Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Signature:

_____   _____
Kanwei Li   Date

Cost Analysis of Joins in RDF Query Processing Using the TripleT Index

By

Kanwei Li
B.S., Emory University, 2008

Mathematics and Computer Science

_____
James J. Lu
Advisor

_____
Li Xiong
Committee Member

_____
Phillip W. Hutto
Committee Member

Accepted:

_____
Lisa A. Tedesco, Ph.D
Dean of the Graduate School

_____
Date

Cost Analysis of Joins in RDF Query Processing Using the TripleT Index

By

Kanwei Li
B.S., Emory University, 2008

Advisor : James J. Lu, Ph.D.

An abstract of
A thesis submitted to the Faculty of the Graduate School of Emory University
in partial fulfillment of the requirements for the degree of
Master of Science
in Mathematics and Computer Science
2009

Abstract

Cost Analysis of Joins in RDF Query Processing Using the TripleT Index
By Kanwei Li

The Semantic Web movement has led to a growing popularity of RDF and its query languages. Clearly, good query performance is important in allowing information to be quickly retrieved from RDF datasets that are ever-increasing in size. We use the TripleT indexing scheme for RDF data as a framework to examine the cost of join operations for RDF. We analyze strategies for efficient join processing for a variety of query patterns. For queries that involve multiple join conditions, we introduce a model to predict the number of I/Os required to best order the join conditions. Experimental results validate the model using three real RDF datasets.

Cost Analysis of Joins in RDF Query Processing Using the TripleT Index

By

Kanwei Li
B.S., Emory University, 2008

Advisor : James J. Lu, Ph.D.

A thesis submitted to the Faculty of the Graduate School of Emory University
in partial fulfillment of the requirements for the degree of
Master of Science
in Mathematics and Computer Science
2009

Acknowledgements

I would like to thank Dr. James Lu for his constant guidance, patience, and editing skills that have made the writing of this thesis both possible and enjoyable. I would also like to my express my gratitude to Dr. George Fletcher for introducing me to the world of RDF, and for allowing me to build upon his wonderful TripleT index.

Dr. Phillip Hutto was the one who convinced me into switching majors as an undergrad. Without our frequent chats, I might never have developed the great interest in Computer Science that I have today. I am also indebted to Dr. Li Xiong, Dr. Eugene Agichtein, Dr. Ken Mandelberg, Dr. Michelangelo Grigni, and Dr. Vaidy Sunderam, who all filled my mind with questions and answers.

Contents

List of Figures

iv

List of Tables

Chapter 1

Introduction

A main goal of the Semantic Web movement is to allow semantic inter-connections between decentralized sources of data on the Web. RDF and SPARQL are formats designed for tagging and querying data, respectively. Up to now, not many websites have adopted Semantic Web practices such as RDFa tagging, Friend of a Friend, and SPARQL entry points. However, the rising popularity of online APIs and web services shows an encouraging trend towards resource-centric entry points to data instead of having to access the data through parsing HTML or through a browser. Recently, the British Broadcasting Corporation (BBC) has setup online SPARQL end-points for their TV and music databases that one can query. This adoption by a large company is encouraging to the future of the RDF, SPARQL, and the Semantic Web.

For widespread adoption and high user satisfaction of using RDF datasets, SPARQL queries must run quickly. This means that it is important to have

good indexing techniques to ensure scalable performance of query processing for RDF data. It is also important to have a query optimizer to help process SPARQL queries that are often very long and complicated.

1.1 Research Objective

This thesis can be viewed as a continuation of the work on an RDF indexing technique called TripleT, developed in 2008 by Fletcher and Beck [2]. Using TripleT as the framework, the goal is to better understand the requirements for building an effective SPARQL query optimizer. Specifically, we study the information necessary to facilitate good join ordering. We develop a model for predicting the number of I/Os required for a join based on TripleT using statistics that are easily collected during the creation of the index. Experiments are conducted to validate the model.

1.2 Prior Work

Fletcher and Beck [2] compared the characteristics and performance of TripleT to other indexing techniques used in production software, HexTree and MAP, and concluded that TripleT is conceptually simpler, more space efficient, while still providing the same level of support for SPARQL queries.

Much literature exists on the subject of join algorithms. Mishra and Eich [4] authored a comprehensive survey paper in 1992 on join algorithms for relational databases. The main type of join in SPARQL is the equi-join, and Mishra and Eich noted that the nested-loop join, hash join, and sort-merge join were the most applicable and performant.

Stocker et al. [9] laid the groundwork for SPARQL query optimization by defining Basic Graph Patterns (BGP) as the basic unit of SPARQL queries. Additionally, they introduced heuristics for join ordering in the cases of having pre-computed statistics and not having pre-computed statistics. The goal was to try to get smaller intermediate result sets for later joins by carrying out the most selective joins first.

Neumann and Weikum [5] introduced a SPARQL query engine called RDF-3X, and discussed how it used a dynamic programming approach with selectivity histograms to estimate cost of join paths.

Chapter 2

RDF and SPARQL

2.1 Background on RDF

RDF is a framework that describes data in the form of (subject, predicate, object) triples. It was released as a W3C recommendation in 1999 [12] and is currently the leading description model for the Semantic Web.

The greatest advantage of RDF lies in its simplicity of only using (subject, predicate, object) triples for all data representation. This creates flexibility and allows the description of both data and metadata in the same fashion. Unlike the relational model, the RDF model is "pay-as-you-go" because a predetermined schema is not required, and structure can be added later on by adding new triples to define relationships present in the data. This lack of required structure is beneficial for the World Wide Web, where there are many participants who are not in close collaboration.

RDF is also a good format for ontologies as relationships can be naturally described. For example, a predicate of one triple can be the subject of another triple. The following example shows a relationship, normally considered as metadata, being described in RDF: (John, friendof, Mark), (friendof, typeof, relationship)

The flexible nature of RDF allows this to be easily described, which is not the case in the relational model. The latter would require extra schema complexity for metadata and may require extra tables to achieve normal form.

RDF has also been used to store data from multiple domains into one dataset. For example, it would be very difficult to store structured Wikipedia content into a relational database without many tables to store different types of data. A table to describe people would have different columns than a table to describe cars, so different tables would have to be used. The RDF model would be much more appropriate since a different set of predicates could be used for each domain. The DBpedia project [1] has adopted the RDF model and converted much of Wikipedia into an RDF dataset. RDF fits naturally into the requirements of a decentralized Semantic Web.

