WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

Master Thesis

HDT Quads: Compressed Triple Store for Linked Data

Julian Reindorf

Date of Birth: 24.08.1992
Student ID: 1151548

Subject Area: Information Business

Studienkennzahl: J066925

Supervisor: Univ.Prof. Dr.techn. Axel Polleres

Co-Supervisor: Dr. Javier David Fernández García

Date of Submission: 28.09.2017

Department of Information Systems and Operations, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR INFORMATIONS-
VERARBEITUNG UND PROZESS-
MANAGEMENT DEPARTMENT
OF INFORMATION SYSTEMS AND
OPERATIONS

Contents

List of Figures

List of Tables

Abstract

The Linked Data philosophy extends the Web through standards promoting semi-structured data publication, exchange and consumption. In this regard, the predominant standard for modeling data on the Web is RDF, which is a graph-based model to represent relationships between resources. However, with increasing size, compressibility and queryability of RDF graphs become an issue. These challenges are addressed by the HDT (Header-Dictionary-Triples) format, which is a well-known compressed representation of RDF datasets that supports retrieval features without prior decompression. Yet, data often contains additional graph information, such as the origin or version of a triple. Traditional HDT is not capable of handling this additional parameter. This work introduces HDTQ (HDT Quads), an extension of HDT, which is able to represent additional graph information while still being highly compact and queryable. Two approaches of this extension, Annotated Triples and Annotated Graphs, are introduced and their performance is compared to the leading open-source RDF stores on the market, Apache Jena and Virtuoso. Results show that HDTQ is a competitive alternative to these two established systems.

1 Introduction

In the last decades, the Web evolved from a space of linked documents to a space where documents and data are both linked. This fostered the formation of a set of best practices for publishing and connecting structured data on the Web, commonly referred to as Linked Data. The spread of Linked Data enabled the extension of the Web connecting data from various domains such as films, music, people, companies, genes, statistical data and online communities [26]. In recent years the use of the Resource Description Framework (RDF) [1] on the Web has also experienced an impressive growth. To describe facts about arbitrary fields of knowledge in the Web, RDF became the de facto standard [31].

RDF does not impose any physical storage solution, it is only a logical data model. For this reason, RDF stores are either based on existing (relational or not only SQL (noSQL) database management systems) or are designed from scratch [68]. RDF graphs, consisting of a set of triples, each having a subject, a predicate and an object, can reach millions or billions of nodes. As an example, the Linked Open Data (LOD) cloud contains datasets with more than 55 billion triples (the openlink-lod-cache), 20 billion triples (the uniref graph) and about 1.5 billion triples (the DBpedia graph) [42]. Even though the transmission speed increases and storage capacity grows, such graphs can quickly become cumbersome to share with others. This is of specific interest when it comes to mobile devices where memory constraints and transmission costs still play a non negligible role [92].

Therefore, it is important to compress RDF data. Compression becomes particularly interesting if one does not even need to decompress data before querying it. One representation format for such RDF graphs, which achieves large spatial savings is Header-Dictionary-Triples (HDT). It is based on the three main components Header (includes metadata), Dictionary (organizes identifiers) and Triples (contains triple data) [46].

In its most basic form, all triples in an RDF dataset belong to the same graph, the default graph. However, it is also possible that triples in a dataset belong the different graphs, so-called named graphs. With the basic subject - predicate - object structure, it is not possible to store this additional graph information. For this reason, triples are enriched with a fourth component, namely the graph. Such extended triples consisting of a subject, a predicate, an object and a graph are called RDF quadruples, or RDF quads [3].

By storing information about the origin of the RDF data in this fourth component, one can combine data from different sources in a single RDF dataset. The combined dataset can be queried for getting the context of a triple, for example, provenance information. One particular application of

quads is RDF versioning [48]. In the same manner in which databases or Web pages are not static, RDF data changes, too. Thus, instead of storing information about the origin in the fourth part of the quads, one could also store information about the version of a triple.

Several RDF storage strategies exist to handle such version information. The first one is called independent copies (IC) where for each version an entire graph is stored. The second one is called changed-based (CB) approach, in which only the first version of the graph is stored completely, while the others are stored as differences to their previous version. When using the third strategy, the timestamp-based (TB) approach, triples are enriched with additional information which determines the versions in which the triple is valid. Lastly, there are hybrid approaches which are a combination of the aforementioned techniques [48].

The IC and the TB approach are also suitable to be applied on RDF data in which the fourth component is the origin and not a version. Of these two, the TB approach is of particular interest for this thesis. Section 3.2 deals with this matter in detail.

The HDT format was not constructed to handle quads. Yet, for the aforementioned reasons, it is very favorable to work with such extended triples. In the course of this paper, the HDT format is extended, so it can cope with quads and is still compact and queryable.

In a nutshell, this paper covers:

- Theoretical background about the Semantic Web and RDF in Section 2.
- A study on existing RDF triple stores (with a focus on Jena and Virtuoso) and more details on RDF versioning approaches in Section 3.
- Background information on HDT, including the header, dictionary and triples parts in Section 4.
- The proposed extension of HDT to handle RDF quads in Section 5.
- An overview of the actual implementation of this extension in Java is given in Section 6.
- An evaluation of the extended HDT format in Section 7.
- Summary, limitations and thoughts on future research in Section 8.

Furthermore, a running example for an RDF graph will be introduced in Section 2.2.1 and will be used for better comprehension in subsequent sections of this work.

1.1 Research Question

The goal of this work is to extend HDT with the functionality to handle quads. The resulting format should still be compressed and the generation still be fast. Moreover, it should be queryable without the need to decompress it first. After the HDT representation is extended to cater for quads, a performance evaluation in comparison to Apache Jena [98] and Virtuoso [41], regarding space requirements, creation time and querying speed for different datasets is conducted.

The research question of this thesis is: "How can compressed RDF formats like HDT be extended to handle quad information and keep its compact and queryable features?".

Although our proposal must be general enough to cope with general quad information, we particularly focus our decisions in supporting quads where the graph information denotes the version of each triple.

1.2 Research Method

We applied a Design Science Research (DSR) methodology [115] as a research method to investigate and solve well-identified problems in IT. First, a literature research is performed to get an overview of other available systems and to understand HDT in detail. Then, the existing Java implementation of HDT is extended, such that it can handle graph information data (i.e. it can handle quads). The focus should be on performance and good compression. Still, the resulting code should be easily understandable, maintainable and extensible.

After that, the performance of the system will be tested in a server environment. The performance (space requirement, creation time and querying speed) will be compared to other systems handling quads. Results will be reported in this thesis and should be the basis for further development.

2 Theoretical Background

This section introduces important concepts used throughout this work. In particular, the Semantic Web, RDF and SPARQL Protocol and RDF Query Language (SPARQL) are introduced and linked data applications are discussed.

2.1 Semantic Web

While the hypertext Web is a technology for sharing documents, the Semantic Web [73] is for sharing data [23]. It is a highly interconnected network of data which can easily be accessed and understood by any desktop or handheld machine. Originally, Tim Berners-Lee, James Hendler and Ora Lassila had the vision of software agents which use the World Wide Web to update medical records, book flights and hotels for trips and retrieve customized answers to particular questions without searching for information or poring through results [44].

In order for this visions to become true, they presented the following technologies:

- A common language: this is needed to represent data that can be understood by all kinds of software agents.
- Ontologies: sets of statements that translate information from disparate databases into common terms.
- Rules: these allow software agents to reason about the information described in those terms.

Together, these technologies would analyze the raw data stored in online databases and all the data about text, images, video and communications on the Web.

The Semantic Web can only be a success if there are well established standards to express shared meanings. Such common conceptualizations have been developed in multiple domains and are commonly referred to as ontologies [126].

Broadly speaking, the Semantic Web is an enhancement to the World Wide Web which gives it far greater utility. When people agree on a common scheme to represent information they care about, the Semantic Web comes to life [44].

Besides the need of common schemes to represent information, a common approach for publishing and connecting structured data on the Web is also needed. For this, Tim Berners-Lee published the Linked Data principles [20], which provide guidelines on how to use standardized Web technologies to link between data from different sources on a data-level. By linking such data, a global data space is formed, similar to the classic Web. This Web is called the Web of Linked Data. The idea behind Linked Data is to use HTTP Uniform Resource Identifiers (URIs) to identify Web documents as well as arbitrary real world entities. The RDF is used to represent data about these entities. The URIs can then be resolved by Web clients and an RDF description of the

identified entity is provided by a Web server. These descriptions can even contain links to entities in other data sources [25].

A practical and widely used vocabulary to annotate web resources is schema.org. Developed by an open community process, it is used in over 12 million sites to markup web pages and email messages [63].

Not only companies, but also consumers are beginning to use the data language and ontologies directly. One example is the Friend of a Friend (FOAF) [28] project, a decentralized social-networking system. There is Semantic Web vocabulary for describing people, their name, age, location, job and their relationship to other people. While MySpace's and Facebook's fields are incompatible and not open to translation, FOAF users can post information and images in any format they like and still connect it all. More than a million users already interlinked their FOAF profiles [44].

2.2 RDF

Information on the Web can be represented with the Resource Description Framework (RDF). The abstract syntax of RDF uses triples as its core structure. Each triple consists of exactly one subject, one predicate and one object. An RDF graph is a set of such triples. Such a graph can be visualized as a directed graph, where the subject and object are nodes and the predicate is the edge between them [2].

An RDF graph can consist of three kinds of nodes:

- **IRIs**: Internationalized Resource Identifiers (IRIs) are similar to URIs [21]. However, the characters they can contain additionally include characters from the Universal Coded Character Set (UCS) [40].
- **Literals**: The range of possible values for a literal is restricted by its datatype, which can be, but is not limited to strings, numbers and dates.
- **Blank nodes**: No specific resources are identified by Blank nodes. Blank nodes act as local existential variables in a dataset.

Items denoted by IRIs and literals are called resources or entities. These resources can be anything from physical objects, documents and abstract concepts to numbers and strings. While resources denoted by an IRI are called *referent*, resources denoted by a literal are called *literal value*. RDF graphs are static, they are snapshots of information (atemporal) [2].

As noted above, RDF graphs are sets of RDF triples. An RDF triple always consists of three components:

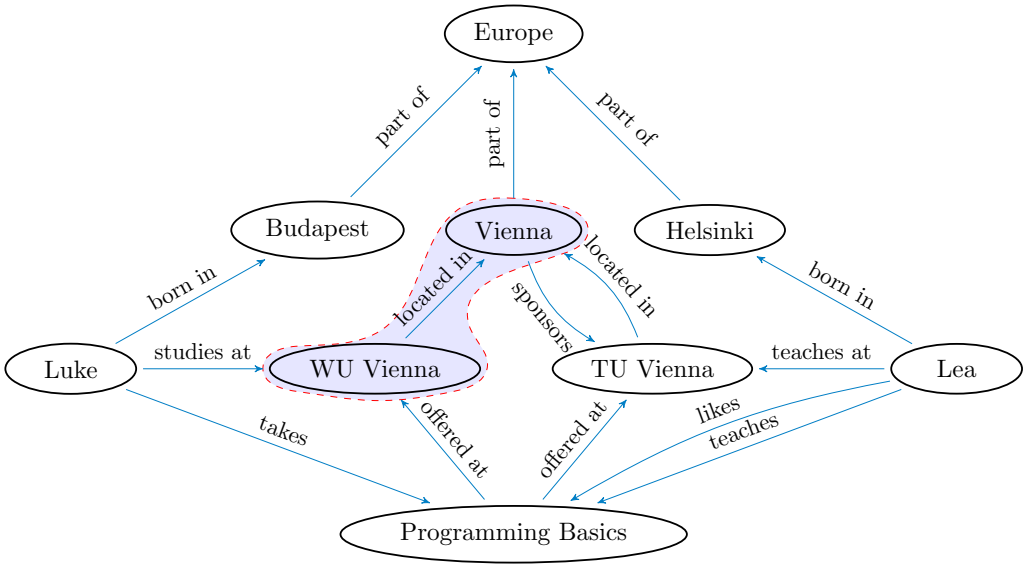


Figure 1: Graph_A. An RDF graph in graphical representation.

- **Subject**: The subject can either be an IRI or a blank node.
- **Predicate**: The predicate is always an IRI.
- **Object**: The object can either be an IRI, a literal or a blank node.

The order in which the triple's components are written is by convention subject, predicate, object [2].

For denoting RDF graphs, several concrete syntaxes exist, for example Notation3 (N3) [22], N-Triples [19], N-Quads [29], JSON-LD [131], RDF/XML [17] and Turtle [18]. Some of these syntaxes will be discussed in further detail in Section 2.2.2.

2.2.1 Running Example

An example for an RDF graph can be seen in Figure 1. The figure shows a graphical representation of an RDF graph containing information about two universities, including their location and participants (a teacher and a student) as well as some information about the participants. The highlighted area marks one basic triple. Its subject is "WU Vienna", its predicate is "located in" and its object is "Vienna".

The exemplary RDF graph, which will be referred to as Graph_A, consists altogether of 15 triples. As can be seen in the figure, the object of one triple can also be the subject of another triple. This is the case for the "Vienna"

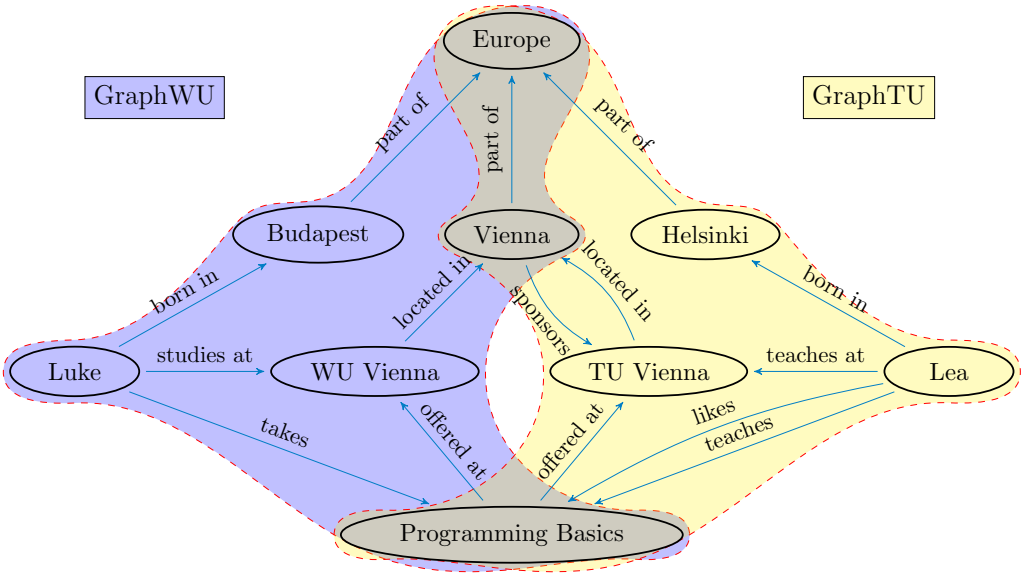


Figure 2: Graph_B. An RDF graph consisting of two graphs.

node. It is the object of the highlighted triple, and the subject of the "Vienna - part of - Europe" triple. Furthermore, nodes can be the subject of multiple triples as well as the object of multiple triples. What is more, predicates can be used as predicates in multiple triples (like the "born in" predicate in the example).

The graph is directed, which is a result of the subject - predicate - object structure of triples of RDF graphs. It is multi edged ("Lea" both, teaches and likes "Programming Basics") and contains a cycle ("TU Vienna" is located in "Vienna" and "Vienna" sponsors "TU Vienna").

For legibility reasons, the graph only contains literals in the graphical representation. All subjects, predicates and objects in this graph are in fact IRIs, their exact values can be seen in further examples in the subsequent sections.

Another graph, from now on referred to as Graph_B, can be seen in Figure 2. Basically, the graph is the same as Graph_A. However, this graph consists of two subgraphs coming from different sources, where the blue background marks the triples from the WU database and the yellow background marks the triples from the TU database. The overlapping area contains subjects / objects that are contained in both graphs ("Europe", "Vienna" and "Programming Basics").

Graph_A and Graph_B will be used as running examples throughout this work.

2.2.2 Notations

Traditionally, an RDF dataset consisted of a single graph. This view was soon superseded to consider datasets conformed by multiple graphs. That is a mechanism to group RDF statements into multiple graphs (as shown in Figure 2) and associate each of the graphs with an IRI. The extension is a result of the need for a mechanism to talk about subsets of a collection of triples. First, multiple graphs were introduced in the RDF query language SPARQL. Consequently, the RDF data model was extended with a notation for multiple graphs, closely aligned with SPARQL [1].

There are different serialization formats for writing down RDF graphs. All of them lead to the very same triples and are therefore logically equivalent. However, not all of the existing formats are capable of multiple graphs [1].

There is a group of closely related RDF languages. It is called the "Turtle family of RDF languages". Members of the group are N-Triples, Turtle, N-Quads and TriG. The former two do not support multiple graphs, while the latter two, which are extensions of the former ones, do support multiple graphs [1].

N-Triples is a very simple line-based, plain-text format. A triple is represented by a line, IRIs are enclosed in angle brackets. Each line is delimited by a period, which also marks the end of a triple. Literals are followed by a delimiter and its datatype, where the string datatype can be omitted [1].

N-Quads is very similar to N-Triples. However it allows to add a fourth element to a line. The fourth element (the graph) specifies in which graph the triple of this very line is contained. The graph itself is also denoted as an IRI [29].

The third member of the Turtle family is called Turtle. Turtle extends the N-Triples syntax with a number of syntactic shortcuts. This includes support for namespace prefixes, lists and shorthands for datatyped literals. In doing so, Turtle is more compact than N-Triples and provides a trade-off between ease of writing, ease of parsing and readability [1].

While Turtle only supports the specification of a single graph, its extension TriG supports the specification of multiple graphs in the form of an RDF dataset. Note that Turtle and TriG could be seen as one language, as in RDF 1.1 (which is the most recent version at the time of writing) any legal Turtle document is a legal TriG document as well [1]. Graph_B can be seen in TriG syntax in Figure 3. Note that the two listings shown are actually one physical file. As can be seen, the names of the graphs in the graphset are "http://-graph.org/graphWU" and "http://graph.org/graphTU". The triples of the respectively named graph are placed in between matching curly braces.

Other RDF notations exist that support naming of graphs, like TriX [30],

```
BASE  <http://example.org/>
PREFIX rel: <http://relation.org/>

GRAPH <http://graph.org/graphWU>
{
 <entity#Luke>
   rel:takes <course#ProgrammingBasics> ;
   rel:studiesAt <entity#WUVienna> ;
   rel:bornIn <location#Budapest> .
 <course#ProgrammingBasics>
   rel:offeredAt <entity#WUVienna> .
 <entity#WUVienna>
   rel:locatedIn <location#Vienna> .
 <location#Budapest>
   rel:partOf <location#Europe> .
 <location#Vienna>
   rel:partOf <location#Europe> .
}
```

```
GRAPH <http://graph.org/graphTU>
{
 <entity#Lea>
   rel:bornIn <location#Helsinki> ;
   rel:teaches <course#ProgrammingBasics> ;
   rel:teachesAt <entity#TUVienna> ;
   rel:likes <course#ProgrammingBasics> .
 <course#ProgrammingBasics>
   rel:offeredAt <entity#TUVienna> .
 <entity#TUVienna>
   rel:locatedIn <location#Vienna> .
 <location#Helsinki>
   rel:partOf <location#Europe> .
 <location#Vienna>
   rel:partOf <location#Europe> ;
   rel:sponsors <entity#TUVienna> .
 <entity#WUVienna>
   rel:locatedIn <location#Vienna> .
}
```

Figure 3: Graph_B in TriG syntax.

which stores RDF triples in XML.

Another highly compact RDF representation format is HDT, which is in the focus of this work, and will be described in detail in Section 4.

2.3 SPARQL

There are a number of query languages for RDF data, like RQL, RDQL and SeRQL which where competing to become the W3C standard [66]. Another prominent example is SquishQL [102]. In the end, SPARQL [116] arose as the winner [76], becoming a W3C recommendation in 2008 [118]. In 2013, the latest release at the time of writing, SPARQL 1.1 was published as a W3C recommendation [62].

SPARQL 1.1 covers a set of specifications that provides protocols and languages to query and manipulate RDF graph content. As such, not only queries against an RDF store are covered, but also different result formats, federated queries, update languages, service descriptions and test cases [62]. What is more, extensions to SPARQL exist, like SPARQL^T, a temporal extension for SPARQL [54].

The SPARQL queries are built under the notion of graph pattern matching. The smaller component of a graph pattern is a triple pattern, i.e. triples

```
SELECT ?s ?g WHERE {
  GRAPH ?g {
    ?s <http://relation.org/locatedIn> <http://example.org/location#Vienna>
  }
}
```

Figure 4: An example of a SPARQL query.

in which each of the subject, predicate and object may be a variable (these are the atomic queries that will be used in our experiments in Section 7.6.

An example SPARQL statement, querying Graph_B for all entities located in Vienna is shown in Figure 4.

2.4 Linked Data Applications

On the one hand, Linked Data can be used in various domains, including biology [152], statistics [71], software engineering [82], multimedia [72] and finance [94]. On the other hand, Web applications can be built on top of Linked Data [69]. The Linked Data community provides a list of such datasets and links between datasets in the Linked Open Data cloud [1]. The mentioned two options surely are intertwined, yet in this section the examples mostly fall into the latter category.

As a rough categorization, Linked Data applications can be grouped into four categories, which highlight the main aspects from a Linked Data usage point-of-view [69]:

- **Content reuse.** These are applications that mainly reuse existing content of datasets in the LOD cloud in order to save time and resources. An example application of this category is the Understanding Advertising (UAd) Analyzer [128]. It is a research prototype of a Linked Data Web application for market researchers built to trace discussions on the Web. It interlinks discussions throughout a couple of Web-based discussion forums (via SIOC [27] and FOAF [28]) and uses DBPedia [36, 13] categories to pull in domain-specific information. Another example in this category is BBC music [15] which pulls data from other sources like Musicbrainz [133] and DBPedia [87].

- **Semantic tagging and rating.** Applications of this category make use of HTTP URIs in the datasets to unambiguously talk about things. The reviewing and rating site Revyu [74] applies the Linked Data principles. Besides publishing Linked Data, the Web application exploits the interlinking with DBPedia.

- **Integrated question-answering systems.** Answering a user's question is in the focus of the applications of this category. CrunchBase [81] is a free dictionary of technology companies, people and investors. Around its public API a Linked Data wrapper was built [43]. The CrunchBase Twitter bot [107] uses this data to answer questions about

[1] http://linkeddata.org/

Silicon Valley companies. Another example is DBpedia mobile [16], which is basically a location-centric DBpedia for mobile devices [70]. Based on the GPS of the device a map is rendered that shows nearby locations from the DBpedia dataset. In doing so, the application answers the question "What places are around my current location?".

- **Event data management systems.** This category covers applications which allow people to organize and query event-related data. For the European Semantic Web Conference 2009 the so called "confx" tool was created. It is a Linked Data dashboard which allows users to view papers, sessions and discussions in and during a conference [108]. What is more, OpenLink Data Spaces (ODS) [130], a distributed collaborative application platform for creating presence in the Semantic Web, supports a range of applications, including a calendar. A number of query services such as SPARQL are supported, too.

Virtuoso is an example for a data store for Linked Data. It is further described in Section 3.1.2 and has been used for several projects in the Linking Open Data Project [24], including Dbpedia [12], Musicbrainz [133], Geonames [148] and PingTheSemanticWeb [56] [41].

3 Related Work

In this section, we briefly review the state of the art of RDF triple stores, with a focus on Jena [99] and Virtuoso [41], the two most popular open RDF stores. The focus is on their capability of handling RDF graphs and especially their used indexes for RDF query resolution.

After that, different approaches for versioning of RDF graphs are discussed, namely independent copies, change based and timestamp based as well as hybrid approaches.

3.1 RDF Triple Stores

RDF data can be persisted in relational databases [129, 112] to make the data not only modifiable, but also queryable. However, as relational database management systems (RDBMSs) were not built to support the distinct structure of RDF graphs, several problems arise for the triple storage. These include difficult to calculate query costs and the fact that cast rules are more permissive than in Structured Query Language (SQL) [41].

Besides the widely used relational model, newer formats like noSQL emerged and are becoming increasingly popular, especially in niche areas. Such databases

store data in a nonrelational way, which includes column-oriented stores (like Facebook's Cassandra [89]), key-value stores (like Amazon's Dynamo [38] and Amos II [120]), document-based stores (like CouchDB [8] and MongoDB [32]) and graph databases (like Neo4j [121]) [93].

In addition to that, there are native stores, that is, stores specifically designed for RDF. These are commonly based on B-tree indexes over multiple orders of the triples (e.g. SPO, POS, OPS). Two popular representatives are RDF3x [106] and Hexastore [146].

Although RDF graphs can be stored in such noSQL databases [35], it is still a valid and popular approach to persist RDF graphs in RDBMS systems and even to run SPARQL queries against them [145, 78]. Two RDF triple stores, Apache Jena and Virtuoso, are briefly introduced in the subsequent sections.

3.1.1 Apache Jena

Jena is a toolkit for Semantic Web programmers [99]. With the use of Jena, an API in the Java programming language, one can create and manipulate RDF graphs. Moreover, Jena includes a data publishing server which is very convenient as it is a common requirement to publish data over the Internet in modern applications. The server is called Fuseki and can present and update RDF models over the Web using SPARQL and HTTP [52]. Open source implementations of Jena are available on the Web [98, 149].

Jena uses a JDBC connection to an SQL database to persist RDF graphs. It supports a number of database engines (e.g. PostgreSQL, MySQL, Oracle) and has a flexible architecture that makes it possible to port to new SQL database engines [149].

A Jena TDB store, the Jena component for storing and querying RDF graphs, is stored in a single directory in the filing system and consists of a node table, triple and quad indexes and the prefixes table. The first of these, the node table, stores the representation of RDF terms. It can be called a dictionary as it maps nodes to node IDs and vice versa. A large cache is usually provided by node table implementations, as node ID to node mapping is heavily used in query processing [53].

To resolve queries efficiently, Jena makes use of indexes. A total of 9 indexes are created in a TDB store. 3 of them are for triples, namely: SPO, POS and OSP. The other 6 are for quads, namely the SPOG, POSG, OSPG, GSPO, GPOS and GOSP. As can be seen, the 3-component indexes and the 4-component indexes have the same order, except that the 4-component indexes additionally have the graph component as their first or last element. While in Virtuoso secondary (partial) indexes are used to answer queries (as

explained below), in Jena each index has all information about a triple [53].

The third main part of TDB is the prefixes table, which uses a node table and an index for GPU (Graph-Prefix-URI). It does not take part in query processing and is usually small. Mainly it is used for presentation and serialisation of triples in RDF/XML or Turtle [53].

3.1.2 Virtuoso

Virtuoso is a server providing ODBC/JDBC access to relational data. Data can either be stored in Virtuoso itself or in an external relational database. What is more, Virtuoso has a built-in HTTP server which provides several protocol end points and supports a variety of scripting languages to build dynamic Web pages. Because of the growing importance of the Semantic Web, RDF functionality was incorporated into the system [41].

Quads are stored in Virtuoso in a quite simple way. There is a single table of four columns, one for the subject, the predicate, the object and the graph. While the subject, the predicate and the graph are IRI ID's, for which a custom data type exists, the object column is of SQL type ANY, so it can hold any serializable SQL object [41]. If the RDF data consists of multiple graphs, the data can either be stored in a single table or multiple tables (one per graph). Which of these options is used depends on the number and size of the graphs [129].

Virtuoso offers SPARQL inside SQL and inherits all the grouping and aggregation functions of SQL, as well as user defined and any built-in functions. All supported command-line interfaces directly work with SPARQL without modifications. When the system is asked to evaluate a SPARQL query, the query is first translated into SQL [41]. SPARQL goes wherever SQL does, as a SPARQL statement is a valid top-level SQL select [129].

To be able to answer queries efficiently, a number of indexes are defined on the tables storing the RDF data. Firstly, there are two full indexes over the RDF quads. The first one, PSOG, is at the same time the primary key. The second one, POGS, is a bitmap index for lookups on object value. Secondly, there are 3 partial indexes. The SP index is for cases in which only the subject is specified. The OP index, on the other hand, is for cases in which only the object is specified. Lastly, the GS index is for cases where only the graph is specified [135].

