Università degli studi *“Roma Tre”*

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

A SPARQL front-end for MonetDB

Relatore	Correlatore	Laureando
Paolo Atzeni	*Peter A. Boncz*	*Marco Antonelli*
		274455

Anno Accademico 2007/2008

Preface

This thesis starts with a collaboration with the CWI, Center for Mathematics and Informatics (in Dutch: Centrum voor Wiskunde en Informatica), a prestigious Dutch research center located in Amsterdam, one of the most important in Europe in these fields and a member of the ERCIM, the European Research Consortium for Informatics and Mathematics.

One of the research themes at the CWI concerns the problems relative to the "data explosion": how to find relevant information in the increasing amount of the available data?

On this theme one of the research groups of the CWI has been developing since 1994 MonetDB, an open-source database management system specialized in obtaining high performances in query-intensive applications like decision support (OLAP), data mining, geographical information systems (GIS) and XQuery; this DBMS is and has been since its early stages a scientific research platform in the database field.

A new application of relational systems is RDF data storage and query; the goal of this language is to formally express metadata in a *subject-predicate-object* form, making the information that this language describes intelligible to a computer. A web page is now comprehensible only to humans; but if the meaning of that page is expressed in a formal language then it can be automatically processed making content search, for instance, much more effective. RDF constitute thus the foundations of the web of the future, the "Semantic Web".

How relational engines can manage and query effectively considerable amounts of RDF triples, in the order of hundreds of millions, is still the object of a remarkable scientific research effort.

My job at CWI was to kick-start the MonetDB front/end for SPARQL, the RDF query language.

The developed code, in C language, brought on one side to the creation of a new module of MonetDB, described in chapter 5, that defines the relational structures that contain the RDF data and the import and export functions from and to plain textual documents; on the other to a SPARQL parser which, given a query, it translates it to its algebraic form.

The theoretical work concerned the translation of the SPARQL algebra in relational algebra, proposed in the last section of chapter 4.

The first chapter describes RDF, both in its syntax and its semantics; chapter 2 introduces MonetDB, its fundamental principles, its architecture and the data structure on which this DBMS is centered on, the binary table.

Chapter 3 illustrates the general RDF storage techniques and gives an overview of the main projects related to this subject.

Chapter 4 exposes, in examples and formally, the SPARQL query language and its algebra; the last section, referref to above, proposes for each operator of this algebra an equivalent relational expression.

The last chapter describes the data structures used in MonetDB to contain the RDF triples, how these are imported from a textual document, and which are advantages and drawbacks of the suggested solution.

Appendix A, finally, examines a set of choices that MonetDB/SPARQL may perform to get the maximum benefit from the adopted data structures.

Prefazione

Questa tesi nasce da un esperienza di lavoro presso il CWI, Centro per la Matematica e l'Informatica (in olandese: Centrum voor Wiskunde en Informatica), un prestigioso centro di ricerca olandese situato ad Amsterdam, uno dei più importanti in Europa in questi campi e membro dell'ERCIM, il consorzio europeo per l'Informatica e la Matematica, di cui fa parte anche il CNR italiano.

Una delle tematiche di ricerca scientifica presso il CWI riguarda le problematiche relative alla "esplosione dei dati": come trovare informazioni rilevanti nella sempre crescente quantità di informazioni disponibili?

In quest'ambito uno dei gruppo di ricerca del CWI sviluppa sin dal 1994 MonetDB, un DBMS open-source specializzato per ottenere alte prestazioni in applicazioni "query-intensive" come il supporto decisionale (OLAP), il data mining, i sistemi informativi geografici (GIS) e XQuery; questo DBMS è ed è stato per tutti questi anni una piattaforma per la ricerca scientifica nel campo delle basi di dati.

Una nuova applicazione dei sistemi relazionali consiste nell'immagazzinamento e la ricerca di dati espressi in RDF; lo scopo di questo linguaggio è di esprimere in maniera formale dei metadati sotto forma di triple *soggetto-predicato-oggetto*, rendendo in tal modo intellegibile per un calcolatore le informazioni che questo linguaggio descrive. Una pagina web, oggi, è comprensibile solo ad un essere umano; ma se le informazioni contenute sono espresse in un linguaggio formale allora queste possono essere processate automaticamente rendendo le ricerche di contenuti, ad esempio, molto più efficaci. RDF costituisce dunque le fondamenta del web del futuro, il "Web Semantico".

Come possano però riuscire i sistemi relazionali a contenere ed interrogare efficacemente quantità considerevoli di triple RDF, dell'ordine di centinaia di milioni, è ancora oggetto di un notevole sforzo di ricerca.

Il mio compito presso il CWI è stato quello di iniziare il front/end di MonetDB per SPARQL, il linguaggio di interrogazione per RDF.

Lo sviluppo di codice, in linguaggio C, ha portato da una parte alla creazione di un nuovo modulo di MonetDB, descritto nel capitolo 5, che definisce le strutture relazionali per la rappresentazione dei dati RDF e le funzioni di importazione ed

esportazione di documenti in forma testuale; dall'altra ad un parser per SPARQL, che data una query la traduce nella sua forma algebrica.

Il lavoro teorico ha riguardato la traduzione dell'algebra SPARQL in algebra relazionale, proposta nell'ultima sezione del 4° capitolo.

Il 1° capitolo della tesi descrive RDF, sia nella sintassi che nella semantica; il secondo mentre il 2° introduce MonetDB, i sui principi basilari, la sua architettura e la struttura dati sulla quale questo DBMS è incentrato, la tabella binaria.

Il capitolo 3 illustra le tecniche generali di immagazzinamento di RDF ed effettua una panoramica dei principali progetti correlati a questo argomento.

Il quarto capitolo espone sia per esempi sia formalmente il linguaggio di interrogazione SPARQL e la sua algebra; l'ultima sezione, come detto sopra, propone per ogni operatore di quest'algebra una espressione relazionale equivalente.

L'ultimo capitolo descrive le strutture dati utilizzate in MonetDB progettate per contenere le triple RDF, come queste vengano importante da un documento testuale, e quali siano i vantaggi e gli svantaggi della soluzione proposta.

L'Appendice A, infine, esamina un insieme di scelte che MonetDB/SPARQL può intraprendere per trarre il massimo vantaggio dalle strutture dati utlizzate.

Contents

Chapter 1

The Resource Description Framework

1.1 Introduction

The Resource Description Framework [5] is "a language for representing information about resources in the World Wide Web" [38].

RDF is based on the idea that each piece of information is a resource that has properties that have values. The resources can be described, therefore, by a set of *statements* in the subject-predicate-object format: the *subject* is that part of the statement that identifies the Web resource under description, the *predicate* identifies a property of the subject, and the *object* is the value of that property. Because all statements have this structure, they are also called *triples*.

A statement with this simple subject-predicate-object structure may be

```
The page http://example.org/index.html has a creator whose value is John Smith
```

where `http://example.org/index.html` is the subject, `creator` is the predicate and `John Smith` is the object.

1.2 Uniform Resource Identifiers

The above example, however, does not unequivocally identify what the concept of `creator` or who `John Smith` is. Any resource, that might be a web page, a book, a person, or any abstract concept has to be described by an *Uniform Resource Identifier* or *URI*. URIs are a generalization of URLs (Uniform Resource Locators), that identify a resource by its access mechanism. URL are well suited for web pages or mail boxes, but not for any other resource that is not physically accessible on the Web.

The above statement may be represented by an RDF triple having:

- a subject `http://example.org/index.html`
- a predicate `http://purl.org/dc/elements/1.1/creator`
- an object `http://example.org/staffid/85740`

where `http://purl.org/dc/elements/1.1/creator` is a URI that identifies the "creator" concept, and `http://example.org/staffid/85740` unequivocally identifies a specific John Smith.

A further generalization of URIs are *IRIs*, i.e. *Internationalized Resource Identifiers*, that are not restricted to the ASCII character set but allow also Unicode characters. Every URI is also an IRI, and every IRI can be translated to an URI, substituting every non-ASCII character with the equivalent "percent encoding", that consists of a '%' followed by the Unicode codepoint that identifies the character.

1.3 Graph Data Model

Since the object of an RDF statement may be a subject of another triple, a set of statement forms a labeled and directed graph, where subjects and objects are nodes and each predicate is an edge directed from a subject to an object.

Figure 1.1 is a simple RDF graph that extends the above example.

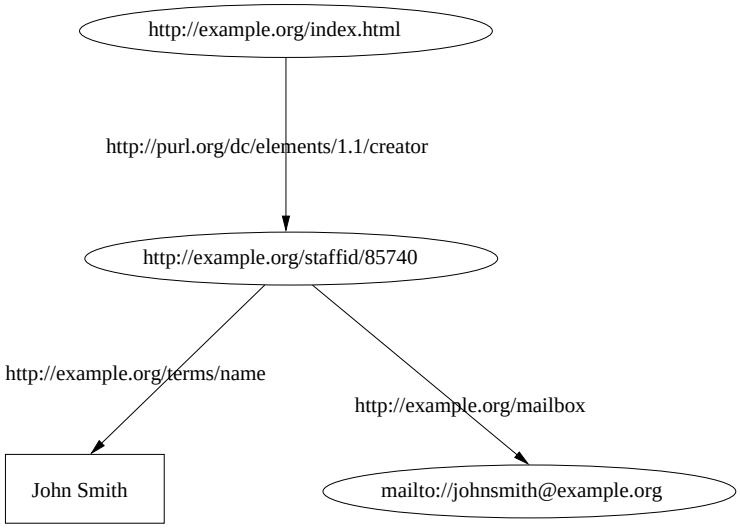

Figure 1.1: Simple RDF graph

The URI `http://example.org/staffid/85740` has two additional properties: the name of the person represented by the URI and his mailbox. As figure 1.1 may suggest, object can be either URIs or constant values, called *literals*. In the figure, literals are shown as boxes, and URIs as ellipses.

1.4 RDF serialization languages

This section introduces the languages to express RDF data in plain text files. It is not intended to be a complete reference, but just an introduction needed to show example data in a rigorous manner; many details will be skipped for the moment and introduced later in this chapter, when necessary.

The recommended standard language is RDF/XML [13], that encodes the triples in the tree structure of XML. Since it is not easily readable for humans, the Notation3 (or N3) [15] language has been developed: the approach of N3 and its dialects, Turtle [14] and N-Triples [29], is to explicitly list the RDF statements one after the other.

1.4.1 Notation3, Turtle and N-Triples

These languages are each a subset of the other, with Notation3 being the largest and N-Triples the smallest; for this reason, and because of the total compatibility of the smaller languages with the larger ones, N3, Turtle and N-Triples are described together.

Each statement of an RDF graph is listed on a different line, terminated by a dot. The subject, the predicate and the object are separated by white spaces, the URIs are written between '<' and '>' characters and the literals are quoted.

The RDF graph in figure 1.1 can expressed with

```
<http://example.org/index.html> <http://purl.org/dc/elements/1.1/creator> <http://example.org/staffid/85740> .
<http://example.org/staffid/85740> <http://example.org/terms/name> "John Smith" .
<http://example.org/staffid/85740> <http://example.org/terms/mailbox> <mailto://johnsmith@example.org> .
```

or more compactly, in Turtle and N3, with

```
@prefix ex: <http://example.org/> .
@prefix exterms: <http://example.org/terms/> .
@prefix exstaff: <http://example.org/staffid/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

ex:index.html dc:creator exstaff:85740 .
exstaff:85740 exterms:name "John Smith" .
exstaff:85740 exterms:mailbox <mailto://johnsmith@example.org> .
```

N3 and Turtle permit one to declare URI prefixes, while N-Triples does not allow it. This language, in fact, was intended as a test-case language, and thus N-Triples documents were not supposed to be written or read by humans.

A URI reference can thus be expressed in N3 and Turtle with a *qualified name*, that consists of a prefix that has been assigned to a namespace URI, a colon and a local name, without angle brackets. The full URI reference is the concatenation of the namespace associated with the prefix and the local name.

1.4.2 URI namespaces used in this thesis

From now on, this thesis will make use of the following "well-known" prefixes to keep URI references short and to avoid repetition:

```
@prefix rdf:    http://www.w3.org/1999/02/22-rdf-syntax-ns#
@prefix rdfs:   http://www.w3.org/2000/01/rdf-schema#
@prefix dc:     http://purl.org/dc/elements/1.1/
@prefix xsd:    http://www.w3.org/2001/XMLSchema#
```

1.4.3 RDF/XML

RDF/XML [13] is the recommended serialization language for RDF, but since N3 and its subsets are easier to read, their use will be preferred for the examples of this thesis.

The graph in figure 1.1 in RDF/XML can be expressed as:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY rdf     "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <!ENTITY ex      "http://example.org/">
    <!ENTITY exstaff "http://example.org/staffid/">
    <!ENTITY exterms "http://example.org/terms/">
    <!ENTITY dc      "http://purl.org/dc/elements/1.1/">
]>
<rdf:RDF    xmlns:rdf     = "&rdf;"
            xmlns:exterms = "&exterms;"
            xmlns:dc      = "&dc;">

    <rdf:Description rdf:about="&ex;index.html">
        <dc:creator rdf:resource="&exstaff;85740"/>
    </rdf:Description>

    <rdf:Description rdf:about="&exstaff;85740">
        <exterms:name>John Smith</exterms:name>
        <exterms:mailbox rdf:resource="mailto://johnsmith@example.org"/>
    </rdf:Description>

</rdf:RDF>
```

The `ENTITY` declarations are shorthand: the string associated with the entity `rdf` can be referenced further in the document by `&rdf;` . The names of the tags are qualified names, and are expanded as in N3; the prefix in this case is declared as an XML namespace (i.e. `xmlns`).

The URI references of a subject of a statement are generally declared in the `rdf:about` attribute of an `rdf:Description` tag, whose internal nodes represent the properties of that subject and their values.

1.5 Blank nodes

Other kinds of nodes that can be found in RDF graphs, together with URI references and literals, are blank nodes. These, unlike literals and like the URIrefs can be both subject and objects, but with the difference that they do not have a universal name; blank nodes are therefore local to an RDF graph.

Blank nodes are frequently used to encapsulate structured data, as shown in figure 1.2 for an address.

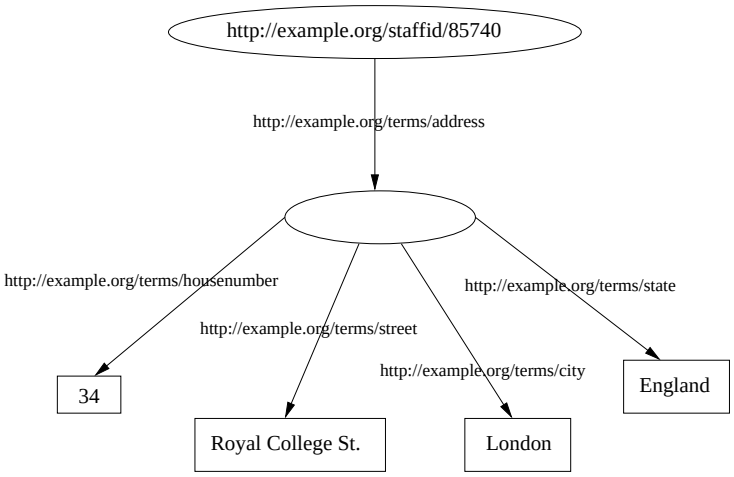

Figure 1.2: A blank node representing an address

In N3 a blank node is represented by a blank node identifier. Two identical ids in a graph refer to the same blank node, but equal identifiers in different graphs refer to different nodes, since separate graphs do not share any of them.

The graph in figure 1.2 can be expressed in N3 as

```
@prefix exstaff: http://example.org/staffid/
@prefix exterms: http://example.org/terms/
exstaff:85740   exterms:address         _:address .
_:address       exterms:housenumber     "34" .
_:address       exterms:street          "Royal College St." .
_:address       exterms:city            "London" .
_:address       exterms:state           "England" .
```

Blank node identifiers start with an underscore and a colon, followed by a label: in the example `_:address` is the identifier of the blank node that represents the address.

1.6 Literals

The literals presented heretofore this section were untyped, just sequences of characters. Using the RDF terminology they are *plain literals*. RDF permits also *typed literals*, were the type is identified by a URI reference.

Since XML Schema already defines a complete type system [16], RDF does not define any new type except one, `rdf:XMLLiteral` , used for embedding XML in RDF.

1.6.1 Datatypes

Formally, a datatype consists of a lexical space, a value space and a lexical-to-value mapping.

The XML boolean datatype `xsd:boolean` , for example, has a value space of two elements:

$$V = \{\text{T, F}\}.$$

a lexical space of four elements:

$$L = \{\text{“true”, “false”, “1”, “0”}\}.$$

and the following lexical-to-value mapping:

$$M = \{\text{<“true”, T>, <“1”, T>, <“0”, F>, <“false”, F>}\}.$$

1.6.2 Typed literals

The general way to express a typed literal in the Notation3 dialects is:

```
"[Lexical Form]"^^<[URI reference]>
```

and only in N3 and Turtle:

```
"[Lexical Form]"^^[Qualified name]
```

The integer 24 is thus `"24"^^<http://www.w3.org/2001/XMLSchema#integer>` or `"24"^^xsd:integer` . N3 and Turtle can also parse numeric and boolean literals with no datatype URI and cast them automatically. The constant value `true` , with no quotes, is equivalent to `"true"^^xsd:boolean` , and `123` is equivalent to `123^^xsd:integer` .

XML literals

XML literals are literals whose value space is an XML tree. In RDF/XML documents, custom XML markups can be embedded with the `rdf:parseType="Literal"` attribute:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY rdf     "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <!ENTITY ex      "http://example.org/">
]>
<rdf:RDF xmlns:rdf = "&rdf;">

    <rdf:Description rdf:about="&ex;someXML">
        <rdf:value rdf:parseType="Literal">
<root/>
    <node prop="value"/>
</root>
    </rdf:Description>

</rdf:RDF>
```

XML literals can be expressed in the N3 dialects as well, but it requires a lot of escaping:

```
@prefix ex:  <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ex:someXML rdf:value "<root/>\n\t<node prop=\"value\"/>\n</root>"^^rdf:XMLLiteral .
```

1.6.3 Plain literals

A literal that has only the lexical form is called in RDF a *plain* literal. Plain literals may specify a *language tag* as defined by RFC 3066 [11], normalized to lowercase.

In N3, Turtle and N-Triples, the optional language tag follows the lexical form and the '@' separator character. For example, the literal `"Firenze"` with Italian language tag is `"Firenze"@it` .

1.7 RDF Schema

When RDF users want to describe their resources, they are also creating a *vocabulary*: a well defined set of terms of different classes, each with specific properties.

For example, people interested in describing bibliographic resources would describe classes such as `ex:book` , and use properties such as `ex:author` and `ex:title` .

RDF Schema (or RDFS) [19] is a standard vocabulary that provides the terms to describe such classes and properties: for example it permits one to say that `ex:author` is an expected property of an `ex:book`. In this sense RDFS provides a type system for RDF, since it allows one to define classes, subclasses and their properties. But this information is not a constraint like in object-oriented languages, but just provide an additional description about the RDF resources.

1.7.1 Classes

In RDF Schema, a class is an instance of the `rdfs:Class` resource, thus a class is any resource having an `rdf:type` property whose value is `rdfs:Class`.

This example defines a class of motor vehicles:

```
@prefix ex: http://example.org/schemas/vehicles .

ex:MotorVehicle rdf:type rdfs:Class .
```

A particular vehicle is then an instance of `ex:MotorVehicle`:

```
@prefix ex: http://example.org/schemas/vehicles/ .
@prefix exterms: http://example.org/terms/ .

ex:MotorVehicle       rdf:type rdfs:Class .
exterms:johnSmithsCar rdf:type ex:MotorVehicle .
```

Differently from some object-oriented languages, a resource can be an instance of more than a single class.

Subclass relationships are defined with the standard `rdfs:subClassOf` predicate. Trucks and vans, for example, are subclasses of the motor vehicle class, and the minivan category is a subclass of van:

```
ex:Truck   rdfs:subClassOf ex:MotorVehicle .
ex:Van     rdfs:subClassOf ex:MotorVehicle .
ex:MiniVan rdfs:subClassOf ex:Van .
```

RDF software that understands the meaning of RDFS can infer, at this point, that `ex:MiniVan` is also a subclass of `ex:MotorVehicle`, since `rdfs:subClassOf` is a *transitive* property (see [19], section 3.4), and that `ex:Van`, `ex:MiniVan` and `ex:Truck` are classes as well.

1.7.2 Properties

In RDF Schema, the properties of the classes are described using the RDF class `rdf:Property`, and the RDF Schema properties `rdfs:domain`, `rdfs:range`, and `rdfs:subPropertyOf`.

A resource can be defined as a property by declaring it to be an instance of `rdf:Property`. The RDFS term `rdfs:domain` can be used to indicate that a particular property applies to a designated class. For example, books should have an author property:

```
ex:Book     rdf:type      rdfs:Class .
ex:author   rdf:type      rdf:Property .
ex:author   rdfs:domain   ex:Book .
```

The triple `ex:author rdfs:domain ex:Book` does not specify only that books have an "author" property, but also that every resource that has an "author" property is an instance of `ex:Book`.