2.1.1 Representation Formats

There are two main representation formats for RDF: XML and Notation 3. Additionally, a subset of Notation 3 called Turtle is in development. Notation 3 and Turtle are designed for human-readability and are more concise than XML, but require a proper parser instead of an XML parser. What is important is that while the representation format may differ, the data represented is the same. Figures 2.1 and 2.2 show different representations of two triples: (http://wikipedia.org/wiki/France, dc:title, France), (http://wikipedia.org/wiki/France, dc:publisher, Wikipedia)

```
<rdf:RDF
    xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://wikipedia.org/wiki/France">
    <dc:title>France</dc:title>
    <dc:publisher>Wikipedia</dc:publisher>
  </rdf:Description>
</rdf:RDF>
```

Figure 2.1: XML representation of RDF

The subject of both triples is "http://wikipedia.org/wiki/France", and XML conveys this by having the predicate be the parent node for both of the predicate/object pairs. In Notation 3, this is accomplished by the nested indentation and having a semicolon after the first triple instead of a period.

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.

<http://wikipedia.org/wiki/France>
  dc:title "France";
  dc:publisher "Wikipedia".
```

Figure 2.2: Notation 3 representation of RDF

This is purely for compactness and readability; nothing would be different if the subject were explicitly written twice.

RDF supports namespaces by allowing the declaration of prefixes. In the example above, the "dc" prefix is defined, and "dc:title" is thus short for "http://purl.org/dc/elements/1.1/title". This convention allows readable name-spacing while also reducing data size.

Additionally, RDF allows the assignment of types to atoms, such as integer, double, decimal, and boolean.

2.2   RDF Datasets

Fundamentally, an RDF dataset is a collection of RDF triples. As such, there are many possible storage formats, but most RDF datasets are distributed in plain-text format for universal portability.

Many projects that use RDF data use a relational database to store triples

by using a table with three columns: subject, predicate, and object. Triples can then be accessed using SQL. While this method leverages current relational database technology, tabular relational stores do not fit the graph-like nature of RDF, and thus new RDF-native stores have been developed.

As of 2009, the primary RDF-native stores available are:

1) Jena [3], an open-source Java framework that supports SPARQL

2) Sesame [7], an open-source framework in C that uses its own querying language called SeRQL (Sesame RDF Query Language)

3) Virtuoso [11], a framework that is available in both commercial and open-source packages, both supporting SPARQL

Additionally, a framework named RDF-3X [5] was introduced in 2008, and the authors state that it is available on request for non-commercial use.

2.3   Datasets Used in the Thesis

It is important to perform experiments on real-world data. This thesis will use a subset of one million triples of each of:

1) The DBpedia dataset [1], an extract of Wikipedia articles converted into RDF triples. Semantic information, such as categories and infoboxes, have been kept as well.

2) The Uniprot dataset [10] that describes proteins and provides annotation data.

3) The SP^{2}Bench [8] dataset, a recently developed dataset that tries to emulate an authorship database, such as DBLP.

2.3.1 Dataset Statistics

To get an idea of the distribution of atoms in the datasets, the number of unique atoms in the subject, predicate and object positions were computed and tabulated in Table 2.1. Additionally, the number of atoms that could possibly take part in one of the 3 possible joins between different positions were calculated and tabulated in Table 2.2. The abbreviations S, P, and O denote subject, predicate, and object respectively.

Dataset	Uniq Subj	Uniq Pred	Uniq Obj
DBpedia	136108	8878	282199
Uniprot	592639	79	294676
SP^{2}Bench	31629	61	81919

Table 2.1: Subject, Predicate, Object statistics

Dataset	SP	SO	PO
DBpedia	0	48085	0
Uniprot	0	105560	8
SP^{2}Bench	0	14816	0

Table 2.2: Join statistics

2.3.2 Dataset Discussion

DBpedia is interesting in that it has many unique predicates, while Uniprot and SP^{2}Bench have very few. This is because DBpedia contains multiple domains of knowledge, and each domain uses its own set of predicates. For example, predicates to describe people include "dbpedia:ontology/religion", "dbpedia:ontology/spouse" and "dbpedia:ontology/birthdate", which would logically only be used to describe people and not other kinds of resources. The broad scope of DBpedia means it has more unique predicates than the other two.

Also, it is interesting that predicates never appear as subjects and very seldomly appear as objects. This shows that none of these datasets create an ontology where metadata is treated as data and a hierarchy is created.

2.4 Background on SPARQL

SPARQL is the W3C-backed query language for RDF and became an official W3C recommendation in 2008 [13]. The language is inspired by SQL and adopts many of its keywords, such as SELECT, DISTINCT, FROM, and WHERE. Because of RDF's graph-like nature, additional keywords are necessary, such as PREFIX, OPTIONAL, UNION, and GRAPH. An example

SPARQL query is shown in Figure 2.3.

