Growth of RDF datasets presents a challenge for the performance of query evaluation. By its nature even relatively simple RDF queries (e.g. “find all authors who have written at least one crime novel”) often already involve a large number of joins. Join evaluation poses a significant performance challenge on all current state-of-the-art RDF engines, which often fail to deliver interactive response times when faced with real queries on real-world datasets with hundreds of millions or even billions of triples. The Three-way Triple Tree (TripleT) is a proposal for a secondary-memory RDF store and a novel value-based, role-free indexing technique. It is a current state-of-the-art for join processing. Query optimization on TripleT has not been systematically studied up to this point. Using TripleT, we investigate how the use of (i) heuristics and (ii) external information (e.g. dataset statistics) can contribute towards a more intelligent way of generating query plans, which we evaluate on our implementation of the TripleT engine using real-world datasets. We propose a generic framework for query optimization, and show that this framework, combined with key heuristics rules, consistently produces efficient query evaluation plans. We also discuss the non-trivial problems posed by the use of statistics, and show how current approaches offer only minimal added value.
In the spring of 2012, right before that year’s summer break, I first approached dr. George Fletcher to discuss a few possible ideas for a graduation project. The reason I did so is because at the time dr. Fletcher was responsible for teaching the most interesting (and fun!) course I have attended during my years at Eindhoven University of Technology, and I was of course interested in doing more work along the same lines. A few months after this first meeting I found myself reading papers and writing code, designing algorithms and implementing a database engine. Both the science and the engineering parts that make up Computer Science and Engineering were well represented in my project, which for me made it the perfect way to finish my master’s degree.

My project was executed internally at Eindhoven University of Technology, within the Department of Mathematics and Computer Science, Information Systems Section, Databases and Hypermedia Group. First of all I would like to thank dr. George Fletcher, for bringing this highly interesting area of research to my attention, and for guiding and advising me during the course of my project. I would also like to thank prof. dr. Paul de Bra and dr. Herman Haverkort, for serving on my assessment committee and for providing me with a healthy dose of feedback and questions about my work.

Special mention goes out first of all to my parents for supporting me in my pursuit of a master’s degree after I had finished my bachelor’s, and secondly to the members of “Den Faalhazen” for making my time at university a highly enjoyable one.

Bart Wolff
Helmond, February 28, 2013
# Contents

1 Introduction 1

2 Background 3
   2.1 RDF 3
      2.1.1 Definition 3
      2.1.2 Serialization formats 5
      2.1.3 Queries 5
   2.2 RDF stores 7
      2.2.1 TripleT 8
   2.3 Berkeley DB 9
   2.4 Related work 9

3 On the use of external information for BGP query plan optimization 11
   3.1 Motivation 11
   3.2 Our requirements 13
   3.3 Current work 13
   3.4 Intermediate results 14
   3.5 Our solutions 14
      3.5.1 Stored statistics 14
      3.5.2 Output size estimation 15
   3.6 Further work 17
   3.7 Chapter summary 17

4 A TripleT database engine 19
   4.1 Overview 19
   4.2 System architecture 21
      4.2.1 Physical aspects 21
      4.2.2 Internal operations 27
      4.2.3 Memory management 37
   4.3 Implementation details 38
   4.4 Chapter summary 38

5 Query optimization over the TripleT engine 39
5.1 Atom collapses .................................................. 39
  5.1.1 Examples .................................................. 40
  5.1.2 Computing the atom collapse ........................... 42
5.2 Join graphs ..................................................... 42
  5.2.1 Examples .................................................. 42
  5.2.2 Computing the join graph ................................. 45
5.3 Query plans ..................................................... 45
  5.3.1 Examples .................................................. 46
  5.3.2 Computing the query plan ................................. 48
5.4 Rules .......................................................... 49
  5.4.1 A note on statistics ....................................... 50
  5.4.2 Decision point 1: Selecting a seed node ............... 50
  5.4.3 Decision point 2: Selecting a join edge ................ 51
  5.4.4 Decision point 3: Selecting a join type ............... 52
  5.4.5 Decision point 4: Selecting a scan operator .......... 53
5.5 Example query ................................................ 53
5.6 Chapter summary ............................................... 55

6 Experiment setup ................................................. 57
  6.1 Goals ........................................................ 57
    6.1.1 The value of rules ..................................... 57
    6.1.2 The value of statistics ................................ 58
    6.1.3 Different datasets .................................... 58
    6.1.4 Different queries ..................................... 58
  6.2 Setup ......................................................... 59
    6.2.1 Variables ............................................... 59
    6.2.2 Measurements .......................................... 59
    6.2.3 Environment ............................................ 60
  6.3 Datasets ....................................................... 60
    6.3.1 DBpedia ............................................... 61
    6.3.2 SP²Bench ............................................... 61
    6.3.3 UniProt ................................................ 62
  6.4 Queries ....................................................... 62
  6.5 Chapter summary ............................................... 63

7 Empirical results ................................................ 65
  7.1 Heuristics versus statistics ................................. 65
  7.2 The value of heuristics rules ............................... 70
  7.3 Rule ordering ............................................... 75
  7.4 Rule subsets ................................................ 79
  7.5 Merging versus hashing ..................................... 82
  7.6 Can we beat an ape? ......................................... 84
  7.7 Discussion ................................................... 85
  7.8 Chapter summary ............................................... 86

8 Conclusion ....................................................... 87
  8.1 Future work ................................................... 87

A Queries .......................................................... 89
  A.1 DBpedia ....................................................... 89
The goal of the Semantic Web and linked-data vision is to create a so-called “web of data”: an infrastructure for machine-readable semantics for data on the web [14, 17]. Still a work in progress as well as an active area of research, the semantic web aims to make data from a wide variety of sources available under a single unified standard, allowing for this data to be shared across different domains. Current W3C recommendations for the various technologies and standards supporting the Semantic Web stack include the Resource Description Framework (RDF) for description of data, RDFS and OWL for description of schemas and ontologies, and SPARQL for querying of RDF data.

As adoption of the Semantic Web and its related technologies grows, RDF and its various underlying data stores have to be able to deal with increasingly large datasets. This poses a scalability problem, both for storage and indexing, as well as for query evaluation. By its triple-centric nature, even the most basic RDF queries already involve a large number of (self-)joins, which pose a significant performance challenge on all current state-of-the-art RDF database engines. At present, real-world RDF datasets can involve hundreds of millions or even billions of triples, reaching the point at which current RDF engines have difficulty offering interactive query response times.

When compared to relational database technology, RDF stores are still a relatively new concept. And although a large amount of work has already been done in this area, much also remains in what is still considered an open area of research. A number of RDF stores exist, one of them being TripleT [9], which features a value-based, role-free indexing scheme, which is unique among the current state-of-the-art. In this thesis, we focus on the topic of query optimization on TripleT, which has not been systematically studied up to this point. More specifically:

*We investigate how the use of (i) heuristics and (ii) external information (e.g. dataset statistics) can contribute towards a more intelligent way of generating query plans, minimizing query execution time over the TripleT RDF store.*

These query plans are tailored to (and evaluated on) our implementation of the TripleT store, using both synthetic and real-world datasets. We focus on a comparison of TripleT against itself: our aim is to investigate the effectiveness of various parts of the heuristics that influence our query plan generator; we are not looking to directly compare against (or compete with) existing open-source or commercial RDF stores.

A number of novel contributions are made in this thesis:

- First, we present a secondary-memory implementation of the TripleT RDF store and index, which includes its own physical language for describing query plans.
Next, we propose a framework for query optimization in the form of a generic, rule-based algorithm used for generating query plans for given Basic Graph Pattern queries.

We discuss our implementation of this optimization framework on top of our TripleT store.

We describe a number of rules for use by our optimization framework.

Lastly, we present an extensive empirical study into the effectiveness of our rules in generating optimized query plans.

We show how our optimization framework, together with a few key heuristics rules, is able to consistently produce efficient query plans for a wide variety of different queries and different datasets. We will also see how the use of statistics poses a non-trivial problem, and how our current statistical solutions provide only minimal added value to the query plan generation process.

The remainder of this thesis is structured as follows: Chapter 2 discusses the preliminaries for our work, in Chapter 3 we discuss the use of statistics for query optimization, in Chapter 4 we describe in detail the TripleT store and our implementation thereof, in Chapter 5 we dive into our various query optimization techniques, Chapter 6 describes the experiments we have performed, Chapter 7 follows with a detailed discussion of their outcome, and Chapter 8 concludes with a summary of the work we have done and the results that have been obtained, as well as a few pointers for future work.
In this chapter we discuss the preliminaries for our work, namely the Resource Description Framework (RDF) and the TripleT RDF store. The Berkeley DB database system, used in a part of our work, is also briefly discussed. We conclude with an overview of related work.

2.1 RDF

The Resource Description Framework [24], from here on RDF, is a W3C recommendation for a language aimed at describing metadata about resources available on the Web. Originally introduced in 1998, the current version of RDF was standardized by the W3C in February of 2004. RDF is the result of one of the two original goals set by the Semantic Web program: to develop a language for describing metadata in a distributed, extensible setting [13].

Although standards exist for the description of RDF vocabularies (also known as ontologies) [5, 11], use of RDF does not require presence of an explicit schema, making it a flexible and highly extendable language. This kind of flexibility has proven key in RDF’s role as the lingua franca of the Semantic Web, and has allowed people and groups from all over the world to contribute, share, and use data belonging to a wide variety of domains.

Today, RDF is used in a number of large-scale datasets. Projects such as DBpedia and Freebase aim to build vast knowledge bases on a wide breadth of data (à la Wikipedia). UniProt is an example of a more specialized dataset, providing the scientific community with information on protein sequences. Further uses vary from examples such as the DBLP bibliography database, the United States census, and programming schedules of the BBC.

2.1.1 Definition

An RDF dataset consists entirely of an unordered set of triples: \((\text{subject}, \text{predicate}, \text{object})\) 3-tuples that describe either relationships between entities, or properties of entities, depending on the value of the \textit{object} field. Consider the following informal example:

\begin{align*}
\text{(Chell, worksAt, ApertureLabs)} \\
\text{(Chell, hasOccupation, "Test Subject")}
\end{align*}

\footnote{World Wide Web Consortium (http://www.w3.org/)}
(ApertureLabs, hasSlogan, "A Trusted Friend in Science")
(Caroline, worksAt, ApertureLabs)
(Caroline, dislikes, Chell)

Here, the “Chell” entity has both a relationship to another entity, and to a property: she works at “Aperture Laboratories” (itself another entity in the dataset), and has the occupation “Test Subject” (a literal value). In order to distinguish between entities and literal values, entities are generally written as URIs, where the URI may contain additional information about the domain or namespace of a given entity; conversely, literal values are generally written as plain text strings. It is important to note that although entities can occur in any position in an RDF triple, literal values can only occur in the object position.

A natural way of thinking about an RDF dataset is as a graph, where the subject and object parts of each triple correspond to nodes and the predicate parts correspond to the directed, labeled edges between them. The example above is visualized as a graph in Figure 2.1. We also give a formal definition of an RDF dataset in Definitions 1 and 2.

**Definition 1** Let $\mathbb{U}$ be a set of URIs and let $\mathbb{L}$ be a set of literals, such that $\mathbb{U} \cap \mathbb{L} = \emptyset$. Then we define an RDF triple as an element $(s, p, o) \in \mathbb{U} \times \mathbb{U} \times \mathbb{U} \cup \mathbb{L}$.

**Definition 2** We define an RDF dataset (or, alternatively, RDF graph), denoted $\mathbb{T}$, as a set of $n \geq 1$ RDF triples: $\mathbb{T} = \{t_1, t_2, \cdots, t_n\}$.

Observe that predicates, which live in the set of URIs, are themselves also entities in the dataset, and can therefor also occur in the subject or object positions of a triple. Building on the above example, we might imagine the following:

(dislikes, isTypeOf, relationship)
(hates, isTypeOf, relationship)
(hates, implies, dislikes)

Here, the “dislikes” entity, which was previously used as a predicate, is used as both the subject and the object in a number of triples. In this way, it is possible for an RDF graph to contain metadata about its own make-up, essentially making it self-describing.
2.1.2 Serialization formats

Although numerous formats exist for representing RDF datasets, for the sake of interoperability generally either one of two common formats are used that have been recommended by the W3C. Historically, RDF was originally represented in XML [4], though later a more human-readable format was developed by the name of Notation3 (N3) [3].

Consider a subset of our running example, annotated with namespaces:

(ex:Chell, ex:worksAt, ex:ApertureLabs)
(ex:Chell, ex:hasOccupation, "Test Subject")
(ex:Caroline, ex:dislikes, ex:Chell)

In XML, this dataset would be represented as follows:

```xml
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:ex="http://example.org/">
   <rdf:Description rdf:about="http://example.org/Chell">
     <ex:worksAt>http://example.org/ApertureLabs</ex:worksAt>
     <ex:hasOccupation>Test Subject</ex:hasOccupation>
   </rdf:Description>
   <rdf:Description rdf:about="http://example.org/Caroline">
     <ex:dislikes>http://example.org/Chell</ex:dislikes>
   </rdf:Description>
</rdf:RDF>
```

In Notation3, this dataset would be represented as follows:

```not3
@prefix ex: <http://example.org/>.

<http://example.org/Chell>
  ex:worksAt http://example.org/ApertureLabs;
  ex:hasOccupation "Test Subject".

<http://example.org/Caroline>
  ex:dislikes http://example.org/Chell.
```

Both formats utilize some space-saving techniques: the subject-part of the triples is only present once, with the predicate and object parts being nested under them. The use of namespaces also removes a lot of repetitive data, and also contributes to human readability of the document. It should be obvious from this example that compared to Notation3, XML is much more verbose about its document structure, and less readable. On the other hand, XML itself is an extremely well-known and widely accepted standard, with parsers being readily available (often through the standard library) in virtually every programming language used today. From a purely interoperability standpoint, this might make it preferable to the more compact Notation3.

2.1.3 Queries

Since its introduction, RDF has gained significant popularity within the Semantic Web community, and has been used on numerous occasions to build large-scale datasets. Of course, all that data is
useless without a way of querying it. As of January 2008, the W3C recommendation for querying RDF datasets has been SPARQL [30].

Inspired by SQL, SPARQL has a familiar syntax. Consider the following example, where we ask for the slogan of the company that both Chell and Caroline work for:

```
PREFIX ex: <http://example.org/>
SELECT ?slogan
WHERE {
  ex:Chell ex:worksAt ?x.
ex:Caroline ex:worksAt ?x.
  ?x ex:hasSlogan ?slogan.
}
```

Although the SPARQL language contains many advanced features which are outside the scope of this investigation, at its core lies a relatively simple concept: Basic Graph Patterns (BGPs) [14]. A BGP is a conjunction of Simple Access Patterns (SAPs), where each SAP is a triple consisting of some combination of fixed values (atoms) and unfixed values (variables). A formal definition of a BGP is provided through Definitions 3 and 4.

**Definition 3** Let $\mathbb{A} = \mathbb{U} \cup \mathbb{L}$ be a set of atoms, and let $\mathbb{V}$ be a set of variables, such that $\mathbb{A} \cap \mathbb{V} = \emptyset$. Then we define a Simple Access Pattern as a triple $S = (s, p, o) \in (\mathbb{A} \cup \mathbb{V}) \times (\mathbb{A} \cup \mathbb{V}) \times (\mathbb{A} \cup \mathbb{V})$.

**Definition 4** We define a Basic Graph Pattern as a conjunction of $n \geq 1$ Simple Access Patterns:

$$P = S_1 \land S_2 \land \cdots \land S_n.$$ 

The result of querying an RDF dataset with some BGP is a set of bindings. Each binding consists of a set of (variable, atom) pairs, where there is exactly one such pair for each variable existing in the BGP. Each binding in the result set is such that when the variables from the BGP are replaced by the corresponding atom values from this binding, the resulting set of triples occur in the original RDF dataset. We formally define querying through Definitions 5, 6, and 7.

**Definition 5** We define a binding as a non-empty set of (variable, atom) pairs, denoted $B = \{(v_1, a_1), (v_2, a_2), \ldots, (v_n, a_n)\}$ with $n \geq 1$, such that $(v, a) \in B \Rightarrow \lnot \exists (v', a') \in B : (a \neq a' \land v = v')$.

**Definition 6** Let $P$ be a BGP, and let $B$ be a binding. Then we define the application of $P$ to $B$, denoted $P(B)$, as the set of triples from $P$ such that for each triple $t \in P$ any variable $v \in t$ has been replaced with the atom a belonging to the pair $(v, a) \in B$.

**Definition 7** Let $P$ be a BGP and let $\mathbb{T}$ be an RDF dataset. Then the result of querying $\mathbb{T}$ with $P$, denoted $P(\mathbb{T})$, is the largest possible set of bindings $\mathbb{B}$ such that $\forall B \in \mathbb{B}. (P(B) \subseteq \mathbb{T})$.

Computing query results $P(\mathbb{T})$ for BGP $P = S_1 \land S_2 \land \cdots \land S_n$ means we will need to compute the natural join $S_1(\mathbb{T}) \Join S_2(\mathbb{T}) \Join \cdots \Join S_n(\mathbb{T})$. Hence, we can also write a BGP as $P = S_1 \Join S_2 \Join \cdots \Join S_n$.

Consider the following BGP query on our running example:

```
(?x, worksAt, ?y)
```

Semantically, we ask for a list of people and their place of employment. It will produce the following results:

```
{(?x : Chell), (?y : ApertureLabs)}
{(?x : Caroline), (?y : ApertureLabs)}
```
Next, consider the following BGP query:

\{(Chell, worksAt, ?x), (?x, hasSlogan, ?y)\}

Here, we ask for the workplace of Chell, as well as that company’s slogan. It will produce the following results:

\{(?x : ApertureLabs), (?y : "A Trusted Friend in Science")\}

In this way, the Basic Graph Pattern offers a simple, yet extremely powerful and well-suited way of querying RDF datasets. For the remainder of this work, when we talk about queries on RDF data, this will be a synonym for BGPs; similarly, query results are always sets of bindings.

## 2.2 RDF stores

To cope with large amounts of RDF data and the queries that are executed on them, efficient storage and indexing schemes are required. A recent survey in [22] distinguishes between three different perspectives on RDF storage:

- The **relational perspective** considers RDF data to be just another type of relational data, and leverages existing storage and indexing techniques originally developed for relational database systems.

- The **entity perspective** treats resources in an RDF dataset as “entities” associated with a number of \(\text{attribute, value}\) pairs, and aims to support queries that retrieve entities based on these pairs. This perspective originates from the information retrieval community.

- The **graph-based perspective** views an RDF dataset as a classical graph, where the \text{subject} and \text{object} parts of each triple correspond to nodes and the \text{predicate} parts correspond to the directed, labeled edges between them. It aims to support graph navigation and answering of graph-theoretic queries. This perspective originates from research on semi-structured- and graph databases.

Within the relational perspective, one can distinguish between **vertical** and **horizontal** representations of the data that is being stored. The primary difference between the two is that a horizontal representation (conceptually) stores all data in a single table, where the rows correspond to the \text{subject} parts of the triples in the dataset, the columns correspond to the \text{predicate} parts, and the values in the table itself correspond to the \text{object} values for the triples; conversely, a vertical representation (conceptually) stores all data in a single table with columns \(\text{subject, predicate, object}\), each row representing a single triple.

Several techniques exist for storing and indexing data in a vertical representation. While it is possible to store the data in a single physical triple table on which multiple indexes are built (an unclustered index approach), a more common approach is to have several indexes over multiple distinct physical tables which are each sorted in different ways (a clustered index approach). Although less efficient in terms of storage space and data locality, a clustered index approach generally allows for significantly improved performance when processing queries. It is used in a number of existing RDF stores:

- Virtuoso [8], RDF-3X [26], and YARS [16] are examples of systems using the Multiple Access Patterns (MAP) approach, which stores and indexes all triples sorted in each of the different orderings made possible by permuting the \text{subject, predicate, and object} positions: \text{SPO, SOP, PSO, POS, OSP, OPS}. 

7
• Similarly, HexaStore [34] uses the HexTree approach, which stores and indexes two roles at the
time, again in all possible permutations: SP, SO, PS, PO, OS, OP.

A hybrid clustered/unclustered approach is proposed in [9]. The Three-way Triple Tree (TripleT)
RDF store aims to increase data locality by having only a single index, built over the atoms in the
dataset, while also storing the triples in multiple orderings. Three buckets are used to store triples,
in orderings SOP, PSO, and OSP. Looking up an atom \( a \) in the index returns a set of pointers to
ranges in those buckets that contain \( a \) in subject, predicate, and object positions, respectively. We
will discuss TripleT in more detail in the next section.

2.2.1 TripleT

In our work, we use TripleT as the underlying RDF storage scheme. TripleT is a current state-of-
the-art for physical representation for join evaluation. It is a secondary-memory indexing technique
designed specifically for efficient processing of joins on RDF datasets.

The TripleT storage system consists of a single index, and a number of buckets containing the actual
data. The index itself is built over the atoms (i.e. the individual \( s, p \) or \( o \) parts that make up an
\((s, p, o)\) triple) in the dataset, while the buckets contain the actual data sorted in several different
ways. There are three different buckets: an S-bucket sorted in SOP order, a P-bucket sorted in PSO
order, and an O-bucket sorted in OSP order. For each atom present in the index, the payload contains
information pointing to a section of each bucket where triples are stored which contain that atom in
their \( s, p \), or \( o \) position, respectively. To save space, the S-bucket itself does not contain the \( s \) values
of its triples as this information is already present implicitly as part of the index. The same holds for
the P- and O-buckets.

