UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

A COMPACT AND DYNAMIC CACHING SYSTEM FOR RDF GRAPH DATABASES

TESIS PARA OPTAR AL MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN Y MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

CRISTÓBAL OSVALDO MIRANDA TORRES

PROFESORES GUÍA:
Gonzalo Navarro Badino
Aidan Hogan

PROFESOR CO-GUÍA:
Diego Arroyuelo Billiardi

MIEMBROS DE LA COMISIÓN:
Éric Tanter
Eduardo Godoy Vega
Renzo Anglés Rojas

SANTIAGO DE CHILE
2024

RESUMEN DE LA TESIS PARA OPTAR AL
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN
Y MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN
POR: CRISTÓBAL OSVALDO MIRANDA TORRES
FECHA: 2024
PROFESORES GUÍA:
Gonzalo Navarro Badino, Aidan Hogan
PROFESOR CO-GUÍA:
Diego Arroyuelo Billiardi

UN SISTEMA DE CACHING DINÁMICO Y COMPACTO PARA BASES DE DATOS DE GRAFOS RDF

En este trabajo estudiamos la aplicación de un k^2-tree dinámico y compacto como índice para un sistema de caching de bases de datos RDF. A diferencia de cualquiera de los otros sistemas existentes, este le da al motor principal de bases de datos resultados parciales, que pueden ser obtenidos desde memoria en vez de disco y conseguir mejores tiempos que los B+trees del motor principal cuando se tienen que hacer muchos accesos aleatorios en disco. Esta configuración tiene como objetivo reducir los tiempos de respuesta de consultas SPARQL, que son comunes en sistemas de bases de datos RDF y más aun para datasets tan grandes como Wikidata, para los cuales la mayoría de los sistemas se enfrentan con problemas al ejecutar incluso algunas consultas simples.

RESUMEN DE LA TESIS PARA OPTAR AL
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN
Y MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN
POR: CRISTÓBAL OSVALDO MIRANDA TORRES
FECHA: 2024
PROFESORES GUÍA:
Gonzalo Navarro Badino, Aidan Hogan
PROFESOR CO-GUÍA:
Diego Arroyuelo Billiardi

A COMPACT AND DYNAMIC CACHING SYSTEM FOR RDF GRAPH DATABASES

In this work, we study the application of a dynamic and compact k^2-tree as an index for a caching system of RDF databases. Unlike any other existing caching system, this one feeds the main engine with partial results that can be retrieved from memory instead of disk and achieve better times than the B+trees from the main engine when they have to make too many random disk accesses. This setup aims to reduce SPARQL query response times that are typical in RDF database systems and more so with massive datasets such as Wikidata for which most systems have trouble running even some simple queries.

A mi madre Gladys y padre Segundo.

Acknowledgments

Primero agradezco a mis profesores guía, Gonzalo y Diego por el apoyo en las estructuras de datos e ideas clave y Aidan que me ayudó a diseñar el sistema en varias partes importantes y con todo el tema relacionado a bases de datos, además de ser de gran ayuda en la etapa final para poder concluir el trabajo. También les agradezco a todos su paciencia y buena disposición.

Agradezco a mis amigos y familia por ser un apoyo moral durante este proceso que fue muy esencial para llegar al punto de darle término.

Table of Contents

1. Introduction

Why would one choose a graph database instead of a relational one? That is a question to which you either may already know the answer. Maybe never questioned yourself about it. Nonetheless, it is something to always keep in mind, with more reason if you work with graph databases. Otherwise, one might have a feeling of constant uneasiness, which mostly comes from the thought of relational databases. Their time-proven effectiveness makes one wonder, why didn't we choose a relational data model instead?

Graph databases thrive for inter-connected data [1], such as social graphs where nodes are people, and edges are some relationship between those people (e.g., friendship). Other examples include web graphs, where nodes are web pages, and edges are links referencing other pages[2]. When having this kind of data, it feels unnatural to work with relational databases. We want to perform path queries in graphs, which have an indefinite number of joins. Graph database systems offer a simple syntax for those queries, for example in the SPARQL language[3], whereas SQL was not designed with that in mind. It then becomes burdensome doing those queries in SQL, which require advanced features of the query language that not all engines have implemented and can lead to verbose and hard to interpret expressions, for example Postgres [4], a very popular object-relational database management system (ORDBMS), in its extended SQL language.

Within the realm of graph databases, there are plenty of options. Some of those are for general use and others for a more specific intent. In this work, we focus on a somewhat specialized data model for graph databases, the Resource Description Framework (RDF). RDF was created to fulfill the Semantic Web's purpose of providing machine-to-machine communication tools, complementing the original Web's objective of allowing humans to interact with it.

RDF gives a flexible way to model data without predetermined schemas. For example, Wikidata uses RDF to model relationships among a massive amount of entities with data intended to complement Wikidata. Also, researchers in some areas of biology, such as genetics, prefer RDF because of this and its simplicity.

Furthermore, RDF can be extended by specifications like RDF Schema [5], enriching the semantic description of the data, allowing for the data to be "reasoned" over machines.

RDF Stores have challenges in terms of storage and querying. The one that we address in this work is the slow response times of queries, which can occur due to high amounts of data, for example in databases such as Wikidata and DBPedia where we can have billions of RDF triples. Another factor for this is the complexity of the queries one might want to evaluate over these data stores. It tends to be that many interesting queries tend to be highly complex. The fact that query services like Wikidata receive millions of queries per day [6] make the resource requirements more challenging.

There are seemingly two general ways to approach this problem of scale. One of them is to

convert the system into a distributed system, which has the disadvantage that it might bring more challenges, or in a bad scenario amplify the main issues, for example, when machines need to coordinate and share lots of data over the network to evaluate queries. The other way is to improve upon how fast we retrieve the data from its source. The course of action in this thesis is part of the second category. These two methods are by no means exclusive to each other. In practice, hybrid approaches tend to win over single-strategy ones.

Several works explore ways to represent RDF stores, mostly in secondary memory [7]. Here, we consider using an in-memory cache, auxiliary to the primary database, that at any given time holds all the relations of the most relevant predicates in the RDF store. The relevance of the predicates can be configured according to some criteria. In this work, we use the Least Recently Used (LRU) strategy.

This thesis proposes an approach of indexing data within a cache for RDF databases based on a relatively new data structure, a compact and dynamic k^2-tree. We also explore the exclusive use of these trees as indexes for an in-memory database, for which we perform query evaluation completely in RAM.

1.1. Objectives

Our first objective is to study the use of a compact data structure in a cache of RDF databases.

Second, we want to prove that the cache can improve query response times.

Third, would be to show that apart from allowing queries, we can also support updates to the cache structure.

1.2. Research questions

Our first research question would be, what is the impact of using a dynamic and compact k^2-tree as a cache for RDF databases?

Secondly, can we improve query response times by using a dynamic and compact k^2-tree as cache?

And last, is it possible to also support updates to the structure, without impacting performance significantly?

2. Background

In this chapter, we explore the most important topics from which this works builds upon.

First, we describe some of the state-of-the-art caching mechanisms for RDF/SPARQL systems.

Second, we discuss the Semantic Web, which gave birth to many technologies like RDF Databases, aiming to fulfill its vision.

Then, we go into more detail about the Resource Description Framework model, from which this work greatly depends, as it provides the unit structure to build indices like the one we study in this work.

After that, we delve a bit onto SPARQL, the query language that the users have to write to request some data from RDF databases.

Having considered these technologies, we focus then on the public dataset used in this work, Wikidata, which holds an immense number of RDF triples.

Lastly, we introduce the dynamic and compact k^2-tree data structure used in this work, we explain, at a high level, how it works and what it looks like.

2.1. State of the art

In the literature we can find some works of caching systems that are specialized for RDF/SPARQL databases.

2.1.1. Caching of intermediate results for Distributed Hash Table RDF Stores

Battré [8] worked on one of the first caching systems for RDF databases. Specifically, on the caching of intermediate results to mitigate the effect of too many networking messages in Distributed Hash Table (DHT) RDF stores, which can be very expensive in query processing.

This method doesn't directly compare to this work, because here we don't use a distributed database and we don't cache results either, but indexes. In other words, this falls into a different category of caching systems.

However, some of its benefits include optimized cache utilization by giving precedence to popular triple patterns when looking for cache hits and also reducing the network overhead. Furthermore, the caching of intermediate results can help in avoiding to compute repeatedly some of the most recurrent sub-queries among several different queries.

One of the drawbacks of the solution from Battré or any caching of results solution is that it invokes the graph isomorphism problem, which in this scenario means that different equivalent queries will yield the same result, so it is desirable that the caching algorithm can

identify isomorphic queries, with the purpose of saving resources, but the graph isomorphism problem is NP. In the solution from Battré there is a normalization by variable renaming, but it doesn't address isomorphic queries more broadly than that.

Another issue with the solution is that it relies too much on a greedy heuristic which may or may not yield good results often. Also, as briefly mentioned above, the caching mechanism itself needs several network messages for coordination, which can contribute a lot to the costs of the processing.

2.1.2. Improving the performance of semantic web applications with SPARQL query caching

Martin et al. [9] introduced a cache of query results that depends on the assumption that most of the data in the RDF datasets remains unchanged while there are a few updates during a short timeframe. It relies on invalidating cache results when there are updates that make the cached results no longer correct.

Some of its benefits include being able to work with most RDF databases, because it is implemented on a proxy that processes queries before they are sent to a RDF database and in this proxy layer it decides whether or not to execute the queries according to its caching algorithm.

It is also smart enough to reuse a same cache result for different queries, instead of duplicating it. However, it neither considers equivalent queries after variable renaming nor attempts to find other equivalent queries.

One of its main selling points is the ability to cache application objects like full HTML pages, which is done in addition to caching query results. This comes with the promise to speed up much more the query responses.

2.1.3. Graph-Aware, Workload-Adaptive SPARQL Query Caching

Papailiou et al. [10] worked on a solution that involves canonical labelling, which attempts to cover the case of isomorph graphs when deciding what queries should be considered the same at the moment of retrieving a cache result.

The work also contemplates query planning in order to decide which cache results to use, because it can happen that there are several cache results that can be used in a single query, but some are more efficient than others. This is done by adapting a well known dynamic programming planner to the context of SPARQL/RDF.

The cache replacement strategy depends on a benefit estimation of the cache results based on the number of triples cached and the cost of the query. However, depending only on that can cause that less relevant results are held in the cache, while more popular results could use that space and therefore increase cache hits. To deal with that problem, the benefits are not set as a fixed property of a graph pattern, but as a changing property that can be modified during each selection of query patterns by the planner.

In general, this is a quite sophisticated caching solution for the category of results caching.

2.1.4. Caching and Prefetching Strategies for SPARQL Queries

Lorey et al. [11] explore the idea of caching results that will aid in the evaluation of subsequent queries, on the same assumption that there are not many updates in a short

timeframe.

One of the main characteristics of this work is that instead of only caching previous results, it attempts to predict what other results will be needed in the future. The way in which they predict other results to cache is by rewriting past queries into new ones that are expected to yield more relevant results. These prefetched results may or may not be used in future queries. This kind of caching is referred to as "semantic caching".

The way this work "predicts" results is by considering different methods of "query augmentation" that modify queries, such as adding more triple patterns to the query that include new variables, replacing some concrete subject, predicate or objects by variables, or by removing some patterns in a way that scope of results is increased.

Some of the costs in this solution are increased requirements of persistent storage and memory. Also, there is the possibility of data staleness that would cause outdated results and the requirement of recomputation more often, which can become very expensive.

Another possible disadvantage of the solution is incurring in too much cost of precomputing values while attempting to maximize cache hits.

2.1.5. Identifying and Caching Hot Triples for Efficient RDF Query Processing

Zhang et al. [12] studied the caching of results based on query hit rate and considered a frequency-based replacement mechanism and a forecasting method to assign cost to triples.

They use an offline algorithm that runs in the background of the main system to process queries found in the query log. Similarly to [11], the solution rewrites those queries and estimates frequency of the resulting triples. With the frequencies at hand they rank the triples and keep the top ranked triples in memory.

The solution is an improvement over [11]. It covers more types of queries and it also addresses cache replacement with their frequency-based algorithm. Unlike the other solution which only describes "semantic caching" as a way to select results based on a heuristic to rewrite queries, but without giving any indication on how to deal with capacity issues in any real world system.

One of the issues of the solution is the cost of the forecasting algorithm, which requires processing previous queries, rewrite them, and run them in a query endpoint. This needs to happen constantly to keep the data fresh.

2.1.6. This work

In this work, we take a completely different approach than the previously mentioned strategies, where instead of caching results, we index the full dataset in a compact and dynamic data structure that can hold in memory many RDF triples at once.

For each predicate in the dataset, there is a k^2-tree index that contains all the subject-object relationships with that predicate. These indices have a very low memory usage, because they use a specialized implementation of the k^2-tree that compacts branches by a using 4-bit representation for each node in the tree.

The replacement strategy runs each time there is a user query and it scans all the predicates in it to increase the hit count of the indices. Based on the least recently used strategy for replacement, it can retrieve or discard indices, having a maximum capacity defined beforehand. This replacement strategy contributes significantly to a lower memory usage.

One of the big concerns of this solution is that even if the data is in memory, it is not directly accessible, but it needs to be decompressed on the fly, which can reduce some CPU cache benefits during query processing that other structures with directly accessible data can have, such as B+trees.

Another issue is that if the indices need to contain a high number of triples, the index build time can be considerable. This is not an impediment for production scenarios, since an extra structure is considered in the work to support the live scenario and it is accompanied by a periodic full rebuild that can happen offline.

2.2. Semantic Web

The Semantic Web is an initiative aiming to revolutionize the way data is used on the Web, by allowing machines to interact with it not only for displaying purposes, but also reasoning about it to produce additional information hidden by the relationships among the data.

The Semantic Web was introduced into popularity by the famous 2001 paper with the same name by Tim Berners Lee, together with James Hendler and Ora Lassila [13]. Nevertheless, Tim Berners Lee had earlier published in 1998 his Semantic Web Road Map, a more technical description of what the Semantic Web would become [14].

The interface between any user exploring the web and the Semantic Web would be an agent, an entity knowledgeable about the Semantic Web which could navigate it easily to produce relevant information on behalf of the user. This agent is an abstract entity, meaning that it does not restrict how it should work exactly, just what it does. This property leaves it open for the community to create its agents. Those agents could be web services or bots.

A large amount of work has been done since then, from standards to software. The standards picked the Resource Description Framework (RDF) for data modeling [15], introduced with the Semantic Web. Also, the standard considered SPARQL as a query language for data manipulation [16]. This query language offers a natural way to retrieve RDF data with pattern-matching statements at its core called basic graph patterns (BGP), enriched with other higher-level operations, allowing the user to have highly expressive freedom for querying. We will cover RDF and SPARQL in the sections that follow.

Initially, there was no single consensus on how to describe semantics within the data, which is one of the main points of the Semantic Web. Instead, several alternatives were in exploration by the community and some of those are still in use today. Nonetheless, the W3C recommended the Web Ontology Language (OWL), and it seems that it is the most adopted language for semantics [17], followed by RDF Schema.

As for software, there are several alternatives developed for data management. For example, some of the most famous are Apache Jena, Blazegraph, Virtuoso, AllegroGraph, GraphDB, MarkLogic, and Oracle's triple-store.

2.3. Resource Description Framework

The Resource Description Framework (RDF) is a framework for representing information on the Web [18]. The W3C recommendation defines the abstract syntax for the framework and in the following will be summarized and in some technical parts paraphrased or textually cited. Similarly, we collected some portions from external RFC documents.

Its core unit is the RDF Triple, a tuple of three elements where each has a name, respectively, subject, predicate, and object.

```
Magento ProgrammedIn PHP .
Log4j ProgrammedIn Java .
"Star Wars: Republic Commando: Order 66" ProgrammedIn Java .
Vue.js  ProgrammedIn Javascript .
.NET ProgrammedIn C# .
Chromium  ProgrammedIn C++ .
```

Figure 2.1: RDF Example: Software programmed in a language (Labels)

```
wd:Q1884012  wdt:P277  wd:Q59 .
wd:Q286923  wdt:P277  wd:Q251 .
wd:Q55237 wdt:P277 wd:Q251 .
wd:Q24589705 wdt:P277 wd:Q2005 .
wd:Q21622213 wdt:P277 wd:Q2370 .
wd:Q48524 wdt:P277 wd:Q2407 .
```

Figure 2.2: RDF Example: Software programmed in a language (IRIs)

```
Chromium ProgrammedIn C++ .
Chromium InstanceOf WebBrowser .
Chromium FoundedBy Google .
Chromium Inception 2008 .
Chromium OperatingSystem GNU/Linux .
Chromium OperatingSystem BSD .
Chromium OperatingSystem Android .
Chromium OperatingSystem macOS .
Chromium OperatingSystem Windows .
Chromium SoftwareEngine V8 .
Chromium SourceURL https://chromium.googlesource.com/chromium/src .
Chromium SocialMediaFollowers 390797 .
Chromium CopyrightStatus Copyrighted .
```

Figure 2.3: RDF Example: Facts about Chromium (Labels)

```
Firefox InstanceOf WebBrowser.
Chromium InstanceOf WebBrowser .
"Internet Explorer 10" InstanceOf WebBrowser .
"Yandex Browser" InstanceOf WebBrowser .
Mosaic InstanceOf WebBrowser .
```

Figure 2.4: RDF Example: Web browsers (Labels)

A set of these triples can be viewed as a directed labeled graph, where subject and objects are the nodes and predicates are the labels. In this directed graph, a subject corresponds to the origin node of an edge. The edge is labeled with a predicate, and an object is the destination of this connection. With this representation, one node can be both a subject or an object for different triples. This modeling, allows terms to take the role of predicates in some triples to behave either as subject or object in different triples within the same graph.

For example, in figure 2.1 we have a set of RDF triples, where subjects are a software title, predicate is "ProgrammedIn" and objects are a programming language. In practice, IRIs are more commonly used for establishing these kinds of relationships and the same example would look like what we have in figure 2.2, but we use the labels for more clarity in the examples. The reason why IRIs are more used is because they work better as unique identifiers and support multiple languages. We will discuss IRIs in more detail in the following parts.

If we pick one software like Chromium in figure 2.3 and see what else can be found about it in the graph, we can see several other interesting facts that can be represented by RDF, like what type of software it is, who founded it, when was it created, what operating systems it supports, and so on.

Also, if we are interested in other web browsers, we can focus on subjects that have predicate InstanceOf and object WebBrowser, like we do in figure 2.4.

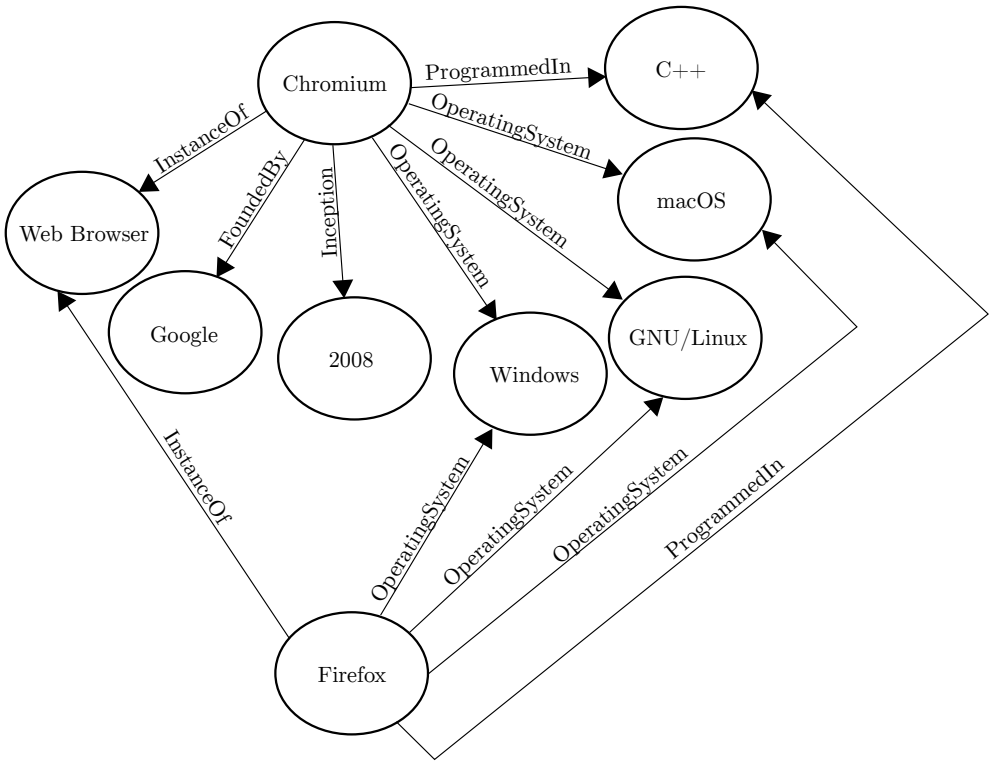

Figure 2.5: Web Browsers graph. Focused in Chromium, but it can be seen that it has some shared information with Firefox

In figure 2.5, we have a web browsers graph that can be built using RDF information.

There are restrictions on what can be put into an RDF triple. Those restrictions are the following:

- A subject can only be an IRI or a blank node.
- A predicate can only be an IRI.
- An object can only be an IRI, literal or a blank node.

IRIs and literals represent something in the world, called a resource, which can be anything physical or abstract. Blank nodes also represent something, but in an anonymous way, serving as an artifact to allow establishing relationships between other entities. These types of terms will be discussed in more detail in the following.

```
<https://www.wikidata.org/wiki/Q48524>
    <http://www.wikidata.org/prop/direct/P275> _:b1 .

_:b1 <http://www.w3.org/2000/01/rdf-schema#label> "BSD-3 and others"@en .
```

Figure 2.6: RDF example with IRIs, blanks and literals

2.3.1. IRI

Internationalized Resource Identifier, abbreviated IRI, is a protocol element that complements the URI element (Universal Resource Identifier) [19]. A URI serves its purpose well in the English language, but becomes insufficient for use with the other vast amount of languages that exist, as it does not support characters from other alphabets.

In this aspect, one might say that IRI extends URI because every URI is also an IRI and not the other way necessarily. But the RFC specification explicitly calls it a complement to emphasize that it is not a new version of URI, but a different protocol element. This is done to avoid any compatibility issues with systems already using URIs.

Both IRIs and URIs have the purpose of identifying resources or entities. More than this, it's desirable that every IRI is unique on a global basis for a single resource. Ultimately, this is in the hands of each system claiming to be using IRIs to comply with the norm.