```
SELECT ?capital ?state
WHERE {
  ?x city_name ?capital ;
  ?x capital_of ?y ;
  ?y state_name ?state ;
}
```

Figure 2.3: SPARQL: Return all states and their capitals

SPARQL queries are defined by basic graph patterns (BGPs), which are conjunctions of simple access patterns (SAPs). An SAP is triple whose elements are any combination of atoms or variables, where variables are prefixed by a "?". Atoms in SAPs are called bound patterns, and variables are called unbound patterns. An SAP selects triples that match all of its bound and unbound patterns. A few examples:

The SAP (John, friendof, ?x) would match any triple with both subject "John" and predicate "friendof". It would thus match (John, friendof, Mark), but not (John, parentof, Tim).

A BGP can have more than one SAP. For example, consider the BGP: (John, friendof, ?x) ∧ (?x, friendof, Tim). The left SAP matches any triple with subject "John" and with predicate "friendof"; the right SAP matches any triple with predicate "friendof" and with object "Tim". To compute the

result set of ?x, atoms that appear in both the object position of the triples matched by the left SAP and the subject position of the triples matched by the right SAP are selected.

If a query for the above BGP were run on the datset consisting of (John, friendof, Mark), (John, friendof, Alex), (Mark, friendof, Tim), the result set of ?x would be { Mark }. Semantically, the query asks, "Which friends of John are also friends of Tim?" The query requires the use of a join on the object position of the left SAP and subject position of the right SAP.

The major differences between SPARQL and SQL are:

1) SPARQL allows the use of variables in SAPs that match any atom in that position. The SAP (?a, ?b, ?c) by itself would match all the triples in the dataset.

2) There is no explicit equi-join operator in SPARQL, so the query engine has to deduce what needs to be joined. Consider the example query in Figure 2.3. There are four distinct variables in the query, and two joins are necessary: Subject–Subject on ?x, and Object–Subject on ?y.

In conclusion, RDF is a flexible format that makes it well suited to the Semantic Web, and SPARQL provides a natural way to query RDF datasets.

Chapter 3

Indexing Techniques

Real-world SPARQL queries usually consist of multiple SAPs, and most SAPs require lookup of atoms. Thus, before querying is performed, RDF data should first be indexed so that the lookup of atoms for SPARQL queries can scale to larger datasets. Most RDF datasets are designed to be read, so fast lookup is more important in the long run than fast index creation.

3.1 B+ Trees

Much like relational databases, indexing schemes for RDF generally rely on the B+ tree as the data structure of choice. The string representation of an atom can be used as the key in the tree, and the triples that contain the atom as the values.

A B+ tree is a balanced tree whose depth and width can be adjusted by setting parameters that determine the level of branching. A deep tree

allows faster comparisons of keys at any level, but requires more levels to be traversed on average. A shallow and wide tree requires more comparisons at every level and more space per block, but there are fewer levels on average and thus fewer I/O operations needed. The parameters should be set to best match the filesystem and hardware.

Because datasets can potentially be gigabytes in size, it is often necessary for the index itself to reside on disk. A B+ tree allows an index to be stored on disk by using block-oriented storage, meaning that each individual node, along with its pointers and/or data, are stored in blocks which can be accessed with I/O operations. This way, data can be accessed through a few I/O operations to go from the root of the tree to a leaf node with data. Because I/O operations are orders of magnitude slower than main memory accesses, the classical measure of query performance is a function of I/O operations [2].

For the experiments performed in this thesis, all nodes and data are stored in main memory, and I/O accesses are simulated by recording an I/O access whenever a node in the B+ tree is visited. This way, good performance can be achieved when conducting experiments while still measuring the correct I/O cost.

3.2 RDF Indexing Schemes

The current state of the art indexing schemes for RDF are MAP and HexTree [2]. TripleT is an indexing scheme developed by Fletcher in Beck in 2008. While they all use B+ trees, they differ in how many trees are used and what the leaf nodes of the trees contain.

3.2.1 MAP and HexTree

MAP is an indexing scheme that utilizes 6 indices, one for each possible ordering of subject, predicate and object. For example, to look up (John, friendof, ?x), one could either look in the SPO index for "John#friendof#?x", or the OPS index for "?x#friendof#John", etc.

HexTree is a similar scheme that also utilizes 6 indices, one for each possible pairing of atoms: SO, OS, SP, PS, OP, PO. For a query like (John, friendof, ?x), (?x, friendof, Tim), the SP index is queried to obtain the result set for the first SAP, and the PO index is queried for the second SAP. The two result sets are then joined to get the final result.

The disadvantages of these two schemes is the loss of data locality that results from having multiple B+ trees, and thus queries may incur extra I/O costs of having to look in multiple indices [2].

3.2.2   TripleT

TripleT (diagrammed in Figure 3.1) is an indexing scheme that uses only one B+ tree whose leaf nodes represent atoms. Each leaf node holds a payload that is further split into three buckets: subject, predicate, and object. For example, if the triple (Shakespeare, dc:author, Romeo and Juliet) were stored into a TripleT index, there would be three leaf nodes in the B+ tree, "Shakespeare", "dc:author", and "Romeo and Juliet", each of which have a payload. The triple would exist in the subject bucket of the "Shakespeare" payload, in the predicate bucket of the "dc:author" payload, and finally in the object bucket of the "Romeo and Juliet" payload. There is duplication in this scheme, as each triple appears in three different buckets. However, the triple can be accessed from any of its atoms through one index lookup. Additionally, only one B+ tree is necessary to store all of the data.