As an example, recall our toy dataset:

(Chell, worksAt, ApertureLabs)
(Chell, hasOccupation, "Test Subject")
(ApertureLabs, hasSlogan, "A Trusted Friend in Science")
(Caroline, worksAt, ApertureLabs)
(Caroline, dislikes, Chell)

If we would enter this dataset into a TripleT store, then our index would contain one entry for each
of the 9 unique atoms in the dataset, and the S-bucket would look like the following:

<table>
<thead>
<tr>
<th>S-bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasSlogen</td>
</tr>
<tr>
<td>worksAt</td>
</tr>
<tr>
<td>dislikes</td>
</tr>
<tr>
<td>hasOccupation</td>
</tr>
<tr>
<td>worksAt</td>
</tr>
</tbody>
</table>

When we would look up the value for “Chell” in the index, we would find a pointer to the \((\text{worksAt}, \text{ApertureLabs})\) and \((\text{hasOccupation}, "\text{Test Subject}"
\)) tuples in this bucket. In this way, data can be retrieved.

TripleT provides a conceptually simple yet powerful mechanism for indexing RDF datasets, and
features a strong data locality that greatly reduces storage costs. We discuss TripleT, as well as our
Berkeley DB

Berkeley DB\(^2\) is an embedded database system written in C and with APIs available in a wide variety of programming languages. It is an open source system and as of today is one of the most widely used database systems in the world [6, Chapter 4].

Contrary to the more advanced concept of a relational database, Berkeley DB simply exposes a key/value store for binary data. Internally, it can use either B+ trees or hash tables as a data structure for storage. Being an embedded database, the system is lightweight, portable, and easy to distribute along with software packages that use it. It does not require any installation or configuration before it can be used on a system.

Although in development since the early 90s and packed with a range of advanced features such as concurrency, transactions, and high-availability replicas, in our work we use Berkeley DB as little more than a convenient secondary-memory key/value store, as part of our implementation of TripleT.

We use Berkeley DB for both B+ trees and hash tables in parts of our work. We have used Berkeley DB version 5.3.21, built for Windows 7 x32 by Visual Studio 2010.

Related work

Much work has been done on the subjects of RDF storage, indexing, and querying. We will provide a brief overview of related background work, as well as work that is relevant to ours.

The basics of the theory of RDF databases are discussed in [2], including a formal definition of the RDF model and a study of the query language SPARQL. An overview in [13] provides an introductory discussion on the problem of modeling the data in the Semantic Web, and aims to provide a large amount of pointers to current, related literature on this subject. A more theoretical discussion of Semantic Web databases is provided in [14], which includes a formal discussion of the RDF data model, as well as the abstract notions of answering queries on that model.

Numerous RDF stores and indexes have been developed over the years. Notable examples include Virtuoso [8], RDF-3X [26], YARS [16], HexaStore [34], and Tridex [20]. A recent survey in [22] provides an up-to-date overview of current work on the storing and indexing of large-scale RDF datasets. An interesting discussion and comparison between row-stores and column-stores is given in [1].

The topic of database query processing and optimization is almost as old as the topic of database systems themselves, and subsequently much prior work exists in this field. It is outside of the scope of this thesis to provide a complete summary, though we will highlight a few selected papers. In [19] an abstract overview is given on the topic of query optimizers and in [10] a comprehensive survey is provided discussing the design and implementation of query execution facilities, although it should be mentioned that with both papers the focus is on relational database systems; RDF was not yet introduced at the time that they were published. On the topic of RDF, [27] addresses some of the scalability problems that arise when processing join queries on RDF datasets containing hundreds of millions of triples. Optimizations for Basic Graph Pattern queries using statistics for

selectivity estimation are discussed in [32], while in [33] a number of heuristics for static analysis and optimization of BGPs are introduced. Work in [25] and [18] describes several techniques for selectivity and cardinality estimation using precomputed information over RDF datasets.

Some prior work also exists featuring TripleT. The performance of different join algorithms on the TripleT index is investigated in [21], where the impact of some basic statistics for query answering is also discussed. A slightly modified version of TripleT is used in [15] as physical representation of data used for a proposed RDFS entailment algorithm, where it is concluded to be decent candidate for RDFS data storage.

Regarding SPARQL testing and benchmarking, [31] describes the SP²Bench SPARQL benchmark with synthetically generated data, while [7] provides an extensive study of such data generated for benchmarks and compares its characteristics to those of real-world RDF datasets.
An interesting challenge that emerged partway through our investigation is found in the use of external information (generally in the form of pre-computed database statistics) for the purpose of query plan optimization. It appears that up to this point not a lot of work has been done in this area with regards to native RDF data management. In this chapter, we aim to provide an overview of the challenges we have encountered, the current solutions (or lack of thereof) provided by the state-of-the-art, our own solutions, as well as any remaining problems worth investigating further.

3.1 Motivation

Although work in [33] has shown that a set of well-defined heuristics is able to produce query plans that are able to compete on execution time with statistics-heavy systems such as RDF-3X, heuristics themselves do suffer from a number of inherent flaws.

When given a BGP and tasked with generating a suitable query plan, heuristics make their decisions based on the structure of the BGP while ignoring the contents (i.e. the actual values of any atoms present). This leads to a number of shortcomings. One can easily imagine the case where two BGPs, which are structurally identical but have different contents in terms of the values of their atoms, can have wildly different optimal query plans. Secondly, consider the case where a number of SAPs belonging to a BGP have the same structure but different contents. Deciding in which order to join those SAPs can have a major impact on overall query execution time. Unfortunately, heuristics alone cannot provide the answer in either of these cases.

3.2 Our requirements

The obvious solution to the shortcomings of heuristics is to fill in their blind spots with the right kind of information that will allow decision making based not just on the structure of a BGP, but on its contents as well. In those cases that it is present this information generally takes the shape of database statistics (e.g. histograms, join tables, etc.).

In our case, with our queries consisting of BGPs, we have identified several bits of information that, ideally, we would like to have available when computing an optimized query plan for a BGP:
• For any atom $a$ in position $p \in \{S, P, O\}$, we would like to know the number of triples in the dataset that have $a$ in position $p$.

• For any SAP $(s, p, o)$, with any combination of atoms and variables, we would like to know the number of triples in the dataset matching it.

• For any join between SAPs $P_1 \bowtie P_2$, where $P_1$ and $P_2$ can have any combination of atoms and variables, we would like to know the number of triples this join produces.

• For any join between sets of bindings $B_1 \bowtie B_2$, where $B_1$ and $B_2$ can have any number of variables, we would like to know the number of triples this join produces.

The need for this particular set of information is motivated by the kinds of decisions that we need to make when generating a query plan (for details, please refer to Chapter 5). The primary challenge that we have identified is estimating the output sizes of intermediate results that are produced above the leaves and below the root of a query plan. Figure 3.1 helps to illustrate this point.

Here, we see a partial query plan. To complete the plan, we need to join the results from the merge-join operation with the results produced by the scans for the SAPs $(?x, a, b)$ and $(?x, c, d)$. We can do this one of three different ways:

1. We could join the results from the $(?x, a, b)$ scan with the results produced by the existing merge-join operator, and then join the results this produces with the results from the $(?x, c, d)$ scan.

2. We could do the same thing but reverse the order in which we join the $(?x, a, b)$ and $(?x, c, d)$ scan results.

3. We could first join the results from the $(?x, a, b)$ and $(?x, c, d)$ scans together, and then join the results this produces with the results from the existing merge-join operator.

To make a decision, we would like to know which option produces the smallest intermediate result set, but in the case of the first two options there is no clear way this can be accurately computed. It is at this point that we encounter a gap in the current state-of-the-art.
3.3 Current work

The first relevant work we know of has been done in [32]. Here, a number of statistics are described to aid in selectivity estimation, both for single triple patterns as well as for join patterns. Statistics introduced include a histogram containing the number of triples containing some atom \( a \) in positions \( S, P, \) and \( O \). A join table is also described, containing cardinality of joins \((?x, p_1, ?y) \bowtie (?x, p_2, ?z), (\forall x, p_1, ?y) \bowtie (?z, p_2, ?x), (\forall x, p_1, ?y) \bowtie (?y, p_2, ?z), (\forall x, p_1, ?y) \bowtie (?z, p_2, ?y)\), for any predicate values \( p_1 \) and \( p_2 \) in the dataset. This join table can then be combined with selectivity values of atoms in \( S \)- or \( O \)-positions to estimate output cardinality of any join between SAPs that do not feature variables in the \( P \)-position (which is uncommon), making it a powerful help at a small storage cost.

A well known RDF engine, RDF-3X, relies heavily on statistics for plan generation, with much of the relevant work on selectivity and cardinality estimation being discussed in [27, 28, 29], where several solutions are proposed using histograms.

Specific to TripleT, [21] investigates the impact of some rudimentary statistics on query answering using the TripleT index.

In [25], pre-computed data takes the shape of so-called characteristic sets, which are then used for accurate cardinality estimation of star-joins, although they are limited to subject – subject-joins. An alternative solution for star-joins is proposed in [18] using Bayesian networks. Chain patterns, another common query pattern, are also discussed in [18], where a solution is proposed using histograms. In [12], cardinality estimation of chain patterns is proposed using Dijkstra’s algorithm.

Finally, [23] discusses tree-based solutions for subgraph frequency estimation, which, although not directly aimed at RDF graphs, uses a data model that captures several semantic data models, RDF included.

3.4 Intermediate results

When computing an optimized query plan, in the ideal case we would always like to know what kind of result sizes are produced when performing a join between two arbitrary operators. And while current work provides excellent solutions for estimating results of joins between two SAPs, we find it lacking in accurate estimators for those joins above a plan’s leaf level that take place between results produced by other join operators.

As an example, consider the following BGP \( P \):

\[(a, b, ?x)_1 \land (?x, c, ?y)_2 \land (?y, d, ?x)_3 \land (?y, e, ?x)_4.\]

We might start our query plan generation by producing intermediate result \( R \):

\[(a, b, ?x)_1 \bowtie (?x, c, ?y)_2 \rightarrow R.\]

Our next step is then to join \( R \) with either \( P_3 \) or \( P_4 \). To do so, we would like to know which option produces the smallest intermediate result, but there is no clear way which provides us with an accurate estimate because set \( R \) lacks the structural information that we would otherwise use in our computation.

In the general case, the problem involves a series of intermediate result sets:

\[R_1, R_2, \cdots, R_n.\]
These sets are sets of bindings that need to be joined with each other, the challenge being finding the optimal order in which this is to be done. Joining two SAPs, we know the positions of the join variables within the SAPs, and we know the context (the SAPs themselves); when joining two sets of intermediate results we have none of this information, and the lack of this structural information is what prevents us from making an accurate estimate.

Current solutions from the state-of-the-art do not address this problem as they assume the presence of structural information, either when estimating result size of a join between two SAPs or when estimating result size of larger query patterns.

3.5 Our solutions

In our implementation of the TripleT store, we have chosen to keep a number of different dataset statistics which will allow us to perform output size estimation. Using these statistics, we compute our estimates in a number of ways, dependent on the amount of structural information that is available. In this section, we discuss in detail what statistics we keep and how we compute output size estimations.

3.5.1 Stored statistics

The first set of statistics we store consists of a number of global summarizing statistics, the storage size of which is independent from the dataset size. We store:

- The total number of records (triples) in the dataset.
- The total number of unique atoms in the dataset.
- The total number of unique atoms in the S-position of the triples in the dataset.
- The total number of unique atoms in the P-position of the triples in the dataset.
- The total number of unique atoms in the O-position of the triples in the dataset.

Next, we store a set of detailed statistics, the expected size of which grows quadratically with the number of tuples in the dataset. We store:

- For each atom \( a \), the number of triples in the dataset matching \((a, ?, x, ?y), (?, x, a, ?y), \text{ and } (?, x, ?, y, a)\).
- For each pair of atoms \( a_1 \) and \( a_2 \), the number of triples in the dataset matching \((a_1, a_2, ?, x), (a_1, ?, x, a_2), (?, x, a_1, a_2), (a_2, a_1, ?, x), (a_2, ?x, a_1), \text{ and } (?, x, a_2, a_1)\).
- For each pair of atoms \( a_1 \) and \( a_2 \) where both \( a_1 \) and \( a_2 \) occur in the P-position of at least one triple in the dataset, the number of triples in the dataset matching \( (?, x, a_1, ?y) \land (?, x, a_2, ?z), (?, x, a_1, ?y) \land (?, z, a_2, ?x), (?, x, a_1, ?y) \land (?, y, a_2, ?z), \text{ and } (?, x, a_1, ?y) \land (?, z, a_2, ?y)\).

Note that in all of the above, we assume \(?x, ?y, \text{ and } ?z\) are independent variables. This implies we do not store any statistics for:

- The number of triples in the dataset matching an SAP featuring a single atom \( a \) in one position and a single variable \(?x\) in the remaining two positions.
- The number of triples matching \((_, a_1, _) \land (_, a_2, _)\) where the SAPs share more than one variable, or where an SAP has a single variable in both open positions.
These cases are merely additional variations on the same general idea. We do not consider any queries that contain patterns from the list above, as doing so would not provide us with additional insight relevant to our overall investigation. This choice is also made to save on computation- and storage costs, and is based on our observation of what does and does not frequently occur in real-world queries.

Lastly, we define a number of relevant statistical helper functions and notations which will be used in the next part of this section:

- Let $|T|$ denote the total number of triples in the dataset.
- Let $|P(T)|$ denote the number of triples matching BGP $P$.
- Let $\text{uni}(p)$ denote the number of unique atoms in the $p$-position of the triples in the dataset.
- Let $\text{num}(a,p)$ denote the number of triples in the dataset with atom $a$ in their $p$-position.
- Let $\text{num}(a_1,p_1,a_2,p_2)$ denote the number of triples in the dataset with atom $a_1$ in their $p_1$-position and atom $a_2$ in their $p_2$ position.
- Let $\text{sel}(a,p) = \frac{\text{num}(a,p)}{|T|}$ denote the selectivity of atom $a$ in position $p$.
- Let $\text{sel}(a_1,p_1,a_2,p_2) = \frac{\text{num}(a_1,p_1,a_2,p_2)}{|T|}$ denote the selectivity of atom $a_1$ in position $p_1$ and atom $a_2$ in position $p_2$.
- Let $\text{join}(a_1,p_1,a_2,p_2)$ denote the number of tuples produced by the natural join $(\_ , a_1, \_ ) \bowtie (\_ , a_2, \_ )$ where the join variable $?x$ is in the $p_1$-position of the left SAP and in the $p_2$-position of the right SAP, with the remaining positions being filled by variables unequal to $?x$ and to each other.

### 3.5.2 Output size estimation

During query plan computation, output size estimation can take place in a number of different cases. Ranked from trivial to difficult to estimate, they are:

1. Estimating the result size of a single SAP.
2. Estimating the result size of the natural join between two SAPs.
3. Estimating the result size of the natural join between an SAP, and the output produced by another join operation.
4. Estimating the result size of the natural join between two sets of output produced by two different join operations.

Estimating the result size of a single SAP can be done by a single lookup into the statistics database. For an SAP $S_1$ consisting of two variables $?x, ?y$ and a single atom $a$, we look up $\text{num}(a,p)$, where $p$ is the position of $a$ in $S_1$. We thus get:

$$\text{est}(S_1) = \text{num}(a,p).$$

In the case where $?x \neq ?y$ the estimation provided is exact; if $?x = ?y$ then the estimation is an upper bound. Exact statistics for the latter case are not kept due to the rarity of SAPs featuring one variable in two positions that we have observed in real-world queries. For an SAP $S_2$ consisting of a
single variable ?x and two atoms \(a_1, a_2\), we look up \(\text{num}(a_1, p_1, a_2, p_2)\), where \(p_1\) and \(p_2\) denote the positions of \(a_1\) and \(a_2\) in \(S_2\), respectively. Here, the estimation provided is always exact, and we get:

\[
est(S_2) = \text{num}(a_1, p_1, a_2, p_2).
\]

Estimating the result size of the natural join between two SAPs \(S_1 \bowtie S_2\) can be performed with varying levels of accuracy, depending of the make-up of the SAPs. In most cases we observed in real-world queries, both SAPs will feature at least an atom in their P-position and will share a single join variable. We will first consider this case, where \(a_1\) is the P-atom of \(S_1\), \(a_2\) is the P-atom of \(S_2\), \(p_1\) is the position of join variable \(?x\) in \(S_1\), and \(p_2\) is the position of join variable \(?x\) in \(S_2\). We first look up \(\text{join}(a_1, p_1, a_2, p_2)\) from the statistics database. Then, for each atom \(a \in S_1 \cup S_2\) in position \(p \in \{s, o\}\), we look up \(\text{sel}(a, p)\) and multiply our original guess with this value. We thus arrive at:

\[
est(S_1 \bowtie S_2) = \text{join}(a_1, p_1, a_2, p_2) \cdot \prod_{(a, p) \in A} \text{sel}(a, p)
\]

where

\[
A = \{(a, p) \mid a \in S_1 \cup S_2 \land \text{ position } p \text{ of } a \text{ in } S_1 \cup S_2 \text{ is such that } p \in \{s, o\}\}.
\]

If any of the two SAPs do not feature an atom in the P-position, our pre-computed join table can not help us and we need to fall back to less accurate estimation methods. We first compute the product of \(\nest(S_1)\) and \(\nest(S_2)\), where estimations for the individual SAPs are computed through methods described above. This provides us with the expected size of cartesian product \(S_1 \times S_2\), which serves as an upper bound for \(S_1 \bowtie S_2\). Next, for some SAP \(S\), let \(V(S, ?x) = \nest(S) \times \frac{\text{num}(p)}{|T|}\), where \(p\) denotes the position of \(?x\) in \(S\). We make the simplifying assumption that for every join variable \(?x\) shared between some SAPs \(S\) and \(S'\), \(V(S, ?x) \leq V(S', ?x)\) implies that if a binding pair \((?x, a)\) exists in some binding in \(S(T)\), then this binding pair also occurs in some binding in \(S'(T)\). Now consider some bindings \(B \in S(T)\) and \(B' \in S'(T)\), with some shared variable \(?x\), and \(V(S, ?x) \leq V(S', ?x)\). Then by assumption any binding pair \((?x, a)\) in \(B\) must also occur in some binding in \(S'(T)\). Hence, the chance that \((?x, a) \in B'\) equals \(\frac{1}{V(S', ?x)}\). In the general case, the chance that two bindings \(B \in S\) and \(B' \in S'\) share \((?x, a)\) equals \(\frac{1}{\max(V(S, ?x), V(S', ?x))}\). We can now make our estimation: we take the product of \(\nest(S_1)\) and \(\nest(S_2)\), which denotes the size of the cartesian product, and divide it by \(\max(V(S_1, ?x), V(S_2, ?x))\) for each variable \(?x\) shared by \(S_1\) and \(S_2\), assuming variables are independent. We thus get:

\[
est(S_1 \bowtie S_2) = \frac{\nest(S_1) \times \nest(S_2)}{\prod_{(?x, p_1, p_2) \in A} \max\left(\nest(S_1) \times \frac{\text{num}(p_1)}{|T|}, \nest(S_2) \times \frac{\text{num}(p_2)}{|T|}\right)}
\]

where

\[
A = \{(?x, p_1, p_2) \mid ?x \in S_1 \text{ at position } p_1 \land ?x \in S_2 \text{ at position } p_2\}.
\]
by SAP scans) required to compute \( V(S, ?x) \). We hence remove the structural information from the
computation above to arrive at:

\[
est(S_1 \bowtie S_2) = \frac{\est(S_1) \cdot \est(S_2)}{\prod_{?x \in A} \max(\est(S_1), \est(S_2))}
\]

where

\[
A = \{?x \mid ?x \in S_1 \land ?x \in S_2 \}.
\]

and \( S_1 \) and \( S_2 \) can be both results from SAP scans or join operations. A consequence of this method of
computation is that estimations for join operators further up from the leaves of a query plan are
based upon estimations for the nodes below them. Hence, the higher up we go in a query plan, the
less accurate output size estimations become.

### 3.6 Further work

Statistics for use in query plan computation and optimization represents an important part of our
investigation. However, as the scope of our investigation was intended to be restricted to applying
current statistical estimation techniques to the TripleT store, we were a little surprised when we
found that existing work in this specific area is rather limited.

The most obvious gap in the knowledge of the state-of-the-art that we have encountered is found at
intermediate result size estimation. In our literature survey we have not found any usable techniques
that would allow us to make more than rudimentary estimations in these cases. The single most
interesting subject for further work in this area we feel would hence be an investigation into the use
of external information for estimating intermediate result sizes produced by BGP queries.

Additionally, we feel another interesting avenue for further work in this area could lie in efficient ways
of computing and maintaining statistical data structures on the TripleT index. Again, the scope of
our investigation prevented us from looking into this ourselves.

### 3.7 Chapter summary

In this chapter we have talked about the use of statistical information in query optimization, particularly
for use in result size estimation. We have discussed prior work in this area, and found that surprisingly
little work exists with direct relevance to RDF data management. We have talked about our
requirements in this regard, and learned that although there are good statistical estimators that work
in some cases, there are also gaps in the current state-of-the-art. Especially intermediate result size
estimation, which takes place during construction of a query plan, has proven difficult to perform
due to the lack of structural information that is available at that point. Next, we have discussed the
solutions that we have come up, both in terms of how we compute our various result size estimations,
and in terms of the statistical information that we use to do so. We have concluded this chapter with
some brief pointers for further interesting work in this area.
A TripleT database engine

The work in our investigation is centered around the Three-way Triple Tree (TripleT) secondary-memory RDF store and index. TripleT was originally proposed in [9], and features as its primary novelty an index which is built over the individual atoms in a dataset, rather than over complete triple patterns. In this chapter, we briefly discuss the idea behind TripleT, and we discuss in detail our implementation of the TripleT database engine as it was used throughout our investigation.

4.1 Overview

Although we refer to [9] for a more in-depth discussion of TripleT, we will provide a brief overview of the idea which is proposed. When we look at how the data is stored, we first see that TripleT uses a number of buckets that store the actual triples in the dataset. Each bucket stores all the \((s, p, o)\) triples in the dataset, ordered on some permutation of \(\{s, p, o\}\). For instance, an SOP-bucket would store the triples sorted first on subject, then on object, and lastly on predicate. The possible bucket orderings are thus SPO, SOP, PSO, POS, OSP, and OPS, though in our implementation we limit ourselves to using SOP-, PSO-, and OSP-buckets only. The remaining permutations, SPO, POS, and OPS, are symmetrical and are not considered in our investigation.