In programming languages, many classes (and thus their instances) may have properties with the same name; in RDFS if the same property applies to two different classes, then every resource that has that property defined *must* be an instance of both classes. For example:

```
ex:weight rdf:type    rdf:Property .
ex:weight rdfs:domain ex:Book .
ex:weight rdfs:domain ex:MotorVehicle .

exterms:someResource ex:weight "10"^^xsd:integer .
```

means also that `exterms:someResource` is both an instance of `ex:Book` and of `ex:MotorVehicle`.

In the same way as `rdfs:domain` tells one which is the class of the *subject* of a triple using a certain property, `rdfs:range` allows one to specify the class of the *object*; the author of book, for example, should be an instance of the `ex:Person` class:

```
ex:Person rdf:type   rdfs:Class .
ex:Book   rdf:type   rdfs:Class .
ex:author rdf:type    rdf:Property .
ex:author rdfs:domain ex:Book .

ex:author rdfs:range  ex:Person .
```

RDF Schema provides a way to specialize properties as well as classes, using the standard `rdfs:subPropertyOf` property. All `rdfs:range` and `rdfs:domain` predicates that apply to an RDF property also apply to each of its sub-properties:

```
ex:driver        rdf:type           rdf:Property .
ex:driver        rdfs:domain        ex:MotoVehicle .
ex:driver        rdfs:range         ex:Person

ex:primaryDriver rdfs:subPropertyOf ex:driver .
```

The primary driver of a vehicle, therefore, is, of course, also a `ex:driver` of it.

1.7.3 Richer schema languages

RDF Schema provides basic capabilities for describing RDF vocabularies, but additional capabilities are also possible and useful, like adding cardinality constraints on properties, e.g. that a `ex:Person` has exactly one biological father, or that a basketball team has five players; or specifying that two different resources, with different URI references, actually represent the same concept.

These capabilities, and many others, are the targets of *ontology languages* such as OWL [40]. OWL is based on RDF and RDF Schema, and its intent is to provide additional machine-processable semantics for resources, that is, to make the machine representations of resources more closely resemble their intended real world counterparts. Both RDF and OWL are part of the development of the *Semantic Web*.

Chapter 2

MonetDB

MonetDB [2] is an open source database management system developed at CWI [1], the Dutch National Research Institute for Mathematics and Computer Science (in Dutch: Centrum voor Wiskunde en Informatica), one of the leading European research centers in the field of mathematics and theoretical computer science. MonetDB is a platform for scientific research in the database field; a list of all the publications related to this system can be found at [7].

2.1 Design principles

MonetDB has been designed to efficiently process query intensive workloads over large datasets, in application fields like data mining, OLAP (On-Line Analytical Processing), GIS (Geographic information system), XML Query, text and multimedia retrieval.

To achieve this goal, MonetDB adopts a decomposed storage model (DSM), opposed to the conventional N-ary storage model (NSM). The DSM approach models relations as sets of columns instead of sets of tuples, where each column is represented by a binary table, or *BAT* in MonetDB, which consists of a *head* and a *tail* column, with the first containing a row identifier and the latter containing the actual data (figure 2.1).

2.1.1 A simple binary algebra

The immediate benefit of the column-wise storage is that it saves I/O when scan-intensive queries on tables with a large number of columns need just a few of them, since only the ones needed are accessed: in an OLAP application, for instance, where the fact tables are normally huge and with many columns, DSM would perform significantly faster than NSM if only a few columns are needed.

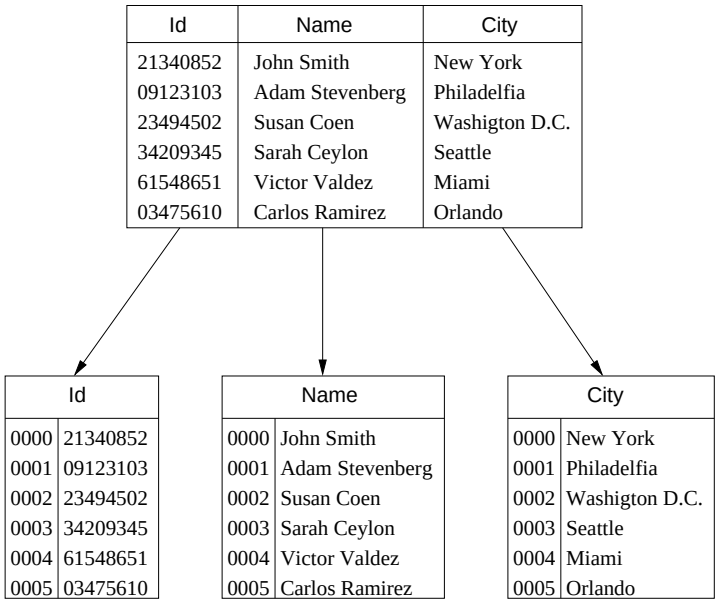

Figure 2.1: Decomposed storage model

The most important reason for which vertical fragmentation has been chosen, however, is that it improves computational efficiency since it does not suffer from problems generated by tuple-at-a-time interpretation. MonetDB, instead, processes data a column at a time, essentially looping over an array; this improves the performances dramatically, since it leads to predictable instructions that can be pipelined by modern CPUs, thus avoiding branch mispredictions and achieving a good instruction-per-cycle ratio.

The disadvantage of this simple approach is that query execution cannot be pipelined, in the sense that the result of an operator cannot flow directly into the next one; in a row-store, each operator eats tuples and produces tuples that can flow to the next operator, in a pipeline. MonetDB, on the contraty, has to materialize every intermediate result, and therefore does not scale well on problems significantly larger that main memory.

2.1.2 Main memory DBMS

MonetDB makes aggressive use of main memory by assuming that the database hot-set fits into it. It does not mean that all the data has to be loaded into memory: for large databases, MonetDB relies on the underlying operating system's virtual memory by mapping large BATs into it. This aspect is taken into account in the BAT design, that must have the same representation on disk and in main memory in order to take advantage of memory mapping, thus avoiding the use of hard

pointers [17]. In this way the hot pages are kept in memory, and the less accessed ones can be automatically swapped out on disk by the OS.

This important assumption makes memory access a severe concern. A general observation about main memory access is that CPU speed increased much faster than memory latency has decreased, turning it into an increasing bottleneck.

MonetDB's execution engine is therefore focused on exploiting CPU caches through cache-conscious algorithms; the DSM approach was chosen also for this reason [36].

The system also packages a calibrator tool [37] that calculates the L1 and L2 cache sizes, their line-size and their access and miss latencies; it extracts the number of the Translation Lookaside Buffer levels, the capacity of each level, and measures the main memory and TLB miss latencies.

2.2 Architecture overview

The architecture of MonetDB has a front-end and back-end layout (fig. 2.2); the back-end is the heart of the system, that provides the binary data model, the query execution engine and basic concurrency and transaction mechanisms, while the front-ends are query-language processors that may support different data models, which are all mapped onto the back-end's binary algebra.

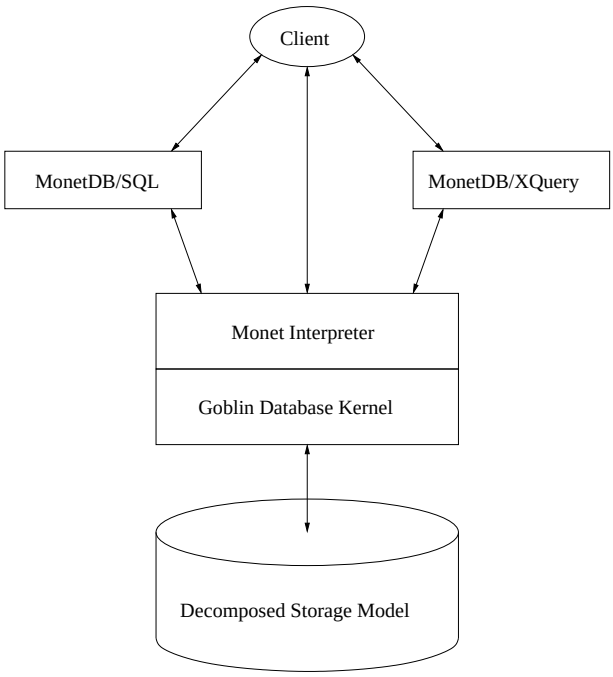

Figure 2.2: MonetDB architecture

The front-ends currently distributed with MonetDB are MonetDB/SQL and MonetDB/XQuery; MonetDB/SPARQL was just started as part of the work of this thesis.

The interface between the back and the front-ends is provided by the *MonetDB Assembly Language* (or MAL) for the current version of MonetDB (ver. 5) and by the *MonetDB Interpreter Language*, or MIL, for version 4 of MonetDB. The latter is still used by the XQuery front-end.

The low-level table-handling code supplying the binary tables, the facilities to map them into virtual memory and the concurrency mechanisms is Goblin Database Kernel (GDK).

MAL (as well as MIL) is a Turing-complete interpreted and procedural language whose operators form a closed algebra on the binary tables, targeted to performance (in terms of parsing, analysis, and ease of target compilation by query compilers) and extensibility.

The clients can communicate with the MonetDB server through the standard database interfaces JDBC and ODBC, or through the native MonetDB Programming Interface (MAPI). The Perl, PHP, and Python API are build on top of the MAPI routines.

2.3 Binary tables structure

A BAT (fig. 2.3) is a binary table, hence it has a *head* and a *tail* column. It can be accessed through a pointer to a *BAT descriptor*, that points to two *column descriptors*, one for the head and one for the tail. A column descriptor holds column-specific information, such as the type of the stored data and search accelerators such as if the column is sorted or not, or if it contains unique values. The actual data is stored in the *BUN heap*, an array of binary tuples, called *BUNs* (Binary Units). The BUN heap can be reached from a BAT descriptor through the *BUN descriptor*.

Fixed size data, like integers, floating point numbers or timestamps, are stored directly in the BUN record; variable size records like strings are kept in a separate heap, with the BUN storing an offset into it.

In such a way BUNs always hold fixed size data, allowing a simple array representation.

The columns can be of quite a large number of types; these are:

- `oid` : integer values used as object identifier. Their length depends on the system MonetDB is built on: 32-bits on 32-bit systems and 64-bits on 64-bit systems. If MonetDB knows that a `oid` column is a dense ascending sequence, it can be represented by virtual `oids` .

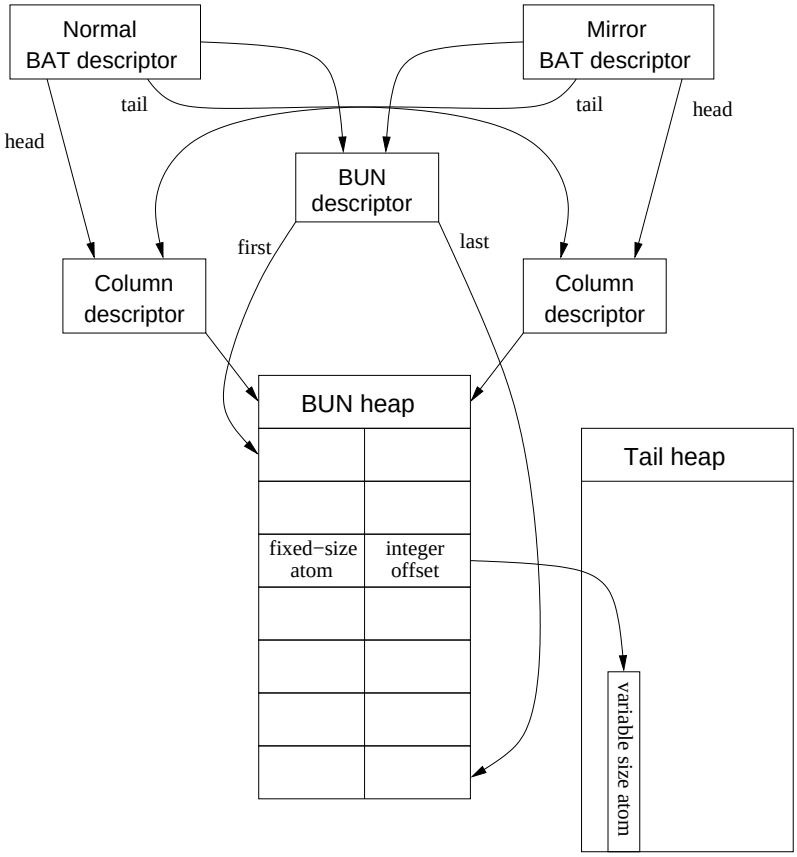

Figure 2.3: BAT structure

- `void` : virtual `oids` . They are dense ascending sequences of `oids` starting from a base `oid` , that is sufficient to represent the whole sequence. Virtual `oids` take therefore no storage space, and since they represent the array index of the other column (plus the base of the sequence), value lookup by virtual `oid` can be done with extreme efficiency by position.
- `bit` : booleans, implemented with one-byte values.
- `chr` : single 8-bit character.
- `bte` : tiny 8-bit integers.
- `sht` : short 16-bit integers.
- `int` : the C language 32-bit integers.

- `wrd` : machine-word sized integers (32-bits on 32-bit systems, 64-bits on 64-bit systems).
- `ptr` : memory pointer values. Their length is also system-dependent.
- `flt` : the IEEE 32-bit float type.
- `dbl` : the IEEE 64-bit double type.
- `lng` : 64-bit integers.
- `str` : zero-terminated UTF-8 strings.
- `bat` : a column of type `bat` holds BAT descriptor numbers.

New types can be defined for MonetDB, although it is a complex operation that requires registering the new atom (and the routines related to it) into the database kernel, by writing an extension module.

A number of user-defined types, like date, time, timestamp, URL and blob for instance, is shipped with the system.

2.4 Binary table optimizations

Reverse view

The complex structure of BATs allows the performance of many optimizations. Every binary table, for instance, has two incarnations (see figure 2.3): the *normal* view and the *reversed* view, that coexist. The reverse view has the the pointers to the head and tail column descriptors swapped. The MAL `mirror` operator, that returns the reverse view, is therefore free of cost.

Void view

The MAL `mark` operator, given a BAT, creates a new view introducing a new tail column of virtual `oids` . The new view shares the head column descriptor and the BUN heap of the given BAT, and has a new column descriptor for the tail (see fig. 2.4). To introduce a new head of `voids` , it is sufficient to call the `mark` operator on the reverse view of the original BAT.

This operation is almost free of cost and independent of the number of binary tuples in the heap, and since MonetDB very often needs to introduce a sequence of dense system-generated `oids` during query processing, this simple optimization is very profitable.

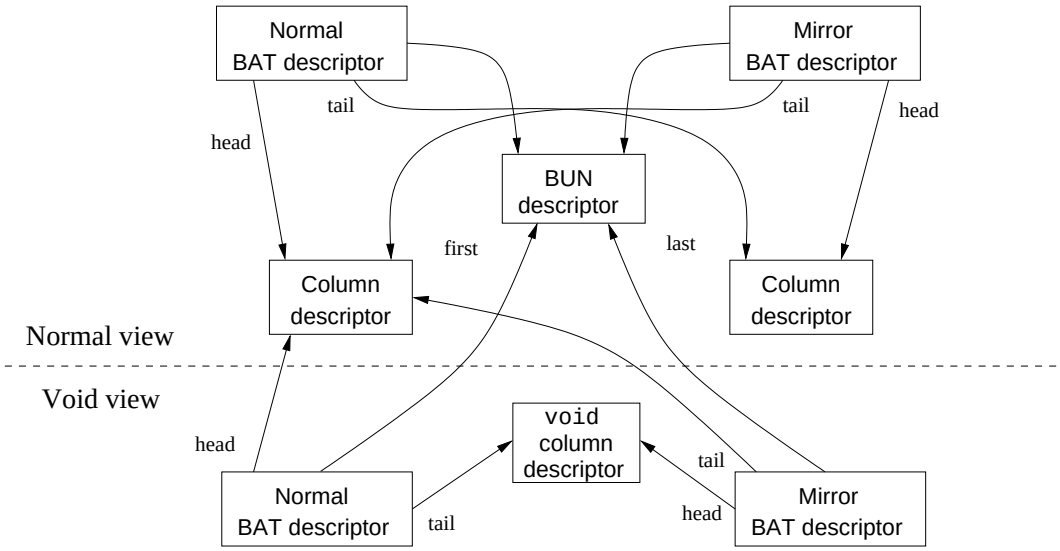

Figure 2.4: BAT void view

Slice views

Range-selects performed on ordered values of a BAT are implemented as a *slice view*. The BUN descriptor of this view points to the part of the BUN heap that satisfies the selection predicate, as shown in figure 2.5.

Since the data is sorted, the lookup of the values that satisfy the selection predicate can be performed with a fast binary search, or even faster by position if the column contain `voids` .

2.5 Current status and future

MonetDB by now has almost fifteen years of maturity, and has therefore all the features that one would expect from a modern database system.

Since it started as an OLAP and data-mining tool, and thus geared to high-performance in query-intensive scenarios, it is not suited for update-intensive applications like OLTP.

On the other hand, MonetDB exhibits extremly good performance in the application fields it was developed for, as shown by the TPC-H benchmark [8].

The future is the MonetDB/X100 kernel [18, 53], that squeezes the CPU until the last cycle, better utilizing the caches by processing vectors of values (of appropriate size to make them fit into the cache) at once in a Volcano-style execution pipeline. The current version of MonetDB, instead, processes one column at a time and therefore is bound by the memory latency and by the fact that it has to

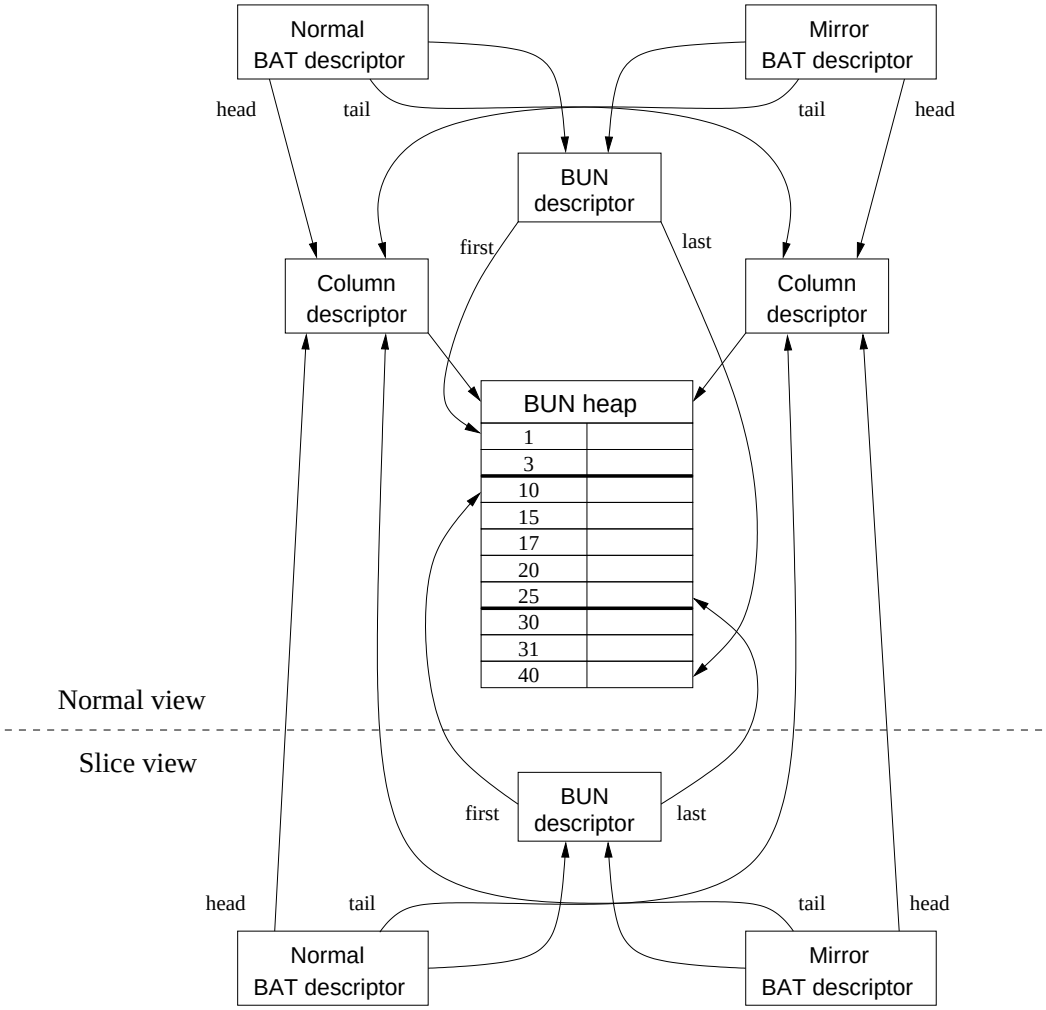

Figure 2.5: A range-select of values between 10 and 25

materialize every intermediate result.

X100 also gets rid of MonetDB's assumption that the dataset fits into main memory, in order to deal with problems significantly larger than the available RAM; this new kernel can process data at an incredible speed, but it would be useless if the data itself cannot be loaded fast enough from disk. To overcome this problem, X100 adopts a proprietary lightweight compression, that permits the increase the disk bandwidth by storing the data compressed, trading this larger bandwidth with some CPU utilization to decompress the data. Another way in which X100 speeds up the perceived disk speed is to share the scans between concurrent queries.