Because TripleT is designed to be stored on disk, the payloads also have to be stored on disk. Thus, the buckets themselves may be split up into separate blocks whose size should be set to fit the filesystem. For TripleT, a traversal of all triples in a bucket will incur an I/O cost equal to the total number of blocks needed to store that bucket. This factor become an important when performing experiments using TripleT.

17

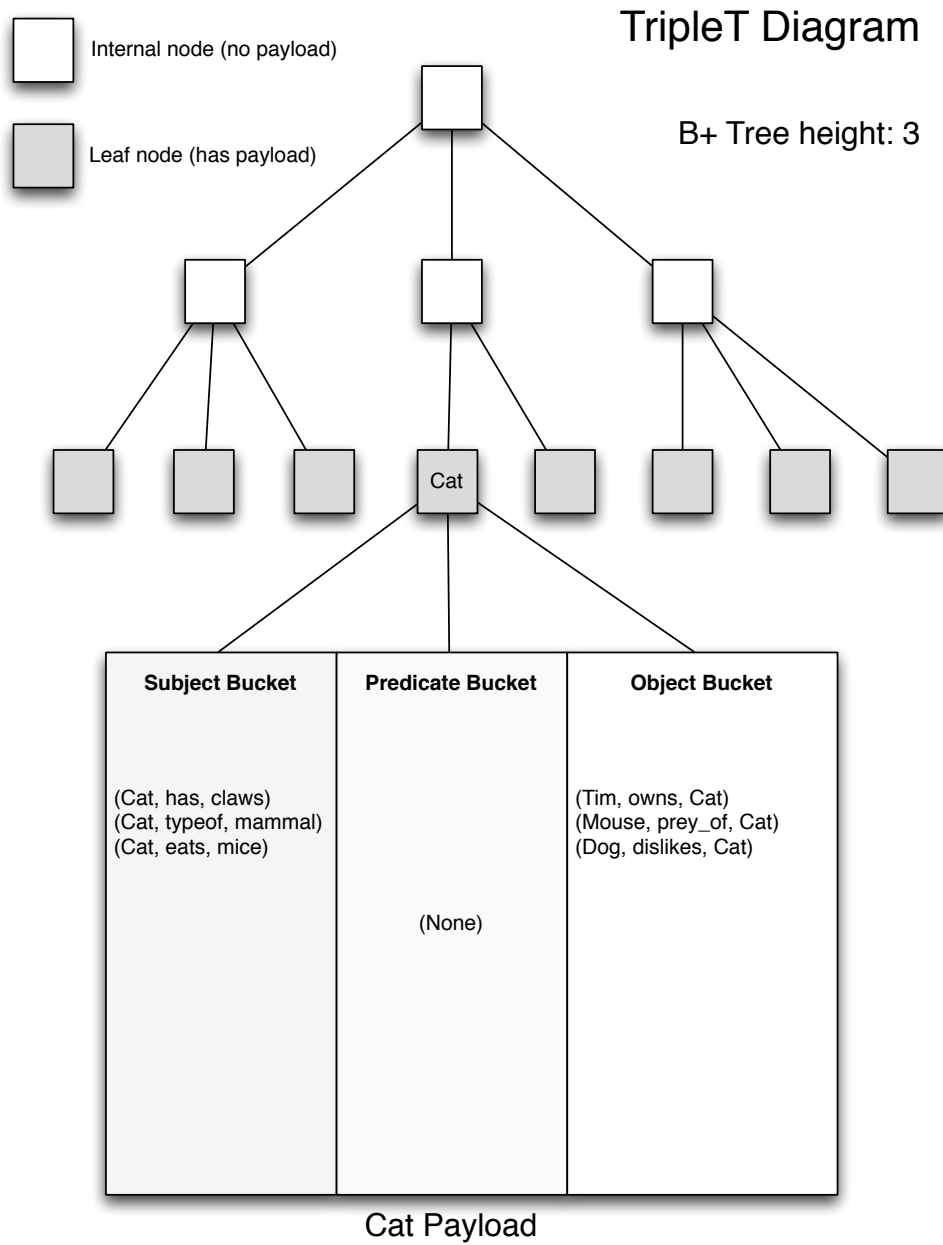Figure 3.1: TripleT diagram

This thesis builds on an implementation of TripleT written in Python that stores the index in main memory and simulates I/O accesses by keeping track of I/O operations needed if the index were stored on disk. All I/O measurements use this simulated metric. A "blockenize" function is used to calculate the number of blocks necessary to store any particular payload bucket.

Chapter 4

Join Algorithms

One of the most important and useful database operations is the join operation. In relational databases, this means that multiple tables are joined together based on some criteria, most often equality, resulting in a subset of the cartesian product of the participating tables.

In SPARQL, a join occurs when the same variable exists in different SAPs. In the example of (John, friendof, ?x), (?x, friendof, Tim), ?x exists in both SAPs and represents an equi-join. Because these joins frequently occur in SPARQL queries and are costly when processed naively, optimizing them can yield large performance benefits.

Mishra's survey paper on relational joins [4] discusses three major algorithms used for equi-joins that look for equality between elements in two different tables: the nested-loop join, hash join, and sort-merge join. These joins can also be used for SPARQL queries. Additionally, there exists the

index nested-loop join that will be introduced and used in later chapters.

4.1 Nested-loop join

The nested-loop join iterates one relation over another relation in a nested loop fashion. The I/O cost is $\#blocks(R1) + |R1| \times \#blocks(R2)$, where R1 and R2 are the joining relations. This join is easy to implement and does not require sorted relations to work. However, the complexity makes it unattractive for joining relations with many triples.

Additionally, because triples to be joined are all contained in physical blocks, the nested-loop join could exhibit very bad behavior of continually looking at blocks over again in the case that the inner loop spans multiple blocks. A simple modification that processes all of R2 for each block of R1 may improve the I/O cost to $\#blocks(R1) + \#blocks(R1) \times \#blocks(R2)$. This variant is also known as the block nested-loop join.

4.2 Hash join

The hash join requires the creation of a hash table for the triples of one relation, with the key being the atom and the values being the triples containing that atom. Looking up an atom in the hash table can be achieved in amor-

tized O(1) time. To minimize collisions, it is better to hash the triples of the smaller list. Since all triples in both relations have to be visited, it does not help to hash the triples of the larger relation.

Once the hash table has been built for the smaller relation, the triples of the larger relation are iterated and use the hash table to see if they satisfy the join condition. Each time two triples satisfy the condition, they are both are added to the result set.

The I/O cost for the hash join is $\#blocks(R1) + \#blocks(R2)$, where R1 and R2 are the joining relations. All blocks of both relations have to be visited, so the hash join has a very fixed I/O performance profile.

4.3   Sort-Merge join

The sort-merge join takes advantage of the merge operation, which can be carried out on two sorted relations with $\#blocks(R1) + \#blocks(R2)$ I/O operations, where R1 and R2 are the joining relations. The requirement for being sorted is extremely important since otherwise it would take additional CPU time and I/Os to sort the relations. The structure of TripleT allows the triples of each payload bucket to be sorted on one position: subject, predicate, or object. Statistics could help determine which position would benefit most

for being sorted. If there are many joins on the subject position, then it is better to have the buckets sorted on the subject position so that a merge join can be used without additional sorting. If it is found that sort-merge is much faster than the alternatives, then extra buckets could potentially be created so that triples are sorted on every position, thus trading index size for join performance.

Another interesting aspect of the sort-merge join is that unlike the nested-loop and hash joins, not all blocks in each bucket have to always be visited. If the merge operation gets to the end of one relation, it can then stop even if it does not visit the rest of the other relation. Thus, while all the blocks of one relation always have to be accessed, it is possible to skip blocks of the other relation.

4.4   Measuring Join Performance

The main performance metric for join algorithms is the number of I/O operations required, since these are almost always the bottleneck and are also platform-independent. An I/O operation is performed when accessing any node in the B+ tree. Because the B+ tree is balanced and all payloads are at the leaf nodes, the I/O cost for looking up any atom is always constant

and equal to the height of the B+ tree.

I/O operations are also necessary when traversing payload buckets that may be stored across multiple blocks to fit all the triples. Thus, common predicate atoms often take many blocks to store, while most subject and object atoms are not too common and only require one block to store. This statistic is considered in later chapters and tabulated in Table 6.2 for the three datasets.