As an example, recall the dataset from the background chapter:

(Chell, worksAt, ApertureLabs)
(Chell, hasOccupation, "Test Subject")
(ApertureLabs, hasSlogan, "A Trusted Friend in Science")
(Caroline, worksAt, ApertureLabs)
(Caroline, dislikes, Chell)

When stored in TripleT, the following buckets would be created:

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-bucket</td>
<td></td>
</tr>
<tr>
<td>hasSlogan</td>
<td>&quot;A Trusted Friend in Science&quot;</td>
</tr>
<tr>
<td>worksAt</td>
<td>ApertureLabs</td>
</tr>
<tr>
<td>dislikes</td>
<td>Chell</td>
</tr>
<tr>
<td>hasOccupation</td>
<td>&quot;Test Subject&quot;</td>
</tr>
<tr>
<td>worksAt</td>
<td>ApertureLabs</td>
</tr>
</tbody>
</table>
CHAPTER 4. A TRIPLET DATABASE ENGINE

P-bucket

<table>
<thead>
<tr>
<th>Caroline</th>
<th>Chell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chell</td>
<td>&quot;Test Subject&quot;</td>
</tr>
<tr>
<td>ApertureLabs</td>
<td>&quot;A Trusted Friend in Science&quot;</td>
</tr>
<tr>
<td>Caroline</td>
<td>ApertureLabs</td>
</tr>
<tr>
<td>Chell</td>
<td>ApertureLabs</td>
</tr>
</tbody>
</table>

O-bucket

| ApertureLabs | hasSlogan |
| Chell        | hasOccupation |
| Caroline     | worksAt |
| Chell        | worksAt |
| Caroline     | dislikes |

Of important note is that each bucket does not contain the triple part (s, p, or o) that corresponds to its primary sort order. The reason for this is that this information is implicitly present as part of the index and does not need to be stored here.

The index of TripleT is built over the atoms in the dataset and contains pointers to the buckets. There is one entry in the index for each unique atom in the dataset. This entry contains a number of pointers to triple ranges in each of the bucket files. For instance, the index entry for an atom a might contain a pointer to range $[x \cdots y]$ in the SOP-bucket, meaning that in this bucket, which is primarily sorted on subject, triples from position $x$ to position $y$ contain the value $a$ in their subject position. Similarly, the same entry might contain pointers to triple ranges in the PSO- and OSP-buckets that contain $a$ in the predicate- and object positions, respectively.

For our example dataset, the index would consist of 9 entries. When we would look up the value for “Chell” in the index, we would find:

- A pointer to the (worksAt, ApertureLabs) and (hasOccupation, "Test Subject") tuples in the SOP-bucket.
- A pointer to the (Caroline, dislikes) tuple in the OSP-bucket.

The partial information retrieved from the buckets is then combined with the information from the index in order to reconstruct the original triples:

(Chell, worksAt, ApertureLabs)
(Chell, hasOccupation, "Test Subject")
(Caroline, dislikes, Chell)

Hence, data can be retrieved and queries can be answered. To do so, data always needs to be read from one of the available buckets. The sort ordering of a bucket determines how suitable (with regards to I/O efficiency) it is for retrieving triples matching a given SAP. An SOP-bucket would be well suited for retrieving triples matching $(a, ?x, ?y)$, but would be inefficient at retrieving triples matching $(?x, a, ?y)$. For the latter case the bucket ordering implies the entire bucket needs to be read in order to find all possible matches, while for the former case the index entry for $a$ directly points to the range in the bucket where any matches must be contained.
4.2 System architecture

This section describes the global architecture and component structure of our implementation of the TripleT RDF store. It should be noted that our implementation is a possible implementation of the idea proposed in [9], and that it is strictly a research prototype limited to the execution of BGP queries and a (non-optimized) method for building the index.

4.2.1 Physical aspects

A single TripleT database is stored on disk across eight different physical files. An overview is provided in Figure 4.1. We discuss the individual components below.

![Figure 4.1: TripleT physical architecture](image)

### Dictionary

The dictionary is stored in two BerkelyDB hash databases and uses a flat binary file to keep track of the auto-incrementing integer value used to assign to new entries. The dictionary is used to translate between “friendly” representations and internal representations of atoms. It consists of three individual, physical files, listed in Table 4.1.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_name.dict.str.dat</td>
<td>A BerkeleyDB hash database containing a string-to-integer dictionary used to translate the original data values (strings) to an internally used representation (integers).</td>
</tr>
<tr>
<td>db_name.dict.int.dat</td>
<td>A BerkeleyDB hash database containing an integer-to-string dictionary used to translate from internal data representations (integers) to friendly names (strings) containing the original data values.</td>
</tr>
<tr>
<td>db_name.dict.dat</td>
<td>A flat binary file containing the auto-incrementing next internal representation value to use.</td>
</tr>
</tbody>
</table>

Table 4.1: TripleT dictionary physical files
The file containing the auto-incrementing identifier is 8 bytes large and only contains a single 64-bit integer in binary format. The BerkeleyDB database containing the string-to-integer dictionary is used to translate from “friendly” atom representations to their internal counterparts. The key for each entry consists of a 1-dimensional byte-array that is obtained by taking the UTF-8 byte encoding of a given string. The payload for each entry consists of a 1-dimensional byte-array representing a single 64-bit signed integer.

**Example:** we want to look up the internal representation of the atom “Abraham Lincoln”. We first obtain the UTF-8 encoding of this string:

<table>
<thead>
<tr>
<th>String</th>
<th>UTF-8 encoding (byte array)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Abraham Lincoln&quot;</td>
<td>065 098 114 097 104 097 109 032 076 105 110 099 111 108 110</td>
</tr>
</tbody>
</table>

We then use the byte-array of this value as the key to look up in the dictionary, and retrieve the internal representation as the payload:

<table>
<thead>
<tr>
<th>Byte Array</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>017 000 000 000 000 000 000 000</td>
<td>17</td>
</tr>
</tbody>
</table>

The BerkeleyDB database containing the integer-to-string dictionary is used to translate from internal atom representations to their “friendly” counterparts. The key for each entry consists of a 1-dimensional byte-array representing a 64-bit signed integer that is the representation of the internal value of an atom. The payload for each entry consists of a 1-dimensional byte-array representing a UTF-8 encoded string.

**Example:** we want to look up the “friendly” representation of the atom internally represented by the integer 17.

<table>
<thead>
<tr>
<th>Value</th>
<th>Byte Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>017 000 000 000 000 000 000 000</td>
</tr>
</tbody>
</table>

We use the byte-array of this value as the key to look up in the dictionary, and retrieve the UTF-8 encoded friendly representation as the payload:

<table>
<thead>
<tr>
<th>UTF-8 encoding (byte array)</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>065 098 114 097 104 097 109 032 076 105 110 099 111 108 110</td>
<td>&quot;Abraham Lincoln&quot;</td>
</tr>
</tbody>
</table>

**Index**

The TripleT index is stored in a BerkeleyDB hash database. There is one entry for each unique atom in the dataset. It consists of a single physical file, listed in Table 4.2.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_name.index.dat</td>
<td>A BerkeleyDB hash database containing the TripleT index.</td>
</tr>
</tbody>
</table>

Table 4.2: TripleT index physical files
The key for each entry is a 1-dimensional byte-array representing a 64-bit signed integer that is the internal representation of an atom. The payload for each entry is a 1-dimensional byte-array containing six 64-bit signed integers. The representation of each part of the byte-array is listed in Table 4.3.

<table>
<thead>
<tr>
<th>Range</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 ··· 7]</td>
<td>Position (offset) in the S-bucket file (measured in numbers of triples) where triples containing the requested atom in the S-position start.</td>
</tr>
<tr>
<td>[8 ··· 15]</td>
<td>Number of triples in the S-bucket file containing the requested atom in the S-position.</td>
</tr>
<tr>
<td>[16 ··· 23]</td>
<td>Position (offset) in the P-bucket file (measured in numbers of triples) where triples containing the requested atom in the P-position start.</td>
</tr>
<tr>
<td>[24 ··· 31]</td>
<td>Number of triples in the P-bucket file containing the requested atom in the P-position.</td>
</tr>
<tr>
<td>[32 ··· 39]</td>
<td>Position (offset) in the O-bucket file (measured in numbers of triples) where triples containing the requested atom in the O-position start.</td>
</tr>
<tr>
<td>[40 ··· 47]</td>
<td>Number of triples in the O-bucket file containing the requested atom in the O-position.</td>
</tr>
</tbody>
</table>

Table 4.3: TripleT index payload contents

If an atom does not exist in a certain position anywhere in the dataset, then the index will contain the −1 special value to indicate this.

**Example:** we want to look up the atom “Abraham Lincoln” in the index. We first use the dictionary to retrieve the internal representation for this atom:

<table>
<thead>
<tr>
<th>Value</th>
<th>Byte Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>017 000 000 000 000 000 000 000</td>
</tr>
</tbody>
</table>

We then use the byte-array of this value as the key to look up in the index, and retrieve the payload:

<table>
<thead>
<tr>
<th>Byte Array</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>004 000 000 000 000 000 000 009</td>
<td>4</td>
</tr>
<tr>
<td>008 000 000 000 000 000 000 000</td>
<td>8</td>
</tr>
<tr>
<td>255 255 255 255 255 255 255 255</td>
<td>-1</td>
</tr>
<tr>
<td>255 255 255 255 255 255 255 255</td>
<td>-1</td>
</tr>
<tr>
<td>114 004 000 000 000 000 000 000</td>
<td>1138</td>
</tr>
<tr>
<td>003 000 000 000 000 000 000 000</td>
<td>3</td>
</tr>
</tbody>
</table>

In this case, we see that “Abraham Lincoln” is in eight triples in the S-position, starting from the fourth triple in the S-bucket file; in three triples in the O-position, starting from the 1138th triple in the O-bucket file; and is not in any triples in the P-position.
Buckets

Each bucket belonging to a TripleT database is stored in its own separate file. There are three buckets for each database, listed in Table 4.4.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_name.bucket.SOP.dat</td>
<td>A flat binary file containing the SOP bucket, which are all the values in the database in internal representation and sorted in the S-O-P order.</td>
</tr>
<tr>
<td>db_name.bucket.PSO.dat</td>
<td>A flat binary file containing the PSO bucket, which are all the values in the database in internal representation and sorted in the P-S-O order.</td>
</tr>
<tr>
<td>db_name.bucket.OSP.dat</td>
<td>A flat binary file containing the OSP bucket, which are all the values in the database in internal representation and sorted in the O-S-P order.</td>
</tr>
</tbody>
</table>

Table 4.4: TripleT buckets physical files

The bucket files themselves are flat binary files containing sequences of triples. Each bucket contains all triples belonging to the dataset, although the files themselves contain only the parts of each triple that are not already present in the index. For instance, the S-bucket contains only the \((p, o)\) part of each triple, with the \(s\) value being implicitly present as part of the index.

The files are made up of a sequence of signed 64-bit integers stored in binary format. The first two integers are part of the first \((s, p, o)\) triple in the bucket, the next two are part of the next triple, etcetera. The bucket files always store the atoms of individual triples in \((s, p, o)\) ordering, independent from the type of bucket or the sort order of the triples in them.

**Example**: consider the following example of buckets belonging to a toy dataset:

<table>
<thead>
<tr>
<th>S-bucket</th>
<th>P-bucket</th>
<th>O-bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1, 2, 1))</td>
<td>((1, 1, 7))</td>
<td>((1, 2, 1))</td>
</tr>
<tr>
<td>((1, 1, 7))</td>
<td>((1, 1, 9))</td>
<td>((2, 2, 1))</td>
</tr>
<tr>
<td>((1, 1, 9))</td>
<td>((2, 1, 4))</td>
<td>((2, 1, 4))</td>
</tr>
<tr>
<td>((2, 2, 1))</td>
<td>((1, 2, 1))</td>
<td>((1, 1, 7))</td>
</tr>
<tr>
<td>((2, 1, 4))</td>
<td>((2, 2, 1))</td>
<td>((1, 1, 9))</td>
</tr>
</tbody>
</table>

Each bucket file then contains the following:

<table>
<thead>
<tr>
<th>S-bucket</th>
<th>P-bucket</th>
<th>O-bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>((2, 1))</td>
<td>((1, 7))</td>
<td>((1, 2))</td>
</tr>
<tr>
<td>((1, 7))</td>
<td>((1, 9))</td>
<td>((2, 2))</td>
</tr>
<tr>
<td>((1, 9))</td>
<td>((2, 4))</td>
<td>((2, 1))</td>
</tr>
<tr>
<td>((2, 1))</td>
<td>((1, 1))</td>
<td>((1, 1))</td>
</tr>
<tr>
<td>((1, 4))</td>
<td>((2, 1))</td>
<td>((1, 1))</td>
</tr>
</tbody>
</table>

The binary file containing the S-bucket will then look as follows:

**S-bucket**

```
2 1 1 7 1 9 2 1 1 4
```
In this case, the index contains the remaining information that makes up the triples from the S-bucket: the fact that \( s = 1 \) belongs to the first three triple parts in the bucket, and \( s = 2 \) belongs to the 4th and 5th triple parts in the bucket.

### Statistics

The statistics of a TripleT database are stored in a BerkeleyDB hash database. It consists of a single physical file, listed in Table 4.5.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_name.stats.dat</td>
<td>A BerkeleyDB hash database containing the TripleT database statistics.</td>
</tr>
</tbody>
</table>

Table 4.5: TripleT statistics physical files

The statistical database contains information for estimating output sizes for single SAPs or joins between two SAPs, as well as some summarizing statistics. As such, the payload for each entry is a 1-dimensional byte-array containing a variable number of 64-bit signed integers, which always represent numbers of triples. The key for each entry is a 1-dimensional byte-array containing a variable number of 64-bit signed integers. The possible keys are listed in Table 4.6.

<table>
<thead>
<tr>
<th>Key</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>([-1])</td>
<td>Reserved key. Payload will contain the total number of triples in the dataset.</td>
</tr>
<tr>
<td>([-2])</td>
<td>Reserved key. Payload will contain the total number of unique atoms in the dataset.</td>
</tr>
<tr>
<td>([-3])</td>
<td>Reserved key. Payload will contain the total number of unique atoms in the S-position in all triples in dataset.</td>
</tr>
<tr>
<td>([-4])</td>
<td>Reserved key. Payload will contain the total number of unique atoms in the P-position in all triples in dataset.</td>
</tr>
<tr>
<td>([-5])</td>
<td>Reserved key. Payload will contain the total number of unique atoms in the O-position in all triples in dataset.</td>
</tr>
<tr>
<td>([-1, x, y])</td>
<td>Key for retrieving statistics for joins between atoms ( x ) and ( y ), where ( x ) and ( y ) are the atoms’ internal representations.</td>
</tr>
<tr>
<td>([-2, x, y])</td>
<td>Key for retrieving statistics for SAPs containing both atoms ( x ) and ( y ), where ( x ) and ( y ) are the atoms’ internal representations.</td>
</tr>
</tbody>
</table>

Table 4.6: TripleT statistics key layout

For the reserved keys that retrieve summarizing statistics, the payload consists of a 1-dimensional byte-array containing a single 64-bit signed integer, representing a number of triples.

**Example:** we want to look up the number of unique atoms in the S-position of all triples in the dataset. We do this using reserved key with value \(-3\):

<table>
<thead>
<tr>
<th>Value</th>
<th>Byte Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-3)</td>
<td>(253 \ 255 \ 255 \ 255 \ 255 \ 255 \ 255 \ 255)</td>
</tr>
</tbody>
</table>

We then use the byte-array of this value as the key to look up in the statistics database, and retrieve the payload.
In this case, we see that there are 47 unique subjects in the dataset.

For the keys used for retrieving join statistics, the payload consists of a 1-dimensional byte-array containing four 64-bit signed integers. The representation of each part of the byte-array is listed in Table 4.7, where $p_1$ and $p_2$ are the atoms used in the key.

<table>
<thead>
<tr>
<th>Range</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0 \cdots 7]$</td>
<td>The number of result bindings produced by join ($?x, p_1, ?y) \bowtie (?x, p_2, ?z)$.</td>
</tr>
<tr>
<td>$[8 \cdots 15]$</td>
<td>The number of result bindings produced by join ($?x, p_1, ?y) \bowtie (?z, p_2, ?x)$.</td>
</tr>
<tr>
<td>$[16 \cdots 23]$</td>
<td>The number of result bindings produced by join ($?x, p_1, ?y) \bowtie (?y, p_2, ?z)$.</td>
</tr>
<tr>
<td>$[24 \cdots 31]$</td>
<td>The number of result bindings produced by join ($?x, p_1, ?y) \bowtie (?z, p_2, ?y)$.</td>
</tr>
</tbody>
</table>

Table 4.7: TripleT join statistics payload contents

**Example:** we want to know how many results are produced by join ($?x, 57, ?y) \bowtie (?y, 28, ?z)$, where 57 and 28 the are internal representations of two atoms in the dataset. We do this using a key with value $[-1, 57, 28]$:

<table>
<thead>
<tr>
<th>Value</th>
<th>Byte Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>255 255 255 255 255 255 255 255</td>
</tr>
<tr>
<td>57</td>
<td>057 000 000 000 000 000 000 000</td>
</tr>
<tr>
<td>28</td>
<td>028 000 000 000 000 000 000 000</td>
</tr>
</tbody>
</table>

We then use the byte-array of this value as the key to look up in the statistics database, and retrieve the payload:

<table>
<thead>
<tr>
<th>Byte Array</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 000 000 000 000 000 000 000</td>
<td>0</td>
</tr>
<tr>
<td>003 000 000 000 000 000 000 000</td>
<td>3</td>
</tr>
<tr>
<td>118 000 000 000 000 000 000 000</td>
<td>118</td>
</tr>
<tr>
<td>020 000 000 000 000 000 000 000</td>
<td>20</td>
</tr>
</tbody>
</table>

In this case, we see that there are 118 result bindings produced by this join.

For the keys used for retrieving statistics for 2-atom SAPs, the payload consists of a 1-dimensional byte-array containing six 64-bit signed integers. The representation of each part of the byte-array is listed in Table 4.8, where $p_1$ and $p_2$ are the atoms used in the key.

**Example:** we want to know how many results match $(28, 57, ?x)$, where 57 and 28 the are internal representations of two atoms in the dataset. We do this using a key with value $[-2, 57, 28]$:

<table>
<thead>
<tr>
<th>Value</th>
<th>Byte Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>255 255 255 255 255 255</td>
</tr>
<tr>
<td>57</td>
<td>057 000 000 000 000 000</td>
</tr>
</tbody>
</table>
4.2. SYSTEM ARCHITECTURE

Table 4.8: TripleT join statistics payload contents

<table>
<thead>
<tr>
<th>Range</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 ··· 7]</td>
<td>The number of results matching SAP ((p_1, p_2, ?x)).</td>
</tr>
<tr>
<td>[8 ··· 15]</td>
<td>The number of results matching SAP ((p_1, ?x, p_2)).</td>
</tr>
<tr>
<td>[16 ··· 23]</td>
<td>The number of results matching SAP ((p_2, p_1, ?x)).</td>
</tr>
<tr>
<td>[24 ··· 31]</td>
<td>The number of results matching SAP ((?x, p_1, p_2)).</td>
</tr>
<tr>
<td>[32 ··· 39]</td>
<td>The number of results matching SAP ((p_2, ?x, p_1)).</td>
</tr>
<tr>
<td>[40 ··· 47]</td>
<td>The number of results matching SAP ((?x, p_2, p_1)).</td>
</tr>
</tbody>
</table>

We then use the byte-array of this value as the key to look up in the statistics database, and retrieve the payload:

<table>
<thead>
<tr>
<th>Byte Array</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>006 000 000 000 000 000 000 000 000</td>
<td>6</td>
</tr>
<tr>
<td>000 000 000 000 000 000 000 000 000</td>
<td>0</td>
</tr>
<tr>
<td>027 000 000 000 000 000 000 000 000</td>
<td>27</td>
</tr>
<tr>
<td>000 000 000 000 000 000 000 000 000</td>
<td>0</td>
</tr>
<tr>
<td>038 000 000 000 000 000 000 000 000</td>
<td>38</td>
</tr>
<tr>
<td>002 000 000 000 000 000 000 000 000</td>
<td>2</td>
</tr>
</tbody>
</table>

In this case, we see that there are 27 results that match SAP \((28, 57, ?x)\).

4.2.2 Internal operations

Internally, the components of TripleT resemble those of a bare-bones database engine. It features only those parts required for answering BGP queries (our non-optimized methods for inserting external datasets into a TripleT database are not discussed here). An overview of the relevant components is provided in Figure 4.2.

The parser is not discussed. For an in-depth discussion of the plan generator, please refer to Chapter 5. In the remainder of this section, we will discuss the plan executor. The plan executor takes the query plans generated by the plan generator and executes them, producing the results of the query. First of all, we define a query plan as a tree consisting of physical operators as its nodes. We formally define the language for such a query plan as follows:

\[
\text{QueryPlan ::= Operator}
\]

\[
\text{Operator ::= Scan} \mid \text{Sort} \mid \text{MergeJoin} \mid \text{HashJoin} \mid \text{Filter}
\]

\[
\text{Scan ::= (Bucket, Pattern, IndexLookup)}
\]

\[
\text{Sort ::= (Operator, SortOrder)}
\]
Figure 4.2: TripleT internal component architecture

MergeJoin ::= (Operator, Operator, SortOrder)
HashJoin ::= (Operator, Operator)
Filter ::= (Operator, FilterExpression)
Bucket ::= "S" | "P" | "O"
Pattern ::= Triple
Triple ::= (TripleItem, TripleItem, TripleItem)
TripleItem ::= Atom | Variable
Atom ::= [1-9][0-9]*
Variable ::= ["?"[1-9][0-9]*
IndexLookup ::= (([1-9][0-9]*),5  [1-9][0-9]*)
SortOrder ::= Variable+
FilterExpression ::= (Variable, Condition)+
Condition ::= And |
            Or |
            Predicate
And ::= (Condition, Condition)

Or ::= (Condition, Condition)

Predicate ::= (("<" | ">" | "="), [1-9][0-9]*)

Note that this definition is a simplification for the sake of clarity and should not be treated as a parsable grammar. Of course, query plans can be easily visualized as trees. An example is provided in Figure 4.3.

