal)=md5
   ('"http://dbpedia.org/property/members"')) and
d3.reification=d4.subject and
d4.predicate=(select id from dictionary where md5(stringVal)=md5
   ('"context"')) and
d1.object=dict1.id and
d3.object=dict3.id and
d5.subject=(select id from dictionary where md5(stringVal)=md5('
   "http://dbpedia.org/resource/Family_of_Barack_Obama"')) and
d5.predicate=(select id from dictionary where md5(stringVal)=md5
   ('"http://xmlns.com/foaf/0.1/depiction"')) and
d5.reification=d6.subject and
d6.predicate=(select id from dictionary where md5(stringVal)=md5
```

```
        ('"context"')) and d6.object=d2.object and
    dict5.id=d5.object;
```

Query 2 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?c
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate ?b.
  ?r rdf:object <http://dbpedia.org/resource/Category:
     Buildings_and_structures_in_Massac_County,_Illinois>.
  ?r <context> ?c
}
```

Query 2 (SQL):

```
select dict1.stringVal, dict2.stringVal, dict3.stringVal from
   dbpedianumeric d1, dbpedianumeric d2, dictionary dict1, dictionary
   dict2, dictionary dict3 where d1.object=(select id from dictionary
   where md5(stringVal)=md5('"http://dbpedia.org/resource/Category:
   Buildings_and_structures_in_Massac_County,_Illinois"')) and d1.
   reification=d2.subject and d2.predicate=(select id from dictionary
   where md5(stringVal)=md5('"context"')) and d1.subject=dict1.id and
   d1.predicate=dict2.id and d2.object=dict3.id;
```

Query 3 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?c
WHERE {
  ?r rdf:subject <http://dbpedia.org/resource/George_Lyle_Ashe>.
  ?r rdf:predicate <http://dbpedia.org/property/religion>.
  ?r rdf:object ?b.
  ?r <context> ?a.
  ?r1 rdf:subject <http://dbpedia.org/resource/George_Lyle_Ashe>.
  ?r1 rdf:predicate <http://dbpedia.org/property/placeOfBirth>.
```

```
  ?r1 rdf:object ?c.
  ?r1 <context> ?a
}
```

Query 3 (SQL):

```
        select
        dict2.stringVal,
        dict1.stringVal,
        dict3.stringVal
        from
        dbpedianumeric d1,
        dbpedianumeric d2,
        dbpedianumeric d3,
        dbpedianumeric d4,
        dictionary dict1,
        dictionary dict2,
        dictionary dict3
        where
        d1.subject=(select id from dictionary where md5(stringVal)=md5('
           "http://dbpedia.org/resource/George_Lyle_Ashe"')) and
        d1.predicate=(select id from dictionary where md5(stringVal)=md5
           ('"http://dbpedia.org/property/religion"')) and
        d1.reification=d2.subject and
        d2.predicate=(select id from dictionary where md5(stringVal)=md5
           ('"context"')) and
        d3.subject=(select id from dictionary where md5(stringVal)=md5('
           "http://dbpedia.org/resource/George_Lyle_Ashe"')) and
        d3.predicate=(select id from dictionary where md5(stringVal)=md5
           ('"http://dbpedia.org/property/placeOfBirth"')) and
        d3.reification=d4.subject and
        d4.predicate=(select id from dictionary where md5(stringVal)=md5
           ('"context"')) and
        d2.object=d4.object and
        d2.object=dict2.id and
        d1.object=dict1.id and
        d3.object=dict3.id;
```

Query 4 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?c
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate ?b.
  ?r rdf:object ?c.
  ?r <context> <http://dbpedia.org/data/Plasmodium_hegneri.xml>
}
```

Query 4 (SQL):

```
        select
        dict1.stringVal,
        dict2.stringVal,
        dict3.stringVal
        from
        dbpedianumeric d1,
        dbpedianumeric d2,
        dictionary dict1,
        dictionary dict2,
        dictionary dict3
        where
        d1.reification=d2.subject and
        d2.predicate=(select id from dictionary where md5(stringVal)=md5
           ('"context"')) and
        d2.object=(select id from dictionary where md5(stringVal)=md5('"
           http://dbpedia.org/data/Plasmodium_hegneri.xml"')) and
        d1.subject=dict1.id and
        d1.predicate=dict2.id and
        d1.object=dict3.id;
```

Query 5 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
SELECT ?a?b
WHERE {
  ?r rdf:subject <http://dbpedia.org/resource/United_States_Navy_SEALs>.
  ?r rdf:predicate <http://dbpedia.org/property/battles>.
  ?r rdf:object ?a.
  ?r <context> <http://dbpedia.org/data/United_States_Navy_SEALs.xml>.
  ?r1 rdf:subject ?a.
  ?r1 rdf:predicate <http://dbpedia.org/property/battles>.
  ?r1 rdf:object ?b.
  ?r1 <context> <http://dbpedia.org/data/War_on_Terror.xml>
}
```

Query 5 (SQL):

