UNIVERSITÄT ZU LÜBECK

From the Institute of Information Systems
of the University of Lübeck
Director: Univ.-Prof. Dr. rer. nat. habil. Ralf Möller

Parallel Execution Approaches on Data and Index Structures in the Context of Semantic Web Database Management Systems

Dissertation for Fulfillment of Requirements for the Doctoral Degree
of the University of Lübeck
— from the Department of Computer Sciences —
Submitted by Dennis Heinrich from Bad Segeberg.

Lübeck, 2018

First referee:	PD Dr. rer. nat. habil. Sven Groppe
Second referee:	Prof. Dr. rer. nat. habil. Josef Ingenerf
Date of oral examination:	September 19^{th}, 2018

Approved for printing. Lübeck, September 24^{th}, 2018

Abstract

Information is crucial in the development of mankind. Since the dawn of time, humanity has preserved knowledge in different media, like stone tablets, papyrus scrolls or books. Each media has different advantages and disadvantages, like stone tablets do not burn, papyrus scrolls are light-weight and books combine multiple pages in an ergonomic form for the human hand. If these different types of media are stored in a collection, they are sorted and cataloged in a way that a reader can find the needed information faster and easier. These choices of how and where we store information persists if a computer is used to store data. The choice of the underlying data structure and the used index structure both directly impact how efficient and effective the data can be stored and retrieved. This efficiency and effectiveness becomes more important with advanced concepts of structuring data, like the *Semantic Web* that stores sentence-like statements, consisting of subject, predicate and object, in the billions. For a long period of time, the advancements in the frequency of processing units could counter the growth of the amount of data that is processed. At some point the consumed power for greater processing speed became unbearable and the distribution of tasks into different units of a multi-core processor was the new direction the development was heading. Specialized hardware with parallel execution paths have become the new assistants and/or competitors to the parallel approaches of processors. Be it either the reprogramming of Graphics Processing Unit for general tasks than graphic computation or the use of Field-Programmable Gate Arrays (FPGAs) for a user-specific circuit, the chosen index and data structures have to be adapted to the new parallel execution principle. This leads to a new examination of already existing data and index structures which are advanced and combined under the aspect of parallel execution.

In this work, the index and data structures of a Semantic Web Database Management System are extended and restructured for a highly parallel execution. The new sorting approach PatTrieSort uses patricia tries to not only sort but also compress the input strings at the same time allowing an efficient usage of the main memory. Multiple tries, so called initial runs, can be merged later in the process resulting in one final patricia trie. This new sorting approach can be integrated into the dictionary and six collation orders generation of the chosen Semantic Web Database LUPOSDATE. In the multiple steps of this generation process, each opportunity is used to execute tasks in parallel. At the last part of this work, the B^+-tree index structures of the collation orders are extended for parallel searches on an FPGA. This leads to a hybrid system consisting of a CPU-based host system and an FPGA-based accelerator card. Each part of this work is carefully evaluated

for its benefits and limitations in context of parallel execution and acceleration of the Semantic Web system.

Kurzfassung

Information ist entscheidend für die Entwicklung der Menschheit. Seit Anbeginn der Zeit hat die Menschheit Wissen in verschiedenen Medien wie Steintafeln, Papyrusrollen oder Büchern bewahrt. Jedes Medium hat verschiedene Vor- und Nachteile, wie zum Beispiel, dass Steintafeln nicht brennen, Papyrusrollen leicht sind und Bücher kombinieren mehrere Seiten in einer ergonomischen Form für die menschliche Hand. Wenn diese verschiedenen Arten von Medien in einer Sammlung zusammengefasst sind, werden sie so sortiert und katalogisiert, dass ein Leser die benötigten Informationen schneller und einfacher finden kann. Diese Möglichkeiten, wie und wo wir Informationen speichern, bleiben bestehen, wenn ein Computer zum Speichern von Daten verwendet wird. Die Wahl der zugrundeliegenden Datenstruktur und die verwendete Indexstruktur wirken sich direkt darauf aus, wie effizient und effektiv die Daten gespeichert und abgerufen werden können. Diese Effizienz und Effektivität wird mit fortgeschrittenen Konzepten der Strukturierung von Daten wichtiger, wie das *Semantic Web*, welches satzartige Aussagen, bestehend aus Subjekt, Prädikat und Objekt, in Milliardenhöhe speichert. Die Fortschritte bei der Taktrate von Prozessoren konnten dem Wachstum der Menge der verarbeiteten Daten über einen langen Zeitraum entgegenwirken. Irgendwann wurde der Stromverbrauch für eine höhere Verarbeitungsgeschwindigkeit allerdings untragbar und die Verteilung der Aufgaben auf verschiedene Einheiten eines Multi-Core-Prozessors war die neue Richtung, in die sich die Entwicklung bewegte. Spezialisierte Hardware mit parallelen Ausführungspfaden sind die neuen Assistenten und/oder Konkurrenten der parallelen Ansätze von Prozessoren geworden. Sei es die Neuprogrammierung der Graphics Processing Unit für allgemeine Aufgaben anstatt nur grafische Berechnungen durchzuführen oder die Verwendung von Field-Programmable Gate Arrays (FPGAs) für eine anwenderspezifische Schaltung, die gewählten Index- und Datenstrukturen müssen an das neue parallele Ausführungsprinzip angepasst werden. Dies führt zu einer neuen Untersuchung bereits existierender Daten und Indexstrukturen, die unter dem Aspekt der parallelen Ausführung weiterentwickelt und kombiniert werden.