Consequently, Virtuoso favors queries where the predicate is specified. Quads can be accessed efficiently if the predicate and at least either the subject or the object is known. If only the subject is known, the SP index is used to find the corresponding predicates. Then, the PSOG index can be used to find the other quad parts. The same is true if only the object

is known. Using the OP index, the predicates are found. Using the POGS index the full quads are retrieved. If only the graph component is specified, the GS index is used to find the respective subjects and subsequently the SP index is used to find the corresponding predicates. Lastly, to find the whole quads, the PSOG index is used [135].

3.2 RDF Versioning Approaches

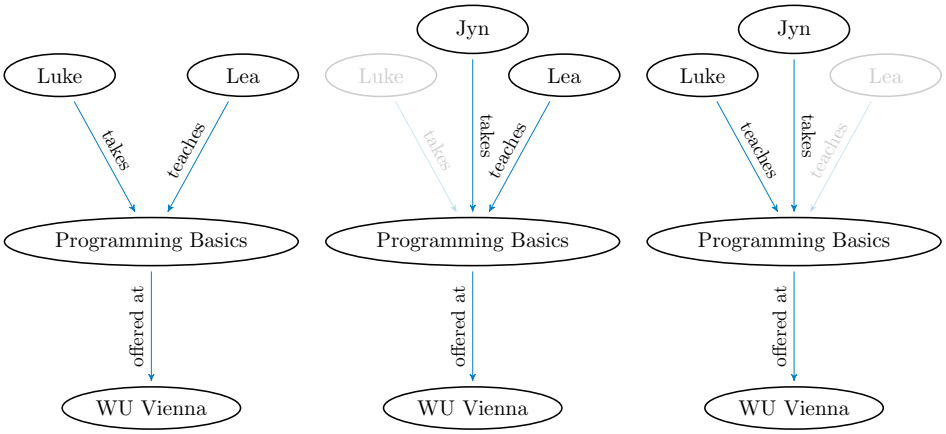
Published data is continuously undergoing changes (not only on a data, but also on a schema level) [141, 84]. Following the scale-free nature of the Web, such changes neither happen with centralized monitoring nor pre-defined policy. Applications and businesses that make use of the availability of data over time and are interested in data changes and evolution, need to build their own infrastructures to preserve and query data over time [48].

Aimed at assuring quality and traceability of Semantic Web data over time, archiving policies of LOD collections are a novel challenge. The overall objective is the same as for traditional Web archives, like the Internet Archive [10]. However, capabilities for time-traversing structured queries should additionally be offered by archives for the Web of Data. Neither SPARQL nor existing temporal extensions of it do natively support time-based capabilities, such as knowing whether a dataset or a particular entity has changed [134, 58, 117, 153]. Recent systems [47, 54], however, are starting to offer such time-based capabilities [48].

Because of the relative novelty of archiving and querying Semantic Web data, retrieval needs are not broadly implemented in practical implementations. What is more, these retrieval needs are not even fully described. However, first categorizations [132, 47, 88] distinguish six different types of retrieval needs, which are version materialization, single-version structured queries, cross-version structured queries, delta materialization, single-delta structured queries and cross-delta structured queries [47]. These needs can be classified by query type (materialization or structured queries) and by the main focus (version or delta) of the query [48].

In order to satisfy the retrieval needs, one needs to store different versions of the same graph. Several research efforts address this challenge. Mainly, there are three different storage strategies [138]: IC, CB and TB approaches [47]. These approaches are described in the upcoming sections.

To explain the different versioning approaches, Graph_C, a subgraph of Graph_A, is introduced. The graph and its three versions can be seen in Figure 5. In the first version, Lea teaches the Programming Basics course and Luke takes it. In the second version, Luke left the course and now Jyn takes it instead. In the third version, Luke reappears, but now teaches the



(a) Version 1 (b) Version 2 (c) Version 3

Figure 5: Graph_C in its different versions.

course and therefore replaces Lea who is no longer part of the graph. In all three versions, Programming Basics is offered at WU Vienna.

The subject of this work is to extend HDT with graph information, which is not necessarily versioning information. However, in general a certain similarity of versioning and adding graph information cannot be denied. Adding version information to a graph can be seen as a special case of adding graph information. For this reason, after introducing versioning approaches and HDT in the upcoming sections, their suitability to add graph information to HDT is discussed in Section 5.

3.2.1 Independent Copies

The IC policy [86, 109] stores and manages each version as a different, completely isolated dataset. This is shown for the three versions of Graph_C in Figure 6. Additionally, a metadata characterization can be built on top to provide a catalogue of the different available versions, e.g. using the Provenance Ontology (PROV) [60] and the Data Catalog Vocabulary (DCAT) [9]. As the so called "static core", which are triples that do not change, is fully repeated across versions, IC suffers from scalability problems [47]. In the example this can easily be seen. The triple "Programming Basics - offered at - WU Vienna" is stored in all three versions of the graph.

While triples appearing in all versions of the RDF graph are the worst case and lead to the biggest scalability inefficiencies, triples that appear in several, but not all, versions are a scalability problem as well. Assuming that there are 100 versions, where a triple appears in 99 of them and does not

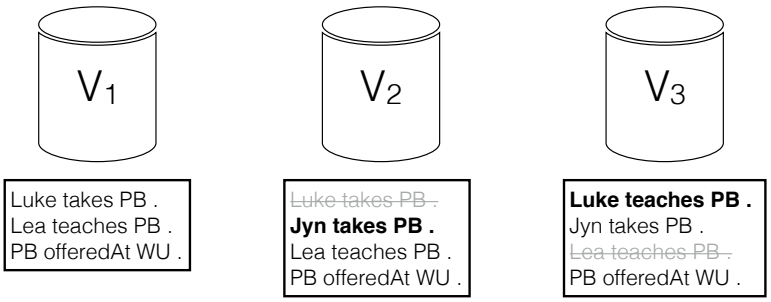


Figure 6: Independent copies of Graph_C.

appear in the last version, the triple would be repeated 99 times across the copies.

On the one hand, IC suffers from the aforementioned scalability problems in space. However on the other hand, it is a very simple, straightforward approach. Especially for basic retrieval purposes, like version materialization, IC fits very well and comes with low effort. On the contrary, the rest of the operations require medium or high processing efforts. To answer the other queries, a retrieval mediator should be placed on top of the versions, which has the challenging tasks of [47]:

- Calculating deltas at query time to satisfy delta-focused queries.
- Solving the structured queries after loading and accessing the appropriate version(s).
- Performing both previous tasks for the case of structured queries that deal with deltas.

The IC approach is widely used to directly provide historical version dumps (although typically compressed to reduce space needs of textual RDF formats), like in DBpedia [36], the dynamic Linked Data Observatory [140] as well as other projects serving Linked Open Data snapshots [47].

3.2.2 Change Based

The CB approach addresses the scalability problems of IC by storing the differences (deltas) between versions [47]. One option is to store the differences on the level of triples (so called low-level deltas [144, 79, 151]), which can be seen in Figure 7. Here, only the first version is stored completely and the other versions are expressed as the difference to the respective previous version, i.e. as additions (Δ^+) and deletions (Δ^-).

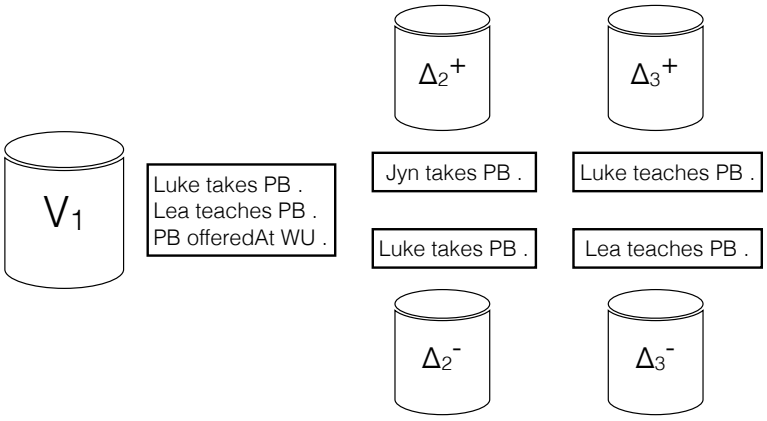


Figure 7: Change based archiving of Graph_C.

The change operations are based on a basic language of changes, which typically marks added and deleted triples and which can be shared in markup languages like RDF Patch [125]. There are complementary works that focus on computing dataset differences in a distributed way, like rdfDiff [77] and G-Diff [4], which are both working on MapReduce [37]. Other approaches work on extracting high-level deltas [114, 110] which are human readable. These focus on obtaining a more concise explication on the whys and hows of the changes. Low-level deltas apply to any RDF dataset and are easy to detect and manage. In contrast, high-level deltas are more descriptive and can be more concise. This comes at the cost of being more complex to detect and manage and relying on underlying semantics (such as Resource Description Framework Schema (RDFS) and the W3C Web Ontology Language (OWL)) [150, 47].

On the one hand, required space is reduced compared to IC. On the other hand, additional computational costs occur for delta propagation and therefore version-focused retrieving operations. Here, a query mediator accesses a materialized version and the subsequent deltas. Using the CB approach leads to a very cheap delta materialization operation. However, version-focused queries can be quite costly because of the aforementioned delta materialization. Structured queries also require some effort, since the appropriate deltas or re-created versions need to be loaded or accessed [47].

One advantage of this approach is its high configurability [47]:

- One can choose between different approaches to detect and store the differences (e.g. low or high level deltas), as explained before.
- Deltas can be calculated backward (the changes of version V_i with respect to version V_{i-1} are computed) or reverse (the changes of version

V_{i-1} with respect to version V_i are computed).

- Either only one fully materialized version is stored and all other versions are represented as deltas, or some intermediate versions are stored fully materialized, too.

Tradeoffs of the latter are much discussed in the literature. While there are attempts to precompute an aggregation of all deltas to improve cross-delta computation at the cost of augmenting space overheads [79], others propose a theoretical cost model to adopt a hybrid (IC + CB) approach [132]. The difficulties of constructing and reconstructing versions and deltas, which depend on multiple and variable factors, are a driving factor for these costs. Building a partial order index to keep a hierarchical track of changes is also proposed [138]. This approach, however, is a limited variation of delta computation and is not tested with datasets on a large scale. Also, there are hypergraph-based solutions [80], which store the information of version in hyperedges [47].

3.2.3 Timestamp Based

Three forms of annotating RDF respectively Linked Data with temporal information are distinguished by research works in the Semantic Web area [122, 7, 47, 31]:

- **Document-centric**. Here, time points are associated with whole RDF documents. This annotation can be implicit or explicit. An example for the former one is HTTP metadata which can be used to detect changes [84]. Vocabularies to annotate metadata about datasets, like the Vocabulary of Interlinked Datasets (VoID) can be used in the latter case. Because of the distributed nature of LOD, provenance information of data collections is of increasing interest. To query RDF documents with time, the W3C PROV [60] standards can be used [124].

- **Sentence-centric**. Using sentence-centric annotation, temporal validity (a time point or interval) is defined at the level of statements or triples [139, 134, 119, 65, 153].

- **Relationship-centric**. Here, time is encapsulated in n-ary relations [111]. Specific resources identify the time relation and make use of it to link to other related resources [147, 101]. This is one special case of multidimensional modeling [85, 55].

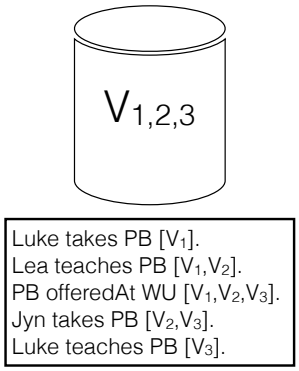


Figure 8: Timestamp based archiving of Graph_C.

The timestamp-based approach can be seen as a particular case of the sentence-centric annotation to model the time in RDF. Each sentence locally holds the timestamp of the version instead of explicitly defining the temporal validity of statements or triples. Note that the static core of the archive would again produce repetitions, since static triples would be labeled with all the present timestamps. To avoid such repetitions and therefore save space, practical proposals annotate the triples only when they are added or deleted. So the triples are extended by two different fields, namely the created and (if present) the deleted timestamps [105, 124, 59]. These approaches manage versions/deltas under named/virtual graphs, so that a retrieval mediator, which acts on top of the archive, can rely on existing solutions providing named/virtual graphs [47].

While delta materialization can be satisfied with low effort, all other retrieval demands can be done with medium processing efforts, given that version materialization requires to rebuild the delta similarly to CB and irrelevant triples may need to be skipped by structured queries [105, 47].

Figure 8 shows the timestamp based archiving of Graph_C. Note that, using this approach, each triple appears exactly once in the archive. The figure shows the not so efficient way of marking each triple with all the versions in which this very triple appears, instead of only highlighting the created and deleted timestamps. For this reason, the triple "Programming Basics - offered at - WU Vienna", which appears in all three versions of the graph, is marked with V_1, V_2 and V_3.

3.2.4 Hybrid

To strike a balance between query performance and storage space, a hybrid approach for archiving can be chosen. Stefanidis et al. suggest a solution

where some of the versions are stored under full materialization, while others are only stored as deltas. Applied correctly, this would lead to modest space requirements while at the same time only a small overhead at storage and query time is introduced, so the best of both worlds can be enjoyed. To determine if a specific version should be stored fully materialized or only as a delta is the challenge of this approach. Thus, an appropriate cost model is needed which quantifies and compares additional time and space overhead of the two policies. In other words, if an existing Version V_{i-1} evolves into V_i it must be decided whether to store V_i itself, or the appropriate δ_i [132].

4 HDT Overview

Several challenges come up when processing huge real-world RDF graphs. These challenges include [46]:

- Metadata about the graph (like statistics and a content summary) for published RDF datasets is often neither complete nor systematically published along with the dataset. An additional challenge is that such data is often not published in a machine-readable format. Instead it is e.g. in natural language posted on the Web page of the data.
- RDF graphs can have millions or billions of nodes and links. Graphs of such sizes raise problems concerning management, exchange and consumption.
- Already in 2012 the number of smartphones used around the globe exceeded 1 billion [123]. With the growing importance of mobile devices, one needs to consider their memory constraints and transmission costs when managing huge graph information [92].
- Basic data operations like simple lookups become very inefficient because of the sequentiality of information in the published files. To evaluate a query, the entire data must be parsed.

From the problems listed above, one can derive some basic requirements that need to be covered when publishing and exchanging large RDF datasets. First, at the logical level, such huge datasets need standardized meta data information. This information includes statistics (like size, quality and type of data), intellectual property information (copyright), provenance (source, providers, date of publication) and editorial data (publisher, version). Second, at the physical level, an RDF representation should enable efficient exchange, management and processing. Third, at the operational level, simple query patterns should be natively supported [46].

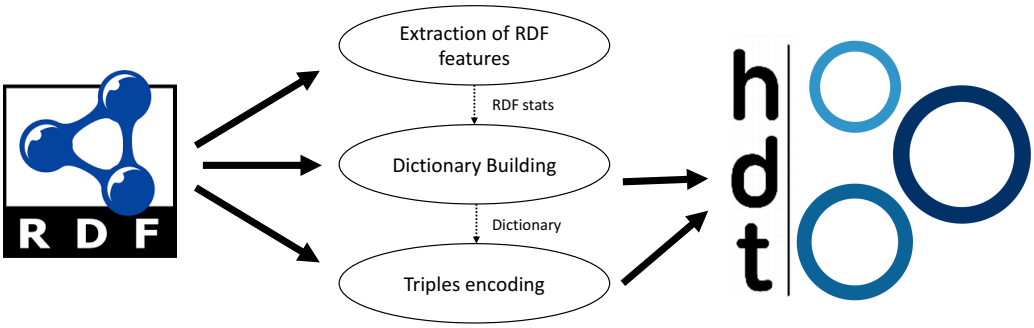


Figure 9: A step-by-step creation of HDT from an RDF graph.

HDT is an RDF publication and serialization format. It addresses the issues outlined above. HDT modularizes the data and exploits the skewed structure of big RDF graphs [39, 113, 137] to achieve large spatial savings. It is based on three main components matching the levels outlined above [46]:

- **Header**: The header includes the metadata of the RDF dataset. It can be seen as the entry point to the information of the graph. The header component is described in more detail in Section 4.1.
- **Dictionary**: The dictionary organizes all identifiers of the RDF graph. The RDF terms (IRIs, literals and blank nodes, described in Section 2.2) mentioned in the graph are provided in a catalog with high levels of compression. The dictionary component is described in more detail in Section 4.2.
- **Triples**: The pure structure of the underlying RDF graph is comprised in the triples component. It encodes the set of triples in a compact way, while removing unnecessary repetition (redundancy) of long labels. The triples component is described in more detail in Section 4.3.

The name Header-Dictionary-Triples (HDT) is deduced from the components listed above.

The schematic steps of the process to obtain an HDT representation of an RDF graph can be seen in Figure 9. The first steps extract information from the graph necessary to build the header, the dictionary and the triples. These components are then combined in HDT [46].

4.1 Header

The header component provides metadata information about the RDF dataset. There are other approaches to store the metadata in the graph itself. Namely

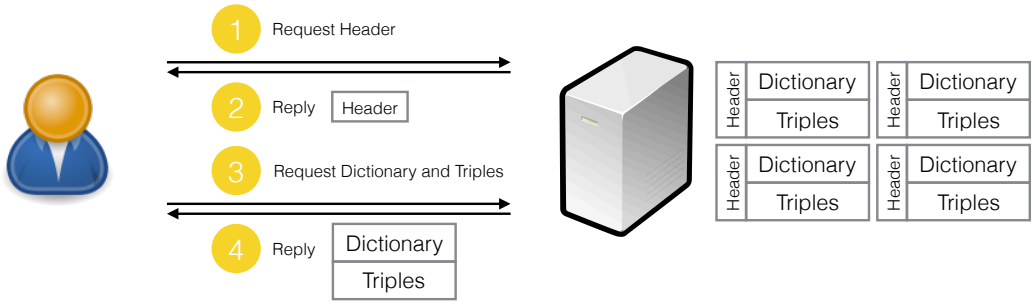


Figure 10: A potential use-case for the HDT Header.

VoiD [5] or the OWL vocabulary, which lists various annotation properties [103]. However, this makes it difficult to automatically distinguish between data and metadata. Compared to other serialization formats, which do not provide any means or at least best practices on how to provide metadata along with the datasets, HDT gives metadata a dedicated place as part of the header information [46].

The data provider can include a desired set of features in the header component, which makes it flexible. Four different kinds of metadata can be stored in the header [46]:

- **Publication information**. This part contains information about the publication act. It can contain the site of publication, dates (creation and modification), the version of the dataset, encoding, language, namespaces etc. Additionally, authority information about the source of data can be included.

- **Dataset statistics**. Statistics about the dataset which are often needed can be precomputed and included. Examples are the total number of triples and the number of distinct subjects in the graph.

- **Format information**. In this section information about the specific format of the RDF dataset is stated. The concrete dictionary and triples implementations (and their physical locations) are specified.

- **Other information**. Any other information that helps to understand and manage the data can be included by the provider.

The HDT header for Graph_A introduced in Section 2.2.1 can be seen in Turtle syntax in Figure 11. In this human- and machinereadable format, one can easily get an overview about the graph. There are 14 triples in the graph and there are 8 distinct subjects, 10 distinct predicates and 7 distinct

objects. Even though the graph is very small, HDT reduces the size to about 20% of the original file size. The compression ratio becomes far better with a bigger graph.

```
@prefix void: <http://rdfs.org/ns/void#>
@prefix hdt:  <http://purl.org/HDT/hdt#>
@prefix dc:   <http://purl.org/dc/terms/>
@prefix w3:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

<http://example.com/3.5> w3:type hdt:Dataset ;
                         hdt:publication _:publication ;
                         hdt:statisticalInformation _:statistics ;
                         hdt:formatInformation      _:format .

_:publication dc:issued "2017-05-27T14:05Z" .

_:statistics hdt:originalSize "1630" ;
             hdt:hdtSize "327" ;
             void:triples "15" ;
             void:properties "10" ;
             void:distinctSubjects "8" ;
             void:distinctObjects "7" .

_:format hdt:dictionary "_:dictionary" ;
         hdt:triples    "_:triples" .

_:dictionary dc:format hdt:dictionaryFour ;
             hdt:dictionarynumSharedSubjectObject "6" ;
             hdt:dictionarysizeStrings            "296" .

_:triples dc:format             hdt:triplesBitmap ;
          hdt:triplesnumTriples "15" ;
          hdt:triplesOrder      "SPO" .
```

Figure 11: An example of an HDT header.

A potential use case of the HDT header can be seen in Figure 10. A machine or a user visits the Web page of a provider. The headers of the published datasets can either be queried online or the header is downloaded (step 1 and 2 in the figure, the user requests the header and the server replies with it). Consequently, the consumer can access publication information, dataset statistics, format information and other metadata. The metadata could inform the consumer that the data is available in different formats, is distributed in several chunks and multiple versions are published. As a result, consumers can retrieve [46]:

- The relevant chunk from the large collection of published datasets (which minimizes the exchange).
- The best-fitting format for the consumers specific use-case (when considering the trade-off between functionality and compression ratio).

- The desired version.

When the user chooses to download a specific HDT file, the server is asked to send the rest of the HDT file (step 3). Consequently, the server replies with the desired data (step 4).

The use case described above is what was envisioned for the header originally. However, current HDT libraries only support a more trivial use case, where the full HDT is downloaded.

The operations over the header are not more complex than operations over a general metadata file. Typically, the header is written once by the publisher, but it could also be updated with newer information. Consumers either consume the header using SPARQL queries or download and access the header locally. There are only two constraints: The metadata should be machine readable and it should be possible to query a given type of metadata [46].

4.2 Dictionary

A data dictionary may be defined as a centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format [100].

Most RDF formats allow the abbreviation of long, repeated strings (IRIs, Literals, etc.). One example for an entry in such an elementary version of a dictionary for namespaces and prefixes is "http://www.w3.org/1999/02/22-rdf-syntax-ns#type", which is a very common component of triples in the billion triple challenge dataset [67]. Abbreviations of this kind are provided in XML (in the form of namespaces in conjunction with XML Base) as well as in several RDF formats (@base, @prefix in N3 and Turtle) [46].

The fact that RDF datasets should be managed by automatic processes favors a very effective replacement. This replacement is done by the dictionary by assigning a unique ID to each distinct element in the dataset. In doing so, long repeated strings in triples are replaced by short IDs, which contributes to the goal of compactness. Often, the first step in RDF indexing is this very assignment of IDs, referred to as URI to identifier mapping [33]. While there are other approaches which exploit the dictionary construction besides the RDF stores [96, 142], HDT is the first proposed RDF representation syntax that includes a dictionary [46].

Multiple configurations and implementations of the dictionary component are allowed in HDT. Usually RDF engines map shared elements (the sets of subjects, predicates and objects in RDF are not disjoint) with the same

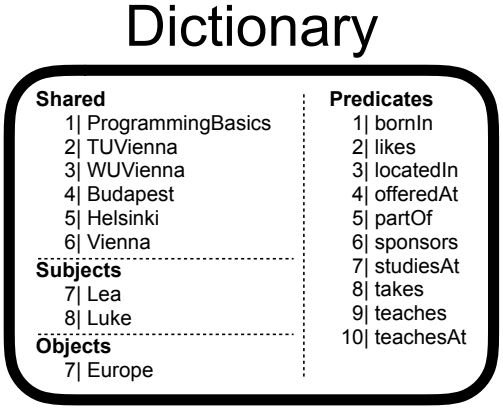


Figure 12: An HDT dictionary.

ID [11]. Elements within each set could be sorted by some property, for instance the alphabetical order or frequency of use, or be in random order [46].

Figure 12 shows the dictionary created for Graph_A. As can be seen, the same ID is assigned multiple times (e.g. ID "7", which is simultaneously present in the subjects, objects and predicates list). However, with the knowledge of the position in the triple and the ID, the dictionary can unambiguously resolve the ID and return the correct string. The "Shared" section contains values that appear in both the subjects and the objects. As the example graph holds quite a number of such cases, this contributes greatly to the compactness of the dictionary and therefore to the compactness of HDT.

As the dictionary replaces IDs, two important and minimum operations are required [46]:

- **locate(element)**: If the given element appears in the dictionary, its identifier is returned.
- **extract(id)**: If there is an element behind the identifier id, it is returned.

Apart from its compacting features, the dictionary can also help in query evaluation and resolution. An example is SPARQL's filter operation, which restricts the final result by a given condition. The condition can be evaluated first over the dictionary, as it usually refers to a regular expression, language or datatype selection. What is more, the range to search in the structure of triples will be delimited by the elements satisfying the condition [46].

4.3 Triples

By using the dictionary component, one can express original RDF triples as triples of IDs, where each element in the former triple is replaced by the corresponding ID from the dictionary [46]. Looking at the highlighted triple of Graph_A in Figure 1 "WU Vienna - located in - Vienna" together with the dictionary shown in Figure 12, the triple can be transformed to "3 - 3 - 6". Note that the triples are denoted in subject-predicate-object (SPO) order, which resolves the ambiguity of the IDs with value 3. The first one must be a subject, which leads to the "WU Vienna" entry of the shared section, the second one must be a predicate, which therefore leads to the "located in" entry in the predicates section.

The original stream of strings is now a stream of IDs. By this, the triple component also contributes to the compactness of the information. Moreover, the triple component is the key component to access and query RDF graph information. Different configurations and implementations of the triple component can focus on other aspects, like the compression ratio or the supported operations over the triples. The RDF triples' format should be designed to optimize common operations and uses of them. Four fundamental operations can be distinguished [46]:

- **Exchange**. The basic feature of an RDF triples component is to compact the RDF statements of the graph, which optimizes the objective of efficient exchange. An additional feature could be the functionality to exchange only a subgraph instead of the entire graph, but this is not yet implemented.

- **Basic Search**. RDF triples where one or more elements can be a variable are called triple patterns. These patterns are an important foundation for any search over RDF triples. Overall, there are eight patterns (SPO, ?PO, S?O, SP?, ??O, ?P?, S??, ???). As many of these patterns as possible should be resolved efficiently by an RDF triples component.

- **Join Resolution**. Joins, which imply matching two or more triples patterns which share one or more variable, are among the most expensive operations in RDF queries. The most common types of joins (i.e. Subject-Subject, Object-Object, Subject-Object) should be supported by RDF triples components.

- **Complex Querying**. Efficient answering of any SPARQL query would be ideal. This addresses query evaluation optimization techniques as well as many operations and modifiers (like union and optional).

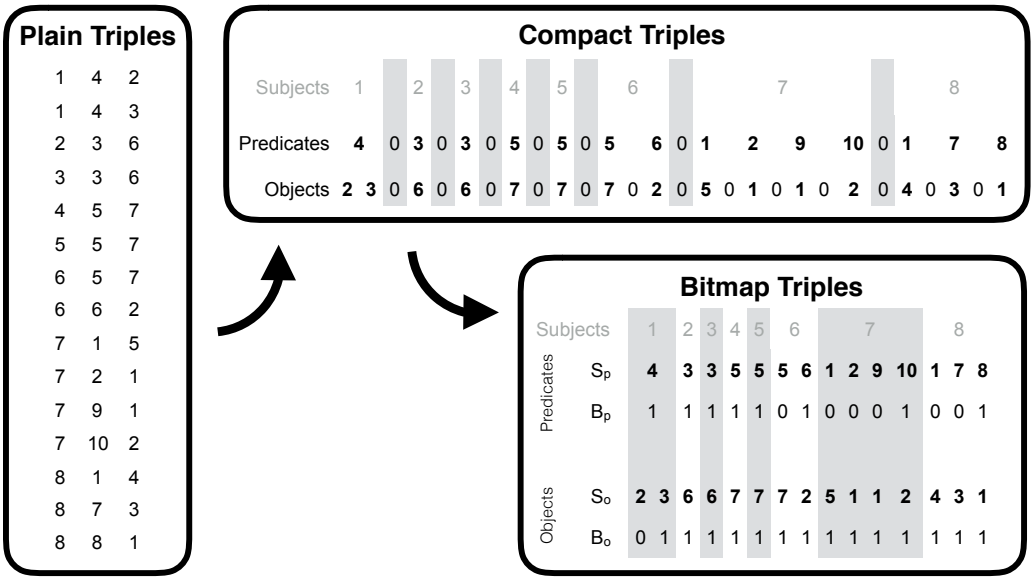


Figure 13: Three encodings for triples in HDT.