Chapter 3

RDF storage techniques and related work

Since RDF [5] became a W3C Recommendation in 1999, a considerable number of storage engines have been developed for this kind of data; the most known tools are OpenLink Virtuoso [49], Sesame [46, 20] and Jena [33, 51], while an updated survey on RDF storage systems is available in [48].

3.1 RDF storage techniques

The most natural way to store an RDF graph in a relational database management system is in a three column table, with each row containing the subject, property and object of every triple in the graph. In some cases a forth column is present to store the graph IRI; the alternative is to store each graph in a different table of triples.

Normalization Since IRIs are long strings, and since object literals may even represent an entire book, it is common to normalize the table so that same IRIs or literals are mapped to a same 32 or 64-bit integer identifier, in order to save space. The mapping between ids and IRIs or literals is done by one or more dictionary tables; since many IRIs have the same prefix, it is possible to save even more space by assigning them an id as well.

Property tables It is usual to find patterns in the RDF data, that comes both from the RDF specification itself and from the user data. For example, RDF permits one to define sequences and bags of objects, that all have the same structure. It is possible to optimize the relational schema to better fit these patterns: the use of property tables is a way to capture them. A property table has one column for

the subject of an RDF statement, and one or more columns, holding the the object values of one or more properties for that subject. It is useful when there are groups of properties that are often accessed together; for example it may be common to retrieve all the data of a person, like “name”, “surname” and “city”, at the same time. If these properties are stored altogether in a property table, as shown in figure 3.1, the retrieval is faster than in the common three-column layout.

subject	name	surname	city
http://xmlns.com/foaf/0.1/Alice	Alice	Green	London
http://xmlns.com/foaf/0.1/Bob	Bob	Adams	New York
http://xmlns.com/foaf/0.1/Cindy	Cindy	Logan	Liverpool
http://xmlns.com/foaf/0.1/George	George	Smith	Edinburgh

Figure 3.1: A property table

Multi-column property tables are not suited for multi-valued properties, i.e. properties that may have more than one value for a single subject: in this case for each different object value, a new row would be needed in the property table, that has `null` values in all the columns except for the subject and the property that caused the new row to be added. Two-column property tables do not have this complication, since `null` values are always avoided.

Vertical Partitioning A recent proposal [9] suggests using only two-column property tables (fig. 3.2, with normalized subject), ordered on the subject. It has the disadvantage of spreading properties that may be often accessed together and it requires more joins than with multi-column property tables, but has the advantage of avoiding the usual giant three-column table and `null` values, generating less I/O, since only the tables with the needed properties are accessed, while equi-joins on subjects can be executed with the merge algorithm, since the data is ordered. The advantages may be even more considerable when using a column-oriented database like C-Store [47] or MonetDB.

Materialized Join Views Since the most relevant cost of queries on RDF data is represented by the joins needed to traverse the graph, a materialized view of some of these would speed up processing, as discussed in [21] and [9].

In the latter, this approach is recommended for path expressions, for example to find all the works of authors who were born in a certain year. This query requires finding a path in the RDF graph from a work to a date, passing through an author, which can be done with a equi-join on object (an author of some work) and

dictionary	
00	http://xmlns.com/foaf/0.1/Alice
01	http://xmlns.com/foaf/0.1/Bob
02	http://xmlns.com/foaf/0.1/Cindy
03	http://xmlns.com/foaf/0.1/George

name	
00	Alice
01	Bob
02	Cindy
03	George

surname	
00	Green
01	Adams
02	Logan
03	Smith

city	
00	London
01	New York
02	Liverpool
03	Edinburgh

Figure 3.2: Vertical partitioning

subject (authors born in a certain year). In a vertically partitioned schema, moreover, the new path can be stored in a two column property table like all the others in this approach, whose name is the concatenation of the two properties traversed by the path; in the example, the new table would be called author:wasBorn, as shown in figures 3.3 and 3.4.

```
_:a dc:author _:z .
_:a dc:title  "The Cherry Orchard" .
_:b dc:author _:y .
_:a dc:title  "Moby Dick" .

_:y dc:name    "Herman Melville" .
_:y dc:wasBorn 1819^^xsd:gYear .
_:z dc:name    "Anton Chekhov" .
_:z dc:wasBorn 1860^^xsd:gYear .
```

Figure 3.3: Works and authors graph

Searching for a work whose author was in born in 1860, for example, is much faster with this new table, since no joins are required any longer.

While in [9] only object-subject join materialization is cited, [21] recommends also materializing subject-subject and object-object joins. After all, materializing

author:wasBorn	
_:a	1860^^xsd:gYear
_:b	1829^^xsd:gYear

Figure 3.4: Materialized join view in a vertical partitioning approach

these views in a vertically partitioned store would create new tables that would not respect the usual two-column schema.

A second approach to materialize joins presented in this paper is the "Subject-Property Matrix Materialized View". This matrix is a property table that contains not only direct properties, but also nested ones. A property p_1 is direct for a subject s_1 if there exists a triple (s_1, p_1, x), while p_m is nested when there exists a set of triples such as (s_1, p_1, o_1), (o_1, p_2, o_2), ..., (o_{m-1}, p_m, o_m). Nested property tables, thus, are a way to implement path expressions as proposed in [9], but with the limitation that only single-valued properties can be used.

3.2 OpenLink Virtuoso

Virtuoso is an open source and commercial product that combines an ORDBMS engine, a Web Application and File server in a single product. It supports Web Services, XQuery and XPath for XML data queries, RDF data storage and SPARQL, among many other functionalities.

Its relational RDF storage system consists in six tables [3]:

- A Quad table, with columns G, S, P, and O, that store respectively graph, subject and predicate IRI ids, and the object, of type *any*.
- An Obj table, that stores long string objects. It has three columns, an object ID as primary key, and the VAL and LONG_VAL columns.
- Four id-to-string mapping tables, for IRIs, IRI prefixes, datatypes, and language tags.

If the object value is a non-string SQL scalar, such as a number or date, an IRI, or a string of less than 20 characters, it is stored in its native binary representation in the O column of the Quad table. Long strings and RDF literals with non-default type or language are stored using an rdf_box composite object. Its fields are datatype, language, content (or beginning characters of a long string content) of the object, and a possible reference to the Obj table, which holds string literals longer than a certain threshold or that should be free-text indexed. Depending on

the length of the text, this is stored into the VAL or in the LONG_VAL column. The truncated value present in the O column of the Quad table can be used for determining equality and range matching, even if closely matching values need to look at the real string in Obj. When LONG_VAL is used to store a very long value, VAL contains a checksum of the value, to accelerate search for identical values when the table is populated by new values.

3.2.1 Main table indexing

The main Quad table is represented by two indexes, one on GSPO and another on PGOS. These indexes have proven to be effective for two common and practical classes of queries: those that, given a subject and a property, retrieve the associated objects; and those that find subjects for some defined property set to a value. In both cases G has to be known, otherwise the queries are next to impossible to evaluate, as stated in [27].

The PGOS index represents the subject column as a bitmap, in order to obtain a compression of the index itself (a detailed description can be found in [26]). Instead of saving the subject IRI id in its binary representation for each PGO, up to 8K different subject IRI ids are stored together in a bitmap string, as long as they have the same PGO and fall in the same segment of the integer domain, which is divided in blocks of 8K values. This approach saves space twice: it avoids many repetitions of identical PGO's, and may store up to 8192 subjects in a bit array, with just a small overhead for identifying a block in the integer domain.

If in a segment there are less than 512 IRI ids to represent, an 8K bitmap would waste space; in this case compression is achieved storing a subject as a 16 bit entry in a list; each of the entries is an offset from the start of the block. If in one of the blocks there is only one IRI id to save, this is stored "as is".

With the Wikipedia links set, the PGOS index size is a quarter of the size of the GSPO index, which cannot represent the objects as a bitmap since these are not fixed length integers in Virtuoso. It took 60% of the space of GSPO with the WorldNet set. Both datasets can be found at [25].

3.2.2 Query optimization through data sampling

It is common for SQL optimizers to have statistics about tables to be queried, such as the number of rows, or the number of distinct values in a column and their distribution. These kinds of metadata become much less useful when all the data is stored in a single table [27].

A solution for this problem is to have a look at the actual data: when a query is compiled, Virtuoso's optimizer takes a sample of the index, counting in each level of the tree how many ways it branches out and how many of the leaf pointers match

the search condition. For example, in a query where some G, S, and P values have to be matched, it is possible to know how many siblings of the index tree have the same given G, S, and P, allowing it to accurately estimate the cardinality of the matching set. The same estimate can be made for the whole index if no key part is known, using a few random samples of the index.

3.3 Sesame

Sesame is a store and a reasoning tool for RDF. It can be backed on many RDBMS, but it may also use plain files or main memory for storing the RDF triples; the abstraction of the storage mechanism is provided by the SAIL layer (Storage And Inference Layer), which also exploits the features of the particular DBMS.

3.3.1 Architecture of Sesame

Sesame has a layered architecture (fig. 3.5), where each layer has a well-defined and highly-cohesive set of responsibilities. The uppermost layer is composed of a set of ProtocolHandlers, namely HTTP, SOAP and RMI, which receive the requests of the clients. The RequestRouter directs these requests to one of the underlying application modules, which are the query, admin and export modules.

The query module parses and optimizes a query, that can be performed in the last version of Sesame in SeRQL (Sesame RDF Query Language) and SPARQL; the optimized query is then passed to the SAIL layer. The admin module allows one to incrementally add data to an RDF repository or to delete it, while the role of the export module, as the name may suggest, is to make batch exports of the RDF data.

3.3.2 SAIL

This layer transparently abstracts the specific storage method to the upper layers of Sesame, and translates the requests (queries, incremental inserts and batch exports) to DBMS-specific SQL code, or to Java method calls that manage main memory and file storage. Thus, its API defines a basic interface for storing, inserting and deleting RDF data.

The SAIL is also able to deal with RDFSchema: it offers methods for querying class and property subsumption, and domain and range restrictions. Since any SAIL implementation has a complete knowledge of the underlying storage engine, for example the specific RDBMS schema, it can use this knowledge to infer class subsumption more efficiently.

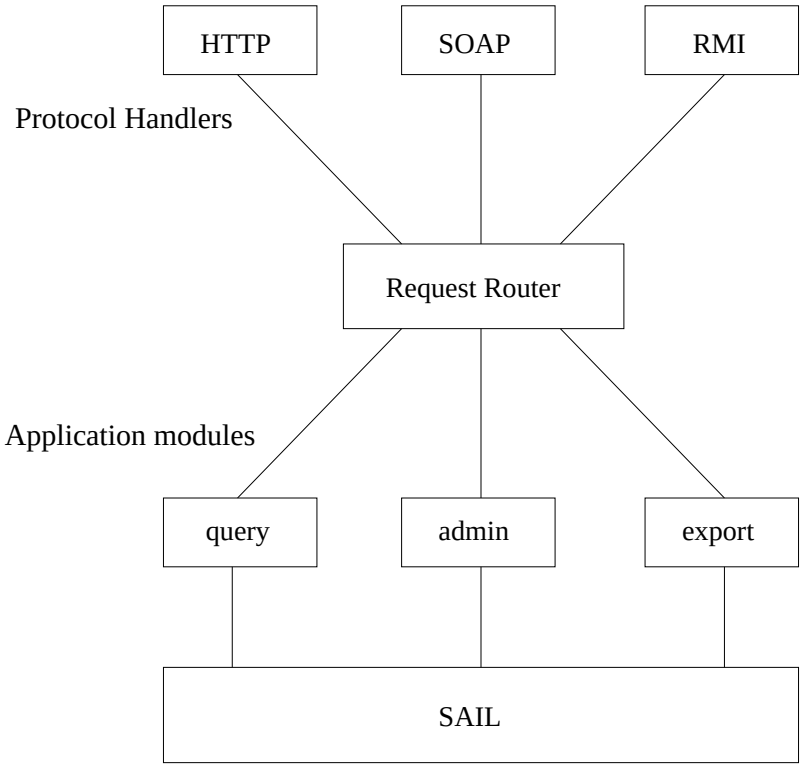

Figure 3.5: Architecture of Sesame

The SAIL implementations that deal with DBMSs are currently two, one that integrates MySQL and one PostgreSQL.

SAIL/PostgreSQL

The PostgreSQL specific implementation exploits its object-oriented features, in particular subtables and table hierarchy.

As in many RDF engines, also in SAIL/PostgreSQL the IRIs and the literals are normalized by mapping them to numeric ids, but this is done in an object-oriented fashion: if a resource does not have a defined rdf:type property, then it will be mapped to an id in the Resource table, otherwise in a table named as the class, that extends Resource (in figure 3.6, Writer and Book extend Resource, and FamousWriter extends Writer). Thus, if a new class is added to the store, a new table has to be created.

If one class extends some other one, the two tables that represent them will constitute a row entry in the SubClassOf table, as subtables; FamousWriter and Writer tables are an example of this situation in figure 3.6. The same approach is used for properties and subproperties.

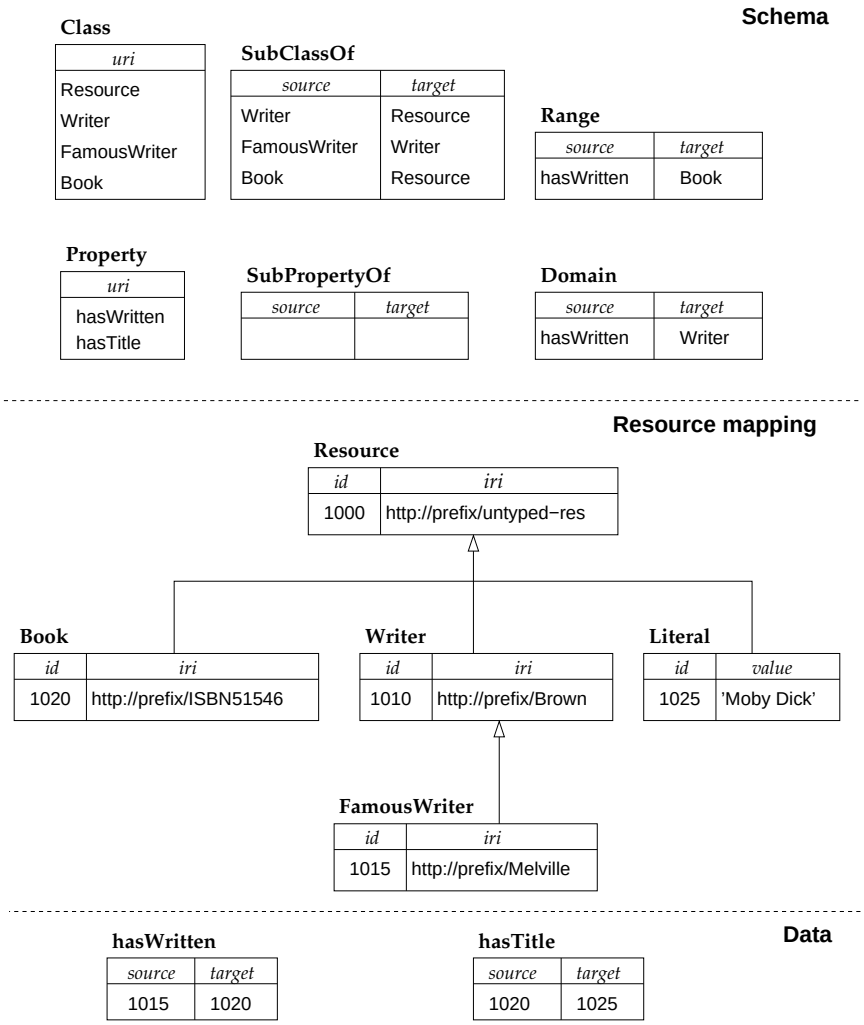

Figure 3.6: SAIL/PostgreSQL database schema

This schema has proven to be satisfactory in querying scenarios, but slow during inserting, since in PostgreSQL the creation of new relations is an expensive operation, and also since subtables cannot be inserted as normal values, requiring the destruction and rebuilding of the SubClassOf table every time a new subclass relationship has to be added; the only way to have subtables as values is to specify them at the time of creation of the container.

SAIL/MySQL

MySQL's specific implementation adopts a complex but strictly relational schema (see [20] for details), that stores RDFSchema information (like type, class, sub-

ClassOf, property or subPropertyOf) in separate tables from the triples, and normalizes IRIs and the IRI prefixes. A column is_derived is added in the triples table and in the RDFS relations to encode the fact that a triple, a property or class subsumption, for instance, has been created by the RDFSchema reasoner in the SAIL. This schema has the advantages over PostgreSQL that does not change when new RDFSchema information is added, and performs significantly better especially in inserting new data.

3.4 Jena

Jena is an open source project written in Java, which is currently in its second version. The main storage problems addressed by Jena2 are:

- the excessive number of joins between the triples' table and the id-to-string dictionary
- the hugeness of the main triples' table, which lead to scalability complications
- the reified statements storage, that would normally require four statements for each statement to reify
- query optimization, which in Jena1 was performed in the Java layer and did not rely on the DBMS.

3.4.1 Storage schema

In its first version, Jena used to store its statements in a four-column table, where the object was stored in one of two different columns, depending on if it was an IRI or a literal. The schema was normalized, so two other tables served as dictionaries, one for IRIs and one for literals.

This schema was adopted with any DBMS, except with BerkleyDB. In this case, the schema was not normalized, and replicated three times, indexed once on subject, once on property and once on object. In many cases this approach proved to be faster, in part because of the lack of transactional support in BerkleyDB, but mostly because of the fewer number of joins required by the denormalized schema.

Thus, in its second version Jena stores the IRI strings and the literals directly in the main table, which consists of the classical three-column layout, except for those which exceed a configurable threshold, whose default is 256 characters. Different RDF graphs can be stored in different statement tables, in order to keep the

table size for each graph low. Common IRI prefixes are compressed by assigning them an id and replacing their occurrences in the main table with a database reference; since the number of different prefixes is expected to be low, the prefix table would be held in main memory, so that expanding the ids would not require any I/O.

Exploiting data patterns

As discussed in section 3.1, RDF data may contain patterns that can better fit in property tables that in the usual three column approach. Jena allows one to define property and property-class tables; the latter are a kinds of property tables that have a double purpose: each of them keeps the instances of an rdfs:class in the first column and the values of the properties of each instance in the remaining columns.

Jena also permits one to create two-column property tables, in order to support multi-valued properties.

By default, a Jena store is created with no property tables and one property-class table that stores reified statements; these are statements about statements, each of them made of four triples: one declaring an IRI of type rdf:statement, and three to associate this IRI to the subject, the property and the object of the triple to reify. A four-column property-class table can store a reified statement in a single row. In this manner much space is saved, especially in those applications that need to reify every statement.

3.4.2 Architecture

The core of Jena consists in a set of interfaces defined in a Model layer that lets one to manipulate the RDF graph, adding, removing and searching statements. Along these functionalities, there are importing and exporting operations for all the main RDF serialization languages, such as RDF/XML, N3 and N-triples. Client applications interact with the Model, which translates high-level operations in low-level and storage technique-dependent operations.

Specialized Graph Interface

The layer underlying the model abstracts each RDF graph in a different logical graph; each of them is implemented as an ordered list of specialized graphs, optimized for storing a particular style of statements. Any operation on a logical graph is performed by invoking it on each specialized graph; this process can be optimized if an operation can be completely processed by a single specialized graph.

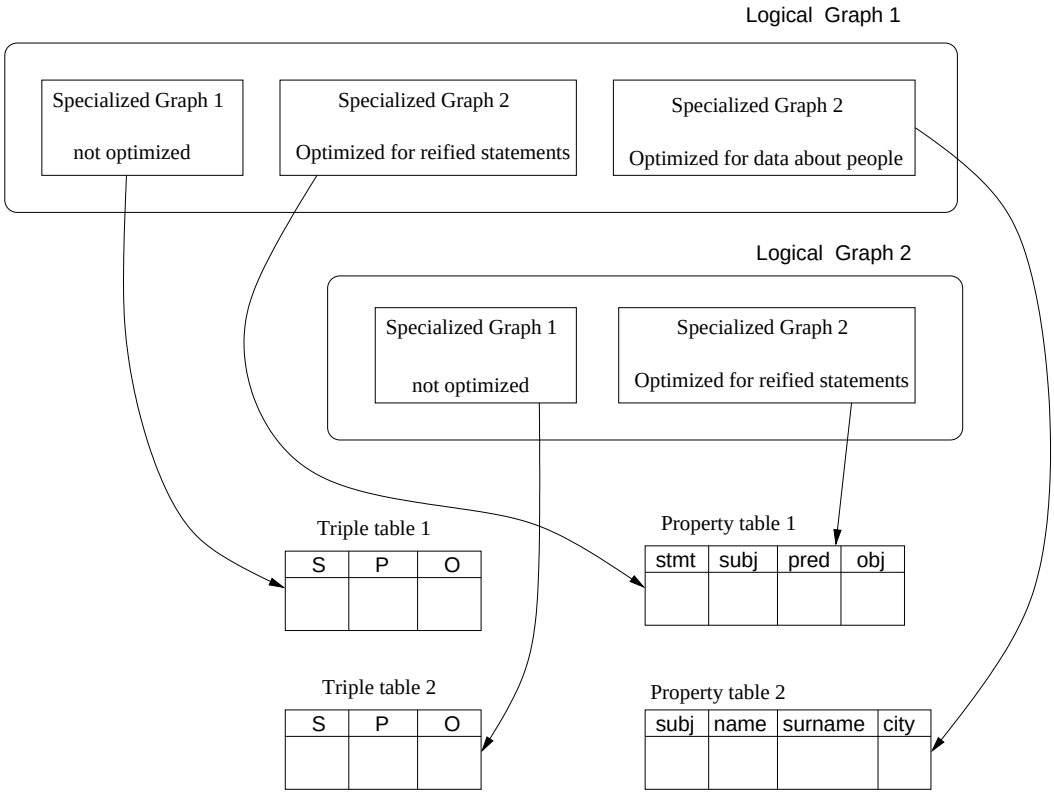

Figure 3.7: Specialized Graph Interface in Jena

Figure 3.7 shows two logical graphs. The first contains a non-optimized specialized graph and two optimized ones; the second contains only a single optimized graph together with the non-optimized one.