In dieser Arbeit werden die Index- und Datenstrukturen eines Semantic Web Datenbankmanagementsystems für eine hochparallele Ausführung erweitert und restrukturiert. Der neue Sortieransatz PatTrieSort verwendet Patricia Tries, die Zeichenketten nicht nur sortieren, sondern auch komprimieren, was eine effiziente Nutzung des Hauptspeichers ermöglicht. Mehrere Tries, sogenannte initial runs, können später in dem Prozess zusammengeführt werden, was zu einem einzigen PatriciaTrie führt. Dieser neue Sortieransatz kann in die Wörterbücher- and Indexerstellung der ausgewählten Semantic Web Datenbank LUPOSDATE integriert werden. In den mehreren Schritten dieses Generierungsprozesses wird jede Möglichkeit dazu

verwendet, Aufgaben parallel auszuführen. Im letzten Teil dieser Arbeit werden die B^+ -Baum Indexstrukturen für parallele Suchen auf einem FPGA erweitert. Dies führt zu einem Hybridsystem, das aus einem CPU-basierten Hostsystem und einer FPGA-basierten Beschleunigerkarte besteht. Jeder Teil dieser Arbeit wird sorgfältig auf seine Vorteile und Einschränkungen im Zusammenhang mit der parallelen Ausführung und Beschleunigung des Semantic Web-Systems untersucht.

Contents

1 Introduction

In this chapter a motivation will be given why the treatment of indicies is a current research field even though there has been a long period of time of research in this field and which kind of new solutions still can be found. After this, the scientific contribution of this work is highlighted and what has been achieved. At the end of this chapter the organization of the following chapters is explained.

1.1 Motivation

Collecting, storing and searching for data is just common for human beings. From making own experiences and memories stored inside our brain to sharing them in pictures, sounds and words using language, books or other media, humanity made a huge profit of this behavior. Not stepping back and building our knowledge on the knowledge of others is a successful way of making progress through generations. Reaching the Information Age makes it clear that in current times knowledge is an important resource. Today computers aid people to collect and find the information they desire. This is a big challenge because the amount of data collected grows steadily. This growth has not only caused by humans, sensors and processors embedded in everyday things collect and process data, like in the Internet of Things (IOT). However, the most important factor contributing to the growth is the World Wide Web (WWW). Only after a few decades the knowledge inside the Web seems to grow limitless. This growth is reinforced by the development of the Web 2.0 where the average person with an Internet connection can participate in the Web. This leads to the need of bigger and faster databases to handle the continuous flow of information in the Internet. Still the aid of computers to find the important data for their users is limited because most data is lacking some sort of context to help the computer understand the connections between different information. At this point the idea of the Semantic Web arouse. Enhancing data with context sensitive annotations made it possible to process data in a way that machines could evaluate if certain information belong together or contradict each other. With more complex data structures the need for memory and processing power grows further. The use of single-core processors was limited because of its exponential growth of consumed

power compared to its clock frequency. Thus, it became clear that tasks needed to be performed parallel to make use of the new multi-core systems. Following this path of parallelism there is a need for new approaches on more parallel data structures inside databases. These approaches do not need to be limited to multi-core processors only, but can use specialized hardware to take a step further into parallelism.

1.2 Scientific Contributions of this Work

The author of this work published several contributions in the five main research fields of Cloud Computing, Query Processing, Semantic Web, Hardware Acceleration and Index Structures [1–15]. Figure 1.1 shows these contributions in context to the research fields they are the most contributing to.

Since handling all research fields would inflate this work, it only investigates the influence of different index structures for Semantic Web (SW) database systems and therefore focuses on the fields of Semantic Web, Hardware Acceleration and Index Structures [1–5].

There are further publications of the author that address the field of Cloud Computing [6–8] and Query Processing [9–13]. Further patricia tries are handled without the index construction focus in [14, 15].

The key features of this work are the following:

In Chapter 3 a new external sorting algorithm *PatTrieSort* based on patricia tries is introduced. The main contributions of this chapter include:

- a new sorting approach *PatTrieSort* as variant of external merge sort for sorting strings, where the initial runs are generated by using patricia tries. The initial runs are swapped in form of patricia tries to disk and are merged in a later processing step.
- a new algorithm for merging patricia tries used within the merge phase of the new sorting approach.
- a complexity analysis for the new merging algorithm in terms of runtime, I/O costs and memory consumption.
- a comprehensive performance evaluation and analysis of PatTrieSort compared to the other external merge sort variants using large-scale datasets with over 1 billion strings.