As discussed previously, the join algorithms usually have to visit every block of the relations joined on at least once. The exception is the sort-merge join, which can in some cases finish traversing one relation and thus stop traversing the other relation. If the other relation had unvisited blocks, then it is possible to save some I/O operations.

4.4.1 Join CPU Performance on Synthetic Data

Even though I/O performance is usually the better metric for joins, it would be interesting to see the pure algorithmic performance of the nested-loop, hash, and sort-merge joins, and also to see if CPU performance could potentially become a bottleneck. Two random sets of triples were created, each triple populated with a certain number of unique atoms. The atoms were

scattered across all positions, which would not be the case for real-world data; for example, atoms found as predicates rarely also exist as subjects or objects. Sorting time was included for the sort-merge join experiments, as it cannot be guaranteed that the data will always be pre-sorted.

The three kinds of joins described in this paper were implemented in Python. All experiments were run on a MacBook Pro 2.2 GHz with 4GB RAM and a standard build of Python 2.6.2. The timings were taken using Python's *timeit* module which is designed to profile code.

# Unique Atoms	Number of Triples	Nested-loop	Hash	Sort-merge
50	10000	39.974s	0.822s	3.187s
50	100000	2201.645s	6.035s	621.417s
500	10000	38.745s	0.164s	2.942s
500	100000	2711.327s	79.355s	638.210s

Table 4.1: Join CPU performance

The nested-loop join exhibits expected super-linear run-time behavior, as increasing the size of the data by a factor of 10 increased run-time by a factor of over 35. The hash join performed better than sort-merge join but did not scale as well to larger synthetic datasets.

This experiment was also performed to ensure that the joins were correctly implemented as the result sets of the different joins were compared at the end and were found to be the same.

4.4.2   Join I/O and CPU Performance on Datasets

I/O performance is a combination of bucket traversals and index lookups. The nature of B+ trees makes all data appear at leaf nodes, so the I/O cost of index lookups for all atoms is the same. However, each atom will have a different payload with three buckets of varying size, resulting in different numbers of blocks that have to be traversed. This is how I/O performance can differ when joining on buckets of different atoms.

To compare the I/O performances of the three different joins, queries of the form of (a, b, ?v) $\wedge$ (?v, c, d) were performed on the datasets, and I/O and CPU performance were measured. Here, "a", "b", "c", and "d" represent arbitrarily picked but fixed atoms.

Dataset	Nested-loop Join		Hash Join		Sort-merge Join	
	Avg I/O	Avg CPU	Avg I/O	Avg CPU	Avg I/O	Avg CPU
DBpedia	2.0221	1.832e-5	2.0092	9.732e-6	2.0092	2.246e-5
Uniprot	2.0365	2.011e-5	2.0043	8.848e-6	2.0043	2.037e-5
SP^{2}Bench	2.0503	1.163e-5	2.0503	5.545e-6	2.0503	9.971e-5

Table 4.2: Join CPU and I/O performance (CPU measured in seconds)

The CPU times were all fairly similar, because the buckets in the real datasets contained fewer triples than the buckets used in the synthetic benchmark. As predicted, I/O performance of the nested-loop join was the worst as it would often have to visit blocks that had been visited before. No I/O

difference was found between the hash join and sort-merge join, so the special case where the sort-merge join could use less I/Os never occurred.

In conclusion, the hash join seems to be the equi-join of choice for our purposes as it does not require relations to be sorted while still providing similar CPU and I/O performance compared to the sort-merge join.

Chapter 5

Query Optimization

In relational databases, when multiple join conditions exist in a query, the choice of which condition to check first may significantly affect the performance of the overall query process. Similar decisions and trade-offs exist for SPARQL queries, where each variable that is shared by a pair of SAPs represents one join condition.

5.1 Join Ordering

Assuming two equi-join conditions, it would be more efficient to compute the condition that produces the smaller result set first. This would minimize the number of checks necessary to ensure the second join condition. This may be seen in the following simple example.

Consider the BGP: (?x, a, ?y) ∧ (?y, b, ?z) ∧ (?x, c, ?z). To satisfy the query, the initial join can be performed between any two SAPs. Assume 1000

atoms satisfy the join on ?x, 100 atoms satisfy the join on ?y, and 10 atoms satisfy the join on ?z. If the left SAP and right SAP were joined on ?x first, the resulting relation with 1000 atoms would then have to be joined with the middle SAP. If the middle SAP and right SAP were joined on ?z first, the resulting relation would only have 10 atoms to join with the left SAP. The final end result for both join orderings would be the same, so performing the join that produces the smaller result set first is usually best.

5.2 Processing SPARQL Queries with TripleT

One important difference between SPARQL joins using TripleT and traditional relational joins is that TripleT provides an index for all atoms in the graph. Thus, any SAP with an atom can be computed efficiently through the index.

To get a sense of what processing SPARQL queries in TripleT entails and what kind of query paths are available, an example query will be analyzed. We first introduce some notations and definitions:

Given a dataset G, let T_G denote the TripleT index associated with G, and let $S(G)$, $P(G)$, $O(G)$ denote the set of subject, predicate, and object atoms in G respectively. Given an atom "a", $S_a(G)$ represents the subject

bucket of the "a" payload in T_G. Similarly, $P_a(G)$ and $O_a(G)$ correspond to the predicate and object buckets of the "a" payload, respectively.

Consider the BGP: (a, b, ?v) $\wedge$ (?v, c, d), where "a", "b", "c" and "d" are distinct atoms. There are many practical applications for this query form, such as, "What journal citations does resource 'Q2P320' cite?" This query would match the Uniprot triples shown in Figure 5.1.