One of the keys for good RDF query performance is the efficient indexing of the triples structure. As RDF data is usually exchanged in verbose, plain formats, these indexes need to be created locally. The triples component of HDT is designed in a way that the compressed triples can be queried without the need to decompress it first, which encourages the exchange of the data [6, 46].

Fernández, et al. provide three different implementations for the triples component encoding [46]. The different triple encodings (Plain Triples, Compact Triples and Bitmap Triples) for Graph_A can be seen in Figure 13.

Plain Triples. The Plain Triples encoding is the most basic of the three shown encodings. Using the dictionary, each triple is translated into a triple of IDs [46].

Compact Triples. For this option, triples are sorted by subject. Furthermore, Compact Triples implies the creation of two adjacency lists, one for the predicates and one for the objects. Such adjacency lists are compact data structures that facilitate searching and maintaining [46]. Note that the gray subjects list in Figure 13 is not created in HDT, it is only drawn for better understandability.

After the triples are sorted by their subjects, they are grouped per subject, while maintaining the established order. As the first group (list) belongs to

the first subject, the second group to the second subject and so on, the subject representation can be omitted, which leads to an immediate saving.

In the next step, two coordinated streams of predicates and objects are created by splitting the representation. While maintaining the grouping order, the first stream corresponds to the lists of predicates associated with subjects. To mark the end of a predicate list, a zero is inserted. As the dictionary does not assign zero as an ID, this is a valid separator. The so marked end of a predicate list implies a change of the subject [46]. The zeros are highlighted in Figure 13 to better illustrate the ending of a predicate list and therefore the beginning of a new subject. Spaces between the figures are inserted for legibility reasons, in fact in the example the predicates stream contains less elements than the object stream (actually, the object stream always contains the same number of or more elements than the predicate stream).

The second stream, which represents objects, groups the list of objects for each subject - predicate pair. In the same way zeros in the predicate stream mark the end of a predicate list, a zero in the objects stream indicates the end of an object list and indicates the need to move forward in the predicate stream [46].

The Compact Triples representation is a compact ID-based triples representation which reduces the classical three-dimensional view of RDF into a two-dimensional view, by making the third dimension implicit [46].

Bitmap Triples. Compact Triples, described above, use two coordinated ID-based streams (one for predicates, one for objects) to draw the RDF graph using an implicit subject-grouping strategy. Both streams are basically sequences of non-negative integers, where zeros mark the end of the respective adjacency list. The graph structure is denoted by the positive integers (predicates and objects) and embedded auxiliary values (the zeros). The Bitmap Triples representation splits these two parts [46].

The Bitmap Triples encoding extracts the zeros from the adjacency lists of the predicate and object streams. Two bitsequences (B_p and B_o), one for each adjacency list are created, in which ones mark the end of an adjacency list [46].

In Figure 13 this transformation can be seen. The zeros are removed from the predicates listed in the Compact Triples representation, which leads to the sequence S_p and the equal-length bitsequence B_p. The same steps are carried out for the objects, which leads to the sequences S_o and B_o [46].

The sequence of predicates and its corresponding bitmap can be interpreted as follows. The end of the predicate adjacency list of the i-th subject (the list is referred to as P_i) is marked by the i-th 1-bit of the bitmap. To

get the number of predicates in the respective list, one needs to subtract the position of two consecutive 1-bits [46]. When looking at the example, the 7^{th} 1-bit marks the end of the predicate adjacency list of subject 7. By subtracting the position of the 6^{th} 1-bit (7) from the position of the 7^{th} 1-bit (11), the length of the predicate list for subject 7 can be calculated ($11 - 7 = 4$). Therefore, the 8^{th}, 9^{th}, 10^{th} and 11^{th} predicate belong to subject 7, which results in $P_7 = \{1, 2, 9, 10\}$.

Note that in this representation too, the subjects are omitted. The subject IDs in the figure should help to understand the representation more easily. The subjects are further highlighted in alternating colors. Every highlighted area ends with a 1-bit in the predicate adjacency list (as the 1-bit marks the end of a subject).

The sequence list and the bitmap for the objects work in the same way as the respective lists for the predicates. Here, the end of the object adjacency list for the j-th subject/predicate pair is marked by the j-th 1-bit in B_o. The predicate can easily be found, it is represented by the j-th position in B_p and is retrieved from the j-th position in S_p. For example, the sixth 1-bit in B_o refers to the end of the object adjacency list of the sixth predicate in S_p, which is related to the sixth subject, as can be seen in Figure 13 and as explained above. The list therefore holds the objects o for all triples $(6, 5, o)$ in the graph [46].

The bitsequences that are used for B_p and B_o make use of succinct structures. Two important operations are supported by these [46]:

- **rank**. The rank operation counts the number of occurrences of a symbol (in this case "0" or "1") in a specific subsequence of the bitsequence.
- **select**. The select operation finds a specific occurrence of a symbol (in this case "0" or "1") in the bitsequence.

This has been solved with a space requirement of $n + o(n)$ bits and a constant query answering time [34]. Fernández et al. make use of the approach of González et al. [57] to implement B_p and B_o [46].

The bitmap triples representation is the most compact solution of the three encodings described above. Furthermore, it offers the possibility to directly access the compressed data [46]. For this reason, the bitmap triples encoding is the preferred triples representation and will be used for the rest of the thesis.

To be able to resolve triple pattern queries efficiently, HDT makes use of 3 indexes. Firstly, the BT (Bitmap Triples) SPO index, further referred to as subject index, which is used to resolve patterns with the subject component given. The subject index is basically the HDT representation itself, as

described above with its bitmaps and adjacency lists. A special case is the S?O pattern, which cannot directly be resolved using the subject index, but needs an additional sequential iterator.

Secondly, the HDT-FoQ (HDT Focused on Querying) [97] PSO index, further referred to as predicate index, which is used to resolve ?P? queries. For this, an additional integer sequence is created, which stores, for each predicate, a sorted list of references to the subject-object pairs related to it. By using this list, subject-object pairs related to a given predicate can be retrieved efficiently. Thirdly, the HDT-FoQ OPS index, further referred to as object index, which is used to resolve ?PO and ??O queries. The object index works in the same way as the predicate index. For each object, it stores a sorted list of references to the subject-predicate pairs related to it.

The pattern ??? can be resolved by any of the three indexes. In practice, the subject index is used to resolve the pattern.

5 Adding Graph Information in HDT

RDF data can not only contain triples, but also quads. In this case, the fourth component, the graph, assigns each triple to a particular graph. As the same triple can appear multiple times in an RDF dataset, it can be assigned to multiple graphs as well. If an RDF dataset contains a graph component for its triples (and therefore quads), it is said to contain named graphs.

While a SPARQL triple pattern tp is defined as $(I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$, where I is an IRI, B is a blank node, V is a variable and L is a literal, a quad pattern may be defined as $tp \times (I \cup V)$, where the last component denotes the graph of the triple pattern (an IRI or variable).

In the course of this work, we use the convention SPOG to refer to the SPARQL query GRAPH G { S P O . } where G is the graph $\in (I \cup V)$ and SPO is the triple pattern tp.

The graph components of the triples can for example represent the origin of the given triple. Hence, a given triple can be assigned to a number of potentially overlapping graphs (sources). What is more, the graph component of the triples can be used for representing timestamps. So the fourth component would indicate in which version a triple is valid.

Particularly if using the graph component as a timestamp, one can see a similarity to the versioning approaches discussed in Section 3.2. Because of this similarity, it is briefly discussed whether an adapted form of the presented versioning approaches can be used to extend HDT with graph information.

The first versioning approach discussed above is the independent copies

approach. At first glance, this approach can easily be applied to extend HDT with graph information. This is by creating one HDT instance per graph. On the one hand, this would favor queries that ask for triples in a specific graph. On the other hand, however, creating multiple HDT instances would not favor HDT's compactness. In an advanced approach, a common header and dictionary could be used, but the triple's section would still appear multiple times and can include duplicates. For this reason, the independent copies approach does fit well to extend HDT with graph information.

The second versioning approach is the change based approach. While this approach can work well if the fourth component is used for versioning information, it does not if it is used otherwise (e.g. for the origin of the triple). If used as the origin, the different graphs will most likely be greatly diverse. Because of that, the long list of additions and deletions in each version would again stand against the desired compactness of HDT. The IC and CB approaches in HDT have been evaluated by Fernández et al. [48].

The last versioning approach is the timestamp based approach. Adding a list of graphs to each triple, which indicates in which graphs the respective triple is present seems like a valid approach. No duplication of any of HDT's sections is necessary and thus the resulting representation would still be compact. While querying for all triples in a given graph might not be as easy as with the independent copies approach, this drawback can be mitigated by using an appropriate data structure for the list, which can be searched efficiently.

This section introduces an extended version of HDT, named HDT Quads (HDTQ), which supports graph information on the basis of the timestamp based versioning approach for RDF graphs. As such, HDTQ still supports triples and keeps HDT's compact form of storing them. Additionally, each triple's graph information is being represented in HDTQ. Two approaches of how this additional information can be handled are described in the subsequent section 5.1. After that, operations on HDTQ are discussed.

5.1 Graph Annotation

Two possibilities on how to store graph information in HDTQ based on [31] are presented in this section, namely *Annotated Triples* and *Annotated Graphs*. Both versions can represent any RDF dataset containing any number of named graphs.

Also, both versions are based on the following key concepts, further developed below:

- RDF triples are ordered in HDT, hence we can implicitly assign a

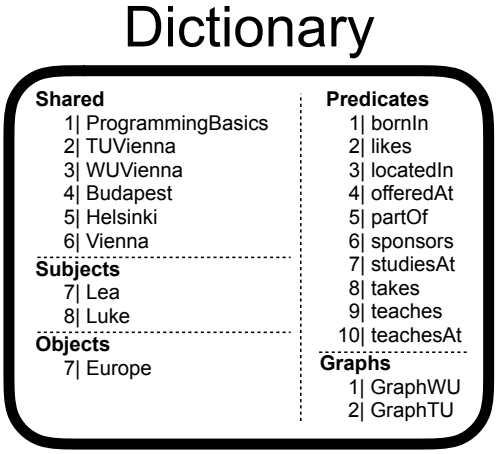


Figure 14: An HDT dictionary with graph information.

sequential number $1..N$ to each of the triples, where N is the total number of triples.

- We can use a dictionary to store the different graphs, and assign an integer $\{1..G\}$ to each of them, where G is the different number of graphs in the dataset.
- The membership of a triple t (where $t \in \{1..N\}$) to a graph g (where $g \in \{1..G\}$) can be modeled with a boolean function $graph(t, g) = \{0, 1\}$, where 1 denotes that t appears in g, or 0 otherwise.
- Annotated Triples and Annotated Graphs use a different organization of bitmaps to implement the graph function. In both cases, they use $N * G$ bits.

Efficient querying using quad patterns (which are combinations of a subject, a predicate, an object and a graph, where one or more of the components may be variable) can be performed with both versions.

However, depending on the specific structure of the RDF graph, the query to be executed and the particular implementation of HDTQ, one of the two possibilities might be more favorable than the other.

For both approaches, Annotated Triples and Annotated Graphs, HDT's dictionary is extended with a fifth component, which holds the names of the graphs. The dictionary compacts HDT by replacing long IRIs with IDs. To be specific, the IDs, as assigned by the subject, predicate, object and shared (subject + object) sections of the dictionary are used for replacing the

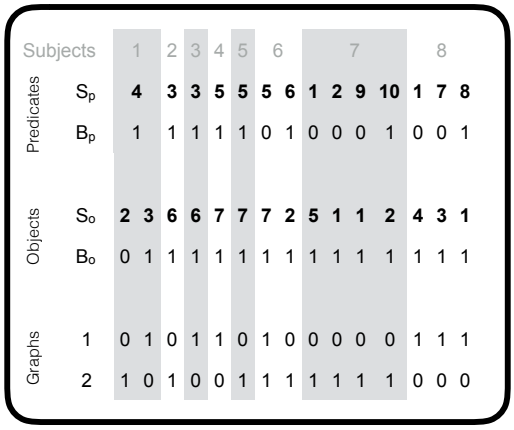


Figure 15: Bitmap Triples for Graph_B with graph information.

subject, predicate and object strings of the triples, which results in compact triples of IDs, see Section 4.2.

In HDTQ, the graph names are stored in the dictionary for two reasons. First, each graph name is only stored once. So, even if some hundreds or thousands of triples have the same graph component, the graph's name is stored only once in the dictionary. This also supports HDT's compact features. Second, by storing the graph names in the dictionary, they are assigned a specific ID. This ID is needed for both, Annotated Triples and Annotated Graphs.

The dictionary for Graph_B can be seen in Figure 14. When compared to the dictionary shown in Figure 12 (which is the dictionary for Graph_A, which is, besides the graph information, equal to Graph_B) one can notice the additional section "Graphs". As Graph_B consists of two named graphs, the graph names are assigned the IDs 1 and 2.

In HDT the triples have a specific order. This can be seen in all of the three encodings in Figure 13, where "1 - 4 - 2" is the first triple, "1 - 4 - 3" is the second and so on. By making use of this specific order and the order induced by storing the graph names in the dictionary, one can create a matrix which includes (for every *triple* - *graph* combination) the information whether a specific triple appears in a specific graph. For Graph_B this can be seen in Figure 15. In the matrix, a 1 indicates that the respective triple appears in the respective graph whereas a 0 indicates that the respective triple does not appear in the respective graph.

To create the aforementioned matrix, two approaches are possible, namely Annotated Triples and Annotated Graphs. For the former, one bitmap is "attached" to each triple. The latter uses one bitmap per graph. In either

case, the bitmap implementation used for the bitmap triples representation in HDT (see Section 4.3) can be reused for the additional bitmaps in HDTQ.

For the rest of the paper HDTQ always refers to both approaches, HDT-AG and HDT-AT. If HDT-AG or HDT-AT is mentioned, it is specifically referred to the respective approach.

5.1.1 Annotated Triples

Using the Annotated Triples approach, a bitmap is assigned to each triple, containing the information in which graphs this particular triple is present. Thus, a graph containing N triples in G different graphs has N bitmaps each of size G. The i^{th} position in the bitmap of triple t marks that t is (marked with a 1) or is not (marked with a 0) present in the i^{th} graph.

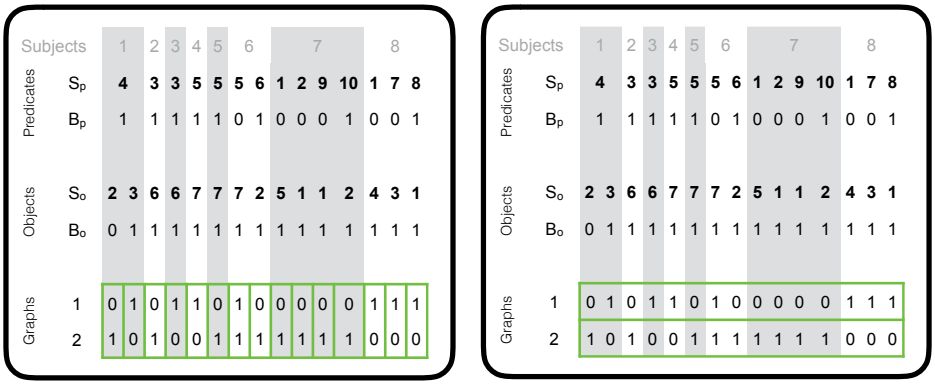
For Graph_B that means that 15 bitmaps each of size 2 must be created. This can be seen in Figure 16a. In this example, the bitmap for the first triple holds $\{0, 1\}$, the second triple $\{1, 0\}$ and so on. This means that the first triple appears in the first graph, which is GraphWU, but does not appear in the second graph, GraphTU. For the second triple, the opposite is the case. Of course there can be triples that appear in multiple or all of the graphs, in the example this is the case for the 7^{th} triple "6 - 5 - 7". It appears in all graphs, so its bitmap contains only 1s.

Using Annotated Triples makes searches for patterns that have the graph component as a variable, like SPO?, very efficient, as only a single bitmap needs to be browsed. If, on the other hand, the graph is given, like in ???G, all of the bitmaps need to be browsed. Further information on the pattern resolution algorithms can be found in Section 5.2 and an evaluation on the pattern resolution speed compared to other systems can be found in Section 7.

5.1.2 Annotated Graphs

Annotated Graphs is to some extend the inverse of the Annotated Triples approach. The same matrix is constructed, but the bitmaps are now "attached" to the graphs instead of to the triples. That means, using the Annotated Graphs approach, a bitmap is assigned to each graph, containing the information which triples are present in the particular graph. Thus, a graph containing N_t triples in N_g graphs has N_g bitmaps each of size N_t. The i^{th} position in the bitmap of graph g marks that the i^{th} triple is (marked with a 1) or is not (marked with a 0) present in g.

For Graph_B that means that 2 bitmaps each of size 15 must be created. This can be seen in Figure 16b. In this example, the bitmap for the first graph, GraphWU, holds $\{0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1\}$ and the bitmap



(a) Annotated Triples (b) Annotated Graphs

Figure 16: Highlighted graph information bitmaps for Graph_B.

for the second graph, GraphTU, holds $\{1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0\}$. This means that GraphWU contains the 2^{nd}, 4^{th}, 5^{th}, 7^{th}, 13^{th}, 14^{th} and 15^{th} triple. On the other hand, GraphTU contains the 1^{st}, 3^{rd} and 6^{th} to 12^{th} triple. Like in the Annotated Triples approach, triples can also appear in multiple graphs. This is the case for the 7^{th} triple, as both bitmaps hold a 1 in their 7^{th} position.

Compared to Annotated Triples, Annotated Graphs makes search patterns that have the graph component given, like ???G, very efficient, as only a single bitmap (the bitmap of the given graph G) needs to be browsed. On the other hand, patterns that have the graph component as a variable, like SPO? are not so efficient, because all bitmaps need to be browsed to answer the query. Further information on the pattern resolution algorithms can be found in Section 5.2 and an evaluation on the pattern resolution speed compared to other systems can be found in Section 7.

5.2 Operations

While HDT can handle 8 different search patterns (see Section 4.3), the added component, graph, in HDTQ increases the number of search patterns to 16. By adding the fourth component to each of the 8 patterns, once as a variable, once as bounded term, one comes to the 16 quad patterns shown in Figure 17.

For the 8 quad patterns in which the graph is not given, the same algorithm can be applied in HDTQ to resolve the search pattern. The algorithm and its supplement algorithms are described in Section 5.2.1.

For the 8 quad patterns in which the graph is given, different algorithms

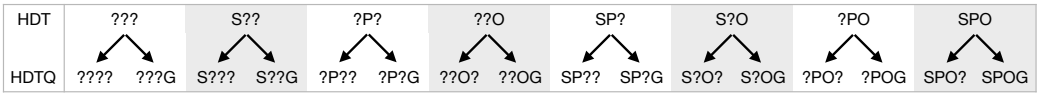


Figure 17: The 16 patterns of HDTQ.

need to be applied in HDTQ to resolve the search pattern. This is due to the specific structure of HDT, in which subjects appear consecutively, but predicates and objects do not. The algorithms and their supplement algorithms are described in Section 5.2.2.

Important operations used in these algorithms are:

- **getBitmap(position)**. The getBitmap operation returns the bitmap identified by the supplied integer parameter from all bitmaps in the GraphInformation object. The GraphInformation object holds all bitmaps that together compose the matrix of triples and graphs as described in the previous sections.
- **access(position)**. The access operation returns the value of a given position in a bitmap (either "0" or "1").
- **getNext1(position)**. The getNext1 operation returns the next "1" in a bitmap, starting with a given position (see also the *select* operation in Section 4.3).

5.2.1 Quad Pattern Queries with Unbounded Graph

The algorithm to resolve quad patterns in which the graph component is not given, i.e. ????, S???, ?P??, ??O?, SP??, S?O?, ?PO? and SPO?, makes use of another algorithm, *selectNext1Triple*. *selectNext1Triple* retrieves for a given triple in a given graph the next graph in which this triple appears. This supporting algorithm is different for HDT-AT and HDT-AG, however returns the same result. Below, these algorithms are discussed briefly, before the "main" algorithm is introduced.

Algorithm 1: selectNext1Triple (HDT-AT) retrieves the position of the next graph that the given triple appears in for HDT-AT.

Input: GraphInformation G, int posTriple, int graph
Output: The position of the next graph
1 $bitmap \leftarrow G.getBitmap(posTriple)$
2 **return** $bitmap.getNext1(graph)$

Both supporting algorithms are supplied with the graph information matrix and a position in the matrix. The position determines a triple and a graph. The algorithm then returns the next graph in which the given triple appears, or null if does not appear in any further graphs.

Algorithm 1 shows this algorithm for HDT-AT. The algorithm is very simple, first the bitmap corresponding to the given triple is retrieved from the GraphInformation object. Then, within this bitmap, the location of the next 1 starting with the provided integer position is retrieved and returned.

Algorithm 2: selectNext1Triple (HDT-AG) retrieves the position of the next graph that the given triple appears in for HDT-AG.

Input: GraphInformation G, int posTriple, int graph
Output: The position of the next graph

```
1 bitmap ← G.getBitmap(graph)
2 while bitmap.access(posTriple) ≠ 1 do
3 |   graph ← graph + 1
4 |   bitmap ← G.getBitmap(graph)
5 return graph
```

Algorithm 3: SearchQuads retrieves all quads for ????, S???, ?P??, ??O?, SP??, S?O?, ?PO?, SPO?.

Input: BitmapTriples B, GraphInformation G, pattern p
Output: The quads matching the given pattern

```
1  result ← ()
2  (triple, posTriple) ← B.getNextSolution(p, 0)
3  graph ← 0
4  while posTriple ≠ null do
5  |   graph ← G.selectNext1Triple(posTriple, graph + 1)
6  |   if graph ≠ null then
7  |   |   result.append(quad(triple, graph))
8  |   else
9  |   |   (triple, posTriple) ← B.getNextSolution(p, posTriple)
10 |   |   graph ← 0
11 return result
```

Algorithm 2 shows the algorithm for HDT-AG. There now is no single bitmap associated with a triple, but instead one bitmap per graph. For this reason, one bitmap after the other (starting with the provided one) must be

accessed. In these bitmaps, it is checked whether the given triple appears in the graph or not. The first graph that contains the triple is then returned, if there is no graph that contains the triple, null is returned.

Algorithm 3 shows the main algorithm, *searchQuads*, to resolve quad patterns where the graph component is not given. The algorithm first uses the *getNextSolution* method from HDT to find a triple solution for the given pattern (where the graph component is ignored). Then, the respective *selectNext1Triple* algorithm described above is used to find one graph after another in which the found triple appears. Each of the so found graphs is, together with the triple, one solution. If the triple does not appear in any further graphs, the next triple solution is retrieved. If no further triples can be found, the algorithm ends and all found quads are returned.

5.2.2 Quad Pattern Queries with Bounded Graph

To resolve the eight quad patterns in which the graph component is given (???G, S??G, ?P?G, ??OG, SP?G, S?OG, ?POG and SPOG), three main algorithms are needed.

The first one, *searchQuadsG*, resolves ???G, S??G, SP?G, S?OG, and SPOG and makes use of the subject index of HDT. The second one, *SearchQuadsPG*, resolves ?P?G and makes use of HDT's predicate index. The third one, *SearchQuadsOG*, resolves ??OG and ?POG and makes use of the object index.

The first two again make use of a supporting algorithm, *selectNext1Graph*, which is similar to the supporting algorithm in the previous section, Section 5.2.1. It is again a slightly different algorithm for HDT-AT and HDT-AG, but both are returning the same result. Again, both algorithms are supplied with the graph information matrix and a position in the matrix, where the position determines a triple and a graph. The algorithm then returns the position of the next triple that appears in the given graph.

Algorithm 4 shows *selectNext1Graph* for HDT-AT. It retrieves one bitmap after the other (where each bitmap belongs to a triple) until it finds a bitmap that contains a 1 for the given graph. Then the position of the triple that belongs to the graph is returned.

Algorithm 5 shows *selectNext1Graph* for HDT-AG. It retrieves the bitmap of the graph in question from the GraphInformation object. Then, the *getNext1* operation is applied to find the position of the next 1 in this bitmap. Finally, this position is returned.

Algorithm 6 resolves the patterns ???G, S??G, SP?G, S?OG and SPOG. To do so, first, in line 2, the graph ID is retrieved from the GraphInformation object. As the graph is given in the input quad pattern, this ID is part of all

Algorithm 4: selectNext1Graph (HDT-AT) retrieves the position of the next triple that appears in the given graph for HDT-AT.

```
Input: GraphInformation G, int posTriple, int graph
Output: The position of the next triple in the given graph
1 bitmap ← G.getBitmap(posTriple)
2 while bitmap.access(posGraph) ≠ 1 do
3 |   posTriple ← posTriple + 1
4 |   bitmap ← G.getBitmap(posTriple)
5 return posTriple
```

Algorithm 5: selectNext1Graph (HDT-AG) retrieves the position of the next triple that appears in the given graph for HDT-AG.

```
Input: GraphInformation G, int posTriple, int graph
Output: The position of the next triple in the given graph
1 bitmap ← G.getBitmap(graph)
2 return bitmap.getNext1(posTriple)
```

results of the algorithm.

Then, the *getRange* operation of HDT is applied to find the range of triple positions in which solution candidates appear, which is then stored as the *min* and *max* positions. For ???G this is simply the full range of all triples (as subject, predicate and object are all variables). For the other patterns this limits the further search to a specific range of triples, i.e. the range of all triples that have the given subject. For SP?G, it is further limited to those triples that have the given predicate as well. For SPOG it is limited to a single triple.

Then, starting with the *min* position, the *selectNext1Graph* algorithm is used to find the next triple that appears in the given graph and its position is stored in *posTriple*. Next, HDT's *getSolution* operation is used to extract the triple for the found position. As results are consecutive for the patterns in question (except for S?OG), it is not necessary to check again whether the found triple is actually a solution, as long as *posTriple* is less or equal to the upper range limit, it must be a solution.

The resulting quad (the found triple together with the graph) is added to the list of solutions. After that, the *selectNext1Graph* algorithm is again used to jump to the next solution. This is repeated until no more solutions within the $\{min, max\}$ range can be found. The list of solutions (quads) is then returned.

For the special case of S?OG the returned results are not necessarily correct. As the objects do not appear consecutively in the bitmaps of HDT, the range and therefore solutions can only be found for S??G. Most likely the algorithm will return too many results for this pattern (probably some will have the wrong object). Therefore, the found solutions are then filtered using HDT's *Sequential Search Algorithm* which filters those results that really match the given pattern to get the correct results.

Algorithm 6: searchQuadsG retrieves all quads for ???G, S??G, SP?G, S?OG, SPOG.

Input: BitmapTriples B, GraphInformation G, pattern p
Output: The quads matching the given pattern

```
1 result ← ()
2 graph ← G.getGraph(p)
3 (min, max) ← B.getRange(p)
4 posTriple ← G.selectNext1Graph(min, graph)
5 while posTriple ≠ null and posTriple ≤ max do
6 |   triple ← B.getSolution(p, posTriple)
7 |   result.append(quad(triple, graph))
8 |   posTriple ← G.selectNext1Graph(posTriple + 1, graph)
9 return result
```

Algorithm 7 resolves ?P?G queries. As the subject is not provided, but the predicate is, the range of solution candidates cannot be determined as easily as in Algorithm 6 (because results do not appear consecutively). Thus, to find the triples that have the given predicate, HDT's predicate index is used.

First, in line 2, the graph ID is retrieved from the GraphInformation object. As the graph is given in the input quad pattern, this ID is part of all results of the algorithm. Then, HDT's *getPredicateOccurences* operation is used to determine the number of occurrences of the given predicate in the predicate adjacency list.

Now, in line 6, the triple position range for the first predicate occurrence is retrieved from HDT (there is a range for each predicate occurrence because the predicate can be associated with multiple objects). As the search is limited to a specific graph only, the triples in the range are not necessarily solutions, but rather solution candidates. Next, the *selectNext1Graph* algorithm is used to find the position of the first triple within the range that appears in the graph.

The triple components are retrieved from HDT using the found triple

position. Together with the graph, the quad is added to the lists of results. Now the *selectNext1Graph* algorithm is used to find further triples in the range that appear in the graph. Once there are no further triples in the range that appear in the graph, the algorithm jumps to the next predicate occurrence and again starts to determine the range of candidate solutions. If there are no further predicate occurrences, the algorithm ends and the list of solutions is returned.