For example, the IRI https://www.wikidata.org/wiki/Q48524 in figure 2.6 identifies Chromium.

2.3.2. Literal

Literals are values such as strings, numbers and dates with, optionally, a special syntax that enriches them with metadata specifying their type and language.

A literal term consists of three parts, of which the two first are required followed by an optional part. Those are, respectively, a lexical form, a datatype IRI, and a language tag. The datatype IRI, although it is required, can be omitted in an explicit representation of the literal, in which case we tacitly assume that it is a string, with its datatype IRI being specifically http://www.w3.org/2001/XMLSchema#string.

The language tag can be defined if and only if the datatype IRI is exactly http://www.w3.org/1999/02/22-rdf-syntax-ns#langString, in which case the datatype IRI can also be omitted in concrete representations if the language tag is present.

One example of a literal would be the label "BSD-3 and others"@en from the example in figure 2.6.

2.3.3. Blank

The W3C Recommendation establishes that, in the Concepts and Abstract Syntax document, blank nodes are local identifiers that depend on the concrete implementation of the RDF system, and it doesn't impose any rules on them except being disjoint from IRIs and literals. The document also mentions that new IRIs can replace blank nodes.

The situation that blank nodes can be anything creates a technical problem of manageability within systems. The W3C recommendation about the SPARQL query language comes to the rescue in this context, dictating that blank nodes should start with "_:" followed by some string. Also, SPARQL allows for using blank nodes implicitly as the term that completes some defined triple.

One example is _:b1 in figure 2.6, where it is used to refer to a copyright license having label "BSD-3 and others"@en. Note that this blank node doesn't necessarily exist, it was invented to serve as an example here.

2.4. SPARQL

SPARQL is a recursive acronym, meaning SPARQL Protocol And RDF Query Language. This language is the standard way of querying RDF databases and is also adopted in this work.

The W3C Recommendation document (https://www.w3.org/TR/rdf-sparql-query/) serves as a good reference and description of SPARQL. Also, other resources go much further in detail and theory for the interested reader [20] [21]. Here are highlighted the most relevant parts of the W3C Recommendation for the topic of this thesis.

In figure 2.7, there is a simple query that collects all the web browsers. Specifically, it will search for all RDF triples having as object <https://www.wikidata.org/wiki/Q6368>, as predicate <https://www.wikidata.org/wiki/Property:P31> and as object any other resource. For each of those matching triples, it will select its subject and put it into a table.

The ?browser is a *variable* that *binds* to each term matching the triple pattern in the dataset. Variables are always strings starting with a *?*.

In the same figure 2.7, we have some extra notation, wdt:P31 is a shortcut for <https://www.wikidata.org/wiki/Property:P31> and wd:Q6368 is a shortcut for <https://www.wikidata.org/wiki/Q6368>. We also have a SERVICE section, which instructs the query system to use labels in the English language. Thanks to this, the query system gives us the extra binding `?browserLabel`, having an English human-readable representation for `?browser`.

```
SELECT DISTINCT ?browserLabel WHERE {
  ?browser wdt:P31 wd:Q6368 .
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
  }
}
```

Figure 2.7: SPARQL query example: Find all distinct web browser names.

Table 2.1: Results table example

?browserLabel
SeaMonkey
Avant Browser
IBM WebExplorer
Windows Internet Explorer 7
...

After processing this query one would get a table as in table 2.1.

2.4.1. Basic Graph Pattern

```
SELECT DISTINCT ?browserLabel ?operatingSystemLabel WHERE {
  ?browser wdt:P31 wd:Q6368 . # web browsers
  ?browser wdt:P277 wd:Q2407 . # programmed in C++
  ?browser wdt:P306 ?operatingSystem . # find operating systems supported
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
  }
}
```

Figure 2.8: SPARQL query with three triples-BGP

A *triple pattern* is an extension of an RDF triple, where members of the triple can be variables, for example, the triple in the figure 2.7 is a triple pattern because it contains the variable `?browser`.

A *basic graph pattern*, abbreviated BGP, is a set of those triples, for example, the WHERE clause in the figure 2.8 denotes a BGP of three triple patterns.

A *binding* is a map from variables to RDF resources. A binding satisfies a BGP if applying the binding to the BGP gives a subgraph that exists in the dataset.

We can think about result tables as a representation of a multiset of bindings where their order might matter.

The query in figure 2.8 is asking to match all the bindings with variables {`?browser`, `?operatingSystem`} that satisfy the BGP and then project those into bindings with variables {`?browser`, `?operatingSystem`}. For the SELECT statement, we discard `?browser` and `?operatingSystem`, and instead use `?browserLabel` and `?operatingSystemLabel`, which are given by the SERVICE instruction.

As three triple patterns share the same variable `?browser`, the query evaluation has to perform an inner join over them at some point. If they weren't sharing a variable, the query evaluation would have to make a Cartesian product.

2.4.2. OPTIONAL

SPARQL allows to include values that might match, but if they aren't available, it won't affect the whole pattern, and they will be represented by null values if necessary. The OPTIONAL clause allows doing this. The syntax is shown in figure 2.9.

```
pattern OPTIONAL { pattern }
```

Figure 2.9: OPTIONAL syntax

The way to process this operation is by a left outer join between the patterns using their shared variables.

As an example, the query of figure 2.10 will return results as shown in the table 2.2.

```
SELECT DISTINCT ?browserLabel ?operatingSystemLabel ?inception WHERE {
  ?browser wdt:P31 wd:Q6368 .
  ?browser wdt:P277 wd:Q2407 .
  ?browser wdt:P306 ?operatingSystem .
  OPTIONAL {
    ?browser wdt:P571 ?inception .
  }
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
  }
}
```

Figure 2.10: Example of a SPARQL query with OPTIONAL clause

Table 2.2: Some results for optional query

browserLabel	**operatingSystemLabel**	**inception**
“SeaMonkey”	Gnu/Linux	30 January 2006
“Dooble”	Microsoft Windows	1 January 2009
“Opera”	macOS	10 April 1995

2.4.3. UNION

The UNION clause allows including alternatives from two patterns, where those two patterns will be evaluated and added to the results. The syntax is the following:

```
  pattern UNION pattern
```

Figure 2.11: UNION syntax

As an example, the query of figure 2.12 will return the table 2.3.

```
SELECT DISTINCT ?browserLabel ?operatingSystem WHERE {
  {
  BIND("GNU/Linux" AS ?operatingSystem)
  ?browser wdt:P31 wd:Q6368 . # Web browser
  ?browser wdt:P277 wd:Q2407 . # C++
  ?browser wdt:P306 wd:Q3251801 . # Linux
  }
  UNION
  {
  BIND("Windows" AS ?operatingSystem)
  ?browser wdt:P31 wd:Q6368 . # Web browser
  ?browser wdt:P277 wd:Q2407 . # C++
  ?browser wdt:P306 wd:Q1406 . # Windows
  }
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
  }
}
```

Figure 2.12: SPARQL query using UNION

Table 2.3: Results table example for UNION query

browserLabel	**operatingSystem**
Chromium	GNU/Linux
Chromium	Windows
Internet Explorer	Windows

Results from table 2.3 are formed from running two union parts separately and then concatenating them into a single table.

2.4.4. FILTER

A query result can be refined by the means of filters. These consist of several boolean functions which can use a wider variety of functions to test bindings values. When the test evaluates to TRUE the binding is accepted or discarded otherwise.

For example, in figure 2.13 we have a query that filters the results to only the ones that have inception with a year greater or equal than 2010.

```
SELECT DISTINCT ?browserLabel ?operatingSystemLabel ?inception WHERE {
  ?browser wdt:P31 wd:Q6368 .
  ?browser wdt:P277 wd:Q2407 .
  ?browser wdt:P306 ?operatingSystem .
  ?browser wdt:P571 ?inception .
  FILTER (YEAR(?inception) >= 2010)
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
  }
}
```

Figure 2.13: SPARQL query with a filter

2.5. Wikidata

In this section, we briefly describe the dataset used for experiments in this work.

Wikidata's dataset choice serves two purposes; one is to solve the scalability problem that Wikidata has and the other to show the usefulness of the tools developed here which could be utilized in similar contexts.

Wikidata is a community-driven central storage of data for its sibling projects, such as Wikipedia. This centralization allows Wikipedia to have a multilingual source of information that remains consistent in all of the languages it supports. The information held in this project pretends to be secondary or even tertiary, meaning that its purpose is collecting it from other more relevant sources instead of producing it. This makes it a service that will always contain objectionable information due to its openly collaborative nature, but valuable nonetheless for its easy accessibility [22].

Establishing the relationships between items is a fundamental issue for Wikidata, because manual analysis and management are infeasible in this case due to the huge amount of data.

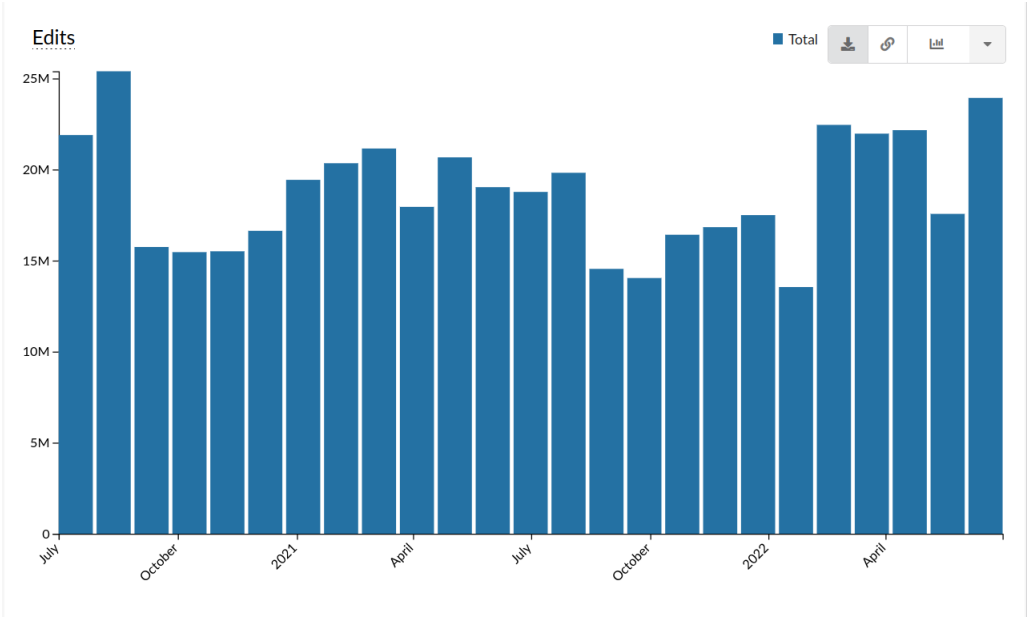

Figure 2.14: Wikidata edits from July 2020 to July 2022 [23]

In figure 2.14, we can see the number of edits between 2020 and 2022, which gives us an idea about what to expect on the frequency of updates, so we can build a system capable of supporting it.

Our chosen dataset is from the year 2021 and in a .nt file format, where each triple is in one line, and it has 5.7 billion triples. It takes 684 GB of disk space. In the Jena indexed format it takes 769 GB of disk space.

2.6. Compact and dynamic k^2-tree

The k^2-tree, also named quad-tree, is a data structure that allows representing a highly sparse matrix of zeros and ones in space proportional to the number of ones. The matrix in the figure 2.15 will serve to show how the k^2-tree can represent it.

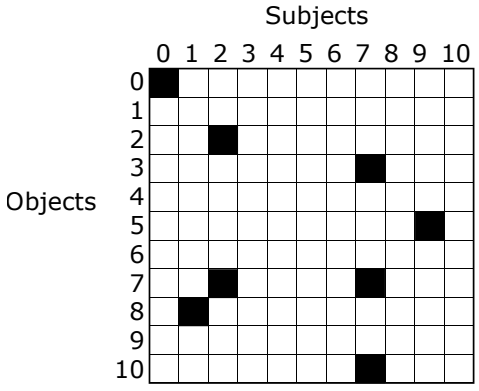

Figure 2.15: Example of a sparse matrix. Suppose black boxes are ones and white boxes are zeros.

Now consider figure 2.16. Each path from the root to a leaf describes a position on the matrix. If we take one of those paths and traverse it downwards to a leaf from the root, we

can notice that for every step, we are choosing a child based on the position of this point within the matrix. The act of picking a child, in terms of the represented matrix, is equivalent to choose one of the four equal-sized sub-regions of the matrix. We recursively continue doing this in the next steps with a smaller region until we find that the next region is of size 1×1, which means that we reached a leaf and deciphered the point.

For this to work as expected, the tree in question should be cardinal, meaning that each branch has to be labeled, and some tree branches might not go into a child. In this case, there are four branches, one per region. Let's describe regions as TL, BL, TR, BR (top-left, bottom-left, top-right, bottom-right) and assign them to a branch number, from now on, TL: 0, BL: 1, TR: 2, BR: 3. The codes within each node in figure 2.16 describe the path taken from the root until that point using the branch numbers for the regions selected. If the sides of the matrix aren't a power of two, we virtually pad them with zeros until the next power of two such that both sides are the same size. In the example figure, this padding would be until reaching a side size of 2^4.

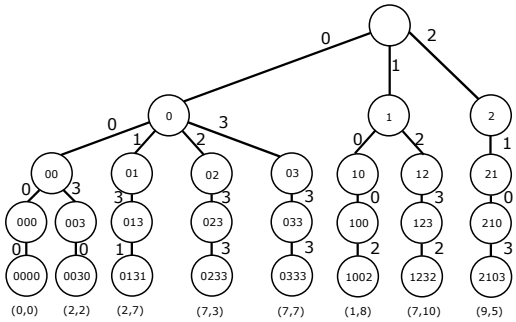

Figure 2.16: k^2-tree represention of the matrix in figure 2.15

2.6.1. Compact and dynamic k^2-tree

A compact representation of the k^2-tree allows the usage of less space and, at the same time, can be queried without having to decompress the complete structure.

Here we use an existing compact data structure [24], described as follows. Considering any node in the k^2-tree, we will represent it by which of the $\{0, 1, 2, 3\}$ children it has. Taking, for example, the root node in the figure 2.16, we can see that it has the children $\{0, 1, 2\}$ and not the child 3, therefore its 4-bit representation will be 1110. More generally, the i-th bit (i in $\{0, 1, 2, 3\}$) is turned on if and only if the considered node has an i-th child, counting from the leftmost bit to the rightmost.

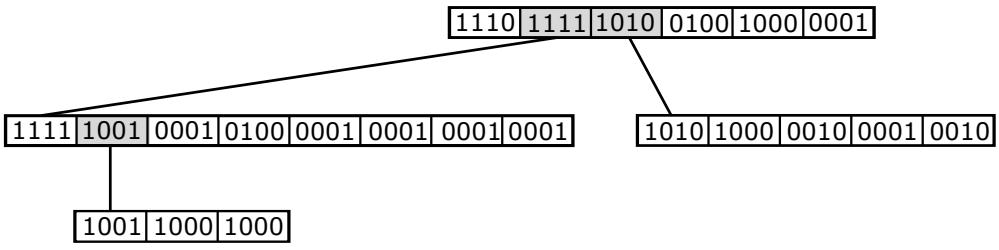

Figure 2.17: Compact representation of the k^2-tree with a blocks' tree

To organize nodes of the tree into bit sequences, we start by adding the root's four bits into the start of a resulting sequence. Then, we recursively add the four bits from its children

from the leftmost to the rightmost. We do this by adding the first child's subtree entirely before continuing with the other children. Viewed from another angle, following the resulting bit sequence from left to right sequentially is the same as traversing the k^2-tree in a pre-order depth-first-search traversal.

Having that, to perform visits to leaves in constant time, we introduce the notion of a tree of blocks, in which we denote blocks as segments of 4-bit nodes that will be the nodes in this tree of blocks. Each of these blocks might have zero or more children having as source one of the block's 4-bit nodes. We call those 4-bit source nodes frontier nodes and their children are other blocks within the tree of blocks.

To build this tree, we predetermine a maximum block size. When, at some point, we surpass that size, we perform a split algorithm that generates two new blocks. One is the parent block, which points via a new frontier node to the other block.

This tree of blocks is a dynamic structure because we can insert points to it or delete them without rebuilding it again, which is usual for other compact data structures that have to possess the data in advance before building them.

2.6.2. Practical Variation: Regular tree on top

The operations on the tree of blocks can be enhanced by using a regular tree with pointers on the first levels and starting from a given depth, switching to the tree of blocks, such that in this regular tree the leaves are actually our trees of blocks. This way, we are reducing bitwise operations by replacing a portion of blocks with pointer dereferencing, which tends to be faster than the former.

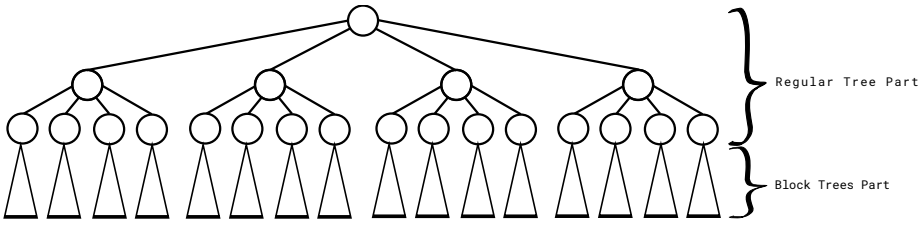

Figure 2.18: Mixed Tree variant with Regular Tree on top and blocks trees as leaves

3. K^2-Tree Serialization

One of the main ideas of this work is to use the previously described k^2-tree as storage for an in-memory index that can be used for a caching system. Each predicate can be represented together with all its (subject, object) pairs in a single k^2-tree. To be able to make operations with one k^2-tree, we need to have it loaded into memory. Then, it is possible to do things like scanning all objects for a given subject or the other way around, and scan all (subject, object) pairs for a given index. We can also make insertions and deletions for an input (subject, object) pair.

In this work, the mixed k^2-tree variant was enriched with some additional features to make it more usable for the purposes of the caching system. These include a serialization algorithm that allows us to store the index in persistent storage, so we don't have to wait for the index to be built more than once. Instead, we can load it from the serialized data once we have it. Also, lazy scanning algorithms were implemented, giving us the ability to serve results partially from the k^2-tree without having to materialize everything at once in memory. Furthermore, an alternative memory allocation strategy was conceived to overcome the big overhead that can be incurred by a high number of small allocations.

Building an index with massive amounts of data is expensive and we want to avoid doing it as much as possible. To achieve this, we serialize the data to store it in secondary memory and use it afterwards from there, instead of trying to build the index each time we need it. This way we only have to build the index from scratch once, in a preprocessing phase. Deserializing the index from disk is much less expensive than building it.

We are going to serialize the mixed tree with the regular tree on top, as that is the one we are going to be using in this work. Nonetheless, that tree and the block's tree serialization can be viewed independently, so let's begin with the block's tree serialization.

3.1. Block's Tree Serialization

For each block, we don't have to explicitly serialize the pointers to other blocks. This is based on the fact that we can infer which are the children of a given block by the number of children it has, the ordering in which we put the blocks and also that we are going to deserialize a tree fully always. This allows us to store these indices in less space than if we load them in primary memory, where we have to create the frontier pointers, but in consequence the deserialization is much faster because there is less data to scan from disk.

We recursively place blocks in a post-order sequence, that is, we will have all the children of a node to the left of their parent, which is followed by its right siblings. This way, the leftmost leaf would be placed first in the sequence and the root would be placed last.

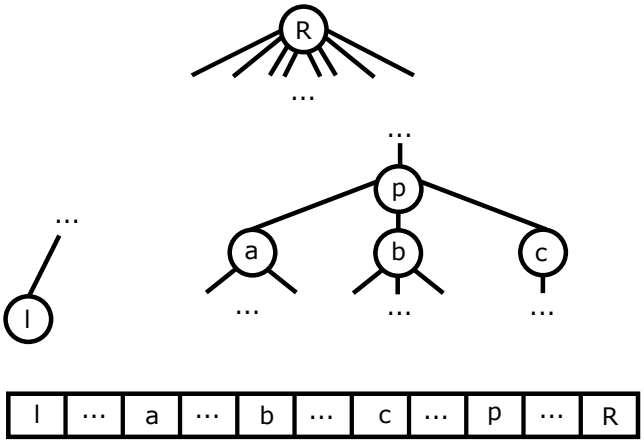

Figure 3.1: Block's tree serialization

In figure 3.1 we can see the tree of blocks on top and the serialization order below. The tree of blocks shown here is an abstraction to illustrate the serialization procedure, having in consideration that these nodes represent blocks such as the ones in figure 2.17, where the edges are the ones connecting frontier nodes (from the k^2-tree) to other blocks. There is the leftmost leaf l first and the root R last. Furthermore, there is an internal node p which is laid after all of its children subtrees, which are put from left to right in the same order they are as a child of p. Lastly, notice that in a tree of blocks, there is no defined number of children per node and the order among siblings does matter.

Nodes amount(N)	Children amount(C)	Container size (S)	Frontier preorders	Block topology
2 bytes	2 bytes	2 bytes	2C bytes	4S bytes

Figure 3.2: Block's serialization

As for the serialization of each block, we save three 2-byte integers first, the number of nodes, the number of children and the size of the block's container. Furthermore, we have to store the sequence of preorders corresponding to frontier nodes and the block's topology. This placement can be seen in figure 3.2.

When we want to deserialize the block's tree we can perform the following procedure:

- Begin by putting blocks from the sequence in figure 3.1 into a linked list, starting from the left. Each block has to be deserialized from a structure as shown in figure 3.2 from the sequence. The linked list will contain deserialized blocks.
- Each time a block is going to be put into the linked list, we check for its number of children. If that is greater than zero, then we consume that amount of children from the list starting from the rightmost one. This consumption adds those blocks as children of the one currently being processed in a reverse ordering.

When we are done with this, we will only have the root of the block's tree loaded into the linked list and the deserialization will be complete for this block's tree.

For example, consider the matrix from figure 2.15. We already have its k^2-tree block representation in figure 2.17. We label each of the blocks A, B, C and D. Following the serialization mechanism above we can serialize it as in figure 3.3.