Figure 4.3: An example of a query plan

The results of queries are sets of bindings, which we have formally defined in Definition 7 in Chapter 2. A consequence of this is that individual operators that make up a query plan also produce bindings as their output, and all but the Scan operator also take bindings as their input. A list of operators is provided in Table 4.9.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>Represents a single, sequential scan over a bucket file.</td>
</tr>
<tr>
<td>Sort</td>
<td>Represents a sorting operation on a stream of bindings received.</td>
</tr>
<tr>
<td>Merge join</td>
<td>Represents a join of two streams of bindings received. It is assumed that each stream is sorted on the join variables.</td>
</tr>
<tr>
<td>Hash join</td>
<td>Represents a join of two streams of bindings received. No assumptions on the input are made.</td>
</tr>
<tr>
<td>Filter</td>
<td>Represents a filtering (or domain restriction) operation on a stream of bindings received.</td>
</tr>
</tbody>
</table>

Table 4.9: Operators in TripleT’s physical query language

These operators form the heart of the TripleT database engine. We discuss them individually in the remainder of this section. A few notes:

- When we say an operator is *lazy*, we mean the operator only yields results one at the time as
they are being requested by other consumers of its output. The opposite would be an operator that pushes its results as soon as they are available.

- When we say an operator supports *streaming* (also called *non-blocking*), we mean the operator is both lazy and only needs to consume results from its input operator(s) up until the point where it can yield a result itself. The opposite would be an operator that first needs to consume all results from its input operator(s) before it can start yielding results itself (also called a *blocking* operator).

### Scan operator

This operator represents a single, sequential scan over a TripleT bucket file.

**Input:** The input consists of a designation indicating which bucket to open, a pattern to match, and an index payload belonging to the primary atom in the pattern. The pattern is a Single Access Pattern as defined in Definition 3 in Chapter 2.

**Output:** The output consists of a sequence of bindings, where there is one binding for each variable in the input pattern. This operator yields its output in a lazy manner, e.g. not unless requested by another operator. This operator supports streaming.

The operator’s information flow is visualized in Figure 4.4.

![Figure 4.4: Scan operator](image)

The operator’s procedure in described in listing Procedure 1.

**Example:** we want to have a scan that retrieves all triples that feature “Abraham Lincoln” as a subject. For this, we need a scan that matches the pattern (“Abraham Lincoln”, ?x, ?y). We will need to tell the scan to look in the S-bucket, and we need to supply it with the payload belonging to our pattern atom in the S-position.

When the query plan is executed, the scan operator will receive its payload information from a lookup operator and will combine this with the remaining supplied information to open the S-bucket file, scan to the correct position as indicated in the payload, and start sequentially reading off triples, creating and returning bindings for those triples that match the pattern.
When the scan encounters the triple (“Abraham Lincoln”, “Birth Date”, “1809-02-12”), it yields the binding (?x : “Birth Date”, ?y : “1809-02-12”). When the scan encounters the triple (“Alain Connes”, “Surname”, “Connes”), it knows to stop as no more triples featuring “Abraham Lincoln” in the S-position can occur.

Sort operator

This operator represents a sorting operation on a stream of bindings received.

**Input:** The input consists of a sequence of bindings, and an indication of how they should be sorted.

**Output:** The output consists of a sequence of bindings, which have been sorted in the order that was requested. This operator yields its output in a lazy manner, e.g. not unless requested by another operator. This operator does not support streaming; the input first needs to be read in full and then sorted before the output can be yielded.

The operator’s information flow is visualized in Figure 4.5.

![Figure 4.5: Sort operator](image)

The operator’s procedure is described in listing Procedure 2.

**Example:** we want to sort a sequence of (?x : abc, ?y : def) bindings returned by a scan operator in the (?x, ?y) order. To do so, we need to create a sort operator, tell it to sort in the (?x, ?y) order, and provide it with the scan operator to serve as its input.
CHAPTER 4. A TRIPLET DATABASE ENGINE

### Procedure 2: Sort(input, sortOrder)

1. $tmp \leftarrow$ a new file
2. while $input.HasNext$ do
3.   $next \leftarrow input.ReadNext()$
4.   $tmp.Write(next)$
5. $ExternalSort(tmp, sortOrder)$
6. $cursor \leftarrow$ a cursor for reading bindings from $tmp$
7. while $cursor.HasNext$ do
8.   $next \leftarrow cursor.ReadNext()$
9.   yield $next$

When the query plan is executed, the sort operator will read all bindings from its input source, output them to a flat binary file, and perform an external sort on this file. After sorting is complete, the operator will read in and yield the bindings from the sorted file in a lazy manner.

**Merge Join operator**

This operator represents a natural join on two streams of bindings received as input, which are sorted on their shared join variables.

**Input:** The input consists of two sequences of bindings which have at least one variable in common. The bindings are assumed to have been sorted on these shared join variables.

**Output:** The output consists of a sequence of bindings which are the result of a join on the two input sequences and where each binding contains the combined variables from both input streams. This operator yields its output in a lazy manner, e.g. not unless requested by another operator. This operator supports streaming.

The operator’s information flow is visualized in Figure 4.6.

![Merge Join operator](image)

**Figure 4.6: Merge Join operator**

The operator’s procedure in described in listings Procedure 3, 4, 5, and 6.

**Example:** we want to join a sequence of ($x : abc, y : def$) bindings returned by a scan with a sequence of ($y : ijk, z : lmn$) bindings returned by some other scan. Both streams of bindings have been sorted in on the $y$ variable. To accomplish our join we need to create a merge join operator
Procedure 3: MergeJoin(left, right, sortOrder)

1 buffer ← a new buffer
2 nextLeft ← a placeholder for bindings
3 nextRight ← a placeholder for bindings
4 mode ← Advance
5 while true do
6   switch mode do
7     case Advance
8       if MergeJoinAdvance() then
9         if IsJoinMatch(nextLeft, nextRight) then
10            buffer.BeginBlock()
11            mode ← BlockRead
12            yield the merged bindings nextLeft and nextRight
13       else
14           return
15     case BlockRead
16       if MergeJoinAdvanceLeft() ∧ IsJoinMatch(nextLeft, nextRight) then
17         yield the merged bindings nextLeft and nextRight
18       else
19         prevRight ← nextRight
20         if MergeJoinAdvanceRight() then
21            if rightNext ≻ prevRight then
22              if IsJoinMatch(nextLeft, nextRight) then
23                 buffer.BeginBlock()
24                 mode ← BlockRead
25                 yield the merged bindings nextLeft and nextRight
26              else
27                mode ← Advance
28            else
29               buffer.RepeatBlock()
30               mode ← BlockRepeat
31       else
32         return
33     case BlockRepeat
34       if buffer.BlockHasNext then
35          leftNext ← buffer.NextFromBlock()
36          yield the merged bindings nextLeft and nextRight
37       else if MergeJoinAdvanceLeft() then
38          goto 19
39       else
40         return
and provide it with the two scan operators that will serve as its input, as well as the sort ordering of
the results returned by the inputs. The merge join operator itself will then figure out on the fly which
variables the two input streams have in common and need to be joined on.

When the query plan is executed, the merge join operator will start reading from the two scans and
performs a regular sort-merge join on the results. The results from one of the inputs will be cached
in a dynamic buffer while the results from the other input are joined against them. If two sets of
bindings can make a join, then all unique variables belonging to both sets of bindings will be collected
in a new set of bindings which is then yielded as a result.

In our example, the join operator will cache the (?z,?y) bindings from one input scan in the buffer
as they are being read. When it reads in the (?x,?y) bindings from the other input scan, for each
(?x,?y) binding encountered a regular sort-merge join will be performed using the (?z,?y) bindings
from the buffer. If a match is found the merged (?x,?y,?z) binding set will be yielded as output. The
buffer meanwhile shrinks and expands dynamically as the operation is performed, only keeping the
minimal amount of results in memory (and on disk if needed).

Hash Join operator

This operator represents a natural join between two streams of bindings received as input.
4.2. SYSTEM ARCHITECTURE

**Input**: The input consists of two sequences of bindings that are assumed to have at least one variable in common. No other assumptions about the input data are made.

**Output**: The output consists of a sequence of bindings which are the result of a join on the two input sequences and where each binding contains the combined variables from both input streams. This operator yields its output in a lazy manner, e.g. not unless requested by another operator. This operator does not support streaming; both inputs need to be read in full and then stored in a hash database before they can be joined and the output can be yielded.

The operator’s information flow is visualized in Figure 4.7.

![Diagram of Hash Join operator](image-url)

**Figure 4.7: Hash Join operator**

The operator’s procedure is described in listing Procedure 7.

**Example**: we want to join a sequence of (?x : abc, ?y : def) bindings returned by a scan with a sequence of (?y : ijk, ?z : lmn) bindings returned by some other scan. Both streams of bindings are unsorted. To accomplish our join we need to create a hash join operator and provide it with the two scan operators that will serve as its input. The hash join operator itself will then figure out on the fly which variables the two input streams have in common and need to be joined on.

When the query plan is executed, the hash join operator will first read in the two input streams in full and output them into two BerkeleyDB B-Tree databases for temporary storage. The operator will read the first bindings from each input stream to determine the shared join variables. These shared join variables will then be fed to a hash function to produce the key in the database, whereas the full variables themselves will be stored as the payload.

Next, the databases will be read sequentially and the chunks of bindings with the same keys will be joined in-memory. To do so the bindings from one database will be stored in a hash table in memory while the bindings from the other database are read one by one to see if a join partner exists in the hash table. If two sets of bindings can make a join, then all unique variables belonging to both sets of bindings will be merged in a new set of bindings which is then yielded as a result.

In our example, the join operator will first read in all bindings from both input scans and store them in two B-Tree databases, with a hash of the shared ?y binding as the key and the bindings as the payload. It will then start reading in the (?x, ?y) bindings from one database in chunks, grouped by the value of their key. For each chunk of (?x, ?y) bindings encountered it will read in any (?y, ?z) bindings with the same key value and attempt to locate any join partners (if they exist). When found, it will yield the merged (?x, ?y, ?z) binding set as output.
Procedure 7: HashJoin(left, right)

1. \( shared \leftarrow \) set of shared variables between bindings from \( left \) and \( right \)
2. \( dbLeft \leftarrow \) a new B-Tree database
3. \( dbRight \leftarrow \) a new B-Tree database
4. \( \text{while } left.HasNext \) do
   5. \( next \leftarrow left.ReadNext() \)
   6. \( vars \leftarrow next[shared] \)
   7. \( dbLeft.Insert(Hash(vars), next) \)
5. \( \text{while } right.HasNext \) do
   8. \( next \leftarrow right.ReadNext() \)
   9. \( vars \leftarrow next[shared] \)
   10. \( dbRight.Insert(Hash(vars), next) \)
6. \( buffer \leftarrow \) a new hash table
7. \( currentChunk \leftarrow \) next set of values from \( dbLeft \) with the same key values
8. \( buffer.Insert(currentChunk) \)
9. \( \text{while } dbRight.HasNext \) do
   10. \( next \leftarrow dbRight.ReadNext() \)
   11. \( \text{if } next.Key = \text{key from currentChunk} \) then
       12. \( vars \leftarrow next[shared] \)
       13. \( \text{if } buffer \text{ contains bindings matching } vars \) then
           14. \( \text{forall the bindings } b \text{ from } buffer \text{ that match } vars \) do
               15. \( \text{yield the merged bindings } next \text{ and } b \)
   16. \( \text{else if } next.Key \succ \text{key from currentChunk} \) then
       17. \( buffer.Clear() \)
       18. \( currentChunk \leftarrow \) next set of values from \( dbLeft \) with the same key values
       19. \( buffer.Insert(currentChunk) \)

Filter operator

This operator represents a filtering (or domain restriction) operation on a stream of bindings received.

**Input:** The input consists of a sequence of bindings, and a series of filter conditions to filter them on. Optionally, information on how the input is sorted can be provided.

**Output:** The output consists of a sequence of bindings which meet the conditions set by the filter. This operator yields its output in a lazy manner, e.g. not unless requested by another operator. This operator supports streaming.

The operator’s information flow is visualized in Figure 4.8.

The operator’s procedure in described in listing Procedure 8.

**Example:** we want to filter a series of \( (\mathit{x}, \mathit{y}) \) bindings yielded by some sort operator such that only bindings with \( \mathit{x} > 20 \) and \( \mathit{x} < 40 \). To accomplish this we need to create a filter operator, feed it the sort operator as input, and provide it with an expression of our desired filter condition. Additionally, because the input comes from a sort operator we can provide the filter operator with information about this sort order, which is not necessary but will aid it during its operation.

When the query plan is executed, the filter operator will start reading in bindings from the sort
operator, discarding all those which do not meet the condition and yielding as result those that do. Then, when a binding with value \((?x: 42)\) is encountered, the filter operator knows that because the input is sorted on \((?x, ?y)\), no more bindings which meet the condition can occur, and it stops reading.

### 4.2.3 Memory management

After a query plan has been generated, the available system memory is distributed over the operators in the plan. There are no limitations on how much memory TripleT can use, though our testing machine where we run our experiments (see Section 6.2.3 in Chapter 6) is equipped with 4 GB, and as such we have artificially limited the memory TripleT can use to this amount.

Operators which are closer to the leaves of the query plan are expected to handle more data and would thus benefit from more available memory. Hence, for some query plan \(Q_P\) for BGP \(P\), memory is divided over its operators using the following heuristic:

1. For every operator \(o \in Q_P\), which is a node in the query plan, we compute its average distance to a leaf, denoted \(d(o)\).

2. For every operator, we then compute the fraction \(0 < f(o) \leq 1\) of memory as

\[
f(o) = \frac{1}{d(o) \cdot \sum_{o \in Q_P} d(o)}.
\]

3. Each operator \(o \in Q_P\) will then be assigned \(f(o) \cdot M\) memory, where \(M\) denotes the size of the available memory in bytes.
Scan- and filter operators do not require any memory because they are lazy (or, non-blocking); the sort operator and both join operators do. Hence, in the calculation above we actually only consider operators from the latter group. The sort operator uses its memory for sorting, the merge join operator uses its memory for its buffer, and the hash join operator uses its memory for the two BerkeleyDB B-Tree databases used for temporary storage of its input.

4.3 Implementation details

Our implementation of TripleT is written in C$\sharp$ and builds on the Microsoft .NET Framework, version 4.0.30319 SP1Rel. All database-related functionality is contained in a single class library. For testing purposes, this library is called through a command-line executable, also written in C$\sharp$ for .NET 4.

The TripleT library consists of 4214 lines of code, as measured by Visual Studio’s Code Metrics. The build was performed by Microsoft Visual Studio 2010 Ultimate, version 10.0.40219.1 SP1Rel. The target platform for the TripleT library was Any CPU; for the testing executable this was x86. In the post-build event we call

```bash
call "$(DevEnvDir)\..\tools\vsvars32.bat"
editbin /largeaddressaware $(TargetPath)
```

to make the built executables large address aware, meaning when executed on a 64-bit platform they can address up to 4 GB of memory, even though as 32-bit processes they would otherwise be limited to addressing 2 GB.

We also use Berkeley DB, version 5.3.21, as the external library which provides us with the B+ and hash databases that we use in our implementation. We have built Berkeley DB and its C$\sharp$ API on the same machine as TripleT, targeted for the x86 platform. Here too we have used the editbin utility to make the Berkeley DB libraries large address aware.

A brief description of the source code for our implementation, as well as its programming API, is provided in Appendix B.

4.4 Chapter summary

In this chapter we’ve presented a thorough discussion of our implementation of the TripleT RDF store. We have briefly talked about the idea behind TripleT, and its novel value-based, role-free indexing technique. We have then presented our secondary-memory implementation of this system, which we have used throughout our investigation. We have discussed the global architecture of our implementation, and the physical layout of the various files used for data storage. We have concluded with a description of the internal language used for query plans, including a listing of each of the physical operators that has been implemented.
In this chapter we describe in detail the framework we have developed for generating optimized query plans for evaluating Basic Graph Patterns (BGPs) on the TripleT RDF index. The framework consists of a generic, rule-based algorithm in which a number of decision points can be manipulated by a given set of rules. We also describe the specific rules we have used in our implementation.

Throughout this chapter, \( P \) is defined as a BGP, consisting of \( k \) Simple Access Patterns (SAPs)

\[
(s_1, p_1, o_1), (s_2, p_2, o_2), \ldots, (s_k, p_k, o_k) \equiv S_1, S_2, \ldots, S_k.
\]

We follow the path this BGP takes through our framework. We first discuss the notion of an atom collapse, which can be thought of as the solution space for finding a join graph for \( P \). The join graph itself represents the physical access plan for retrieving results matching each SAP in \( P \), as well as the order in which these results need to be joined together to produce the query results. This graph forms the basis for computing an actual physical query plan, in which the ordering of the joins from the join graph is determined and physical operators are assigned for each step.

### 5.1 Atom collapses

We define the atom collapse \( C_P \) of \( P \) as the undirected edge-labeled graph with atoms and variables from the SAPs \( S_i \in P \) as its nodes, and edges between them to indicate there is a shared variable between the SAPs from the two nodes.

Formally, the set of nodes consists of atom-nodes and variable-nodes. For each SAP \( S_i \in P \) we have an atom-node \( (a, \{ S_i \}) \) for each unique atom \( a \in S_i \). We have a variable-node \( (v, P_v) \) for each unique variable \( v \in P \) with \( P_v \subseteq P \) being the set of SAPs which contain \( v \). A special case arrives for SAPs that do not contain any atoms. For those SAPs, we have a special atom node \( (a_0, \{ S_i \}) \) in the set of nodes, where \( a_0 \) is a nil-atom.

Let \((x, X)\) and \((y, Y)\) be nodes in the collapse graph. In the set of edges we have an undirected edge \((x, X) - (y, Y)\) with label \( L \) if and only if there exists some variable \( v \) such that \( v \in S_X, S_X \in X \) and \( v \in S_Y, S_Y \in Y \) and \( S_X \neq S_Y \). Label \( L \) consists of a set of \((S_i, S_j, v, p_i, p_j)\) tuples, where there are tuples for every variable \( v \) such that \( v \in S_i, S_i \in X \) and \( v \in S_j, S_j \in Y \) with \( S_i \neq S_j \) and with \( p_i \) and \( p_j \) denoting the positions (s, p, or o) variable \( v \) has in \( S_i \) and \( S_j \) respectively. Note that one variable

\[
\]
can occur in multiple tuples in one label, as long as each tuple as a whole is unique within $L$. Also note that, because of the undirected nature of the edges, if we have an edge $(x, X) - (y, Y) : L$ then this implies we do not have an edge $(y, Y) - (x, X) : L'$.

### 5.1.1 Examples

We describe a number of example BGPs and their respective collapse graphs.

1. Let $P = (a, b, ?x)_1$. Then $C_P$ has nodes

   $(a, \{1\})$

   $(b, \{1\})$

   $(?x, \{1\})$

   and no edges.

2. Let $P = (a, b, ?x)_1 \land (\?x, a, a)_2$. Then $C_P$ has nodes

   $(a, \{1\})$

   $(b, \{1\})$

   $(?x, \{1, 2\})$

   $(a, \{2\})$

   and edges

   $(a, \{1\}) - (a, \{2\}) : \{(1, 2, ?x, o, s)\}$

   $(b, \{1\}) - (a, \{2\}) : \{(1, 2, ?x, o, s)\}$

   $(a, \{1\}) - (?x, \{1, 2\}) : \{(1, 2, ?x, o, s)\}$

   $(b, \{1\}) - (?x, \{1, 2\}) : \{(1, 2, ?x, o, s)\}$

   $(a, \{2\}) - (?x, \{1, 2\}) : \{(2, 1, ?x, s, o)\}$.