Algorithm 7: SearchQuadsPG retrieves all quads for ?P?G.

Input: BitmapTriples B, GraphInformation G, pattern p
Output: The quads matching the given pattern

```
1  result ← ()
2  graph ← G.getGraph(p)
3  occurences ← B.getPredicateOccurences(p)
4  occurence ← 1
5  while occurence ≤ occurences do
6      (min, max) ← B.getRange(p, occurence)
7      posTriple ← G.selectNext1Graph(min, graph)
8      while posTriple ≠ null and posTriple ≤ max do
9          triple ← B.getSolution(p, posTriple)
10         result.append(quad(triple, graph))
11         posTriple ← G.selectNext1Graph(posTriple + 1, graph)
12     occurence ← occurence + 1
13 return result
```

Algorithm 8 resolves ?POG and ??OG queries. As the subject is not provided, but the object is, the range of solution candidates can again not be easily determined (because results do not appear consecutively). Thus, to find the solution, the object index of HDT is used.

First, in line 2, the graph ID is retrieved from the GraphInformation object. As the graph is given in the input quad pattern, this ID is part of all results of the algorithm. Then, HDT's *getIndexRange* operation is used to determine the range in the object index for the given object.

Then, the first triple position is retrieved from the index, using HDT's *getNextTriplePosition* operation. The position returned by this function points to a triple that fulfills the object (and if present also the predicate) requirement. However, it is still a candidate solution as it must be checked whether the triple appears in the given graph. For this, the *access* operation is applied to the triple position.

If the triple at the triple position appears in the given graph, the triple components are retrieved using HDT's *getSolution* operation. Then the triple together with its graph is added to the list of solutions. If the triple does not appear in the given graph, this step is omitted.

In any case, as a next step, the next position is retrieved from the index and again the *getNextTriplePosition* operation is applied to retrieve the next triple position. These steps are repeated until the end of the index range is reached. The list of results is then returned.

Because the results are not consecutive and the index must be used to efficiently find the solution candidates, no jumping using the *selectNext1Graph* algorithm is possible for the ?POG and ??OG patterns.

Algorithm 8: SearchQuadsOG retrieves all quads for ?POG and ??OG.

Input: BitmapTriples B, GraphInformation G, pattern p
Output: The quads matching the given pattern

```
1  result ← ()
2  graph ← G.getGraph(p)
3  (min, max) ← B.getIndexRange(p)
4  posIndex ← min
5  while posIndex ≤ max do
6  |  posTriple ← B.getNextTriplePosition(p, posIndex)
7  |  if G.access(posTriple, graph) = 1 then
8  |  |  triple ← B.getSolution(p, posTriple)
9  |  |  result.append(quad(triple, graph))
10 |  posIndex ← posIndex + 1
11 return result
```

In the following, we present a practical implementation of these algorithms (Section 6), which are then evaluated in Section 7.6.

6 Practical Implementation of HDTQ

In this section a java prototype implementation of HDTQ[2] is introduced. It is extending the current HDT-java library[3]. Especially the differences to the original HDT implementation are highlighted.

[2]https://github.com/JulianRei/hdtq-java
[3]https://github.com/rdfhdt/hdt-java/

To be capable of handling RDF graphs that contain named graphs (i.e. RDF graphs that contain quads), HDT was extended to also support the N-Quads and TriG syntax (see Section 2.2.2).

HDT's four-section dictionary was extended by a fifth section to store graph names. By this, long graph IRIs are replaced by short identifiers, as described in Section 5.1. In HDT, when first parsing the triples of the input graph, the subjects, predicates and objects are added to the dictionary. Also, the triples are stored in a collection. In HDTQ, also the graph component is added to the dictionary and instead of triples, quads are added to the collection. Like the other dictionary sections, also the graph section is sorted, so a better compression can be reached. Furthermore, triples are also sorted by their IDs.

In a next step, HDT removes duplicates from the RDF graph data (two triples are duplicates, if their subject, predicate and object components are equal). As the triples are sorted by their IDs, duplicates must be adjacent to one another. In classic HDT such triples are simply removed in this step. HDTQ merges the found duplicates to one triple and adds the graph of each duplicate into a list (values in the list are unique, so duplicates where even the graph is equal are removed). Once all duplicates of a triple are found, the list contains all graphs in which the triple appears. The triple is then processed further as in classic HDT. The list of graphs is handed over to the graph information object, which uses the graph- or triple-annotator (depending on which approach was chosen in the configuration) object to add this information to the graph information matrix. The matrix itself is implemented as a collection of bitmaps, each being attached either to a triple or a graph. Navarros approach was chosen for the implementation of the bitmaps [104].

Lastly the header section is extended by the number of graphs and the annotation approach used (annotated graphs or annotated triples). To define whether the annotated graphs or annotated triples approach should be used, the configuration was extended by the optional variable `graph.type`. Possible values are `<http://purl.org/HDT/hdt#AG>` for annotated graphs and `<http://purl.org/HDT/hdt#AT>` for annotated triples. If no value is specified, annotated triples is chosen.

The implementation described above is optimized for speed and compared to classic HDT no additional iteration over the triples is needed.

The algorithms described in Section 5.2 are implemented using the iterator pattern. To reduce code clones, the implementation of classic HDT was reused, where possible.

To test the HDTQ implementation, JUnit [83] tests were written. The tasks take an arbitrary RDF graph (that has named graphs) as an input.

The graph is imported into a Jena instance, into HDTQ using the annotated graphs and into HDTQ using the annotated triple approach. Then, all subjects, predicates, objects and graphs are extracted from the input graph and for each of the 16 patterns each possible query (using the extracted components) is executed against the three systems. Only if all three systems report exactly the same results, the test is accepted.

7 Evaluation

In this section, the implementation of HDTQ, as described in the previous section, is compared to Jena and Virtuoso regarding creation time, space requirement and querying speed for various datasets.

7.1 Setup

All tests described in this section where run on a server running "Ubuntu 14.04.5 LTS". The machine has 16 sockets with 1 core per socket and 1 thread per core, which makes a total of 16 CPUs. Each of these processors is a "Intel Xeon E312xx (Sandy Bridge)" and has a frequence of 2.6GHz. What is more, the machine has 177GB of RAM and 150GB Swap space.

7.2 Systems

The performance of five different systems / configurations were tested and compared. These systems include:

- **HDT-AT.** HDT-AT is the Java implementation of HDTQ using annotated triples, as described in Section 5.1.1. For each of the datasets, a separate HDT-AT instance was created.
- **HDT-AG.** HDT-AG is the Java implementation of HDTQ using annotated graphs, as described in Section 5.1.2. For each of the datasets, a separate HDT-AG instance was created.
- **Jena.** Jena refers to an Apache Jena TDB store. Data is imported into the store and queried via SPARQL using the ARQ query engine version 3.0.0. To create the TDB store, the tdbloader2 loader and index builder [51] was used. For each of the datasets, a separate Jena TDB instance was created.
- **Virtuoso.** Virtuoso refers to a Virtuoso database, version 7.2.4.2.3217. Note however, that the datasets where not split into multiple files, as

suggested by [136], but instead imported as one file per dataset (to have equal conditions for all datasets). For each of the datasets, a separate Virtuoso instance was created. What is more, the virtuoso.ini was configured so that Virtuoso can use 180 gigabytes of RAM.

- **Virtuoso+.** Virtuoso+ refers to a Virtuoso database, version 7.2.4.2.3217. Virtuoso+ is almost identical to Virtuoso, but Virtuoso+ is created by copying the Virtuoso database (with all data imported) and after that, creating an additional index. The Virtuoso performance tuning guide [135] suggests to create an additional index (GPOS), that is, according to the guide, sometimes helpful if the subject is not given, but the predicate and graph are given. To see if this additional index has an influence on the performance, it was created for Virtuoso+ after the data was imported. For each of the datasets, a separate Virtuoso+ instance was created.

7.3 Datasets

		Subjects	Predicates	Objects	Graphs	Triples	Quads
BEAR	A	74,908,887	41,209	64,215,355	58	378,476,570	2,071,287,964
	B day	100	1,725	69,650	89	82,401	3,460,896
	B hour	100	1,744	148,866	1,299	167,281	51,632,164
LUBM	500	10,847,183	17	8,072,358	1...9,998*	66,731,200	66,731,200+**
	1000	21,673,510	17	16,126,103	1...7,000†	133,319,232	133,319,232+††
LDBC		668,711	16	2,743,645	190,961	5,000,197	5,000,197
Liddi		392,344	23	981,928	392,340	1,952,822	2,051,959

* 1, 10, 20,..., 100, 1,000, 2,000,..., 9,000, 9,998
† 1, 10, 20,..., 100, 1,000, 2,000,..., 7,000
** 66,731,200 ... 68,823,803
†† 133,319,232 ... 136,725,784

Table 1: Attributes of the datasets.

To test the performance of the systems described in Section 7.2, several datasets where used. The datasets differ in various perspectives, including their origin (whether it is real world data or generated), the number of subjects, predicates, objects and graphs as well as the total number of triples and quads. An overview of these figures can be seen in Table 1. Note that the figures in the table show distinct values, e.g. the number of distinct subjects in the dataset.

The table shows the number of triples and the number of quads. If a triple appears in multiple (e.g. 5) graphs, it counts only one time in the triple column, but multiple times (e.g. 5 times) in the quads column.

To have equal starting conditions, all data was (if not already provided in this format) transformed to the N-Quads (see Section 2.2.2) format, as

this is the rawest, uncompressed format for quads. The datasets and their specifics are discussed below.

- **BEAR-A.** The Dynamic Linked Data Observatory [140] monitors more than 650 different domains across time and does weekly crawl of these domains. Fernández et al. built an RDF archive on top of 58 of such weekly snapshots [49]. Each of the snapshots is considered to be a graph, resulting in the BEAR-A dataset which consists of 58 graphs.

  The different versions grow slowly, except for the last versions which mostly contribute to the growth of the dataset. The size of the last version is more than double the initial size (the size of the first version). What is also noticeable is the very small static core of 3.5 million entries [49].

  As can be seen, BEAR-A is by far the biggest dataset under review with more than 2 billion quads. As there is a huge gap between the number of triples and quads, that means that a lot of triples appear in multiple graphs in this dataset.

- **BEAR-B day and BEAR-B hour.** DBPedia Live [75] records all updates to Wikipedia articles and updates the respective DBpedia Live resource descriptions. The BEAR-B dataset contains resource descriptions of the 100 most volatile resources and their updates in the time range of August to October 2015. The most volatile record in the dataset changed 1,305 times, the least volatile in the dataset 263 times over this timespan [49].

  For every change in an Wikipedia article, DBPedia Live creates a new version. In the respective timeframe the looked-at dataset changed 21,046 times. Note that changes also include added or deleted triples, overall the dataset grew by 31%. BEAR-B day includes the aggregated data on a daily level. It therefore contains 89 versions. BEAR-B hour is the same on a hourly level and contains 1,299 versions [49].

  For the course of this thesis, each of the versions in BEAR-B day and BEAR-B hour is seen as a graph, therefore BEAR-B day consists of 89 graphs and BEAR-B hour of 1,299 graphs. As can be seen in Table 1 BEAR-B day and BEAR-B hour both include 100 distinct subjects. These arise from the extraction of the 100 most frequent changing articles. Both datasets include around 1,700 predicates, which is much higher than the number of predicates of the rest of the datasets in our evaluation. On the other hand, the number of objects is comparatively low. One notes that the number of predicates and objects is lower for

BEAR-B day than those of BEAR-B hour. This is due to the fact that some changes are reverted so quickly that they are reflected in the more frequent version BEAR-B hour but do not appear in BEAR-B day.

Like in BEAR-A, BEAR-B day and BEAR-B hour both show a huge difference between the number of triples and the number of quads. That means that a lot of triples appear in multiple graphs.

- **LUBM500 and LUBM1000.** The Lehigh University Benchmark for OWL (LUBM) is a widely used benchmark to compare OWL engines. As part of the benchmark, Guo et al. developed the UBA (Univ-Bench Artificial data generator)[4]. Random and repeatable data can be generated using the tool. The minimum unit of data generation is a university and for each university a set of OWL files, which describe the departments of this very university are generated. To make the data more realistic, some restrictions are applied, like a minimum and maximum number of departments per university, a meaningful ratio between students and faculties and that students take at least one, but not too many courses [64].

  The tool allows to specify the number of generated universities, also a seed can be provided to the random number generator. The universities are named University⟨ID⟩ with an zero-based index as ID. Thus, the first university is named "University0". The starting index can also be defined by the user. The generated data is in OWL Lite sublanguage [64].

  To create the LUBM datasets, the following steps were taken. First, the data generator UBA1.7 using 500 (for LUBM500) respectively 1000 (for LUBM1000) universities, a starting index of 0 and a seed of 0 was used to generate the data in RDF/XML format. With this input parameters, the generator produces 9,998 (for 500 universities) and 19,992 (for 1,000 universities) files. Second, Apache Any23 [50] was used to convert each of the files to the N-triples format. Third, as the UBA1.7 generator is not capable of producing data in multiple named graphs, the files were assigned to a number of graphs. To get n graphs, when having j files, each file f_i was assigned to a graph g by the following formula: $i \mod n = g$ where $i = \{0, \ldots, j-1\}$. All files assigned to the same graph were merged into a single file. Fourth, each of the triples in such a merged file was now extended with a fourth component, namely "http://www.example.org/graph/⟨ID⟩", where ID is an incrementing

[4] http://swat.cse.lehigh.edu/projects/lubm/

unique number starting with 1. Finally, all files (that now already contain N-quads) were merged into a single file.

To get not only one dataset for LUBM500 and one for LUBM1000, the generated data (the N-triples data, resulting from the second step above), was assigned to a variety of different numbers of graphs. The upper possible limit of number of graphs is the number of files generated by the UBA generator. For LUBM1000 the number of graphs where limited to 7,000, as the server used in the evaluation did not have enough RAM to generate HDTQ with a larger number of graphs (Section 8.2 discusses this limitation of the current prototype). For the smaller dataset LUBM500, although having a higher number of graphs (9,998), RAM was sufficient.

The chosen number of graphs for LUBM500 are: 1, 10, 20,. . . , 100, 1,000, 2,000,. . . , 9,000, 9,998.
The chosen number of graphs for LUBM1000 are: 1, 10, 20,. . . , 100, 1,000, 2,000,. . . , 7,000.

All generated LUBM data only have 17 predicates, which is among the lowest number in the compared datasets. However, there are a considerable amount of about 11 million (for LUBM500) and about 21 million (for LUBM1000) distinct subjects in the data and with 8 and 16 million distinct objects also a high number of objects respectively.

When extracting only a single graph, the number of triples and quads are obviously the same, as no triple can appear in more than one graph. With an increasing number of graphs, the number of quads slowly increase for LUBM500 and LUBM1000, while the number of triples of course stay the same. Given our approach to assign the graph to each triple (based on the file it appears in), most triples only appear in one graph, with very few triples occurring in multiple graphs. Nonetheless, note that this assignation is the approach taken by other state of the art approaches like LOD Laundromat [5] or Slavov et al. [127]. However, as there is only a small number of such multiple appearing triples, all LUBM500 and LUBM1000 datasets are considered to be equal (except for the different number of graphs) for the rest of the thesis.

- **LDBC.** The Semantic Publishing Benchmark (SPB) is a benchmark offered by the Linked Data Benchmark Council (LDBC). It is inspired by the Media/Publishing industry and it considers a media or a publishing organization that deals with large colume of streaming content

[5] http://lodlaundromat.org/

(news, articles or media assets) as its application scenario. Metadata, which describes the content and links it to reference knowledge, enriches the content [90].

The benchmark assumes that (mostly static) reference knowledge, but also metadata that grows constantly is stored in an RDF database. Main interactions are updates, that add new metadata or alter it or queries to retrieve content [90].

The SPB offers a data generator to generate RDF data as described above. For this thesis, the SPB version 2.0 generator, which is freely available on GitHub [91], was used. The named graphs generated by the generator where concatenated to receive the LDBC dataset.

Looking at Table 1 one can see that LDBC has only 16 distinct predicates, which is the smallest number in the datasets used. With its about 5 million triples, LDBC is the biggest of the smaller datasets used.

As the number of triples and the number of quads are equal for LDBC, that means that each triple appears only in one graph.

- **Liddi.** Drug-drug interactions (DDI) occur when the effect of one drug is altered by another drug, which leads to unpredictable effects. The LInked Drug-Drug Interactions (LIDDI) dataset consolidates multiple data collections of DDI predications from public databases and reports and biomedical literature into one [14]. The Liddi dataset is based on nanopublications [61] that closely link data to their provenance and meta-data. As such, a nanopublication consists of:

  - An assertion graph with triples expressing an atomic statement (about drug-drug interactions in the case of Liddi).
  - A provenance graph that reports how this assertion came about (for example, where it was extracted from or what mechanism was used to derive it).
  - A publication information graph that provides meta-data for the publication (like creators and a timestamp).

  The Liddi dataset contains 98,085 nanopublications, each of which is stored as 4 graphs (an assertion graph, a provenance graph, a publication graph and a head graph that "glues" these graphs together). This results in a total of 392,340 graphs, which makes Liddi by far the dataset with the greatest number of graphs in this work.

As the gap between the number of triples and quads is quite small for Liddi, most triples only appear in a single graph.

7.4 Space Requirement

As one of the desired features for HDTQ is being compact, in this section the compactness of HDTQ is compared to the other systems listed in Section 7.2. Table 2 lists the space requirements for the different datasets for the respective uncompressed RDF graph in N-Quads notation (column "Size"), the gzipped N-Quads file (column "gzip") and the 5 systems under review.

The size of the uncompressed RDF graph is denoted in gigabytes, while all other sizes are expressed as the ratio between the size for the respective system and the uncompressed size.

For HDT-AG and HDT-AT the size contains the size of the respective HDT file itself together with the size of its index file. For Jena, the size comprises the sizes of all files in the Jena TDB store, including all `.dat`, `.idn`, `.jrnl`, `.opt` and `.lock` files. For Virtuoso and Virtuoso+ the size comprises the size of all files for the respective Virtuoso instance, containing `.db`, `.ini`, `.log`, `.pxa`, `.db` and `.trx` files.

Looking at Table 2 one quickly sees that the compression with gzip outperformes HDTQ, Jena and Virtuoso for all datasets except for the BEAR datasets. This is most likely due to the additional indexes that are created for the other systems, which are not created for the purely compressed gzip format. While the compression of gzip is very handy especially when transferring the data, the downside of gzip is that the compressed data cannot be easily searched through like in the HDTQ format or using by Jena / Virtuoso. Because of this inability, for the rest of this section gzip is excluded in the comparison.

BEAR-A. BEAR-A is a considerably big dataset with more than 2 billion quads. It has been shown that RDF graphs at big scale are highly compressible [45]. These results can be confirmed as HDT-AG compresses

		Size (GB)	gzip	HDT-AG	HDT-AT	Jena	Virtuoso	Virtuoso+
BEAR	A	396.85	5.82%	**2.33**%	2.75%	96.84%	NA	NA
	B day	0.64	4.83%	**0.65**%	0.71%	97.73%	13.69%	33.75%
	B hour	9.66	4.83%	0.33%	**0.25**%	96.39%	4.35%	25.62%
LUBM	500 (1 graph)	11.42	3.04%	**6.71**%	11%	118.76%	17.23%	21%
	500 (9998 graphs)	11.61	3.02%	675.58%	345.92%	120.1%	**17.46**%	27.51%
	1000 (1 graph)	22.84	3.04%	**6.9**%	11.18%	118.67%	16.23%	19.98%
	1000 (7000 graphs)	23.21	3.02%	474.85%	236.44%	119.47%	**16.38**%	22.65%
LDBC		0.92	9.7%	12111.47%	6081.25%	126.28%	**71.2**%	80.77%
Liddi		0.67	3.74%	13254.03%	6637.71%	78.06%	**49.88**%	53.36%

Table 2: Space requirements of different systems.

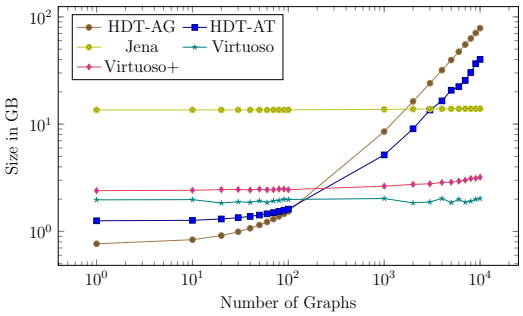


Figure 18: Space requirements for the LUBM500 graphs.

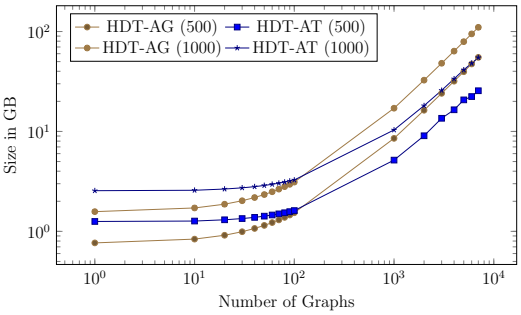


Figure 19: Space requirements for the LUBM500 and LUBM1000 graphs.

the BEAR-A dataset very well. By reducing the size to only 2.33% it even outperforms gzipping. Also, HDT-AT performs well, with 2.75% it is still much better than gzipping. Jena only marginally reduces the size and therefore does not fit very well for the BEAR-A dataset. Virtuoso, in its tested version, was not capable of importing the BEAR-A dataset at all. Thus, no statement about its compression features can be made. As Virtuoso+ is based on Virtuoso, it was not possible to import the data into this system either.

BEAR-B day and BEAR-B hour. BEAR-B day has a very low number of graphs, 89, which seems to fit very well for HDT-AG, as it compresses the data to only 0.65% of its original size. With 0.71% HDT-AT is also compressing this data very well. Jena does only marginally reduce the space required. While Virtuoso has a decent compression ratio, it is still by far outperformed by HDTQ (about a factor of 20 for BEAR-B day and about 15 for BEAR-B hour). The additional index created for Virtuoso+ also needs considerable space and more than doubles the size.

BEAR-B hour has considerable more graphs than BEAR-B day (1,299 vs. 89). All systems have a better ratio for this dataset than for BEAR-B day. HDT-AT improved it's compression to 0.25% and is therefore the system that compressed the data the most. Jena still does not perform very well and while Virtuoso improves it performance, it is still behind HDTQ.

LUBM500 and LUBM1000. Looking at the results for LUBM500 and LUBM1000 one notices that HDT-AG and HDT-AT both compresses very well for a low number of graphs (the numbers for 1 graph are shown in the table). However, for a bigger number of graphs the compression ratio deteriorates and the resulting HDTQ size exceeds the original data's size. As all LUBM500 and LUBM1000 datasets are equal except for the number of graphs, this effect is due to the number of graphs. For LUBM500 this can also be seen in Figure 18. Until approximately 100 graphs, HDTQ outper-

formes Jena and Virtuoso. With 1000 graphs, HDTQ is worse than Virtuoso and Virtuoso+, but still better than Jena. Starting with about 2000-3000 graphs, HDTQ becomes the worst of the compared systems. The graph also shows that Jena and Virtuoso have a very constant space requirement, almost independent of the number of graphs. HDTQ needs considerable more space with a higher number of graphs, which is a result of the used plain bitmap implementation. Note that testing with different compressed bitmaps is left for future work. Nonetheless, as we will show in Section 7.6, query performance of HDTQ is much better than the most compact Virtuoso approach and similar to Jena.

For LUBM1000 only HDT-AG and HDT-AT, but not Jena or Virtuoso were created. HDTQ's compression shows a very similar picture like for LUBM500. The trend can be seen in Figure 19. As for LUBM500, the needed space first grows slowly and increases a lot between 100 and 1000 graphs. Obviously HDTQ's size is bigger for LUBM1000 then for LUBM500, as the LUBM1000 is about double the size. Apart from this shift in the y-axes, the trend can be considered the same.

LDBC. The LDBC dataset has about 190,000 graphs and as expected from the observations outlined above, HDTQ performs poorly for such a dataset. The required space by far exceeds the original space. With its 5 million triples and 190,000 graphs, the graph information matrix has an uncompressed size of 950 billion bits, or about 110 gigabytes. Virtuoso / Virtuoso+ and Jena show far more promising results regarding compression, ranging from about 70% to 126% of the original size.

Liddi. The Liddi dataset with its especially many graphs (about 392,000) is the worst case for HDTQ. Because of the bad compression of the bitmaps of the graph information matrix, the result is not only not compressed, but instead multiple times bigger than the original dataset, ranging from 66 times (HDT-AT) to 132 times (HDT-AG) the size of the original data. While Jena can reduce the size of the dataset to about 78%, Virtuoso (Virtuoso+) can even reduce it to about 50% (53%) of the original size.

Regarding the needed space it can be said that HDTQ is compressing datasets with a low number of graphs very well. However, as the LUBM500 datasets illustrates well, with a higher number of graphs together with a high number of triples it scales badly. The reason for this is that $N * G$ (number of triples times number of graphs) bits are required for the bitmaps in HDTQ. The algorithms from Section 5.2 are still valid as soon as compression is used in these bitmaps, but the application is devoted to future work.

HDTQ positions itself as the perfect candidate to deal with versioned datasets such as the BEAR datasets, that is, hundreds of millions of triples with hundreds of versions/graphs, or thousand of triples with thousands of

versions/graphs. This can also be observed in Table 3, which compares space requirements of HDTQ for the BEAR datasets against storing the BEAR data with classic HDT using the IC and CB approaches (IC and CB numbers are taken from [49]). Both approaches of HDTQ (HDT-AG and HDT-AT) outperform classic HDT IC and CB. Even compared to the better performing approach, CB, HDTQ outperforms CB by a factor of about 3 for BEAR-A, about 1.7 for BEAR-B day and about 17 for BEAR-B hour.

		Size (GB)	gzip	HDT-AG	HDT-AT	HDT IC	HDT CB
BEAR	A	396.85	5.82%	**2.33**%	2.75%	12.1%	7.06%
	B day	0.64	4.83%	**0.65**%	0.71%	22.63%	1.06%
	B hour	9.66	4.83%	0.33%	**0.25**%	338.6%	5.32%

Table 3: Space requirements of versioning strategies.

7.5 Creation Time

The times to import the different datasets in the respective system can be seen in Table 4. All times are denoted in seconds. The shown times were measures from the beginning of the import, until all data is imported and all indexes are created. That means, after the listed number of seconds of processing, the system is ready to answer queries.

		HDT-AG	HDT-AT	Jena	Virtuoso	Virtuoso+
BEAR	A	**34,212**	42,876	93,918	NA	NA
	B day	21	**20**	86	**20**	33
	B hour	293	**264**	1,328	301	534
LUBM	500 (1 graph)	770	776	1,912	**337**	432
	500 (9998 graphs)	2,123	1,619	1,686	**338**	439
	1000 (1 graph)	2,218	2,364	4,459	**674**	863
	1000 (7000 graphs)	4,447	4,808	3,745	**679**	877
LDBC		791	581	115	**50**	70
Liddi		838	576	56	**26**	33

Table 4: Creation times in seconds.

For HDT-AG and HDT-AT the shown times include the needed time to create the standard HDT parts (header, dictionary and triples) as well as the time to create the graph information matrix (the annotated graph bitmaps or the annotated triple bitmaps). Additionally, it includes the time needed to create the additional predicate and object index. For Jena, the denoted time includes the time to create the tdb store and all indexes. For Virtuoso,

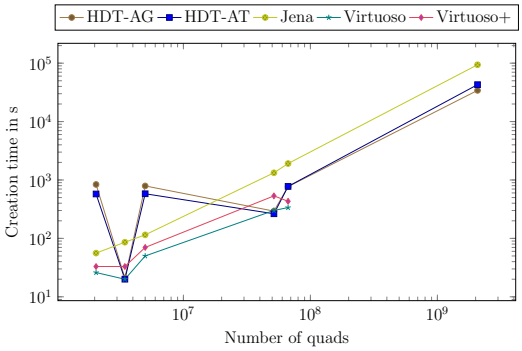


Figure 20: Creation times dependent on number of quads.