Each non-optimized graph is stored in a separate standard triple table; optimized graphs are stored in property tables, which can be shared by different logical graphs.

3.5 Other storage engines

KAON server The KAON server (KArlsruhe ONtology and Semantic Web tool suite [34, 50]), is an ontology management infrastructure that also contains an RDF store.

The KAON server lets one create, manage and query the ontologies it stores, and also provides reasoning mechanisms that can infer new triples from them.

RDFSuite RDFSuite [10], developed by the ICS-FORTH, is "a suite of tools for RDF validation, storage and querying using an on object-relation DBMS", namely PostgreSQL, which can be configured to use property tables; queries against the store are performed in RQL (RDF Query Language), which was developed by ICS-FORTH as well.

Chapter 4

SPARQL

4.1 Introduction

When RDF became a W3C Recommendation in 1999 there was no query language for it as yet, thus several teams developed different languages: for example the Institute of Computer Science of the Foundation for Research and Technology (ICS-FORTH, Greece) proposed RQL [35], the Sesame [46] group developed SeRQL, and HP proposed RDQL [45].

SPARQL [43] initiated as a W3C proposal to become a standard query language for RDF. The first working draft appeared in October 2004, in June 2007 it became a Candidate Recommendation and finally a Recommendation in January 2008.

The `WHERE` clause provides the central concept in SPARQL, that is *graph pattern matching*: given an RDF graph, a query consists of a pattern which is matched against the given graph. The presentation of the result of a graph pattern can be manipulated by *solution modifiers*, similar to the ones that SQL offers, namely projection, distinct, order by, limit and offset; finally the output can be of different types: yes/no answers, selection of the values of the variables that match the pattern, construction of new triples from those values, and description of specified resources.

4.2 Graph Patterns

As previously stated, graph patterns matching is the concept on which SPARQL is built. There are different kinds of graph patterns, which can be combined to build arbitrary complex queries:

- Basic Graph Patterns, where a set of triple patterns must match.

- Group Graph Patterns, where a set of graph patterns must all match.
- Optional Graph Patterns, where additional patterns may extend the solution.
- Union Graph Patterns, where two or more alternative graph patterns are tried.
- Patterns on Named Graphs.

The latter type of patterns will be presented in the RDF Dataset section, at 4.3.1.

4.2.1 Basic Graph Patterns

Basic Graph Patterns, or *BGPs*, are sets of triple patterns, which are like RDF triples except they may present a variable as subject, predicate or object. A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables and the result is equivalent to the subgraph. An example query will make it clearer:

Data:

```
@prefix :<http://library.org/>
@prefix cd:<http://example.org/cd/>
:syntstruct  cd:author "Noam Chomsky" .
:syntstruct  cd:title  "Syntactic structures" .
:refactoring cd:author "Martin Fowler" .
:refactoring cd:title  "Refactoring" .
:poetrycoll  cd:title  "Poetry collection" .
```

Query:

```
PREFIX cd:<http://example.org/cd/>
SELECT *
WHERE
  { ?bookid cd:author ?author .
    ?bookid cd:title  ?title }
```

Result:

bookid	author	title
<http://library.org/syntstruct>	"Noam Chomsky"	"Syntactic structures"
<http://library.org/refactoring>	"Martin Fowler"	"Refactoring"

The first statement of the query, `PREFIX cd:<http://example.org/cd/>` , declares a IRI prefix similar to Turtle; the second statement resembles SQL, both in notation and in meaning: all variables declared in the `WHERE` clause will be returned in the result since a `'*'` is present instead of a list of projection variables. The `WHERE` clause, finally, declares the graph pattern used to match the data.

Each solution is a way in which the variables can be bound so that the basic graph pattern matches the data. The following two subgraphs are matched by the BGP when substituting its variables with the two solutions:

```
<http://library.org/syntstruct>  cd:author "Noam Chomsky" .
<http://library.org/syntstruct>  cd:title  "Syntactic structures" .

<http://library.org/refactoring> cd:author "Martin Fowler" .
<http://library.org/refactoring> cd:title  "Refactoring" .
```

When a variable occurs more than once in the BGP, the same RDF term has to be substituted for each occurrence of that variable for every solution; in the example above, `<http://library.org/syntstruct>` has to be substituted for `?x` in both the triple patterns of the BGP for the first solution, and the same has to be done with `<http://library.org/refactoring>` for the second.

Since in basic graph pattern matching every variable has to be bound in each solution, the triple `:poetrycoll cd:title "Poetry collection"` cannot be matched because the subject `:poetrycoll` has no `cd:author` property, as requested by the query.

Blank nodes in Basic Graph Patterns

A blank node in a BGP behaves like a variable, with the difference that they cannot be part of the result set. For example

```
PREFIX cd:<http://example.org/cd/>
SELECT *
WHERE
  { _:bookid cd:author ?author .
    _:bookid cd:title  ?title }
```

returns

author	title
"Noam Chomsky"	"Syntactic structures"
"Martin Fowler"	"Refactoring"

A formal definition of Basic Graph Patterns can be found in 4.4.5.

4.2.2 Group Graph Patterns

Group graph patterns are sets of graph patterns of any type, delimited by braces, where all the patterns of the set must match. The example at 4.2.1 shows a group graph pattern of one BGP. The following query is different in structure, but will produce the same result, except for the fact that a projection also takes place:

```
PREFIX cd:<http://example.org/cd/>
SELECT ?author ?title
WHERE
  { { ?bookid cd:author ?author } .
    { ?bookid cd:title  ?title  } }
```

Result:

author	title
"Noam Chomsky"	"Syntactic structures"
"Martin Fowler"	"Refactoring"

The `WHERE` clause is made of two nested group graph patterns, each of them of one BGP of a single triple pattern. Other group graph pattern examples will follow in the next section to introduce the other kinds of patterns.

4.2.3 Optional Graph Patterns

Optional graph pattern matching permits one to extend the result set even in those situations where the extra information is not available for each tuple of the result. Querying the same data in section 4.2.1 with:

```
PREFIX cd:<http://example.org/cd/>
SELECT ?title ?author
WHERE
  { ?x cd:title ?title .
    OPTIONAL { ?x cd:author ?author }
  }
```

will result in:

title	author
"Syntactic structures"	"Noam Chomsky"
"Refactoring"	"Martin Fowler"
"Poetry Collection"	

This query looks for all those subjects that have a `cd:title` and optionally a `cd:author` property, and returns their values. Since `:poetrycoll` has no `cd:author` property, `?author` is unbound in its case.

Optional Graph Patterns are left-associative:

```
pattern OPTIONAL { pattern } OPTIONAL { pattern }
```

is the same as

```
{ pattern OPTIONAL { pattern } } OPTIONAL { pattern }
```

4.2.4 Union Graph Patterns

SPARQL provides unions of graph patterns as a mechanism to combine solutions of several alternatives. In the following RDF data graph the same concept of "book title" is expressed with two different IRIs. To retrieve all the book titles in the graph, a union of two graph patterns is needed.

Data:

```
@prefix voc1:  <http://rdfvocabulary1.org/example#> .
@prefix voc2:  <http://rdfvocabulary2.org/example#> .

_:a  voc1:title   "Syntactic structures" .
_:b  voc1:title   "Refactoring" .
_:c  voc2:title   "Poetry Collection" .
_:d  voc2:title   "Ulysses" .
```

Query:

```
PREFIX voc1:  <http://rdfvocabulary1.org/example#> .
PREFIX voc2:  <http://rdfvocabulary2.org/example#> .

SELECT ?title
WHERE{ { ?book voc1:title  ?title }
       UNION
       { ?book voc2:title  ?title } }
```

Result:

title
"Syntactic structures"
"Refactoring"
"Poetry Collection"
"Ulysses"

To determine which vocabulary stores a title, the query has to define a different variable for each pattern:

```
PREFIX voc1:  <http://rdfvocabulary1.org/example#> .
PREFIX voc2:  <http://rdfvocabulary2.org/example#> .

SELECT ?title
WHERE{ { ?book voc1:title  ?title1 }
       UNION
       { ?book voc2:title  ?title2 } }
```

Result:

title1	title2
"Syntactic structures"	
"Refactoring"	
	"Poetry Collection"
	"Ulysses"

4.2.5 Filtering results

As one might expect from a query language, SPARQL provides a certain number of operators to construct arbitrary complex expressions. At this moment the operator set counts 25 elements, among which there are the basic arithmetic and boolean operators, regular expression matching, RDF and SPARQL-specific functions like `isIRI` , `isBlank` , `DATATYPE` and `LANG` .

An example query that uses a `FILTER` may ask only for those books that cost less than a certain price.

Data:

```
@prefix cd:  <http://example.org/cd/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

_:a cd:author "Noam Chomsky" .
_:a cd:title "Syntactic structures" .
_:a cd:price 32.25^^xsd:decimal .

_:b cd:author "Martin Fowler" .
_:b cd:title "Refactoring" .
_:b cd:price 40^^xsd:integer .

_:c cd:title "Poetry collection"
_:c cd:price 9.95^^xsd:decimal .

_:d cd:title "Ulysses" .
_:d cd:price 16.50^^xsd:decimal .
```

Query:

```
PREFIX cd:<http://example.org/cd/>
SELECT ?title ?price
WHERE
  { ?x cd:title ?title .
    ?x cd:price ?price .
    FILTER( ?price < 25 ) }
```

Result:

title	price
"Poetry Collection"	9.95
"Ulysses"	16.50

4.3 RDF Datasets

A SPARQL query is executed against an *RDF Dataset* which represents a collection of graphs. An RDF Dataset comprises an unnamed *default graph*, and zero or more *named graphs*; each graph is identified by an IRI. A query can formulate

different graph patterns against different graphs; the graph that is used for matching a basic graph pattern is called the *active graph*. The `GRAPH` keyword is used to switch the active graph from the default to one of the named graphs.

The dataset can be defined by a query through the `FROM` and `FROM NAMED` clauses. A dataset then consists of:

- A default graph, which is the *RDF-merge* of the graphs specified in the `FROM` clauses.
- A set of (IRI, graph) couples, one from each `FROM NAMED` clause.

The RDF-merge operation, described in [31] at section 0.3, is "the union of a set of graphs that is obtained by replacing the graphs in the set by equivalent graphs that share no blank nodes". The merge of the following two graphs, for example:

```
# graph identified by: <http://example.org/alice>
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:a foaf:mbox <mailto:alice@work.example> .


# graph identified by: <http://example.org/bob>
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Bob" .
_:a foaf:mbox <mailto:bob@oldcorp.example.org> .
```

is

```
# RDF-merge of  <http://example.org/alice> and <http://example.org/bob>
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:x foaf:name "Alice" .
_:x foaf:mbox <mailto:alice@work.example> .
_:y foaf:name "Bob" .
_:y foaf:mbox <mailto:bob@oldcorp.example.org> .
```

Blank nodes and their labels are local to an RDF graph, that means that the label `_:a` represents two distinct resources in the two graphs: a rename must take place before the merge can be performed, as shown in the example.

A query that is matched against such a merged graph is:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/> .

SELECT ?mbox
FROM <http://example.org/alice>
FROM <http://example.org/bob>
WHERE { ?s foaf:mbox ?mbox }
```

Result:

mbox
<mailto:alice@work.example>
<mailto:bob@oldcorp.example.org>

If the query does not specify any FROM nor FROM NAMED clause, like in all the example queries in the previous sections, it is the query engine implementation that decides which RDF graph (or graphs) will be used as default graph. If no FROM clause is present, but there are one or more FROM NAMED clauses, then the dataset includes an empty graph as the default graph.

4.3.1 Patterns on Named Graphs

The GRAPH keyword is used to change the active graph from the default to one of named graphs; a *Graph graph pattern* can be matched against a specific named graph, providing its IRI, or against all named graphs providing a variable instead, which will be bound to the IRI of the graph being matched.

All the following examples will use these two data graphs:

```
# graph id: <http://physicswiki.org/meta/articles>
@prefix : <http://physicswiki.org/metadata/> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:lhc dc:title "Large Hadron Collider" .
:lhc rdfs:seeAlso :higgsboson .
:lhc rdfs:seeAlso :atlas .

:atlas dc:title "ATLAS" .
:atlas rdfs:seeAlso :lhc .
:atlas rdfs:seeAlso :higgsboson .

:higgsboson dc:title "Higgs Boson" .
:higgsboson rdfs:seeAlso :lhc .
```

```
# graph id: <http://itwiki.org/meta/articles>
@prefix : <http://itwiki.org/metadata/> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
:os dc:title "Operating Systems" .
:os rdfs:seeAlso :kernel .

:kernel dc:title "Kernel" .
:kernel rdfs:seeAlso :microkernel .
:kernel rdfs:seeAlso :monolithickernel .

:microkernel dc:title "Microkernel" .
:microkernel rdfs:seeAlso :kernel .
:microkernel rdfs:seeAlso :monolithickernel .

:monolithickernel dc:title "Monolithic kernel" .
:monolithickernel rdfs:seeAlso :kernel .
:monolithickernel rdfs:seeAlso :microkernel .
```

Retrieve or restrict the source of information

This query retrieves all the titles of the articles in the two wikis, and the IRI of the source graph for each of them:

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

SELECT ?src ?title
FROM NAMED <http://physicswiki.org/meta/articles>
FROM NAMED <http://itwiki.org/meta/articles>
WHERE {
        GRAPH ?src
        { ?s dc:title ?title }
}
```

Result:

src	title
<http://physicswiki.org/meta/articles>	"Large Hadron Collider"
<http://physicswiki.org/meta/articles>	"ATLAS"
<http://physicswiki.org/meta/articles>	"Higgs Boson"
<http://itwiki.org/meta/articles>	"Kernel"
<http://itwiki.org/meta/articles>	"Microkernel"
<http://itwiki.org/meta/articles>	"Monolithic kernel"

The `WHERE` clause of the query is a group graph pattern of a single graph graph pattern, that consists of a variable `?src` and a group graph pattern. The latter is matched against every named graph, while `?src` is bound to the source IRI of each tuple of the result.

The same query may restrict the source of information to a single graph:

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

SELECT ?src ?title
FROM NAMED <http://physicswiki.org/meta/articles>
FROM NAMED <http://itwiki.org/meta/articles>
WHERE {
        GRAPH <http://itwiki.org/meta/articles>
        { ?s dc:title ?title }
}
```

Result:

title
"Kernel"
"Microkernel"
"Monolithic kernel"

Named and default graphs

A query can involve both the default graph and the named graphs. In the next query the physics wiki is the only named graph, but it participates also in the default graph together with the IT wiki:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/> .
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

SELECT ?title ?seeAlso
FROM <http://physicswiki.org/meta/articles>
FROM <http://itwiki.org/meta/articles>
FROM NAMED <http://physicswiki.org/meta/articles>
WHERE {
        ?s dc:title ?title .
        OPTIONAL {
          GRAPH <http://physicswiki.org/meta/articles>
          { ?s rdfs:seeAlso :?reference .
            ?reference dc:title ?seeAlso }
        }
}
```

Result:

title	seeAlso
"Large Hadron Collider"	"ATLAS"
"ATLAS"	"Large Hadron Collider"
"ATLAS"	"Higgs Boson"
"Higgs Boson"	"Large Hadron Collider"
"Kernel"	
"Microkernel"	
"Monolithic kernel"	

The query selects the titles of the articles in both wikis and extends this information with the references to other articles, but only for those of the physics wiki.

4.4 SPARQL semantics

Chapter 12 of the current SPARQL specification [44] formally defines which is the correct interpretation of a SPARQL query string, given an RDF Dataset.

The first formal description of SPARQL comes from Peréz et al. in [41] and subsequently in [42] in 2006. The SPARQL Working Draft of March 2007 included this almost unaltered. This section is largely taken from their work.

4.4.1 Initial definitions

RDF Terms, Triples and Variables Let I, B, and L be pairwise disjoint sets of all the IRIs, Blank nodes, and literals. The set of *RDF Terms*, T, is $I \cup L \cup B$.

A tuple $(s, p, o) \in (I \cup B) \times I \times T$ is called an *RDF triple*, where s is the *subject*, p the *predicate* and o the *object*.

Let V be the set of variables, disjoint from all the above sets.

RDF Graph and RDF Dataset An *RDF Graph* is a set of RDF triples. If G is an RDF graph, *term(G)* is the set of all the RDF Terms appearing in the triples of G, and *blank(G)* is the set of blank nodes appearing in G.

An *RDF Dataset* is a set

$$D = \{G_0, (u_1, G_1), (u_2, G_2), ...(u_n, G_n)\}$$

where each G_i is a graph and each u_i is an IRI, with $n \geq 0$. G_0 is called the *default graph*, each (u_i, G_i) is a *named graph*, with u_i the name of G_i. Every G in D has a disjoint set of blank nodes, i.e. for $i \neq j$, $blank(G_i) \cap blank(G_j) = \emptyset$.

Triple Pattern A tuple $t \in (T \cup V) \times (I \cup V) \times (T \cup V)$ is a *triple pattern*. Given a triple pattern t, $var(t)$ and $blank(t)$ are respectively the set of variables and blank nodes occurring in t. It has to be noted here that RDF literals are permitted as subjects: the RDF core working group explained the reason [4]:

> (The RDF core Working Group) noted that it is aware of no reason why literals should not be subjects and a future WG with a less restrictive charter may extend the syntaxes to allow literals as the subjects of statements.

Basic Graph Pattern A *Basic Graph Pattern* is a set of triple patterns. Given a basic graph pattern P, $var(P) = \bigcup_{t \in P} var(t)$ and $blank(P) = \bigcup_{t \in P} blank(t)$ are respectively the set of variables and blank nodes occurring in P.

Solution mapping A mapping μ from V to T is a partial function $\mu : V \rightarrow T$. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. The empty mapping $\mu_\emptyset$ is a mapping such that $dom(\mu_\emptyset) = \emptyset$. Given a triple pattern t and a mapping μ such that $var(t) \subseteq dom(\mu)$, $\mu(t)$ is the triple obtained by replacing the variables in t according to μ.

RDF instance mapping An *RDF instance mapping* σ is a function $\sigma : B \rightarrow T$. Given a triple or a triple pattern t, $\sigma(t)$ is respectively a triple or a triple pattern obtained by replacing the blank nodes in t with RDF terms according to σ.

Pattern instance mapping A *Pattern instance mapping* π is the combination of an RDF instance mapping σ and solution mapping μ. $\pi(x) = \mu(\sigma(x))$.

Multiset of solutions When a graph pattern is evaluated against some graph, the possible solutions form a *multiset*, also called *bag*, that is an unordered collection of elements in which each element can appear more than once. A multiset Ω can be described by a set of the elements in it and a *cardinality function* giving the number of occurrences of each element from the set in Ω. The cardinality of the mapping μ in the bag Ω will be denoted by $card_\Omega(\mu)$; if $\mu \notin \Omega$, then $card_\Omega(\mu) = 0$.

Solution sequence A solution sequence Ψ is a list of solutions μ, possibly unordered. The number of elements in Ψ is denoted as $size(\Psi)$, and elements of Ψ are counted starting from zero:

$$\Psi = [\mu_0, \mu_1 \ldots \mu_{n-1}]$$

where $n = size(\Psi)$. The solution at position i in is denoted as $\Psi[i]$.

Effective Boolean Value - EBV EBV is function $EBV : T \rightarrow \{true, false\}$ that assigns a boolean value to an RDF term $t \in T$. $EBV(t)$ returns:

- *false* if t is boolean or numeric and the lexical form is not valid for that datatype (e.g. `"abc"^^xsd:integer`).
- the value of t if t is a boolean value.
- *false* if t is a zero-length string, T if t is a non zero-length string.
- *false* if t is numeric value equals to zero or NaN, *true* otherwise.
- finally an error is raised in all other cases.

4.4.2 SPARQL abstract query

A SPARQL abstract query is a tuple (E, D, R), where

- E is a *SPARQL algebra* expression
- D is an RDF dataset
- R is a *query form*, one among `SELECT` , `CONSTRUCT` , `DESCRIBE` or `ASK` .