```
select
dict1.stringVal,
dict3.stringVal
from
dbpedianumeric d1,
dbpedianumeric d2,
dbpedianumeric d3,
dbpedianumeric d4,
dictionary dict1,
dictionary dict3
where
d1.subject=(select id from dictionary where md5(stringVal)=md5('
   "http://dbpedia.org/resource/United_States_Navy_SEALs"'))
   and
d1.predicate=(select id from dictionary where md5(stringVal)=md5
   ('"http://dbpedia.org/property/battles"')) and
d1.object=d3.subject and
d1.reification=d2.subject and
d2.predicate=(select id from dictionary where md5(stringVal)=md5
   ('"context"')) and
d2.object=(select id from dictionary where md5(stringVal)=md5('"
   http://dbpedia.org/data/United_States_Navy_SEALs.xml"')) and
d3.predicate=(select id from dictionary where md5(stringVal)=md5
```

```
    ('"http://dbpedia.org/property/battles"')) and
d3.reification=d4.subject and
d4.predicate=(select id from dictionary where md5(stringVal)=md5
    ('"context"')) and
d4.object=(select id from dictionary where md5(stringVal)=md5('"
    http://dbpedia.org/data/War_on_Terror.xml"')) and
d1.object=dict1.id and
d3.object=dict3.id;
```

Query 6 (SPARQL):

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?c
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://dbpedia.org/property/knownFor>.
  ?r rdf:object ?b.
  ?r <context> <http://dbpedia.org/data/Marienetta_Jirkowsky.xml>.
  ?r1 rdf:subject ?a.
  ?r1 rdf:predicate <http://dbpedia.org/ontology/knownFor>.
  ?r1 rdf:object ?c.
  ?r1 <context> <http://dbpedia.org/data/Berlin_Wall.xml>
}
```

Query 6 (SQL):

```
select
dict1.stringVal,
dict2.stringVal,
dict3.stringVal
from
dbpedianumeric d1,
dbpedianumeric d2,
dbpedianumeric d3,
dbpedianumeric d4,
dictionary dict1,
dictionary dict2,
```

```
dictionary dict3
where
d1.predicate=(select id from dictionary where md5(stringVal)=md5
    ('"http://dbpedia.org/property/knownFor"')) and
d1.reification=d2.subject and
d2.predicate=(select id from dictionary where md5(stringVal)=md5
    ('"context"')) and
d2.object=(select id from dictionary where md5(stringVal)=md5('"
    http://dbpedia.org/data/Marienetta_Jirkowsky.xml"')) and
d1.subject=d3.subject and
d3.predicate=(select id from dictionary where md5(stringVal)=md5
    ('"http://dbpedia.org/ontology/knownFor"')) and
d3.reification=d4.subject and
d4.predicate=(select id from dictionary where md5(stringVal)=md5
    ('"context"')) and
d4.object=(select id from dictionary where md5(stringVal)=md5('"
    http://dbpedia.org/data/Berlin_Wall.xml"')) and
d1.subject=dict1.id and
d1.object=dict2.id and
d3.object=dict3.id;
```

A.2 Quark-X Benchmark Queries

A.2.1 YAGO

Query 1:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place?location
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate <http://yago-knowledge.org/resource/isLocatedIn>.
  ?r rdf:object ?location.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasNumberOfPeople> ?
     numberOfPeople.
```

```
  ?place <http://yago-knowledge.org/resource/hasPopulationDensity> ?
     populationDensity
} ORDER BY ASC((5+?numberOfPeople +?populationDensity) * (1+?conf))
   LIMIT 50
```

Query 2:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place?location
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate <http://yago-knowledge.org/resource/isLocatedIn>.
  ?r rdf:object ?location.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?Conf.
  ?place <http://yago-knowledge.org/resource/hasNumberOfPeople> ?
     numberOfPeople.
  ?place <http://yago-knowledge.org/resource/hasPopulationDensity> ?
     populationDensity.
  ?place <http://yago-knowledge.org/resource/hasArea> area
} ORDER BY ASC((389000006+?numberOfPeople+?populationDensity +?area) *
   (1+?Conf)) LIMIT 50
```

Query 3:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?owns
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/owns>.
  ?r rdf:object ?owns.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?owns <http://yago-knowledge.org/resource/hasHeight> ?height.
  ?a <http://yago-knowledge.org/resource/hasArea> ?area
} ORDER BY ASC((389000197+?height+?area) * (1+?conf)) LIMIT 50
```

Query 4:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?owns
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/owns>.
  ?r rdf:object ?owns.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?owns <http://yago-knowledge.org/resource/hasHeight> ?height.
  ?a <http://yago-knowledge.org/resource/hasNumberOfPeople> ?
     numberOfPeople
} ORDER BY ASC((197+?height+?numberOfPeople) * (1+?conf)) LIMIT 50
```