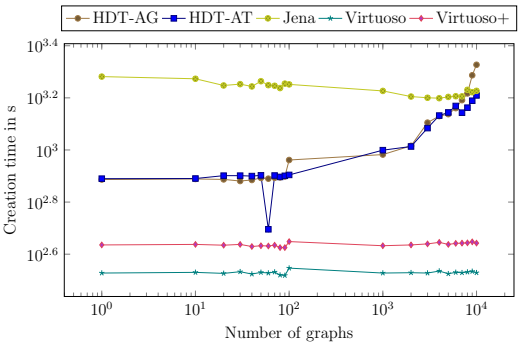


Figure 21: Creation times dependent on number of graphs.

the denoted time includes the creation of a new virtuoso instance, importing all data and creating the indexes. For Virtuoso+ the time to create the additional index was measured and added to the creation time of Virtuoso.

Virtuoso is the fastest system regarding creation time in nearly all cases. Only for BEAR-B hour HDT-AT is faster and for BEAR-B day HDT-AT takes the same time as Virtuoso. The slowest import is the combination of Jena and BEAR-A, taking about 26 hours. As BEAR-A could not be imported into Virtuoso, no times can be reported for Virtuoso and Virtuoso+.

It seems natural that with an increasing number of quads in the dataset, the time to import data into any system increases. To see whether this really is the case, the creation time for all of the systems are drawn in Figure 20. Each mark in the plot stands for one dataset, the datasets were sorted by their number of quads. While there are outliers, it can still be said that with an increasing number of quads, the import time increases.

The two peaks of HDT-AG and HDT-AT (at around 2 million and 5 million quads) can be explained with the structure of the underlying datasets. The 2 million quads dataset is the Liddi dataset, the 5 million quads dataset is the LDBC dataset. These are the datasets with a very high number of graphs (190,000 respectively 392,000). That indicates, that the creation time of HDTQ is not only dependent on the number of quads, but also on the number of graphs.

To verify this assumption, the creation times for the LUBM500 datasets are drawn in Figure 21. One can see that Virtuoso and Virtuoso+ are largely independent of the number of graphs. Jena on the other hand is even becoming slightly faster with an increasing number of graphs. As expected, both, HDT-AG and HDT-AT, need more time to import datasets if the number of graphs grows. However, compared to Jena, HDTQ is much faster if there is a low number of graphs, that even after the increase it is still faster (HDT-

AT) or only slightly slower (HDT-AG) for about 10,000 graphs. Moreover, Virtuoso is faster than the other systems in any case.

7.6 Querying Speed

In this section, the querying speed of the systems under review are compared. For this, random queries were generated[6] and then executed in all systems[7]. As HDTQ is not capable of SPARQL queries, only simple query patterns, where any of the quad components can be a variable, were tested. The testing of simple query patterns only is supported by the Linked Data Fragment approach [143], which promotes query APIs that only provide simple triple (ultimately, quad) query patterns.

Random queries were generated for each of the datasets. However, only those queries that provided the same number of results in all systems were selected for the evaluation. While that should actually be the case anyway, this ensures that if one system is faster than the other, this is really the case because of the speed of the respective system and not because one of the systems erroneously reports the wrong number of results (possibly a by far smaller number of quads, which is found much faster).

The queries were generated by first finding the distinct number of possibilities for the first non-variable in the given pattern (for this, HDTQ is queried, as it replies this information the fastest). One of the results was chosen randomly. If there is another non-variable in the pattern, Virtuoso+ was queried to find the number of possibilities given the previously identified component. One of the results was again chosen randomly. These steps are repeated for all of the non-variables in the pattern.

If the so-found solution was not found before, it is checked whether all of the systems report the same number of results for the query. If so, the query was added to the query collection. Then the steps are repeated, starting with the distinct number of possibilities for the first non-variable. If there are not enough possibilities for the requested number of queries (e.g. it is asked for ?P?? queries, but there are only 10 distinct predicates in the dataset), then the algorithm comes to an end early. All possibilities are taken as candidates, there results are checked and if they are equal, the query is accepted.

Note that this algorithm does not necessarily find a solution. It could be the case that the same query is found over and over again and the query generation never ends. Fortunately, this was not the case for any of the datasets.

[6] https://github.com/JulianRei/hdtq-java-queryGeneration

[7] https://github.com/JulianRei/hdtq-java-performanceTests

To test the performance of the systems, 100 queries were generated for each query pattern. However, there are special cases were less than 100 queries were generated. One of them is the quad pattern ????. As there is no variable in the query, there is not more than 1 query (???? itself) possible, for any dataset. Also, some of the datasets contain only a very low number of graphs, e.g. only 1 or 10 graphs. The pattern ???G therefore has only 1 or 10 queries. The last pattern for which sometimes less than 100 queries were generated is ?P??, as some of the datasets do not contain 100 or more distinct predicates. Also here, all predicates were taken as query candidates.

As it is often the case that some predicates appear in a lot of triples, while other appear in only a very few ones, two additional query types where considered. Both are actually the pattern ?P??, but one of them contains only queries with 100 or less results (further denoted as ?P?? (small)), while the other one contains only those with more than 100 results (further denoted as ?P?? (large)). Some of the datasets do not contain enough distinct predicates to split the ?P?? into such two groups. For these datasets these special cases were therefore not tested.

HDTQ's function *search* can directly be called with a quad pattern's components to resolve the pattern. However, Jena and Virtuoso do not provide for such a mechanism. Therefore, to test the patterns in these systems, the pattern was translated to a SPARQL query and this query was then tested.

The SPARQL query differs depending on whether the graph is given or not. If the graph <g> is given, then the query is:

```
SELECT ?s ?p ?o FROM <g> WHERE {?s ?p ?o}
```

Where the variable subject, predicate and object are replaced by their respective value if they are also given. Only the parts that are not given appear in the select clause.

If the graph is not given, the basic structure is:

```
SELECT ?s ?p ?o ?g WHERE {GRAPH ?g {?s ?p ?o}}
```

Again, the variable subject, predicate and object are replaced by their respective value if they are given. Only the parts that are not given appear in the select clause.

In the special case where all parts are given, i.e. SPOG, a dummy variable is returned, as there are no other variables that could appear in the select clause. In this case, the SPARQL query looks as follows:

```
SELECT ("a" as ?a) FROM <g> WHERE
{<s> <p> <o>}
```

Algorithm 9: MeasureQueryTime measures the average cold and warm execution times for a given query

Input: query, repetitions
Output: Average cold and warm times for the given query

for $i \leftarrow 1$ ***to*** $repetitions$ **do**
    $dropCache()$
    $doWarmup()$
    $coldStart \leftarrow now()$
    $doSearch(query)$
    $coldTime_i \leftarrow now() - coldStart$
    $warmStart \leftarrow now()$
    $doSearch(query)$
    $warmTime_i \leftarrow now() - warmStart$
$averageColdTime \leftarrow average(coldTime_i \dots coldTime_{repetitions})$
$averageWarmTime \leftarrow average(warmTime_i \dots warmTime_{repetitions})$
return $averageColdTime, averageWarmTime$

To get a more robust result, each query was not only executed once, but three times. Then, the average duration of the three measurements was taken as a result for the respective query. What is more, as can be seen in Algorithm 9 cold and warm times were measured. That means, for every iteration, the system cache was first cleared by executing the following statement:

```
sysctl vm.drop_caches=3
```

Then, a warmup was performed by querying any 100 quads using the ???? pattern. After that, the query was performed and the time was measured, this time is one of the three measurements of the cold time. Immediately after that, the query was performed again and the measured time is one of the three measurements of the warm time. After three iterations, the cold and warm times were averaged and the next query was tested.

As suggested by [135] setting the system swappiness to 10 ensures a better performance. This was therefore done before any queries were executed (not only for Virtuoso, but for all tested systems).

Each of the upcoming subsections covers one of the datasets used for the performance tests. The only exception is BEAR-A for which no query speed evaluation was done, as it could not be imported into Virtuoso and Virtuoso+. The bar charts, like Figure 22a compare the resolution speed of different quad patterns in Jena, Virtuoso and Virtuoso+ against HDT-AG or

HDT-AT. The bar height can be computed as $\frac{O_p}{H_p}$ where H_p is the resolution speed of HDT-AG/HDT-AT for the pattern p, O_p and O_p is the resolution speed of one of the others systems for the pattern p. If the other system (Jena, Virtuoso or Virtuoso+) is faster than HDT-AG/HDT-AT, then the result is inversed and the bar is drawn below the x-axis. The range between -1 and 1 is collapsed and forms the x-axis of the chart, indicated by a 1 on the y-axis.

7.6.1 BEAR-B day

Figure 22a shows the performance of Jena, Virtuoso and Virtuoso+ against HDT-AT for BEAR-B day when testing the systems cold. The bars are cut at a factor of 100 for better legibility. For ?POG Virtuoso (Virtuoso+) has a factor of 118 (116), for ??OG 513 (514). This means that HDT-AT is 118 (116) times faster than Virtuoso (Virtuoso+) for ?POG queries and 513 (514) times faster for ??OG queries.

As can be seen, HDT-AT is faster than Virtuoso and Virtuoso+ in all cases. Only the pattern S?O? is resolved at almost the same speed, however HDT-AT is still by factor of 1.04 (1.06) faster than Virtuoso (Virtuoso+). It is interesting to see that Virtuoso+ is mostly slower than Virtuoso, especially for SPOG. Virtuoso+ should be faster if a predicate and a graph is given, but the subject is not, which is the case for ?P?G. But even for this pattern Virtuoso+ is slower than Virtuoso.

Jena's performance is superior to Virtuoso and even outperforms HDT-AT by a small factor in several cases (namely, SP??, S???, ?P??, ???G and ????). Still, Jena is slower for the majority of the patterns, for ??OG it is even about 46 times slower than HDT-AT.

For ?POG and ??OG HDT-AT is clearly faster than Jena, Virtuoso and Virtuoso+. HDT-AT is more than 100 times faster than Virtuoso and Virtuoso+ for ?POG and even more than 500 times faster than Virtuoso and Virtuoso+ for ??OG. Jena is much better for this patterns, but is still much slower than HDT-AT.

One explanation for the bad performance of Jena, Virtuoso and Virtuoso+ is that these systems heavily utilize cache functionality. As the diagram shows results for cold systems, no cache could be used. Therefore, Figure 22b shows the same queries for warm systems.

When comparing warm systems, HDT-AT is still faster than Virtuoso and Virtuoso+ in most cases, but is slower for S?O?. Also, the factor of being faster is reduced greatly. While HDT-AT cold is more than 500 times faster for ??OG it is warm only about 12 (12) times faster than Virtuoso (Virtuoso+). Jena is faster than HDT-AT in a lot of cases and is especially

(a) Cold compared to HDT-AT

(b) Warm compared to HDT-AT

(c) Cold compared to HDT-AG

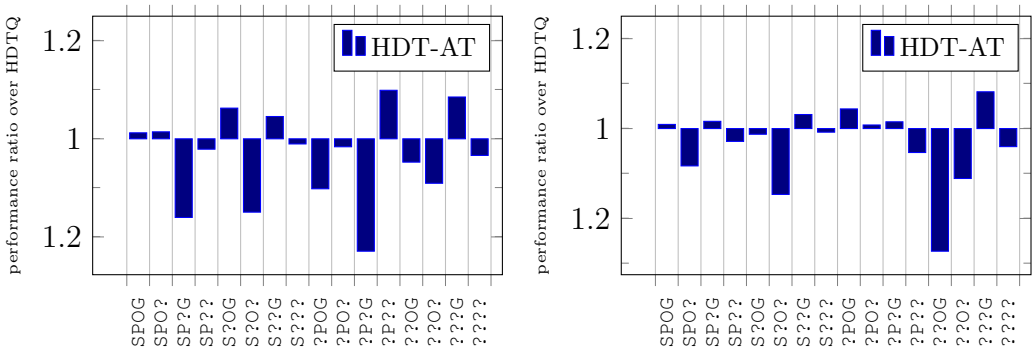
(d) Warm compared to HDT-AG

Figure 22: BEAR-B day quad pattern resolution speed. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

faster for S?O? (about a factor of 7).

Figure 22c and Figure 22d show the performance of Jena, Virtuoso and Virtuoso+ against HDT-AG when testing the systems cold and warm. When comparing the diagrams with the diagrams of HDT-AT, one notices that they appear to be identical. Bars in Figure 22c are again cut at 100. The actual value for ?POG Virtuoso (Virtuoso+) is 107 (105) and for ??OG 490 (491).

To highlight the differences in the performance of HDT-AG and HDT-AT, the two approaches are compared in Figure 23a (cold) and Figure 23b (warm). Most patterns show a very small difference between the two systems. This is most likely to be attributed to the relatively small dataset. The biggest differences are factors of about 1.2, which is fairly low compared to the differences in the performance of the other systems. As discussed in



(a) Cold (b) Warm

Figure 23: BEAR-B day comparing HDT-AG to HDT-AT. A k number above the x-axis means that HDT-AG is k times faster than HDT-AT. A k number below shows that HDT-AT is k times faster than HDT-AG.

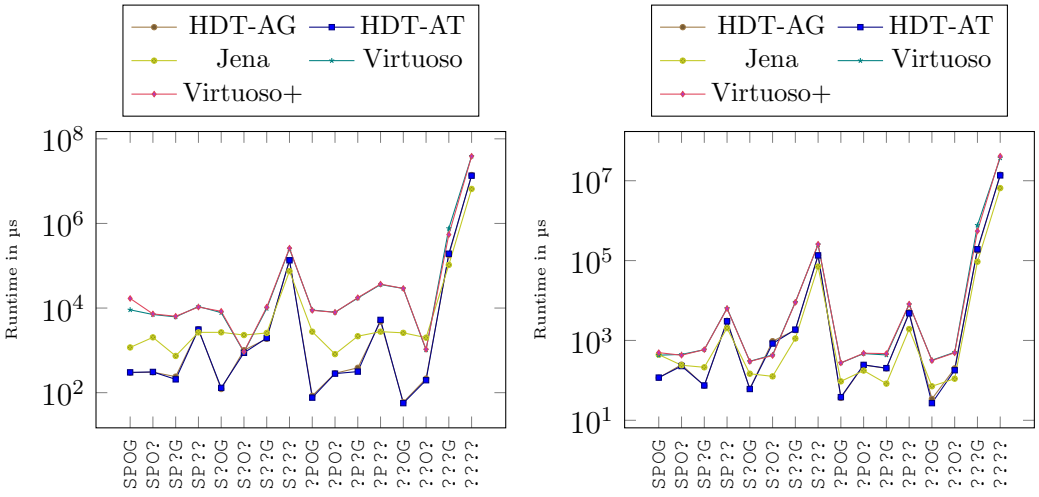
Section 5.1 HDT-AG should be faster than HDT-AT for those patterns that have the graph as given. While this can be observed for some patterns (like ???G) it is not the case for others (like ??OG). Possibly this is also due to the small dataset.

Looking at the patterns with bounded subject, that are resolved by the subject index, one can see that HDTQ outperforms Virtuoso and Virtuoso+ in all cases, except for S?O? (this might be attributable to the additional sequential iterator needed in HDT, see Section 4.3). For Jena, no such clear observation can be made. Conversely, access by predicate (?P?) served by the predicate-based HDT index shows that HDTQ is faster than Virtuoso and Virtuoso+, but slower than Jena in a warm state (except for in a cold state where Jena is slower for ?P?G). Finally, object access patterns (?PO and ??O) are resolved much faster by HDTQ than by the compared systems (especially if the graph is given and systems are tested cold).

The performance of Jena, Virtuoso and Virtuoso+ was clearly affected by

	SPOG	SPO?	SP?G	SP??	S?OG	S?O?	S??G	S???	?POG	?PO?	?P?G	?P??	??OG	??O?	???G	????
HDT-AG	0.39	0.81	0.31	0.98	0.5	0.95	0.97	1	0.43	0.84	0.52	1.05	0.58	0.93	1	1.02
HDT-AT	0.39	0.74	0.36	0.97	0.47	0.95	0.95	1	0.5	0.87	0.64	0.91	0.48	0.91	1	1.01
Jena	0.38	0.12	0.29	0.76	0.05	0.05	0.43	0.95	0.03	0.22	0.04	0.7	0.03	0.06	0.89	1
Virtuoso	0.05	0.06	0.1	0.59	0.04	0.48	0.93	0.99	0.03	0.06	0.03	0.22	0.01	0.49	1.01	0.96
Virtuoso+	0.03	0.06	0.09	0.6	0.04	0.45	0.86	0.99	0.03	0.06	0.03	0.22	0.01	0.46	1.01	1.06

Table 5: Ratios of warm and cold times for BEAR-B day. The smaller the ratio, the better is the performance of a system with respect to the cold scenario.



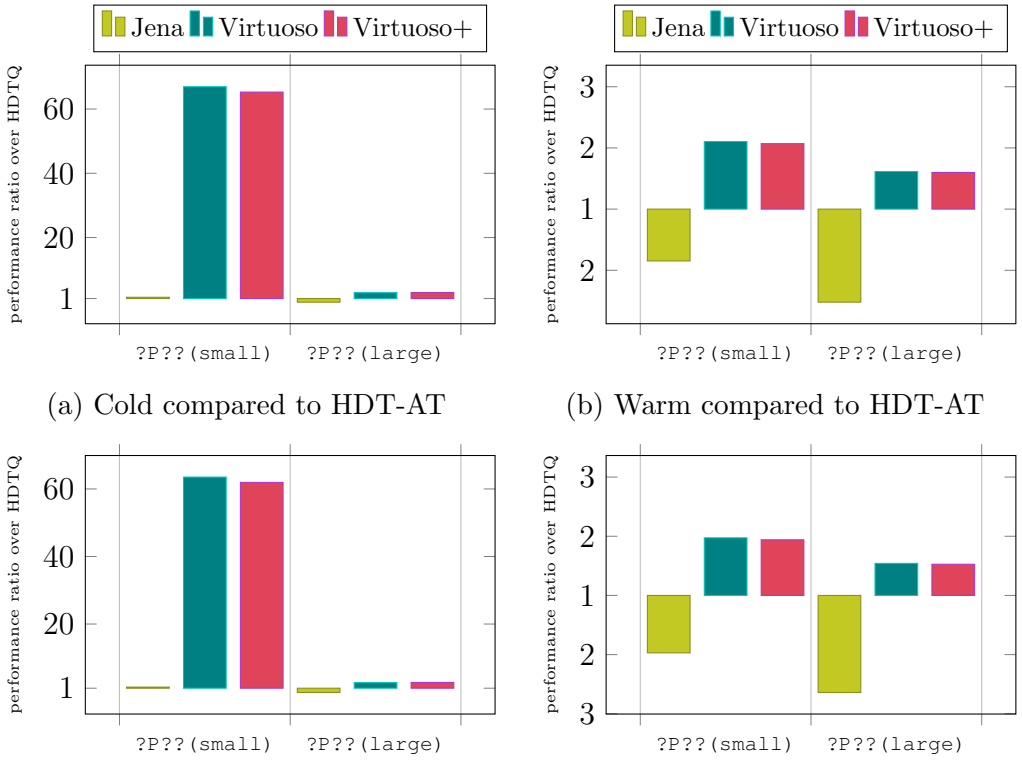
(a) Cold (b) Warm

Figure 24: BEAR-B day quad pattern resolution speed (absolute).

the warm state, whereas HDTQ does not show this strong correlation. There are a couple of possibilities why the systems performances are improving compared to HDTQ. One could be that HDTQ makes no use of caches and therefore does not become faster, but the others do. Another one is that all systems make use of caches, but the other systems become more faster than HDTQ becomes faster. To see why the performance of the other systems increases so much, Table 5 compares the performance of all systems in a cold and in a warm state.

The values in the table are calculated by dividing the average duration for a query in a system in a warm state by the duration of the same query in the same system in a cold state. The performance of Virtuoso and Virtuoso+ is especially improved in a warm state. Some queries need less than 10% of their duration in a cold system. Jena also greatly profits from a warm state, reducing the execution duration to a minimum. HDT-AG and HDT-AT profit much less from a warm system. Their performance still improves, but not as much as the performance of the other systems. Possibly this is due the fact that HDT-AG and HDT-AT are also very fast in a cold state and there is not so much room for improvement like for the other systems.

To see how fast the different systems resolve the patterns in absolute values, the average resolution times are shown in Figure 24a (cold) and Figure 24b (warm). Note that the y-axis is denoted in µs. Most queries for BEAR-B day are therefore executed in a fraction of a second. The diagram shows patterns that have the same combination of subject, predicate and



(a) Cold compared to HDT-AT

(b) Warm compared to HDT-AT

(c) Cold compared to HDT-AG

(d) Warm compared to HDT-AG

Figure 25: BEAR-B day small and large number of results for ?P??. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

object given adjacent to each other (e.g. SP?? and SP?G). The zigzag of the lines is a result of this order. As patterns that have the graph component given will most likely have less results than those were the graph component is not given, the former queries are usually faster. Looking at the figures one can also nicely see that the gap between the systems becomes much smaller in a warm state.

BEAR-B day has enough distinct predicates to split the quadpattern ?P?? into two groups, one with few results and one with many results (as described in Section 7.6). Figure 25 compares the speed of Jena, Virtuoso and Virtuoso+ against HDT-AG and HDT-AT, in a cold and a warm state. Looking at ?P?? (small) in the cold state, Jena performs almost the same as HDT-AG and HDT-AT. However, Virtuoso and Virtuoso+ are much slower (more than 60 times slower). This can, however, only be observed for a small

number of results. If the number of results is high, all systems performed more or less equally. As expected, the gap between the systems becomes much smaller in a warm state. Jena outperformed HDTQ by a small factor and Virtuoso and Virtuoso+ become much faster, but are still a bit slower than HDTQ.

7.6.2 BEAR-B hour

The BEAR-B hour dataset is very similar to the BEAR-B day dataset analyzed in the previous section (as both emerge from the same source, but aggregate the data on a different timescale, see Section 7.3). Noticeable differences are that BEAR-B hour has about twice as many objects and by far more graphs (about 1,300 compared to only 89). Also, the BEAR-B hour dataset contains about 6 times more quads than BEAR-B day.

Figure 26a shows the performance of Jena, Virtuoso and Virtuoso+ against HDT-AT for BEAR-B hour when testing the systems cold. The bars are cut at a factor of 100 for better legibility. For SPOG Virtuoso+ has a factor of 139, for S?OG 192. For ??OG the values for Jena / Virtuoso / Virtuoso+ are 151 / 194 / 269.

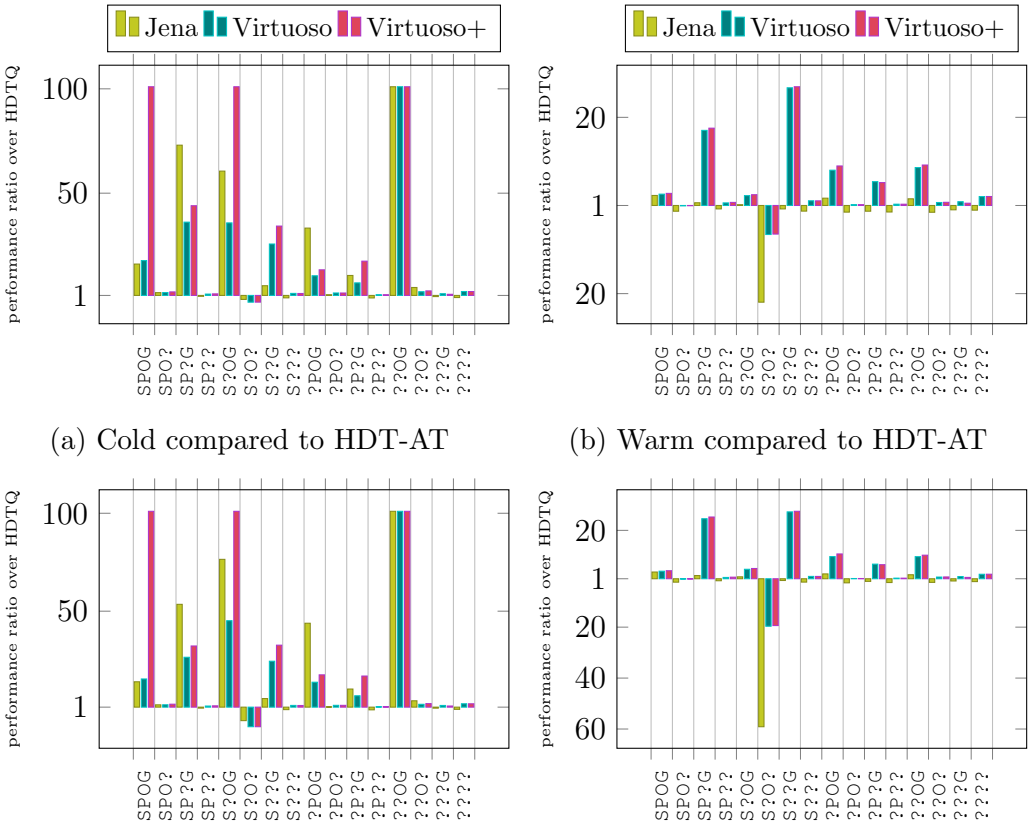
While HDT-AT outperformed Virtuoso and Virtuoso+ in all cases for BEAR-B day, for BEAR-B hour Virtuoso and Virtuoso+ are faster than HDT-AT for S?O? (as for BEAR-B day, this is likely attributable to the additional sequential iterator needed in HDT, see Section 4.3). In all other cases, HDT-AT is still faster.

The effect that Virtuoso+ is slower than Virtuoso grew significantly, leaving a large gap between the two systems, especially for SPOG, S?OG and ??OG.

Similar to BEAR-B day, we can observe that HDT-AT clearly outperforms the compared systems. In contrast to the BEAR-B day dataset, Jena's performance decreases for SPOG, SP?G, S?OG and ??OG. While Virtuoso's and Virtuoso+'s performance improved for SPO?, ?POG, ?PO? and ?P?G,

	SPOG	SPO?	SP?G	SP??	S?OG	S?O?	S??G	S???	?POG	?PO?	?P?G	?P??	??OG	??O?	???G	????
HDT-AG	0.14	0.95	0.44	0.99	0.37	0.98	0.88	0.99	0.14	0.95	0.18	0.97	0.34	0.86	1	1
HDT-AT	0.19	0.97	0.86	1.01	0.44	0.98	0.95	1	0.12	1	0.21	1.01	0.46	0.98	1.01	1
Jena	0.04	0.19	0.02	0.83	0.01	0.13	0.1	0.97	0.01	0.31	0.01	0.93	0.01	0.08	0.81	1
Virtuoso	0.04	0.38	0.41	0.96	0.04	0.57	0.98	1.02	0.1	0.54	0.18	0.98	0.02	0.58	1	1.01
Virtuoso+	0	0.34	0.35	0.96	0.01	0.58	0.74	1.03	0.08	0.54	0.07	0.98	0.02	0.52	0.97	1.02

Table 6: Ratios of warm and cold times for BEAR-B hour. The smaller the ratio, the better is the performance of a system with respect to the cold scenario.



(a) Cold compared to HDT-AT

(b) Warm compared to HDT-AT

(c) Cold compared to HDT-AG

(d) Warm compared to HDT-AG

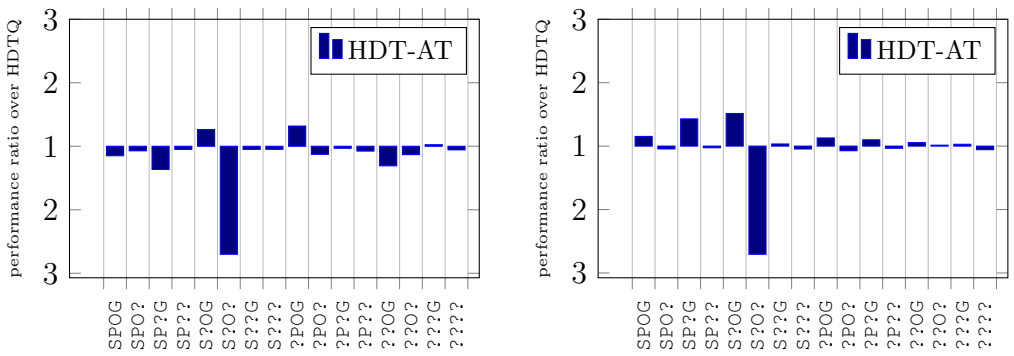
Figure 26: BEAR-B hour quad pattern resolution speed. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

Virtuoso+'s performance decreased for SPOG, SP?G, S?OG and S??G.

Looking at Figure 26b one can see that the results differ only slightly from those of BEAR-B day. The most significant difference is that the factors for SP?G, S?O?, S??G and ?P?G grew noticeably.