3. Let $P = (a, b, ?x)_1 \land (\?x, c, ?y)_2 \land (\?y, d, e)_3$. Then $C_P$ has nodes

   $(a, \{1\})$

   $(b, \{1\})$

   $(c, \{2\})$

   $(d, \{3\})$

   $(e, \{3\})$

   $(?x, \{1, 2\})$

   $(?y, \{2, 3\})$
and edges

\[
(a, \{1\}) - (c, \{2\}) : \{(1, 2, ?x, o, s)\}
\]
\[
(a, \{1\}) - (?x, \{1, 2\}) : \{(1, 2, ?x, o, s)\}
\]
\[
(a, \{1\}) - (?y, \{2, 3\}) : \{(1, 2, ?x, o, s)\}
\]
\[
(b, \{1\}) - (c, \{2\}) : \{(1, 2, ?x, o, s)\}
\]
\[
(b, \{1\}) - (?x, \{1, 2\}) : \{(1, 2, ?x, o, s)\}
\]
\[
(b, \{1\}) - (?y, \{2, 3\}) : \{(1, 2, ?x, o, s)\}
\]
\[
(c, \{2\}) - (d, \{3\}) : \{(2, 3, ?y, o, s)\}
\]
\[
(c, \{2\}) - (?)x, \{1, 2\}) : \{(2, 1, ?x, s, o)\}
\]
\[
(c, \{2\}) - (?y, \{2, 3\}) : \{(2, 3, ?y, o, s)\}
\]
\[
(d, \{3\}) - (?x, \{1, 2\}) : \{(3, 2, ?y, s, o)\}
\]
\[
(d, \{3\}) - (?y, \{2, 3\}) : \{(3, 2, ?y, s, o)\}
\]
\[
(e, \{3\}) - (?x, \{1, 2\}) : \{(3, 2, ?y, s, o)\}
\]
\[
(e, \{3\}) - (?y, \{2, 3\}) : \{(3, 2, ?y, s, o)\}
\]
\[
(?x, \{1, 2\}) - (?y, \{2, 3\}) : \{(1, 2, ?x, o, s), (2, 3, ?y, o, s)\}.
\]

4. Let \( P = (a, ?x, ?y)_1 \land (?y, ?x, a)_2 \). Then \( C_P \) has nodes

\[
(a, \{1\})
\]
\[
(?x, \{1, 2\})
\]
\[
(?y, \{1, 2\})
\]
\[
(a, \{2\})
\]

and edges

\[
(a, \{1\}) - (a, \{2\}) : \{(1, 2, ?x, p, p), (1, 2, ?y, o, s)\}
\]
\[
(a, \{1\}) - (?x, \{1, 2\}) : \{(1, 2, ?x, p, p), (1, 2, ?y, o, s)\}
\]
\[
(a, \{1\}) - (?y, \{1, 2\}) : \{(1, 2, ?x, p, p), (1, 2, ?y, o, s)\}
\]
\[
(a, \{2\}) - (?x, \{1, 2\}) : \{(1, 2, ?x, p, p), (1, 2, ?y, o, s)\}
\]
\[
(a, \{2\}) - (?y, \{1, 2\}) : \{(1, 2, ?x, p, p), (1, 2, ?y, o, s)\}
\]
\[
(?x, \{1, 2\}) - (?y, \{1, 2\}) : \{(1, 2, ?x, p, p), (1, 2, ?y, o, s)\}.
\]
We describe a number of example BGPs and their respective join graphs.

We define a more concrete representation of the atom collapse algorithm in the form of pseudo-code has been provided in listing Procedure 9. For an input BGP of size $k$, this procedure runs in $O(k^4)$ time.

5.1.2 Computing the atom collapse

At a high level, the (non-optimized) algorithm for computation of the atom collapse $C_P$ of BGP $P$ consists of the following steps:

1. Let $NODES = \emptyset$ be a new, empty set. For each SAP $S_i \in P$, let $A_i$ be the set of unique atoms in $S_i$ and let $V_i$ be the set of unique variables in $S_i$. For every atom $a \in A_i$, we add new node $(a, \{S_i\})$ to $NODES$. For every variable $v \in V_i$, if there exists already a node $(v, P_v)$ in $NODES$ for some $P_v$ we add $S_i$ to $P_v$, and if not we add new node $(v, \{S_i\})$ to $NODES$.

2. Let $LABELS = \emptyset$ be a new, empty set. For each pair of SAPs $S_i, S_j \in P$ with $i \neq j$, and for each pair of variables $v_i \in S_i, v_j \in S_j$ such that $v_i = v_j$, we add label $(S_i, S_j, v_i, p_i, p_j)$ to $LABELS$, where $p_i$ and $p_j$ are the positions $(s, p, o)$ of $v_i$ and $v_j$ in $P_i$ and $P_j$ respectively.

3. Let $EDGES = \emptyset$ be a new, empty set. For each label $l = (S_i, S_j, v_i, p_i, p_j) \in LABELS$, for each pair of nodes $(x, X), (y, Y) \in NODES$ such that $S_i \in X$ and $S_j \in Y$, if there exists already an edge $((x, X) - (y, Y) : L)$ in $EDGES$ for some set of labels $L$ we add $l$ to $L$, and if not we add new edge $((x, X) - (y, Y) : \{l\})$ to $EDGES$ provided that there is no edge $((y, Y) - (x, X) : L') \in EDEGES$.

4. We now return atom collapse $C_P$ where $NODES$ is the set of nodes in $C_P$ and $EDGES$ is the set of edges in $C_P$.

A more concrete representation of the atom collapse algorithm in the form of pseudo-code has been provided in listing Procedure 9. For an input BGP of size $k$, this procedure runs in $O(k^4)$ time.

5.2 Join graphs

We define a join graph $J_P$ of $P$ as a subgraph of atom collapse $C_P$, with the nodes from $J_P$ being a subset of the atom-nodes from $C_P$ such that for each SAP $S_i \in P$ there is exactly one node $(a, \{S_i\})$ in $J_P$, and the edges from $J_P$ being the same as those from $C_P$.

Formally, the set of nodes in $J_P$ is defined as

$$NODES(J_P) \subseteq \{ (x, X) \in NODES(C_P) \mid x \text{ is an atom} \}$$

such that $\forall S_i \in P, (\exists! (x, X) \in NODES(J_P). (X = \{S_i\}))$.

The set of edges is defined as

$$EDGES(J_P) = \{ ((x, X) - (y, Y) : L) \in EDEGES(C_P) \mid (x, X), (y, Y) \in NODES(J_P) \}.$$ 

Note that $P$ can have multiple distinct, valid join graphs.

5.2.1 Examples

We describe a number of example BGPs and their respective join graphs.

1. Let $P = (a, b, ?x)_1$. Then $J_P$ has the following options:
   - Nodes
     $$(a, \{1\})$$
### Procedure 9: ComputeAtomCollapse($P$)

1. $\text{NODES} \leftarrow \text{a new list}$
2. For all the SAPs $S_i \in P$ do
3.   $A \leftarrow$ unique atoms from $S_i$
4.   $V \leftarrow$ unique variables from $S_i$
5.   For all the $a \in A$ do
6.     $n \leftarrow \text{a new node } (a, \{S_i\})$
7.     $\text{NODES}.\text{Append}(n)$
8.   For all the $v \in V$ do
9.     If $\text{NODES}$ contains a node $(v, P_v)$ for some $P_v$ then
10.    $P_v.\text{Insert}(S_i)$
11.   Else
12.      $n \leftarrow \text{a new node } (v, \{S_i\})$
13.      $\text{NODES}.\text{Append}(n)$
14. $\text{LABELS} \leftarrow \text{a new list}$
15. For all the pairs $S_i, S_j \in P$ such that $i \neq j$ do
16.   For all the combinations $p_i, p_j \in \{s, p, o\}$ do
17.      If $S_i[p_i]$ and $S_j[p_j]$ are both variables and $S_i[p_i] = S_j[p_j]$ then
18.         $l \leftarrow \text{a new label } (S_i, S_j, S_i[p_i], p_i, p_j)$
19.         $\text{LABELS}.\text{Append}(l)$
20. $\text{EDGES} \leftarrow \text{a new list}$
21. For all the labels $l = (N, M, v, p_N, p_M) \in \text{LABELS}$ do
22.   For $i = 0$ to $\text{NODES}.\text{Count}$ do
23.     For $j = i + 1$ to $\text{NODES}.\text{Count}$ do
24.       $(x, X) \leftarrow \text{NODES}[i]$
25.       $(y, Y) \leftarrow \text{NODES}[j]$
26.       If $N \in X$ and $M \in Y$ then
27.         If $\text{EDGES}$ contains an edge $((x, X) - (y, Y) : L)$ for some $L$ then
28.           $L.\text{Insert}(l)$
29.         Else
30.           $e \leftarrow \text{a new edge } ((x, X) - (y, Y) : \{l\})$
31.           $\text{EDGES}.\text{Append}(l)$
32. $C_P \leftarrow \text{a new atom collapse graph } (\text{NODES}, \text{EDGES})$
33. Return $C_P$

and no edges.

- Nodes

(b, \{1\})

and no edges.

2. Let $P = (a, b, ?x)_1 \land (?x, a, a)_2$. Then $J_P$ has the following options:
• Nodes

\[(a, \{1\})\]
\[ (a, \{2\})\]

and edges

\[ (a, \{1\}) - (a, \{2\}) : \{(1, 2, ?x, o, s)\} \]

• Nodes

\[(b, \{1\})\]
\[ (a, \{2\})\]

and edges

\[ (b, \{1\}) - (a, \{2\}) : \{(1, 2, ?x, o, s)\} \]

3. Let \( P = (a, b, ?x) \land (\neg x, c) \land (\neg y, d) \land (\neg y, e) \). Then two of the options for \( J_P \) are:

• Nodes

\[(a, \{1\})\]
\[ (c, \{2\})\]
\[ (d, \{3\})\]

and edges

\[ (a, \{1\}) - (c, \{2\}) : \{(1, 2, ?x, o, s)\} \]
\[ (c, \{2\}) - (d, \{3\}) : \{(2, 3, ?y, o, s)\} \]

• Nodes

\[(b, \{1\})\]
\[ (c, \{2\})\]
\[ (e, \{3\})\]

and edges

\[ (b, \{1\}) - (c, \{2\}) : \{(1, 2, ?x, o, s)\} \]
\[ (c, \{2\}) - (e, \{3\}) : \{(2, 3, ?y, o, s)\} \]

Two more options exist but are omitted here.
4. Let \( P = (a, ?x, ?y)_1 \land (?y, ?x, a)_2 \). Then \( J_P \) has the following options:

- Nodes

\[
(a, \{1\}) \\
(a, \{2\})
\]

and edges

\[
(a, \{1\}) \rightarrow (a, \{2\}) : \{(1, 2, ?x, p, p), (1, 2, ?y, o, s)\}.
\]

5.2.2 Computing the join graph

We present an algorithm for computing a possible join graph \( J_P \) of BGP \( P \). Because multiple distinct, valid join graphs may be possible for \( P \), the algorithm involves a decision point, where a choice needs to be made. Such a choice can be made based on any set of rules that are independent from the algorithm itself.

At a high level, the (non-optimized) algorithm consists of the following steps:

1. Let \( C_P \) be the atom collapse of \( P \).
2. Let \( SEEDS = \emptyset \) and \( TODO = \emptyset \), be new, empty sets.
3. If \( TODO = \emptyset \), then let \( SEEDS = \{(x, X) \in NODES(C_P) \mid x \text{ is an atom}\} \), and if not then let \( SEEDS = \{(x, X) \in NODES(C_P) \mid x \text{ is an atom} \land X \subseteq TODO\} \).
4. Decision point: let \((s, S)\) be a seed node chosen from \( SEEDS \).
5. Modify \( TODO \) such that \( TODO = TODO \setminus S \).
6. For all nodes \((x, X) \in NODES(C_P)\) such that \((x, X) \neq (s, S)\), remove \( S \) from \( X \). If, after doing so, \( X = \emptyset \), then remove \((x, X)\) from \( NODES(C_P) \). If, after doing so, \( X \neq \emptyset \) and \( x \) is a variable, then modify \( TODO \) such that \( TODO = TODO \cup X \).
7. If \( TODO \neq \emptyset \), then continue at step 3.
8. Return join graph \( J_P \) where \( NODES(J_P) = NODES(C_P) \) and \( EDGES(J_P) = \{(x, X) \rightarrow (y, Y) : L \in EDGES(C_P) \mid (x, X), (y, Y) \in NODES(J_P)\} \).

A more concrete representation of the join graph algorithm in the form of pseudo-code has been provided in listing Procedure 10. Excluding the first line, where the atom collapse is computed, this procedure runs in \( O(k^2 + k \cdot d) \) time for a BGP of size \( k \), where \( d \) represents the time required to resolve the decision point at line 9.

5.3 Query plans

We have previously defined a query plan as a tree consisting of physical operators as its nodes. For this definition, please refer to Section 4.2.2 in Chapter 4.
CHAPTER 5. QUERY OPTIMIZATION OVER THE TRIPLET ENGINE

**Procedure 10:** ComputeJoinGraph($P$)

1. $C_P \leftarrow \text{ComputeAtomCollapse}(P)$
2. $SEEDS \leftarrow$ a new set
3. $TODO \leftarrow$ a new set
4. **repeat**
5.  |   if $TODO = \emptyset$ then
6.  |      $SEEDS = \{(x, X) \in NODES(C_P) \mid x \text{ is an atom}\}$
7.  |   else
8.  |      $SEEDS = \{(x, X) \in NODES(C_P) \mid x \text{ is an atom} \land X \subseteq TODO\}$
9.  |     $(s, S) \leftarrow$ a seed node chosen from $SEEDS$
10.  |     $TODO.\text{Remove}(S)$
11.   |forall the nodes $(x, X) \in NODES(C_P)$ do
12.   |     if $(x, X) \neq (s, S)$ then
13.   |       $X.\text{Remove}(S)$
14.   |       if $X = \emptyset$ then
15.   |         $NODES(C_P).\text{Remove}((x, X))$
16.   |       else if $x$ is a variable then
17.   |         $TODO.\text{Insert}(X)$
18. until $TODO = \emptyset$
19. $J_P \leftarrow$ a new join graph
20. $NODES(J_P) \leftarrow NODES(C_P)$
21. $EDGES(J_P) \leftarrow \{(x, X) - (y, Y) : L \in EDGES(C_P) \mid (x, X), (y, Y) \in NODES(J_P)\}$
22. return $J_P$

### 5.3.1 Examples

We describe a number of example BGPs and their respective query plans.

1. Let $P = (a, b, ?x)_1$. Then a possible plan for this query is:

   ![S-Scan](a, b, ?x)

2. Let $P = (a, b, ?x)_1 \land (?x, a, a)_2$. Then a possible plan for this query is:
3. Let $P = (a, b, ?x)_1 \land (?x, c, ?y)_2 \land (?y, d, e)_3$. Then a possible plan for this query is:

4. Let $P = (a, ?x, ?y)_1 \land (?y, ?x, a)_2$. Then a possible plan for this query is:
5.3.2 Computing the query plan

We present an algorithm for computing a possible query plan $Q_P$ of BGP $P$. Because multiple distinct, valid query plans may be possible for $P$, the algorithm involves decision points, where a choice needs to be made. Such a choice can be made based on any set of rules that are independent from the algorithm itself.

At a high level, the (non-optimized) algorithm consists of the following steps:

1. Let $J_P$ be a join graph of $P$.
2. Decision point: let $((x, X) - (y, Y) : L)$ be an edge chosen from $J_P$.
3. If node $(x, X)$ has not yet been annotated with an operator, then decision point: let $O_X$ be a new Scan-operator which will retrieve results for SAP $X$, and annotate node $(x, X)$ such that $(x, X) : O_X$. If node $(x, X)$ has been annotated with an operator, then let $O_X$ be this operator. Do the same for $(y, Y)$ and let $O_Y$ be its operator.
4. Decision point: let $O_J$ be a new join operator with operators $O_X$ and $O_Y$ as its input.
5. Let $(\_, \_): O_J$ be a new, empty node annotated with operator $O_J$, and insert $(\_, \_): O_J$ into $NODES(J_P)$.
6. Remove $((x, X)-(y, Y) : L)$ from $EDGES(J_P)$ and remove $(x, X)$ and $(y, Y)$ from $NODES(J_P)$.
7. For all edges $((u, U)-(v, V): L') \in EDGES(J_P)$, if $(u, U) = (x, X)$ or $(u, U) = (y, Y)$ then modify $(u, U)$ such that $(u, U) = (\_, \_): O_J$. Do the same for $(v, V)$.
8. If $EDGES(J_P) \neq \emptyset$, then continue at step 2.
9. Let $(w, W) : O$ be the single remaining node in $NODES(J_P)$. We now return query plan $Q_P$ such that $Root(Q_P) = O$.

A more concrete representation of the join graph algorithm in the form of pseudo-code has been provided in listing Procedure 11. Note that for the sake of simplicity, we do not consider the case where there are no joins anywhere in the plan (such as with Cartesian product- or single-SAP BGPs). Excluding the first line, where the join graph is computed, this procedure runs in $O(k^2 + k \cdot (d_1 + d_2))$ time for a BGP of size $k$, where $d_1$ and $d_2$ represent the time required to resolve the decision points at lines 3 and 14, respectively.
5.4 Rules

As we have seen so far, computation of a query plan $Q_P$ starting from a BGP $P$ involves four distinct decision points:

1. Deciding which seed node to select from the collapse graph $C_P$ of $P$.
2. Deciding which join edge to select from a join graph $J_P$ of $P$.
3. Deciding which join type to select for the selected join edge.
4. Deciding what scan to select for a given SAP.

Decision point 3 is resolved by a fixed set of heuristics. For decision points 1, 2, and 4, there is a variable, ordered list of rules $R$ which act as filters and which are applied in sequence on the set of options in order to arrive at a final choice. Each rule $r \in R$ reduces the set of options $I$ to a set of options $I' \subseteq I$ through filter step $I \xrightarrow{r} I'$. Any items in $I'$ are then said to be equivalent under $r$. 

```plaintext
Procedure 11: ComputeQueryPlan($P$)
1. $J_P \leftarrow ComputeJoinGraph(P)$
2. while $EDGES(J_P) \neq \emptyset$ do
3.   $(x, X) - (y, Y) : L) \leftarrow$ an edge chosen from $J_P$
4.   if node $(x, X)$ is annotated with an operator then
5.     $O_X \leftarrow$ operator of $(x, X)$
6.   else
7.     $O_X \leftarrow$ a new Scan-operator for SAP $X$
8.     annotate $(x, X)$ with operator $O_X$
9.   if node $(y, Y)$ is annotated with an operator then
10.    $O_Y \leftarrow$ operator of $(y, Y)$
11.   else
12.    $O_Y \leftarrow$ a new Scan-operator for SAP $Y$
13.    annotate $(y, Y)$ with operator $O_Y$
14.   $O_J \leftarrow$ a new join operator joining $O_X$ and $O_Y$
15.   $(_, _) \leftarrow$ a new, empty node
16.   annotate $(_, _) \leftarrow$ operator $O_J$
17.   $EDGES(J_P).Remove(((x, X) - (y, Y) : L))$
18.   $NODES(J_P).Remove((x, X))$
19.   $NODES(J_P).Remove((y, Y))$
20.   forall the edges $((u, U) - (v, V) : L') \in EDGES(J_P)$ do
21.     if $(u, U) = (x, X)$ or $(u, U) = (y, Y)$ then
22.       $(u, U) \leftarrow (_, _)$
23.     if $(v, V) = (x, X)$ or $(v, V) = (y, Y)$ then
24.       $(v, V) \leftarrow (_, _)$
25.     $NODES(J_P).Insert((_, _))$
26.   $O \leftarrow$ operator annotated to the one remaining node in $NODES(J_P)$
27.   $Q_P \leftarrow$ a new query plan
28.   $Q_P.Root \leftarrow O$
29. return $Q_P$
```
Similarly, an ordered list of rules $R$ performs filtering step $I \xrightarrow{R} I'$, with any items remaining in $I'$ being called equivalent under $R$.

The generic algorithm for making a choice out of a set of items $I$ using rule list $R$ consists of the following steps:

1. Let $R = r_1, r_2, \cdots, r_k$, and let $R' = R \setminus r_k$.
2. Perform filtering step $I \xrightarrow{R'} I'$.
3. Perform filtering step $I' \xrightarrow{r_k} I''$ and force rule $r_k$ to make the final choice out of the options remaining in $I''$.

A more concrete representation of the ruled-choice algorithm in the form of pseudo-code has been provided in listing Procedure 12.

### Procedure 12: MakeChoice(items, rules, context)

```pseudocode
for i = 0 to rules.Count do
    if items.Count = 1 then
        return items[0]
    else if i = rules.Count - 1 then
        // this is our last rule, so force it to make a choice
        item ← rules[i].ChooseOne(items, context)
        return item
    else
        // filter the set of items
        rules[i].Filter(items, context)
```

#### 5.4.1 A note on statistics

We have previously listed a number of statistics on the dataset that are assumed to be available to the plan generator at the time of plan computation. These statistics are used by a number of the rules at various decision points. For a detailed description of the statistics we use, please refer to Section 3.5.1 in Chapter 3.

#### 5.4.2 Decision point 1: Selecting a seed node

This decision point occurs during computation of a join graph. The set of options $I$ consists of a number of potential seeds, which are nodes from collapse graph $C_P$:

$$ I = \{(a, A), (b, B), \cdots, (z, Z)\} \subseteq NODES(C_P). $$

The context consists of the current state of the procedure, e.g. current values of seed- and todo lists, collapse $C_P$, etc.
5.4. RULES

Rule 1: Positional prioritization

This rule selects one preferred seed for each distinct SAP in the input set. It comes into play in those cases that a single SAP contains multiple atoms and therefor has multiple potential seeds. In those cases, we need to select one of those atoms as our seed. We do this based on the positions of the atoms in the SAP, and follow the following ordering:

\[ s \succ o \succ p. \]

Hence, if an SAP contains an atom in the \( s \) position, we will use this atom as our seed. If not, we will use the \( o \)-atom if it exists, and if not then we will use the \( p \)-atom.

Rule 2: Statistical prioritization

This rule selects one preferred seed for each distinct SAP in the input set. It comes into play in those cases that a single SAP contains multiple atoms and therefor has multiple potential seeds. In those cases, we need to select one of those atoms as our seed. We do this based on the expected selectivity of the atoms in the SAP.

Let’s say the SAP has two atoms, \( a_1 \) and \( a_2 \), in positions \( p_1 \) and \( p_2 \), such that:

\[ sel(a_1, p_1) \leq sel(a_2, p_2). \]

We will then choose the most selective atom as the seed for the SAP, i.e. atom \( a_1 \).

5.4.3 Decision point 2: Selecting a join edge

This decision point occurs during computation of a query plan. The set of options \( I \) consists of a number of potential join edges, which are the edges from the current join graph \( J_P \):

\[ I = EDGES(J_P). \]

The context consists of the current state of the procedure, e.g. the current join graph \( J_P \) and the (partially) generated query plan so far.

Rule 1: Merge join maximization

This rule targets both joins between two SAPs and joins involving other joins (i.e. those joins that involve either scan operators or join operators as input). This rule aims to prioritize those joins where it is possible to perform a merge join.