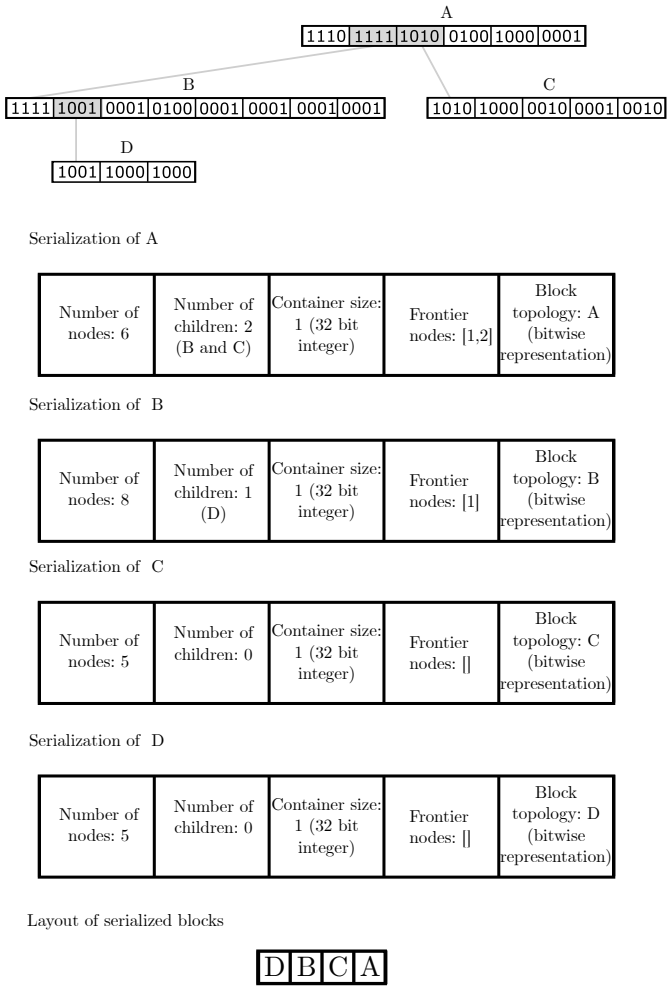

Figure 3.3: Example block-tree serialization. First we label each one of the blocks for convenience. Then, for each block we apply the serialization mechanism. Note that in each block, the last part of the serialization is the block topology or bit sequence describing the block. We chose to describe it in this figure for convenience, instead of writing the full bit sequence. Finally, we concatenate all the parts as described by the serialization algorithm.

The benefits of this serialization are:

- It is even more compact than the deserialized index, as its structure allows ommiting any pointers that reference other blocks. This also implies a good performance in deserialization, as less data needs to be retrieved from disk.
- It can be traversed sequentially in disk, avoiding any disk random accesses.
- It offers good performance as the bit sequences needed by the index are already in the serialization, the deserialization consists of putting the same bit sequences in memory and materializing the pointers.

3.2. Mixed tree serialization

For the mixed tree we are going to use the previous block's tree serialization when dealing with the block's tree part, but first let's see how we are going to serialize the top levels before that.

We are going to save the top levels in a bitwise representation, in the same fashion as we do it with blocks of bits. First, we create a container, which will contain 4-byte integers where we will put 4-bit nodes, telling us which children they have. To materialize this container, we have to count how many nodes there are; we do this simply by traversing the top-level tree once beforehand.

Next, we traverse again the top-level tree in depth-first-search order and turning on the ith bit of a node if its ith child exists. Once we are done with a node, we continue with its children recursively from left to right. With this, we end up with a Depth-first unary degree sequence (DFUDS) bitwise representation of the top-level tree, which we are going to use next.

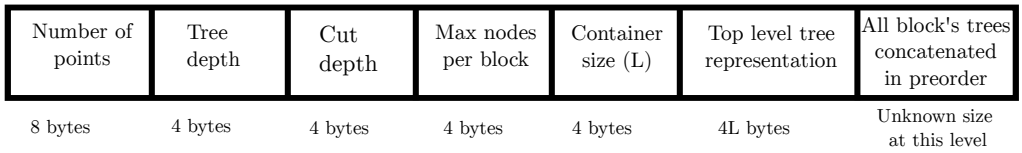

Figure 3.4: Mixed tree serialization

We serialize that data as figure 3.4 indicates; the top-level tree representation is the bitwise representation we just mentioned. The concatenated block's trees in preorder are the trees found at leaves in the mixed tree if we traverse it with a depth-first-search preorder. Each one is serialized as described in section 3.1.

Each part represents the following:

- Number of points: This is the total number of points encoded in the mixed tree. This is going to be useful to keep if we want to ask the index how many points it has stored, without having to traverse it fully, which can be expensive.
- Tree depth: The total depth of the mixed tree. This is fundamental if we want to make queries over the index, as it helps us to identify the leaves of the full tree, not just the top-level part. Then we can know when have found a point, for example.
- Cut depth: This is necessary for the mixed tree, as it tells us when we reach leaves in the top level part and start treating them as trees of blocks.
- Max nodes per block: This is necessary for updating the k^2-tree as it tells us when should we split a block into two blocks, when surpassing this amount in a block.
- Container size: This number will tell us how many 4-bytes we have to read in the top level tree representation next.
- Top level tree representation: We will use this bits sequence to rebuild the mixed tree.
- All block's trees concatenated in preorder: It is as we just mentioned. It is important not to confuse this with the ordering of their blocks, which are laid in post-order within

a tree serialization. What is in preorder here are each full tree serialization with respect to each other.

If we want to deserialize the mixed tree, first we are going to read the data until the top-level tree representation, inclusive. Next, as we have a bitwise representation of the top-level tree loaded in primary memory, we are going to traverse it from left to right with a recursive function which will keep track of the depth it is within the sequence. What we do exactly is to check each 4-bit sequence separately as it represents a node in the mixed tree, and by doing that we can know which children it has. The leftmost bit which is turned on will represent the next 4-bit node in the sequence, the second leftmost is somewhere to the right after scanning the first subtree fully and so on. This way, as the leftmost 4-bit node represents the root of the mixed tree, scanning recursively those bits from left to right will actually traverse the full sequence of bits from left to right. At each step of the recursion we will also be building the nodes of the mixed tree.

As we are keeping track of the current depth, we can know when we are reaching a leaf, corresponding to a tree of blocks, if that depth becomes the cut depth. Whenever that happens, we read a tree of blocks from the serialized data, as explained in section 3.1 and putting its root as a leaf in the mixed tree we are currently creating.

When we are done with the last node in the sequence of bits we are traversing, we have the mixed tree fully rebuilt.

4. Caching System

In this part, we describe how the caching system works.

The data cached are k^2-tree indices that were created beforehand and stored in secondary memory. In each replacement step, some indices might be loaded to RAM, in order to build a response for the user.

The caching system is accessed from a triple database manager, which when executing a query and requiring to retrieve matches from a triple pattern, it fetches those matching values from the cache first if they are available, or from its own storage otherwise.

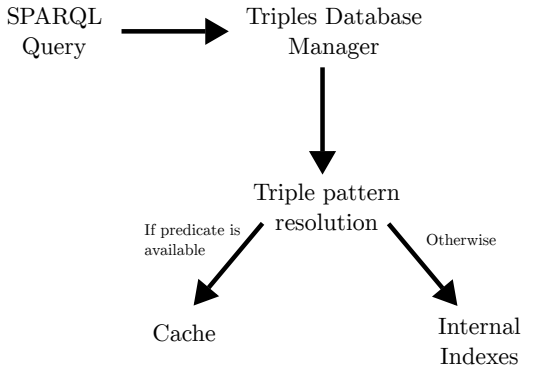

Figure 4.1: Cache general flow

In figure 4.1, we have a very broad representation of the system where the idea is to use the cache as much as possible.

We use our custom-modified implementation of Apache Jena [25] as an RDF store, also referred to as a fork. The modifications that we made to Jena were to make possible the use of the cache for BGP evaluations, and also to notify the caching system for RDF-triple updates.

Apart from this, we implemented an adapter Java server program [26] that uses the fork as a library dependency and receives user queries via a network connection with another Java client program that reads queries from a file for our experiments. The adapter server makes possible communication with the k^2-tree caching system via a TCP/IP persistent bidirectional connection, where the messages are binary data encoded with Google's Protocol Buffers library.

The k^2-tree caching system [27] implemented in this work is in C++, it manages the TCP/IP server connection, and the lifecycle of all k^2-trees, and it implements join algorithms and serialization.

The k^2-tree data structure [28] implemented in this work is in C. This is used as a library in the caching system, and it implements lazy scanning algorithms for rows and columns,

and also lazy full scanning of k^2-trees. It supports insertion and deletion of points.

The C++ caching system is also aware of the Protocol Buffers encoding, because the library generates code that allows it for several well-known programming languages.

4.1. NodeIds pre-processing

One of Jena's storage engines to choose from is TDB, which represents resources as NodeIds. These are eight-byte values that TDB uses for performing query evaluation. We use them as coordinates for the k^2-trees.

Our modified Jena gives us the ability to extract all the triples stored in NodeIds format, which we dump into a file in binary format. Next, we apply a custom external sorting algorithm on this file, which orders the triples by their predicate. This arrangement allows us to index a single k^2-tree at a time, optimizing the memory usage, and also avoiding random access of distinct k^2-trees when traversing the triples file.

4.2. NodeIds streaming

When Jena receives a SPARQL query string, it first transforms it into an Abstract Syntax Tree (AST). We traverse this AST and collect the predicates into a set. We send this set to the cache, asking which of those predicates it has available. The caching system will check which of those are loaded in its table of loaded predicates. At the same time, it will lock them to prevent them from being removed from memory while streaming data from those specific k^2-trees. Next, it will send Jena a subset of the set given to it, containing the predicates that the cache can currently access.

Then, Jena will evaluate the query as usual until the step in which it has to retrieve a triple pattern from somewhere. We check if this triple pattern has a variable predicate or not. If it has a variable predicate, we obtain the triples from TDB. If it is constant, we check whether it is within the set of available predicates we got from the cache. If it is not there, then we retrieve the triples from TDB. Otherwise, we ask the caching system to stream the triples matching the pattern. To do this procedure fast, we retrieve batches of matching triples from the cache. When a batch gets fully consumed by Jena, we continue retrieving them until there are no more triples matching the pattern.

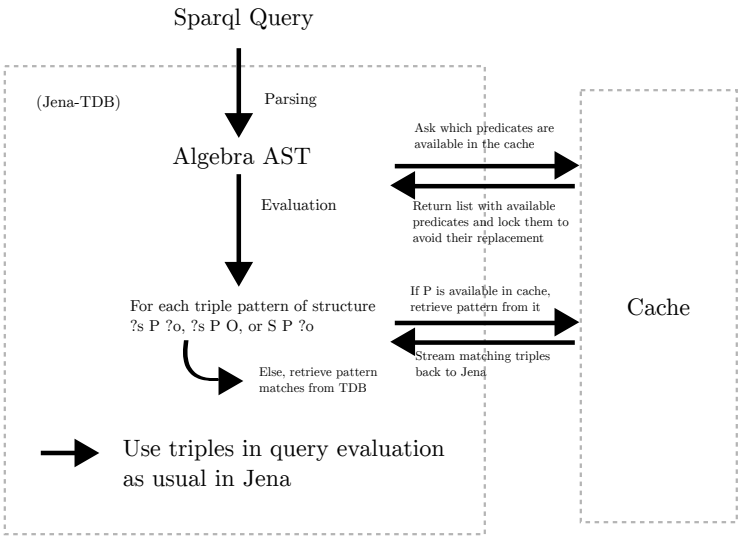

Figure 4.2: Diagram of interactions between Jena and the cache, where Jena first parses a query and receives results from the cache.

In figure 4.2, we have a diagram that shows the steps since receiving a query, consider triple patterns for resolution in the cache and evaluating the query.

4.3. Mapping NodeIds to internal IDs

For using the NodeIds as coordinates in the k^2-trees, we could simply use the 8-byte values as they come from the Jena side. This means that we would have to use k^2-trees of height 64 to cover the full range of values that 8-byte coordinates need. In practice, datasets can have a massive number of NodeIds. For example, the Wikidata snapshot this work is based on has around 1.1 billion NodeIds.

Even if it had a hundred billion, a height of 64 for the k^2-tree is way more than we need; a height of 37 is enough to cover that number. The problem of adding an extra level of height is that it adds an extra 4-bit node in the block structure of the trees, which makes the scanning and insertion performances worse by a considerable amount, and also adds up on the size of the trees.

To optimize around this in this work, static and dynamic map structures were implemented within the C++ k^2-tree caching system. The static structure is an array of increasingly sorted NodeIds, in which we use the position in the array as a coordinate in the k^2-tree, we also refer to the k^2-tree coordinate as an internal ID. The dynamic structure maps between NodeIds and a sequence of consecutive integers that is the continuation of the static structure positions.

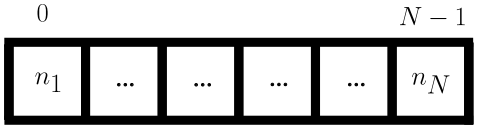

Figure 4.3: Static sequence. If $i < j$ then $n_i < n_j$. Each stored value is a NodeId and each position is an internal ID

In figure 4.3, we have a representation of the static structure. As mentioned before, each number occupies up to 8 bytes and elements are sorted in increasing order, which allows

finding an internal ID by a given NodeId in time-complexity $O(\log N)$ with a binary search algorithm. More importantly, it allows translating from internal ID to NodeId in time $O(1)$. This $O(1)$ translation is the most used one, because results stored in the k^2-tree caching system use internal IDs.

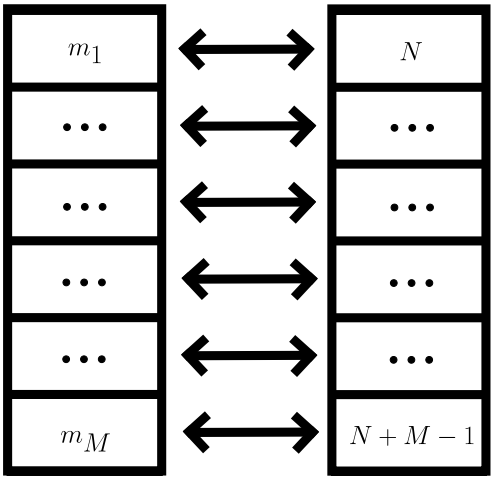

Figure 4.4: Dynamic mapping. No restrictions. Each value on the left side is an internal ID and each value on the right side is a NodeId

In figure 4.4, we have a representation of the dynamic structure, where M is the number of elements added to the dynamic structure and N is the number of elements in the static structure. We want to use this structure when new resources are added to Jena as part of RDF triples, but they get entered after the indexing by k^2-trees. By doing this, we avoid having to modify the static array structure and also avoid rebuilding the k^2-trees indices, which would be very slow to do each time there was a new update. In a sense, this dynamic mapping can be thought of as a helper structure that delays reindexing.

The dynamic structure, by nature, is much bigger than the static structure, so ideally we would want to have most of the NodeIds in the static structure, rather than in the dynamic structure. It is also more expensive performance-wise than the array, because the array has a $O(1)$ time-complexity for mapping internal IDs to NodeIds, while the dynamic structure in the best case would need to be implemented with a hash table to achieve this time complexity for the average case, rather than every time. In this work, we decided to optimize the dynamic structure by space, rather than speed, so we used C++'s STL map, which typically is implemented by a red-black tree, giving us $O(\log M)$ lookup time. Also, note that to achieve the bidirectional mapping, we not only have one map, but two, one for each direction.

Transforming data from the dynamic structure to the static structure is expensive, because doing so also requires that reindexing happens on all the k^2-trees, since the static structure depends on the ascending order of the NodeIds, and adding new elements to it means affecting the original positions of the existing NodeIds (there is no guaranteed ordering for newly added NodeIds), making the existing k^2-trees corrupt.

Full reindexing of Wikidata using k^2-trees takes around 4.8–12.8 hours (see Table 7.3), depending on the chosen configuration for the indices, so it is reasonable doing it if it doesn't happen too often. We don't need to do it too often if the dynamic structure grows at a sufficiently low rate.

This hybrid mapping depends on the number of new resources added within a time window

being low enough that we don't end up reindexing almost all the time. The number of edits in Wikidata over a 2-year period shows that there are on average about 18 million edits per month [23].

Each edit can have several triples added, meaning that several resources are added. In this work, we don't have a real estimation of number of triples per edit, but as any user can add something, we could use an upper bound estimation by using something like 10 triples per edit, and assuming that each resource in a triple is new, with the exception of predicates, which are much rarely added. In total, for our estimation, it would be 20 resources per edit, which implies 360 million new resources each month. Each NodeId in the dynamic map requires 32 bytes, and 8 bytes in the static one, so in a worst-case scenario there would be 11.52 GiB worth of NodeIds in the dynamic structure. Then if the server has enough RAM, it is reasonable to do the reindexing once a month or once a week if needed. Clearly, this upper bound estimation is loose, since the snapshot used in this thesis has 1.1 billion distinct resources, for several years worth of data.

One concerning issue with this approach is that full reindexing needs to co-exist with new incoming data, and also the problem of maintaining availability while this processing is occurring. These problems are not being solved by this thesis, they are rather left for future work.

4.3.1. From Jena NodeIds to internal IDs

Given a Jena 8-byte NodeId, its internal representation can be stored either in the array structure or the map structure; we look for it in both if needed. First, we look in the array structure by doing a binary search. If it's there we return the position found as the internal ID. If it's not there, we look it up in the map. The array scan has time complexity $O(\log N)$, where N is the number of elements in the static structure and the number of resources known until indexing time. On the other hand, the map scan has time complexity $O(\log M)$, where M is the current number of NodeIds in the map structure and the number of resources added after indexing time. This last time complexity is considering we are using as a map a red-black tree.

4.3.2. From internal IDs to Jena NodeIds

In case we want to map from internal IDs to Jena NodeIds, as we know the size of the static array, we can compare the ID with that size and if the position is below the size, we can be certain that the NodeId is stored in there. To get the NodeId, the only needed operation is to fetch it from the array, using the internal ID as the position. If the value is greater or equal than the size, it could be inside the dynamic map, so we perform a map lookup. The array fetch takes $O(1)$, while the map lookup takes $O(\log M)$.

4.3.3. Efficiency of translation

In this system, NodeIds are translated to internal IDs to fetch data matching a triple pattern. On the other hand, internal IDs are translated to NodeIds in order to get the results in the original set of IDs. The second kind happens much more often, because for each query we might have a huge amount of results that need to be translated, while the first kind depends on the query size. This means that it is very convenient having a $O(1)$ solution to

translate from internal IDs to NodeIds, which we have more or less if the array structure is much bigger than the map structure. All of this means reindexing not only helps with memory usage, but also with the overall performance of the system.

4.4. Replacement Strategy

One essential part of any caching system is its replacement strategy, which chooses what objects are going to be discarded or kept in order to stay within the memory boundaries previously established and at the same time trying to minimize the number of cache misses the system gets.

We built a mechanism in which any replacement strategy can be added, based on a priority queue. This priority queue is actually a C++ STL set, which is a red-black tree in most systems, but could vary according to the C++ implementation of the system running our caching system. We can use the STL set as a priority queue, because it is guaranteed to have its elements in order, given by an ordering function which we can provide by comparing the result of applying a cost function to keys, offered by the concrete replacement strategy. Also, the main reason to choose this structure is that we can remove its elements in $O(log|P|)$ time, with P the set of predicates, whereas the actual priority queue from the STL doesn't support erasing elements by their key.

One tricky issue of the STL's set is that when a cost function assigns the same cost to two different keys, it will regard them as the same and as the set keeps uniqueness among its elements, it wouldn't add two different keys with the same cost value. However, we can bypass this by checking if they have the same cost value and are different, in which case we use as a second comparison function the comparison between key values.

In this work, we try one strategy for replacement, least recently used or LRU.

4.4.1. Least Recently Used

To implement LRU, we have to provide a cost function for the keys. Also, this strategy should be notified when a key is hit, in order to update the cost functions.

We keep a map from keys to a value, which we call the LRU value. In addition, we keep two variables, named *low* and *high*, where *low* will be the lowest of LRU values held in the map and *high* the highest plus one. The *low* variable represents the element with the lowest priority and next to be removed, in this context, it is the least recently used element. On the other hand, *high* represents the element with the highest priority and the most recently used element.

Each time we are notified about a key being hit, we check if it exists within the map; if it does, we check if it corresponds to *low*, in which case we reassign *low* to the next smaller element and remove the key from the map, followed by reinserting it with the value of *high* and then incrementing *high*. In case it didn't exist in the map, we skip the steps of updating *low* and only inserting the key into the map with the value of *high* and incrementing *high*.

We don't touch *low* just yet at this point, this is deferred until we remove a k^2-tree from memory, which happens on a different flow in which the system determines that the next k^2-tree would exceed capacity and in that case it would start freeing up memory from lower-priority elements and updating *low* at the same time.

The cost function of a key will be its LRU value minus *low*. Which would avoid any overflows if we simply used the LRU value.

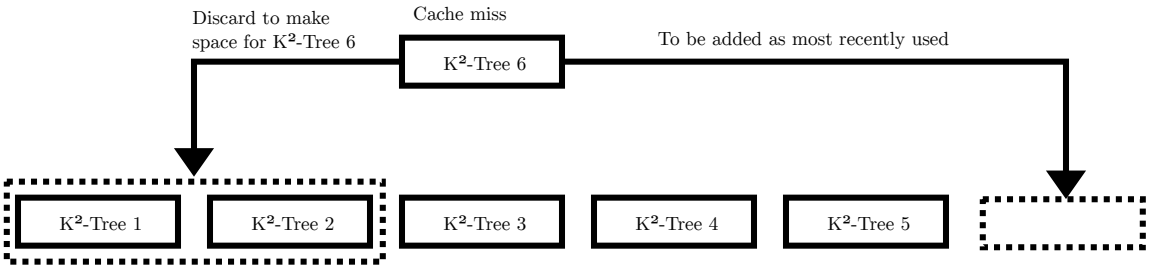

Figure 4.5: LRU replacement algorithm

In figure 4.5, we have a visual representation of what happens in the LRU replacement. Here we have a scenario in which adding k^2-tree 6 would exceed the capacity of the cache, so we start freeing up elements with lower priority. In this case, freeing up the k^2-tree space is not enough to be under capacity after adding k^2-tree 6, so we need to keep going on the deletion of trees with low priorities, that is why we also discard k^2-tree 2, so after adding k^2-tree 6 we would be occupying space below the capacity given to the cache.

5. Query processing

The data cached are k^2-tree indices that were created beforehand and stored in secondary memory. In each replacement step some indices might be loaded to RAM, in order to use them to build a response for the user.

The flow of the processing starts with a query tree, given by the user of the system, that describes steps to execute the original SPARQL query. All of this is done by Jena; extra processing in this work is done on a basic graph pattern level, which is a sequence of triple patterns that can imply joins or products between the triple patterns.