Figure 26c and Figure 26d show the performance of Jena, Virtuoso and Virtuoso+ against HDT-AG when testing the systems cold and warm. Bars in Figure 26c are again cut at 100. The actual values for Virtuoso+ are 121 (for SPOG) and 242 (for S?OG). For ??OG the factors for Jena / Virtuoso / Virtuoso+ are 115 / 148 / 206.

When comparing the results of HDT-AG for BEAR-B hour in a cold state with the results for BEAR-B day, the same observations can be made as for HDT-AT described above. In a warm state, factors also grew compared to



(a) Cold (b) Warm

Figure 27: BEAR-B hour comparing HDT-AG to HDT-AT. A k number above the x-axis means that HDT-AG is k times faster than HDT-AT. A k number below shows that HDT-AT is k times faster than HDT-AG.

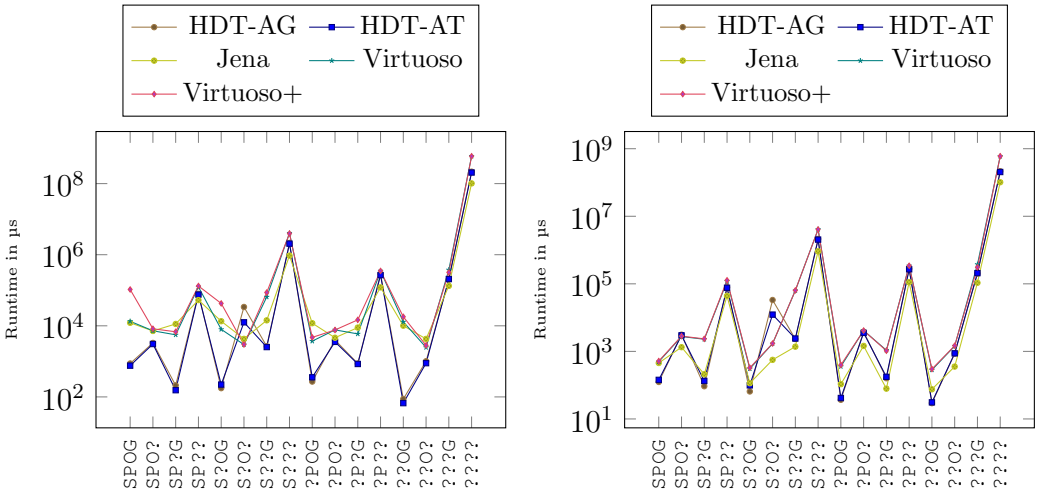
BEAR-B day, in the corner case of S?O? for Jena even to a factor of about 60.

The performance of HDT-AG and HDT-AT is compared in Figure 27a (cold) and Figure 27b (warm). While the differences between HDT-AG and HDT-AT for BEAR-B day were only marginal, for BEAR-B hour first differences can be observed. While for the BEAR-B day dataset the maximum difference was a factor of 1.2, now the maximum difference is a factor of 3.

Looking at the patterns with bounded subject, which are resolved by HDT's subject index, one can see similar results like in the previous section for BEAR-B day. HDTQ outperforms Virtuoso and Virtuoso+ partly even by a factor greater than 100 and is marginally slower for SPO?. Again, Virtuoso and Virtuoso+ are faster for S?O?. For Jena, no clear trend for these patterns can be observed. Also, the queries that are served by the predicate-based HDT index show a similar results. HDTQ is faster than Virtuoso and Virtuoso+, but slower than Jena (except for in a cold state where Jena is slower for ?P?G). Finally, object access patterns (?PO and ??O) are usually resolved faster by HDTQ than by the compared systems (again, especially in a cold state if the graph is given). Only in a warm state, if the graph is given, Jena is faster than HDTQ.

Table 6 compares the performance of all systems in a cold and in a warm state. Virtuoso and Virtuoso+ profit a lot from a warm state, but not as much as for the BEAR-B day dataset. Jena also greatly profits from the warm state, similar to the measured values for BEAR-B day. HDTQ profits more from a cache with the bigger dataset.

To see how fast the different systems resolve the patterns in absolute



(a) Cold (b) Warm

Figure 28: BEAR-B hour quad pattern resolution speed (absolute).

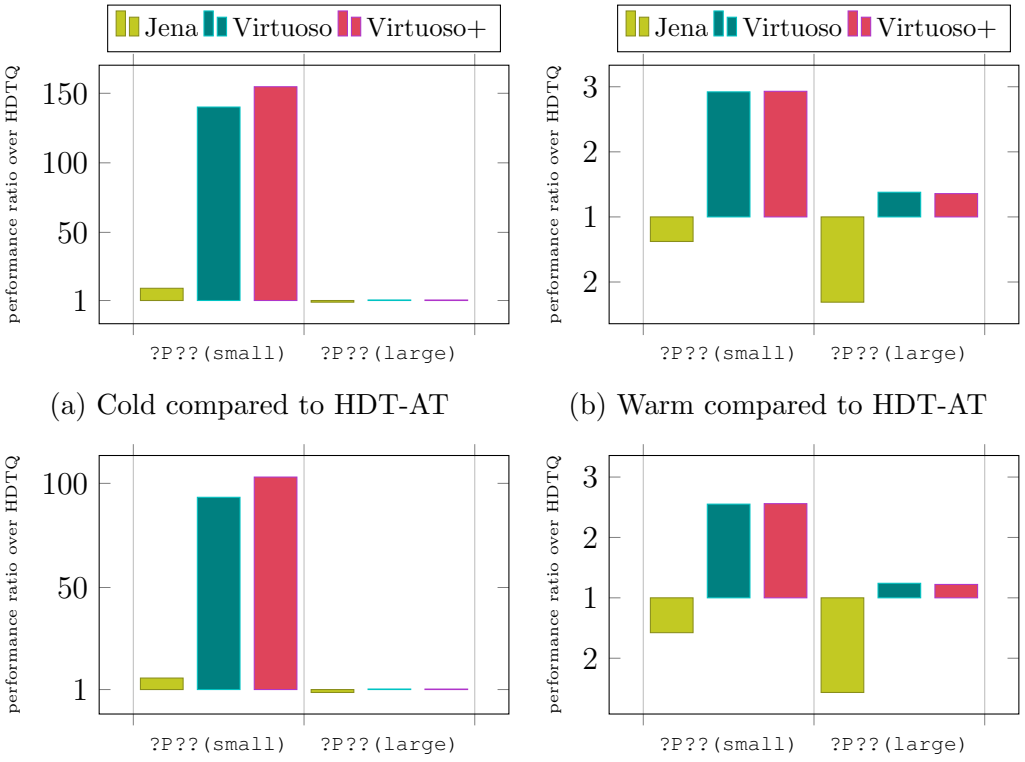
values, the average resolution times are shown in Figure 28a (cold) and Figure 28b (warm). The observations for these diagrams are the same as for BEAR-B day.

Figure 29 compares the speed of Jena, Virtuoso and Virtuoso+ against HDT-AG and HDT-AT, in a cold and a warm state for the two groups of ?P??. Differences to the results for the BEAR-B day dataset are that Virtuoso and Virtuoso+ are significantly slower than HDTQ for ?P?? (small) in a cold state. Furthermore, Jena's performance for ?P?? (small) increases marginally while the performance of Virtuoso and Virtuoso+ for ?P?? (small) marginally decreases in a warm state.

7.6.3 LDBC

The LDBC dataset is very different compared to the former analyzed datasets BEAR-B day and BEAR-B hour. Looking at Table 1 one can see that LDBC has much more distinct subjects, objects and graphs (190,000 compared to 89 respectively 1,299) and triples (5,000,000 compared to 82,000 respectively 167,000). However, LDBC has only 16 distinct predicates (compared to more than 1,700 for BEAR-B day and BEAR-B hour). Also, the number of triples and quads is the same for LDBC meaning that each triple appears in exactly one graph.

Figure 30a shows the performance of Jena, Virtuoso and Virtuoso+ against HDT-AT for LDBC when testing the systems cold. The bars are cut at a factor of 100 for better legibility. The actual value of ???G for Virtuoso



(a) Cold compared to HDT-AT

(b) Warm compared to HDT-AT

(c) Cold compared to HDT-AG

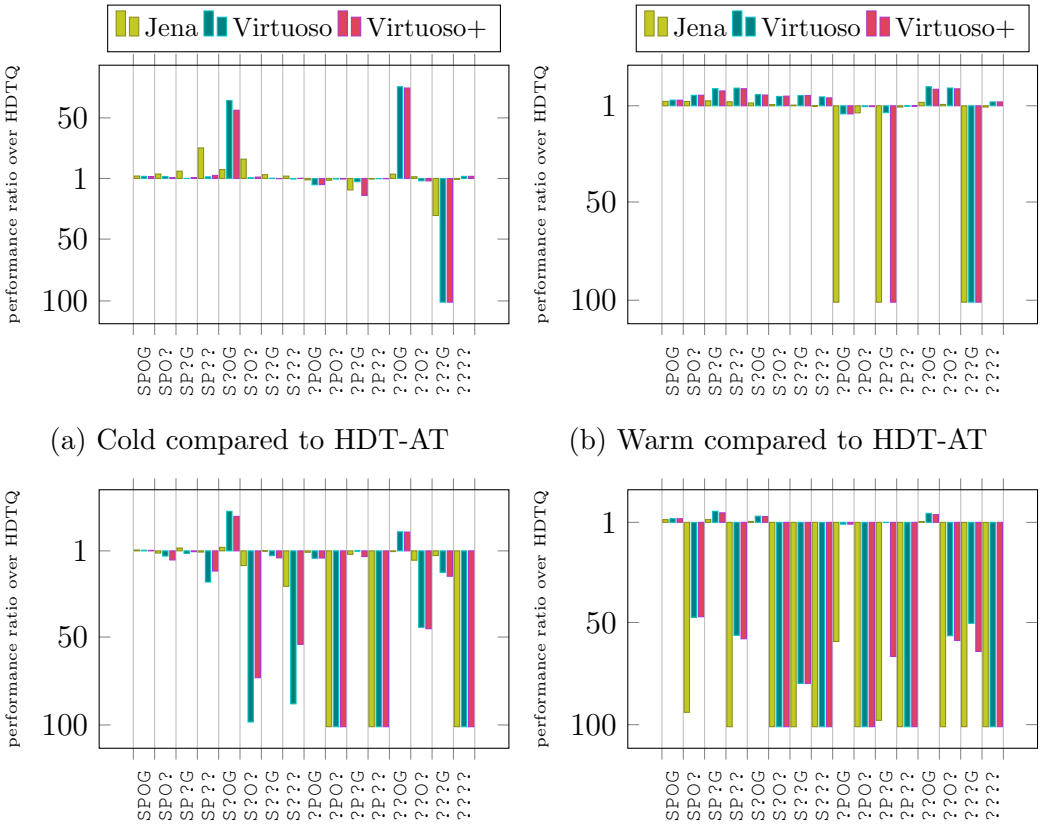
(d) Warm compared to HDT-AG

Figure 29: BEAR-B hour small and large number of results for ?P??. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

(Virtuoso+) is about 111 (130).

HDT-AT is much slower than the compared systems for the ???G pattern. Apart from that, HDT-AT is also considerably slower for ?POG and ?P?G and marginally slower for S??? (only compared to Virtuoso), ?PO?, ?P??, ??O? (only compared to Virtuoso and Virtuoso+) and ???? (only compared to Jena). The 3 patterns for which HDT-AT is the slowest system all have the graph component given. As HDT-AT is not well suited for such patterns, it is reasonable that is slower in these cases.

On the other hand, HDT-AT is much faster for S?OG and ??OG (especially compared to Virtuoso and Virtuoso+), which is counterintuitive, as the graph component is also given for these patterns. Compared to Jena, HDT-AT is also considerably faster for SP?? and S?O?. For all other patterns, the difference in the quad pattern resolution speed is negligible.



(a) Cold compared to HDT-AT

(b) Warm compared to HDT-AT

(c) Cold compared to HDT-AG

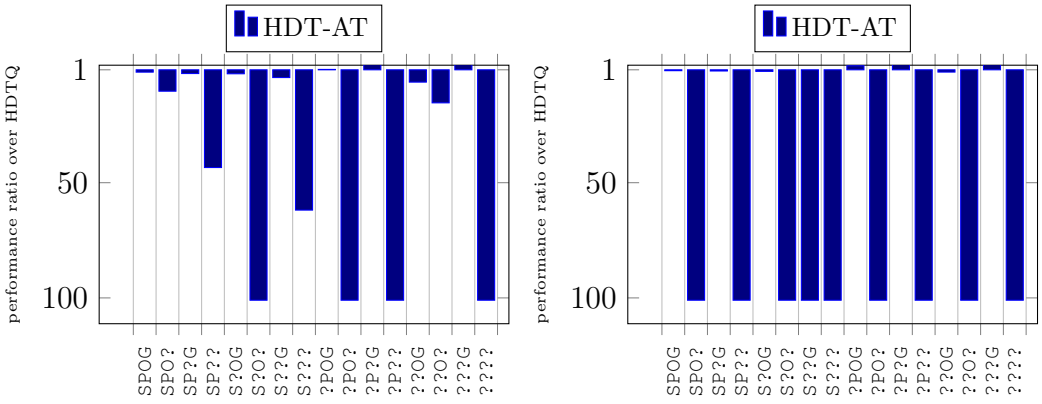
(d) Warm compared to HDT-AG

Figure 30: LDBC quad pattern resolution speed. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

Figure 30b shows the warm results for Jena, Virtuoso and Virtuoso+ against HDT-AT. Bars are again cut at 100 for better legibility. The actual values for ???G are 1192 / 345 / 440 for Jena / Virtuoso / Virtuoso+, for ?P?G they are 414 / 282 for Jena / Virtuoso+ and for ?POG it is 162 for Jena.

HDT-AT performs better than Jena, Virtuoso and Virtuoso+ in most of the cases. Only in the corner cases (S???, ?P?G and ???G) does HDT-AT loose considerably against the compared systems. Also, it can be seen that again the other systems were able to achieve greater performance gains in the warm state than HDT-AT.

While for the datasets analyzed in the previous sections (BEAR-B day and BEAR-B hour) Virtuoso+ was not significantly faster (often even slower)



(a) Cold (b) Warm

Figure 31: LDBC comparing HDT-AG to HDT-AT. A k number above the x-axis means that HDT-AG is k times faster than HDT-AT. A k number below shows that HDT-AT is k times faster than HDT-AG.

than Virtuoso, for LDBC there is a significant difference in the performance of ?P?G in a warm state. This difference can be explained as Virtuoso+ should perform better than Virtuoso if the predicate and the graph are given, but the subject is not, which is exactly the case for the pattern ?P?G.

Figure 30c and Figure 30d show the performance of Jena, Virtuoso and Virtuoso+ against HDT-AG when testing the systems cold and warm. As HDT-AG was not capable of answering ?PO?, ?P?? and ???? queries within a reasonable time (none of the patterns was resolved within 10 hours), the execution was stopped. Therefore all bars show a value of 100 for these patterns. Further cut values in Figure 30d are SP?? with a value of 185 for Jena, S?O? with values of 939 / 260 / 253 for Jena / Virtuoso / Virtuoso+, S??G with a value of 364 for Jena, S??? with values of 1837 / 261 / 287 for Jena / Virtuoso / Virtuoso+, ??O? with a value of 339 for Jena and ???G with a value of 174 for Jena.

In a cold state, HDT-AG outperforms the compared systems for S?OG and SPOG as well as Virtuoso, Virtuoso+ for ??OG and Jena for SP?G. However, it is slower for all other patterns and three of the patterns did not finish in reasonable time. This makes HDT-AG the worst of the compared systems for the LDBC dataset in a cold state. In a warm state HDT-AG's performance is even worse, while still being faster in some cases (SPOG, SP?G, S?OG and ??OG), the performance for the other patterns is extremely bad.

The bad performance of HDT-AG is a result of the huge graph information matrix resulting from the high amount of graphs together with a relative

high number of quads in the LDBC dataset. While searching for results for most of the patterns the RAM of the testing environment was fully occupied and swap space was being used. This slowed the system down, leading to bad performance results.

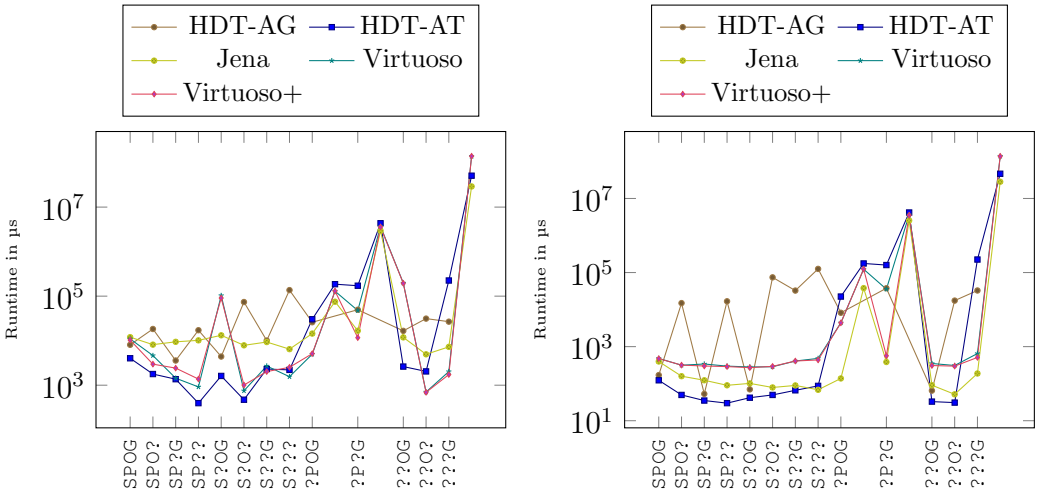
As can already be assumed, the performance of HDT-AT is much better than the one of HDT-AG for LDBC. For the sake of completeness, the performance of HDT-AT and HDT-AG is depicted in Figure 31a (cold) and Figure 31b (warm). The figures confirm the observations from Figure 30. In a cold state, HDT-AT is way faster than HDT-AG in many cases. Only for ?POG, ?P?G and ???G HDT-AG is marginally faster. Especially in the warm state, one can nicely see that HDT-AT is faster if the graph component is variable, while HDT-AG is faster if the graph component is given. In the warm state there is one exception to this, namely S??G, where HDT-AT is faster than HDT-AG.

Looking at the patterns with bounded subject, which are resolved by HDT's subject index, one can see that HDT-AT is superior in basically all cases. Only for S??? Virtuoso is marginally faster when comparing cold systems and Jena is marginally faster when comparing warm systems. The queries that are served by the predicate-based HDT index are resolved at equal speed if the graph is not given and faster by the compared systems if the graph is given. Lastly, object access patterns are resolved faster by HDT-AT if the predicate is not given (??OG and ??O?) and faster by the compared system if the predicate is given (?POG and ?PO?). As HDT-AG performs poor for basically all cases, no comparison for each of HDT's indexes is drawn.

Table 7 compares the performance of all systems in a cold and a warm state. Values in the table denoted as 0 are actually values smaller than 0.005 and therefore rounded down to zero. Compared to the warm-cold ratios of the datasets discussed in the previous sections, for LDBC HDT-AT profits greatly from a warm system. HDT-AG does not profit as much and for

	SPOG	SPO?	SP?G	SP??	S?OG	S?O?	S??G	S???	?POG	?PO?	?P?G	?P??	??OG	??O?	???G	????
HDT-AG	0.02	0.82	0.01	0.98	0.02	1.01	3.24	0.92	0.32	NA	0.76	NA	0	0.56	1.23	NA
HDT-AT	0.03	0.03	0.03	0.08	0.03	0.11	0.03	0.04	0.74	0.95	0.93	0.96	0.01	0.02	1	0.92
Jena	0.03	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.51	0.02	0.86	0.01	0.01	0.03	0.97
Virtuoso	0.04	0.07	0.24	0.33	0	0.38	0.15	0.31	0.88	0.95	0.75	0.99	0	0.44	0.32	1
Virtuoso+	0.05	0.11	0.12	0.21	0	0.29	0.2	0.17	0.86	0.96	0.05	0.99	0	0.43	0.29	0.99

Table 7: Ratios of warm and cold times for LDBC. The smaller the ratio, the better is the performance of a system with
respect to the cold scenario.



(a) Cold (b) Warm

Figure 32: LDBC quad pattern resolution speed (absolute).

???G, S?O? and S??G the duration is getting even worse in a warm state. Values denoted as NA could not be measured as the system did not finish the queries in a reasonable timeframe. The ratios are, except for HDT-AG, very similar, so all systems profit greatly from a warm state. Still, there are patterns where no system becomes significantly faster in a warm state (?P?? and ????) or only some of them do (?PO?, ?P?G, ??O? and ???G).

Figure 32 shows the average resolution times for all systems in a cold and in a warm state. Note that some values (?PO?, ?P?? and ????) are missing for HDT-AG and are therefore omitted for the respective series in the diagrams.

It is interesting to see that for some patterns HDT-AT, Jena, Virtuoso and Virtuoso+ become faster if the graph component is variable. This is the case for SPO, SP?, S?O, S?? and ??O (each once with the graph component given and once as a variable). While this is counterintuitive, as the respective pattern with the graph component as a variable must have the same or more number of results than their counterpart, this can be explained by the specific structure of the LDBC dataset. As no triple appears in more than exactly one graph, the difference in the number of results if the graph component is given or not is minimal. In the extreme case of SPOG and SPO?, the number of results is in both cases 1. Apparently, if there is only 1 result (or a very small number of results), it is faster to query for all results in all graphs than to specify in which graph to search. HDT-AG does not show this changed behavior, it shows the same zigzag pattern that all systems show for the

previously discussed datasets.

What is more, in the cold and the warm state, one can clearly see the very good performance of HDT-AT compared to the other systems (especially for those patterns that have the subject given). While the zigzag effect for HDT-AG becomes stronger in a warm state (because fast patterns get even faster), the differences between adjacent patterns (where the graph component is once given, once variable) smoothens out in the warm state for all other systems. Note that the y-axis is denoted in µs, which means that patterns that have the subject given as well as ??OG and ??O? are resolved by HDT-AT extremely fast, in less than 0.1 milliseconds.

As there are no more than 16 distinct predicates in the LDBC dataset, the pattern ?P?? cannot be split into two groups. Therefore, for this dataset, diagrams showing two ?P?? groups are omitted.

7.6.4 Liddi

Compared to the LDBC dataset, the Liddi dataset is relatively small. It has about half the number of distinct subjects, and less than half the number of objects and quads. However, the number of graphs is about double the number of graphs of LDBC. With a little less than 400,000 graphs, Liddi has the most graphs of all analyzed datasets.

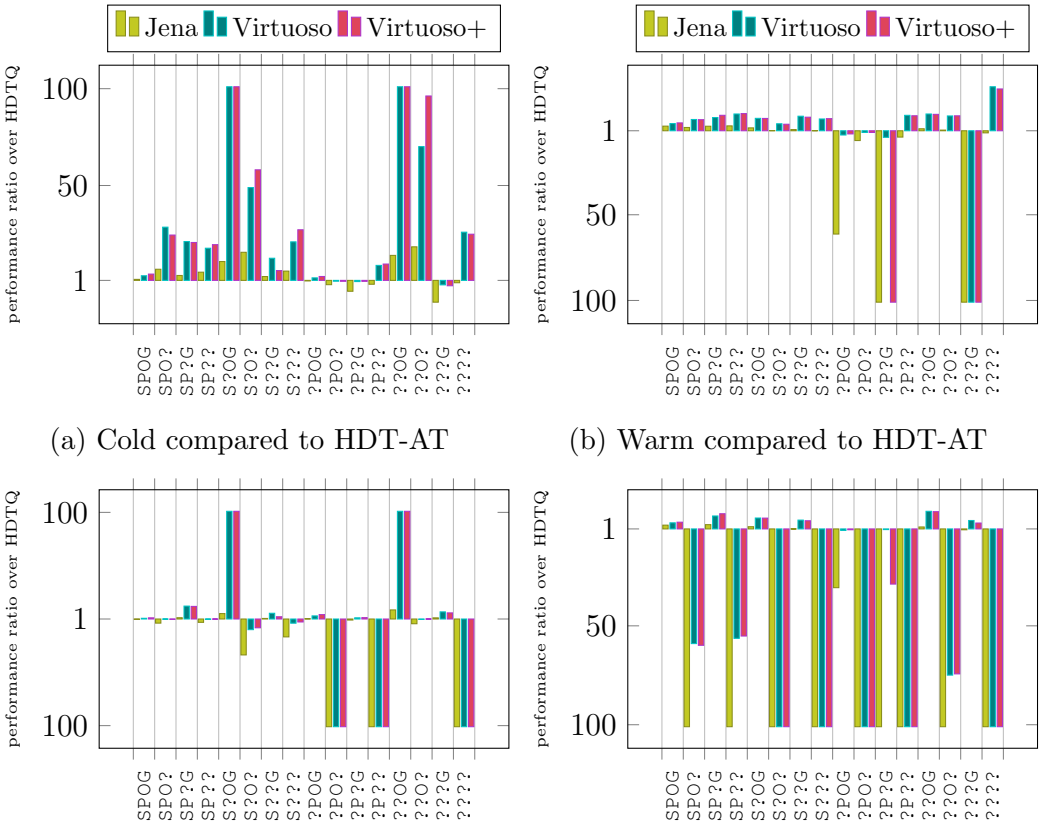
Figure 33a shows the performance of Jena, Virtuoso and Virtuoso+ against HDT-AT for Liddi when testing the systems cold. The bars are cut at a factor of 100 for better legibility. The actual value of S?OG for Virtuoso (Virtuoso+) is 182 (181). The value of ??OG for Virtuoso (Virtuoso+) is 266 (266).

In contrast to the LDBC dataset, for the Liddi dataset HDT-AT outperforms all other systems by far for almost all patterns when testing the systems cold. For the few patterns in which one of the competing systems is faster (?PO?, ?P?G, ?P??, ???G) the difference is only marginal. Particularly the performance of HDT-AT for ???G improved greatly, from being more than 100 times slower compared to Virtuoso to being roughly the same.

Figure 33b shows the warm results for Jena, Virtuoso and Virtuoso+ against HDT-AT. Bars are again cut at 100 for better legibility. The actual values for ?P?G are 628 / 104 for Jena / Virtuoso+, for ???G they are 1154 / 152 / 201 for Jena / Virtuoso / Virtuoso+.

When comparing HDT-AT to the other systems in a warm state, the results are very similar to the results of the LDBC dataset. Noticeable differences are an improved performance of HDT-AT compared to Jena for ?POG and compared to Virtuoso and Virtuoso+ for ?P?? and ????.

Figure 33c and Figure 33d show the performance of Jena, Virtuoso and



(a) Cold compared to HDT-AT

(b) Warm compared to HDT-AT

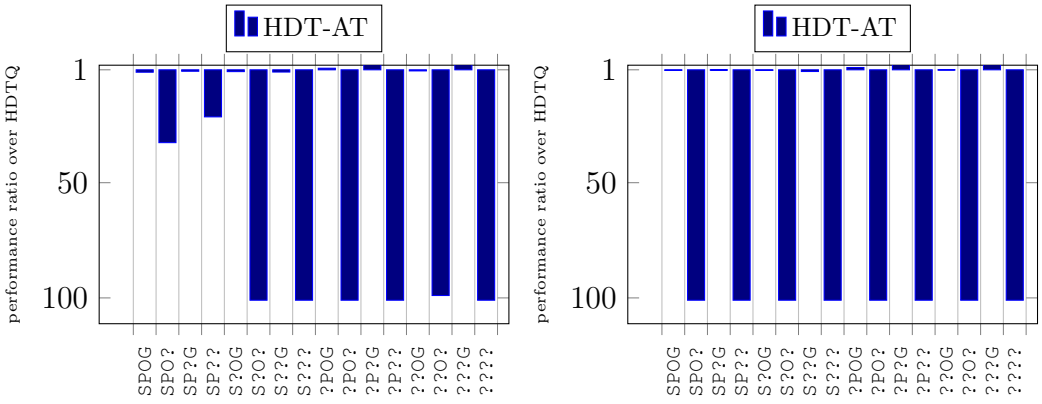
(c) Cold compared to HDT-AG

(d) Warm compared to HDT-AG

Figure 33: Liddi quad pattern resolution speed. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

Virtuoso+ against HDT-AG when testing the systems cold and warm. As HDT-AG was again not capable of answering ?PO?, ?P?? and ???? queries within a reasonable time (none of the patterns was resolved within 10 hours), the execution was stopped. Therefore all bars show a value of 100 for these patterns. Further cut values in Figure 33c are S?OG with a value of 102 (102) for Virtuoso (Virtuoso+) and for ??OG 180 (181) for Virtuoso (Virtuoso+). In Figure 33d cut values include for S?O? 2566 / 548 / 579 and for S??? 1519 / 199 / 194 each for Jena / Virtuoso / Virtuoso+ and additionally for Jena for SPO? / SP?? / ?P?G / ??O? values of 153 / 158 / 174 / 519.