When a query string is parsed, it is converted into an abstract syntax tree composed of:

Graph Patterns	Modifiers	Query forms
Basic	Distinct	Select
Group	Reduced	Construct
Optional	Project	Describe
Union	Order By	Ask
Graph	Limit	
Filter	Offset	

Such an abstract tree is converted in SPARQL algebra expression that comprises the following operators:

Graph Pattern operators	Solution modifiers
BGP	ToList
Join	OrderBy
LeftJoin	Project
Union	Distinct
Graph	Reduced
Filter	Slice

4.4.3 Graph Pattern translation to SPARQL algebra

The SPARQL specification [44], section 12.2, describes the algorithm to translate a graph pattern in a SPARQL algebra expression. The root graph pattern is the group graph pattern that forms the `WHERE` clause; its translation proceeds as follows:

```
procedure TransformGroupGraphPattern(GroupGraphPattern)

Let FS := ∅
Let G := ∅

For each element E in the GroupGraphPattern
    If E is of the form FILTER(expr)
        FS := FS ∪ expr
    If E is of the form OPTIONAL { P }
        Let A := TransformGroupGraphPattern(P)
        If A is of the form Filter(F, A2)
            G := LeftJoin(G, A2, F)
        else
            G := LeftJoin(G, A, true)
    Else
        Let A := undefined
        If E is of the form TriplesBlock
           Let A := BGP(E)
        If E is of form UnionGraphPattern
           Let A := TransformUnionGraphPattern(E)
        If E is of form GraphGraphPattern
           Let A := TransformGraphGraphPattern(E)
        G := Join(G, A)

If FS is not empty:
  Let X := Conjunction of expressions in FS
  G := Filter(X, G)

The result is G.
end
```

```
procedure TransformUnionGraphPattern(UnionGraphPattern)

Let A := undefined

For each element G in the UnionGraphPattern
    If A is undefined
        A := TransformGroupGraphPattern(G)
    Else
        A := Union(A, TransformGroupGraphPattern(G))

The result is A
end
```

```
procedure TransformGraphGraphPattern(GraphGraphPattern)

If the form is GRAPH IRI GroupGraphPattern
    The result is Graph(IRI, TransformGroupGraphPattern(GroupGraphPattern))
If the form is GRAPH Var GroupGraphPattern
    The result is Graph(Var, TransformGroupGraphPattern(GroupGraphPattern))
end
```

Group graph patterns of a single basic graph pattern *A* become a *Join* of *A* with the empty graph pattern; since the latter is the identity for the *Join* operator, the following simplification step can be performed:

```
Replace Join(∅, A) by A
Replace Join(A, ∅) by A
```

4.4.4 Modifiers translation to SPARQL algebra

A series of steps transform the solution modifiers of a query to algebra operators; these take place after the translation of the graph patterns:

1. `ToList` : turns the multiset into a solution sequence with the same elements and cardinality; this step is always performed

```
Let M := ToList(AlgebraExpression)
```

2. `ORDER BY` : if the query string contains an `ORDER BY` clause

```
Let M := OrderBy(M, list of order comparators)
```

3. `DISTINCT` : if the query string contains a `DISTINCT` clause

```
Let M := Distinct(M)
```

4. `REDUCED` : if the query string contains a `REDUCED` clause

```
Let M := Reduced(M)
```

5. `OFFSET` and `LIMIT` : if the query contains "`OFFSET start`" or "`LIMIT length`"

```
start defaults to 0
length defaults to (size(M)-start)

Let M := Slice(M, start, length)
```

4.4.5 Basic Graph Patterns

Definitions

BGPs and solution mappings Given a basic graph pattern P and a mapping μ such that $var(P) \subseteq dom(\mu)$, $\mu(P) = \bigcup_{t \in P} \mu(t)$, i.e. $\mu(P)$ is the set of triples obtained by replacing the variables in the triples of P according to μ.

BGPs and RDF instance mappings Given a BGP P and a graph G, let σ be an RDF instance mapping that substitutes the blank nodes in P with RDF Terms in G

$$\sigma : blank(P) \rightarrow term(G)$$

$\sigma(P)$ is the basic graph pattern that results from replacing the blank nodes in P according to σ.

BGPs and Pattern instance mappings Given a BGP P, a graph G, an RDF instance mapping $\sigma : blank(P) \rightarrow term(G)$ and a solution mapping μ such that $var(P) \subseteq dom(\mu)$, $\pi(P) = \mu(\sigma(P))$ is the set of triples that results from the application of σ and μ to P.

Basic Graph Pattern evaluation

Given an RDF graph G and a basic graph pattern P, the *evaluation* of P over G, denoted as $[[P]]_G$ is defined as the set of mappings

$$[[P]]_G = \{\mu : V \rightarrow T \mid$$

$$(dom(\mu) = var(P)) \wedge (\exists \sigma : blank(P) \rightarrow term(G) \mid \mu(\sigma(P)) \subseteq G)\}$$

If $\mu \in [[P]]_G$, then μ is a *solution* for P in G. If $P = \emptyset$, then $[[P]]_G = \{\mu_\emptyset\}$, and if $G = \emptyset$, for every $P \neq \emptyset$, $[[P]]_G = \{\emptyset\}$.

Cardinality of Basic Graph Pattern Solutions Given a BGP P and a graph G, the cardinality of a solution $\mu \in [[P]]_G$ is defined as the number of distinct substitutions $\sigma : blank(P) \rightarrow term(G)$ such that $\mu(\sigma(P)) \subseteq G$, formally

$$card_{[[P]]_G}(\mu) = |\{\sigma : blank(P) \rightarrow term(G) \mid \mu(\sigma(P)) \subseteq G\}|$$

4.4.6 SPARQL algebra

This section formally describes the correct interpretation of each operator of the algebra, except for basic graph patterns which were covered in 4.4.5.

Filter semantics

Let Ω be a multiset of solution mappings and $expr$ be an expression. Then

$$Filter(expr, \Omega) = \{\mu : V \rightarrow T \mid \mu \in \Omega \wedge EBV(expr(\mu)) = true\}$$
$$card_{Filter(expr,\Omega)}(\mu) = card_\Omega(\mu)$$

Join semantics

Compatible Mappings Two solution mappings μ_1 and μ_2 are *compatible* if for every variable $?v \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(?v) = \mu_2(?v)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. For example

$$\mu_1 = \begin{cases} ?a \to 10 \\ ?b \to 5 \end{cases} \quad \mu_2 = \begin{cases} ?a \to -1 \\ ?c \to 3 \end{cases}$$

are not compatible, since for $?a \in dom(\mu_1) \cap dom(\mu_2), \mu_1(?a) \neq \mu_2(?a)$, while

$$\mu_1 = \begin{cases} ?a \to 10 \\ ?b \to 5 \end{cases} \quad \mu_2 = \begin{cases} ?a \to 10 \\ ?c \to 3 \end{cases}$$

are compatible, thus their union is a valid mapping:

$$\mu_1 \cup \mu_2 = \begin{cases} ?a \to 10 \\ ?b \to 5 \\ ?c \to 3 \end{cases}$$

Given this definition, two mappings with disjoint domains, i.e. $dom(\mu_1) \cap dom(\mu_2) = \emptyset$, are always compatible. The union of the following mappings

$$\mu_1 = \begin{cases} ?a \to 1 \\ ?b \to 2 \end{cases} \quad \mu_2 = \begin{cases} ?c \to 3 \\ ?d \to 4 \end{cases}$$

is therefore

$$\mu_1 \cup \mu_2 = \begin{cases} ?a \to 1 \\ ?b \to 2 \\ ?c \to 3 \\ ?d \to 4 \end{cases}$$

Join Given two multisets of solution mappings Ω_1 and Ω_2

$$Join(\Omega_1, \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible}\}$$
$$card_{Join(\Omega_1,\Omega_2)}(\mu) = \sum_{\mu=\mu_1\cup\mu_2} card_{\Omega_1}(\mu_1) \cdot card_{\Omega_2}(\mu_2)$$

Union semantics

Given two multisets of solution mappings Ω_1 and Ω_2

$$Union(\Omega_1, \Omega_2) = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$$
$$card_{Union(\Omega_1,\Omega_2)}(\mu) = card_{\Omega_1}(\mu) + card_{\Omega_2}(\mu)$$

LeftJoin semantics

Difference Given two multisets of solution mappings Ω_1 and Ω_2 and an expression $expr$

$$Diff(\Omega_1, \Omega_2, expr) = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2, (\mu \text{ and } \mu' \text{ are not compatible}) \vee EBV(expr(\mu \cup \mu')) = false\}$$
$$card_{Diff(\Omega_1,\Omega_2,expr)}(\mu) = card_{\Omega_1}(\mu)$$

Left Join Given two multisets of solution mappings Ω_1 and Ω_2 and an expression $expr$

$$LeftJoin(\Omega_1, \Omega_2, expr) = Filter(expr, Join(\Omega_1, \Omega_2)) \cup Diff(\Omega_1, \Omega_2, expr)$$
$$card_{LeftJoin(\Omega_1,\Omega_2,expr)}(\mu) = card_{Filter(expr,\ Join(\Omega_1,\Omega_2))}(\mu) + card_{Diff(\Omega_1,\Omega_2,expr)}(\mu)$$

ToList semantics

Given a multiset of solution mappings Ω,

$$ToList(\Omega) = [\mu \in \Omega], \text{in any order}$$
$$card_{ToList(\Omega)}(\mu) = card_{\Omega}(\mu)$$

OrderBy semantics

Given a sequence of solution mappings Ψ and an order condition *cond*

$$OrderBy(\Psi, cond) = [\mu \in \Omega \mid \text{the sequence satisfies } cond]$$
$$card_{OrderBy(\Psi,cond)}(\mu) = card_{\Psi}(\mu)$$

Project semantics

Given a solution mapping $\mu : V \to T$ and a set of variables $W \subseteq V$, the *restriction* of μ to W, denoted by $\mu_{|W}$, is a mapping such that $dom(\mu_{|W}) = dom(\mu) \cap W$ and $\mu_{|W}(?x) = \mu(?x), \forall ?x \in dom(\mu) \cap W$. The *Projection* of a solution sequence Ψ on the variables of W is then:

$$Project(\Psi, W) = [\mu_{|W} \mid \mu \in \Psi]$$
$$card_{Project(\Psi,W)}(\mu) = card_{\Psi}(\mu)$$

Distinct semantics

Given a sequence of solution mappings Ψ

$$Distinct(\Psi) = [\mu \in \Psi]$$
$$card_{Distinct(\Psi)}(\mu) = 1$$

Reduced semantics

Given a sequence of solution mappings Ψ

$$Reduced(\Psi) = [\mu \in \Psi]$$
$$1 \leq card_{Reduced(\Psi)}(\mu) \leq card_{\Psi}(\mu)$$

Slice semantics Given a sequence of solution mappings Ψ, and two natural numbers *start* and *length*

$$Slice(\Psi, start, length) = [\mu \in \Psi \mid \mu = \Psi[start + i], \forall i = 0 \ldots (length - 1)]$$

4.4.7 Expression Evaluation

Let D be an RDF Dataset with active graph G, $D[i]$ the named graph with IRI i in D and let $D[dft]$ be the default graph of D. The set of named graph IRIs is $name(D)$. The *evaluation* of a SPARQL algebra expression P over the RDF Dataset D with active graph G is denoted as $[[P]]^D_G$, and the evaluation of P in the dataset D as $[[P]]^D = [[P]]^D_{D[dft]}$.

The evaluation semantics is defined as follows:

- $[[BGP]]^D_G = [[BGP]]_G$, see also 4.4.5
- $[[Filter(expr, P)]]^D_G = Filter(expr, [[P]]^D_G)$
- $[[Join(P_1, P_2)]]^D_G = Join([[P_1]]^D_G, [[P_1]]^D_G)$
- $[[LeftJoin(P_1, P_2, expr)]]^D_G = LeftJoin([[P_1]]^D_G, [[P_1]]^D_G, expr)$
- $[[Uniom(P_1, P_2)]]^D_G = Union([[P_1]]^D_G, [[P_1]]^D_G)$
- $[[Graph(Iri, P)]]^D_G =$
  - $[[P]]^D_{D[Iri]}$ if $Iri \in name(D)$
  - $\emptyset$ if $Iri \notin name(D)$

- $[[Graph(?x, P)]]^D_G = \bigcup_{g \in name(D)} \left(Join\left([[P]]^D_{D[g]}, \{\mu_{?x \to g}\} \right) \right)$
  where $\{\mu_{?x \to g}\}$ is a multiset that contains a single solution, that maps the variable $?x$ to the graph name g, and where the $\bigcup$ is the SPARQL algebra *Union* operator.
- $[[ToList(P)]]^D = ToList([[P]]^D_{D[dft]})$
- $[[Distinct(L)]]^D = Distinct([[L]]^D)$
- $[[Reduced(L)]]^D = Reduced([[L]]^D)$
- $[[Project(L, vars)]]^D = Project([[L]]^D, vars)$
- $[[OrderBy(L, cond)]]^D = OrderBy([[L]]^D, cond)$
- $[[Slice(L, start, length)]]^D = Slice([[L]]^D, start, length)$

4.5 SPARQL to Relational Algebra translation

As seen in the previous section (in particular in 4.4.5), SPARQL is defined in terms of *solutions*: the formal model describes which properties a mapping μ needs to have to be a solution of a graph pattern. This definition does not tell *how* to find them, given an RDF graph.

Relational algebra on the contrary builds the result from the data through a set of operators: this approach is not only easier to understand and to implement, but moreover makes available to the developers the large body of work on relational engines, in terms of query optimization, transaction isolation and reliability that these mature systems offer.

This topic had been discussed in previous works by Cyganiak [24], Harris [30] and Newman [39].

4.5.1 Relational algebra on multisets

The most evident mismatch between SPARQL and relational algebra is that multisets of solutions of SPARQL are collections of elements that may appear more than once, where relations are pure sets. Even if a formal mapping from one algebra to the other is for this reason impossible, nonetheless real systems usually treat relations as multisets, as the `DISTINCT` keyword in SQL may suggest. In this section, therefore, relational algebra operators are redefined in order to deal with and produce multisets as results.

RDF relations

A multiset of solutions, or *RDF relation*, is a relation that admits duplicates. As in the previous section, they will be described with the set of elements appearing in them and a cardinality function which returns the number of occurrences of each element in the multiset. Each solution mapping is a tuple of this relation. The terms *RDF tuple*, *tuple* and *solution mapping* will be considered synonyms.

If a solution mapping in an RDF relation does not define which is the value of one of the attributes of the heading, then that value is said to be *unbound*. Since in many relational engines unbound values are represented by NULLs, unbound and NULL will be considered synonyms as well.

The columns of an RDF relation are in general of type T, that is the set of all RDF terms. Since in T there are all kind of possible RDF values, for example IRIs, strings and numerics, the columns of an RDF relation are roughly speaking untyped.

An RDF relation Ω may have two distinct sets of columns: $var(\Omega)$ is the set of columns whose name is a variable name, and $blank(\Omega)$ is the set columns whose name is a blank node label. The latter kind of columns are present during a BGP match, where blank nodes act as variables; the final result of a BGP evaluation, however, contains only variables.

An RDF graph is an relation with three columns, *subject*, *predicate* and *object*, or briefly as *s*, *p*, *o*; every triple in the graph is a tuple of this relation. As formally described in 4.4.1, the domain of each column is the set of IRIs and blank nodes for *s*, the set of IRIs for *p*, and the set of all RDF Terms for *o*. An RDF graph is a special case of RDF relation, that does not admit duplicates.

In the following sections the storage schema for RDF triples is a simple table of three columns, in which the RDF Terms are stored directly into it; no dictionaries nor other kind of structures will be considered.

Selection on multisets

The *select* operator is not much different from its pure relational algebra version: given a *selection predicate* and a (multi)set of tuples, it returns those tuples that satisfy the predicate.

The only difference therefore consists of the possibility, in the RDF case, to have duplicate tuples in the operand and in the result.

Projection on multisets

The result of a relational algebra projection is defined as the set obtained when the components of the tuples of the relation are restricted to a subset of those components.

In this section the projection will be only a column-selection, with no duplicate elimination. The semantics of this operator is the same of SPARQL's projection as found in 4.4.6, with the addition that new columns can be built from the values of each tuple; this addition is useful to formally describe joins when unbound values appear in the join columns. Thus, given a solution mapping $\mu : V \to T$, two sets of variables $W \subseteq dom(\mu)$ and U:

$$U = \{u = expr(v_1 \dots v_n) \mid u \in (V \setminus W), v_1 \dots v_n \in dom(\mu)\}$$

i.e. a set of new variables, each built as a (possibly different) expression *expr* of the components of the tuple μ, then $\mu_{|W \cup U}$ is a mapping such that

- $dom(\mu_{|W \cup U}) = (dom(\mu) \cap W) \cup U$
- $\mu_{|W \cup U}(x) = \begin{cases} \mu(x), \text{if } x \in dom(\mu) \cap W \\ expr(v_1 \dots v_n), \text{if } x \in U \end{cases}$

The *projection* of an RDF relation Ω is then:

$$\pi_{W,U}(\Omega) = \{\mu_{|W \cup U} \mid \mu \in \Omega\}$$
$$card_{\pi_{W,U}(\Omega)}(\mu) = card_{\Omega}(\mu)$$

An example of construction of new columns may be the following, given a relation *R* with two columns of integers:

a	b
13	-1
5	2
-0.5	0.5
10	4

The projection permits one to calculate sum and difference of *a* and *b*, and store them in two new columns; in this example $U = \{sum = a + b, diff = a - b\}$

$$\pi_{a,b,sum=a+b,diff=a-b}(R)$$

a	b	sum	diff
13	-1	12	14
5	2	7	3
-0.5	0.5	0	-1
10	4	14	6

Natural join on multisets

The natural join will be used when no unbound values are present on the join attributes. Under this assumption, the difference from the classical natural join is again the cardinality of each tuple in the operands and in the result, that can be greater than one in RDF relations.

The natural join in this situation behaves as normally does in relational engines that operate on multisets: given two RDF relations Ω_L and Ω_R with common attributes $v_1 \dots v_n$, their natural join is

$$\Omega_L \bowtie \Omega_R = \sigma_{\substack{\Omega_L.v_1 = \Omega_R.v1 \\ \vdots \\ \Omega_L.v_n = \Omega_R.v_n}} (\Omega_L \times \Omega_R)$$

As usual for natural joins, the common columns $v_1 \dots v_n$ are not repeated twice in the result, one for the left and one for the right operand as in other joins, but only once.

The cardinality of each element in the result relation conforms to the indication of SPARQL algebra, i.e. the cardinality of a solution mapping μ in a join result is $\sum_{\mu=\mu_L \cup \mu_R} card_{\Omega_L}(\mu_L) \cdot card_{\Omega_R}(\mu_R)$, that is the sum, for each $\mu_L \in \Omega_L$ and $\mu_R \in \Omega_R$ that may generate μ, of the product of the cardinalities of such μ_L and μ_R.

For example, given the relations

a	b
1	11
1	11
2	12
3	13
4	14

b	c
11	21
11	21
12	22
12	22
14	24
15	25

their natural join is:

a	b	c
1	11	21
1	11	21
1	11	21
1	11	21
2	12	22
2	12	22
4	14	24

that is the correct behavior also for SPARQL.

Difference on multisets

The multiset difference is defined here as a "not in" expression: given two RDF relations Ω_L and Ω_R such that $var(\Omega_L) = var(\Omega_R)$ i.e. with same schema, their difference is

$$\Omega_L \setminus \Omega_R = \{\mu \in \Omega_L \mid \mu \notin \Omega_R\}$$
$$card_{\Omega_L \setminus \Omega_R}(\mu) = card_{\Omega_L}(\mu)$$

The multiplicity of the elements does not matter: if a tuple has cardinality equals to two in Ω_L and one in Ω_R, it will not take part in the difference.

4.5.2 Filter translation

As seen in 4.4.7, the *Filter* operator evaluation has to pick from a multiset of solutions those ones that satisfy an expression. This semantics is identical to the the relational *select* operator. Given a expression *expr* and an RDF relation Ω:

$$Filter(expr, \Omega) = \sigma_{expr}(\Omega)$$

Even *Filter* and *select* are conceptually identical, the SPARQL operators have to deal with untyped columns; for example, a filter may select all the triples of an RDF graph where *object* < 24. This implies also that *object* has to be a numeric value.

4.5.3 BGP translation

Single pattern matching

Triple patterns (4.4.1) can be expressed as a selection of the triples of the active graph, followed by a projection and rename. The selection condition is determined by the fixed terms in the triple pattern. Blank nodes act exactly as variables.

For example, given the active graph G, the triple pattern

```
_:person foaf:name ?name
```

becomes

$$\pi_{\substack{\_:person \leftarrow subject \\ ?name \leftarrow object}} (\sigma_{predicate=\texttt{foaf:name}}(G))$$

The evaluation of a triple pattern t on the active graph G is a multiset of solutions denoted by $[[t]]_G$.

Triple pattern join

Given two triple patterns t_1 and t_2 on the active graph G, and their evaluations $\Omega_1 = [[t_1]]_G$ and $\Omega_2 = [[t_2]]_G$, $\Omega_1 \bowtie_{tp} \Omega_2$ is a multiset defined as follows:

$$\Omega_1 \bowtie_{tp} \Omega_2 = \begin{cases} \Omega_1 \times \Omega_2 & \text{if } var(\Omega_1) \cap var(\Omega_2) = \emptyset \wedge blank(\Omega_1) \cap blank(\Omega_2) = \emptyset \\ \Omega_1 \bowtie \Omega_2 & \text{if } var(\Omega_1) \cap var(\Omega_2) \neq \emptyset \vee blank(\Omega_1) \cap blank(\Omega_2) \neq \emptyset \end{cases}$$

where $\times$ is the cartesian product and $\bowtie$ is the natural join as defined in 4.5.1.