Initially in this work, the idea was to delegate the BGP evaluation to Jena, but that meant too much overhead for the message sending, which is done by TCP/IP sockets, because in many occassions during intermediate BGP evaluation there is only one triple that has to be fetched from k^2-tree cache. Due to this issue, it was decided to perform BGP evaluation within the k^2-tree cache, which decreases the network messages overhead significantly; with this there are fewer messages to send between Jena and the k^2-tree caching system.

Here we describe the way we process BGPs in order to use the k^2-tree indices. The triple pattern ordering within the BGP can have a high impact on the query performance, so we decided not to change the ordering given by Jena and with this, we can use a good enough query optimizer and also avoid introducing an extra source of variability in the experiments.

To avoid complete materialization of intermediate results in a way that occupies a large size of memory, so we can build iterators that can produce results on demand, we need to lazily scan the k^2-trees and to allow this we implemented two lazy scanning algorithms to traverse the trees. The first is a band scanner and the second is a full scanner.

In a BGP, several operations could be necessary to perform, depending on what kind of triples we get. We recognize them as one variable Cartesian product, two variable Cartesian product, inner join, one variable intersection and two variable intersection and implemented them as part of the BGP Iterator.

In the following, we describe in detail the scanners, the operations and the algorithm that uses them.

5.1. Band Scanner

This scanner consists of traversing a band of a k^2-tree, which is a generic name given here that refers to either a row or a column in the matrix representation of a k^2-tree. The scanning is done lazily, in this case meaning that we can get one result and halt the processing completely until we want to get the next result.

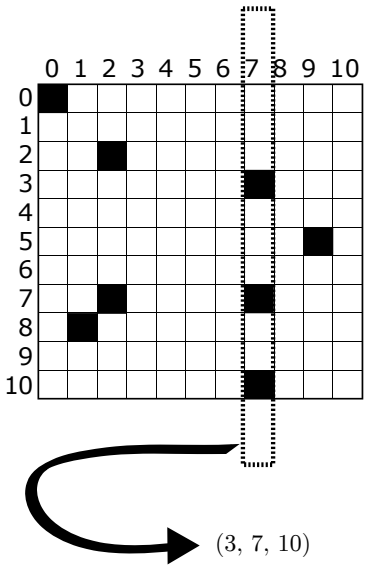

Figure 5.1: Band scan of column 7

For simplicity let's assume first that this is a column scanner. In this case, we are given a column ID, which is identifying a subject, and also we are given a predicate ID that identifies a k^2-tree we currently have loaded in RAM. The task is to scan the column for the column ID given, such that at each step, the scanner will report the next row ID, identifying an object within that column and that has not been previously reported.

We start this algorithm from the root of the tree. As we have four options of nodes we could descend to, we have to choose which ones to go in, depending on the column ID. As every one of the four nodes represents a quadrant in the k^2-tree we have to select those where the given column lays in. For the column scanner, we notice that there are always two quadrants to consider, either both the TL and BL or both the TR and BR (top-left, bottom-left, top-right, bottom-right respectively).

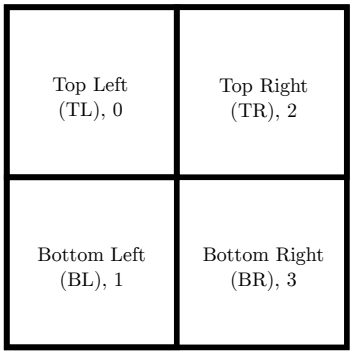

Figure 5.2: K^2-tree quadrants

For the quadrants selection, we make use of the matrix representation. If the column ID given is less than half of the matrix's column size, then it must be that the column for this ID is in the two left quadrants, alternatively, if it is greater than the half of the matrix's column size, it has to be in the two right quadrants. We proceed recursively with this idea, until we reach the leaves of the k^2-tree, where at each next depth, the matrix representation column size is reduced by half and the column ID to check becomes the previous one, modulo half the previous column size. When we reach a leaf, we will have a 2×2 matrix, described by a

4-bit sequence. In the case of the column scan, we have to check the first and second bit if the current column ID is 0 or the third and fourth bit if it is 1.

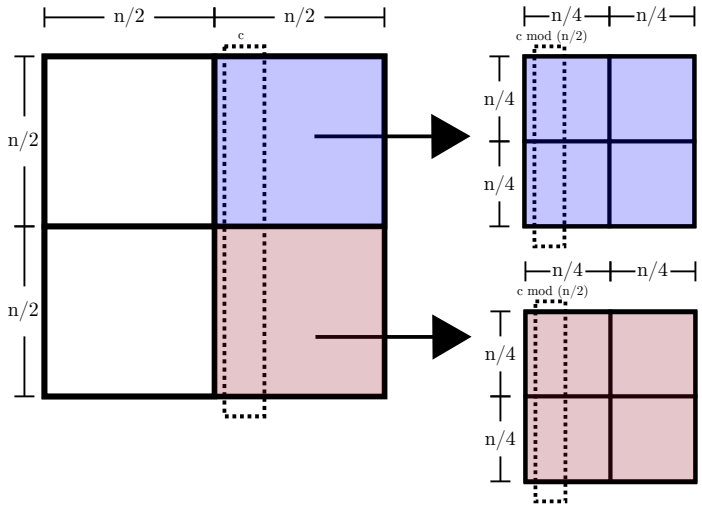

Figure 5.3: Reducing search space while scanning

In figure 5.3, we can see how the search space is reduced by half at each step while applying modulo to the coordinate being scanned. We always suppose that these matrices have sides of length being a power of 2, that is, there is a positive integer m such that $n = 2^m$ for n in the figure. Also, take into consideration that whenever we find an empty quadrant, there is no path to follow from there in the k^2-tree, then we are not really scanning the full space, which is one of the reasons of why the k^2-tree is convenient.

As we have to report row ids, at each step of the recursion, when going down a quadrant/node, we increase a variable by half the size of the side of the corresponding matrix only if the quadrant selected is in the right.

For the row scanner, the procedure is similar, with roles reversed; instead of caring about the left or right quadrants, we choose between top and bottom quadrants. Instead of adding half the size of the side of the matrix when choosing a right quadrant, we do it when choosing a bottom quadrant.

As for achieving laziness, we keep the recursion states inside our own stack instead of delegating that part to a recursion by function calling. Also, we implement *next* and *hasNext* functions and we explicitly call the first *next*, storing its result, in order to have a result for the first call to *hasNext*. With this procedure, a call to *next* will stop the tree traversing when we have a new resulting row ID in case of the column scanner or column ID in the case of the row scanner.

5.2. Full Scanner

This scanner will output every pair of points stored in a k^2-tree in a lazy manner, similarly to Band Scanner, implementing *next* and *hasNext* functions.

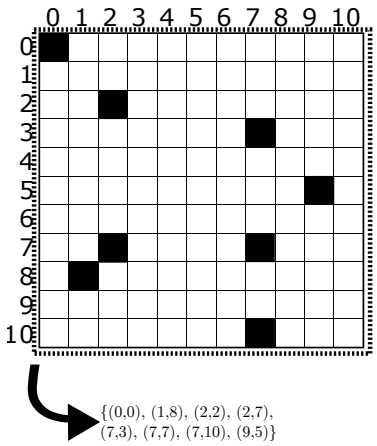

Figure 5.4: Full scan example

The procedure is even simpler than Band Scanner. At each step we descend into all of the quadrants, keeping two variables; the first will hold a column ID result and the second a row ID result once we process a leaf. We will add half the size of the current matrix side to the first variable if going through a right quadrant and add the same value to the second variable if whatever quadrant was chosen is also a bottom quadrant.

5.3. One Variable Cartesian Product

We need to apply a one-variable Cartesian product when encountering a triple pattern having only one variable that doesn't appear before, most commonly referred as not being bound. Suppose at this point, we have a partial result binding; then, to continue building the binding, we will perform a band scan of the k^2-tree represented by the predicate given in the triple and using as band ID the other term which is not bound in the triple.

```
SELECT * {
  ?a Predicate1 Object1 . # Triple Pattern 1
  ?b Predicate2 Object2 . # Triple Pattern 2
}
```

Figure 5.5: Triple pattern with one variable `?b` not appearing before.

In this case, for each previous partial binding given and each value in the band, we will produce a new partial binding.

Consider the case of the query in figure 5.5. First we have the pattern 1, which we will use to perform a row band scan on the k^2-tree associated with predicate Predicate1 on the band ID we can obtain from Object1. This band scan will get us values for `?a`. As we have a second triple pattern to deal with and we haven't other source of values for `?b`, we have to make a Cartesian product between the values we get from pattern 1 and the values we get from pattern 2. For `?b`, we will do the same thing we did to obtain values for `?a` using the pattern 2, as it also has a single variable. With this, for each value we get for `?a` from the pattern 1, we have to get a value for `?b` from pattern 2.

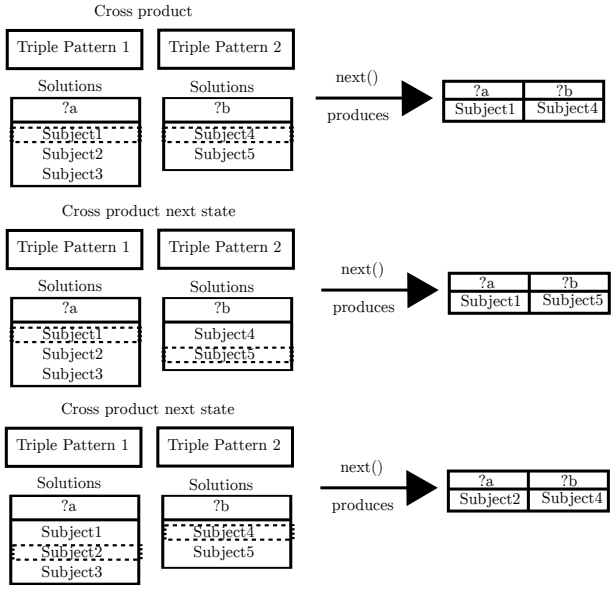

Figure 5.6: Iterator only having Cartesian product

In figure 5.6 we can see what happens when having a BGP iterator with only a Cartesian product operation between two triple patterns, which would happen with a query like the one in figure 5.5. First, there are two band scanners in play, the one for pattern 1 in the left and other for pattern 2 in the right. Both start from the beginning of their scanning, but the first only advances once we scanned fully the second and when that happens, the second scanner is reset.

This was a simple case when only having one other triple pattern as source of previous bindings, but in general we can have any number of triple patterns before, of all kinds. We can even have zero other previous bindings, in which case, we simply perform a band scan and output those values.

5.4. Two Variable Cartesian Product

We require a two variable Cartesian product when we have a triple pattern with two variables which are not bound. As before, we have a partial binding and now we perform a full scan, which will produce a column and row ids in each step. For each one of those pairs, we produce a new partial binding.

```
SELECT * {
  ?a Predicate1 Object1 . # Triple Pattern 1
  ?b Predicate2 ?c .      # Triple Pattern 2
}
```

Figure 5.7: Triple pattern with two variables `?b` and `?c` not appearing before.

Take for example figure 5.7, where this time we have a triple pattern with two variables not appearing before.

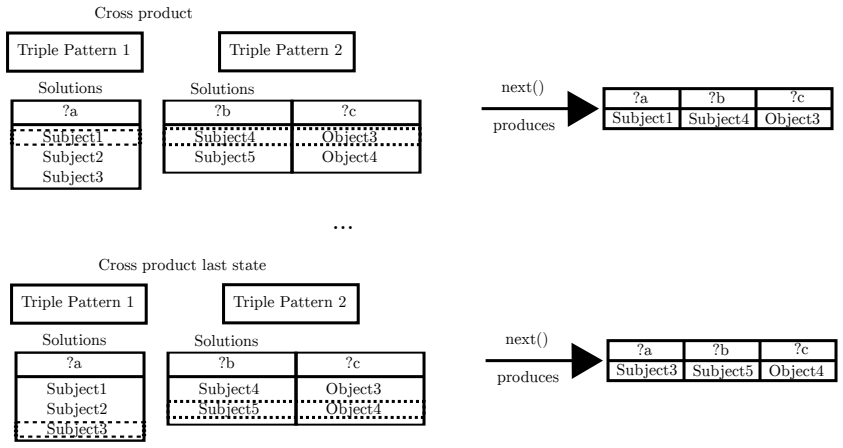

Figure 5.8: Iterator only having Cartesian product with two variable in a triple pattern

In figure 5.8 we can visualize how each call to *next* would affect the iterator and produce new bindings while scanning one band on the first triple pattern and scanning a full k^2-tree on the second triple.

In the same way as the case with only having one variable in a pattern, this generalizes to any number of patterns coming before, even zero, in which case the bindings would only feed from a full scanning of a k^2-tree.

5.5. Index nested-loop join

The index nested-loop join; sometimes referred to as inner join, is used when we have a triple pattern with two variables, one of them is bound by previous triple patterns and the other is not.

Suppose, for simplicity, that the subject is bound and the object is not, like in figure 5.9. We are receiving a partial binding where the subject variable has a value, so we perform a column band scan on the k^2-tree represented by the predicate given and with the subject ID of the partial binding as our band ID. For each row ID retrieved from the scanner we will produce a new partial binding.

```
SELECT * {
  ?b P1 D1 . # Triple Pattern 1
  ?b P2 ?c .  # Triple Pattern 2
}
```

Figure 5.9: Triple pattern with two variables `?b` and `?c` and `?b` also appears before.

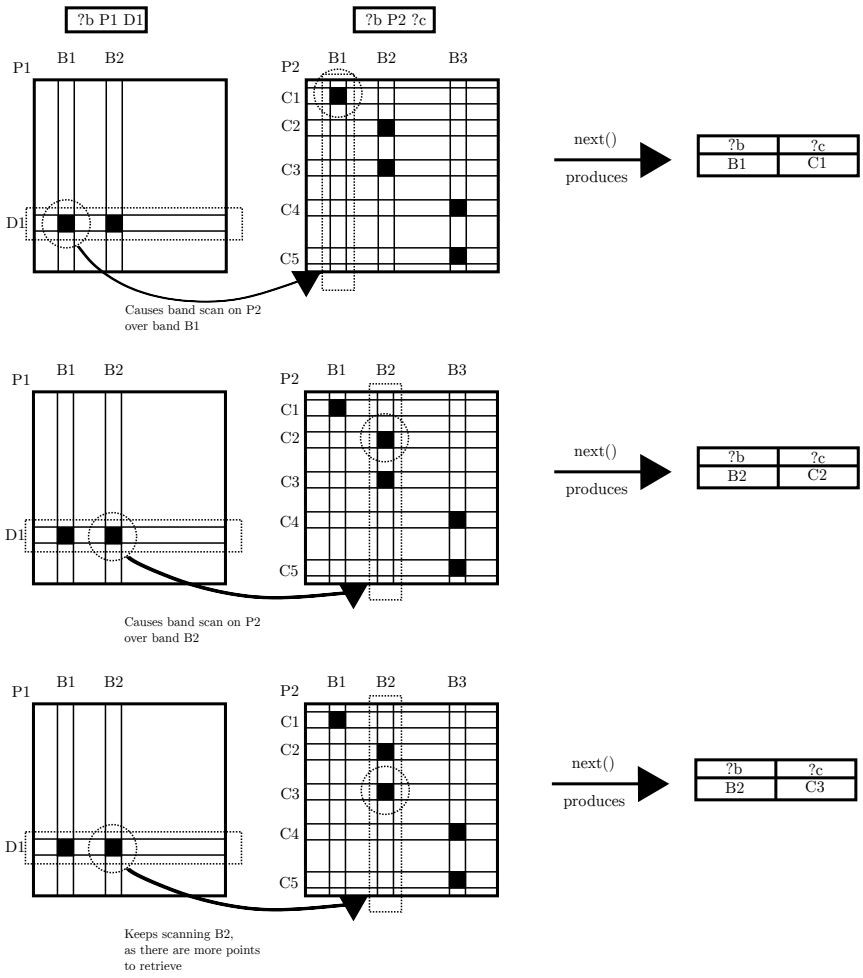

Figure 5.10: Inner join with a single triple pattern as left source

For example, in figure 5.10 we can see that the left triple pattern `?b P1 D1` is being used to perform a row band scan over the row `D1` and each of its values will bind to `?b` in the second triple pattern. This temporary binding will allow us to perform a band scan for each new value coming from the left. Each time we extract a value from a band on the right we will be able to produce more bindings which satisfy the given query, until we reach the last value coming from the left and for that value there are no more values on the right.

This procedure can be generalized to any number of triple patterns on the left, of all kinds, as long as we have a triple pattern on the right with two variables in both subject and predicate and also one of them appears before. In case there were no triple patterns on the left, this wouldn't be an inner join, instead it would be a Cartesian product with two variables.

Notice the difference between this and two variable Cartesian product; here we do a scan based on a previous binding, whereas in the Cartesian product we only use information given by the triple pattern.

5.6. One Variable Intersection

Here we receive a triple pattern with only one variable, which is previously bounded. Suppose that the subject is the only variable. Then we only have to check if a pair with

known column ID and known row ID (given by the pattern) are in the k^2-tree associated to the predicate in the triple pattern given. For this purpose we just ask directly to the k^2-tree using the *has* operation.

```
SELECT * {
  ?a P1 B1 . # Triple Pattern 1
  ?a P2 C1 . # Triple Pattern 2
}
```

Figure 5.11: Triple pattern with one variable `?a` which appears before.

For example, to process the query from figure 5.11, we first perform a band scan on the first triple pattern and feed those `?a` values to the *has* operation, such that if has_{P2}(`?a`, `C1`) is true, we keep the concrete value for `?a` as a resulting solution and if it is false, we discard it. The described procedure can be visualized in figure 5.12.

We can also generalize this for any amount of triple patterns on the left, except zero, which would yield bindings with more variables and the triple pattern of interest for the intersection must have its only variable bound to those bindings.

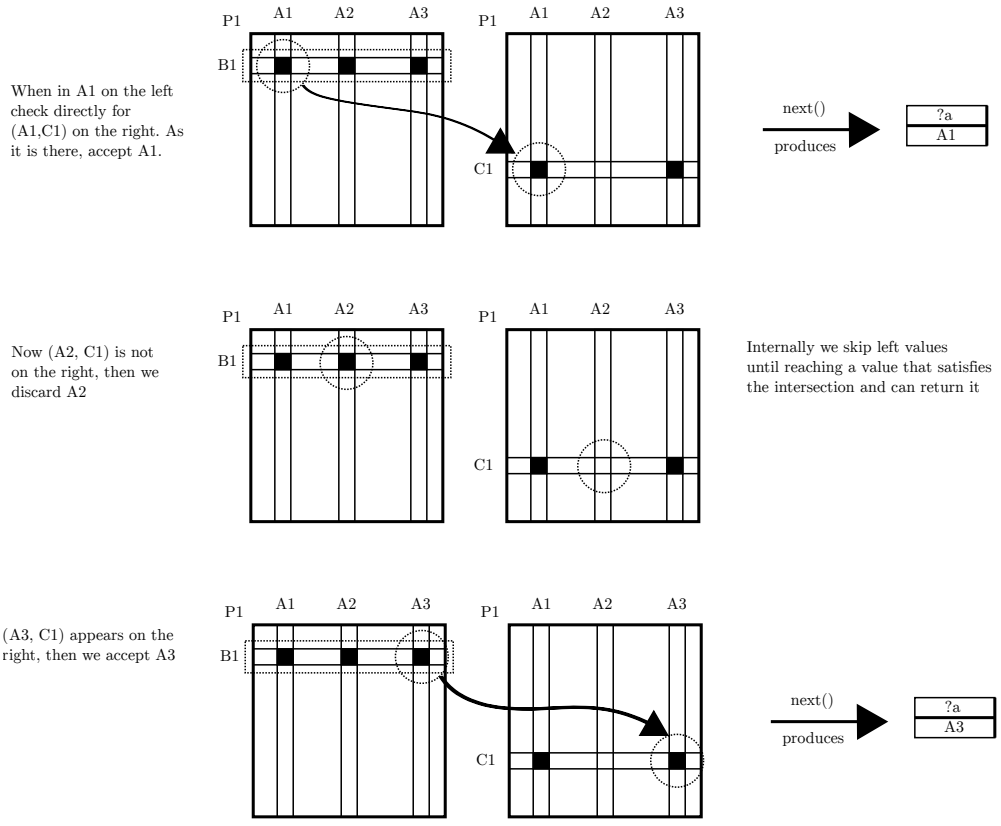

Figure 5.12: Intersection by using a band scan on the left and *has* on the k^2-tree on the right

5.7. Two Variable Intersection

Now we have a triple pattern with two variables, one in the subject and the other in the object. This time both are bound and we perform a *has* operation in the k^2-tree as in the one variable intersection case.

This is essentially the same as in one variable intersection, the only difference will be that instead of assigning one variable in the right to values coming from the left, we will assign two, one for subject and one for predicate and then apply *has* to accept or discard bindings.

5.8. Iterator algorithm

We implement *next* and *hasNext* operations for the iterator, as any iterator must have. These are implemented per the previous operations depending on the particular case. This whole idea is also known as a physical plan.

This and all the following iterators will output a binding when calling *next* over them. We represent those bindings as arrays of IDs, where each iterator has an array of variables in the same corresponding order of the results it will return, allowing us to map variables to values.

First, we perform a setup step, where we identify which are the variables that we will output and in what order they will appear in the variables array. We create with this a buffer, with the same size as the variables array, in which we will construct a binding. Also, in the setup step, we create the operations we previously defined, depending on the BGP input and put them in a sequence to be executed at each step.