Query 5:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?b?c?location
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/owns>.
  ?r rdf:object ?b.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?b <http://yago-knowledge.org/resource/isConnectedTo> ?c.
  ?c <http://yago-knowledge.org/resource/isLocatedIn> ?location.
  ?location <http://yago-knowledge.org/resource/hasExpenses> ?elevation
} ORDER BY ASC(?elevation * (1+?conf)) LIMIT 50
```

Query 6:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?event?place
WHERE {
  ?r rdf:subject ?event.
  ?r rdf:predicate <http://yago-knowledge.org/resource/happenedIn>.
  ?r rdf:object ?place.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
```

```
  ?place <http://yago-knowledge.org/resource/hasInflation> ?inflation
} ORDER BY ASC((1+?inflation) * (1+?conf)) LIMIT 50
```

Query 7:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person?book
WHERE {
  ?r rdf:subject ?person.
  ?r rdf:predicate <http://yago-knowledge.org/resource/created>.
  ?r rdf:object ?book.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?book <http://yago-knowledge.org/resource/hasPages> ?pages
} ORDER BY ASC((1+?pages) * (1+?conf)) LIMIT 50
```

Query 8:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person?book?personInfluenced
WHERE {
  ?r rdf:subject ?person.
  ?r rdf:predicate <http://yago-knowledge.org/resource/created>.
  ?r rdf:object ?book. ?r <http://yago-knowledge.org/resource/
     hasConfidence> ?conf.
  ?person <http://yago-knowledge.org/resource/hasGender> <http://yago-
     knowledge.org/resource/male>.
  ?person <http://yago-knowledge.org/resource/influences> ?
     personInfluenced.
  ?book <http://yago-knowledge.org/resource/hasPages> ?pages
} ORDER BY ASC((1+?pages) * (1+?conf)) LIMIT 50
```

Query 9:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person?book?personInfluenced
WHERE {
```

```
  ?r rdf:subject ?person.
  ?r rdf:predicate <http://yago-knowledge.org/resource/created>.
  ?r rdf:object ?book.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?person <http://yago-knowledge.org/resource/hasGender> <http://yago-
     knowledge.org/resource/male>.
  ?person <http://yago-knowledge.org/resource/influences> ?
     personInfluenced.
  ?book <http://yago-knowledge.org/resource/hasPages> ?pages.
  ?person <http://yago-knowledge.org/resource/isMarriedTo> ?f
} ORDER BY ASC((1+?pages) * (1+?conf)) LIMIT 50
```

Query 10:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person?b
WHERE {
  ?r rdf:subject ?person.
  ?r rdf:predicate <http://yago-knowledge.org/resource/created>.
  ?r rdf:object ?b.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?b <http://yago-knowledge.org/resource/hasDuration> ?duration
} ORDER BY ASC((21601+?duration) * (1+?conf)) LIMIT 50
```

Query 11:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person?residesIn?c
WHERE {
  ?r rdf:subject ?person.
  ?r rdf:predicate ?residesIn.
  ?r rdf:object <http://yago-knowledge.org/resource/Mumbai>.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?r1 rdf:subject ?person.
  ?r1 rdf:predicate ?c.
  ?r1 rdf:object <http://yago-knowledge.org/resource/male>.
```

```
  ?r1 <http://yago-knowledge.org/resource/hasConfidence> ?conf1
} ORDER BY ASC((1+?conf) * (1+?conf1)) LIMIT 50
```

A.2.2 DBpedia

Query 1:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person ?birthPlace ?country
WHERE {
  ?person <http://dbpedia.org/ontology/birthPlace> ?birthPlace.
  ?person <http://dbpedia.org/ontology/height> ?height.
  ?person <http://dbpedia.org/property/weightLb> ?weight.
  ?r rdf:subject ?birthPlace.
  ?r rdf:predicate <http://dbpedia.org/ontology/country>.
  ?r rdf:object ?country.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf
} ORDER BY ASC((?height+?weight+2) * (1+?conf)) LIMIT 50
```

Query 2:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a ?country
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://dbpedia.org/ontology/country>.
  ?r rdf:object ?country.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?a <http://dbpedia.org/ontology/elevation> ?elevation.
  ?a <http://dbpedia.org/ontology/area> ?area
} ORDER BY ASC((?elevation+?area+1151) * (1+?conf)) LIMIT 50
```

Query 3:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?product ?vehicle ?bodyStyle ?depiction
WHERE {
```

```
  ?product <http://dbpedia.org/property/related> ?vehicle.
  ?vehicle <http://dbpedia.org/ontology/bodyStyle> ?bodyStyle.
  ?vehicle <http://dbpedia.org/ontology/fuelCapacity> ?fuelCapacity.
  ?r rdf:subject ?bodyStyle.
  ?r rdf:predicate <http://xmlns.com/foaf/0.1/depiction>.
  ?r rdf:object ?depiction.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?Conf.
  ?vehicle <http://dbpedia.org/ontology/height> ?height
} ORDER BY ASC((?fuelCapacity+?height+2) * (1+?Conf)) LIMIT 50
```