Basic graph pattern translation

Given a BGP $P = \{t_1, t_2 \dots t_n\}$ and the active graph G, and denoting as $\Omega_i = [[t_i]]_G$ the evaluation of triple pattern t_i, $[[P]]_G$ can be expressed as:

$$[[P]]_G = \pi_{var(\Omega_1) \cup var(\Omega_2) \dots \cup var(\Omega_n)}(\Omega_1 \bowtie_{tp} \Omega_2 \bowtie_{tp} \dots \bowtie_{tp} \Omega_n)$$

The projection removes all columns whose name is a blank node label.

4.5.4 Join translation

The *Join* definition in SPARQL is much different from the one in relational algebra: two mappings can be part of a *Join* if there is no conflict between them, that is when a common attribute is bound on both sides of the *Join* with different values. Such mappings are called *compatible mappings*. Two disjoint mappings are therefore always compatible, and an unbound value will match with every value, even another unbound one. `NULL` values in relational algebra would cause the join to fail.

Here is an example of SPARQL join between two RDF relations:

?a	?b
20	1
21	2
22	
23	4

?b	?c
1	30
	31
4	32
5	33

The result is

?a	?b	?c
20	1	30
20	1	31
21	2	31
22	1	30
22		31
22	4	32
22	5	33
23	4	31
23	4	32

The SPARQL *Join* can be relationally defined as a subset of the cartesian product where two mappings are compatible; given two RDF relations Ω_L and Ω_R with common attributes $var(\Omega_L) \cap var(\Omega_R) = \{v_1 \dots v_n\}$, the *CompMappings* predicate checks if a tuple of the cross product takes part of the SPARQL *Join*:

$$CompMappings = (\Omega_L.v_1 = \Omega_R.v_1 \vee \Omega_L.v_1 = \texttt{NULL} \vee \Omega_R.v_1 = \texttt{NULL}) \wedge \dots$$

$$\wedge(\Omega_L.v_n = \Omega_R.v_n \vee \Omega_L.v_n = \texttt{NULL} \vee \Omega_R.v_n = \texttt{NULL})$$

The *Join* operator can then be expressed as

$$Join(\Omega_L, \Omega_R) = \pi_{\substack{v_1 = (\Omega_L.v_1 \neq \texttt{NULL}\ ?\ \Omega_L.v_1 : \Omega_R.v_1) \\ \vdots \\ v_n = (\Omega_L.v_n \neq \texttt{NULL}\ ?\ \Omega_L.v_n : \Omega_R.v_n) \\ (var(\Omega_L) \cup var(\Omega_R)) \setminus \{v_1 \dots v_n\}}} \left(\sigma_{CompMappings}\,(\Omega_L \times \Omega_R)\right)$$

The projection builds the column v_i from $\Omega_L.v_i$ and $\Omega_R.v_i$ picking from them the bound value, if any; it then selects all the columns that are not in $var(\Omega_L) \cap var(\Omega_R)$. In case $var(\Omega_L) \cap var(\Omega_R) = \emptyset$ no selection nor projection takes place and only the cartesian product is performed.

Another semantics for SPARQL joins may be defined when the join attribute is only one. In this case the cartesian product can be limited only to those tuples that present an unbound value on the join attribute.

Given two RDF relations Ω_L and Ω_R with common attribute v

$$Join(\Omega_L, \Omega_R) = (\Omega_L \bowtie \Omega_R) \cup$$

$$\pi_{\substack{v \leftarrow \Omega_R.v \\ (var(\Omega_L) \cup var(\Omega_R)) \setminus \{v\}}} (\sigma_{v=NULL}(\Omega_L) \times \sigma_{v \neq NULL}(\Omega_R)) \cup$$

$$\pi_{\substack{v \leftarrow \Omega_L.v \\ (var(\Omega_L) \cup var(\Omega_R)) \setminus \{v\}}} (\sigma_{v \neq NULL}(\Omega_L) \times \sigma_{v=NULL}(\Omega_R)) \cup$$

$$\pi_{\substack{v \leftarrow \Omega_L.v \\ (var(\Omega_L) \cup var(\Omega_R)) \setminus \{v\}}} (\sigma_{v=NULL}(\Omega_L) \times \sigma_{v=NULL}(\Omega_R))$$

where *Join* is the natural join and $\cup$ is a union that does not drop duplicates. The last projection can select indifferently $\Omega_L.v$ or $\Omega_R.v$, since both present NULLs on each row.

4.5.5 LeftJoin translation

Diff

The SPARQL *Diff* operator, given two relations Ω_L and Ω_R and an expression *expr*, returns those mappings μ of Ω_L that either are not compatible with *all* the mappings μ' of Ω_R, or for which the evaluation of *expr* is false for all the mappings $\mu \cup \mu'$.

In SPARQL these united mappings do not replicate the common attributes, while in a relational context the expression *expr* has to be evaluated against tuples that replicate the common attributes twice, one for the left tuple μ and one for the right one, μ'. If the expression is defined on one or more of these, the expression has to pick the bound value, if any. Every occurrence of an attribute x that participates in the join must be substituted with the ternary expression $(\Omega_L.x \neq \texttt{NULL} \ ? \ \Omega_L.x : \Omega_R.x)$.

For instance, the expression:

$$a + b > 4$$

with both a and b join attributes, has to be rewritten as

$$(\Omega_L.a \neq \texttt{NULL} \ ? \ \Omega_L.a : \Omega_R.a) + (\Omega_L.b \neq \texttt{NULL} \ ? \ \Omega_L.b : \Omega_R.b)$$

The expression *expr* modified in such manner will be denoted with *expr'*.

The SPARQL *Diff* can the be expressed with:

$$Diff(\Omega_L, \Omega_R, expr) = \Omega_L \setminus \pi_{var(\Omega_L)} \left(\sigma_{CompMappings \wedge expr'} (\Omega_L \times \Omega_R) \right)$$

The second operand of the difference builds a multiset that contains those tuples of Ω_L that are compatible at least with one tuple of Ω_R, and among these, those for which at least one evaluation of *expr* is true. The result of the difference contains therefore only tuples that are either incompatible with all the tuples of Ω_R or for which the evaluation of *expr* is always false for all their combinations with the compatible tuples of Ω_R.

LeftJoin

The *LeftJoin* is expressed in the SPARQL standard as a union of a *Join* and a *Diff*. Having already a definition of these operators in relational algebra, the *LeftJoin* translation is trivial:

$$LeftJoin(\Omega_L, \Omega_R, expr) =$$

$$\left(\pi_{\substack{v_1 = (\Omega_L.v_1 \neq \texttt{NULL} \ ? \ \Omega_L.v_1 : \Omega_R.v_1) \\ \vdots \\ v_n = (\Omega_L.v_n \neq \texttt{NULL} \ ? \ \Omega_L.v_n : \Omega_R.v_n) \\ (var(\Omega_L) \cup var(\Omega_R)) \setminus \{v_1 \ldots v_n\}}} \left(\sigma_{CompMappings \wedge expr,} (\Omega_L \times \Omega_R)\right) \right)$$

$$\cup$$

$$\left(\Omega_L \setminus \pi_{var(\Omega_L)} \left(\sigma_{CompMappings \wedge expr,} (\Omega_L \times \Omega_R)\right)\right)$$

4.5.6 Union translation

SPARQL union cannot be a result of any of the classical relational operators, but nonetheless there are already some implementations that define an `OUTER UNION`, as described in [22]. The difference with relational union is that in this latter the schema of the tables being united must be the same. In SPARQL unions this condition is not needed: all the tuples of both relations take part in the result without duplicate elimination, and missing information is filled with unbound values. For example the union of the following tables

?a	?b
1	21
2	22
3	23
4	

?a	?c
4	
5	32
6	33
	34

is

?a	?b	?c
1	21	
2	22	
3	23	
4		
4		
5		32
6		33
		34

4.5.7 Graph expression translation

Graph expressions on a fixed named graph of the dataset are translated without any effort, since these kind of SPARQL algebra expressions just change the active graph.

The situation in which a variable is present as graph IRI is different: in this case the evaluation of the graph pattern contained in the graph expression has to be performed once for each named graph in the dataset. In 4.4.7 it has been shown that the evaluation of a Graph pattern in this case is

$$[[Graph(?x, P)]]^D_G = \bigcup_{g \in name(D)} \left(Join \left([[P]]^D_{D[g]}, \{\mu_{?x \to g}\} \right) \right)$$

A relational translation is still a union of all evaluations against each named graph; only the *Join* has to be translated.

There are two possible cases: $?x \in var(P)$, or $?x \notin var(P)$

In the first case, the join on $?x$ can follow the definition given in 4.5.4 for joins on a single attribute:

$$[[Graph(?x, P)]]^D_G =$$

$$\bigcup_{g \in name(D)} \left(\left([[P]]^D_{D[g]} \bowtie \{\mu_{?x \to g}\} \right) \cup \pi_{\substack{?x \leftarrow ?x_R \\ var(P) \setminus \{?x\}}} \left(\sigma_{?x = \texttt{NULL}} \left([[P]]^D_{D[g]} \right) \times \{\mu_{?x \to g}\} \right) \right)$$

where x_R is the x column of the cross product given by $\{\mu_{?x \to g}\}$.

In the second case the only operation to perform is to extend the RDF relation returned by $[[P]]^D_{D[g]}$ with a column containing the graph IRI g for each tuple; relationally, this can be done with a cartesian product:

$$[[Graph(?x, P)]]^D_G = \bigcup_{g \in name(D)} \left([[P]]^D_{D[g]} \times \{\mu_{?x \to g}\} \right)$$

Chapter 5

RDF storage in MonetDB

5.1 Data structures

As discussed in Chapter 3, the most conventional and natural manner to store RDF triples is a three-column relational table, with dictionary compression for IRIs and literals.

The solution proposed in MonetDB follows this approach, but materializes the triples table six times, each sorted on one of the six permutations of the columns. A single dictionary table maps integers to the RDF terms for all the views.

Figures 5.1, 5.2 and 5.3 show how some example RDF data is saved in MonetDB.

5.1.1 Data tables

Each of the six data tables is represented in MonetDB by three binary tables of type `(:void, :oid)`, one for each column. The virtual `oid` sequence identifies the row number starting from zero, while the oid column actually stores the ids of the RDF terms.

The first column of every table is sorted by ascending id values; those triples that present the same value on the first column are arranged according to the ids of the second. Finally, triples with same values on the first two columns are sorted according to the third.

5.1.2 Dictionary table

Since the id order has to reflect the one of the RDF terms, also the dictionary has to be sorted, so that if an id is smaller than another one then or the two terms represented by the ids are not comparable (e.g. because they are different in type) or the first term is smaller than the second.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:f0 foaf:name "Alice" .
_:f0 foaf:mailboxes _:b0 .
_:b0 rdf:first mailto://alice@isp.com .
_:b0 rdf:rest _:b1 .
_:b1 rdf:first mailto://alice@foaf.org .
_:b1 rdf:rest rdf:nil .
```

Figure 5.1: Example RDF data

S	P	O
00	03	09
00	05	01
01	03	08
01	05	04
02	06	00
02	07	10

P	S	O
03	00	09
03	01	08
05	00	01
05	01	04
06	02	00
07	02	10

O	S	P
00	02	06
01	00	05
04	01	05
08	01	03
09	00	03
10	02	07

S	O	P
00	01	05
00	09	03
01	04	05
01	08	03
02	00	06
02	10	07

P	O	S
03	08	01
03	09	00
05	01	00
05	04	01
06	00	02
07	10	02

O	P	S
00	06	02
01	05	00
04	05	01
08	03	01
09	03	00
10	07	02

Figure 5.2: The six materialized views

Id	RDF Term
00	b0
01	b1
02	f0
03	http://www.w3.org/1999/02/22-rdf-syntax-ns#first
04	http://www.w3.org/1999/02/22-rdf-syntax-ns#nil
05	http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
06	http://xmlns.com/foaf/0.1/mailboxes
07	http://xmlns.com/foaf/0.1/name
08	mailto://alice@foaf.org
09	mailto://alice@isp.com
10	Alice

Figure 5.3: The dictionary table

For the same reason terms with the same value must be mapped to the same integer, even if the lexical form or the type can be different (e.g. the numbers ‘5’ and ‘5.0’).

In MonetDB this table is a single BAT of type `(:void, :str)`, lexicographically ordered on the tail. This forces one to represent every RDF term as a string and to make the lexicographical order be equivalent to the natural one; for numeric values in particular, this situation required some expedient.

Order of the RDF terms

The SPARQL standard defines an ordering among the three sets that compose the superset of the RDF terms: blank nodes precede IRIs, which are sorted before the literals ([44], section 9.1). The dictionary table, therefore, needs to force the order of these terms in such manner.

The way this is achieved in Monet is to prefix every term with a single character that identifies a section in the dictionary: the relative order of the sections is thus imposed by the prefixes.

The set of literals is divided into several sections; on the one hand comparable terms are grouped together, while on the other sections with same effective boolean value (see 4.4.1) are placed one next to the other. This strategy lets one to implement the EBV function as a range check (see also figure 5.4. Finally, different sections that contain comparable terms are sorted properly (e.g. positive numerics follow negative numerics and zero).

The following is the list of the sections in the dictionary, with their respective prefixes:

- Blank nodes - ‘0’
- IRIs - ‘1’
- Negative infinity - ‘2’
- Negative numerics - ‘3’
- Boolean false - ‘4’
- Numeric zero - ‘5’
- Not a Number (NaN) - ‘6’
- Empty string - ‘7’
- Empty string with language tag - ‘8’

- Strings - ‘9’
- Strings with language tag - ‘A’
- Boolean true - ‘B’
- Positive Numerics - ‘C’
- Positive infinity - ‘D’
- Datetime values - ‘E’
- XML literals - ‘F’
- Literals with unsupported datatypes - ‘G’

Figure 5.4 is an example dictionary that contains at least one element in each section. It should be noted that sections that contain only a single element, for example negative and positive infinity, do not need to store anything more than the prefix itself: they may not even be present if the RDF document from which data is loaded does not contain such values.

Storing type information of literals

The dictionary does not save the XML type of numeric literals and strings; numeric values can be of type `xsd:integer` , `xsd:float` , `xsd:decimal` and many more (see [16] for all XML datatypes), while what in the dictionary is categorized as “string” can be a simple literal (a plain literal with no language tag) or a literal of type `xsd:string` .

This information cannot be lost, but cannot even be stored in the dictionary itself: literals with same value but different in type must be mapped to the same identifier. The type information is therefore stored together with the data tables, thus for each permutation of the S, P and O columns an additional BAT *T* of type `(:void, :bte)` (where `:bte` is the smallest type in Monet) is present.

Figure 5.5 shows a complete data table complete with its T column, but unnormalized for ease of reading.

String literals

The dictionary distinguishes four kinds of strings: empty string, empty strings with language tag and non-empty strings, with or without language tag.

The language tag is placed in front of the lexical form, so that strings of the same language are sorted together. This is an extension to the SPARQL standard which does not define an order between literals with a language tag.

	Id	RDF term
	00	0blank0
	01	0blank1
	02	1http://example.com/iri1
	03	1http://example.com/iri2
EBV true	04	2
	05	33febfffffffffff
	06	33ff3fffffffffff
EBV false	07	4
	08	5
	09	6
	10	7
	11	8en
	12	8it
EBV true	13	9Another plain literal or string
	14	9Plain literal or string
	15	Aen@English literal
	16	Ait@Stringa in italiano
	17	B
	18	Cc014000000000000
	19	Cc02b000000000000
	20	D
	21	E800b31fa01ee6280
	22	F<xmlTag name="xml literal"/>
	23	Ghttp://types.org/custom^lexicalForm

Figure 5.4: Dictionary table in detail

As said in the previous subsection, simple literals and typed strings (literals with type `xsd:string`) are grouped together and considered simply “strings”. Two string literals with same lexical form, one typed and one not, are considered equals and mapped to the same integer.

At the moment the current SPARQL documentation ([44], section 9.1) contains an inconsistency on how plain literals and typed strings should be sorted: it states both that they are not comparable and that the typed strings follow plain literals. A discussion on this topic with one of SPARQL’s authors can be found in the W3C mailing list [6].

In MonetDB this problem is faced by assigning the same id to equal strings (typed or not), but at the same time giving a lower type code (in the T column) to plain literals, so that they are sorted first.

S	P	O	T
id:1	dc:value	“string”	plain
id:2	dc:value	“string”	xsd:string
id:3	dc:value	5	xsd:integer
id:4	dc:value	5	xsd:double
id:5	dc:value	5	xsd:decimal

Figure 5.5: Type column for the SPO table

Numeric literals

A numeric value can be expressed in many forms, for example the literals `10` , `10.0` , `"10"^^xsd:integer` and `"10.0"^^xsd:float` are different representations of the same number.

In order to assign to numeric literals with identical values the same oid in the dictionary, they are all (integers, decimal, floats etc.) converted to a double-precision representation and then converted to strings of hexadecimal characters.

The lexicographical order of these hexadecimal strings, however, is not equivalent to the natural one: positives would be lower than negatives, since the most significant bit, the sign bit, is zero for positives; moreover negatives would be sorted inversely, as shown in figure 5.6.

Value	Double representation
-0.50	bfe0000000000000
-0.45	bfdcccccccccccd
-0.40	bfd999999999999a
-0.35	bfd6666666666667
-0.30	bfd3333333333334
-0.25	bfd0000000000001
-0.20	bfc999999999999c
-0.15	bfc3333333333336
-0.10	bfb999999999999f
-0.05	bfa99999999999a4
0.05	3fa9999999999990
0.10	3fb9999999999995
0.15	3fc3333333333331
0.20	3fc9999999999998
0.25	3fcfffffffffffe
0.30	3fd3333333333332
0.35	3fd6666666666665
0.40	3fd9999999999998
0.45	3fdcccccccccccb
0.50	3fdfffffffffffe

Double representation	Value
3fa9999999999990	0.05
3fb9999999999995	0.10
3fc3333333333331	0.15
3fc9999999999998	0.20
3fcfffffffffffe	0.25
3fd3333333333332	0.30
3fd6666666666665	0.35
3fd9999999999998	0.40
3fdcccccccccccb	0.45
3fdfffffffffffe	0.50
bfa99999999999a4	-0.05
bfb999999999999f	-0.10
bfc3333333333336	-0.15
bfc999999999999c	-0.20
bfd0000000000001	-0.25
bfd3333333333334	-0.30
bfd6666666666667	-0.35
bfd999999999999a	-0.40
bfdcccccccccccd	-0.45
bfe0000000000000	-0.50

Figure 5.6: Numeric values and their double representation, sorted on value on the left and on the representation on the right

A simple solution consists in XORing the 64 bits of the double representation. Two different bit masks are needed, one for positive and one for negative numbers.

The first has just to invert the first bit (i.e. `8000000000000000` in hexadecimal digits), while the second and to revert every bit (i.e. `ffffffffffffffff`). The result is shown in figure 5.7.

Value	Representation
-0.50	401fffffffffffff
-0.45	4023333333333332
-0.40	4026666666666665
-0.35	4029999999999998
-0.30	402cccccccccccb
-0.25	402ffffffffffffe
-0.20	4036666666666663
-0.15	403ccccccccccc9
-0.10	4046666666666660
-0.05	405666666666665b
0.05	bfa9999999999990
0.10	bfb9999999999995
0.15	bfc3333333333331
0.20	bfc9999999999998
0.25	bfcffffffffffffe
0.30	bfd3333333333332
0.35	bfd6666666666665
0.40	bfd9999999999998
0.45	bfdccccccccccccb
0.50	bfdffffffffffffe

Figure 5.7: Representation order reflects value order

`xsd:dateTime` literals

Literals with XML type `xsd:dateTime` are stored as a concatenation of Monet's timestamp (a 64-bit binary format), converted to a string of hexadecimal characters, with a character string that stores the fractions of a second. The concatenation is needed because the fractions of a second in Monet have millisecond precision, while they may contain an unspecified number of digits in XML's type system.

If a timezone information is present, the timestamp is converted and stored in UTC, and when returned by a query it has to be converted back to the client's timezone. If this information is not present, the timestamp is saved and returned "as is", without any conversion. In order to distinguish between datetimes with and without timezone, the RDF module defines two internal subtypes, whose code is stored in the T column of the data table.

Also in this case, two datetime literals with same value are mapped to the same numeric identifier. Two literals with same date and time values, one without timezone information and the other in UTC are considered equals as well.

So as the double-precision representation of numbers, also Monet's timestamps need to be XORed with a bitmask to let the lexical order be equivalent to the natural one. The mask in this case has just to reverse the first bit.

Figure 5.8 shows some datetime literals and their representations in the dictionary table, without prefix. Literals with same representation are obviously mapped to the same identifier.

XML dateTime	Representation
2007-08-07T21:15:00	800b304c048f4c20
2008-07-07T21:15:00	800b319b048f4c20
2008-08-06T21:15:00	800b31b9048f4c20
2008-08-07T20:15:00	800b31ba04585da0
2008-08-07T21:14:00	800b31ba048e61c0
2008-08-07T21:14:59	800b31ba048f4838
2008-08-07T21:14:59.999	800b31ba048f4838999
2008-08-07T21:15:00	800b31ba048f4c20
2008-08-07T21:15:00+00:00	800b31ba048f4c20
2008-08-07T22:15:00+01:00	800b31ba048f4c20
2008-08-07T20:15:00-01:00	800b31ba048f4c20
2008-08-07T21:15:00.0123	800b31ba048f4c200123
2008-08-07T21:15:00.123	800b31ba048f4c20123
2008-08-07T21:15:00.987+00:00	800b31ba048f4c20987
2008-08-07T21:15:00.9876+00:00	800b31ba048f4c209876

Figure 5.8: Representation order reflects value order

Unsupported datatypes and `rdf:XMLLiteral`

Numeric, boolean, string and datetime literals have a special treatment since they are required by SPARQL's standard operators, but other kind of literals may appear in RDF documents as well.

In general, they are stored in section 'G', with the datatype IRI as a prefix of the lexical form; in this manner literals of the same type are grouped together and sorted on the lexical form.

Literals of type `rdf:XMLLiteral` , that are XML strings embedded in an RDF/XML document, are stored separately in section 'F'; hence they do not need to include the datatype IRI together with the lexical form.

5.2 Importing algorithm

The RDF document importing is performed in two phases; in the first one the document is parsed and a first dictionary and data table (with columns S, P, O and T) are filled incrementally. In the second phase the dictionary is sorted and the six permutations of the data tables are created.

5.2.1 First phase

MonetDB uses the Raptor Library [12] to parse RDF. This library invokes a callback function for each triple of the document, that can be expressed in almost every RDF serialization language.

Each time the triples handler function is called, MonetDB inserts a mapping in the dictionary for every term in the triple not previously encountered, and adds a row in the data table with the ids of terms and the one-byte code of the object's type.

Figure 5.10 shows the situation after the first phase of the importing of the RDF data in figure 5.1, which is copied to figure 5.9 for ease of reading (T column is omitted).

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:f0 foaf:name "Alice" .
_:f0 foaf:mailboxes _:b0 .
_:b0 rdf:first mailto://alice@isp.com .
_:b0 rdf:rest _:b1 .
_:b1 rdf:first mailto://alice@foaf.org .
_:b1 rdf:rest rdf:nil .
```