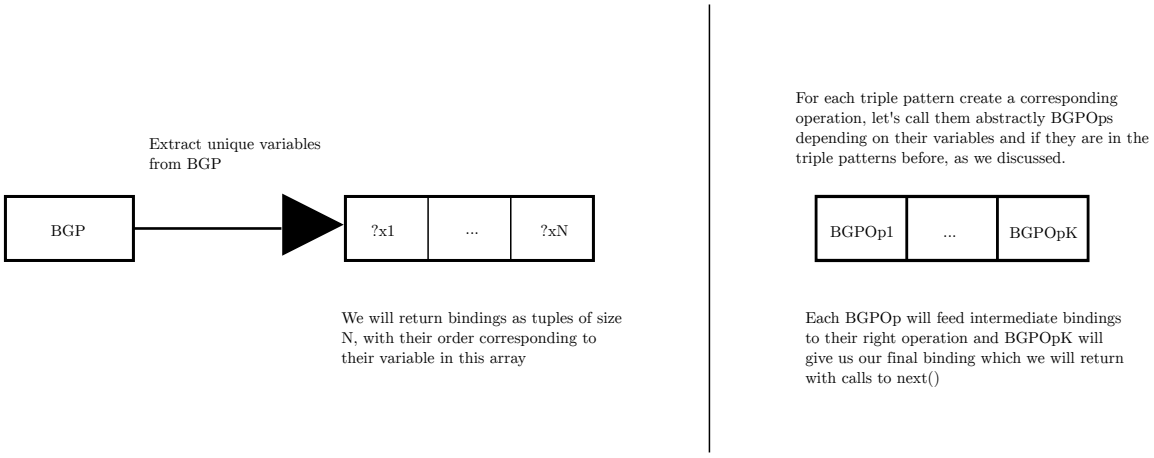

Figure 5.13: BGP Iterator setup

When calling *next* over the iterator, we traverse the operations sequence from left to right and when reaching the last one and retrieving a result from it we will have a complete binding for this BGP iterator which we will output in this call and advancing the necessary pointers to prepare the result for the *next* call. When the last operation has exhausted its results, we go back one step in the operations sequence, call *next* on that one and reset the last operation. The same can be done for any of the intermediate operations when they have no more elements, going backwards and resetting all of the following operations accordingly. The iterator will have no more results once all the operations in the sequence have reached their end.

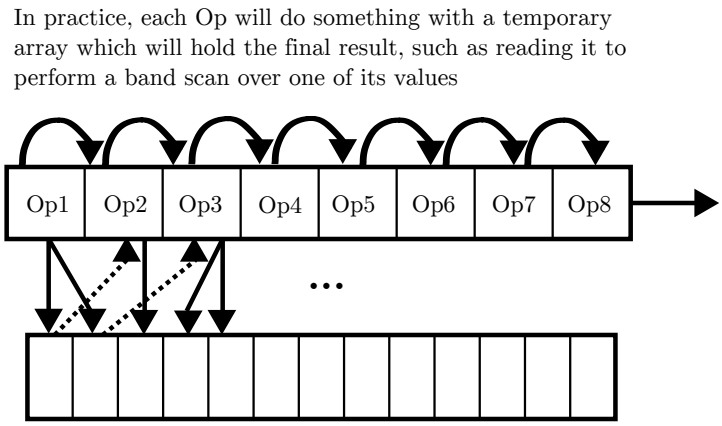

Figure 5.14: BGP Iterator Operations

In order to be able to answer the first *hasNext* call, we perform a first *next* call just after setup and store its result, such that each time we call *next* afterwards we will have the requested result stored, which will be returned once we compute the *next* result if there is one. In the case there was no result, the first call to *next* would produce a null result internally and *hasNext* would be false.

6. Updates

Updates are one of the trickiest parts of this system but are essential for keeping part of the cached data up-to-date as the database receives update requests. For one, we have to make the updates in both Jena and also in the k^2-tree cache. We also have to make sure that any future results that we get from Jena can also be received from the cache and vice versa. As our cache is an in-memory structure pre-computed and stored on disk for initialization, there is the need to modify the data stored on disk for future initializations, apart from changing the in-memory structure during live updates.

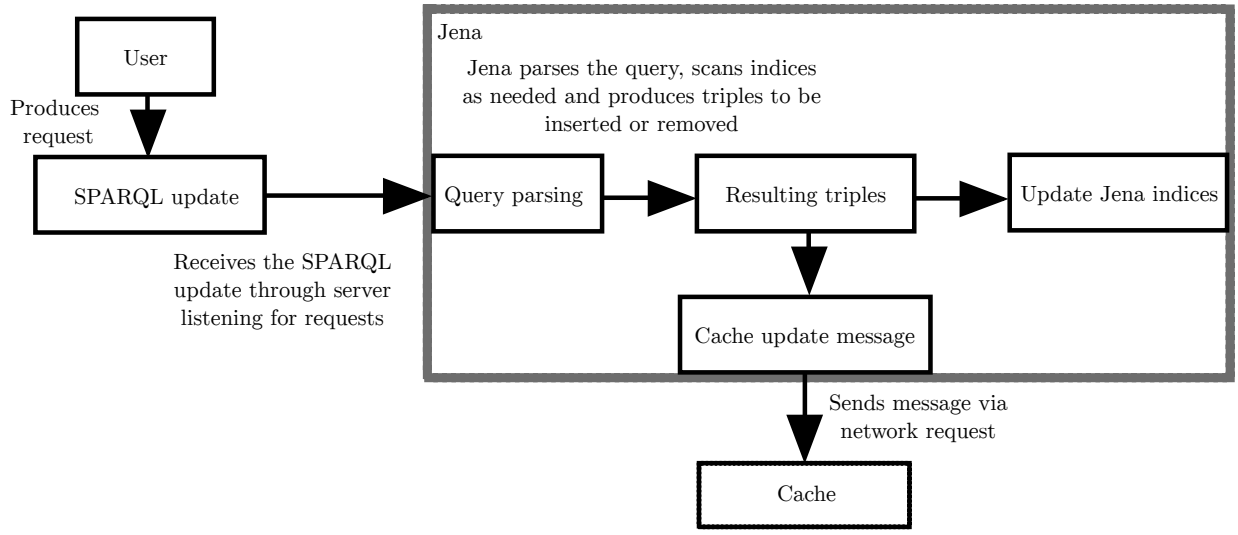

Figure 6.1: Updates in Jena

In figure 6.1, there is a diagram of the processing steps occurring in Jena during an update. First, the user sends an update message to an endpoint connected to Jena. As this message is in a SPARQL format, it is processed by Jena and converted to triples to be inserted or removed. Then, the updated is applied to the B+tree indices in Jena. Finally, the triples are wrapped in messages to be inserted or deleted in the cache. The step added in this work is the last one.

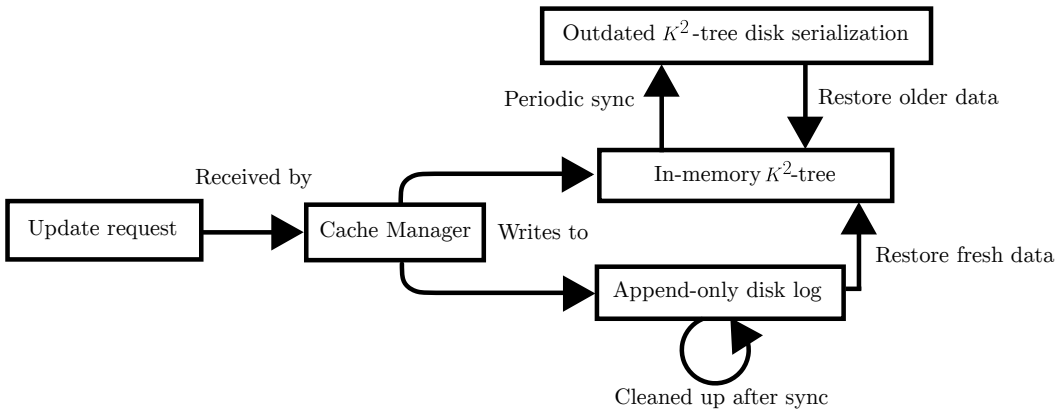

Figure 6.2: Updating the cache

In figure 6.2, we can see the full update flow on the cache side. First, the cache receives an update request from Jena and with that it updates a disk write-ahead log and the affected in-memory k^2-trees. Apart from this live update flow, there is a bulk processing flow to update the outdated k^2-trees disk serialization which at the same time cleans up the append-only log. The remaining step of restoring fresh data into the in-memory structure occurs after it has been restored from disk and there is some newer data in the cache.

6.1. Live updates

The first step in an update is that the user makes an update request, in the language of SPARQL, which is sent to Jena. This update request is then processed by Jena and transformed into two sets, one for insertions and one for deletions. This pair of sets goes into two paths, one is the internal Jena storage and the other is the caching system.

The caching system can receive new updates at any time, so it has to be prepared to process them quickly and to be ready for the next update each time. This becomes possible due to making only the minimal necessary and less expensive processing on each request and delaying any costly operations into later bulk processing stages.

The less expensive processing tasks are mainly two. One is to update the in-memory tree structure, which is regarded as relatively cheap due to only consisting of memory operations. And the other is writing the same updates to an append-only disk log. This disk log serves as a recovery backup, so we write to it before writing to the in-memory structure.

6.2. Initialization

If we are loading a k^2-tree, once we have loaded the tree from the compact structure in disk to RAM, we process the updates log, and make the insertions in the k^2-tree currently being initialized.

In the situation that we are pre-loading every tree on startup, we only process the log once after the pre-load.

If we are in cache-replacement mode, we have to process the log each time we load a k^2-tree, as it may contain data that did not get a chance to be synchronized with the disk serialization of the k^2-tree.

6.3. Re-indexing

If the updates log has grown too much, we can perform a full reindexing of the dataset and that would empty the log and recreate all the compact data structures stored in-disk. This has to be done during maintenance time because the cache will have to be offline. To alleviate this issue, two cache instances can be running at the same time, and for maintenance, we could index only one at a time. Another measure that can be taken is to make the synchronization for each k^2-tree separately instead of the full dataset at once.

In this thesis, we do not go into further depth on the study of re-indexing.

6.4. Deletions

Deletions are handled analogously to insertions. In the append-only log, we differentiate them from insertions with a flag, and on the k^2-tree we call the delete function instead of the insert function. Everything else is the same.

6.5. Performance considerations

The writes to a disk log can be expensive but is a price we have to pay to have some consistency and reliability guarantees. On the more positive side, the append-only configuration is as fast as it gets when writing to disk. Compared to B+tree writes, it will be much faster, because the fragmentation of the append-only file will occur only when the file system decides to do it. B+trees have to explicitly make random access patterns on the disk to perform updates.

On the other hand, the kind of databases that would use this system have often much more reads than writes, so the trade-off to make writes slower than reads should end up being worthwhile overall.

Regarding the processing of the write log, there are optimizations that are applied.

Firstly, we compact the log with some pre-defined frequency or condition. This is beneficial because with this we glue together several updates to the same tree and the switching between trees for doing insertions has some extra cost.

Secondly, we index the offsets of the file where each predicate has insertions. As there are typically a low number of predicates, this shouldn't come with much cost, and mixed with our first optimization it should the reduce number of random accesses significantly.

And our last optimization was to compress the updates to a single tree, which is done with the same k^2-tree data structure than the one used for the caching system. This reduces the overall size of the file, so file reads are potentially much less expensive.

7. Experiments

In this chapter, we explore some of the experiments done in this work. There are several questions that we want to address:

First, we want to know about the space occupied both in memory and on disk by the k^2-tree indices. The space occupied in memory determines how much data we can hold in memory at a time. On the other hand, the space occupied in disk storage is of concern because it helps determining the speed at which we can retrieve the indices from disk into memory.

Related to the previous point, we want to understand how long does it take to load up k^2-tree indices from disk into memory. If it can happen quickly, it means that using a replacement strategy with a predetermined capacity in which we can load a subset of all of the indices becomes more feasible.

Regarding the query evaluation, we want to measure how efficiently the queries can be evaluated with the entire graph cached in memory, which skips the cache replacement completely and tries to use the k^2-trees as much as possible. Apart from this, we want to see how well Jena behaves without the caching system complementing it and compare both cases. If the caching system has better performance than Jena alone, it could be worthwhile using it.

We also want to study the impact of using a replacement strategy on query performance, as we have to spend time loading indices from disk, it is expected that it takes longer time than loading everything beforehand, but it is important to know how much longer it would take, because if it is a low enough time it may show more value as a solution for caching as it would require reduced memory needs.

Lastly, we want to study the impact on update time as apart from updating the Jena database, we also need to update the cache storage.

7.1. Experiments configuration

7.1.1. Dataset

Our chosen dataset for experimenting is Wikidata. It contains a huge size of data and in that regard, it helps to showcase why this work is relevant, as we are using a compact data structure that can represent a large size of data in a very lightweight form. Apart from the size, Wikidata is not a laboratory dataset, but rather a real one that is used in production, which adds value to our experiments. We used a Wikidata complete (truthy) snapshot from May 2020.

We also use another synthetic dataset generated with the WatDiv tool [29]. This tool generates a dataset and different types of queries. Even though this is not the best way to analyze the system introduced in this thesis, it gives us some rough idea on how the system

performs in a somewhat random scenario, and it has the advantage of being validated by other researchers.

7.1.2. Queries and Updates

For queries, we selected a public log of queries [30] that is anonymized from 2017. This makes sense to use because together with the chosen dataset we can simulate the real scenario.

For updates, we took another Wikidata snapshot from 2021 and extracted the difference of RDF triples between that and the 2020 snapshot. We extracted a random sample of triples from this difference that can be used for experiments.

7.1.3. K^2-tree

Regarding the k^2-tree indices, we chose several configurations and identify them with the syntax `N-C-H`, where `N` is the maximum number of nodes per block, `C` is the cut-depth, and `H` is the height of the trees. Recalling from section 2.6, N indicates how much data we put into the bit sequence that encodes a portion of the k^2-tree, so this number lets us control the compression level at the expense of computing cost on the bit sequence. C indicates the depth at which we start using the block structure in the structure explained in the sub-section 2.6.2. `H` is the height of the represented k^2-tree, considering both the pointer structure on top and the block structure at the bottom.

7.1.4. Hardware Specs

The hardware specs are:

- CPU: *Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz* with 2 sockets and 16 cores per socket, totaling 32 cores. 4x32KB L1 Cache, 4x1MB L2 Cache, 11MB L3 Cache.
- Memory: 748 GiB RAM
- Storage: 21.7 TiB disk space HDD
- Operating System: *Devuan GNU/Linux 3 (beowulf)*. Devuan is a Debian fork

7.2. Index statistics

In this section, we cover some of the statistics that give some context for the next sections. It is worth mentioning here that the number of indices is 8547, each represented by a predicate, and this applies to all the next statistics.

7.2.1. Number of triples

In table 7.1, we show some statistics about the number of triples. The number of triples is shared among all k^2-tree configurations. The most interesting part here is that most indices don't have such a large number of triples as the largest index

Table 7.1: Number of triples per k^2-tree index

	Number of triples
Avg	670,058
P50	506
P95	107,655
P99	1,426,449
Max	2,430,879,380
Sum	5,726,991,458

In table 7.2, we show the top indices in the number of triples for the 128-0-32 configuration. As it can be expected, the number of triples seems to correlate with sizes. It is interesting to see that this is not necessarily a rule, for example the fifth item in this table <http://www.wikidata.org/prop/direct/P2860> has less triples than some of the indices, but it has a larger size than those. This can be explained by differences in the structure of the k^2-trees, sparse structures should have more size than dense structures when the number of triples is the same.

Regarding size and retrieval time correlation, it looks like it is more direct than between number and size; the bigger sizes should take more time to retrieve from the disk.

Table 7.2: Index statistics for top indices in the number of RDF triples with the 128-0-32 configuration

Triples number [millions]	Size in memory [MB]	Size in disk [MB]	Retrieval time [ms]	Resource
2430	5429	3608	32213	<http://schema.org/description>
483	1190	794	7078	<http://www.w3.org/2000/01/rdf-schema#label>
483	1190	795	7428	<http://www.w3.org/2004/02/skos/core#prefLabel>
483	1190	795	6965	<http://schema.org/name>
217	2103	1406	12270	<http://www.wikidata.org/prop/direct/P2860>
189	590	401	3452	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
135	1482	990	8744	<http://www.wikidata.org/prop/direct/P2093>
99	215	142	1281	<http://www.w3.org/2004/02/skos/core#altLabel>
96	924	619	5432	<http://schema.org/dateModified>
96	651	438	3789	<http://schema.org/about>
96	979	655	5825	<http://schema.org/version>
94	491	333	2926	<http://www.wikidata.org/prop/direct/P31>
40	357	241	2132	<http://www.wikidata.org/prop/direct/P1476>

We will focus on index build time, index size, and index retrieval time in the next parts.

7.2.2. Index build time

We consider configurations with 128, 256, 512 and 1024 max nodes per block. For cut depth we tried only 0 and 10, and for all scenarios we use k^2-tree total depth of 32.

Table 7.3: Index build times for different configurations

Configuration (N-C-H)	Build Time (hours)
128-0-32	6.5
256-0-32	7.1
512-0-32	10.4
1024-0-32	12.8
128-10-32	4.8
256-10-32	5.1
512-10-32	5.4
1024-10-32	6.7

The table 7.3 has the build times for each configuration. Note that build times are reduced considerably for the cut depth (C) of 10, compared to the cut depth of 0. This is because the superior part of the trees are just pointers which need much less processing than the compact part during insertions, specially when there are tree splits that are very expensive for the block trees. In contrast, the cut depth of 0 doesn't have a superior part containing a tree of pointers, it is completely a block tree.

Another thing to notice from the table 7.3 is that with higher values of max nodes per block (N), the more time it takes to build the indices. This can be explained by the tradeoff between compression and peformance in which larger blocks means more compression by having more data in bitwise representation and less pointers.

7.2.3. Index size

In this subsection, we study the index size, both in memory and in disk.

7.2.3.1. Size in memory

Here we have measurements for the size that the indices take once they are loaded in memory. In figure 7.1 we took several k^2-tree configurations, measured their size in memory, and aggregated them by average and percentiles 50, 95 and 99. This was done by recursively scanning k^2-trees and measuring the sizes of each block, pointer and metadata for every tree, and then summing them.

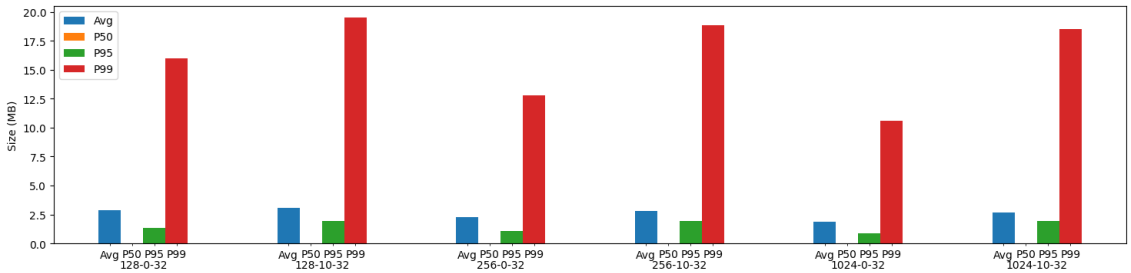

Figure 7.1: Index size in-memory for several configurations in MB

Apart from the averages and percentiles that help us understand behavior in most cases, it can be relevant to have some idea of the indices that have more data. In figure 7.2, we computed the maximum sizes for each configuration. If we focus on the 128-0-32 configuration, this implies that we would need to have at least nearly 5GB of capacity to allow all indices to be included in the cache-replacement flow, otherwise it would start ignoring indices of that size.

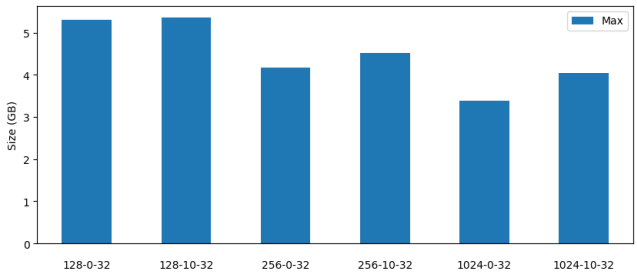

Figure 7.2: Index size in-memory for several configurations, max size in GB

In figure 7.3, we have the sum of sizes for each configuration. This can help us decide if we want to load all indices in memory or not.

Both figure 7.2 and 7.3 gives us some idea of the compression level based on the `N` and `C` parameters. The `N` parameter indicates the block size, as it gets higher we are representing more edges in a tree by the blocks' bit-sequence, as opposed to using regular memory pointers of 8-bytes (in 64-bit processors). As we increase the `C` value, we are increasing the size of the pointer tree on top of the block trees, which explains the size increase.

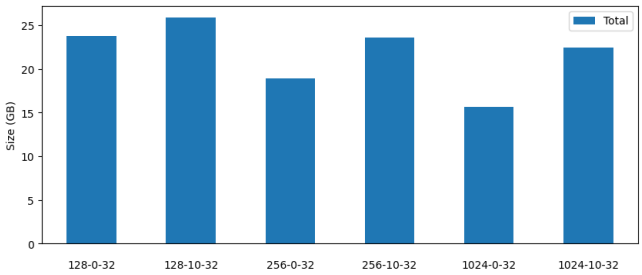

Figure 7.3: Index size in-memory for several configurations, total size in GB

7.2.3.2. Size in disk

Here we show some measurements to sizes of indices on disk. In figure 7.4, the measurements are classified by k^2-tree configuration, and for each one, we take average and percentiles 50, 95 and 99.

Interestingly, these occupy 25-50 % less space in disk than in memory, the main reason for this is that in the disk serialization we omit the explicit representation of pointers, and instead used an implicit representation based on the ordering of blocks.

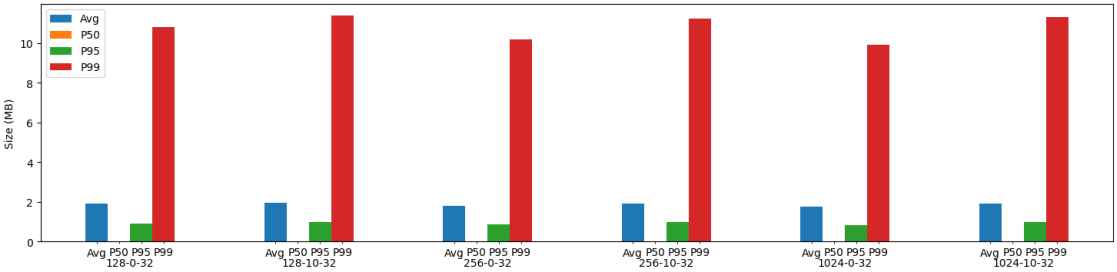

Figure 7.4: Index size in-disk for several configurations in MB

In figure 7.4, we measure the maximum size a k^2-tree can occupy for each configuration on disk. When comparing with figure 7.2 we can see the difference in size can be quite drastic due to pointers for referencing in memory. Also, the difference tends to decrease when increasing the N parameter that affects block sizes.

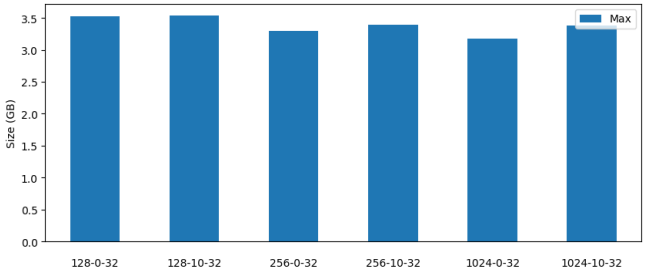

Figure 7.5: Index size in-disk for several configurations, max size in GB

In figure 7.6 we have the total size in disk occupied by all k^2-trees for each configuration measured. Following the same idea as above, when compared with figure 7.3 we can notice an even more drastic difference when it is loaded in memory, specially for lower N values.