Query 4:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?airport ?city
WHERE {
  ?airport <http://dbpedia.org/ontology/city> ?city.
  ?airport <http://dbpedia.org/ontology/elevation> ?elevation.
  ?airport <http://dbpedia.org/ontology/runwayLength> ?runwayLength.
  ?airport <http://dbpedia.org/property/width> ?width
} ORDER BY ASC((?elevation+?runwayLength+?width+1152)) LIMIT 50
```

Query 5:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?spaceVehicle ?discoverer
WHERE {
  ?spaceVehicle <http://dbpedia.org/ontology/orbitalPeriod> ?
     orbitalPeriod.
  ?spaceVehicle <http://dbpedia.org/ontology/apoapsis> ?apoapsis.
  ?spaceVehicle <http://dbpedia.org/ontology/periapsis> ?periapsis.
  ?spaceVehicle <http://dbpedia.org/ontology/absoluteMagnitude> ?
     absoluteMagnitude.
  ?spaceVehicle <http://dbpedia.org/ontology/averageSpeed> ?averageSpeed
     .
  ?spaceVehicle <http://dbpedia.org/ontology/mass> ?mass.
  ?r rdf:subject ?spaceVehicle.
```

```
  ?r rdf:predicate <http://dbpedia.org/ontology/discoverer>.
  ?r rdf:object ?discoverer.
  ?spaceVehicle <http://dbpedia.org/ontology/escapeVelocity> ?
     escapeVelocity.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf
} ORDER BY ASC((?orbitalPeriod + ?apoapsis + ?periapsis + ?
   absoluteMagnitude + ?averageSpeed + ?mass +  ?escapeVelocity) * ?
   conf) LIMIT 50
```

Query 6:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?city ?country ?h
WHERE {
  ?place <http://dbpedia.org/property/populationTotal> ?populationTotal.
  ?place <http://dbpedia.org/property/city> ?city.
  ?city <http://dbpedia.org/ontology/isPartOf> ?country.
  ?r rdf:subject ?country.
  ?r rdf:predicate <http://dbpedia.org/ontology/country>.
  ?r rdf:object ?h.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf
} ORDER BY ASC((?populationTotal+1001) * (1+?conf)) LIMIT 50
```

Query 7:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?movie ?producer ?birthPlace
WHERE {
  ?r rdf:subject ?movie.
  ?r rdf:predicate <http://dbpedia.org/property/producer>.
  ?r rdf:object ?producer.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?r1 rdf:subject ?producer.
  ?r1 rdf:predicate <http://dbpedia.org/ontology/birthPlace>.
  ?r1 rdf:object ?birthPlace.
  ?r1 <http://yago-knowledge.org/resource/hasConfidence> ?conf1
```

```
} ORDER BY ASC((1+?conf) * (1+?conf1)) LIMIT 50
```

Query 8:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?movie ?director ?starring
WHERE {
  ?movie <http://dbpedia.org/ontology/director> ?director.
  ?movie <http://dbpedia.org/ontology/starring> ?starring.
  ?movie <http://dbpedia.org/ontology/runtime> ?runtime
} ORDER BY ASC(?runtime) LIMIT 50
```

Query 9:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a ?d ?nationality
WHERE {
  ?a <http://dbpedia.org/property/losses> ?losses.
  ?a <http://dbpedia.org/property/wins> ?wins.
  ?a <http://dbpedia.org/property/after> ?d.
  ?d <http://dbpedia.org/ontology/nationality> ?nationality
} ORDER BY ASC(?losses+?wins+2) LIMIT 50
```

Query 10:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a ?d ?n
WHERE {
  ?a <http://dbpedia.org/property/losses> ?losses.
  ?a <http://dbpedia.org/property/wins> ?wins.
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://dbpedia.org/property/after>.
  ?r rdf:object ?d.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?d <http://dbpedia.org/ontology/nationality> ?n
} ORDER BY ASC((?losses+?wins+2) * (1+?conf)) LIMIT 50
```

Query 11:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?televisionShow ?b ?c
WHERE {
  ?televisionShow ?b <http://dbpedia.org/ontology/TelevisionShow>.
  ?televisionShow ?c <http://dbpedia.org/resource/Color>.
  ?televisionShow <http://dbpedia.org/ontology/runtime> ?runtime
} ORDER BY ASC(?runtime) LIMIT 50
```

A.3 STREAK Benchmark Queries

A.3.1 YAGO

Query 1:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?loc1 ?loc2
WHERE {
  ?place <http://yago-knowledge.org/resource/hasPopulationDensity> ?
     popul.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?place <http://yago-knowledge.org/resource/isLocatedIn> ?loc1.
  ?nplace <http://yago-knowledge.org/resource/isLocatedIn> ?loc2.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://yago-knowledge.org/resource/hasNumberOfPeople> ?popul1
     .
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?popul + ?popul1) LIMIT k
```

Query 2:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?loc1 ?loc2
WHERE {
```

```
  ?place <http://yago-knowledge.org/resource/hasPopulationDensity> ?
     popul.
  ?place <http://yago-knowledge.org/resource/hasEconomicGrowth> ?
     ecoGrowth.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?place <http://yago-knowledge.org/resource/isLocatedIn> ?loc1.
  ?nplace <http://yago-knowledge.org/resource/isLocatedIn> ?loc2.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://yago-knowledge.org/resource/hasNumberOfPeople> ?popul1
     .
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?popul + ?popul1 + ?ecoGrowth) LIMIT k
```