```
@prefix uniprot: <purl.uniprot.org/>.
@prefix w3c: <www.w3.org/1999/02/22-rdf-syntax-ns>.

uniprot:uniprot/Q2P320 uniprot:core/citation uniprot.rdf#_5C0A .
uniprot.rdf#_5C0A, w3c:type, uniprot:core/Journal_Citation .
```

Figure 5.1: Triples matching (a, b, ?v) $\wedge$ (?v, c, d)

This query is interesting in there are two ways to obtain its answers. One way, shown in Figure 5.2, is to find the triples that match the atoms of the left SAP, the triples that match the atoms of the right SAP, and then perform a join on ?v.

To get the triples that match the left SAP, we could look at T_G to obtain $S_a(G)$, and within those triples, select those that have predicate "b". The resulting set would be the triples that match the left SAP. We might first compute $P_b(G)$ instead, but as Table 2.1 shows, there are typically more

unique subject than predicate atoms. Hence, the selectivity in the subject position is higher, and a smaller set of triples is likely to be obtained from looking up a subject atom than a predicate atom.

1) Let $Left = \sigma_{pred=b}(S_a(G))$
2) Let $Right = \sigma_{pred=c}(O_d(G))$
3) return join(Object–Subject, $Left$, $Right$)
Note: In step 3, "join" can be any of the three algorithms discussed previously, and the first parameter indicates the join condition to be checked in $Left$ and $Right$.

Figure 5.2: Joining Left and Right SAPs

Another strategy, shown in Figure 5.3, is to find all the triples that match one SAP and then do an index lookup for each unique atom in the variable position. This approach is also known as the index nested-loop join. If the left SAP were chosen, then the unique atoms that appear as objects of the triples satisfying the left SAP are determined. For each of these atoms, an index lookup is done and the subject buckets are concatenated to create an intermediate set of triples. Finally, a join is performed between the intermediate set of triples and the set of triples that matched the left SAP. This approach is usually not optimal because more than 2 index lookups are necessary, but might be reasonable if the atoms in one SAP had large buckets that would incur high I/O cost to scan through.

1) Let $Left = \sigma_{pred=b}(S_a(G))$
2) Let $unique\_objects = \{t[obj] \mid t \in Left\}$
3) Let $Right = [\,]$
4) for obj in $unique\_objects$: $Right$.append($S_{obj}(G)$)
5) return join(Object–Subject, $Left$, $Right$)

Figure 5.3: Index lookup for each unique atom in the variable position

5.3   Discussion

From the above discussion, the following basic query processing strategies are clear:

1) When processing an SAP with two variables and one atom, use the index to retrieve the bucket associated with the atom.

2) When processing an SAP with more than one atom, use the index to retrieve the bucket of the most selective atom. This suggests maintaining a count of the number of occurrences for each atom, which may be unfeasible for a large dataset. A reasonable approximation, on the other hand, is to maintain the number of unique atoms in the subject, predicate and object positions. The higher the number, the greater the likelihood of that position having a high selectivity.

Even when more than one join condition exists, observation #1 implies that efficient processing is achievable as long as at least one atom exists for each SAP. The most challenging queries are those that contain SAPs with

variables in each position. Indeed, such queries may require multiple index traversals that cannot be determined apriori and may give rise to different join ordering. Investigating these types of queries is the focus of the next chapter.

Chapter 6

Models and Experiments for All-Variable SAPs

Consider the BGP with one SAP that contains all variables: (a, ?y, ?x) $\wedge$ (?x, ?y, ?z). The all-variable SAP would match all triples in the dataset by itself, but the restrictions placed by the first SAP through the "a" atom will often eliminate a large number of the triples. Assuming we first retrieve $S_a(G)$ for the left SAP, two reasonable paths to the answer set may be taken:

1) For each atom b in the object position of $S_a(G)$, retrieve $S_b(G)$ and join with $S_a(G)$ on the variable ?y. Concatenate the results of the joins.

2) For each atom b in the predicate position of $S_a(G)$, retrieve $P_b(G)$ and join with $S_a(G)$ on the variable ?x. Concatenate the results of the joins.

Formally, these can be expressed as:

$$S_a(G) \bowtie \bigcup \{S_b(G) \mid b \in \pi_{obj}(S_a(G))\} \qquad (6.1)$$

$$S_a(G) \bowtie \bigcup \{P_b(G) \mid b \in \pi_{pred}(S_a(G))\} \qquad (6.2)$$

To choose correctly between options #1 and #2 requires knowing:

(a) the number of unique atoms in $\pi_{obj}(S_a(G))$ and $\pi_{pred}(S_a(G))$, and

(b) the size of the bucket $S_b(G)$ or $P_b(G)$ for each b in $\pi_{obj}(S_a(G))$ and $\pi_{pred}(S_a(G))$.

The first contributes to the number of I/Os required to find the relevant buckets for the all-variable SAP, and the second contributes to the number of I/Os required to join these buckets with $S_a(G)$. The number of I/Os necessary for a join path can be modeled by Equation 6.3, where α is either $S_{t[obj]}(G)$ or $P_{t[pred]}(G)$ for our example.

$$\sum_{t \in S_a(G)} (height(T_G) + \#blocks(\alpha)) \qquad (6.3)$$

The actual calculation of Equation 6.3 is difficult, however, as the amount of information that needs to be maintained for (a) and (b) can be very large.

Intuitively, an easy to maintain measure that will give a fairly accurate indication of the preferred join path is the selectivity values for pairs of positions, e.g. Subject–Object. This is a function of the number of atoms that can possibly participate in any pair of positions. The higher the number,

Cat

Option #1:
Going through ?y

(Cat, **?y**, ?x)

Unique predicates:
has, typeof, eats

Subject Bucket

(Cat, has, claws)
(Cat, typeof, mammal)
(Cat, eats, mouse)
(Cat, typeof, pet)

Option #2:
Going through ?x

(Cat, ?y, **?x**)

Unique objects:
claws, mammal, mouse, pet

(Cat, has, claws)
(Dog, has, paws)
(Dog, typeof, mammal)
(Cat, typeof, mammal)

(Mouse, typeof, vermin)
(Cat, typeof, pet)
(Cat, eats, mouse)
(Mouse, eats, cheese)

(Cat, has, claws)
(Eagle, has, claws)
(Dog, typeof, mammal)
(Hair, attr_of, mammal)

(Cat, typeof, mammal)
(Cat, eats, mouse)
(Farmer, dislike, mouse)
(Human, own, pet)

(Cat, typeof, pet)
(Friendly, attr_of, pet)

Join

Join

Join Result
(same result for both)

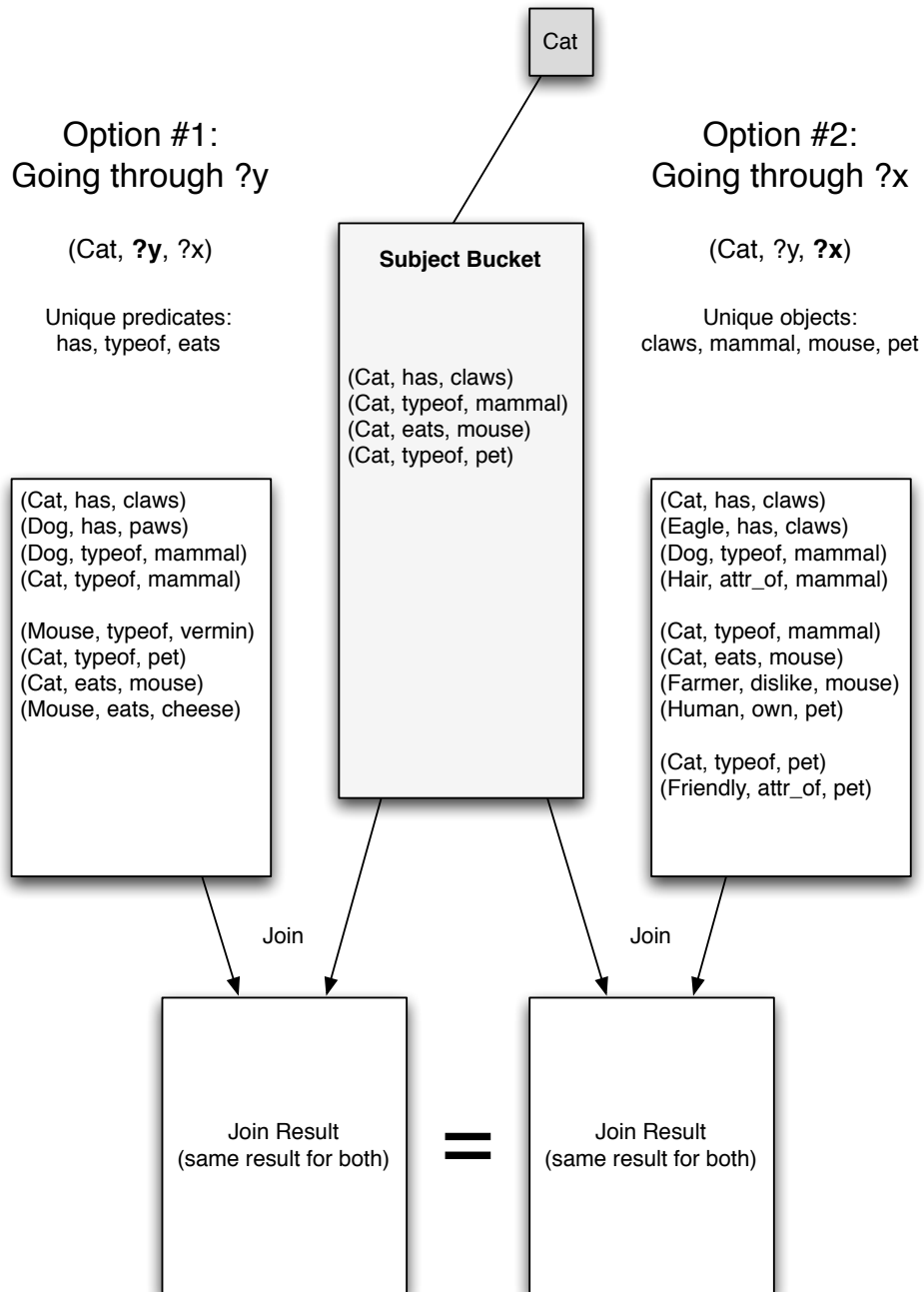**=**

Join Result
(same result for both)

Figure 6.1: Join diagram

the lower its selectivity. For instance, Table 2.2 tells us that a Subject–Object join is less selective than a Predicate–Object join since many more atoms appear in both positions in the first pair than in the second pair.

We do not consider this measure in our study since it does not provide particularly useful information for joins on the same position. Instead, we propose a different approximation based on the following:

(a') The average number of unique subject, predicate, and object atoms per subject, predicate and object bucket respectively.

(b') The average number of blocks per subject, predicate and object bucket.

These two statistics for the three datasets we use are shown in Tables 6.1 and 6.2 respectively.

Dataset	Subject Bucket		Predicate Bucket		Object Bucket	
	Uniq Pred	Uniq Obj	Uniq Subj	Uniq Obj	Uniq Subj	Uniq Pred