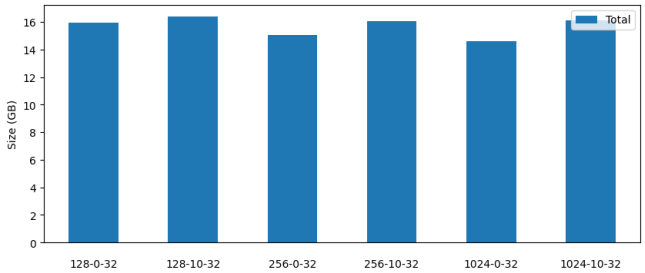

Figure 7.6: Index size in-disk for several configurations, total size in GB

7.2.4. Index loading

In this subsection, we measured the time it took to pick a k^2-tree index from disk and loading it into memory so it is ready to be used.

In figure 7.7, following the same idea as for size measurements, we measured the loading times in average and percentiles 50, 95 and 99. Seeing that P99 is under 200 ms in all cases shows that we are generally loading indices very quickly, and indicates that the use of a replacement strategy such as LRU should not have much impact on performance when it needs to load indices.

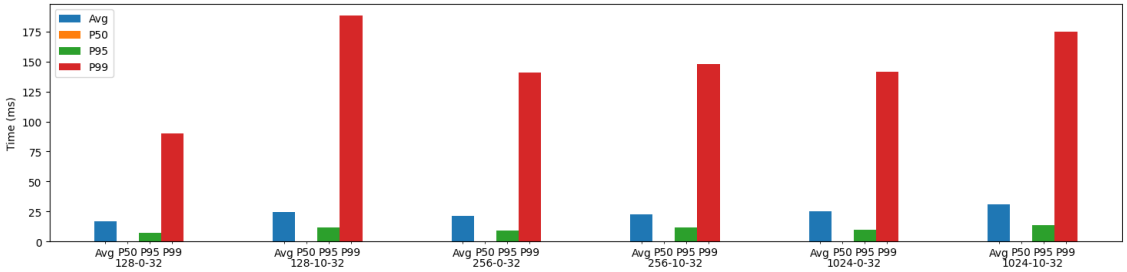

Figure 7.7: Index loading times for several configurations, times in milliseconds

In figure 7.8, we have measurements for the maximum loading time. If we take a look at the `128-0-32` configuration and the same time look at the table 7.2, we can see that the value from the bar graph shows the loading time for the index represented by resource <http://schema.org/description>, which at the same time is the index with more triples and with more size both in memory and disk. 30 seconds of loading time can be a big problem in production scenarios, so these big indices probably need to be dealt with differently from the rest, for example if it is an index that has high usage demand it might be desirable to have it loaded at all times, outside of the LRU flow, or in the contrary case, to never load it.

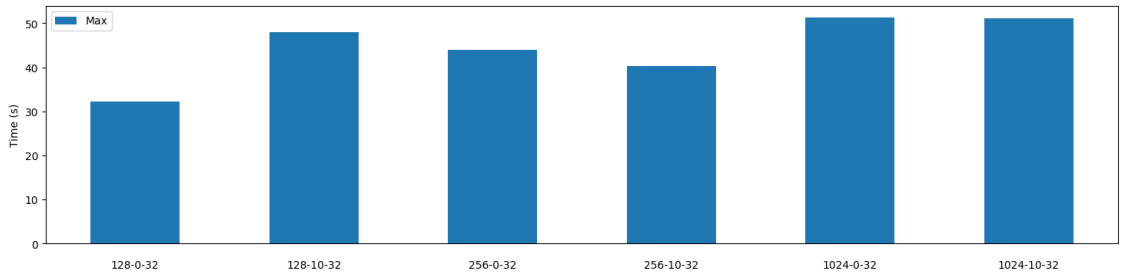

Figure 7.8: Index loading times for several configurations, max times in seconds

7.3. Querying

For measuring query performance by time, we considered as our dataset a Wikidata snapshot which is indexed with the k^2-trees and another dataset of real user queries on Wikidata from [30]. Some queries were filtered out so we could measure the effect of the cache more accurately without too many variables to be considered. For this, it was necessary to remove any queries that would use the NodeId to resources map inside Jena, for example, FILTER or GROUP queries that need to inspect the real resources. We also removed some other queries with operators like SERVICE, GRAPH and path queries for simplicity.

Each of the queries was wrapped with a COUNT operator, so we don't materialize results or use the resources dictionary in Jena.

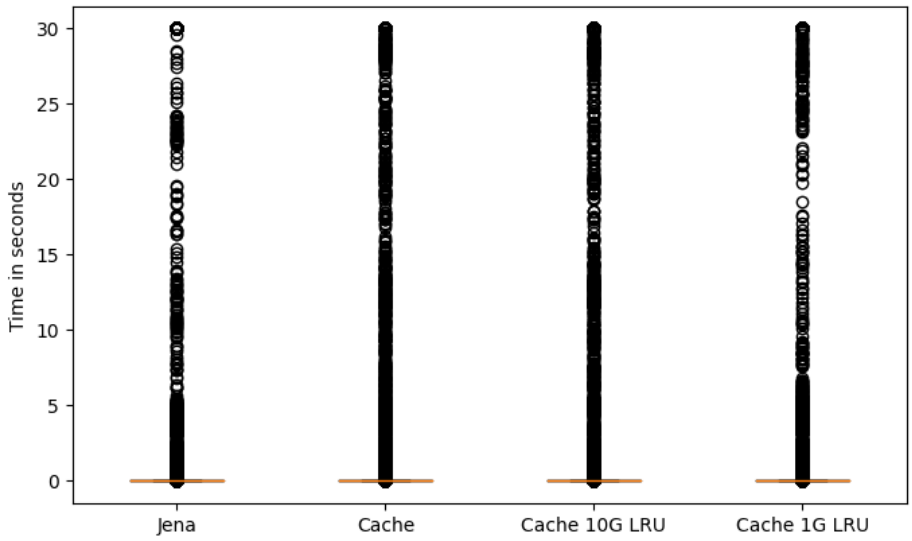

Figure 7.9: Boxplot with raw time results for 19628 queries. 1) Jena: without using our caching system. 2) Cache: Using the caching system without replacement, loading everything on memory before the first query is executed. 3) Cache 10G LRU: caching system with 10GB capacity using LRU replacement. 4) Cache 1G LRU: 1GB of capacity

To evaluate more queries we also put a timeout of 30 seconds. This is realistic because more expensive queries will be stopped like this in most services.

In the figure 7.9 we can see that most queries in all scenarios considered are near the 0 seconds by looking at the orange horizontal bars, where the 25 % to 75 % percentiles are. Here we considered 19628 user queries.

The boxplot doesn't say much comparatively, but at least it says that most results take very little time and the caching system doesn't make things significantly worse on the average query.

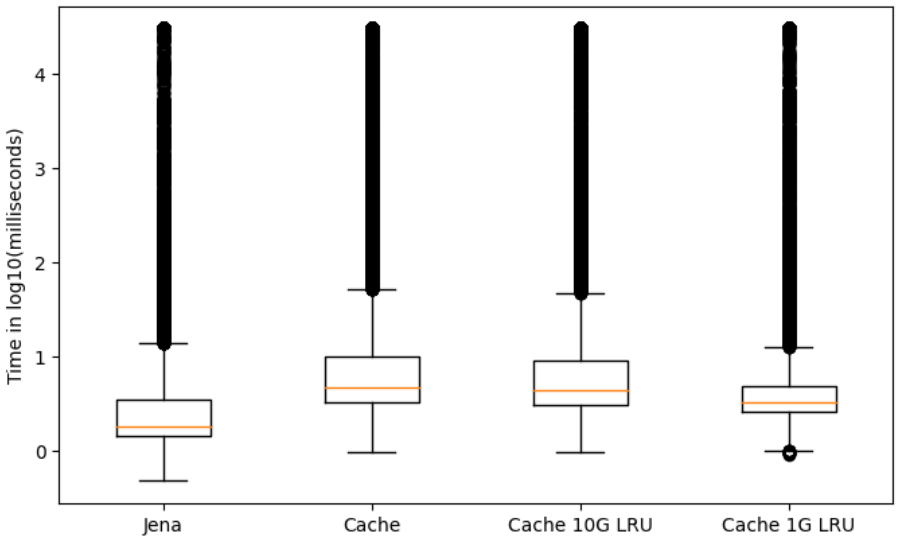

Figure 7.10: Boxplot with $log_{10}()$ time results in milliseconds for 19628 queries. 1) Jena: without using our caching system. 2) Cache: Using the caching system without replacement, loading everything on memory before the first query is executed. 3) Cache 10G LRU: caching system with 10GB capacity using LRU replacement. 4) Cache 1G LRU: 1GB of capacity

In figure 7.10, we took $log_{10}()$ to the times in milliseconds by using the same data as in figure 7.9, here we can visualize more clearly that most queries take a few milliseconds to process in all configurations.

Since we want to analyze cases where the caching system will make a difference for worse or better, we take a different approach for analyzing results. First, we join the result sets of Jena only and Cache without replacement by matching the same query. The result table of this is such that each row contains times for both cases on the same query.

Now, we take the two disjoint sets, one in which Jena alone wins and the other one where the cache wins.

Next, we take differences in time for each set, where the left side is the time of the loser and the right side is the time of the winner, so the difference is always positive (winner is lesser time).

Having these differences, we compute the percentiles and this will result in two curves, one representing the cache winning and the other one Jena alone winning.

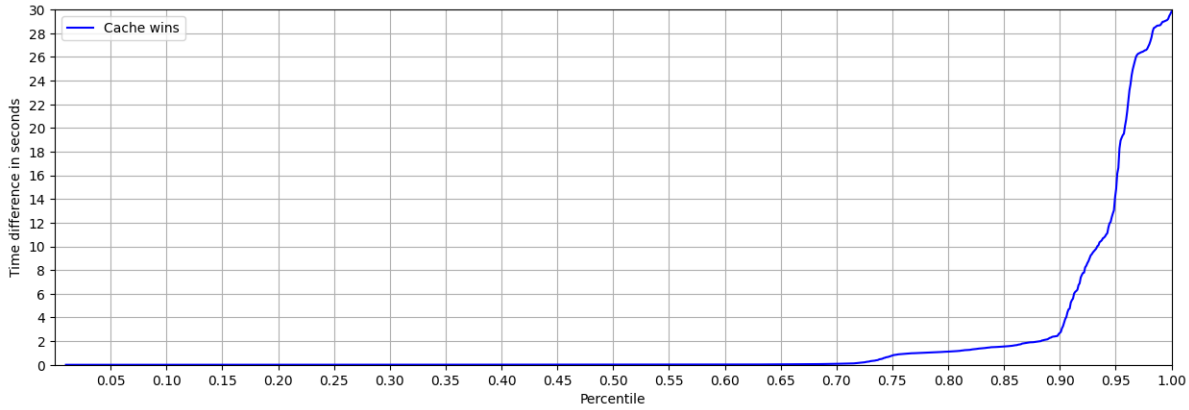

Figure 7.11: Percentiles of time differences where the cache wins over Jena. The cache wins 2976 times

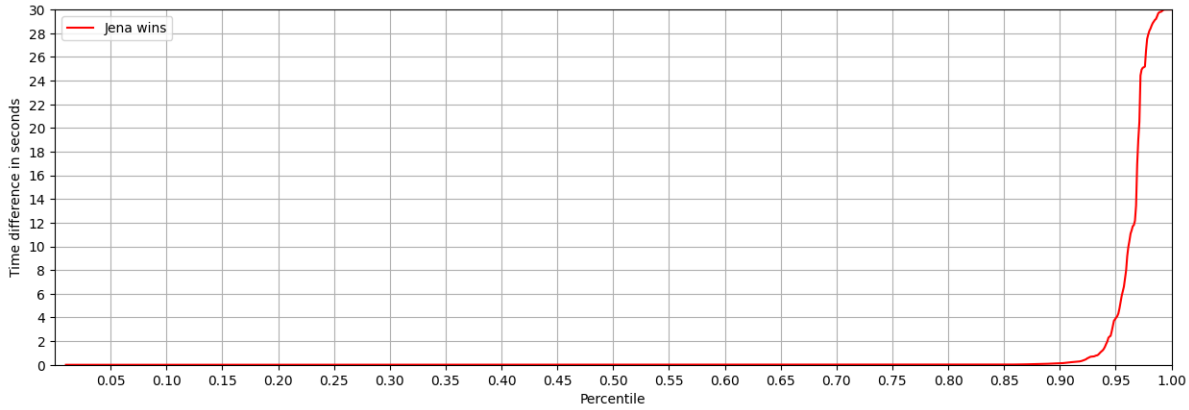

Figure 7.12: Percentiles of time differences where Jena wins over the cache. Jena wins 16401 times

Figures 7.11 and 7.12 show the results for this. Here we can see that at around percentile 72 % we start having some real improvement by using the cache, while Jena alone keeps the same near 0 seconds of improvement over the cache. This means that 28 % of the winning results for the cache have significant improvement, that is 830 results.

When approaching percentile 92 %, some queries start showing better performance with Jena alone, while we keep having queries where the Cache wins. This means that 8 % of winning results for Jena have significant improvement, that is 1306 results.

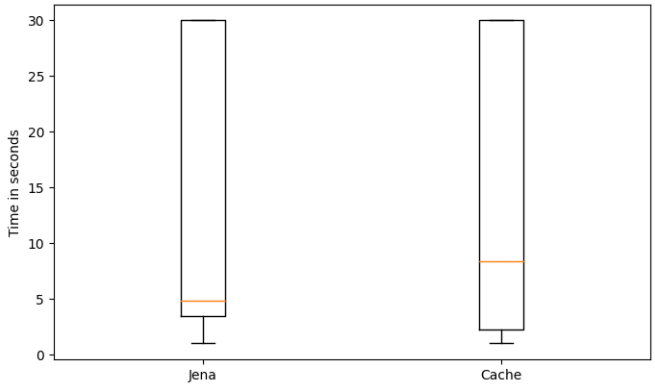

Figure 7.13: Boxplot for times above one second. Jena vs Cache fully preloaded (no LRU).

In figure 7.13, we filtered results to those that took above one second. The median is higher for the cache as it can be seen. One thing we noticed from this is that if we remove the timeouts the median becomes lower than Jena's. There are 657 timeouts for the Cache, while there are 467 for Jena. This can be explained by queries favoring the locality of CPU cache offered by B+trees, while for the k^2-trees that locality is somewhat lost due to decompression requirements, which tends to be more accentuated when there are joins that need to traverse the same tree paths multiple times.

In this case, we didn't consider the cache with replacement, because that can only be worse than the cache with all preloaded indices.

7.3.1. Timed out queries

It is also interesting to study queries that timed out in any of the two cases. For this, we select those and run them again, but this time with a timeout set to five minutes.

We also include a different cache configuration for the analysis. Our default go-to configuration is 128 nodes per block, pointer three (mixed) of depth 0 and covering a NodeId space up to 2^{32} possible coordinates. The new configuration we are testing here is similar, with the same values, except the pointer three is of depth 10. We identify it here as "Cache 128-10-32".

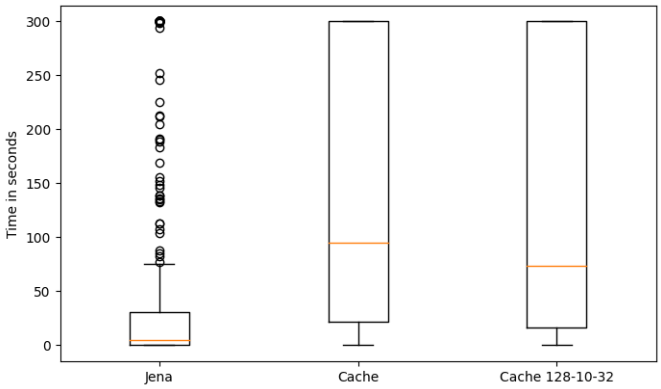

Figure 7.14: Boxplot comparison using a timeout of 5 minutes for previously timed-out queries at 30 seconds. Number of queries: 560.

In figure 7.14, we measure the times of all previously timed-out queries. In this case, we still have time outs which are, 214 for default cache configuration, 201 for the 128-10-32 configuration and 74 for Jena alone. Those are all gathered at the 300-second mark in the boxplot.

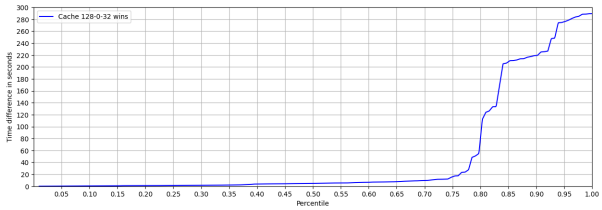

Figure 7.15: Percentile comparison using time-out of 5 minutes for previously timed-out queries at 30 seconds. Cache 128-0-32 has 163 winning results

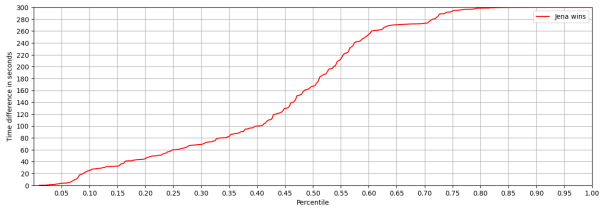

Figure 7.16: Percentile comparison using time-out of 5 minutes for previously timed-out queries at 30 seconds. Jena has 355 winning results.

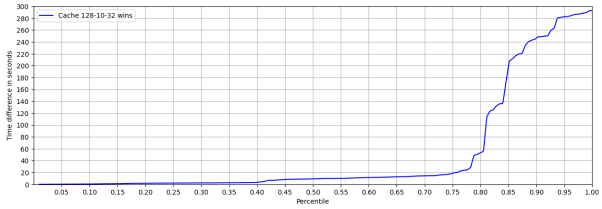

Figure 7.17: Percentile comparison using time-out of 5 minutes for previously timed-out queries at 30 seconds. Cache 128-10-32 has 171 winning results.

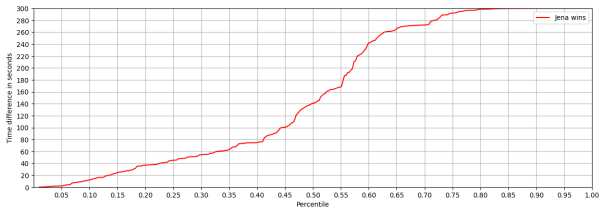

Figure 7.18: Percentile comparison using time-out of 5 minutes for previously timed-out queries at 30 seconds. Jena has 347 winning results.

To compare on a query-by-query basis we measured time differences percentiles as before, and also added a plot for the new cache configuration.

For these plots to make sense it is relevant to consider the number of values each curve has. These curves quantify how many of the winning cases achieve considerable speed-up when compared to the other scenario. Both cases are similar, so we focus on one. In figures 7.15 and 7.16, we see that the Jena curve has a smoother increase in the time difference, which means that if we consider all cases it is winning ordered increasingly by their time difference to the cache result, we can see improvements early on. This is opposed to the cache curve, which has a sudden increase near the percentile 75 %, or said differently, 25 % of the cases in which it is winning it achieves very considerable performance over Jena. The same applies to figures 7.17 and 7.18, where we can see a slight improvement in the cache over Jena.

7.3.2. Few queries run in cold boot

One explanation for why B+trees are so often winning is that the Jena implementation caches good portions of their blocks in memory. So in the next experiment, we run a few of the queries where Jena won against the cache, but this time with cold storage. We take the top 10 results where Jena was better when compared to the cache results in the query to query time difference comparison.

The results show that Jena can be done with these 10 queries almost immediately. When examining the queries that are being run, we noticed that every one of them has a LIMIT instruction with a small number. This explains a lot of the wins for Jena. The cache was configured to compute the minimum of resulting rows between all results and 10 million rows before sending any results back to Jena. Jena is stopping much earlier than that.

We could solve this by trying to adjust the number of rows computed before partially reporting results from the caching system, but this also can be tricky to determine. Too low of a value can add more network overhead, since it would require sending more messages to send the full response.

Instead, we choose to run the same queries without the LIMIT restriction and compare with that.

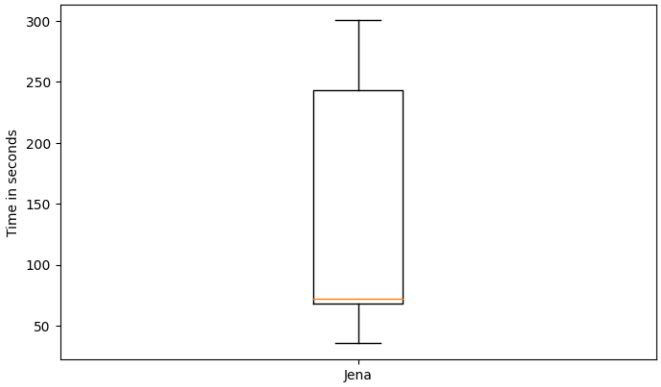

Figure 7.19: Running the 10 worst queries for the caching system in Jena, modified so that the LIMIT instruction is removed. Each run is done after a cold boot of Jena.

Figure 7.19 has the time results for running Jena against the worst 10 queries for the Cache, which all timed out. These were also run in a cold boot, which means the server was started every time for each of the query runs.

Even with that, we get better results with Jena. Given all of this, we move our focus to

study queries one by one.