Query 3:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?loc1 ?loc2 ?connection
WHERE {
  ?place <http://yago-knowledge.org/resource/hasEconomicGrowth> ?
     ecoGrowth.
  ?connection <http://yago-knowledge.org/resource/isConnectedTo> ?place.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?place <http://yago-knowledge.org/resource/isLocatedIn> ?loc1.
  ?nplace <http://yago-knowledge.org/resource/isLocatedIn> ?loc2.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://yago-knowledge.org/resource/hasNumberOfPeople> ?popul1
     .
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?popul1 + ?ecoGrowth) LIMIT k
```

Query 4:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?loc1 ?loc2 ?birthPlace
WHERE {
  ?place <http://yago-knowledge.org/resource/hasPopulationDensity> ?
```

```
    popul.
  ?place <http://yago-knowledge.org/resource/hasEconomicGrowth> ?
    ecoGrowth.
  ?place <http://yago-knowledge.org/resource/hasNeighbor> ?birthPlace.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?place <http://yago-knowledge.org/resource/isLocatedIn> ?loc1.
  ?nplace <http://yago-knowledge.org/resource/isLocatedIn> ?loc2.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://yago-knowledge.org/resource/hasNumberOfPeople> ?popul1
    .
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?popul + ?popul1 + ?ecoGrowth) LIMIT k
```

Query 5:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?d?location?nplace?a?b?person
WHERE {
  ?r rdf:subject ?b.
  ?r rdf:predicate <http://yago-knowledge.org/resource/diedIn>.
  ?r rdf:object ?a.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?a <http://yago-knowledge.org/resource/isLocatedIn> ?d.
  ?d <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?r1 rdf:subject ?person.
  ?r1 rdf:predicate <http://yago-knowledge.org/resource/wasBornIn>.
  ?r1 rdf:object ?nplace.
  ?nplace <http://yago-knowledge.org/resource/isLocatedIn> ?location.
  ?r1 <http://yago-knowledge.org/resource/hasConfidence> ?popul1.
  ?location <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?popul1 + ?conf) LIMIT k
```

Query 6:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
SELECT ?b?nplace?a?loc2
WHERE {
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/happenedIn>.
  ?r rdf:object ?b.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?b <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?b <http://yago-knowledge.org/resource/hasInflation> ?d.
  ?nplace <http://yago-knowledge.org/resource/isLocatedIn> ?loc2.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://yago-knowledge.org/resource/hasNumberOfPeople> ?popul1
     .
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?d + ?popul1 + ?conf) LIMIT k
```

Query 7:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?a?nplace?b?loc2
WHERE {
  ?nplace <http://yago-knowledge.org/resource/isLocatedIn> ?loc2.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://yago-knowledge.org/resource/hasEconomicGrowth> ?popul1
     .
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/isLocatedIn>.
  ?r rdf:object ?b.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?a <http://yago-knowledge.org/resource/hasGeometry> ?long.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?popul1 + ?conf) LIMIT k
```

Query 8:

```
SELECT ?b?nplace?a?loc2
WHERE {
  ?r1 rdf:subject ?loc2.
```

```
  ?r1 rdf:predicate <http://yago-knowledge.org/resource/wasBornIn>.
  ?r1 rdf:object ?nplace.
  ?r1 <http://yago-knowledge.org/resource/hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?r rdf:subject ?a.
  ?r rdf:predicate <http://yago-knowledge.org/resource/isLocatedIn>.
  ?r rdf:object ?b.
  ?r <http://yago-knowledge.org/resource/hasConfidence> ?conf.
  ?b <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?a <http://yago-knowledge.org/resource/hasPopulationDensity> ?d.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?d + ?conf1 + ?conf) LIMIT k
```

A.3.2 LGD

Query 1:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_points#hotel>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_natural#park>.
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?conf + ?conf1) LIMIT k
```

Query 2:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2
```

```
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_points#hotel>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_points#police>.
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?conf + ?conf1) LIMIT k
```

Query 3:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2 ?label1 ?name2
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_points#hotel>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?place <http://www.w3.org/2000/01/rdf-schema#label> ?label1.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_points#police>.
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://geoknow.eu/uk_points#name> ?name2.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?conf + ?conf1) LIMIT k
```

Query 4:

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

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2 ?label1 ?name1 ?name2
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_points#pub>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?place <http://www.w3.org/2000/01/rdf-schema#label> ?label1.
  ?place <http://geoknow.eu/uk_points#name> ?name1.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_points#police>.
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://geoknow.eu/uk_points#name> ?name2.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?conf + ?conf1) LIMIT k
```

Query 5:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2 ?label1 ?name1 ?name2 ?
   label2
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_natural#park>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?place <http://www.w3.org/2000/01/rdf-schema#label> ?label1.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_points#police>.
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
```

```
  ?nplace <http://geoknow.eu/uk_points#name> ?name2.
  ?nplace <http://www.w3.org/2000/01/rdf-schema#label> ?label2.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?conf + ?conf1) LIMIT k
```

Query 6:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_natural#park>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_roads#roads>.
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?conf + ?conf1) LIMIT k
```

Query 7:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_points#hotel>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_roads#roads>.
```

```
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC((++0+1*?conf/1) (++0+1*?conf1/1)) LIMIT k
```

Query 8:

```
BASE <http://yago-knowledge.org/resource/> .
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?place ?nplace ?typePred1 ?typePred2 ?label2
WHERE {
  ?r rdf:subject ?place.
  ?r rdf:predicate ?typePred1.
  ?r rdf:object <http://geoknow.eu/uk_natural#park>.
  ?r <hasConfidence> ?conf.
  ?place <http://yago-knowledge.org/resource/hasGeometry> ?long.
  ?r1 rdf:subject ?nplace.
  ?r1 rdf:predicate ?typePred2.
  ?r1 rdf:object <http://geoknow.eu/uk_roads#roads>.
  ?r1 <hasConfidence> ?conf1.
  ?nplace <http://yago-knowledge.org/resource/hasGeometry> ?nlong.
  ?nplace <http://www.w3.org/2000/01/rdf-schema#label> ?label2.
  FILTER((?long, ?nlong) < 50)
} ORDER BY ASC(?conf + ?conf1) LIMIT k
```

Bibliography

[1] RDF 1.1 Primer. Technical report. https://www.w3.org/TR/rdf11-primer/.

[2] RDF 1.1 N-Quads: A line-based syntax for RDF datasets. W3C Recommendation, February 2014. http://www.w3.org/TR/n-quads/.

[3] Uniprot: a hub for protein information. *Nucleic Acids Research*, 43(Database-Issue):204–212, 2015.

[4] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable Semantic Web Data Management using vertical partitioning. In *PVLDB*, 2007.

[5] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *The VLDB Journal*, 18(2), 2009.

[6] Nicole Alexander and Siva Ravada. RDF object type and reification in the database. In *ICDE*, 2006.

[7] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*. 2014.

[8] Güneş Aluç, M Tamer Özsu, Khuzaima Daudjee, and Olaf Hartig. Executing queries over schemaless RDF databases. In *ICDE*, 2015.

[9] Vo Ngoc Anh and Alistair Moffat. Simplified similarity scoring using term ranks. In *SIGIR*, 2005.

[10] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR*, 2006.

[11] Jens Auer, Sören Lehmann and Sebastian Hellmann. LinkedGeoData: Adding a spatial dimension to the web of data. *ISWC*, 2009.

[12] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. *ISWC*, 2007.

[13] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[14] Robert Battle and Dave Kolas. Enabling the geospatial semantic web with parliament and GeoSPARQL. *Semantic Web*, 3(4), 2012.

[15] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 Turtle Terse RDF Triple Language. W3C recommendation. http://www.w3.org/TR/turtle/.

[16] Joanna Biega, Erdal Kuzey, and Fabian M Suchanek. Inside YAGO2s: A transparent information extraction architecture. In *WWW*, 2013.

[17] Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *PVLDB*, 2005.

[18] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[19] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A Crystallization Point for the Web of Data. *J. Web Sem.*, 7(3), 2009.

[20] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *Int. Journal of Semantic Web Inf. Syst.*, 5(2), 2009.

[21] Olivier Bodenreider. Provenance information in biomedical knowledge repositories: a use case. In *First International Conference on Semantic Web in Provenance Management*, 2009.

[22] Peter A. Boncz, Orri Erling, and Minh-Duc Pham. Advances in large-scale RDF data management. In *Linked Open Data - Creating Knowledge Out of Interlinked Data - Results of the LOD2 Project*. 2014.

[23] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, 2013.

[24] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, 1993.

[25] Matthias Bröcheler, Andrea Pugliese, and Venkatramanan S Subrahmanian. DOGMA: A disk-oriented graph matching algorithm for RDF databases. In *ISWC*. 2009.

[26] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *ISWC*, 2002.

[27] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An architecture for storin gand querying RDF data and schema information. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential [outcome of a Dagstuhl seminar]*, 2003.

[28] Semantic web challenge 2008. billion triples track. http://challenge.semanticweb.org/.

[29] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[30] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. Computational geometry. In *Computational geometry*. 2000.

[31] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1), 2007.

[32] Shuai Ding and Torsten Suel. Faster Top-k Document Retrieval using Block-max Indexes. In *SIGIR*, 2011.

[33] Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge-Networked Media*. 2009.

[34] Raphael A Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1), 1974.