Figure 5.9: Example RDF data

Id	RDF Term
00	0f0
01	1http://xmlns.com/foaf/0.1/name
02	9Alice
03	1http://xmlns.com/foaf/0.1/mailboxes
04	0b0
05	1http://www.w3.org/1999/02/22-rdf-syntax-ns#first
06	1mailto://alice@isp.com
07	1http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
08	0b1
09	1mailto://alice@foaf.org
10	1http://www.w3.org/1999/02/22-rdf-syntax-ns#nil

S	P	O
00	01	02
00	03	04
04	05	06
04	07	08
08	05	09
08	07	10

Figure 5.10: Dictionary and data table after the first phase of importing

5.2.2 Second phase – sorting

Sorting the dictionary

The first step of the second phase sorts the dictionary on the tail. As said above, this is a simple lexicographical sort, for which MonetDB is highly optimized. Since the head of the BAT is not a sequence anymore, it cannot be of type `:void` but it is rather materialized and of type `:oid` , as shown in figure 5.11.

Id	RDF Term
04	0b0
08	0b1
00	0f0
05	1http://www.w3.org/1999/02/22-rdf-syntax-ns#first
10	1http://www.w3.org/1999/02/22-rdf-syntax-ns#nil
07	1http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
03	1http://xmlns.com/foaf/0.1/mailboxes
01	1http://xmlns.com/foaf/0.1/name
09	1mailto://alice@foaf.org
06	1mailto://alice@isp.com
02	9Alice

Figure 5.11: Dictionary just after sorting

Id translation

The final dictionary in figure 5.12 has the same tail of the BAT in figure 5.11, but with a `:void` sequence on the head; a mapping of the ids created during the RDF document parsing and the final ones is therefore needed. This mapping is created as a `void` view (see 2.4) of the head of the dictionary in figure 5.11, as shown in figure 5.13.

This BAT is joined with the S, P and O BATs, creating a new triples table with the final identifiers, shown in figure 5.14.

Sorting the triples table

The last step of the process creates the six copies of the triples table. The algorithm sorts one of three columns, then refines the order of the other two columns twice, one for each permutation of the these latters. For example, if it sorts first the S column, then it refines the order on P and subsequently on O , creating a triples table ordered on SPO; then it refines the order on O and P, creating the SOP table.

Id	RDF Term
00	0b0
01	0b1
02	0f0
03	1http://www.w3.org/1999/02/22-rdf-syntax-ns#first
04	1http://www.w3.org/1999/02/22-rdf-syntax-ns#nil
05	1http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
06	1http://xmlns.com/foaf/0.1/mailboxes
07	1http://xmlns.com/foaf/0.1/name
08	1mailto://alice@foaf.org
09	1mailto://alice@isp.com
10	9Alice

Figure 5.12: Final dictionary BAT

Old id	New id
04	00
08	01
00	02
05	03
10	04
07	05
03	06
01	07
09	08
06	09
02	10

Figure 5.13: Id translation BAT

This is done three times, each of them sorts one of three column and refines the other two twice.

The first unsorted copy of the triple table, the one shown figure 5.14 is finally deallocated.

5.3 Conclusions

Summarizing the concepts presented in this chapter, the key features of the physical layer adopted in Monet for RDF storage are:

- Equal RDF terms have same identifiers

S	P	O
02	07	10
02	06	00
00	03	09
00	05	01
01	03	08
01	05	04

Figure 5.14: Triples table with final ids

- The order of two comparable RDF terms is the same of their identifiers
- The data is sorted in all possible ways
- Data with same EBV is grouped together

These characteristics permit to obtain several advantages, in terms of searches and joins.

Fast searches and EBV predicate evaluations

One of the most important advantages is that every string search in the dictionary and every id search in the six data tables are performed on sorted data, allowing to implement range selects as *slice views*, as discussed in section 2.4.

An id lookup in the dictionary can be performed by position, i.e. the fastest way possible; every join between the data tables and the dictionary is therefore a positional join.

The search of values in the data tables, moreover, can be performed in the identifier space, thus not requiring a join with the dictionary; a SPARQL triple pattern like `?s foaf:name "Alice"` , for example, can be implemented as a search in the POS or OPS table of those triples that have the id of `foaf:name` in P and the id of `"Alice"` in O. The join with the dictionary is needed only at the end of query execution, before returning the result.

Fast joins

As with searches, value-based joins can be performed in the id space, since identifier equality assures equality of the RDF terms, and conversely identifier inequality assures inequality of the terms. As in any relational engine, MonetDB joins ids much faster than strings.

One of the greatest issues of SPARQL is that requires many self equi-joins of the triples table, that can be an important bottleneck in many relational engines, as

discussed in [9], [21]. The approach adopted in MonetDB permits one to perform merge-joins in many situations, or to have very often at least one of the operands sorted on the join column; also a nested loop would perform much faster in this case if this table is chosen as inner table.

The following two example queries permit one to better guess the above advantages; all SPARQL queries are translated to relational algebra, and their execution explained. The *select* operations make use of the *Id(RDF-Term t)* function, that returns the identifier of the given RDF term, or a special value to suggest that the term is not present in the dictionary; in this case the *select* returns an empty set. The final join with the dictionary to convert back ids is omitted.

This simple query can found in the current SPARQL specification:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

In algebra it can be expressed as:

$$\pi_{\substack{?name \\ ?mbox}} \left(\pi_{\substack{?x \leftarrow s \\ ?name \leftarrow o}} (\sigma_{p = Id(\texttt{foaf:name})}(\text{PSO})) \bowtie \pi_{\substack{?x \leftarrow s \\ ?mbox \leftarrow o}} (\sigma_{p = Id(\texttt{foaf:mbox})}(\text{PSO})) \right)$$

The selections on P are very fast since the data in PSO is sorted on that column and are thus implemented as slice views. Choosing the PSO table, moreover, assures that the S column is sorted for a given value of P, that is id of `foaf:name` on the left table of the join and the id of `foaf:mbox` on the right. Since the join is on S, a merge-join (on the ids) can be performed.

The following query looks for the titles of the resources referenced by those subjects ?*s*, whose title is “RDF”:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/> .
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
SELECT ?seeAlso
WHERE {
        ?s dc:title "RDF".
        ?s rdfs:seeAlso :?x .
        ?x dc:title ?seeAlso
}
```

It can be translated as

$$\left(\sigma_{\substack{o = Id(\text{“RDF”}) \\ p = Id(\texttt{dc:title})}} (\text{OPS}) \bowtie \pi_{?x \leftarrow o} (\sigma_{p = Id(\texttt{rdfs:seeAlso})}(\text{PSO})) \right) \bowtie \pi_{?x \leftarrow s} (\sigma_{p = Id(\texttt{dc:title})}(\text{POS}))$$

Again, all the *select* operations are on sorted data, and the inmost join is performed on two sorted columns. Even if the left operand of the last join is not

sorted on ?*x*, nevertheless the right one is, thus it can be used efficiently as the inner table of a nested loop join; if the left operand is not enormous, moreover, Monet can still decide to sort it. Another benefit of MonetDB's approach in this example is that self joins are avoided, since the three triple patterns operate on three distinct tables.

Although these examples are not barely comprehensive and more exhaustive experiments have to be carried out, the possible benefits of the RDF storage technique adopted in MonetDB, and especially of the exploitation of the sorted and dense properties, should be clear.

Drawbacks

The advantages of having the data tables replicated six times are followed by their disk occupation. For these reason it is already planned that MonetDB/SPARQL will adopt lightweight compression of the data, following its application in MonetDB/X100 [52, 54], and for which an extensive literature is present in the database field [23, 28, 32]).

The dense identifier set makes updates impossible to be merged seamlessly in the proposed design, but they can be rather kept in separate a *Delta* structure that keeps track of all the modifications. If the *Delta* grows until the performance gets noticeably worse, it will be possible to merge it, creating a new dictionary and a new set of data tables.

The particular way the data is stored in the dictionary requires that even the most trivial functions have to be reimplemented in order to deal with this representation: a simple SPARQL `FILTER` like `?o < 50` , for example, has to be RDF-specific: the implementation has to transform 50 in its hexadecimal string representation, find the id of the nearest value to it in the dictionary, `max` , and the smallest numeric value, `min` , and finally express the filter as a range select of ids greater than `min` and smaller than `max` , or equal to the id of zero.

Since different RDF graphs are stored in different sets of tables, a query that involves multiple graphs cannot benefit from all the advantages described above, since the assumptions on the order and equality of the identifiers are lost. Even if graph-specific operations can be optimized in this sense, joining data from different graphs cannot be done in the id space. For this reason it will be possible for a database administrator to merge two or more graphs if a set of queries is frequently executed against those graphs.

Appendix A

Materialized view choice

This appendix shows which are the best materialized views that the MonetDB/-SPARQL optimizer can choose when it has to evaluate a Basic Graph Pattern. The target is to perform selections and joins on sorted columns.

The first two sections of the appendix examine all BGPs of one and two triple patterns respectively, dividing them according to the number of fixed terms in them. The considered kind of joins are *subject-subject*, *subject-object* and *object-object*.

The last section shows how this information can be used to plan the execution of more complex queries.

A.1 Single triple pattern BGPs

In this simple situation the optimizer would chose the view on the basis of the `ORDER BY` clause or, if other group graph patterns are present, on the basis of the kind of join that have to performed higher in the execution plan.

A.1.1 3 variables

View	Result order
SPO	(?*s*, ?*p*, ?*o*)
SOP	(?*s*, ?*o*, ?*p*)
PSO	(?*p*, ?*s*, ?*o*)
POS	(?*p*, ?*o*, ?*s*)
OSP	(?*o*, ?*s*, ?*p*)
OPS	(?*o*, ?*p*, ?*s*)

A.1.2 2 variables

- { f ?p ?o }

View	Result order
SPO	(?p, ?o)
SOP	(?o, ?p)

- { ?s f ?o }

View	Result order
PSO	(?s, ?o)
POS	(?o, ?s)

- { ?s ?p f }

View	Result order
OSP	(?s, ?p)
OPS	(?p, ?s)

A.1.3 1 variable

The choice here depend on the selectivity of the f_0 and f_1 constraints. In the third pattern for instance, the constraint on O is usually much more selective than the one on P, making therefore OPS a better choice.

- { f_0 f_1 ?o }

View	Result order
SPO or PSO	(?o)

- { f_0 ?p f_1 }

View	Result order
SOP or OSP	(?p)

- { ?s f_0 f_1 }

View	Result order
POS or OPS	(?s)

A.2 BGPs of two triple patterns

In many cases in this section the result can be ordered in two different ways. This depends on the merge join behaviour when the same `oid` is encountered multiple times in the join column ?x on both sides, since in these ranges the algorithm performs a nested-loop: for each `oid` of the outer table, it loops on the inner one until a different `oid` is found. In the result, the outer table has its columns sorted before those of the inner one.

The optimizer can choose which should be the outer table on the basis its needs.

A.2.1 No constraints

BGP	View 1	View 2	Result order
{ ?x ?p_1 ?o_1 . ?x ?p_2 ?o_2 }	SPO	SPO	(?x, ?p_1, ?o_1, ?p_2, ?o_2) or (?x, ?p_2, ?o_2, ?p_1, ?o_1)
{ ?x ?p_1 ?o_1 . ?x ?p_2 ?o_2 }	SPO	SOP	(?x, ?p_1, ?o_1, ?o_2, ?p_2) or (?x, ?o_2, ?p_2, ?p_1, ?o_1)
{ ?x ?p_1 ?o_1 . ?x ?p_2 ?o_2 }	SOP	SPO	(?x, ?o_1, ?p_1, ?p_2, ?o_2) or (?x, ?p_2, ?o_2, ?o_1, ?p_1)
{ ?x ?p_1 ?o_1 . ?x ?p_2 ?o_2 }	SOP	SOP	(?x, ?o_1, ?p_1, ?o_2, ?p_2) or (?x, ?o_2, ?p_2, ?o_1, ?p_1)
{ ?x ?p_1 ?o_1 . ?s_2 ?p_2 ?x }	SPO	OPS	(?x, ?p_1, ?o_1, ?p_2, ?s_2) or (?x, ?p_2, ?s_2, ?p_1, ?o_1)
{ ?x ?p_1 ?o_1 . ?s_2 ?p_2 ?x }	SPO	OSP	(?x, ?p_1, ?o_1, ?s_2, ?p_2) or (?x, ?s_2, ?p_2, ?p_1, ?o_1)
{ ?x ?p_1 ?o_1 . ?s_2 ?p_2 ?x }	SOP	OPS	(?x, ?o_1, ?p_1, ?p_2, ?s_2) or (?x, ?p_2, ?s_2, ?o_1, ?p_1)
{ ?x ?p_1 ?o_1 . ?s_2 ?p_2 ?x }	SOP	OSP	(?x, ?o_1, ?p_1, ?s_2, ?p_2) or (?x, ?s_2, ?p_2, ?o_1, ?p_1)

BGP	View 1	View 2	Result order
{ $?s_1$ $?p_1$ $?x$. $?x$ $?p_2$ $?o_2$ }	OPS	SPO	($?x$, $?p_1$, $?s_1$, $?p_2$, $?o_2$) or ($?x$, $?p_2$, $?o_2$, $?p_1$, $?s_1$)
{ $?s_1$ $?p_1$ $?x$. $?x$ $?p_2$ $?o_2$ }	OPS	SOP	($?x$, $?p_1$, $?s_1$, $?o_2$, $?p_2$) or ($?x$, $?o_2$, $?p_2$, $?p_1$, $?s_1$)
{ $?s_1$ $?p_1$ $?x$. $?x$ $?p_2$ $?o_2$ }	OSP	SPO	($?x$, $?s_1$, $?p_1$, $?p_2$, $?o_2$) or ($?x$, $?p_2$, $?o_2$, $?s_1$, $?p_1$)
{ $?s_1$ $?p_1$ $?x$. $?x$ $?p_2$ $?o_2$ }	OSP	SOP	($?x$, $?s_1$, $?p_1$, $?o_2$, $?p_2$) or ($?x$, $?o_2$, $?p_2$, $?s_1$, $?p_1$)
{ $?s_1$ $?p_1$ $?x$. $?s_2$ $?p_2$ $?x$ }	OPS	OPS	($?x$, $?p_1$, $?s_1$, $?p_2$, $?s_2$) or ($?x$, $?p_2$, $?s_2$, $?p_1$, $?s_1$)
{ $?s_1$ $?p_1$ $?x$. $?s_2$ $?p_2$ $?x$ }	OPS	OSP	($?x$, $?p_1$, $?s_1$, $?s_2$, $?p_2$) or ($?x$, $?s_2$, $?p_2$, $?p_1$, $?s_1$)
{ $?s_1$ $?p_1$ $?x$. $?s_2$ $?p_2$ $?x$ }	OSP	OPS	($?x$, $?s_1$, $?p_1$, $?p_2$, $?s_2$) or ($?x$, $?p_2$, $?s_2$, $?s_1$, $?p_1$)
{ $?s_1$ $?p_1$ $?x$. $?s_2$ $?p_2$ $?x$ }	OSP	OSP	($?x$, $?s_1$, $?p_1$, $?s_2$, $?p_2$) or ($?x$, $?s_2$, $?p_2$, $?s_1$, $?p_1$)

A.2.2 1 constraint

- Constraint on subject { f $?p_1$ $?o_1$. $?s_2$ $?p_2$ $?o_2$ }

BGP	View 1	View 2	Result order
{ f $?p_1$ $?x$. $?x$ $?p_2$ $?o_2$ }	SOP	SPO	($?x$, $?p_1$, $?p_2$, $?o_2$) or ($?x$, $?p_2$, $?o_2$, $?p_1$)
{ f $?p_1$ $?x$. $?x$ $?p_2$ $?o_2$ }	SOP	SOP	($?x$, $?p_1$, $?o_2$, $?p_2$) or ($?x$, $?o_2$, $?p_2$, $?p_1$)

BGP	View 1	View 2	Result order
{ f $?p_1$ $?x$. $?s_2$ $?p_2$ $?x$ }	SOP	OPS	($?x$, $?p_1$, $?p_2$, $?s_2$) or ($?x$, $?p_2$, $?s_2$, $?p_1$)
{ f $?p_1$ $?x$. $?s_2$ $?p_2$ $?x$ }	SOP	OSP	($?x$, $?p_1$, $?s_2$, $?p_2$) or ($?x$, $?s_2$, $?p_2$, $?p_1$)

- Constraint on property { $?s_1$ f $?o_1$. $?s_2$ $?p_2$ $?o_2$ }

BGP	View 1	View 2	Result order
{ $?x$ f $?o_1$. $?x$ $?p_2$ $?o_2$ }	PSO	SPO	($?x$, $?o_1$, $?p_2$, $?o_2$) or ($?x$, $?p_2$, $?o_2$, $?o_1$)
{ $?x$ f $?o_1$. $?x$ $?p_2$ $?o_2$ }	PSO	SOP	($?x$, $?o_1$, $?o_2$, $?p_2$) or ($?x$, $?o_2$, $?p_2$, $?o_1$)
{ $?x$ f $?o_1$. $?s_2$ $?p_2$ $?x$ }	PSO	OSP	($?x$, $?o_1$, $?s_2$, $?p_2$) or ($?x$, $?s_2$, $?p_2$, $?o_1$)
{ $?x$ f $?o_1$. $?s_2$ $?p_2$ $?x$ }	PSO	OPS	($?x$, $?o_1$, $?p_2$, $?s_2$) or ($?x$, $?p_2$, $?s_2$, $?o_1$)
{ $?s_1$ f $?x$. $?x$ $?p_2$ $?o_2$ }	POS	SPO	($?x$, $?s_1$, $?p_2$, $?o_2$) or ($?x$, $?p_2$, $?o_2$, $?s_1$)
{ $?s_1$ f $?x$. $?x$ $?p_2$ $?o_2$ }	POS	SOP	($?x$, $?s_1$, $?o_2$, $?p_2$) or ($?x$, $?o_2$, $?p_2$, $?s_1$)
{ $?s_1$ f $?x$. $?s_2$ $?p_2$ $?x$ }	POS	OSP	($?x$, $?s_1$, $?s_2$, $?p_2$) or ($?x$, $?s_2$, $?p_2$, $?s_1$)
{ $?s_1$ f $?x$. $?s_2$ $?p_2$ $?x$ }	POS	OPS	($?x$, $?s_1$, $?p_2$, $?s_2$) or ($?x$, $?p_2$, $?s_2$, $?s_1$)

- Constraint on object { $?s_1$ $?p_1$ f . $?s_2$ $?p_2$ $?o_2$ }

BGP	View 1	View 2	Result order
{ ?x ?p_1 f . ?x ?p_2 ?o_2 }	OSP	SPO	(?x, ?p_1, ?p_2, ?o_2) or (?x, ?p_2, ?o_2, ?p_1)
{ ?x ?p_1 f . ?x ?p_2 ?o_2 }	OSP	SOP	(?x, ?p_1, ?o_2, ?p_2) or (?x, ?o_2, ?p_2, ?p_1)
{ ?x ?p_1 f . ?s_2 ?p_2 ?x }	OSP	OSP	(?x, ?p_1, ?s_2, ?p_2) or (?x, ?s_2, ?p_2, ?p_1)
{ ?x ?p_1 f . ?s_2 ?p_2 ?x }	OSP	OPS	(?x, ?p_1, ?p_2, ?s_2) or (?x, ?p_2, ?s_2, ?p_1)

A.2.3 2 constraints

- { f_a f_b ?o_1 .
?s_2 ?p_2 ?o_2 }

BGP	View 1	View 2	Result order
{ f_a f_b ?x . ?x ?p_2 ?o_2 }	SPO or PSO	SPO	(?x, ?p_2, ?o_2)
{ f_a f_b ?x . ?x ?p_2 ?o_2 }	SPO or PSO	SOP	(?x, ?o_2, ?p_2)
{ f_a f_b ?x . ?s_1 ?p_2 ?x }	SPO or PSO	OSP	(?x, ?s_2, ?p_2)
{ f_a f_b ?x . ?s_1 ?p_2 ?x }	SPO or PSO	OPS	(?x, ?p_2, ?s_2)

- { f_a ?p_1 ?o_1 .
f_b ?p_2 ?o_2 }

BGP	View 1	View 2	Result order
{ f_a ?p_1 ?x . f_b ?p_2 ?x }	SOP	SOP	(?x, ?p_1, ?p_2) or (?x, ?p_2, ?p_1)

- { f_a ?p_1 ?o_1 .
?s_2 f_b ?o_2 }

BGP	View 1	View 2	Result order
{ f_a ?p_1 ?x . ?x f_b ?o_2 }	SOP	PSO	(?x, ?p_1, ?o_2) or (?x, ?o_2, ?p_1)
{ f_a ?p_1 ?x . ?s_2 f_b ?x }	SOP	POS	(?x, ?p_1, ?s_2) or (?x, ?s_2, ?p_1)

- { f_a ?p_1 ?o_1 .
?s_2 ?p_2 f_b }

BGP	View 1	View 2	Result order
{ f_a ?p_1 ?x . ?x ?p_2 f_b }	SOP	OSP	(?x, ?p_1, ?p_2) or (?x, ?p_2, ?p_1)

- { ?s_1 f_a f_b .
?s_2 ?p_2 ?o_2 }

BGP	View 1	View 2	Result order
{ ?x f_a f_b . ?x ?p_2 ?o_2 }	POS or OPS	SPO	(?x, ?p_2, ?o_2)
{ ?x f_a f_b . ?x ?p_2 ?o_2 }	POS or OPS	SOP	(?x, ?o_2, ?p_2)
{ ?x f_a f_b . ?s_2 ?p_2 ?x }	POS or OPS	OSP	(?x, ?s_2, ?p_2)
{ ?x f_a f_b . ?s_2 ?p_2 ?x }	POS or OPS	OPS	(?x, ?p_2, ?s_2)

- { ?s_1 f_a ?o_1 .
?s_2 f_b ?o_2 }

BGP	View 1	View 2	Result order
{ ?x f_a ?o_1 . ?x f_b ?o_2 }	PSO	PSO	(?x, ?o_1, ?o_2) or (?x, ?o_2, ?o_1)
{ ?x f_a ?o_1 . ?s_2 f_b ?x }	PSO	POS	(?x, ?o_1, ?s_2) or (?x, ?s_2, ?o_1)
{ ?s_1 f_a ?x . ?x f_b ?o_2 }	POS	PSO	(?x, ?s_1, ?o_2) or (?x, ?o_2, ?s_1)
{ ?s_1 f_a ?x . ?s_2 f_b ?x }	POS	POS	(?x, ?s_1, ?s_2) or (?x, ?s_2, ?s_1)

- { ?s_1 f_a ?o_1 .
  ?s_2 ?p_2 f_b }

BGP	View 1	View 2	Result order
{ ?x f_a ?o_1 . ?x ?p_2 f_b }	PSO	OSP	(?x, ?o_1, ?p_2) or (?x, ?p_2, ?o_1)
{ ?s_1 f_a ?x . ?x ?p_2 f_b }	POS	OSP	(?x, ?s_1, ?p_2) or (?x, ?p_2, ?s_1)

- { ?s_1 ?p_1 f_a .
  ?s_2 ?p_2 f_b }

BGP	View 1	View 2	Result order
{ ?x ?p_1 f_a . ?x ?p_2 f_b }	OSP	OSP	(?x, ?p_1, ?p_2) or (?x, ?p_2, ?p_1)

A.2.4 3 constraints

- { f_a f_b ?o_1 .
  f_c ?p_2 ?o_2 }

BGP	View 1	View 2	Result order
{ f_a f_b ?x . f_c ?p_2 ?x }	SPO or PSO	SOP	(?x, ?p_2)

- { f_a f_b ?o_1 .
  ?s_2 f_c ?o_2 }

BGP	View 1	View 2	Result order
{ f_a f_b ?x . ?x f_c ?o_2 }	SPO or PSO	PSO	(?x, ?o_2)
{ f_a f_b ?x . ?s_2 f_c ?x }	SPO or PSO	POS	(?x, ?s_2)

- { f_a f_b ?o_1 .
  ?s_2 ?p_2 f_c }

BGP	View 1	View 2	Result order
{ f_a f_b ?x . ?x ?p_2 f_c }	SPO or PSO	OSP	(?x, ?p_2)

- { f_a ?p_1 ?o_1 .
  ?s_2 f_b f_c }

BGP	View 1	View 2	Result order
{ f_a ?p_1 ?x . ?x f_b f_c }	SOP	POS or OPS	(?x, ?p_1)

- { ?s_1 f_a f_b .
  ?s_2 f_c ?o_2 }

BGP	View 1	View 2	Result order
{ ?x f_a f_b . ?x f_c ?o_2 }	POS or OPS	PSO	(?x, ?o_2)
{ ?x f_a f_b . ?s_2 f_c ?x }	POS or OPS	POS	(?x, ?s_2)

- { ?s_1 f_a f_b .
  ?s_2 ?p_2 f_c }

BGP	View 1	View 2	Result order
{ ?x f_a f_b . ?x ?p_2 f_c }	POS or OPS	OSP	(?x, ?p_2)

A.2.5 4 constraints

BGP	View 1	View 2	Result order
{ ?x f_a f_b . ?x f_c f_d }	POS or OPS	POS or OPS	(?x)
{ ?x f_a f_b . f_c f_d ?x }	POS or OPS	SPO or PSO	(?x)
{ f_a f_b ?x . ?x f_c f_d }	SPO or PSO	POS or OPS	(?x)
{ f_a f_b ?x . f_c f_d ?x }	SPO or PSO	SPO or PSO	(?x)

A.3 More complex BGP examples

Since the space of the possible combinations with three (or even more) triple patterns is too wide to be treated extensively, only a few examples will be shown; the MonetDB/SPARQL optimizer, however, can generate such combinations automatically during query execution as well as it can pre-calculate some common ones.

The information of the preceding section will be used as basis to find the most convenient access to the data tables.

A.3.1 Query 1