7.3.3. Some queries with better performance with Jena

```
SELECT (COUNT(*) AS ?count) WHERE {
SELECT ?var1 ?var2 ?var3 ?var4
WHERE {
   BIND ( <http://www.wikidata.org/entity/Q2095> AS ?var5 ).
   ?var1 <http://www.wikidata.org/prop/direct/P360> ?var5 .
   ?var3 <http://www.wikidata.org/prop/direct/P361> ?var1 .
   OPTIONAL {
      ?var1 <http://www.w3.org/2000/01/rdf-schema#label> ?var2 .
   }
   OPTIONAL {
      ?var3 <http://www.w3.org/2000/01/rdf-schema#label> ?var4 .
   }
}}
```

Figure 7.20: P360 has 214k points, P361 has 3.1M points, label has 483M

In figure 7.20 we have a query with two OPTIONAL. It turns out that each OPTIONAL content is processed independently of other BGP sections and in this case it would be executed many times with different ?$var1$ and ?$var3$ bindings. This is good, but the problem with our setting is that each time it goes to execute one OPTIONAL section, with each binding, it has to incur in a network request. In this particular scenario the #label predicate takes part in the two OPTIONALs, and as this is one of the biggest indices it can also add considerable cost to the query execution.

```
SELECT (COUNT(*) AS ?count) WHERE {
SELECT *
WHERE {
   ?var1 <http://www.wikidata.org/prop/direct/P31>
      <http://www.wikidata.org/entity/Q2464485> ;
   <http://www.w3.org/2000/01/rdf-schema#label> ?var2Label .
   OPTIONAL {
   ?var1 <http://www.wikidata.org/prop/direct/P580> ?var3 .
   }
   OPTIONAL {
   ?var1 <http://www.wikidata.org/prop/direct/P582> ?var4 .
   }
}}
```

Figure 7.21: P31 has 94.7M points, label has 483M points, P580 has 605K points, P582 has 534K points

In figure 7.21 we have a similar situation, in which the OPTIONAL content itself has many less points, but still this can be pretty expensive on the cache, because it would incur in many network requests.

```
SELECT (COUNT(*) AS ?count) WHERE {
SELECT *
WHERE {
   ?var1 <http://www.wikidata.org/prop/direct/P351> ?var2 ;
   <http://www.wikidata.org/prop/direct/P352> ?var3 .
}}
```

Figure 7.22: P351 has 768K points, P352 has 620K points

In figure 7.22 we don't have that many points, but surprisingly Jena won anyway. This time everything should run on the cache without too many network requests. In this case, we have a full predicate index scan, followed by a scan with one variable triple pattern. Jena has some advantage here because it has much more freedom for which indices to choose. Recall that Jena has SPO, POS, and OSP indices. For each one of the ?$var1$ it sees on P351 it can choose to scan SPO, and the good thing about SPO index is that all ?$var3$ results will be found physically one after the other. The other good thing there is about this is that, as these are small predicates, Jena will probably load them once into memory completely, and then apply the remaining of the work there, instead of fetching it from disk each time. The cache, on the other hand, has to make one band scan on the P352 predicate index for each of the points in P351. This means a 768K times 620K operation, which will be quite expensive.

```
SELECT (COUNT(*) AS ?count) WHERE {
SELECT ?var1Label ?var2Label ?var3Label ?var4Label
WHERE {
   <http://www.wikidata.org/entity/Q238231>
   <http://www.w3.org/2000/01/rdf-schema#label> ?var1Label .

   <http://www.wikidata.org/entity/Q17278>
   <http://www.w3.org/2000/01/rdf-schema#label> ?var2Label .

   <http://www.wikidata.org/entity/Q843905>
   <http://www.w3.org/2000/01/rdf-schema#label> ?var3Label .

   <http://www.wikidata.org/entity/Q4115331>
   <http://www.w3.org/2000/01/rdf-schema#label> ?var4Label .
}}
```

Figure 7.23: label has 483M points

In figure 7.23 we have a query that only has the #label predicate, multiple times. For Jena, this can be easy because it has to scan SPO index 4 times, with only object variables

each time, and then make a Cartesian product of those. If the contents of each of the 4 results are small, it can have them all in memory, and the most expensive part would be to make a few disk random accesses for each of the 4 results. Once the results are loaded into memory, it only needs to multiply the sizes of each part, because this is a COUNT operation, it doesn't have to materialize anything. But even if it is not that smart, and the count is relatively small, it won't have much trouble in finding the result, because after the initial fetching from disk, it should only make sequential access memory operations when the 4 parts it operates are sufficiently small.

In the cache, on the other hand, we are always making random access in-memory operations. So the real comparison will not be disk versus memory, but few disk accesses with sequential memory access versus all random memory access.

7.3.4. Some queries with better performance with k^2-trees

```
SELECT (COUNT(*) AS ?count) WHERE {
SELECT *
WHERE {
   ?var1 <http://www.wikidata.org/prop/direct/P31>
         <http://www.wikidata.org/entity/Q5> .
   ?var1 <http://www.wikidata.org/prop/direct/P735>
         <http://www.wikidata.org/entity/Q923> .
}
}
```

Figure 7.24: P31 has 94.7M points, P735 has 5.8M points

In figure 7.24, we have a query with two predicates where their k^2-tree indices are quite big. Based on a Jena's typical optimization that applies for both the cache and Jena, in which it tends to start with parts that would yield less results (so it can process less data to achieve the same result), the cache will make one band scan over the index for P735, and for each result, it will make a point scan on P31. Jena, on the other hand, will probably use the POS index for getting $?var1$ values on P735-Q923 and with that, it has to go to some of the three indices to see if the triples with P31-Q5 are there. This takes long probably because it doesn't have any other way than do this by incurring in lots of disk random accesses to find if the triple exists.

```
SELECT (COUNT(*) AS ?count) WHERE {
SELECT ?var1
WHERE {
   ?var1  <http://www.wikidata.org/prop/direct/P31>
          <http://www.wikidata.org/entity/Q5> .
   ?var1  <http://www.wikidata.org/prop/direct/P4072>
          ?var2 .
}}
```

Figure 7.25: P31 has 94.7M points, P4072 has 205 points

In figure 7.25, we have something similar than before, but this time P4072 has only 205 points. In this case, it is not completely clear what processors are doing. One way would be to scan ?*var*1 values from the P31-Q5 portion, and then evaluate those on P4072. Other way would be to completely scan P4072, and then evaluate ?*var*1 values from there in P31-Q5. The second alternative seems to be the right answer for both the cache and Jena, because the left side of the join has fewer elements, but it also means having to make in the order of 205 disk random accesses for Jena. So there is the possibility that Jena would do better by choosing the first alternative to make fewer disk random accesses, but it is not being smart enough to consider that.

The cache seems to do better because it is only having to make those random accesses in memory.

```
SELECT (COUNT(*) AS ?count) WHERE {
SELECT *
WHERE {
   ?var1  <http://www.wikidata.org/prop/direct/P31>
          <http://www.wikidata.org/entity/Q5> .
   ?var1  <http://www.w3.org/2000/01/rdf-schema#label>
          ?var2 .
   ?var1  <http://www.wikidata.org/prop/direct/P17>
          <http://www.wikidata.org/entity/Q298> .
}}
```

Figure 7.26: P31 has 94.7M points, label has 483M points, P17 has 13.8M points

In figure 7.26, we have many operations over quite big indices. In this case, it seems to be less surprising that the k^2-trees win, because Jena would be forced to do a large number of random disk accesses.

7.3.5. Network request optimization

Based on the results from subsection 7.3.3, where all BGP processing was being done in the cache, and incurring in too many network requests due to some queries having several

BGP being processed, we make an optimization to retrieve only the first BGP from the cache so that there is only one BGP request to cache per query (there can still be more than one request to retrieve a large result in several parts on the same BGP). This optimization will help with queries having OPTIONALs and other categories needing more than one BGP per query.

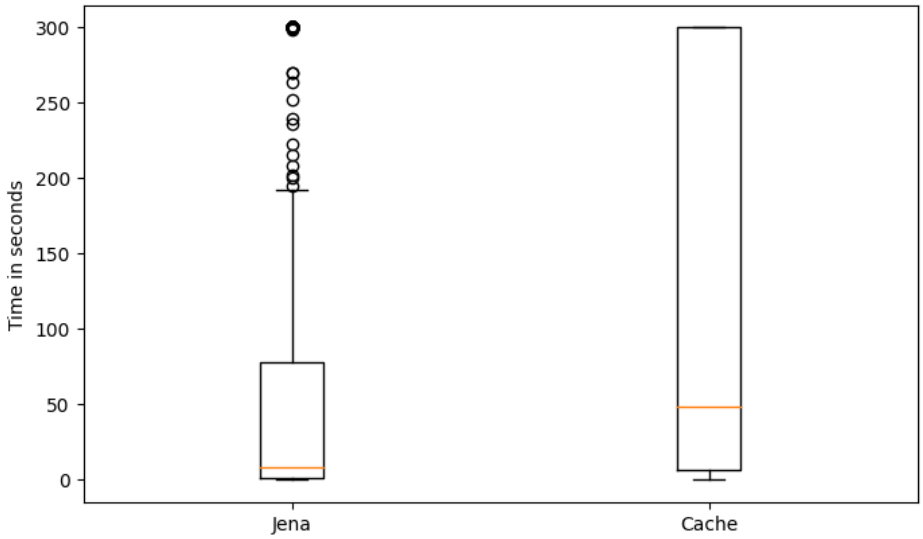

Figure 7.27: Result comparison for Cache and Jena with network request optimization

In figure 7.27, we can see the result of this optimization. While it doesn't seem much different, it reduced the median for the Cache significantly. In figure 7.14 the median was 130 seconds and with the optimization it became 48 seconds. Also, the number of times that the cache is winning now is 234 against the 292 where Jena wins. Previously, we had 163 queries where the cache was winning and 355 where Jena was winning. Note that the sum of queries is different in each scenario because in the optimized scenario some queries are not timing out with the optimized cache usage, while before they were timing out in both cases.

7.3.6. Selective usage of the cache

Due to the cache not being the best solution at all times, we also implement a solution which uses the cache only when it can yield the same result faster than Jena. Our approach was to begin running the BGP evaluation both in the cache and Jena, we detect which one is able to give some results faster and cancel the slower engine, while waiting for the faster one to complete.

This should be effective because we are not wasting much on doing the initial speed evaluation to select the processing engine, it requires a few results while running two engines at the same time. The initial results are a good indication of which one will be faster to complete, because it completes one full step of iteration of the processing, where Jena might have done some random accesses to the disk or the cache some expensive operation.

The solution also adds some cost to the cache because we now need to make its operations cancellable, so we don't make any unnecessary processing, which means asking with some frequency if the query has been canceled in a critical section of the code. As we make this

frequency manageable, we can reduce this cost so it becomes barely noticeable. At the same time, a cancellation message is needed to be sent from Jena to the Cache.

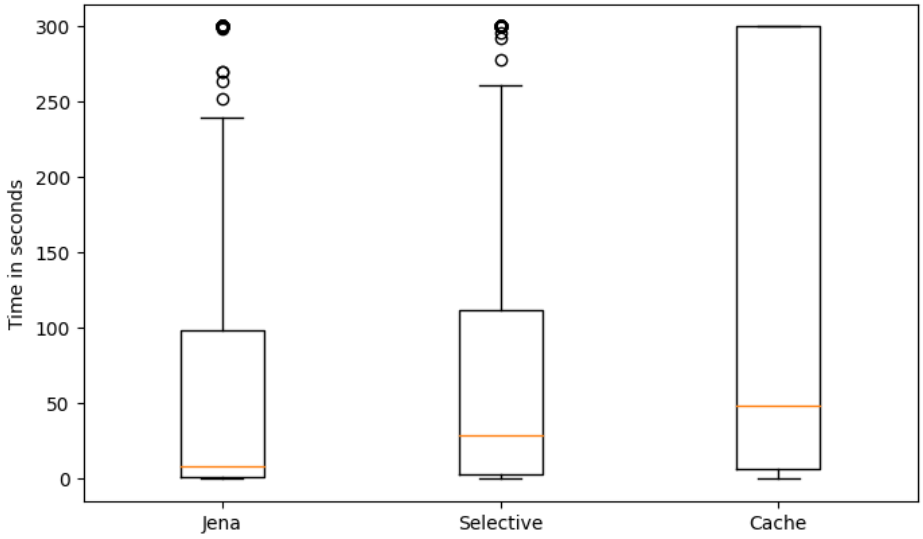

Figure 7.28: Result comparison for Jena, Selective and Cache

In figure 7.28, we have measurements that included the selective usage of the cache. The aggregated visualization shows that the selective results are worse than Jena, but it decreased the number of timeouts, Jena has 101 timeouts, while the selective strategy has 95 timeouts. For this experiment, we took a very conservative approach of selecting the cache, such that we only need one result of Jena first to choose Jena, but 100 results from the Cache before any of Jena. This can cause a discarding of a faster execution on the Cache, due to picking the Jena engine too soon. The slightly worse results can be explained by the cost of synchronization during the selection phase. Also, this time for each case we are using 533 queries.

7.4. Updates

For the updates, we have two kinds of experiments. The first is about random updates, where we only work with k^2-trees in the C++ layer. The second experiment involves Wikidata snapshots and experimenting with the complete Jena/cache system.

7.4.1. Random updates

In the random updates experiments, we select several configurations of k^2-trees. For each one of the `N-C-H` configurations, we run four different experiments based on cluster size. The cluster size indicates how many points around a randomly selected point are being inserted. The chosen cluster sizes are 1, 256, 1024 and 4096.

Each experiment first inserts a million points in a k^2-tree, there are only two time measurements, one before the first point is inserted and the second one after the last point is inserted. With this, an average time is calculated for the insertion of a single point. After all insertions are done, the sequence of inserted points is used to delete all the elements, and the average deletion time is computed in the same way.

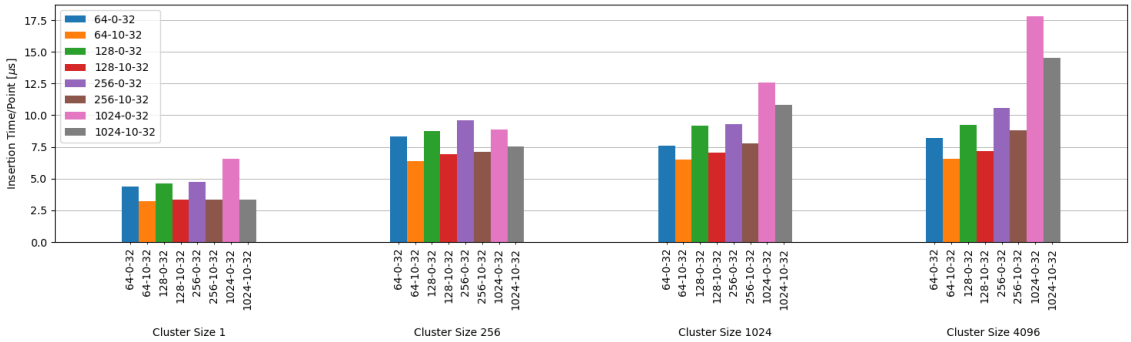

Figure 7.29: Average insertion time per point for different configurations of k^2-trees and cluster size.

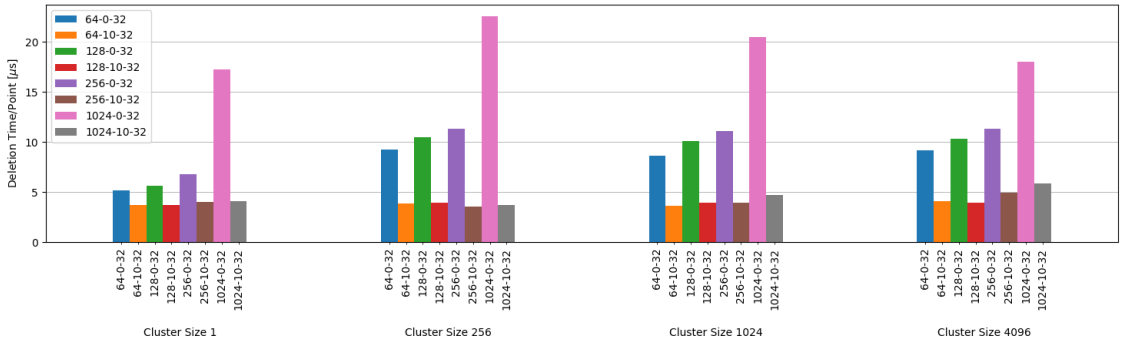

Figure 7.30: Average deletion time per point for different configurations of k^2-trees and cluster size.

In figure 7.29, we have the results of the experiments only for insertions. Here it is clear that as we increase the cluster size, the insertion time becomes higher for each group of N-C-H configurations. This can be explained by the increase of block splits during insertions when the points are more condensed. There is also a clear improvement when using a value of 10 for C, compared to when using 0. This justifies the use of a pointer tree on top of block trees.

Similarly to the insertion results, in figure 7.30 we have the results for deletions. In this case, the cluster size increase doesn't seem to affect the deletion speed significantly when looking at the sizes 256, 1024 and 4096, but there does seem to be a big difference between cluster size 1 and all the other sizes. The small differences are explained by the fact that the merge block operation is less expensive than the split block operation (there is no need to find a frontier node) and therefore it is less sensitive to cluster size differences. On the other hand, for cluster size 1 the chance of the merge block operation is reduced. Other thing to be noticed is that the configuration 1024-0-32 does seem to be more expensive in all cluster size results, and dramatically more so than 1024-10-32. This is not so easy to explain, but it indicates that some undesirable behavior is being triggered with that configuration.

7.4.2. Wikidata insertion experiment

For measuring insertion time, we consider two Wikidata snapshots taken at different times and extract randomly 10000 triples that are in the newest one that are not in the oldest one.

Then, we make 5000 insertions on each case: Jena alone, and Jena with the cache.

The insertions in this case are made in the form of SPARQL insertions, where each triple to be added is wrapped in a SPARQL insertion statement.

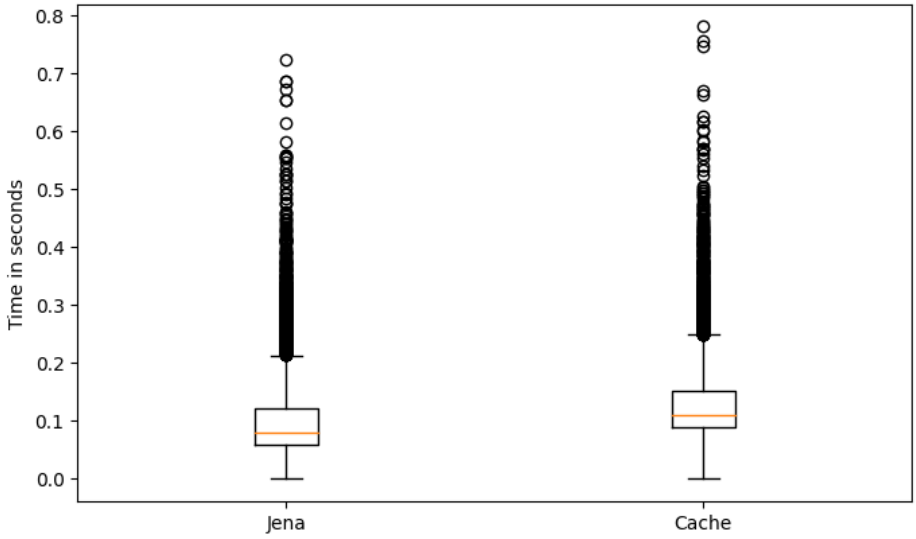

Figure 7.31: Insertions time. Each case was tested with 5000 triple insertions

In figure 7.31, we have a boxplot comparison of Jena running insertions by itself and Jena together with cache running insertions.

7.5. WatDiv experiments

In the WatDiv experimentation we chose only one k^2-tree configuration, 128-10-32. There are 1,091,437,702 triples, which stored in disk amounts for 8 GB and in memory for 12 GB.

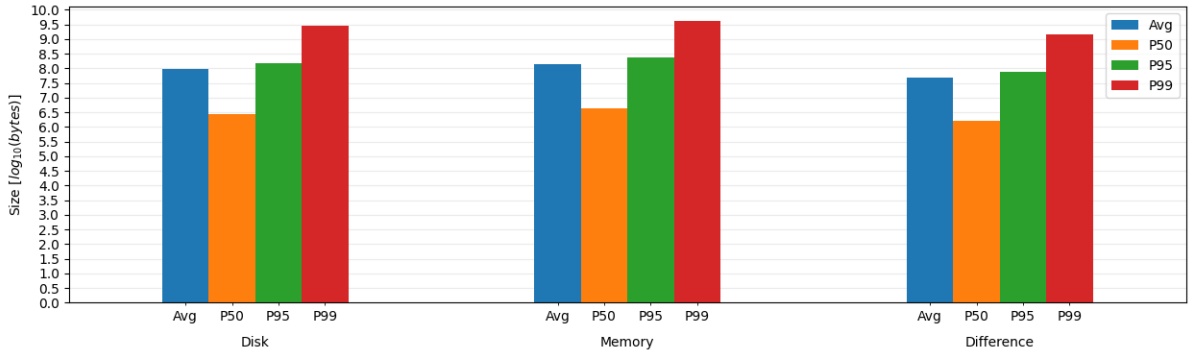

Figure 7.32: Index sizes of WatDiv

In figure 7.32, we have index sizes by some percentiles and by their average. We show both size in disk and size in memory for comparison. As it is expected, size in disk is lower by a significant amount, which can be seen on the third barplot in the figure. Note that the y-axis is shown in the $log_{10}()$ of the sizes, which allows us to visualize the P50 and P95 better at the same time.

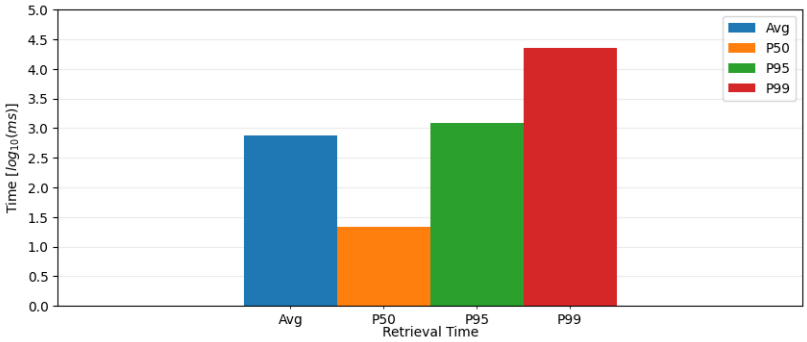

Figure 7.33: Retrieval times for WatDiv

In figure 7.33, we repeated the visualization from the sizes for the retrieval times. This one shows that we can load most indices in less than a second, and up until the median, the times are below 32ms.

WatDiv offers several query templates that are used to generate queries, we worked with the query templates C1, C2, F1, F3, F4, L1, L4, S1, S2, S4, S5. The other templates from WatDiv were tried in experiments, but they caused technical difficulties such as query parsing errors from Jena or timeouts for all engines, so they were skipped. Each template generates a single query that is repeated 10 times in our experiments.