[35] Alejandra Garcia-Rojas, Spiros Athanasiou, Jens Lehmann, and Daniel Hladky. GeoKnow: leveraging geospatial data in the web of data. *ODW*, 2013.

[36] Steve H Garlik, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. *W3C Recommendation*, 21(10), 2013.

[37] Jon Jay Le Grange, Jens Lehmann, Spiros Athanasiou, Alejandra Garcia Rojas, Giorgos Giannopoulos, Daniel Hladky, Robert Isele, AxelCyrille Ngonga Ngomo, Mohamed Ahmed Sherif, Claus Stadler, and Matthias Wauer. The GeoKnow generator: managing geospatial data in the linked data web. *Linking Geospatial Data*, 2014.

[38] Sven Groppe, Jinghua Groppe, Dirk Kukulenz, and Volker Linnemann. A sparql engine for streaming rdf data. In *SITIS*. IEEE, 2007.

[39] Paul T Groth and Yolanda Gil. Linked data for network science. In *LISC*, 2011.

[40] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Sem.*, 3(2), 2005.

[41] Peter J Haas and Joseph M Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD*, 1999.

[42] Stephen Harris and Nigel Shadbolt. SPARQL query processing with conventional relational database systems. In *ICSW*, 2005.

[43] Andreas Harth. Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2012/, 2012.

[44] Andreas Harth and Stefan Decker. Optimized index structures for querying RDF from the web. In *LA-WEB*, 2005.

[45] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *ISWC*. 2007.

[46] Johannes Hoffart, Fabian M Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard De Melo, and Gerhard Weikum. YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In *WWW*, 2011.

[47] Johannes Hoffart, Fabian M Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *AI*, 194, 2013.

[48] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11), 2011.

[49] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. Supporting Top-k Join Queries in Relational Databases. *PVLDB*, 13(3), 2004.

[50] Ihab F Ilyas, Walid G Aref, Ahmed K Elmagarmid, Hicham G Elmongui, Rahul Shah, and Jeffrey Scott Vitter. Adaptive Rank-Aware Query Optimization in Relational Databases. *TODS*, 31(4), 2006.

[51] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *CSUR*, 40(4), 2008.

[52] Ihab F Ilyas, Rahul Shah, Walid G Aref, Jeffrey Scott Vitter, and Ahmed K Elmagarmid. Rank-Aware Query Optimization. In *SIGMOD*, 2004.

[53] Apache Jena-TDB. http://jena.apache.org/documentation/tdb/index.html.

[54] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM*, 20, 1998.

[55] Leonard Kaufman and Peter Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.

[56] Jyoti Leeka and Srikanta Bedathur. RQ-RDF-3X: going beyond triplestores. In *ICDEW (DESWEB)*, 2014.

[57] Jyoti Leeka, Srikanta Bedathur, Debajyoti Bera, and Medha Atre. Quark-X: An Efficient Top-k Processing Framework for RDF Quad Stores. In *CIKM*, 2016.

[58] Jens Lehmann, Spiros Athanasiou, Andreas Both, Alejandra García-Rojas, Giorgos Giannopoulos, Daniel Hladky, Jon Jay Le Grange, Axel-Cyrille Ngonga Ngomo, Mohamed Ahmed Sherif, Claus Stadler, Matthias WAUER, Patrick WESTPHAL, and Vadim ZASLAWSKI. Managing geospatial linked data in the GeoKnow project. *The Semantic Web in Earth and Space Science*, 20, 2015.

[59] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. DBpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2), 2015.

[60] Justin Levandoski and Mohamed Mokbel. RDF Data-Centric Storage. In *ICWS*, 2009.

[61] John Liagouris, Nikos Mamoulis, Panagiotis Bouros, and Manolis Terrovitis. An effective encoding scheme for spatial RDF data. *PVLDB*, 7(12), 2014.

[62] LinkedIn Knowledge Graph. https://engineering.linkedin.com/blog/2016/10/building-the-linkedin-knowledge-graph.

[63] Vebjorn Ljosa and Ambuj K. Singh. Top-k spatial joins of probabilistic objects. In *ICDE*, 2008.

[64] Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle. Efficient Execution of Top-k SPARQL Queries. In *ISWC*. 2012.

[65] Farzaneh Mahdisoltani, Joanna Biega, and Fabian Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *CIDR*, 2014.

[66] Thomas Mandl, Fredric Gey, Giorgio Di Nunzio, Nicola Ferro, Ray Larson, Mark Sanderson, Diana Santos, Christa Womser-Hacker, and Xing Xie. *GeoCLEF 2007: The CLEF 2007 Cross-Language Geographic Information Retrieval Track Overview*. Springer Berlin Heidelberg, 2008.

[67] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. KLEE: a Framework for Distributed Top-k Query Algorithms. In *PVLDB*, 2005.

[68] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

[69] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.

[70] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[71] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, 2013.

[72] HweeHwa Pang, Xuhua Ding, and Baihua Zheng. Efficient Processing of Exact Top-k Queries over Disk-Resident Sorted Lists. *VLDB Journal*, 19(3), 2010.