Let’s say we have a candidate join between two operators \( O_{left} \) and \( O_{right} \), and let \( V_{left} \) and \( V_{right} \) denote the variables of their respective output bindings. Now let \( V' = V_{left} \cap V_{right} \) be the set of shared (join) variables. This rule will then prioritize those joins where the output of both \( O_{left} \) and \( O_{right} \) is sorted on \( V' \), and a merge join is hence possible.
CHAPTER 5. QUERY OPTIMIZATION OVER THE TRIPLET ENGINE

Rule 2: Pattern prioritization

This rule targets joins between two SAPs (i.e. those joins that only feature scan operators as input). This rule aims to prioritize those joins that involve the most selective SAPs. In general, we assume SAPs with more atoms are more selective than SAPs with less atoms (and more variables). We follow the following ordering, from most selective to least selective:

\[(s, p, o) \succ (s, ?, o) \succ (s, p, ?) \succ (?, p, o) \succ (?, ?, o) \succ (?, p, ?) \succ (?, ?, ?).\]

Observe that a (candidate) join involves two SAPs. Hence, the selection process involves first selecting the most selective SAP \(P_{\text{left}}\) out of all candidates, followed by a selection of the most selective SAP \(P_{\text{right}}\) that can join with \(P_{\text{left}}\). In both cases we follow the ordering listed above.

Rule 3: Positional prioritization

This rule targets joins between two SAPs (i.e. those joins that only feature scan operators as input). This rule aims to prioritize joins between two SAPs that have the most selective positioning of join variables. In general, we assume a join is more selective when the join variables are in different positions within the two SAPs, than when the join variables are in the same position in both SAPs. We follow the following ordering, from most selective to least selective:

\[s \bowtie p \succ o \bowtie p \succ s \bowtie o \succ s \bowtie s \succ o \bowtie o \succ p \bowtie p.\]

Rule 4: URIs versus literals

This rule targets joins between two SAPs (i.e. those joins that only feature scan operators as input). This rule aims to prioritize those joins that involve the most selective SAPs. When considering atoms, we assume that literals are more selective than URIs. Hence, when selecting from a set of SAPs that involve both URIs and literals, the one(s) with the most literals will be preferred.

Rule 5: Expected output minimization

This rule targets both joins between two SAPs and joins involving other joins (i.e. those joins that involve either scan operators or join operators as input). This rule aims to prioritize those joins that have the smallest expected output size. The expected output sizes are computed using the available statistical information.

If we have a candidate join between two operators \(O_{\text{left}}\) and \(O_{\text{right}}\), we compute the estimated output of \(O_{\text{left}} \bowtie O_{\text{right}}\) (as discussed in Section 3.5.2 in Chapter 3), and prioritize those joins with the smallest estimated output size.

5.4.4 Decision point 3: Selecting a join type

This decision point occurs during computation of a query plan. The input consists of two operators \(O_{\text{left}}\) and \(O_{\text{right}}\), and the output consists of a join operator that takes \(O_{\text{left}}\) and \(O_{\text{right}}\) as its input.
The actual decision point involves choosing between a *merge join* or a *hash join*. Although work in [21] suggests that the hash join preferable as far as CPU costs are concerned, we are primarily optimizing for I/O efficiency, where we expect merge joins to perform better. Hence, the heuristic we follow is relatively straightforward: we select merge joins whenever we can. Let \( V_{\text{left}} \) and \( V_{\text{right}} \) denote the variables of the output bindings of \( O_{\text{left}} \) and \( O_{\text{right}} \), respectively. Now let \( V' = V_{\text{left}} \cap V_{\text{right}} \) be the set of shared (join) variables. Then the decision is made as follows: if the output of both \( O_{\text{left}} \) and \( O_{\text{right}} \) is sorted on \( V' \) then we produce a merge join; if not we produce a hash join.

### 5.4.5 Decision point 4: Selecting a scan operator

This decision point occurs during computation of a query plan. The *input* consists of a single SAP \((s, p, o)\), and the *output* consists of a scan operator for that SAP.

Though it occurs at a later stage of the plan generation process, this decision point is identical to Decision point 1, and hence the same rules are used. An exception is made for when the SAP contains no atoms, in which case we choose any bucket (either S, P, or O) which contains full triples and which will be iterated in full by the scan operator.

### 5.5 Example query

As a concrete example of the query generation process, consider the following query as it makes its way through the optimization framework:

\[
(a, b, ?x)_1 \land (?x, c, ?y)_2 \land (?y, d, e)_3
\]

When this query is fed into the framework, the first thing that is being done is computation of the atom collapse, which has been visualized in Figure 5.1. For the sake of clarity, the edge labels have been omitted.

![Figure 5.1: The visualized atom collapse for a query.](image)

We see the individual atoms and variables of the SAPs as the seed nodes in the graph, the lines between them indicating their respective SAPs share a variable. After the atom collapse has been
computed, a join graph is obtained from it. This is the first moment where a decision point comes into play, and the rules for selecting a seed node determine what the join graph ends up looking like. For the sake of this example a possible join graph has been visualized in Figure 5.2, where we illustrate how the join graph is indeed a subgraph of the atom collapse.

![Figure 5.2: The visualized join graph for a query.](image)

The last step involves producing a physical query plan from the computed join graph. Here, the remaining decision points (two, three, and four) occur, the most prominent one being decision point two, where the rules for selecting a join edge determine the order of joins as they will appear in the physical query plan. For this example, a possible query plan has been visualized in Figure 5.3.

![Figure 5.3: The visualized physical plan for a query.](image)

We see how the SAPs $(a, b, ?x)$ and $(?x, c, ?y)$ are joined together first using a merge join operator, and the results produced there are then joined with those from the remaining SAP $(?y, d, e)$ using a hash join operator.
5.6 Chapter summary

In this chapter we have presented our proposal for a generic query optimization framework for BGP queries over the TripleT engine. We have introduced the three steps that are taken to deliver a query plan for a given BGP: first from BGP to atom collapse, then from atom collapse to join graph, and lastly from join graph to query plan. We have discussed each of these steps in detail, and have presented algorithms for the computations that need to be done to take each one. As we have seen, on several occasions during computation a choice needs to be made, and we have talked about how each of these choices can be manipulated by one or more rules, which are separate from the algorithms themselves. We have concluded this chapter with a description of each of these so-called decision points, and have presented a number of rules for guiding the choices that need to be made, with their aim being the minimization of overall query execution time.
To put our query optimization techniques to the test we have designed a number of experiments. In this chapter, we discuss our experimental goals, setup, and the datasets we have used. Discussion of the results that have been obtained follows in Chapter 7.

6.1 Goals

The primary purpose of the experiments is to gather evidence relevant to answering the following questions:

1. How effective is each individual rule for generating optimized query plans?
2. How effective are combinations of rules for generating optimized query plans?
3. Does the order in which rules are applied matter?
4. What is the impact of using statistics?
5. How do our optimization techniques perform under different kinds of datasets?
6. How do our optimization techniques perform under different types of queries?

These questions can be divided into four sections: the value of rules, the value of statistics, the difference between datasets, and the difference between queries. We discuss these points in more detail next.

6.1.1 The value of rules

Our optimization techniques make use of a number of different rules, which are applied in a certain sequence when we arrive at a decision point where a choice needs to be made. Most of them work based on some heuristic. One would not expect each rule to be just as effective as the next; in fact, such would be a highly surprising outcome. Instead, one would expect there to be noticeable differences in the effectiveness of individual rules. One would also expect that certain combinations of rules will prove to be highly effective, more so than what the sum of the parts might suggest. The ordering of rules, e.g. in what order they are applied at a decision point, would be expected to matter to a certain degree but be less important than which rules are and are not being used.
In our experiments, we gather data that will let us attach a certain value to each rule that indicates its effectiveness at generating optimized query plans. We also identify the most effective combinations and/or orderings of rules to apply.

### 6.1.2 The value of statistics

Although we have described only two rules which make use of statistics, their purpose is the same as that of all of our heuristics-based rules: to minimize intermediate result sizes produced during query plan execution. The use of these statistics-based rules comes at a cost: a full statistics database needs to be computed over the dataset that the rules are to be used on. Depending on the size of the dataset this process can potentially take up to several days, as we have experienced when working with datasets containing ten million triples. And even then there is the non-trivial problem of keeping the statistics database up-to-date should the underlying dataset be modified.

Although the cost of inserting our test datasets into a TripleT store (including construction of the statistics database) is not something we are directly interested in when performing our experiments, it is still something to keep in the back of our minds when evaluating the performance of statistics-based rules. We are of course interested in the performance of the statistics rules by themselves, though primarily we look at the comparison to heuristics-based rules. We also investigate the performance of statistics rules combined with heuristics rules. Overall, one might expect statistics-based rules to be a valuable addition to the heuristic-based rules, though their stand-alone value might significantly less. One would expect statistics rules to perform more consistently, as they can work on all levels of a query plan rather than just at the leaves where structural information is available.

### 6.1.3 Different datasets

As discussed later on in this chapter, we use a number of different datasets in our experiments. We use nine different datasets in total, from three different sources and in three different sizes. The datasets come from both real-world and synthetically generated data. As such, one might expect there to be differences in the effectiveness of various rules between the datasets, especially involving the highly specialized UniProt dataset.

### 6.1.4 Different queries

Also discussed in a later section in this chapter are the different types of queries we use in our experiments. As our datasets are essentially graphs and our queries are graph patterns, it’s easy to visualize them as such. In Figure 6.1 the four common query shapes that our queries are based around are shown, where a query’s SAPs are represented by nodes which are connected if they share a variable.

Aside from their shape, other variables are query size (in number of SAPs), and query selectivity. The influence of query size on execution time is difficult to predict. On one hand, more SAPs means more joins; on the other, more SAPs can also mean higher selectivity which can be exploited by the plan generator. As for query selectivity, a query which features more atoms in more selective positions in its SAPs generally produces a smaller result set. Again, selective SAPs in a query can be favorably exploited by the plan generator.
6.2 Setup

In this section we discuss the contents of our experiments. We talk about what kind of experiments we run, how we do so, and what we measure. We also talk briefly about the environment our experiments are executed in.

6.2.1 Variables

In our experiments we primarily compare different rule sets to each other. Hence, the composition of the rule set is the main variable in each experimental run. We define a run in our experiment as testing one particular rule set on every dataset using every available query. Here, testing some rule set \( x \) on dataset \( y \) using query \( z \) is comprised of:

1. Opening the TripleT database for dataset \( y \).
2. Telling the query plan generator to use rule set \( x \).
3. Feeding query \( z \) to the database.
4. Enumerating and immediately discarding the query results.
5. Closing the database.

Each run is tested five times, each time “cold”, i.e. without preserving any caches between tests. The runs we have performed are detailed in Table 6.1, where a number indicates that a particular rule was used in that run, the number itself indicating the order (a lower number denoting a higher priority). The names of the rules correspond to the rules described in Section 5.4 in Chapter 5, e.g. Join-3 corresponds to join rule 3 on positional prioritization. The runs have the following purpose:

- The A-runs are designed to test the heuristics rules against the statistics rules.
- The B-runs aim to get a sense of value for the individual heuristics rules.
- The C-runs focus on the ordering of rules.
- The D-runs are used to test different subset combinations of rules.

6.2.2 Measurements

For each test that is run, several data points are collected. The primary unit of measurement is time: how long does it take to retrieve the results produced by a query? When comparing two runs, this is what we will be looking at. A shorter execution time indicates this particular rule set is more effective at generating optimized query plans. We only measure the time actually spent
retrieving results: open/closing the database, retrieving internal representations of atoms through the database dictionary, and generating the query plan are all ignored as these are not the things we are interested in or have attempted to optimize. Internally, we measure elapsed time using .NET’s System.Diagnostics.Stopwatch class, which provides an extremely high accuracy with a resolution of several million ticks per second.

Aside from the query execution time, each test also produces the full query plan which was used for that particular test and in which each individual operator is annotated with information denoting the total time spent on I/O operations and CPU operations (separately measured). To this end, each operator internally uses the Stopwatch class to accurately measure how much time it’s spending on I/O- and CPU work.

### 6.2.3 Environment

All experiments are run on a dedicated testing environment: a machine which is not used for anything except testing. The machine features an Intel Core 2 Quad Q9400 processor at 2666.7 MHz, two Samsung PC2-6400 DDR2-SDRAM UDIMM main memory modules at 2 GB each and running at 400 MHz for a total of 3.72 GB of usable RAM, and a Seagate ST3160318AS harddrive weighing 152627 MB and spinning at 7200 RMP. The harddrive features two partitions: one 76009 MB partition containing the operating system and user- and program files, and one 76261 MB partition containing TripleT and our testing data. The operating system is Microsoft Windows 7 Enterprise (x64) build 7601.

### 6.3 Datasets

We run our experiments on a total of nine different datasets. The data itself comes from three different sources: DBpedia¹, SP²Bench², and UniProt³. From each source we have obtained three

---

¹ [http://dbpedia.org/](http://dbpedia.org/)
² [http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B](http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B)

---

<table>
<thead>
<tr>
<th>Run</th>
<th>Seed-1</th>
<th>Seed-2</th>
<th>Join-1</th>
<th>Join-2</th>
<th>Join-3</th>
<th>Join-4</th>
<th>Join-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>A-2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A-3</td>
<td>1</td>
<td>2</td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>A-4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>B-1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D-1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D-2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D-3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Run / Rule set table
different sized datasets: one 100,000 (100K) triples dataset, one 1,000,000 (1M) triples dataset, and
one 10,000,000 (10M) triples dataset. For all of our datasets, the 100K set is a strict subset of the 1M
set, which in turn is a strict subset of the 10M set. We discuss each dataset in a little more detail
next.

6.3.1 DBpedia

The DBpedia dataset is a general knowledge base which sources its data from Wikipedia\(^4\). Its goal is
to make the information from Wikipedia available on the Web in a structured manner, allowing for
sophisticated queries and links to other knowledge bases. DBpedia uses RDF as its data description
format. Due to its nature DBpedia contains information on a broad spectrum of topics, as can be
expected from an encyclopedia.

We have acquired our data from DBpedia 3.8, at the time the latest version. We have not used the
entire DBpedia dataset, which contains roughly 1.89 billion RDF triples. Instead, we have used only
the English-language “Ontology Infobox Properties” dataset, which contains data extracted from
Wikipedia infoboxes. This dataset contains roughly 20.5 million RDF triples, out of which we have
extracted ten million for use in our experiments. This extraction procedure was semi-random: for
query and testing purposes we made sure data from a number of large domains was present, and
complemented this with randomly selected triples. Some characteristics of the resulting DBpedia
datasets are shown in Table 6.2. Of particular note is the large number of distinct predicate values,
which reflects the many different domains present in DBpedia.

<table>
<thead>
<tr>
<th></th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique atoms (total)</td>
<td>72,019</td>
<td>585,668</td>
<td>5,027,090</td>
</tr>
<tr>
<td>Unique S-atoms</td>
<td>13,496</td>
<td>152,057</td>
<td>2,029,424</td>
</tr>
<tr>
<td>Unique P-atoms</td>
<td>272</td>
<td>318</td>
<td>340</td>
</tr>
<tr>
<td>Unique O-atoms</td>
<td>61,941</td>
<td>469,596</td>
<td>3,420,561</td>
</tr>
<tr>
<td>Bucket size (per bucket)</td>
<td>1,53 MB</td>
<td>15,26 MB</td>
<td>152,59 MB</td>
</tr>
<tr>
<td>Index DB size</td>
<td>10,07 MB</td>
<td>79,92 MB</td>
<td>648,55 MB</td>
</tr>
<tr>
<td>Statistics DB size</td>
<td>40,96 MB</td>
<td>340,08 MB</td>
<td>2805,24 MB</td>
</tr>
</tbody>
</table>

Table 6.2: DBpedia dataset characteristics

6.3.2 SP\(^2\)Bench

The SP\(^2\)Bench SPARQL performance benchmark was designed to serve as a standardized benchmark
suite used for evaluating performance of RDF stores and SPARQL query engines. It consists of a
generator used for creating synthetic datasets of arbitrary size, and a set of SPARQL queries. A
comprehensive technical report is also available [31].

We have used SP\(^2\)Bench version 1.00 for Windows to generate our datasets. More specifically, we
have called

```
sp2b.exe -t 10000000 sp2b-10M.n3
```

to generate the 10M dataset. The 1M and 100K sets are then both strict subsets of this 10M set.
Some figures of the resulting datasets are available in Table 6.3.

\(^4\)http://www.wikipedia.org/
CHAPTER 6. EXPERIMENT SETUP

<table>
<thead>
<tr>
<th></th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique atoms (total)</td>
<td>61.165</td>
<td>593.446</td>
<td>5.582.841</td>
</tr>
<tr>
<td>Unique S-atoms</td>
<td>19.369</td>
<td>187.066</td>
<td>1.730.250</td>
</tr>
<tr>
<td>Unique P-atoms</td>
<td>58</td>
<td>67</td>
<td>77</td>
</tr>
<tr>
<td>Unique O-atoms</td>
<td>50.680</td>
<td>494.586</td>
<td>4.690.662</td>
</tr>
<tr>
<td>Bucket size (per bucket)</td>
<td>1.53 MB</td>
<td>15.26 MB</td>
<td>152.59 MB</td>
</tr>
<tr>
<td>Index DB size</td>
<td>9.86 MB</td>
<td>79.99 MB</td>
<td>652.61 MB</td>
</tr>
<tr>
<td>Statistics DB size</td>
<td>41.06 MB</td>
<td>343.56 MB</td>
<td>2861.59 MB</td>
</tr>
</tbody>
</table>

Table 6.3: SP²Bench dataset characteristics

6.3.3 UniProt

UniProt is a specialized scientific dataset which aims to provide a freely accessible resource with information involving protein sequences. UniProt distributions are available in a number of formats, ranging from XML and plain text to the domain-specific FASTA format. An RDF distribution is also provided.

Our data has been obtained from UniProt release 2012_11. The RDF distribution of UniProt is contained in a single XML file measuring hundreds of gigabytes in size. Because of the difficulty in dealing with a single file of such size, and because we only required a fraction of the billions of triples contained within, we opted for extracting only the first gigabyte of data from the gzip archive the distribution is provided in. We then opened the extracted partial file in a suitable editor and manually repaired the XML by removing the final incomplete entries and closing any open tags. From this we then extracted the first 10 million triples for use in our experiments. Some of the features of the resulting datasets are provided in Table 6.4. Note that compared to the DBpedia and SP²Bench datasets, UniProt features significantly less distinct atoms, resulting in smaller index- and statistics database sizes.

<table>
<thead>
<tr>
<th></th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique atoms (total)</td>
<td>31.940</td>
<td>272.355</td>
<td>2.295.259</td>
</tr>
<tr>
<td>Unique S-atoms</td>
<td>18.937</td>
<td>191.537</td>
<td>1.813.570</td>
</tr>
<tr>
<td>Unique P-atoms</td>
<td>89</td>
<td>101</td>
<td>103</td>
</tr>
<tr>
<td>Unique O-atoms</td>
<td>25.996</td>
<td>211.959</td>
<td>1.627.303</td>
</tr>
<tr>
<td>Bucket size (per bucket)</td>
<td>1.53 MB</td>
<td>15.26 MB</td>
<td>152.59 MB</td>
</tr>
<tr>
<td>Index DB size</td>
<td>4.92 MB</td>
<td>39.64 MB</td>
<td>320.23 MB</td>
</tr>
<tr>
<td>Statistics DB size</td>
<td>21.66 MB</td>
<td>178.30 MB</td>
<td>1465.35 MB</td>
</tr>
</tbody>
</table>

Table 6.4: UniProt dataset characteristics

6.4 Queries

For each of the three dataset sources (DBpedia, SP²Bench, and UniProt) we have prepared a query set, featuring a number of queries according to the four shapes shown in Figure 6.1. For each query shape we have attempted to find four queries: a small selective query, a large selective query, a small non-selective query, and a large non-selective query. Finding meaningful loop queries proved particularly difficult, and as such we only have those for the DBpedia sets. For a complete overview of all queries used in our experiments, please refer to Appendix A.
6.5 Chapter summary

In this chapter we have discussed the setup of the experiments we have designed to assess the performance of our query optimization techniques. We have posed a number of experimental goals, in the form of a series of questions that our experiments are designed to answer. We continued with a discussion of our methods, and we have presented each of the different datasets which are involved. We concluded with a brief description of the benchmark queries that we have used.
Empirical results

In this chapter we discuss in detail the results that have been obtained in our experiments, which we have set up in Chapter 6.

Due to the high number of variables in our experiments, a large amount of raw data is produced on each run. As such, we initially provide only high-level overviews of our results, and then go into more detail where necessary or of particular interest. First, we present an overview of the performance of all runs on the 1M datasets, shown in Table 7.1. Here, for each query shape (Chain, Star, etc.) on each dataset, the average query answer time is shown over all runs for all query types (Small-Selective, Large-Selective, etc.). We use this table as the primary starting point of our results discussion.

<table>
<thead>
<tr>
<th>DBpedia 1M</th>
<th>SP²Bench 1M</th>
<th>UniProt 1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain</td>
<td>Star</td>
<td>Star-chain</td>
</tr>
<tr>
<td>Run A-1</td>
<td>4251.6</td>
<td>1689.2</td>
</tr>
<tr>
<td>Run A-2</td>
<td>5336.6</td>
<td>3148.2</td>
</tr>
<tr>
<td>Run A-3</td>
<td>3280.5</td>
<td>1578.1</td>
</tr>
<tr>
<td>Run A-4</td>
<td>5227.3</td>
<td>2716.6</td>
</tr>
<tr>
<td>Run B-1</td>
<td>6258.6</td>
<td>3599.9</td>
</tr>
<tr>
<td>Run B-2</td>
<td>5231.5</td>
<td>1739.6</td>
</tr>
<tr>
<td>Run B-3</td>
<td>4465.9</td>
<td>1577.5</td>
</tr>
<tr>
<td>Run B-4</td>
<td>4217.4</td>
<td>1708.9</td>
</tr>
<tr>
<td>Run C-1</td>
<td>5581.1</td>
<td>3047.4</td>
</tr>
<tr>
<td>Run C-2</td>
<td>5232.1</td>
<td>2740.6</td>
</tr>
<tr>
<td>Run C-3</td>
<td>5676.8</td>
<td>2311.1</td>
</tr>
<tr>
<td>Run D-1</td>
<td>4357.4</td>
<td>1309.9</td>
</tr>
<tr>
<td>Run D-2</td>
<td>4672.9</td>
<td>2339.1</td>
</tr>
<tr>
<td>Run D-3</td>
<td>5299.9</td>
<td>3437.8</td>
</tr>
</tbody>
</table>

Table 7.1: Results overview on 1M datasets

7.1 Heuristics versus statistics

The A-runs were designed to test the performance of our heuristics-based rules (represented by runs A-1 and A-3) against our statistics-based rules (represented by runs A-2 and A-4). A slightly more detailed overview of these runs is provided in Table 7.2, where we summarize not only on query shape, but also provide an additional table summarizing on query type (selectivity).