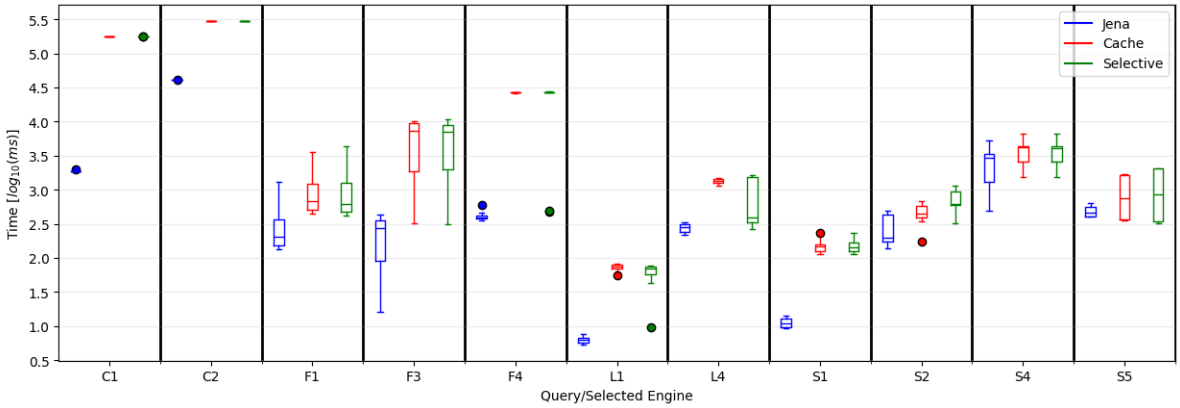

Figure 7.34: Results from the WatDiv dataset experimentation, categorized by query templates and engines

In figure 7.34, we have the results from running the queries categorized by the query template and engine. This time Jena wins in all cases, in contrast with the results from Wikidata where there were some winning cases for the Cache.

```
SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
   ?v0 <http://schema.org/caption> ?v1 . # T1: 249213 results
   ?v0 <http://schema.org/text> ?v2 . # T2: 752007
   ?v0 <http://schema.org/contentRating> ?v3 . # T3:  750734
   ?v0 <http://purl.org/stuff/rev#hasReview> ?v4 . # T4: 14779456
   ?v4 <http://purl.org/stuff/rev#title> ?v5 . # T5: 4501138
   ?v4 <http://purl.org/stuff/rev#reviewer> ?v6 . # T6: 15000000
   ?v7 <http://schema.org/actor> ?v6 . # T7: 1670842
   ?v7 <http://schema.org/language> ?v8 . # T8: 626305
}
```

Figure 7.35: C1 query

In figure 7.35, we have the C1 query. In this query, we added a comment with the number of results for each triple pattern, before any joins are made. This is not completely accurate for understanding the query cost, since many times, with the help of indices, the query engine has to scan only a subset of the triples for query evaluation, but it helps on having a rough estimate of the minimum cost.

For the query C1, we consider an example query plan from Jena/TDB, which might not necessarily be the real query plan. It could take the first triple pattern T1 and scan all the triples having as predicate <http://schema.org/caption>, which are 249213 triples to get from the POS index. Supposing that as a next step the optimizer will pick the triple pattern T3: "?v0 <http://schema.org/contentRating> ?v3 .", for each ?v0 from the first scan, it could pick the SPO index to find the key (?v0, <http://schema.org/contentRating>) and get all the objects having that key. All of those objects should be stored physically in consecutive order within the B+tree. The number of objects to find are in the worst case 249213. The process goes on, and as all the triple patterns are connected by some variable, each index scan does a random access search using a key and it is followed by a sequential search until reaching the last triple pattern.

Now we do the same exercise for the k^2-tree engine, for which we established in the implementation that the same query plan as Jena was to be followed. For the first step, there is a scan of all triples having predicate <http://schema.org/caption>, which means finding the k^2-tree index in constant time using the predicate's NodeId in a simple lookup to a hash table and then getting all triples with the full scan operation from the index. There are 249213 triples to be scanned, same as before, and each of those triples needs to be reconstructed from a compressed representation which involves several random accesses for each triple. For each triple from the full scan on T1, there is a band scan on the k^2-tree index corresponding to <http://schema.org/contentRating> from the triple pattern T3.

The point of these examples is that there is an advantage for the B+trees over the k^2-trees in terms of query evaluation performance, which is that the triples in the B+trees that are needed for the query are more readily available than the k^2-trees. The triples don't need to be reconstructed and they are all stored sequentially. Another advantage is that the B+trees don't need be fetched from disk all the time, it may be that large chunks of them are temporarily residing in memory. The disadvantage of the B+trees over the k^2-trees is that they take considerably more space, and in order to beat them, they need to occupy much more memory than the k^2-trees, which can be restrictive in many cases.

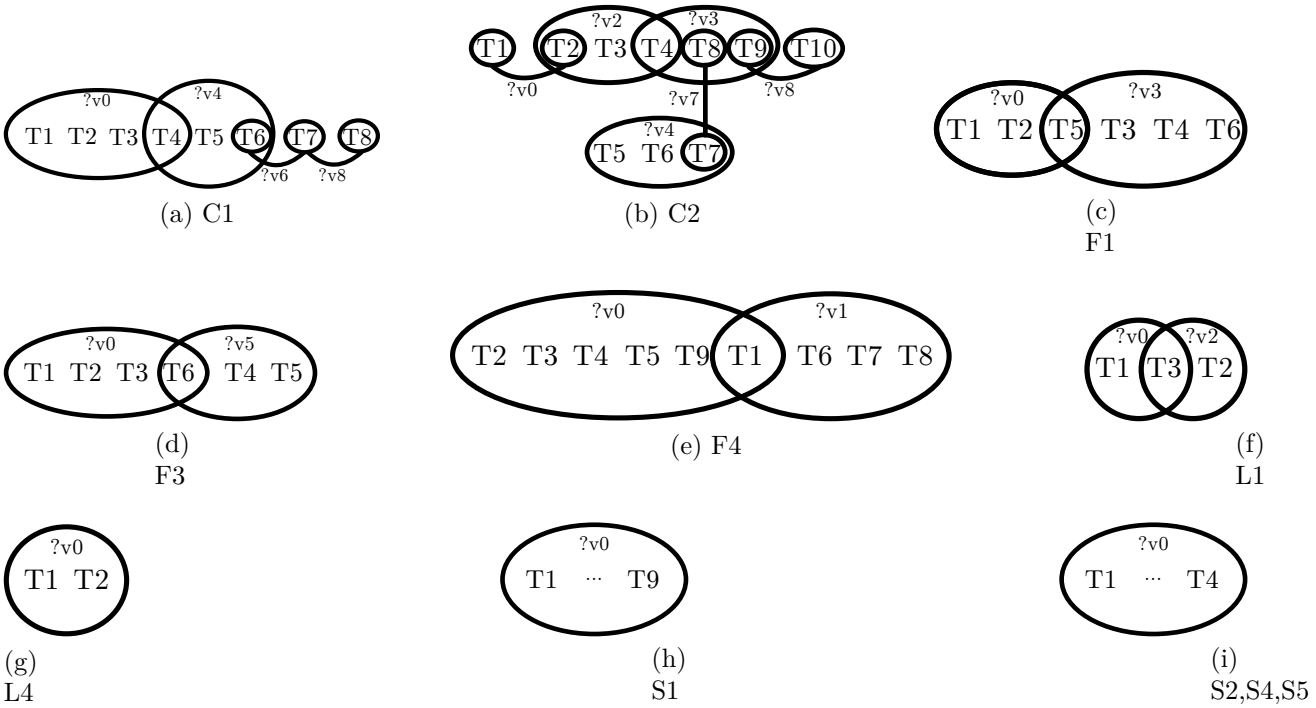

Figure 7.36: Triple pattern graph representation for each of the WatDiv queries. Multiple triple patterns sharing a variable are within the same circle, and its variable name is at the top of each circle. Edges between two triples are also representing that a variable is shared between them, where the corresponding variable is the label accompanying the edge.

In figure 7.36, we consider the triple pattern graph representation for each query. In the sub-figure 7.36.a, the triple patterns T1...T4 are sharing the variable ?v0, T4...T6 share ?v4, T6-T7 share ?v6, and T7-T8 share ?v8. This representation helps in understanding the complexity of each query. For example, the sub-figure 7.36.b, which has the graph representation for query C2, shows us quickly that the query C2 has more shared variables than any other, and it also has more triple patterns. This complexity of query C2 helps with explaining why it is the more expensive query as we measured in the figure 7.34.

The other subfigures from 7.36.c to 7.36.i show varying degrees of complexity for the remaining queries.

7.6. Analysis

Based on the results, we can see that in most queries there is no big performance penalty from using the k^2-tree cache. There is a small overhead that comes from network request messages that have to be passed between Jena and the cache.

The goal of this project is to improve query processing times, so we focus on a smaller set of queries that made the difference. We use figures 7.11 and 7.12 to understand the difference on a query-by-query basis, where it can be seen that there are a considerable number of cases where the cache wins significantly, but more in which Jena wins significantly. This goes against our expectations, since the Cache system uses only in-memory structures when replacement is turned off, as opposed to Jena, that has B+tree indices that are on disk.

When seeing what is going on under the hood, it is no longer that surprising that Jena could win many times. Jena uses memory mapped files for accessing the blocks in a B+tree, and its buffer for this purpose can be quite big. Memory mapped files are in big part managed by the operating system and have their own caching mechanism. We have to also take into consideration that B+trees store their results in big sequential segments that keep values by their key, which means that triple pattern results are all stored in a physically sequential portion of the B+tree and the values are uncompressed. Sequential access in disk is quite fast, where in some rare cases it should be able to outperform complex random accesses on memory, but more than that, due to the OS having readily available those big chunks of B+tree blocks cached in memory, it can beat the random access used by k^2-trees, because sequential access is the fastest kind on RAM due to improved hit rate on L1-L3 CPU caches.

Apart from the good access patterns that Jena can do, we saw in subsection 7.3.3 that there are some types of queries that are not well optimized for the cache usage, such as OPTIONALs that can make it harder on the cache results due to the communication mechanism between Jena and the cache that becomes expensive if it is overutilized. We successfully mitigated this problem in subsection 7.3.5 by allowing at most one BGP extraction from the cache per query, but Jena kept winning regardless of this optimization.

We also noticed there that for some queries the B+tree caching can occur within the processing of the same query, which means a massive optimization over the naive B+tree processing on disk and also outperforming our cache processing. The case in which the B+tree is better due to the caching of B+tree blocks is not very deterministic, in the sense that Jena has no much control over the caching mechanism operated by the OS. This nice performance of Jena comes as a big cost of high memory usage, in our experiments we needed to set its max heap size to 256 GB to avoid out of memory errors from the Java runtime. At the same time, for the Cache we could load the entire k^2-tree indexed data in about 32GB, which is not a requirement, since we can use less than that by using the cache replacement mechanism. During query evaluation, our cache system doesn't need to use much extra memory, because it can stream results partially before finishing the evaluation.

We saw in subsection 7.3.4 that the caching system with k^2-trees can improve query responses significantly for some types of queries. These queries are typically ones that would inevitably cause a high number of random disk access patterns when using only Jena and its local caching mechanism wouldn't help much in mitigating that cost.

As we noticed that some queries are still better in Jena, while some improve with the usage of the cache, we introduced a selective mechanism in subsection 7.3.6 that can start running both kinds against a query so a choice can be made early to select the best processing engine and then continue with the work with that. In that experiment we didn't see much improvement with the selection algorithm, but previous results showing that the cache can have a very good impact indicate that it was due to the conservative way of choosing an engine that mostly preferred Jena. So it should be possible to have better results by improving the selection mechanism.

For the insertions results our main concern was to not increase costs, and we achieved this according to the results in figure 7.31. The reason for which we don't aim to improve Jena insertion times is because in this system we still have to insert things into the B+tree, because we are making a caching system that can improve response times in some occassions, not a replacement for B+trees. Hence the cost of updates to the cache will always be additional to those of updating Jena's persistent storage.

In section 7.5, we ran experiments on a WatDiv dataset. Even if it is a synthetic data-

set, it helps in validating our previous results. This time the results were more consistent in favor of Jena's performance, but at the same time, the number of types of queries was very low in comparison to the types we used for Wikidata, which by nature were hard to categorize, since those were real user queries and added much more variation to the experimentation.

8. Conclusion

In this work, we studied the application of k^2-trees as a storage mechanism for caching of RDF triples in memory. The main outcome from it is that this can be beneficial for query performance, but it has the chance of being worse in some cases. This is explained by the possibility of data managed by B+trees being many times cached in memory and enabling more in-memory sequential access of data that is readily available, without requiring decompression. Also, the required B+tree data in disk is very commonly stored sequentially, which makes its access fairly cheap during disk reads.

There were cases that we identified that had poor performance with the usage of cache, which were explained by a few reasons. One of the reasons was about some queries causing a large number of unnecessary messages between Jena and the caching system, this was mitigated at the expense of using the cache BGP evaluation at most once per query. The second reason is about not enough optimizations that could benefit from L1-L3 CPU caches which Jena takes advantage of when data is loaded into memory. Also, Jena has more index types available to choose, while the cache has only one kind of index. If Jena selects the indices by optimizing for sequential disk accesses over random access, there is a good chance that it will be faster than some expensive patterns made within the cache, for which there is only one choice.

One of the main parts of this work was to be able to do updates on the caching system. The main concern was to support the updates without adding to the cost on the system noticeably. The goal was accomplished by applying some of the ideas that are commonly seen in modern database systems, such as a write-ahead log that was fine-tuned to our system and bulk operations that run periodically offline in the background. These ideas give reliability to the caching system, because the online updates to the in-memory structure are only temporary and serialization of the k^2-trees becomes infeasible for each update, as it is an expensive operation.

As in any software solution, in using this caching system we are subjected to some tradeoffs. First, there is the extra layer of indirection that needs data passing through the network. Even for a local network, there can be some small impact on response times. This is not a big cost to pay for having much better response times. Also, there is the extra management cost and extra complexity that can mean more bugs to solve. Apart from these, there is a cost on resources like memory, but we can control this by using a replacement of indices and reducing the memory footprint while paying for the disk accesses to retrieve the indices into memory, and while yielding fairly good results. We also saw that using Jena is very expensive in terms of memory usage, so if it is affordable to sacrifice performance in favor of less memory usage, a solution like the k^2-tree indices could be a good enough alternative.

8.1. Future work

8.1.1. Move some processing to the cache side

As we saw before, there are some types of queries that are less performant, in Jena because they are done in a mixed way between the Jena side and the cache side, and the messaging is done via network calls. For example OPTIONAL queries.

An optimization of our system would be to move more of the processing to the cache side. Using the OPTIONAL queries example, we could move a bigger part of the query to the cache and implement the OPTIONAL queries on the cache side.

This was found to be an issue late in this work and fixing it was deemed out-of-scope, as it is rather complicated to make the change, and should be done with more urgency on a production-level system.

8.1.2. Optimize queries, specialized to the cache

In this work, we used Jena's optimization for queries. The issue is that what might be good for Jena is not necessarily good for the cache or any other system. The cache uses a different approach to processing queries than Jena, so its optimization should be tuned differently. For simplicity, we chose to stick with Jena's optimization, but more than that we wanted to see that under the same conditions, it was going to be possible to have better results. In a real scenario what matters the most is the performance and having dedicated optimizations that work best for the cache would be an interesting topic for future work.

8.1.3. Optimize the selection of the processing engine

In subsection 7.3.6, we saw that the selection between Jena and the cache was slightly worse than just using Jena for everything, in the aggregated results. This indicates that the selection itself might not be making the best choice at all times, in that case, it almost always prefers selecting Jena first. This should be possible to fix by considering more data points before making a selection. The problem is that it gets more complicated as we select more data points, and possibly more expensive, because it would need more time to have both engines running at the same time.

8.1.4. Choosing B+tree when results are in memory

One big improvement over the strategy used in this work could be to identify if wanted results are cached in memory with B+trees, and in that case, select the B+tree processing over the k^2-tree one. This can be tricky to achieve, as we mentioned earlier, whether some blocks are in memory or not, is controlled by the operating system memory mapped files implementation. Maybe what can be done is something similar on the application side, but it can be complicated to achieve a similar performance.

8.1.5. Optimizations around concurrent requests

In this work, we only optimized for one request at a time. We implemented some support for concurrency with locks, but it is probably not enough if we wanted to move this to a

production-level. For example, it should be possible to provide locking support at sub-tree levels due to some encapsulation of data given by frontier nodes in the k^2-tree structure.

8.1.6. Optimizations to reduce in-memory random access in favor of in-memory sequential access

One of the issues of the caching system when compared to Jena is that Jena takes advantage of L1-L3 CPU cache accesses when data is loaded into memory, because B+trees store their data in big sequential blocks that get loaded into memory as physically sequential buffers. The k^2-trees also take advantage of CPU caches, due to blocks also being stored in sequential buffers, but they have two main drawbacks regarding this; one is that there are a lot of blocks, so even if a block is cached in CPU caches, there are a lot more to process and they are not necessarily near to each other in memory. The other drawback is that the values to be retrieved are not directly accessible, instead we need to compute them by traversing the blocks, and in some cases when having to travel through frontier nodes we lose the L1-L3 cache benefit.

To mitigate the issues with k^2-tree access we could simulate the B+trees behavior by adding another layer of caching with much less capacity that could store concrete values sequentially in-memory. This layer would be meant only for hot memory areas and could grant us the benefit of compact data easily accesible through k^2-trees and very fast sequential in-memory access patterns.

A disadvantage of doing this is that it makes the system much more complex and hard to manage, we would have to worry about more synchronization to be made.

8.1.7. Replacement strategy optimizations

As we mentioned in subsection 7.2.4, regarding the maximum index load times, there are some very few indices that can take 30s to 60s to load, probably even just one index in the Wikidata case. But this alone can have a big impact on performance if the LRU replacement ends up evicting and loading many times that index, so considering this, it might be needed to make some exceptions in the logic so these rare cases are handled differently and don't impact performance that much, for example to never load them or if loaded never evict them.

There can be other optimizations or other replacement strategies that we did not consider and could apply better to the k^2-tree cache scenario.

8.1.8. Resources dictionary

One point of improvement can be finding a better performant resource dictionary. That has a huge potential in terms of reducing response times for Jena. This is orthogonal to the work presented here, since nothing about that was modified.

8. Bibliography

[1] I. Robinson, J. Webber, and E. Eifrem, *Graph databases: new opportunities for connected data*. O'Reilly Media, Inc., 2015.

[2] T. J. Berners-Lee, "Information management: A proposal," tech. rep., 1989.

[3] W. W. W. Consortium *et al.*, "Sparql 1.1 overview," 2013.

[4] T. P. G. D. Group, "What is postgresql?." https://www.postgresql.org/docs/14/intro-whatis.html, 1996-2022.

[5] D. Brickley, R. Guha, and B. McBride, "RDF Schema 1.1." W3C Recommendation, Feb. 2014. https://www.w3.org/TR/rdf-schema/.

[6] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt, "Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph," in *The Semantic Web – ISWC 2018 – 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, Proceedings, Part II*, pp. 376–394, 2018.

[7] W. Ali, M. Saleem, B. Yao, A. Hogan, and A. N. Ngomo, "A survey of RDF stores & SPARQL engines for querying knowledge graphs," *VLDB J.*, vol. 31, no. 3, pp. 1–26, 2022.

[8] D. Battré, "Caching of intermediate results in dht-based rdf stores," *International Journal of Metadata, Semantics and Ontologies*, vol. 3, no. 1, pp. 84–93, 2008.

[9] M. Martin, J. Unbehauen, and S. Auer, "Improving the performance of semantic web applications with sparql query caching," in *The Semantic Web: Research and Applications: 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30–June 3, 2010, Proceedings, Part II 7*, pp. 304–318, Springer, 2010.

[10] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris, "Graph-aware, workload-adaptive sparql query caching," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1777–1792, 2015.

[11] J. Lorey and F. Naumann, "Caching and prefetching strategies for sparql queries," in *The Semantic Web: ESWC 2013 Satellite Events: ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers 10*, pp. 46–65, Springer, 2013.

[12] W. E. Zhang, Q. Z. Sheng, K. Taylor, and Y. Qin, "Identifying and caching hot triples for efficient rdf query processing," in *Database Systems for Advanced Applications: 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part II 20*, pp. 259–274, Springer, 2015.

[13] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[14] T. B. Lee, "Semantic web road map," Sept. 1998.

[15] R. W. Group, "Resource description framework." https://www.w3.org/RDF/, 2014.

[16] S. W. Group, "Resource description framework." https://www.w3.org/TR/sparql11-query/, 2014.

[17] J. Cardoso, "The semantic web vision: Where are we?," *IEEE Intelligent Systems*, vol. 22, no. 5, pp. 84–88, 2007.

[18] W3C, "RDF 1.1 Concepts and Abstract Syntax," Feb. 2014.

[19] M. J. Dürst and M. Suignard, "Internationalized Resource Identifiers (IRIs)." RFC 3987, Jan. 2005.

[20] A. Hogan, "Web of data," in *The Web of Data*, pp. 15–57, Springer, 2020.

[21] J. Salas and A. Hogan, "Semantics and canonicalisation of sparql 1.1," *Semantic Web*, vol. 13, no. 5, pp. 829–893, 2022.

[22] D. Vrandečić and M. Krötzsch, "Wikidata: A Free Collaborative Knowledgebase," *Commun. ACM*, vol. 57, p. 78–85, Sept. 2014.

[23] Wikimedia.org, "Wikidata edits histogram." https://stats.wikimedia.org/#/wikidata.org/contributing/edits/normal|table|2020-07-01~2022-09-01|~total|monthly, Sept. 2022.

[24] D. Arroyuelo, G. de Bernardo, T. Gagie, and G. Navarro, *Faster Dynamic Compressed d-ary Relations*, pp. 419–433. 10 2019.

[25] A. J. Contributors, "Fork of Apache Jena 4.3.0." https://github.com/CristobalM/jena.

[26] C. Miranda, "RDFEWK2C - Java adapter server for Jena and Cache communication." https://github.com/CristobalM/RDFEWK2C.

[27] C. Miranda, "RDFCacheK2 - Dynamic and compact k^2-tree caching system." https://github.com/CristobalM/RDFCacheK2.

[28] C. Miranda, "c-k2tree-dyn - Implementation of dynamic and compact k^2-tree." https://github.com/CristobalM/c-k2tree-dyn.

[29] M. T. O. G. Aluç, O. Hartig and K. Daudjee, "Diversified Stress Testing of RDF Data Management Systems.." In Proc. The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, 2014, pages 197-212. WatDiv available from http://dsg.uwaterloo.ca/watdiv/.

[30] I. C. F. C. Logic, "Wikidata sparql logs." https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en, July 2022.