```
select ?title
where {  ?x dc:title   ?title .
         ?x dc:author  ?y .
         ?y dc:name    "Herman"
         ?y dc:surname "Melville" }
```

This query asks for the book titles whose author is called “Herman Melville”; generalizing its sense, it asks for the object of a resource for which are known some nested property values.

Since the given object values are the most selective conditions in the query, it is better to start evaluating it from those triple patterns that contain such conditions.

Figure A.1 shows the best plan for this query. The *Id* function, already used in section 5.3, returns the `oid` of the its RDF-term parameter; on each edge is shown how a result is sorted.

The two selections are performed first on the most selective condition, thus on *o*, and then on *p*; the OPS view is therefore chosen, so that both selection are run against sorted data, as well as the following join of the results on ?*y*; the merge algorithm can therefore be used.

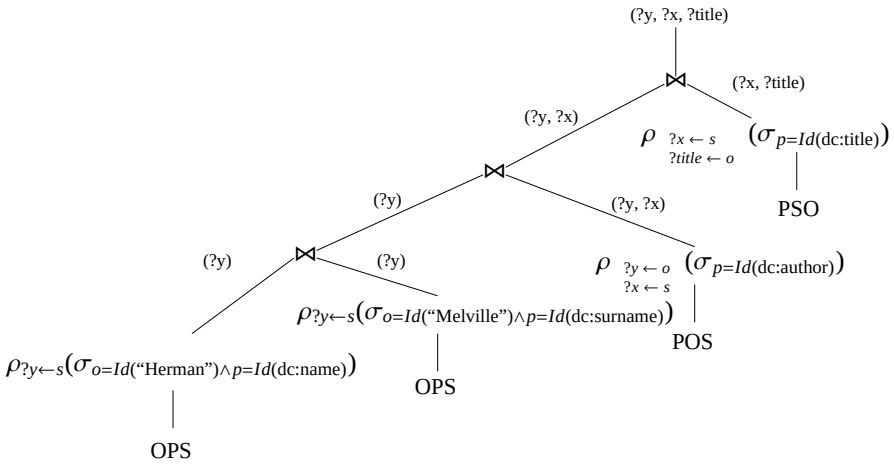

Figure A.1:

As any selection on the data table, also the one POS is performed on ordered data. The result is sorted on ?*y* making another merge join possible, but since the cardinality of the left operand is expected to be very low, instead of scanning the full table as a merge join would normally do, it would be better to perform a binary search on the right operand for each value in the left one.

The same happens in the last join, where the left operand is again small and the search in the right operand is performed on the sorted column ?*x*.

A.3.2 Query 2

```
select ?title ?name ?mbox
where {  ?person foaf:publication ?doc   .
         ?doc    dc:title         ?title .
         ?person foaf:mbox        ?mbox  .
         ?person foaf:name        ?name  }
```

This query lists all the names and emails of those people who published one or more documents, along with the titles of these.

This query can have huge intermediate results if the dataset is large: the join order chosen in figure A.2 keeps them as small as possible, as well as choosing the views in order to join sorted data.

If the first join combined the result of the selection on `p=foaf:publication` and `p=foaf:mbox` which are both multi-valued properties, the intermediate result would be much bigger than in the case shown in figure, since `foaf:name` is a single-valued property.

Also in this query the first two joins can be merge-joins, while the last has only its right operand ordered.

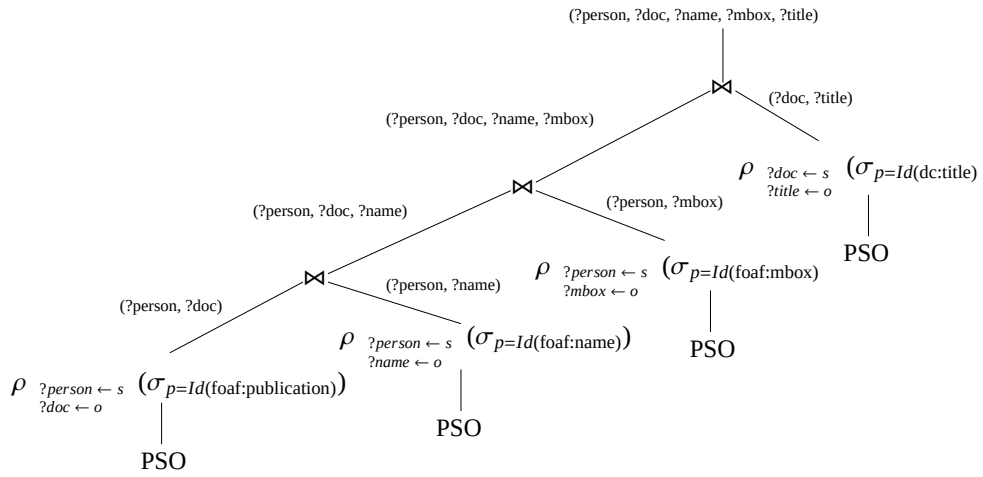

Figure A.2:

A.3.3 Query 3

```
prefix m: <http://motorbikeontology.org/terms/>
select ?name ?price
where {  ?bike   m:modelname ?name .
         ?bike   m:engine    ?engine .
         ?engine m:cylinders 3 .
         ?bike   dc:price    ?price
         FILTER(?price < 7000) }
```

The query searches for those motorbikes that have a three-cylinder engine and a price lower than 7000 Euros.

Two different plans are shown: the first (fig. A.3) selects the bikes with the requested price and joins the result first with the bikes with three cylinders and then retrieves their names; the second (fig. A.4) pulls the selection on price up, looking first for the bikes with the requested engine, then retrieving their names and prices and finally selecting the right price from the result.

The second approach has the advantage to perform only merge joins, except the first one (the leftmost in figure A.4), but the final select on the price has to scan the full result. The first can perform the selection on price with an almost free of cost slice view 2.4, but it cannot perform merge joins; it can anyway count on the fact that at least one operand has always the join column sorted.

The selections on price take also the POS_NUM_MIN_ID argument, that represents the `oid` of the smallest positive numeric in MonetDB's dictionary.

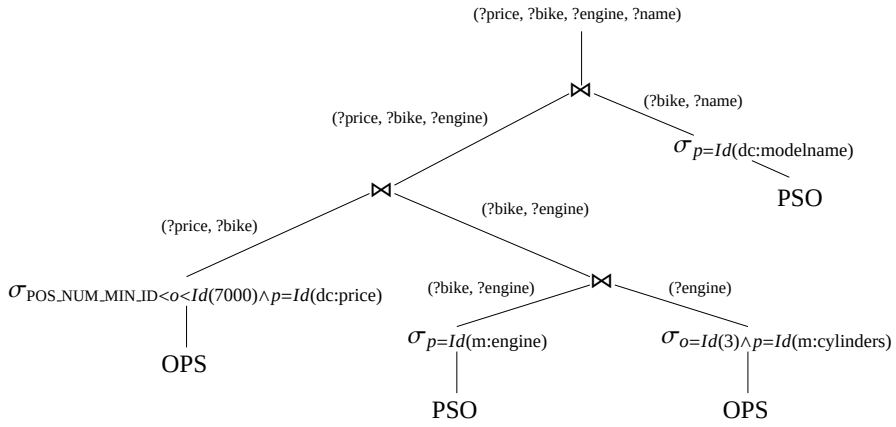

Figure A.3:

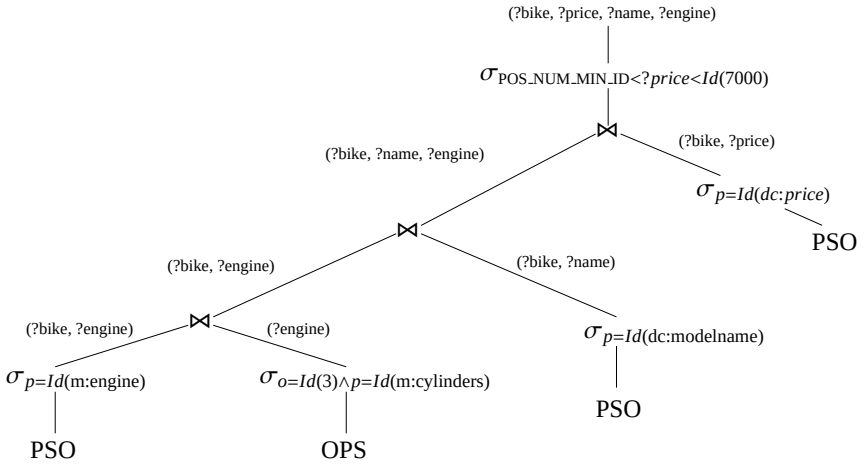

Figure A.4:

Bibliography

[1] Centrum voor Wiskunde en Informatica . http://www.cwi.nl/.

[2] MonetDB - Query Processing at Light Speed. http://monetdb.cwi.nl/.

[3] OpenLink Virtuoso - RDF Database and SPARQL. http://docs.openlinksw.com/virtuoso/rdfdatarepresentation.html.

[4] RDF Issue Tracking. http://www.w3.org/2000/03/rdf-tracking/#rdfms-literalsubjects.

[5] Resource Description Framework (RDF) / W3C Semantic Web Activity. http://www.w3.org/RDF/.

[6] SPARQL specification inconsistency. http://lists.w3.org/Archives/Public/public-sparql-dev/2008JulSep/0010.html.

[7] The research articles related to MonetDB. http://monetdb.cwi.nl/projects/monetdb/Development/Research/Articles/index.html.

[8] TPC-H Benchmark Comparison. http://monetdb.cwi.nl/SQL/Benchmark/TPCH/index.html.

[9] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33th Very Large Data Bases (VLDB) Conference*, Vienna, Austria, 2007.

[10] Sofia Alexaki, Vassilis Christophides, Greg Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web*, Hong Kong, China, 2001.

[11] H. Alvestrand. RFC 3066 - Tags for the Identification of Languages. http://www.ietf.org/rfc/rfc3066.txt, January 2001.

[12] Dave Beckett. Raptor RDF Parser Library. http://librdf.org/raptor/.

[13] Dave Beckett. RDF/XML Syntax Specification (Revised). http://www.w3.org/TR/rdf-syntax-grammar/. Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

[14] David Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language. http://www.w3.org/TeamSubmission/turtle/, January 2008.

[15] Tim Berners-Lee. Notation3 (N3) A readable RDF syntax. http://www.w3.org/DesignIssues/Notation3.html.

[16] Paul V. Biron, Kaiser Permanente, and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition - Built-in datatypes. http://www.w3.org/TR/xmlschema-2/. Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

[17] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999.

[18] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 225–237, Asilomar, CA, USA, January 2005.

[19] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/. Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

[20] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First Internation Semantic Web Conference (ISWC)*, Sardinia, Italy, 2002.

[21] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st International Conference on Very large data bases (VLDB)*, pages 1216–1227. VLDB Endowment, 2005.

[22] Burleson Consulting. Oracle tool tips - Relational Division. http://www.dba-oracle.com/t_sql_patterns_relational_division.htm.

[23] Gordon V. Cormack. Data compression on a database system. *Commun. ACM*, 28(12):1336–1342, 1985.

[24] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP-Labs, 2005.

[25] DBpedia. http://dbpedia.org/.

[26] Orri Erling. Advances in Virtuoso RDF Triple Storage (Bitmap Indexing). http://virtuoso.openlinksw.com/wiki/main/Main/VOSBitmapIndexing.

[27] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticleRDF.

[28] Goetz Graefe and Leonard D. Shapiro. Data Compression and Database Performance. In *In Proc. ACM/IEEE-CS Symp. On Applied Computing*, pages 22–27, 1991.

[29] Jan Grant and David Beckett. N-Triples. http://www.w3.org/TR/rdf-testcases/#ntriples, February 2004.

[30] Stephen Harris. SPARQL query processing with conventional relational database systems. In *WISE Workshops*, pages 235–244, 2005.

[31] Patrick Hayes. RDF Semantics. http://www.w3.org/TR/rdf-mt/. Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

[32] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD '07: Proceedings of the 2007 ACM*

SIGMOD international conference on Management of data, pages 389–400, New York, NY, USA, 2007. ACM.

[33] Jena Semantic Web Framework. `http://jena.sourceforge.net/`.

[34] KAON The KArlsruhe ONtology and Semantic Web tool suite. `http://kaon.semanticweb.org/`.

[35] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In *Proceedings of the 11th International Conference on World Wide Web*, pages 592–603, Honolulu, Hawaii, USA, 2002.

[36] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000.

[37] Stefan Manegold. The Calibrator (v0.9e), a Cache-Memory and TLB Calibration Tool. `http://monetdb.cwi.nl/Calibrator/`.

[38] Frank Manola and Eric Miller. RDF Primer. `http://www.w3.org/TR/rdf-primer/`. Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

[39] Andrew Newman. Querying the Semantic Web using a Relational Based SPARQL. Master's thesis, The University of Queensland, 2006.

[40] Web Ontology Language OWL / W3C Semantic Web Activity. `http://www.w3.org/2004/OWL/`.

[41] Jorge Peréz and Marcel Arenas Claudio Gutierrez. Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference*, Athens, GA, USA, November 2006.

[42] Jorge Peréz and Marcel Arenas Claudio Gutierrez. Semantics of SPARQL. Technical report, Universidad de Chile, October 2006.

[43] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`. Copyright © 2006-2007 World Wide Web Consortium, (Massachusetts

Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

[44] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, January 2008. Copyright © 2006-2007 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

[45] Andy Seaborne. RDQL - A Query Language for RDF. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[46] Sesame: RDF Schema Querying and Storage. http://www.openrdf.org/.

[47] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column Oriented DBMS. In *Proceedings of the Very Large Data Bases (VLDB) Conference*, Trondheim, Norway, 2005.

[48] SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes. http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/.

[49] Virtuoso Universal Server. http://www.openlinksw.com/virtuoso/.

[50] Raphael Volz, Daniel Oberle, Steffen Staab, and Boris Motik. KAON SERVER - a Semantic Web Management System. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungary, 2003.

[51] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proceedings of the 1st International Workshop on Semantic Web and Databases*, Berlin, Germany, 2003.

[52] M. Zukowski. Improving I/O Bandwidth for Data-Intensive Applications. In *Proceedings of the British National Conference on Databases (BNCOD)*, Sunderland, England, UK, July 2005. PhD Workshop.

[53] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, June 2005.

[54] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. Technical Report INS-E0511, CWI, Amsterdam, The Netherlands, July 2005.