DBpedia	4.72	5.77	72.39	44.43	2.79	1.40
Uniprot	1.47	1.69	11003	3859	3.39	1.03
SP^{2}Bench	4.99	5.14	2585	1361	1.99	1.01

Table 6.1: Unique atoms per position per bucket

That (a') and (b') approximate (a) and (b), respectively, is clear. Using these statistics, we can estimate the number of I/Os necessary for both options #1 and #2, and we obtain new formulae for estimating I/Os for options

Dataset	$Avg\ \#blocks(S(G))$	$Avg\ \#blocks(P(G))$	$Avg\ \#blocks(S(G))$
DBpedia	1.03	2.94	1.04
Uniprot	1.00	159.35	1.02
SP^{2}Bench	1.00	66.90	1.02

Table 6.2: Average Subject, Predicate, Object bucket sizes

#1 and #2 in Equations 6.4 and 6.5 respectively.

Notation: Let $A, B \in \{subj, pred, obj\}$. We denote $\gamma^B_{A_a}(G)$ as the average number of unique B-atoms for each A-atom in G, and $\beta_A(G)$ as the average number of blocks for A in G. For instance, $\gamma^{obj}_{subj_a}(G)$ is the average number of unique objects for each subject atom "a".

$$\gamma^{obj}_{subj_a}(G) \times \beta_{subj}(G) \tag{6.4}$$

$$\gamma^{pred}_{subj_a}(G) \times \beta_{pred}(G) \tag{6.5}$$

The hypothesis is that by using the two statistics described above, we can predict the join path that uses less I/Os. The following six queries will be used in our experiments to validate this hypothesis. Note that the underscores represent a variable which does not participate in the join.

Q1. (a, ?y, ?x) ∧ (?x, ?y, _)

Q2. (a, ?y, ?x) ∧ (_, ?y, ?x)

Q3. (?x, a, ?y) ∧ (?x, _, ?y)

Q4. (?x, a, ?y) ∧ (?y, _, ?x)

Q5. (?x, ?y, a) ∧ (?x, ?y, _)

Q6. (?x, ?y, a) ∧ (_, ?y, ?x)

For each query, the predicted number of I/Os required to go through ?x

and I/Os required to go through ?y were calculated using our model. The pre-

dicted difference was also calculated as $(predicted\ ?x\ IOs - predicted\ ?y\ IOs)$.

500 queries were run for Q1, Q2, Q5 and Q6, and 100 queries were run for

Q3 and Q4 as the latter had less than 100 unique joins for that type of query.

The values of the actual difference between the amount of I/Os going through

?x requires and the amount of I/Os going through ?y requires was recorded

along with the number of "victories" for each variable.

6.1 DBpedia Results

	Pred. ?x	Pred. ?y	Pred Diff.	Act. Diff.	?x better	?y better	Tie
Q1	29.72	69.38	-39.66	-1448	497	3	0
Q2	30.00	69.38	-39.38	-1218	461	30	9
Q3	372.81	231.04	141.77	-169	48	16	36
Q4	376.43	228.81	147.62	87	18	54	28
Q5	14.37	69.38	-55.01	-567	447	18	35
Q6	14.51	69.38	-54.87	-500	479	12	9

Table 6.3: I/O results for DBpedia

The results for DBpedia generally fit the expected outcomes, except for Q3 where ?y was predicted to do better but ?x did better instead.

What is interesting to note is that the actual difference values were usually larger than what was calculated from our model based on averages. Also, the margins of victories in Q3 and Q4 were very close, which suggests that the model might not be good for predicting I/O performance for these queries on DBpedia without further tweaking.

6.2   Uniprot Results

	Pred. ?x	Pred. ?y	Pred. Diff.	Act. Diff.	?x better	?y better	Ties
Q1	8.45	1168.65	-1160	-1699	500	0	0
Q2	8.62	1168.65	-1160	-1620	497	3	0
Q3	55020	19681	35339	62989	22	77	1
Q4	56120	19295	36825	43731	21	75	4
Q5	16.95	1168.65	-1152	-489	496	3	1
Q6	17.29	1168.65	-1151	-491	497	2	1

Table 6.4: I/O results for Uniprot

The Uniprot results correctly predicted the better join path for all queries, usually by a large margin.

	Pred. ?x	Pred. ?y	Pred. Diff.	Act. Diff.	?x better	?y better	Ties
Q1	25.70	1669	-1643	-2174	500	0	0
Q2	26.21	1669	-1643	-1854	500	0	0
Q3	12925	6941	5984	7960	4	80	16
Q4	13183	6805	6378	8890	1	93	6
Q5	9.95	1669	-1659	-380	499	1	0
Q6	10.15	1669	-1659	-391	499	0	1

Table 6.5: I/O results for SP^{2}Bench

6.3   SP^{2}Bench Results

Finally, the SP^{2}Bench results show similar numbers to Uniprot, and the statistics correctly predicted the better path for each query by a large margin.

6.4   Variant Query Forms

The models introduced can also be used for variant query forms that involve three SAPs.

For example, consider the BGP: (a, ?y, _) ∧ (_, ?y, ?x) ∧ (b, _, ?x). While the form now has three SAPs and two atoms, the join plans available and number of I/Os necessary are actually very similar to those for our previous example BGP of (a, ?y, ?x) ∧ (?x, ?y, ?z). The only difference is that after joining one SAP to the middle SAP, another join has to be done with the remaining SAP. Our models should be able to use the same method to predict

for the object and subject positions in DBpedia were very similar, which would make the calculated expected I/Os similar as well. Table 6.1 shows that for DBpedia, the average number of unique objects per subject bucket was 5.77, and the average number of unique subjects per object bucket was 2.79. Additionally, the average bucket sizes for both positions were similar. The corresponding statistics for SP^{2}Bench were 5.14 and 1.99 respectively, and the same query was correctly predicted. The limitation of our model is for queries that utilize statistics that have close values.

Note that although the predicted differences and actual differences often varied greatly, the models still predicted the correct outcome.

Chapter 7

Conclusion

We implemented the nested-loop, hash, and sort-merge joins. Synthetic benchmarks found that the nested-loop join predictably performed much worse than the other two in terms of CPU performance. The hash join was about an order of magnitude faster than the sort-merge join when the latter had to sort the lists. Benchmarks performed on real datasets showed much smaller differences in CPU performance as there were much fewer triples considered for the joins. In terms of I/O performance, the nested-loop join was about 10% worse than the hash join and sort-merge join, which performed equally.

We analyzed different kinds of SAP patterns found in SPARQL queries to better understand how to develop a query processing algorithm using Fletcher and Beck's TripleT index. To process an SAP with at least one atom, an index lookup should be performed to obtain the bucket of the atom

in the most selective position. To handle all-variable SAPs, we introduced a model to estimate I/O, apriori, and conducted experiments to verify our model, which was found to be correct in nearly all cases. The technique is attractive in that it only requires the maintaining of a few statistics that can be calculated very efficiently during the index creation process.

For future work, an implementation of TripleT that stores the index on disk instead of in-memory is necessary to obtain a more accurate picture of performance. Doing so would allow accurate timing measurements to be taken instead of having to look at CPU time and I/O accesses separately, and having to guess at the relationship between the two metrics.

For query optimization, the obvious next step is to experiment with larger datasets using a variety of queries and eventually develop a query optimizer that can process more complicated SPARQL queries. For example, consider the SP^{2}Bench benchmark 5b query [8] shown in Figure 7.1. The BGP form of this query is:

(?v, a, b) $\wedge$ (?v, c, ?x) $\wedge$ (?y, a, d) $\wedge$ (?y, c, ?x) $\wedge$ (?x, e, _)

It was reported that all current SPARQL engines performed poorly on this query [6]. A query with this many variables should greatly benefit from having good join ordering.