While HDT-AG was again not capable of answering certain queries within a reasonable timeframe, the performance improved significantly for several other patterns (S?OG, S?O?, S???, ??OG and ??O?) when testing the systems



(a) Cold (b) Warm

Figure 34: Liddi comparing HDT-AG to HDT-AT. A k number above the x-axis means that HDT-AG is k times faster than HDT-AT. A k number below shows that HDT-AT is k times faster than HDT-AG.

cold. Yet, the warm results show a very similar picture to the warm LDBC results, making HDT-AG practically unusable for datasets with a number of triples and graphs in the range of the Liddi dataset.

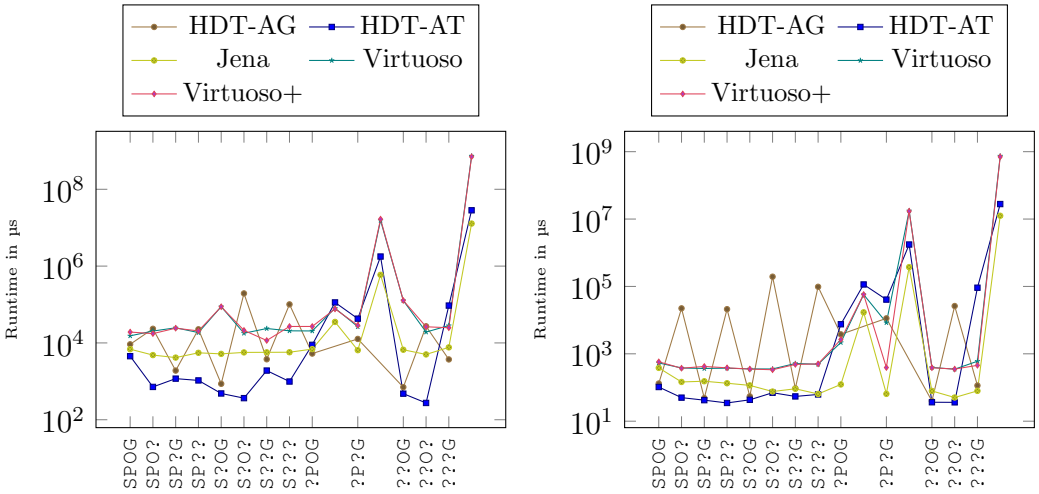
The bad performance of HDT-AG can easily be explained, as, like for the LDBC dataset, the systems memory was again completely occupied. As for the LDBC dataset, this is caused by a very big graph information matrix (consider that Liddi is the dataset with the most graphs under review). The use of swap space slowed the system down, leading to the bad results for HDT-AG.

The performance of HDT-AT and HDT-AG is depicted in Figure 34a (cold) and Figure 34b (warm). The results are almost identical to the results for the LDBC dataset. Again, while their performance for queries with bounded graph is basically equivalent, HDT-AT outperforms HDT-AG for queries without bounded graph by far. Whereas for S??G for the LDBC dataset HDT-AT was unexpectedly faster than HDT-AG in a warm state, this can not be observed for the Liddi dataset anymore.

Table 8 compares the performance of all systems in a cold and a warm state. Values in the table denoted as 0 are actually values smaller than 0.005 and therefore rounded down to zero. The results are very similar to those for LDBC. One exception to this is HDT-AG, which for the Liddi dataset improves for S??G and ???G and becomes worse for ??O? in a warm state compared to the LDBC dataset. Another exception is Virtuoso and Virtuoso+ for the ?POG and ??O? patterns, for which it noticeable improves in a warm state.

	SPOG	SPO?	SP?G	SP??	S?OG	S?O?	S??G	S???	?POG	?PO?	?P?G	?P??	??OG	??O?	???G	????
HDT-AG	0.01	0.96	0.03	0.93	0.06	1	0.03	0.97	0.72	NA	0.89	NA	0.06	0.97	0.03	NA
HDT-AT	0.02	0.07	0.04	0.03	0.09	0.19	0.03	0.06	0.85	1.01	0.95	0.99	0.08	0.13	0.98	0.98
Jena	0.06	0.03	0.04	0.02	0.02	0.01	0.02	0.01	0.02	0.48	0.01	0.63	0.01	0.01	0.01	0.98
Virtuoso	0.04	0.02	0.01	0.02	0	0.02	0.02	0.02	0.11	0.72	0.32	1.13	0	0.02	0.02	1.01
Virtuoso+	0.03	0.02	0.02	0.02	0	0.02	0.04	0.02	0.1	0.75	0.01	1.02	0	0.01	0.02	1

Table 8: Ratios of warm and cold times for Liddi. The smaller the ratio, the better is the performance of a system with respect to the cold scenario.



(a) Cold

(b) Warm

Figure 35: Liddi quad pattern resolution speed (absolute).

Looking at the absolute values of the performance of all systems for Liddi in Figure 35 one can see the same effect like in the previous section again. That is, for a couple of patterns the performance of HDT-AT (and partly also the performance of other systems) improves if the graph is not given, compared to when the graph is given. This is the case for SPO, SP?, S?O, S?? and ??O (each once with the graph component given and once as a variable).

Furthermore, especially when looking at the results for warm systems, one can see an even stronger zigzag pattern for HDT-AG then for the LDBC dataset, which means that the performance of the system is hugely dependent on the presence/absence of the graph component in the search pattern.

As there are again only a few distinct predicates in the dataset (23), the pattern ?P?? cannot be split into two groups. Therefore, for this dataset, diagrams showing two ?P?? groups are again omitted.

7.7 Scalability test with increasing number of graphs

The performance of the different systems where tested for the LUBM500 dataset with 1, 10, 100, 1000 and 9998 graphs (for better readability, in this section the 9998 graphs are referred to as 10,000 graphs). Compared to the previous analyses, the analysis for LUBM500 will therefore look different. Instead of looking at one particular RDF graph, the performance of the different systems for the same pattern for datasets with a varying number of graphs will be looked at. As the number of graphs is the only difference between the different LUBM500 datasets, the datasets fit perfectly for this purpose.

To make the results for a different number of graphs even more comparable, the generated queries for all patterns that have the graph component as a variable were reused for all LUBM500 datasets. For those patterns where the graph component is given, new queries were generated for each of the datasets.

Figure 36 shows the performance of all systems for all patterns when testing the systems cold. Each sub figure shows one pattern. As HDT-AG did not finish within a reasonable timeframe (it was still searching after 10 hours) for ?P?? and ???? for the dataset with the highest number of graphs (10,000), the respective values in the diagrams are omitted.

Looking at the results for those patterns that have the subject component given (i.e. those where HDTQ makes use of the subject index), one can see that the performance of HDT-AG and HDT-AT is mostly unaffected by the number of graphs. The performance of HDT-AG only becomes slightly worse for 10,000 graphs for S?O? (as for previous datasets, this is likely attributable to the additional sequential iterator needed in HDT, see Section 4.3). The performance of HDT-AT becomes a bit worse for 10 graphs for S???, but becomes better again for 100 and more graphs.

The performance of Jena is only constant for S?OG. For SPO? the performance degenerates with 1,000 graphs and becomes even worse for 10,000 graphs. For the other patterns that have the subject given, the performance begins to decrease noticeable with 10,000 graphs.

Virtuoso and Virtuoso+ resolve patterns that have the subject given with varying speed, depending on the number of graphs. While the performance of both systems is rather constant for S?O?, it goes up and down for SPO?, SP?G, SP??, S??G and S???. For the latter, only Virtuoso+ varies, Virtuoso has a stable performance. For SPOG the performance of Virtuoso even becomes slowly better with an increasing number of graphs, for Virtuoso+ the opposite is the case. For S?OG both systems become much slower with 1,000 graphs and again a bit slower with 10,000 graphs.

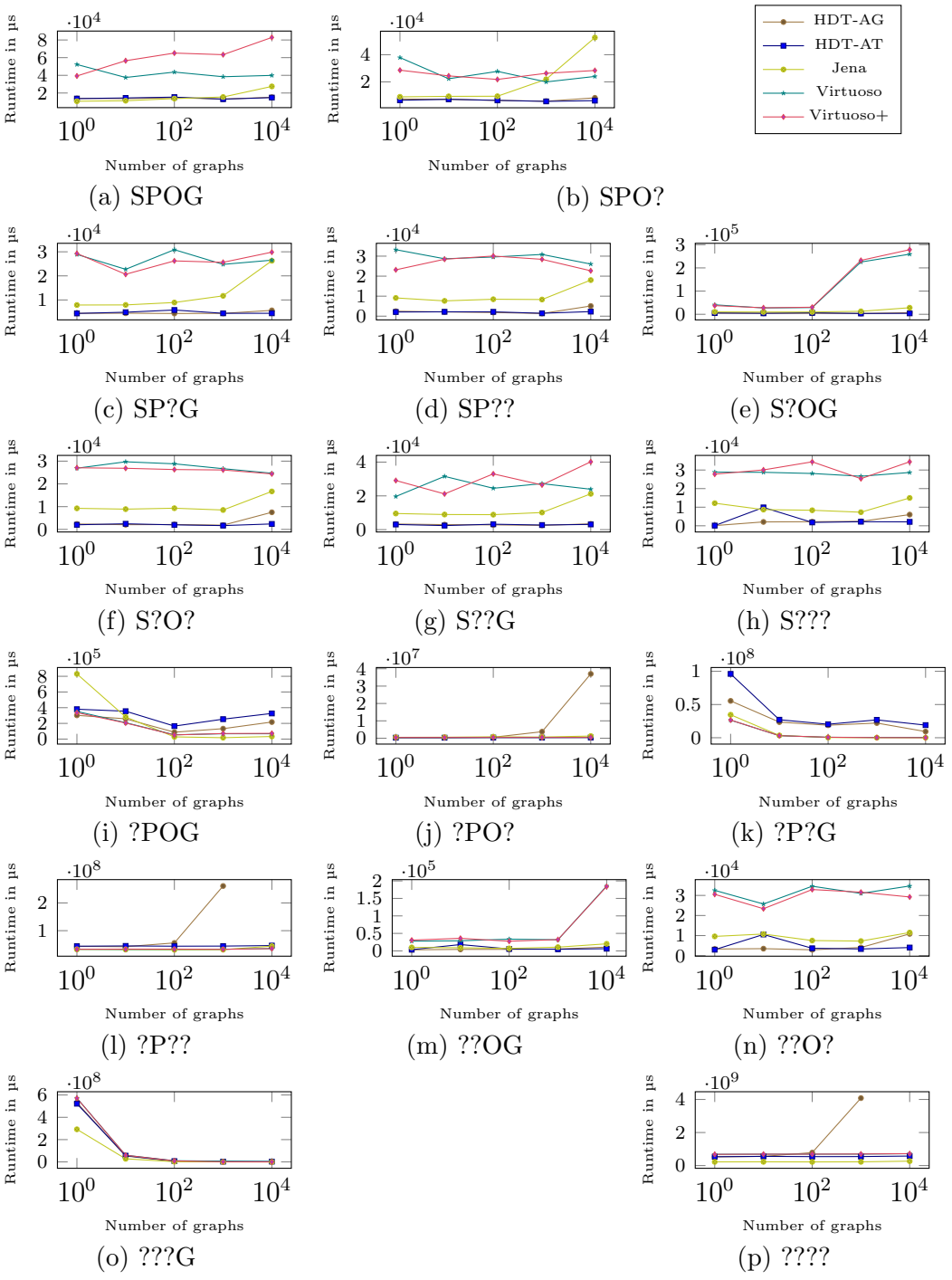


Figure 36: LUBM500 quad pattern resolution speed cold. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

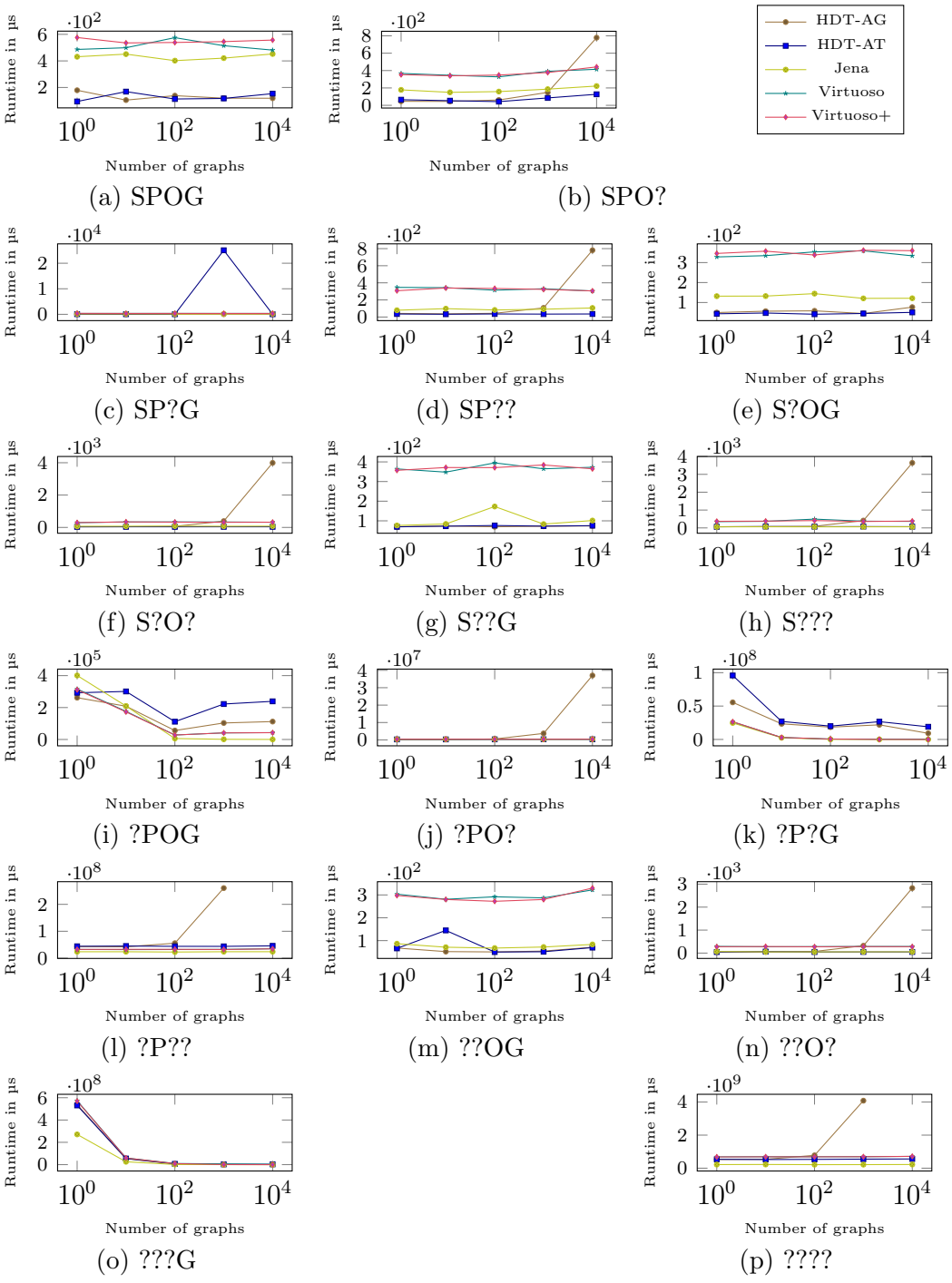


Figure 37: LUBM500 quad pattern resolution speed warm. A k number above the x-axis means that HDT-AG/HDT-AT is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG/HDT-AT.

For the last pattern that is resolved with the subject index by HDT, ????, all systems, except HDT-AG show a constant query time. HDT-AG starts to perform bad for 1,000 graphs and does not finish in a reasonable timeframe for 10,000 graphs.

Overall, for the patterns that have the subject given, HDT-AG and HDT-AT perform very well for the LUBM500 datasets. Both are faster than Virtuoso and Virtuoso+ in all cases and, except for a few cases for SPOG and S??? also faster than Jena.

Looking at the results for the patterns for which HDTQ uses the predicate index (i.e. ?P?? and ?P?G), one can see that for ?P?? HDT-AT, Jena, Virtuoso and Virtuoso+ are unaffected by the number of graphs. HDT-AG, however, performs basically equally for 1, 10 and 100 graphs, but for 1,000 graphs the performance vastly deteriorates. For 10,000 graphs the system did not even find the result in reasonable time.

For the second pattern the results are also interesting. With an increasing number of graphs all systems become faster. This is most likely the result of the decreasing number of results for the queries. The number of results decreases because the number of quads stays the same, but the number of graphs increases, which means that the quads are distributed over a greater number of graphs. As in the ?P?G pattern the graph component is given, with a greater number of graphs in the dataset, the number of results for a given query shrinks.

Overall HDT-AT performs good compared to the other systems for ?P??, but not so well for ?P?G, HDT-AG's performance for ?P?? is very bad and for ?P?G it is better than the one of HDT-AT but still does not reach the speed of the other compared systems.

The remaining patterns are those for which HDTQ uses its object index (?POG, ?PO?, ??OG and ??O?). For ?POG a similar trend can be seen as for ?P?G. Systems become faster with a higher number of graphs, however HDT-AG and HDT-AT become slower again, starting with 1,000 graphs. For ?PO? HDT-AG's performance is stable in the beginning, but starting with 1,000 graphs the performance gets worse and finally becomes very poor for 10,000 graphs. The other systems have a constant speed for an increasing number of graphs.

For ??OG HDT-AG, HDT-AT and Jena have a constant performance. Virtuoso and Virtuoso+ become significantly slower for 10,000 graphs. For ??O? Jena and HDT-AT have a constant execution time (even though HDT-AT has a little outlier for 10 graphs). HDT-AG starts off very good, but for 10,000 graphs the performance starts to become worse. Virtuoso and Virtuoso+ show a varying speed for different numbers of graphs.

Overall HDT-AT performs very well for this last group of patterns. Only

for ?POG the other systems perform better. HDT-AG is also quite good most of the time, but its performance for ?PO? is pretty poor. Jena shows satisfactory results and Virtuoso and Virtuoso+ perform good for ?POG and ?PO?, but not so good for the other two patterns.

To see how the systems perform with a varying number of graphs in a warm state, Figure 37 shows the warm results. If the subject is not given, but the predicate is given and if searching for all quads (with or without bounded graph), no difference to the cold tests can be observed. Moreover, the systems are for SPOG and SPO? as well as SP?G and S??G similar stable to the cold state, but about a factor of 100 faster. What is more, if the graph is not given, but the subject is, or the subject and the predicate are, or the subject and the object are, or only the object is given, HDT-AG starts to perform poor with 10,000 graphs in a warm state. Another effect for these patterns is that the systems resolve them significantly faster than in a cold state. For S?OG and ??OG the systems become more stable and significantly faster.

7.8 Discussion

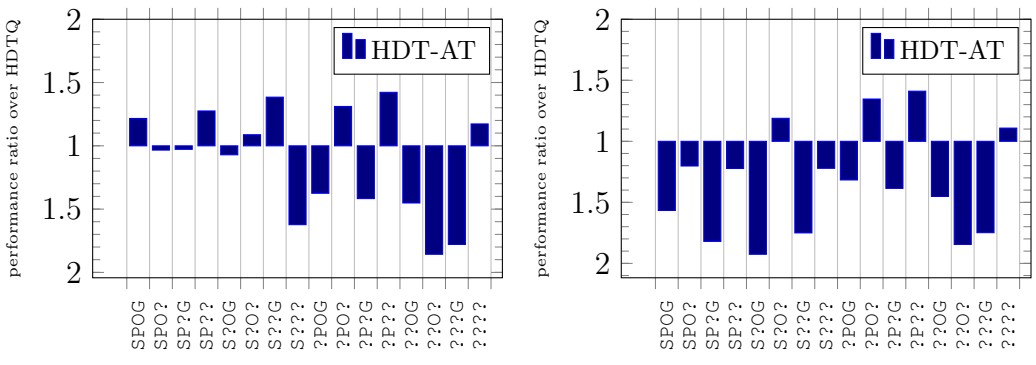
The tests described in this section have shown that HDTQ is extremely powerful for compressing RDF data with a low number of triples and graphs, such as in a versioning scenario. Also, with its very fast querying speed HDTQ outperforms its competitors by far for such RDF data.

However, the current implementation shows weaknesses when it comes to processing of very large RDF data with a high number of graphs and triples. While the querying speed of HDT-AT is still competitive, HDT-AG becomes unusable with about 400,000 graphs and 2 million triples. Yet, the most severe issue is that both approaches show a very poor compression ratio for such RDF graphs.

One way to address this problem is to use compressed bitmaps instead of plain ones. Especially if a lot of consecutive symbols (0s or 1s) appear in the

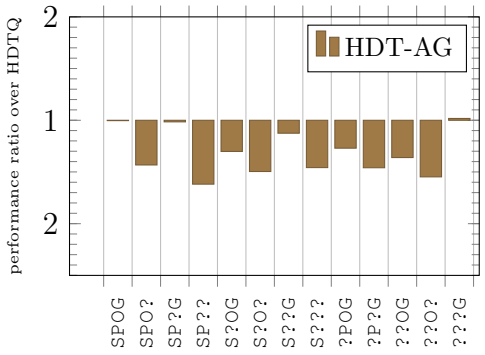
		Size (GB)	gzip	HDT-AG	HDT-AT	Jena	Virtuoso	Virtuoso+	HDT-AG (C)	HDT-AT (C)
BEAR	A	396.85	5.82%	**2.33**%	2.75%	96.84%	NA	NA	NA	NA
	B day	0.64	4.83%	**0.65**%	0.71%	97.73%	13.69%	33.75%	0.67%	0.76%
	B hour	9.66	4.83%	0.33%	0.25%	96.39%	4.35%	25.62%	0.31%	**0.14**%
LUBM	500 (1 graph)	11.42	3.04%	6.71%	11%	118.76%	17.23%	21%	**6.64**%	16.98%
	500 (9998 graphs)	11.61	3.02%	675.58%	345.92%	120.1%	17.46%	27.51%	**6.57**%	16.74%
	1000 (1 graph)	22.84	3.04%	6.9%	11.18%	118.67%	16.23%	19.98%	**6.83**%	17.16%
	1000 (7000 graphs)	23.21	3.02%	474.85%	236.44%	119.47%	16.38%	22.65%	**6.75**%	16.91%
LDBC		0.92	9.7%	12111.47%	6081.25%	126.28%	71.2%	80.77%	**15.91**%	25.06%
Liddi		0.67	3.74%	13254.03%	6637.71%	78.06%	49.88%	53.36%	**11.79**%	15.56%

Table 9: Space requirements of different systems including HDTQ using compressed bitmaps.

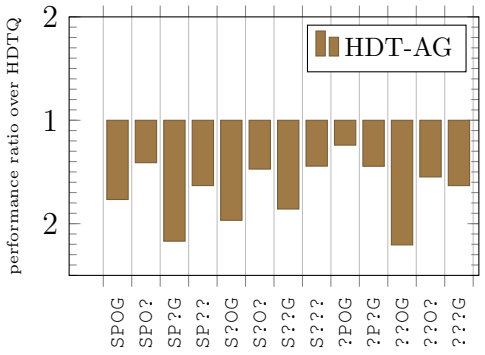


(a) Cold compared to HDT-AT (C)

(b) Warm compared to HDT-AT (C)



(c) Cold compared to HDT-AG (C)



(d) Warm compared to HDT-AG (C)

Figure 38: Liddi quad pattern resolution speed for HDTQ using compressed bitmaps. A k number above the x-axis means that HDT-AG (C) / HDT-AT (C) is k times faster than the compared system. A k number below shows that the system is k times faster than HDT-AG (C) / HDT-AT (C).

graph information matrix, this can reduce the required space significantly. A brief performance test (results are depicted in Table 9) shows that using compressed bitmaps [95] reduces the required space tremendously (in the table, the versions using compressed bitmaps are marked with (C)). Using compressed bitmaps, HDTQ outperforms Jena, Virtuoso and Virtuoso+ for all datasets under review (no statement can be made for BEAR-A as the dataset could not be imported into neither HDTQ (C) within 24 hours). For the Liddi dataset HDT-AG using plain bitmaps showed the worst results (about 13,000% of the original size), for the same dataset, HDT-AG using compressed bitmaps reduces the required space to only about 12%.

To see whether this significant improvement comes with major losses regarding the query resolution times, Figure 38 compares the performance of

both HDTQ approaches using plain bitmaps against both approaches using compressed bitmaps for the Liddi dataset. As can be seen, no query is more than 2.3 times slower using compressed bitmaps. Some of the queries (mostly for HDT-AT) are even resolved faster using the compressed bitmaps. The patterns which could not be resolved by HDT-AG in a reasonable time frame using plain bitmaps, could neither be resolved by HDT-AG using compressed bitmaps and are therefore not included in the figure. As HDT-AT using plain bitmaps significantly outperformed the other systems for the Liddi dataset (especially cold, but for most cases also warm), it is an acceptable trade off to reduce the required space by 99.91% in exchange for a reduction of querying speed of at most factor 2.

8 Conclusion

This section summarizes the main findings of this work and gives an outlook for future research.

8.1 Summary

The research question of this work deals with the matter of how compressed RDF formats like HDT can be extended to handle quad information while keeping its compact and queryable features. Based on the idea of the timestamp based RDF versioning approach, two different approaches have been developed. These two approaches are Annotated Triples (AT) and Annotated Graphs (AG). Using the AT approach, for each triple a bitmap is stored that indicates in which graphs the triple is present, while using the AG approach, for each graph a bitmap is stored that indicates which triples belong to the graph.

HDT was extended by an additional graph information matrix containing graph information. Together with an extension of HDT's dictionary, HDTQ was formed. Efficient algorithms for the resolution of the 16 patterns consisting of subject, predicate, object and graph, have been introduced.

Of the two approaches, HDT-AT is better suited for queries in which the graph component is not provided, as only a single bitmap has to be queried for each found triple. By contrast, HDT-AG is better suited for queries in which the graph component is bound, as only one bitmap needs to be searched for the next result.

The Java implementation of HDT was extended with the features of HDTQ. This implementation was tested in a server environment on several datasets, differing in their size as well as in the number of graphs. In addi-

tion to the HDTQ implementation, Apache Jena and Virtuoso (and Virtuoso with an additional index) were tested and the performance was compared to HDTQ.

Regarding the space requirements, results revealed that HDTQ is a sound alternative to Jena and Virtuoso, especially for datasets containing only a low number (hundreds) of graphs, which is a candidate for most RDF versioning scenarios. When increasing the graph size to a few hundred, HDTQ looses its advantage against Virtuoso. For a few thousand graphs, HDTQ becomes less efficient than Jena. The degradation for large numbers of graphs is owing to the used bitmap implementation. An implementation with a higher compression could contribute to better results and is devoted to future work.

The results of the analysis of the creation times draw a similar picture. HDTQ is a good alternative for datasets with a small number of graphs. With an increasing number of graphs, HDTQ's performance decreases compared to the competing systems. Virtuoso is very fast while Jena's speed is similar to HDTQ.

We used the BEAR benchmark, a corpus of versioned datasets from the Linked Data Observatory and DBpedia dynamic pages, to test the performance of HDTQ with a reasonable number of graphs (89 to 1,299) and triples (82,000 to 167,000). Results show that HDTQ excels for this particular case. In a cold state HDTQ outperforms its competitors by far. While there are performance losses in a warm state, HDTQ is still superior. Also, there is no notable difference in the performance of HDT-AT and HDT-AG for the BEAR datasets.

Furthermore, we generated the LDBC dataset with the generator offered by the Linked Data Benchmark Council (LDBC) with 190,000 graphs and 5,000,000 triples. For this dataset, the performance of HDTQ is still acceptable but already damaged because of the high number of triples and graphs.