[73] Jignesh M Patel and David J DeWitt. Partition based spatial-merge join. In *SIGMOD*, 1996.

[74] Patrick Hayes and Peter Patel-Schneider. RDF 1.1 Semantics. W3C Recommendation, February 2014.

[75] Mirjana Pavlovic et al. TRANSFORMERS: Robust spatial joins on non-uniform data distributions. In *ICDE*, 2016.

[76] Mirjana Pavlovic, Thomas Heinis, Farhan Tauheed, Panagiotis Karras, and Anastasia Ailamaki. TRANSFORMERS: Robust spatial joins on non-uniform data distributions. In *ICDE*, 2016.

[77] Matthew Perry and John Herring. OGC GeoSPARQL-a geographic query language for RDF data. *OGC Candidate Implementation Standard*, 2012.

[78] PostgreSQL. http://www.postgresql.org/download.

[79] Shuyao Qi, Panagiotis Bouros, and Nikos Mamoulis. Efficient top-k spatial distance joins. In *SSTD*, 2013.

[80] Kurt Rohloff and Richard E Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In *DataSys*, 2011.

[81] Sherif Sakr and Ghazi Al-Naymat. Relational processing of RDF queries: a survey. *SIGMOD*, 2010.

[82] Hanan Samet. The quadtree and related hierarchical data structures. *CSUR*, 16(2), 1984.

[83] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *ICDE*, 2009.

[84] John Sheridan and Jeni Tennison. Linking UK government data. In *LDOW*, 2010.

[85] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-Store support for RDF Data Management: not all Swans are White. *PVLDB*, 1(2), 2008.

[86] Michael Sintek and Malte Kiesel. RDFBroker: A Signature-based High-Performance RDF Store. *ESWC*, 2006.

[87] Benjamin Sowell, Marcos Vaz Salles, Tuan Cao, Alan Demers, and Johannes Gehrke. An experimental analysis of iterated spatial joins in main memory. *PVLDB*, 6(14), 2013.

[88] Fabian M Suchanek, Johannes Hoffart, Erdal Kuzey, and Edwin Lewis-Kelham. YAGO2s: Modular high-quality information extraction with an application to flight planning. In *BTW*, 2013.

[89] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: a core of semantic knowledge. In *WWW*, 2007.

[90] Octavian Udrea, Andrea Pugliese, and VS Subrahmanian. GRIN: A graph based RDF index. In *AAAI*, volume 1, 2007.

[91] Jacopo Urbani, Sourav Dutta, Sairam Gurajada, and Gerhard Weikum. KOGNAC: Efficient Encoding of Large Knowledge Graphs. In *IJCAI*, 2016.

[92] Yannis Velegrakis. Relational technologies, metadata and RDF. In *SWIM*. 2010.

[93] Virtuoso 7.2. `http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtTipsAndTricksSPARQL11FeaturesExamplesCollection`, 2016.

[94] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10), 2014.

[95] W3C SPARQL Working Group. SPARQL 1.1 overview. W3C recommendation, 2013.

[96] Andreas Wagner, Thanh Tran Duc, Günter Ladwig, Andreas Harth, and Rudi Studer. Top-k Linked Data Query Processing. In *ESWC*. 2012.

[97] Chih-Jye Wang, Wei-Shinn Ku, and Haiquan Chen. Geo-Store: a spatially-augmented SPARQL query evaluation system. In *SIGSPATIAL*, 2012.

[98] Dong Wang, Lei Zou, Yansong Feng, Xuchuan Shen, Jilei Tian, and Dongyan Zhao. S-store: An engine for large RDF graph integrating spatial information. In *DASFAA*, 2013.

[99] Dong Wang, Lei Zou, and Dongyan Zhao. gst-Store: An engine for large RDF graph integrating spatiotemporal information. In *EDBT*, 2014.

[100] Dong Wang, Lei Zou, and Dongyan Zhao. Top-k Queries on RDF Graphs. *Information Sciences*, 316, 2015.

[101] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.

[102] Kevin Wilkinson. Jena Property Table Implementation. *SSWS*, 2006.

[103] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, 2003.

[104] David Wood, Paul Gearon, and Tom Adams. Kowari: A platform for semantic web storage and analysis. In *XTech*, 2005.

[105] Dong Xin, Jiawei Han, and Kevin C Chang. Progressive and Selective Merge: Computing Top-k with ad-hoc Ranking Functions. In *SIGMOD*, 2007.

[106] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. TripleBit: a fast and compact system for large scale RDF data. *PVLDB*, 6(7), 2013.

[107] Shima Zahmatkesh. Retrieval of the most relevant Combinations of Data Published in Heterogeneous Distributed Datasets on the Web. *ISWC-DC*, 2014.

[108] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. gStore: answering SPARQL queries via subgraph matching. *PVLDB*, 4(8), 2011.

[109] Marcin Zukowski. Balancing vectorized query execution with bandwidth-optimized storage. *University of Amsterdam, PhD Thesis*, 2009.