```
/*
 * Return the names of all persons that occur as author
 * of at least one inproceeding and at least one article
 * (same as (Q5a)).
*/

SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name
}
```

Figure 7.1: SP^{2}Bench 5b query

Additional future work for TripleT will be to investigate compression schemes to reduce the size of the index. Development of a working implementation that can answer basic SPARQL queries would be useful so that existing benchmarks like SP^{2}Bench [8] could be used to compare TripleT with other implementations.

Bibliography

[1] DBPEDIA. The DBpedia Knowledge Base. `http://wiki.dbpedia.org/About`, Accessed July 2009.

[2] FLETCHER, G. H. L., AND BECK, P. W. Scalable indexing of RDF graphs for efficient join processing. In *Proceedings of CIKM* (2009), ACM.

[3] JENA. Jena a Semantic Web framework for Java. `http://jena.sourceforge.net/`, Accessed July 2009.

[4] MISHRA, P., AND EICH, M. H. Join processing in relational databases. *ACM Comput. Surv. 24*, 1 (1992), 63–113.

[5] NEUMANN, T., AND WEIKUM, G. RDF-3X: a RISC-style engine for RDF. *PVLDB 1*, 1 (2008), 647–659.

[6] Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. SP^{2}Bench: A SPARQL performance benchmark. In *ICDE* (2009), pp. 222–233.

[7] Sesame. RDF schema querying and storage. `www.openrdf.org/`, Accessed July 2009.

[8] SP^{2}Bench. The SP^{2}Bench SPARQL performance benchmark. `http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B`, Accessed July 2009.

[9] Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., and Reynolds, D. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW* (2008), pp. 595–604.

[10] UniProt. The UniProt Knowledgebase. `http://www.uniprot.org/help/uniprotkb`, Accessed July 2009.

[11] Virtuoso. Virtuoso RDF. `http://www.openlinksw.com/dataspace/dav/wiki/Main/VOSRDF`, Accessed July 2009.

[12] W3C. RDF primer. `http://www.w3.org/TR/rdf-primer/`, Accessed July 2009.

[13] W3C. SPARQL query language for RDF. `http://www.w3.org/TR/sparql/`, Accessed July 2009.