From Table 7.2 at first glance it appears the heuristics-based rules offer better performance overall, and Figure 7.1 confirms this trend over the different dataset sizes. Here, we plot the average query execution time over all queries on all datasets, along with the average standard deviation.
### Chapter 7. Empirical Results

#### Table 7.2: Results overview on 1M datasets of the A-runs

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Run A-1</th>
<th>Run A-2</th>
<th>Run A-3</th>
<th>Run A-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBpedia 1M</td>
<td>4251.6</td>
<td>5336.6</td>
<td>3280.5</td>
<td>5227.3</td>
</tr>
<tr>
<td>SP²Bench 1M</td>
<td>1689.2</td>
<td>3418.2</td>
<td>1578.1</td>
<td>2716.6</td>
</tr>
<tr>
<td>UniProt 1M</td>
<td>6121.8</td>
<td>9437.5</td>
<td>5685.3</td>
<td>9160.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Run A-1</th>
<th>Run A-2</th>
<th>Run A-3</th>
<th>Run A-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBpedia 1M</td>
<td>18918.9</td>
<td>20223.7</td>
<td>18516.3</td>
<td>18121.6</td>
</tr>
<tr>
<td>SP²Bench 1M</td>
<td>6929.06</td>
<td>6098.2</td>
<td>7862.02</td>
<td>6293.4</td>
</tr>
<tr>
<td>UniProt 1M</td>
<td>10715.9</td>
<td>6987.5</td>
<td>142607.8</td>
<td>198520.0</td>
</tr>
</tbody>
</table>

#### Figure 7.1: Results overview of the A-runs

A more detailed view, per dataset, is provided in Figure 7.2. It should be noted that for runs A-2 and A-4, the star-chain queries failed to complete within reasonable time on the UniProt 10M dataset, and have therefore been omitted in the results discussion. For the A-2 run this means we are excluding 20 out of the 200 data points gathered for the 10M sets (all UniProt star-chain queries failed); for the A-4 run we exclude 15 out of 200 data points (all UniProt star-chain queries except the selective-small ones failed).

#### Figure 7.2: Per-dataset results overview for the A-runs

For both the DBpedia and the SP²Bench datasets, the heuristics-based rules seem to be most effective, with the differences between runs A-1, A-3 and A-2, A-4 reaching almost up to an order of magnitude on the SP²Bench sets. On UniProt the differences seem less pronounced, though again it should be noted that A-2 and A-4 failed to scale up to the 10M set for the star-chain query. Here, a combination
of primarily heuristics rules with statistics as back-up seems to work best, as seen in performance of run A-3, though the difference with run A-1 is small. On virtually all datasets runs A-3 and A-4, which use a combination of heuristics and statistics, perform more consistently, as evident by the smaller error bars compared to runs A-1 and A-2.

One interesting comparison is between the star-chain queries performed by runs A-3 and A-4 on the SP2Bench 1M dataset. In Figures 7.3 and 7.4 we show two plans, generated during runs A-3 and A-4, respectively. This selective query produces no results in the end, though intermediate result sizes can grow quite large. In this case, we see how the A-4 plan of minimizing intermediate result sizes through statistics backfires as the join order it chooses in doing so introduces many hash joins into the plan. Hash joins, ironically, require all input data to be read before producing results themselves; while the merge joins used by the A-3 plan are lazy, and only consume input data when needed. More importantly, using information on the sort order of its input data, the merge join “knows” when no more join matches are possible and it can stop consuming input. This allows the merge join to only consume an absolute minimum of data, which in the case of this highly selective query proves key to a quick response time.
The one query where the statistics-based rules outperformed the heuristics-based ones by an order of magnitude is of course also interesting to investigate in a little more detail. Hence, we take a closer look at the A-1 and A-2 runs on the UniProt 1M dataset. In Figures 7.5 and 7.6 we show two plans, generated during runs A-1 and A-2, respectively. This is again a selective query, which produces no results in the end but has significant intermediate result sizes. More specifically, the two “stars” of the star-chain query do produce results, but the “chain” part that eventually joins them together does not. The A-1 plan, which is produced using heuristics that try to maximize the number of merge joins, reflects the shape of the star-chain query, with two separate “star” parts that are joined together using merge joins, and a “chain” part that uses a hash join to link the two stars together. Because the “star” parts produce significant results, this approach presents the final hash join with a large amount of data, which does not fit in main memory and hence takes a significant I/O-cost to process. On the other hand, the left-deep A-2 plan effectively minimizes its intermediate results by performing the most selective joins early rather than at the very end, resulting in a response time almost two orders of magnitude smaller than the A-1 plan.
Figure 7.5: Run A-1, UniProt 1M, selective-large star-chain query
### 7.2 The value of heuristics rules

The B-runs were designed to get a sense of value for each individual heuristics rule. To this end, we have four runs, where in each run we disable one of the heuristics rules while enabling the remaining rules in a fixed order. We then measure the impact of each rule’s absence on query response time. A slightly more detailed overview of these runs is provided in Table 7.3, where we summarize not only on query shape, but also provide an additional table summarizing on query type (selectivity).
7.2. THE VALUE OF HEURISTICS RULES

<table>
<thead>
<tr>
<th></th>
<th>DBpedia 1M</th>
<th>SP²Bench 1M</th>
<th>UniProt 1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain</td>
<td>Star</td>
<td>Star-chain</td>
<td>Loop</td>
</tr>
<tr>
<td>Run B-1</td>
<td>6258.6</td>
<td>3559.0</td>
<td>9222.4</td>
</tr>
<tr>
<td></td>
<td>2629.8</td>
<td>18607.4</td>
<td>95575.0</td>
</tr>
<tr>
<td>Chain</td>
<td>Star</td>
<td>Star-chain</td>
<td>Loop</td>
</tr>
<tr>
<td>Run B-2</td>
<td>5231.5</td>
<td>1739.6</td>
<td>6131.6</td>
</tr>
<tr>
<td></td>
<td>2315.3</td>
<td>10281.8</td>
<td>5686.9</td>
</tr>
<tr>
<td>Selective</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run B-3</td>
<td>4465.9</td>
<td>1577.5</td>
<td>6078.1</td>
</tr>
<tr>
<td></td>
<td>2213.4</td>
<td>10358.0</td>
<td>5141.3</td>
</tr>
<tr>
<td>Run B-4</td>
<td>4217.4</td>
<td>1708.9</td>
<td>6043.9</td>
</tr>
<tr>
<td></td>
<td>3344.8</td>
<td>10404.7</td>
<td>5258.2</td>
</tr>
<tr>
<td>Non-selective</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run B-1</td>
<td>3983.1</td>
<td>6856.8</td>
<td>6311.4</td>
</tr>
<tr>
<td></td>
<td>74405.4</td>
<td>3966.4</td>
<td>19850.9</td>
</tr>
<tr>
<td>Run B-2</td>
<td>2682.0</td>
<td>5027.0</td>
<td>2769.2</td>
</tr>
<tr>
<td></td>
<td>12585.1</td>
<td>18582.2</td>
<td>17784.7</td>
</tr>
<tr>
<td>Run B-3</td>
<td>2453.1</td>
<td>4714.3</td>
<td>2709.7</td>
</tr>
<tr>
<td></td>
<td>12254.5</td>
<td>17222.4</td>
<td>19320.5</td>
</tr>
<tr>
<td>Run B-4</td>
<td>2646.7</td>
<td>4505.8</td>
<td>2725.1</td>
</tr>
<tr>
<td></td>
<td>12326.3</td>
<td>19608.0</td>
<td>39949.0</td>
</tr>
</tbody>
</table>

Table 7.3: Results overview on 1M datasets of the B-runs

From Table 7.3 we see that in general, the merge join prioritization rule seems to provide the biggest positive contribution. Figure 7.7 shows this trend over all dataset sizes. It should be noted that for run B-1, only the selective-small star-chain queries could be completed within reasonable time on the UniProt 10M dataset, which explains the small standard deviation seen for this run on the 10M datasets as the remaining star-chain queries, if completed, are often performed less consistently. In the same run the nonselective star-chain queries also failed to complete in time on the SP²Bench 10M set.

A more detailed view, per dataset, is provided in Figure 7.8. Here we see a similar trend for both the DBpedia and SP²Bench datasets. Especially the on the SP²Bench set query response times suffer when the merge join prioritization rule is absent. On the UniProt dataset the differences between runs are less pronounced, and even seem to suggest that the merge join prioritization rule (run B-1) has the least amount of positive impact on execution time, while the literal prioritization rule (run B-4) makes the most positive difference, especially on the 1M set.

The results shown on the UniProt dataset seem to counter the trend shown by the rest of the datasets, so we take a closer look at some of the results. In Figure 7.9 we show the results of the B-runs on the UniProt 1M dataset, individually for each query shape. Two things stand out: the star query on run B-4, and the star-chain query on run B-1.

Looking first at the star query on the B-4 run, we find the cause for the poor response time in the large non-selective version of this query, where on two of the five tests run a “bad” plan is produced. In Figure 7.10 one of the “good” plans produced during this run is shown; Figure 7.11 shows one of the “bad” plans. These different plans for the same query are produced because the rules used by the plan generator for determining join order can not operate above the leaf level in a query plan.
where no structural information is available for them to use. Hence, at that level making a choice (i.e. choosing an item from a set) is performed in a non-deterministic manner. In this case, a subtle difference in join ordering makes all the difference in the world. For the first plan, the massive result set this query produces is only produced on relatively small inputs by the final join operator, which does not need to store these results anywhere. For the second plan, the final join operator needs to process an input set as large as the output set in order to join in a final variable. Because the input set is too large to keep in main memory, the merge join needs to use its secondary memory buffer, which incurs a great I/O cost, significantly raising response time for the overall plan.
7.2. THE VALUE OF HEURISTICS RULES

Looking next at the star-chain query for run B-1, it seems that the absence of the merge join prioritization rule provides a significant performance boost for this particular query. It turns out that indeed the remaining runs (B-2, B-3, and B-4), which do feature the merge join prioritization rule,
take a performance hit purely because they use this rule. In Figure 7.12 a plan produced in run B-1 is shown; in Figure 7.13 a plan from run B-2 is shown which is representative of runs B-3 and B-4 as well. We see how the B-1 plan, generated primarily by the pattern prioritization rule, produces only a small amount of intermediate results, and features only negligible I/O costs. On the other hand, the B-2 plan, in its desire to maximize the amount of merge joins, presents the final hash join with a great deal of intermediate results as its input, which incurs a large I/O cost. In this way, the B-1 plan answers the query an order of magnitude faster than the B-2 plan.

Figure 7.12: Run B-1, UniProt 1M, selective-large star-chain query
7.3 Rule ordering

The C-runs were designed to measure the importance of the order in which rules are applied. As there is only one statistical rule these runs look at the heuristics rules only. We have three runs, and each rule is in a different position in each run. A slightly more detailed overview of these runs is provided in Table 7.4, where we summarize not only on query shape, but also provide an additional table summarizing on query type (selectivity).

<table>
<thead>
<tr>
<th>Database</th>
<th>Query Type</th>
<th>C2-1</th>
<th>C2-2</th>
<th>C2-3</th>
<th>C2-1</th>
<th>C2-2</th>
<th>C2-3</th>
<th>C2-1</th>
<th>C2-2</th>
<th>C2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBpedia 1M</td>
<td>Chain</td>
<td>5581.1</td>
<td>5232.1</td>
<td>5676.8</td>
<td>SP²Bench 1M</td>
<td>Chain</td>
<td>3547.9</td>
<td>3908.0</td>
<td>3551.0</td>
<td>UniProt 1M</td>
</tr>
<tr>
<td>Star</td>
<td>3047.4</td>
<td>2740.6</td>
<td>2311.1</td>
<td>Star-chain</td>
<td>9510.0</td>
<td>9100.0</td>
<td>7014.3</td>
<td>Non-selective</td>
<td>6218.7</td>
<td>3864.9</td>
</tr>
<tr>
<td>Star-chain</td>
<td>2618.7</td>
<td>3864.9</td>
<td>2384.2</td>
<td>Loop</td>
<td>19547.6</td>
<td>19349.5</td>
<td>11936.3</td>
<td></td>
<td>6304.9</td>
<td>5006.5</td>
</tr>
<tr>
<td>Loop</td>
<td>2618.7</td>
<td>3864.9</td>
<td>2384.2</td>
<td>Chain</td>
<td>19547.6</td>
<td>19349.5</td>
<td>11936.3</td>
<td></td>
<td>6304.9</td>
<td>5006.5</td>
</tr>
</tbody>
</table>

Table 7.4: Results overview on 1M datasets of the C-runs

From Table 7.4 we see that in general, the lower the priority given to the merge join prioritization- and pattern prioritization rules, the greater the query response times. In Figure 7.14 this trend is shown over all dataset sizes. It should be noted that the C-1 and C-2 runs completed only the selective star-chain queries on the UniProt 10M dataset within reasonable time.
CHAPTER 7. EMPIRICAL RESULTS

Figure 7.14: Results overview of the C-runs

A more detailed view, per dataset, is provided in Figure 7.15. Here we see a similar trend for both the DBpedia and SP\textsuperscript{2}Bench datasets, which is less pronounced on the DBpedia set and more so on the SP\textsuperscript{2}Bench set. Once more the UniProt set appears to be the odd man out, with a poor performance by the C-2 run on the 1M set, as well as a C-3 run performance which is suddenly significantly worse on the 10M set compared to its performance on the 1M and 100K sets.

Figure 7.15: Per-dataset results overview for the C-runs

The poor performance of the C-2 run on the UniProt 1M set appears to be caused by a bad plan generated for one of the five tests run for the nonselective-large star-chain query. In Figure 7.16 this “bad” plan produced during the C-2 run is shown; in Figure 7.17 a plan is shown from the same run that is representative of the remaining “good” plans. Though this is a nonselective query, it does not produce any results in the end. Intermediate results can potentially get extremely large, as is shown in the bad plan, where an unfortunate choice in join ordering presents a hash join operator with an extremely large input set, which causes a great deal of I/O cost to process. In the end, the “bad” query plan is up to three orders of magnitude slower than the remaining plans for the same query, again stressing the importance of minimizing intermediate result sizes as early on in a query plan as possible.
7.3. RULE ORDERING

Figure 7.16: Run C-2, UniProt 1M, nonselective-large star-chain query

Figure 7.17: Run C-2, UniProt 1M, nonselective-large star-chain query
When we look at the performance of the C-3 run on the UniProt 10M set, we see that the performance penalty is due to a cause we have discussed previously for the B-runs: the influence of the merge join prioritization rule, which is greatest in the C-3 run, causes more merge joins to be produced in the plan, which for this particular query has a strong negative effect on response time. This effect is most visible in the 10M dataset, in which large intermediate result sets incur a great I/O cost.

Another interesting stand-out, seen in Table 7.4, is the performance of the C-1 run on the star-chain query of the SP²Bench dataset. In Figures 7.18 and 7.19 we show two plans, generated during runs C-1 and C-2, respectively. Here, we see how the C-1 plan features hash joins near the leaves of the plan, where they are forced to deal with large intermediate result sets, which hash joins need to read in full before producing any output of their own. Because at this point the intermediate result sets are as large as they are, this processing does not fit in main memory, and hence features a great I/O cost. The C-2 plan, on the other hand, by relying on merge joins up until the very top of the query plan tree, manages to deal with these large intermediate result sets without any I/O cost at all. This results in a response time which is up to two orders of magnitude faster.
7.4 Rule subsets

In the D-runs we look at a number of configurations which use a limited subset of rules. There are
three runs, each one using a single seed rule and at most three join rules. A slightly more detailed
overview of these runs is provided in Table 7.5, where we summarize not only on query shape, but
also provide an additional table summarizing on query type (selectivity).

<table>
<thead>
<tr>
<th>DBpedia 1M</th>
<th>SP²Bench 1M</th>
<th>UniProt 1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain</td>
<td>Star</td>
<td>Loop</td>
</tr>
<tr>
<td>Run D-1</td>
<td>3357.4</td>
<td>1320.9</td>
</tr>
<tr>
<td>Run D-2</td>
<td>4672.9</td>
<td>2329.1</td>
</tr>
<tr>
<td>Run D-3</td>
<td>5299.9</td>
<td>3437.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DBpedia 1M</th>
<th>SP²Bench 1M</th>
<th>UniProt 1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selective</td>
<td>Non-selective</td>
<td>Selective</td>
</tr>
<tr>
<td>Run D-1</td>
<td>2365.1</td>
<td>3835.8</td>
</tr>
<tr>
<td>Run D-2</td>
<td>3514.0</td>
<td>4631.8</td>
</tr>
<tr>
<td>Run D-3</td>
<td>3950.6</td>
<td>6285.4</td>
</tr>
</tbody>
</table>

Table 7.5: Results overview on 1M datasets of the D-runs

From Table 7.5 we observe that although certain runs perform consistently better than others, the
differences are fairly small and contained within the same order of magnitude. The only exception
is the performance of run D-1 on star-chain query of the UniProt set, as well as selective queries in
general on that set. In Figure 7.20 we show the performance over all dataset sizes. It should be noted
that the D-3 run completed only the selective-small star-chain queries on the UniProt 10M dataset
within reasonable time.
CHAPTER 7. EMPIRICAL RESULTS

A more detailed view, per dataset, is provided in Figure 7.21. Interestingly, these seem to tell a slightly different story. The D-1 run seems to be the most effective on both the DBpedia and SP²Bench datasets, while performing significantly worse on the UniProt sets. Indeed, as suggested by the overview in Figure 7.20, the D-2 run appears to provide the best, consistent performance over all three datasets. This is somewhat surprising, as this configuration consists of only three heuristics rules (one seed rule, two join rules), and does not use statistics at all.

The poor performance of the D-1 run on the UniProt sets is primarily caused by its performance on the star- and star-chain queries. For the star-chain query, the cause is identical to what we have discussed before: merge join prioritization (used as primary rule in the D-1 run) on this particular query presents the final hash join at the root of the plan with a large set of intermediate results, which takes great I/O cost to process. For the star queries, a little more interesting things are going on. The costs are mainly found in the nonselective-large star query, which is what we will be looking at. In Figure 7.22 we show a plan for this query produced during the D-1 run; in Figure 7.23 a plan is shown for the same query that was produced by the D-2 run. At a first glance it is obvious that the reason the D-1 plan is slower is because the merge join at the root of the plan spends a significant amount of time doing I/O work to process a large intermediate result set. In the D-2 plan there is no such problem, though strangely enough the merge join at the root of this plan has no associated I/O costs at all while dealing with a similarly sized input set. The reason for this is that internally, the merge join only needs to buffer one of its two input sets. In our implementation of TripleT, we always buffer the left input set. Ideally of course we would always like to buffer the smaller of the two input sets, though as we have seen throughout this investigation accurately estimating intermediate result set sizes is a hard problem even with the use of statistics as we have. More importantly, the impact of left/right ordering of input sets for joins is not something we have attempted to optimize or measure.

Figure 7.20: Results overview of the D-runs

Figure 7.21: Per-dataset results overview for the D-runs

Figure 7.22: Plan for the nonselective-large star query produced during the D-1 run

Figure 7.23: Plan for the nonselective-large star query produced during the D-2 run
7.4. RULE SUBSETS

in these experiments, though these query results provide a compelling case for doing so in the future.

Figure 7.22: Run D-1, UniProt 1M, nonselective-large star query
7.5 Merging versus hashing

An important assumption made by our heuristics rules used for query plan generation is that when given a choice, performing a merge join is always preferable to using a hash join, and thus the plan generator should attempt to maximize the number of merge joins. To see if we can validate this assumption, we provide Figure 7.24, where we plot for all 8316 queries performed in all test runs, the query execution time versus the fraction of join operators in the query plan that were merge joins.

Figure 7.24: The performance of joins
In this graph we see that 66.0% of the fastest performing plans (those with execution times between $10^0$ and $10^3$ milliseconds) feature a large fraction (above 0.5) of merge joins. That is not to say that having a large fraction of merge joins is a guarantee for fast plan execution: out of those plans with execution times greater than $10^3$ milliseconds 42.7% has a majority of merge joins, and for the slower plans (those with an execution time of $10^4$ and above) this number becomes 44.0%. There is even a number of plans using only merge joins with execution times between $10^5$ and $10^6$ milliseconds.

Another interesting data point in this debate is the amount of time the two join operators tend to spend on CPU- and I/O work. We first look at the CPU work the operators perform. In Figure 7.25 we plot a histogram of CPU work done by the hash- and merge join operators over all 8316 queries performed in all test runs. The bins of the histogram are the time spent doing CPU work; on the y-axis the number of hash- and merge joins in each bin is plotted. It should be noted that we plot data from 15090 hash join operations and 27547 merge join operations. Proportionally we see that there are more merge joins than hash joins performing small (less than 500 milliseconds) amounts of CPU work, but above this threshold both joins show a similar distribution. If we look at the 95% interior of both distributions we find that the mean time spent on CPU work is 106.2 milliseconds for the hash join and 32.9 milliseconds for the merge join. This finding seems to contradict [21], where it was suggested the hash join was preferable over the merge join in terms of CPU efficiency.