HDTQ shows a stable performance with an increasing number of graphs on the LUBM (Lehigh University Benchmark for OWL) dataset and even outperforms Jena and Virtuoso for most quad patterns. For datasets with a number of graphs exceeding 1,000, the execution time of HDT-AG surges for 3 of the search patterns (bounded predicate, bounded predicate and object and retrieving all quads).

In order to test the functionality range of HDTQ, we evaluated the system in the worst case scenario of having about 400,000 graphs and 2,000,000 triples. Results show that HDT-AT performs very well in a cold state and also shows a solid performance in a warm state. However, HDT-AG is effectively unusable for a dataset like this because of extraordinarily high RAM consumption.

All things considered, HDTQ has shown to be a sound alternative to its

competing systems. Particularly for the initial (cold) execution of queries, HDTQ performs especially well. Considering another implementation for the bitmaps could enable to match or even outperform other systems also for datasets with a high number of graphs.

8.2 Limitations and Future Research

As can be seen in Section 7 the Java implementation of HDTQ has been proven to be a competitive system for RDF graphs that have a relatively low number of graphs. However, starting with 1,000 to 10,000 graphs, the needed space for the graph information matrix becomes an issue.

To be specific, it is not only the number of graphs that contributes to this phenomenon, but the combination of a high number of graphs and a decent number of triples. In HDTQ the graph information matrix holds 1 bit for each triple - graph pair (also if the triple does not appear in that very graph). Thus, the total size in bits is the number of graphs multiplied by the number of triples, which quickly becomes a tremendously large number.

However, as it is very unlikely that triples appear in all graphs (or even close to all graphs), the information matrix is full of repetitive bits (in that case 0s). If, on the other hand, a graph contains a huge share of all triples, there also are a lot of repetitive bits (in that case 1s). Such repetitive patterns can, of course, be compressed well, reducing the needed space significantly.

The chosen bitmap implementation, nevertheless, does not compress the bitmaps well, leading to a large space requirement when storing the HDTQ files on disk, but also leading to a high RAM consumption. In a future work, one could choose a bitmap implementation that compresses the bitmaps more and thus making HDTQ even with a high number of graphs more competitive. After replacing the bitmaps with a more compressed version, the algorithms to resolve the quad patterns described in Section 5.2 can still be applied. As briefly outlined in Section 7.8 this already shows promising results.

Besides improving the bitmap implementation, one could also test the performance of HDTQ with even larger datasets. The BEAR-A dataset with more than 2 billion triples was compressed perfectly with HDTQ. However, unfortunately it is not possible to import the dataset into the Virtuoso system used. It would be interesting to see whether the query resolution speed of HDTQ is still competitive for datasets of that scale.

One limitation of HDTQ is that only simple quad pattern queries can be resolved. Thus, another challenge is to make HDTQ queryable using SPARQL queries. For classic HDT this is already done in the form of a Jena graph implementation that allows accessing HDT files as Jena models. These models can then be used with Jena ARQ to evaluate SPARQL queries.

References

[1] RDF 1.1 Primer. W3C Working Group Note 24 June 2014. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/, 2014. Accessed: 2017-06-06.

[2] RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation 25 February 2014. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, 2014. Accessed: 2017-05-16.

[3] Sarra Abbassi and Rim Faiz. RDF-4X: A Scalable Solution for RDF Quads Store in the Cloud. In *Proceedings of the 8th International Conference on Management of Digital EcoSystems*, pages 231–236. ACM, 2016.

[4] Jinhyun Ahn, Dong-Hyuk Im, Jae-Hong Eom, Nansu Zong, and Hong-Gee Kim. G-Diff: A Grouping Algorithm for RDF Change Detection on MapReduce. In *Joint International Semantic Technology Conference*, pages 230–235. Springer, 2014.

[5] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing Linked Datasets with the VoID Vocabulary. W3C Interest Group Note 03 March 2011. https://www.w3.org/TR/2011/NOTE-void-20110303/, 2011. Accessed: 2017-06-01.

[6] Sandra Álvarez-García, Nieves R Brisaboa, Javier D Fernández, and Miguel A Martínez-Prieto. Compressed k2-triples for Full-In-Memory RDF Engines. *arXiv preprint arXiv:1105.4004*, 2011.

[7] Anastasia Analyti and Ioannis Pachoulakis. A Survey on Models and Query Languages for Temporally Annotated RDF. *International Journal of Advanced Computer Science & Applications*, 1(3):28–35, 2008.

[8] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. " O'Reilly Media, Inc.", 2010.

[9] Phil Archer. Data Catalog Vocabulary (DCAT). https://www.w3.org/TR/2014/REC-vocab-dcat-20140116/, 2014. Accessed: 2017-06-13.

[10] Internet Archive. The Internet WayBack Machine Archive. https://archive.org/, 2017. Accessed: 2017-06-12.

[11] Medha Atre, Vineet Chaoji, Mohammed J Zaki, and James A Hendler. Matrix "Bit"loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *Proceedings of the 19th international conference on World wide web*, pages 41–50. ACM, 2010.

[12] Sören Auer and Jens Lehmann. What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. In *European Semantic Web Conference*, pages 503–517. Springer, 2007.

[13] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A Nucleus for a Web of Open Data. *The semantic web*, pages 722–735, 2007.

[14] Juan M. Banda, Tobias Kuhn, Nigam H. Shah, and Michel Dumontier. *Provenance-Centered Dataset of Drug-Drug Interactions*, pages 293–300. Springer International Publishing, Cham, 2015. ISBN 978-3-319-25010-6. doi: 10.1007/978-3-319-25010-6_18. URL https://doi.org/10.1007/978-3-319-25010-6_18.

[15] BBC. BBC Music. https://www.bbc.co.uk/music, 2017. Accessed: 2017-06-18.

[16] Christian Becker and Christian Bizer. DBpedia Mobile: A Location-Enabled Linked Data Browser. *Ldow*, 369:2008, 2008.

[17] Dave Beckett, Fabien Gandon, and Guus Schreiber. RDF 1.1 XML Syntax. W3C Recommendation 25 February 2014. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/, 2014. Accessed: 2017-05-15.

[18] Dave Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 Turtle-Terse RDF Triple Language. W3C Recommendation 25 February 2014. https://www.w3.org/TR/2014/REC-turtle-20140225/, 2016. Accessed: 2017-05-20.

[19] David Beckett. RDF 1.1 N-Triples. W3C Recommendation 25 February 2014. https://www.w3.org/TR/2014/REC-n-triples-20140225/, 2014. Accessed: 2017-05-20.

[20] T. Berners-Lee. Linked Data. https://www.w3.org/DesignIssues/LinkedData.html, 2009. Accessed: 2017-09-21.

[21] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. http://www.rfc-editor.org/info/rfc3986, 2005. Accessed: 2017-05-15.

[22] Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable RDF syntax. w3c Team Submission (Mar 2011). https://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/, 2011. Accessed: 2017-05-20.

[23] Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and Analyzing linked data on the Semantic Web. In *Proceedings of the 3rd international semantic web user interaction workshop*, volume 2006, page 159. Citeseer, 2006.

[24] Chris Bizer, Tom Heath, Danny Ayers, and Yves Raimond. Interlinking Open Data on the Web. In *Demonstrations track, 4th european semantic web conference, innsbruck, austria*, 2007.

[25] Christian Bizer. The Emerging Web of Linked Data. *IEEE intelligent systems*, 24(5), 2009.

[26] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227, 2009.

[27] John G Breslin, Stefan Decker, Andreas Harth, and Uldis Bojars. SIOC: an approach to connect web-based communities. *International Journal of Web Based Communities*, 2(2):133–142, 2006.

[28] Dan Brickley and Libby Miller. The FOAF Project. http://www.foaf-project.org/, 2017. Accessed: 2017-03-27.

[29] Gavin Carothers and Lex Machina Inc. RDF 1.1 N-Quads. W3C Recommendation 25 February 2014. https://www.w3.org/TR/2014/REC-n-quads-20140225/, 2014. Accessed: 2017-05-20.

[30] Jeremy J Carroll and Patrick Stickler. RDF Triples in XML. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 412–413. ACM, 2004.

[31] Ana Cerdeira-Pena, Antonio Farina, Javier D Fernández, and Miguel A Martínez-Prieto. Self-Indexing RDF Archives. In *Data Compression Conference (DCC), 2016*, pages 526–535. IEEE, 2016.

[32] Kristina Chodorow. *MongoDB: The Definitive Guide*. " O'Reilly Media, Inc.", 2013.

[33] 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*, pages 1216–1227. VLDB Endowment, 2005.

[34] David Clark. *Compact Pat Trees*. PhD thesis, PhD thesis, University of Waterloo, 1996.

[35] Philippe Cudré-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel Miranker, Juan F Sequeda, and Marcin Wylot. NoSQL Databases for RDF: An Empirical Evaluation. In *International Semantic Web Conference*, pages 310–325. Springer, 2013.

[36] DBpedia. DBpedia - Towards a Public Data Infrastructure for a Large, Multilingual, Semantic Knowledge Graph. `http://wiki.dbpedia.org/`, 2017. Accessed: 2017-06-13.

[37] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1): 107–113, 2008.

[38] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[39] Li Ding and Tim Finin. Characterizing the Semantic Web on the Web. In *International Semantic Web Conference*, pages 242–257. Springer, 2006.

[40] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). `http://www.rfc-editor.org/info/rfc3987`, 2005. Accessed: 2017-05-15.

[41] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[42] Ivan Ermilov, Jens Lehmann, Michael Martin, and Sören Auer. LODStats: The Data Web Census Dataset. In *Proceedings of 15th International Semantic Web Conference - Resources Track (ISWC'2016)*, 2016. URL `http://svn.aksw.org/papers/2016/ISWC_LODStats_Resource_Description/public.pdf`.

[43] Michael Färber, Carsten Menne, and Andreas Harth. A Linked Data wrapper for CrunchBase. *Semantic Web*, Preprint(Preprint):1–11, 2016.

[44] Lee Feigenbaum, Ivan Herman, Tonya Hongsermeier, Eric Neumann, and Susie Stephens. The Semantic Web in Action. `https://www.scientificamerican.com/article/semantic-web-in-actio/`, 2009. Accessed: 2017-03-27.

[45] Javier D Fernández, Claudio Gutierrez, and Miguel A Martínez-Prieto. RDF Compression: Basic Approaches. In *Proceedings of the 19th international conference on World wide web*, pages 1091–1092. ACM, 2010.

[46] Javier D Fernández, Miguel A Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF Representation for Publication and Exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41, 2013.

[47] Javier D Fernández, Axel Polleres, and Jürgen Umbrich. Towards Efficient Archiving of Dynamic Linked Open Data. In *DIACRON@ ESWC*, pages 34–49, 2015.

[48] Javier D Fernández, Jürgen Umbrich, Axel Polleres, and Magnus Knuth. Evaluating Query and Storage Strategies for RDF Archives. In *Proceedings of the 12th International Conference on Semantic Systems*, pages 41–48. ACM, 2016.

[49] Javier D Fernández, Jürgen Umbrich, Axel Polleres, and Magnus Knuth. Evaluating Query and Storage Strategies for RDF Archives. Semantic Web Journal. Under review. *Semantic Web Journal*, 2017. Available at `http://www.semantic-web-journal.net/content/evaluating-query-and-storage-strategies-rdf-archives`.

[50] The Apache Software Foundation. Apache Any23. `https://any23.apache.org/`, 2017. Accessed: 2017-08-07.

[51] The Apache Software Foundation. TDB Command-line Utilities. `https://jena.apache.org/documentation/tdb/commands.html#tdbloader2`, 2017. Accessed: 2017-08-17.

[52] The Apache Software Foundation. Jena architecture overview. `https://jena.apache.org/about_jena/architecture.html`, 2017. Accessed: 2017-05-20.

[53] The Apache Software Foundation. TDB architecture. https://jena.apache.org/documentation/tdb/architecture.html, 2017. Accessed: 2017-09-19.

[54] Shi Gao, Jiaqi Gu, and Carlo Zaniolo. RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases. In *Proc. of EDBT*, 2016.

[55] Manolis Gergatsoulis and Pantelis Lilis. Multidimensional RDF. *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 1188–1205, 2005.

[56] Frdrick Giasson. Ping the Semantic Web. http://pingthesemanticweb.com/, 2006. Accessed: 2017-06-17.

[57] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical Implementation of Rank and Select Queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[58] Fabio Grandi. T-SPARQL: A TSQL2-like Temporal Query Language for RDF. In *ADBIS (Local Proceedings)*, pages 21–30, 2010.

[59] Markus Graube, Stephan Hensel, and Leon Urbas. R43ples: Revisions for Triples. In *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)*, 2014.

[60] Paul Groth and Luc Moreau. PROV-Overview. An overview of the PROV Family of Documents. https://www.w3.org/TR/2013/NOTE-prov-overview-20130430/, 2013. Accessed: 2017-06-13.

[61] Paul Groth, Andrew Gibson, and Jan Velterop. The anatomy of a nanopublication. *Information Services & Use*, 30(1-2):51–56, 2010.

[62] The W3C SPARQL Working Group. SPARQL 1.1 Overview. W3C recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/.

[63] R. V. Guha, Dan Brickley, and Steve Macbeth. Schema.org: Evolution of Structured Data on the Web. *Commun. ACM*, 59(2):44–51, January 2016. ISSN 0001-0782. doi: 10.1145/2844544. URL http://doi.acm.org/10.1145/2844544.

[64] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158 – 182, 2005. ISSN 1570-8268. doi: http://dx.doi.org/10.1016/j.websem.2005.06.005. URL `http://www.sciencedirect.com/science/article/pii/S1570826805000132`. Selcted Papers from the International Semantic Web Conference, 2004.

[65] Claudio Gutierrez, Carlos A Hurtado, and Alejandro Vaisman. Introducing Time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2), 2007.

[66] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages. In *International Semantic Web Conference*, volume 3298, pages 502–517. Springer, 2004.

[67] Andreas Harth. Billion Triples Challenge data set. `http://km.aifb.kit.edu/projects/btc-2012/`, 2012. Accessed: 2017-06-02.

[68] Olaf Hartig and Olivier Curé. Semantic Data Management in Practice. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 901–904. International World Wide Web Conferences Steering Committee, 2017.

[69] Michael Hausenblas. Linked Data Applications. *First Community Draft, DERI*, 2009.

[70] Michael Hausenblas. Exploiting Linked Data to Build Web Applications. *IEEE Internet Computing*, 13(4), 2009.

[71] Michael Hausenblas, Wolfgang Halb, Yves Raimond, Lee Feigenbaum, and Danny Ayers. Scovo: Using Statistics on the Web of Data. *The Semantic Web: Research and Applications*, pages 708–722, 2009.

[72] Michael Hausenblas, Raphael Troncy, Yves Raimond, and Tobias Brger. Interlinking Multimedia: How to Apply Linked Data Principles to Multimedia Fragments. *Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09)*, 2009.

[73] Sandro Hawke, Ivan Herman, Phil Archer, and Eric Prud'hommeaux. W3C Semantic Web Activity. `https://www.w3.org/2001/sw/`, 2013. Accessed: 2017-05-16.

[74] Tom Heath and Enrico Motta. Revyu: Linking reviews and ratings into the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):266–273, 2008.

[75] Sebastian Hellmann, Claus Stadler, Jens Lehmann, and Sören Auer. DBpedia Live Extraction. *On the Move to Meaningful Internet Systems: OTM 2009*, pages 1209–1223, 2009.

[76] Florian Holzschuher and René Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.

[77] Paul Houle. rdfDiff. `https://github.com/paulhoule/infovore/wiki/rdfDiff`, 2013. Accessed: 2017-06-14.

[78] Philip Howard. Graph and RDF databases 2016. *Bloor Market Report*, 2016.

[79] Dong-Hyuk Im, Sang-Won Lee, and Hyoung-Joo Kim. A Version Management Framework for RDF Triple Stores. *International Journal of Software Engineering and Knowledge Engineering*, 22(01):85–106, 2012.

[80] Dong-Hyuk Im, Nansu Zong, Eung-Hee Kim, Seokchan Yun, and Hong-Gee Kim. A Hypergraph-based Storage Policy for RDF Version Management System. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 74:1–74:5, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1172-4. doi: 10.1145/2184751.2184840. URL `http://doi.acm.org/10.1145/2184751.2184840`.

[81] Crunchbase Inc. Crunchbase. `https://www.crunchbase.com`, 2017. Accessed: 2017-06-18.

[82] Aftab Iqbal, Oana-Elena Ureche, Michael Hausenblas, and Giovanni Tummarello. LD2SD: Linked Data Driven Software Development. *21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, 2009.

[83] JUnit. JUnit. `http://junit.org/junit4/`, 2017. Accessed: 2017-09-15.

[84] Tobias Käfer, Ahmed Abdelrahman, Jürgen Umbrich, Patrick OByrne, and Aidan Hogan. Observing Linked Data Dynamics. In *Extended Semantic Web Conference*, pages 213–227. Springer, 2013.

[85] Benedikt Kämpgen. *Flexible Integration and Efficient Analysis of Multidimensional Datasets from the Web*. KIT Scientific Publishing, 2015.

[86] Michel Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the Web. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 197–212. Springer, 2002.

[87] Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizer, and Robert Lee. Media Meets Semantic Web–How the BBC Uses DBpedia and Linked Data to Make Connections. *The semantic web: research and applications*, pages 723–737, 2009.

[88] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura. On Graph Deltas for Historical Queries. *CoRR*, abs/1302.5549, 2013. URL `http://arxiv.org/abs/1302.5549`.

[89] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[90] LDBC. Semantic Publishing Benchmark. `http://ldbcouncil.org/developer/spb`, 2017. Accessed: 2017-08-11.

[91] LDBC. Semantic Publishing Benchmark v2.0. `https://github.com/ldbc/ldbc_spb_bm_2.0`, 2017. Accessed: 2017-08-11.

[92] Danh Le-Phuoc, Josiane Xavier Parreira, Vinny Reynolds, and Manfred Hauswirth. RDF On the Go: An RDF Storage and Query Processor for Mobile Devices. In *Proceedings of the 2010 International Conference on Posters & Demonstrations Track-Volume 658*, pages 149–152. CEUR-WS. org, 2010.

[93] Neal Leavitt. Will NoSQL databases Live Up to Their Promise? *Computer*, 43(2), 2010.

[94] Vivian Lee, Masatomo Goto, Bo Hu, Aisha Naseer, Pierre-Yves Vandenbussche, Gofran Shakair, and Eduarda Mendes Rodrigues. Exploiting Linked Data in Financial Engineering. In *International Confer-*

ence on Informatics and Semiotics in Organisations, pages 116–125. Springer, 2014.

[95] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring Bitmaps: Implementation of an Optimized Software Library. *arXiv preprint arXiv:1709.07821*, 2017.

[96] Miguel A Martínez-Prieto, Javier D Fernández, and Rodrigo Cánovas. Querying RDF Dictionaries in Compressed Space. *ACM SIGAPP Applied Computing Review*, 12(2):64–77, 2012.

[97] Miguel A Martínez-Prieto, Mario Arias Gallego, and Javier D Fernández. Exchange and Consumption of Huge RDF Data. In *Extended Semantic Web Conference*, pages 437–452. Springer, 2012.

[98] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. In *Proceedings of the Second International Conference on Semantic Web-Volume 40*, pages 23–28. CEUR-WS. org, 2001.

[99] Brian McBride. Jena: A semantic Web toolkit. *IEEE Internet computing*, 6(6):55–59, 2002.

[100] George McDaniel. *IBM Dictionary of Computing*. McGraw-Hill New York etc., 1994.

[101] Viorel Milea, Flavius Frasincar, and Uzay Kaymak. Knowledge Engineering in a Temporal Semantic Web Context. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 65–74. IEEE, 2008.

[102] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *International Semantic Web Conference*, pages 423–435. Springer, 2002.

[103] Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Michael Smith. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C Recommendation 11 December 2012. `https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/`, 2012. Accessed: 2017-06-01.

[104] Gonzalo Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 570 pages.

[105] Thomas Neumann and Gerhard Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *Proceedings of the VLDB Endowment*, 3(1-2):256–263, 2010.

[106] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB JournalThe International Journal on Very Large Data Bases*, 19(1):91–113, 2010.

[107] Benjamin Nowack. CrunchBase Twitter Bot. `https://twitter.com/cbbot`, 2008. Accessed: 2017-06-18.

[108] Benjamin Nowack. ESWC 2009 Linked Data Dashboards. `http://bnode.org/blog/2009/06/04/eswc-2009-linked-data-dashboards`, 2009. Accessed: 2017-06-18.

[109] Natalya F Noy and Mark A Musen. Ontology Versioning in an Ontology Management Framework. *IEEE Intelligent Systems*, 19(4):6–13, 2004.

[110] Natalya Fridman Noy, Mark A Musen, et al. PromptDiff: A Fixed-Point Algorithm for Comparing Ontology Versions. *AAAI/IAAI*, 2002: 744–750, 2002.

[111] Natasha Noy, Alan Rector, Pat Hayes, and Chris Welty. Defining N-ary Relations on the Semantic Web. *W3C working group note*, 12(4), 2006. Accessed: 2017-06-16.

[112] Oracle. Oracle Spatial and Graph RDF Semantic Graph. `http://www.oracle.com/technetwork/database-options/spatialandgraph/overview/rdfsemantic-graph-1902016.html`, 2017. Accessed: 2017-09-22.

[113] Eyal Oren, Renaud Delbru, Michele Catasta, Richard Cyganiak, Holger Stenzhorn, and Giovanni Tummarello. Sindice.com: a document-oriented lookup index for open linked data. *International Journal of Metadata, Semantics and Ontologies*, 3(1):37–52, 2008.

[114] Vicky Papavasileiou, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides. High-Level Change Detection in RDF(S) KBs. *ACM Transactions on Database Systems (TODS)*, 38 (1):1, 2013.

[115] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of management information systems*, 24(3): 45–77, 2007.

[116] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009. ISSN 0362-5915. doi: 10.1145/1567274.1567278. URL `http://doi.acm.org/10.1145/1567274.1567278`.

[117] Matthew Perry, Prateek Jain, and Amit P Sheth. SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries. In *Geospatial semantics and the semantic web*, pages 61–86. Springer, 2011.

[118] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[119] Andrea Pugliese, Octavian Udrea, and VS Subrahmanian. Scaling RDF with Time. In *Proceedings of the 17th international conference on World Wide Web*, pages 605–614. ACM, 2008.

[120] Tore Risch, Vanja Josifovski, and Timour Katchaounov. Functional Data Integration in a Distributed Mediator System. In Peter M. D. Gray, Larry Kerschberg, Peter J. H. King, and Alexandra Poulovassilis, editors, *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*, pages 211–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-662-05372-0. doi: 10.1007/978-3-662-05372-0_9. URL `http://dx.doi.org/10.1007/978-3-662-05372-0_9`.

[121] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. " O'Reilly Media, Inc.", 2015.

[122] Anisa Rula, Matteo Palmonari, Andreas Harth, Steffen Stadtmüller, and Andrea Maurino. On the Diversity and Availability of Temporal Information in Linked Open Data. *The Semantic Web–ISWC 2012*, pages 492–507, 2012.

[123] Katherine Rushton. Number of smartphones tops one billion. *The telegraph*, 17, 2012. Accessed: 2017-08-20.

[124] MV Sande, P Colpaert, R Verborgh, S Coppens, E Mannens, and RV de Walle. R&Wbase: Git for triples. *LDOW*, 996, 2013.

[125] Andy Seaborne and Rob Vesse. RDF Patch Describing Changes to an RDF Dataset. http://afs.github.io/rdf-patch/, 2014. Accessed: 2017-06-14.

[126] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE intelligent systems*, 21(3):96–101, 2006.

[127] Vasil Slavov, Anas Katib, Praveen Rao, Srivenu Paturi, and Dinesh Barenkala. Fast Processing of SPARQL Queries on RDF Quadruples. *arXiv preprint arXiv:1506.01333*, 2015.

[128] Selver Softic and Michael Hausenblas. Towards Opinion Mining Through Tracing Discussions on the Web. In *The 7th International Semantic Web Conference. Citeseer*, page 79. Citeseer, 2008.

[129] OpenLink Software. Virtuoso RDF. https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSRDF, 2009. Accessed: 2017-05-22.

[130] OpenLink Software. OpenLink Data Spaces (ODS). http://vos.openlinksw.com/owiki/wiki/VOS/Ods, 2016. Accessed: 2017-06-18.

[131] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindstrm. JSON-LD 1.0. A JSON-based Serialization for Linked Data. W3C Recommendation 16 January 2014. https://www.w3.org/TR/2014/REC-json-ld-20140116/, 2014. Accessed: 2017-05-20.

[132] Kostas Stefanidis, Ioannis Chrysakis, and Giorgos Flouris. On Designing Archiving Policies for Evolving RDF Datasets on the Web. In *International Conference on Conceptual Modeling*, pages 43–56. Springer, 2014.

[133] Aaron Swartz. Musicbrainz: A Semantic Web Service. *IEEE Intelligent Systems*, 17(1):76–77, 2002.

[134] Jonas Tappolet and Abraham Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *European Semantic Web Conference*, pages 308–322. Springer, 2009.

[135] OpenLink Software Documentation Team. Performance Tuning Virtuoso for RDF Queries and Other Use. http://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFPerformanceTuning, 2017. Accessed: 2017-08-17.

[136] OpenLink Software Documentation Team. Bulk Loading RDF Source Files into one or more Graph IRIs. http://vos.openlinksw.com/owiki/wiki/VOS/VirtBulkRDFLoader, 2017. Accessed: 2017-08-17.

[137] Yannis Theoharis, Yannis Tzitzikas, Dimitris Kotzinos, and Vassilis Christophides. On graph Features of Semantic Web Schemas. *IEEE Transactions on Knowledge and Data Engineering*, 20(5):692–702, 2008.

[138] Yannis Tzitzikas, Yannis Theoharis, and Dimitris Andreou. On Storage Policies for Semantic Web Repositories that Support Versioning. *The Semantic Web: Research and Applications*, pages 705–719, 2008.

[139] Octavian Udrea, Diego Reforgiato Recupero, and VS Subrahmanian. Annotated RDF. *ACM Transactions on Computational Logic (TOCL)*, 11(2):10, 2010.

[140] Jürgen Umbrich. The Dynamic Linked Data Observatory. http://swse.deri.org/dyldo/, 2013. Accessed: 2017-06-13.

[141] Jürgen Umbrich, Stefan Decker, Michael Hausenblas, Axel Polleres, and Aidan Hogan. Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources. *3rd International Workshop on Linked Data on the Web (LDOW2010), in conjunction with 19th International World Wide Web Conference*, 2010.

[142] Jacopo Urbani, Jason Maassen, and Henri Bal. Massive Semantic Web data compression with MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 795–802. ACM, 2010.

[143] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37–38:184–206, March 2016. ISSN 1570-8268. doi: doi:10.1016/j.websem.2016.03.003. URL http://linkeddatafragments.org/publications/jws2016.pdf.

[144] Max Völkel, Wolf Winkler, York Sure, S Ryszard Kruk, and Marcin Synak. SemVersion: A Versioning System for RDF and Ontologies. In *Proc. of ESWC*, 2005.

[145] Christian Weiske and Sören Auer. Implementing SPARQL Support for Relational Databases and Possible Enhancements. In *CSSW*, pages 69–80, 2007.

[146] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[147] Chris Welty, Richard Fikes, and Selene Makarios. A Reusable Ontology for Fluents in OWL. In *FOIS*, volume 150, pages 226–236, 2006.

[148] Marc Wick. About Geonames. `http://www.geonames.org/about.html`, 2017. Accessed: 2017-06-17.

[149] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, pages 120–139. CEUR-WS. org, 2003.

[150] Fouad Zablith, Grigoris Antoniou, Mathieu d'Aquin, Giorgos Flouris, Haridimos Kondylakis, Enrico Motta, Dimitris Plexousakis, and Marta Sabou. Ontology evolution: a process-centric survey. *The knowledge engineering review*, 30(01):45–75, 2015.

[151] Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On Computing Deltas of RDF/S Knowledge Bases. *ACM Transactions on the Web (TWEB)*, 5(3):14, 2011.

[152] Jun Zhao, Alistair Miles, Graham Klyne, and David Shotton. Linked data and provenance in biological data webs. *Briefings in Bioinformatics*, 10(2):139, 2009. doi: 10.1093/bib/bbn044. URL `http://dx.doi.org/10.1093/bib/bbn044`.

[153] Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia. A General Framework for Representing, Reasoning and Querying with Annotated Semantic Web Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 11:72–95, 2012.