![Figure 7.25: Histogram of CPU performance of join operators](image)

For the I/O work done by the two join operators we have a similar plot in Figure 7.26, except this time we plot time spent on I/O operations, rather than CPU operations. Here we observe a significant amount of hash join operators performing non-trivial I/O work, versus virtually no merge join operators. The difference is even more obvious when we look at the 95% interior of both distributions: here we find that the mean time spent on I/O work is 258.5 milliseconds for the hash join and 0.0 milliseconds for the merge join. It is only in exceptional cases, when it is confronted with a large input set, that the merge join performs any I/O work at all.
CHAPTER 7. EMPIRICAL RESULTS

7.6 Can we beat an ape?

While we have spent a considerable amount of time comparing rule configurations against other rule configurations, an interesting question might be: what if we don’t use any rules at all? We performed a run to test just that. No rules were enabled, and all decision points that occurred during plan generation were handled by making a random choice from the set of available options. Figure 7.27 shows the results, with the ape representing randomized choice.

The ape shows worse performance across all datasets. The performance is also far more erratic, as evident by the large error bars present. On the UniProt 1M set, the ape failed to complete even a single star-chain query within a reasonable time: the first attempt at resolving a selective-small star-chain query on this dataset was manually aborted after 11 hours and 30+ GB of disk space wasted on intermediate result storage. Results for the ape on the 10M datasets are not available due to time constraints.

Looking at the graph, the differences between the ape and the A-1 and A-2 sets might seem relatively small. Here it should be noted that the ape is at least an order of magnitude slower than our best rule configurations, that its performance is erratic and unpredictable, and that the ape failed to scale in a linear fashion on the 10M datasets. In the end, we confidently conclude that our rule sets provide significant improvement over randomized choice, both in response time and in scalability.
7.7 Discussion

We have seen in our experiments how the value of individual rules used by the plan generator can vary greatly. Especially the merge join prioritization rule appears to be invaluable for the generation of efficient query plans, which is unsurprising considering that in general a merge join incurs virtually zero I/O cost, meaning it produces its results faster than when using hash joins. The remaining three heuristics rules sit closer together as far as their value is concerned, with the impact of their absence on overall query response time being roughly the same. Of course, as we have seen the positive impact of the merge join prioritization rule is not universal: on the UniProt datasets, performance generally improved when this rule was disabled. Here, the literal prioritization rule proved to have the largest positive impact. While often not applicable, prioritizing literals (when present) over URIs is a quick way of minimizing intermediate result sizes.

In line with our observation that certain rules are more valuable than others, we have seen how configurations featuring only a subset of the available rules can still produce highly efficient query plans. The biggest surprise in this regard was the configuration used in the D-2 run, which uses only three heuristics rules (one seed rule, two join rules), no statistics, and provides exceptional and consistent performance over all datasets, with no negative outliers. Only the configuration used in the C-3 run, of which D-2 is a strict subset, manages to deliver a similar performance. While configurations primarily focusing on merge join prioritization (such as the one used in run A-1) offer slightly better performance in many cases, the differences are very small, and prioritizing merge joins is shown to result in bad response times on the UniProt star-chain queries.

The order in which heuristics rules are applied matters to a certain degree, though this is closely related to the value of rules that we have observed. We have seen that in general, the lower the priority that is being placed on the merge join prioritization rule, the poorer query response times become. As seen in runs C-3 and D-2, only the pattern prioritization rule can have a higher priority than the merge join rule, in order to approach query response times seen when the merge join rule is used as the primary heuristic.

We have observed how the impact of statistics and the statistical prioritization join rule is measurable but minimal. When used alone or as the primary join rule, the statistics rule produces query plans significantly worse than those produced by the heuristics rules. Indeed, it appears that the value of the statistics rule is found mostly in a supporting role, such as how it is used in the configuration of run D-1. Here, it does have a positive impact for a significant number of queries, though at no point does it affect the order of magnitude of the response times that we observe. Although we have mentioned that we are not directly interested in the costs associated with constructing and maintaining statistics databases, at this point a very relevant question to ask is whether statistics are “worth it”.

Another interesting thing to observe is how the performance of our rules is affected by the different datasets that we have used. We have seen how especially the UniProt set had a tendency to produce results that countered those seen on the DBpedia and SP2Bench sets. Primarily the merge join prioritization rule seemed to do more harm than good when used on the UniProt sets. This was mostly caused by UniProt’s star-chain queries, which were extremely slow to process using plans that prioritized merge joins, and we have seen more than one run in which those queries failed to complete on the largest 10M dataset. Indeed, the star-chain query wins the award for “most difficult query”, with response times often varying in multiple orders of magnitude between rule configurations.

In the end, we recommend the rule configuration used in the D-2 runs for use in a BGP query plan generator, as it is a purely heuristic and minimal approach which delivers excellent results across the board. Overall, our findings corroborate results obtained in [33], where a heuristics-based planner for SPARQL queries is shown to be competitive with the cost-based approach taken in the RDF-3X
store. At this point, we feel the added value of the statistics rule for query optimization on TripleT
does not outweigh the upkeep of a statistical database.

7.8 Chapter summary

In this chapter we have discussed in great detail the extensive experiments that we have performed to
assess the effectiveness of our plan generator and its rules. We have seen how a few simple heuristics
rules can deliver excellent performance in creating optimized query plans. Especially the prioritization
of merge joins is shown to be a key factor in efficient query plan generation. We have also seen how
different datasets respond differently to our rules, and that merge join prioritization is not always
the right answer. We have also discussed the value of statistics and the statistical prioritization rule,
where we have seen that while the statistical rule provides a positive added value, it should not be
used as the primary guideline when constructing query plans. We concluded that the statistical rule
is better served complementing key heuristics rules, though we also feel that at this point the cost
of constructing and maintaining a statistical database is not proportional to the added value the
statistics rule provides.
In this thesis we have discussed the problem of Basic Graph Pattern query optimization on the TripleT RDF store and index. We have proposed a query optimization framework that takes the shape of a generic, rule-based algorithm. We presented a number of rules for use by this algorithm: a number of heuristical rules, as well as two rules using dataset statistics. These rules guide the algorithm responsible for generating an evaluation plan for a given query, and jump into action at a number of fixed points at which a decision needs to be made. The most important one of such decisions is the order in which to join together the intermediate result sets that are produced when executing a query plan. We have seen that this join order can have a massive impact on the time it takes to evaluate a plan: two plans for the same query can have response times several orders of magnitude apart.

We have presented an implementation of the TripleT store, and have implemented our optimization framework on top of that. Using this implementation, we have evaluated the various rules used by the framework in a series of experiments. These experiments have shown that a number of relatively simple heuristics can consistently produce efficient query evaluation plans. Especially the maximization of merge joins proved to have a great positive impact to this end, though we have also seen that merge joins are not effective on every dataset or query type. We have seen how the use of statistics provides a non-trivial challenge, especially for use in intermediate result size estimation. We have concluded that while statistics do provide added value, this value is minimal, and not within reasonable proportion to the costs involved in constructing and maintaining a statistical database over a dataset.

8.1 Future work

A number of avenues might be worth pursuing in future work. A comparison of TripleT and our query optimization framework against other RDF stores is a natural next step. Related to this would be investigating the effect of using six buckets to store the data in TripleT, rather than the minimal three we have used. An implementation of a sideways information passing technique such as discussed in [27] for query evaluation on TripleT would also be an interesting extension. Several other aspects of TripleT may also be optimized, such as the insertion of data into the buckets, the construction of the index, and maintaining both in the face of later insertions, mutations or deletions of triples. In Chapter 3 we have also discussed some challenges we have encountered when using statistics for query optimization, and how relatively few work exists in this area with relation to RDF data management, and we provide some pointers for further work in Section 3.6. Especially intermediate result size estimation, key during generation of query plans, proved to be a hard problem with no
directly applicable solutions provided by the state-of-the-art. Obviously additional work in this area would be most interesting, and may yet help our optimization framework produce even more efficient query plans. Computing the statistical database over an existing dataset also provided a challenge in need of optimization, and may benefit from a better integration into the data management process.
A complete listing of all queries used in the experiments of our investigation is provided below. For each query we list the SAPs that make up this query, with each SAP on its own line. In each SAP, atomic values are presented as strings and are placed between quotes; variables are denoted as a question mark followed by a number.

## A.1 DBpedia

The chain queries for the DBpedia sets are:

<table>
<thead>
<tr>
<th>DBpedia: chain query, small-selective</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DBpedia: chain query, large-selective</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DBpedia: chain query, small-nonselective</th>
</tr>
</thead>
</table>
APPENDIX A. QUERIES

DBpedia: chain query, large-nonselective

(?1, “http://xmlns.com/foaf/0.1/name”, ?6)

The star queries for the DBpedia sets are:

DBpedia: star query, small-selective

(?1, “http://dbpedia.org/ontology/birthDate”, ?3)  

DBpedia: star query, large-selective

(?1, “http://dbpedia.org/ontology/birthDate”, ?3)  

DBpedia: star query, small-nonselective

(?1, “http://dbpedia.org/ontology/birthDate”, ?3)  

DBpedia: star query, large-nonselective

(?1, “http://dbpedia.org/ontology/birthDate”, ?3)  

The star-chain queries for the DBpedia sets are:
DBpedia: star-chain query, small-selective

(?1, "http://dbpedia.org/ontology/birthPlace", "http://dbpedia.org/resource/London")
(?1, "http://dbpedia.org/ontology/birthDate", ?3)
(?1, "http://dbpedia.org/ontology/deathPlace", "http://dbpedia.org/resource/London")
(?1, "http://dbpedia.org/ontology/deathDate", ?5)
(?6, "http://dbpedia.org/ontology/birthPlace", ?7)
(?6, "http://dbpedia.org/ontology/birthDate", ?8)
(?6, "http://dbpedia.org/ontology/deathPlace", ?9)
(?6, "http://dbpedia.org/ontology/deathDate", ?10)

DBpedia: star-chain query, large-selective

(?1, "http://dbpedia.org/ontology/birthPlace", "http://dbpedia.org/resource/London")
(?1, "http://dbpedia.org/ontology/birthDate", ?3)
(?1, "http://dbpedia.org/ontology/deathPlace", ?4)
(?1, "http://dbpedia.org/ontology/deathDate", ?5)
(?1, "http://dbpedia.org/ontology/nationality", ?11)
(?1, "http://dbpedia.org/ontology/parent", ?12)
(?6, "http://dbpedia.org/ontology/birthDate", ?8)
(?6, "http://dbpedia.org/ontology/deathPlace", ?9)
(?6, "http://dbpedia.org/ontology/deathDate", ?10)
(?1, "http://dbpedia.org/ontology/nationality", ?13)
(?1, "http://dbpedia.org/ontology/parent", ?14)

DBpedia: star-chain query, small-nonselective

(?1, "http://dbpedia.org/ontology/birthPlace", ?2)
(?1, "http://dbpedia.org/ontology/birthDate", ?3)
(?1, "http://dbpedia.org/ontology/deathPlace", ?4)
(?1, "http://dbpedia.org/ontology/deathDate", ?5)
(?6, "http://dbpedia.org/ontology/birthPlace", ?7)
(?6, "http://dbpedia.org/ontology/birthDate", ?8)
(?6, "http://dbpedia.org/ontology/deathPlace", ?9)
(?6, "http://dbpedia.org/ontology/deathDate", ?10)

DBpedia: star-chain query, large-nonselective

(?1, "http://dbpedia.org/ontology/birthPlace", ?2)
(?1, "http://dbpedia.org/ontology/birthDate", ?3)
(?1, "http://dbpedia.org/ontology/deathPlace", ?4)
(?1, "http://dbpedia.org/ontology/deathDate", ?5)
(?1, "http://dbpedia.org/ontology/child", ?6)
(?8, "http://dbpedia.org/ontology/birthPlace", ?9)
(?8, "http://dbpedia.org/ontology/birthDate", ?10)
APPENDIX A. QUERIES

The loop queries for the DBpedia sets are:

<table>
<thead>
<tr>
<th>DBpedia: loop query, small-selective</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DBpedia: loop query, large-selective</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DBpedia: loop query, small-nonselective</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DBpedia: loop query, large-nonselective</th>
</tr>
</thead>
</table>

A.2 SP²Bench

The chain queries for the SP²Bench sets are:

<table>
<thead>
<tr>
<th>SP²Bench: chain query, small-selective</th>
</tr>
</thead>
</table>
A.2. SP²BENCH

**SP²Bench: chain query, large-selective**


**SP²Bench: chain query, small-nonselective**


**SP²Bench: chain query, large-nonselective**


The star queries for the SP²Bench sets are:

**SP²Bench: star query, small-selective**

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”,  
“http://localhost/vocabulary/bench/Inproceedings”)  
(?1, “http://purl.org/dc/elements/1.1/creator”, “Bhoomika_Secord”)  

**SP²Bench: star query, large-selective**

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”,  
“http://localhost/vocabulary/bench/Inproceedings”)  
(?1, “http://purl.org/dc/elements/1.1/creator”, “Bhoomika_Secord”)  
(?1, “http://www.w3.org/2000/01/rdf-schema#seeAlso”, ?6)  
The star-chain queries for the SP²Bench sets are:

SP²Bench: star-chain query, small-selective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://localhost/vocabulary/bench/Article”)  
(?5, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://localhost/vocabulary/bench/Article”)  
A.3 UniProt

The chain queries for the UniProt sets are:

UniProt: chain query, small-selective

(\(?1, \text{"http://purl.uniprot.org/core/encodedBy"}, ?2\)
(\(?2, \text{"http://purl.uniprot.org/core/locusName"}, ?3\)
("http://purl.uniprot.org/uniprot/P0C9G1", \"http://purl.uniprot.org/core/classifiedWith\", ?1)

UniProt: chain query, large-selective

(\(?1, \text{"http://www.w3.org/1999/02/22-rdf-syntax-ns\#type"}, \http://purl.uniprot.org/core/Protein\)
(\(?1, \text{"http://purl.uniprot.org/core/encodedBy"}, ?2\)
(\(?2, \text{"http://purl.uniprot.org/core/locusName"}, ?3\)
("http://purl.uniprot.org/uniprot/P0C9G1", \"http://purl.uniprot.org/core/classifiedWith\", ?1)

UniProt: chain query, small-nonselective

(\(?1, \text{"http://purl.uniprot.org/core/encodedBy"}, ?2\)
(\(?2, \text{"http://purl.uniprot.org/core/locusName"}, ?3\)}
APPENDIX A. QUERIES

UniProt: chain query, large-nonselective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)  
(?1, “http://purl.uniprot.org/core/encodedBy”, ?2)  

The star queries for the UniProt sets are:

UniProt: star query, small-selective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)  
(?1, “http://purl.uniprot.org/core/modified”, ?2)  

UniProt: star query, large-selective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)  
(?1, “http://purl.uniprot.org/core/modified”, ?2)  
(?1, “http://purl.uniprot.org/core/citation”, ?3)  
(?1, “http://purl.uniprot.org/core/encodedBy”, ?5)  

UniProt: star query, small-nonselective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)  
(?1, “http://purl.uniprot.org/core/modified”, ?2)  
(?1, “http://purl.uniprot.org/core/citation”, ?3)  

UniProt: star query, large-nonselective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)  
(?1, “http://purl.uniprot.org/core/modified”, ?2)  
(?1, “http://purl.uniprot.org/core/citation”, ?3)  
(?1, “http://purl.uniprot.org/core/encodedBy”, ?5)  
(?1, “http://purl.uniprot.org/core/isolatedFrom”, ?6)  

The star-chain queries for the UniProt sets are:
UniProt: star-chain query, small-selective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)
(?1, “http://purl.uniprot.org/core/modified”, ?2)
(?1, “http://purl.uniprot.org/core/citation”, ?4)

UniProt: star-chain query, large-selective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)
(?1, “http://purl.uniprot.org/core/modified”, ?2)
(?1, “http://purl.uniprot.org/core/isolatedFrom”, ?9)
(?1, “http://purl.uniprot.org/core/citation”, ?4)

UniProt: star-chain query, small-nonselective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)
(?1, “http://purl.uniprot.org/core/modified”, ?2)
(?1, “http://purl.uniprot.org/core/citation”, ?4)

UniProt: star-chain query, large-nonselective

(?1, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, “http://purl.uniprot.org/core/Protein”)
(?1, “http://purl.uniprot.org/core/modified”, ?2)
(?1, “http://purl.uniprot.org/core/isolatedFrom”, ?9)
(?1, “http://purl.uniprot.org/core/citation”, ?4)
We provide a brief overview of the TripleT source code, as well as the programming API used for interacting with a TripleT database.

B.1 Overview

All functionality for the TripleT system is contained within a single .NET class library, called TripleT.dll. The source code is fully commented and documented. All classes are distributed over a number of relevant namespaces. We will discuss those next.

**TripleT**

This is the root namespace of the TripleT library. It only features one class: Database. This class contains the primary API used for interacting with the system, which will be discussed later on in this appendix.

**TripleT.Algorithms**

This namespace contains most of the algorithms used in TripleT, including algorithms for comparing objects, external sorting, and computing the atom collapse. The algorithm used for query plan generation is also found here.

This namespace contains a child namespace, TripleT.Algorithms.Rules, which contains the seed- and join rules, as well as an implementation of the decision engine which is used by the plan generator for applying the rules at various decision points.

**TripleT.Compatibility**

Several classes for importing data from external sources are found in this namespace. More specifically, the functionality contained here deals with reading RDF triples from a variety of formats, such as Notation3 or XML.
**TripleT.Datastructures**

In this namespace we find the internal data structures used by TripleT. This includes elementary
data structures such as triples and bindings, as well as complicated ones such as the database index,
the statistics database, and the atom dictionary.

There are also several child namespaces which contain a number of data structures for specific domains,
such as join graphs or query plans.

**TripleT.IO**

Functionality for interacting with external data is contained in this namespace. We find classes for
dealing with the reading and writing of triples from and to disk, as well as main- and secondary-memory
buffers.

The child namespace **TripleT.IO.Operators** contains implementations of all physical operators used
in a TripleT query plan.

**TripleT.Reporting**

This namespace contains classes for outputting data gathered during the experiments that we have run. Most importantly it contains functionality for producing annotated graphs from query plans
that have been executed, for use in our report.

This namespace is irrelevant for operation of the TripleT system.

**TripleT.Test**

Classes for automated batch testing are contained is this namespace, as well as an internal representa-
tion of the query sets we have used in our experiments.

This namespace is irrelevant for operation of the TripleT system.

**TripleT.Util**

Several utility functions are grouped together in this namespace, such as classes for byte encoding of
data, logging, and random data generation. Numerous extension methods for some of the internal
data structures are also contained here.

### B.2 Dependencies

TripleT is implemented on top of version 4.0.30319 SP1Rel of the Microsoft .NET Framework. Aside
from that, TripleT has a dependency on Berkeley DB, version 5.3.21. We do not include Berkeley DB
in our distribution of the TripleT source code. It should be acquired from the official Oracle website\(^1\).
TripleT requires that the correct Berkeley DB libraries (*libdb_csharp53.dll, libdb_dotnet53.dll*,
\[^1\]http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html}
and `libdb53.dll` can be located, either through the operating system’s `PATH` variable or because they are in the same directory as the executable using TripleT.

## B.3 Programming API

There is no explicitly defined programming API in our implementation of TripleT, though all functionality for interacting with a TripleT data store is contained in the `TripleT.Database` class. In this section we show how to perform basic operations on a TripleT database.

### B.3.1 Inserting data

Inserting data into a TripleT store can occur through multiple channels. We support insertion of single triples:

```csharp
using (var db = new Database("my_database")) {
    db.Open();
    // inserting a triple
    db.Insert("Bart\Wolff", "wrote", "TripleT");
    // inserting another triple as a System.Tuple
    var t = Tuple.Create("TripleT", "is", "quite\nice");
    db.Insert(t);
    db.Close();
}
```

When inserting multiple triples, batch insertion is often quicker:

```csharp
using (var db = new Database("my_database")) {
    db.Open();
    // inserting many triples
    db.BeginBatchInsert();
    foreach (var t in myTriplesSet) {
        db.Insert(t);
    }
    db.EndBatchInsert();
    db.Close();
}
```

Inserting triples using one of TripleT’s triple readers (found in `TripleT.Compatibility`) is also directly supported:

```csharp
using (var db = new Database("my_database")) {
    db.Open();
    // inserting triples from an external source
    db.BeginBatchInsert();
    using (var tr = new Notation3TripleReader(File.OpenRead("triples.n3"))) {
        db.InsertAll(tr);
    }
    db.EndBatchInsert();
    db.Close();
}
```
B.3.2 Building the index

After inserting all triples into the TripleT store, the index can be built. Doing so should only occur after all triples that should be in the store have been inserted; there is no support for inserting, mutating, or deleting triples on an existing TripleT store after the index has been built. Building the index sorts the buckets, indexes them, and builds the statistics database. A single line of code does the job:

```csharp
using (var db = new Database("my_database")) {
    db.Open();
    // [...] assume all triples have been inserted at this point
    // build the index and compute dataset statistics
    db.BuildIndex();
    db.Close();
}
```

B.3.3 Querying

Executing queries on a TripleT store is done by feeding it Basic Graph Patterns. Querying can only be done after data has been inserted into the store and the index has been built. Consider the following example BGP:

```sql
("Bart Wolff", "wrote", ?x)
(?x, "is" ?y)
```

Querying a TripleT store with this BGP is done as follows:

```csharp
using (var db = new Database("my_database")) {
    db.Open();
    // create the query pattern
    var pattern = new Pattern[
        new Pattern("Bart\_Wolff", "wrote", 1),
        new Pattern(1, "is", 2)
    ];
    // compute the query plan for this BGP
    var plan = db.GetQueryPlan(pattern);
    // execute the query, and enumerate the results
    foreach (var bindingSet in db.Query(plan)) {
        // do something with it
    }
    db.Close();
}
```