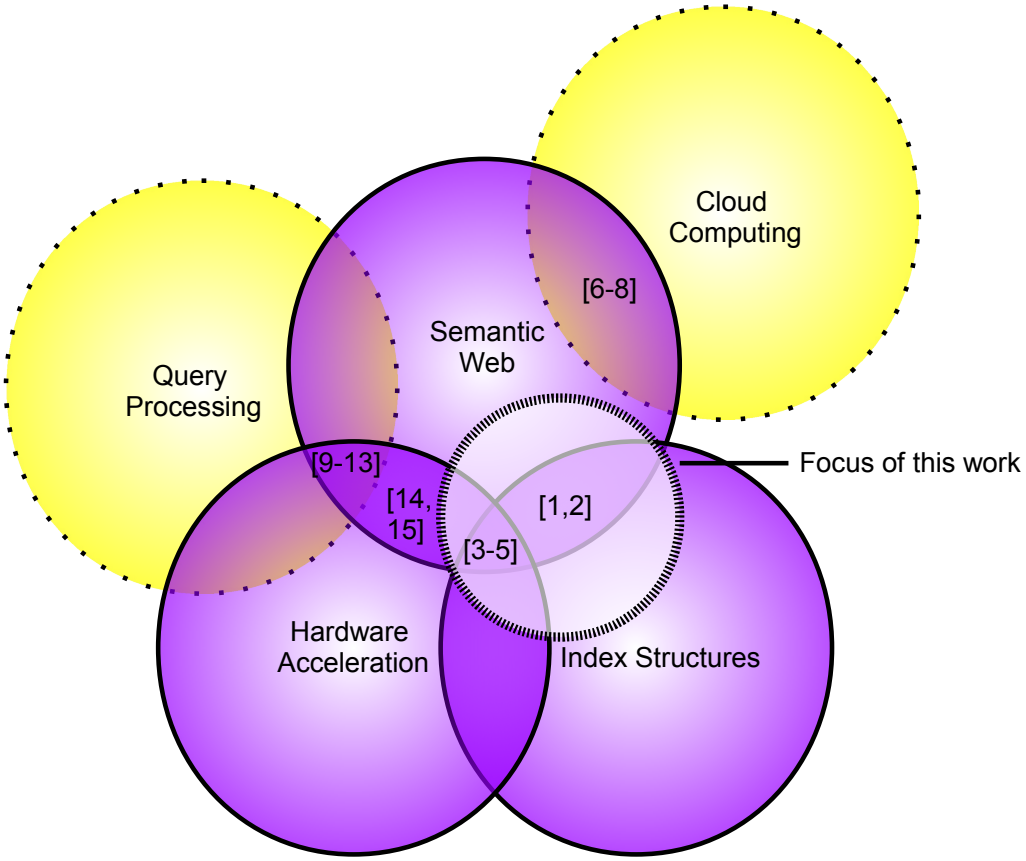


Figure 1.1: The main five research fields where the author published contributions. For this work three fields are highlighted and the focus is set to five main contributions [1–5].

These contributions are published in [1].

In Chapter 4 a very fast algorithm to construct the dictionary of a Semantic Web (SW) Database Management System (DBMS) is introduced, which smoothly incorporates the construction of the 6 evaluation indices into the generation of the dictionary. The main contributions of this chapter include:

- a new index construction approach for SW data smoothly incorporating dictionary construction and evaluation indices construction by taking advantage of parallelism capabilities offered by today's multi-core CPUs, and

- a comprehensive performance evaluation and analysis of our proposed index construction approach using large-scale real-world datasets with over 1 billion triples.

These contributions are published in [2].

In Chapter 5 a hybrid index structure for a SW DBMS (see Figure 1.2) running on a CPU-based host system and a Field-Programmable Gate Array (FPGA) is another contribution.

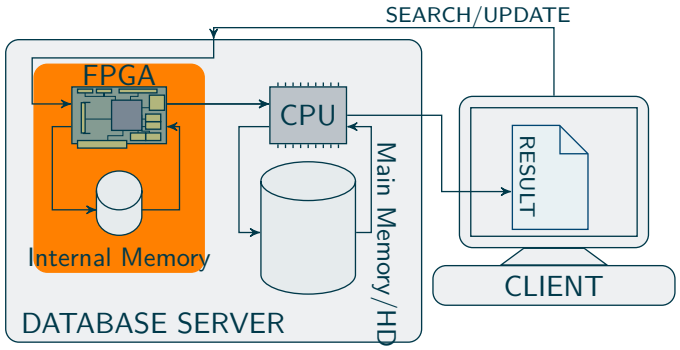


Figure 1.2: Proposed database system using an FPGA as an accelerator.

The main contributions of this chapter include:

- a parallel search that can be performed inside the B^+-tree nodes of the FPGA resulting in shorter search times
- a fully functional SW DBMS by utilizing the Peripheral Component Interconnect Express (PCIe) connection for the communication between CPU and FPGA
- an estimation for a scheduler handling the setup of the system for update heavy scenarios

These contributions are published in [3–5].

1.3 Organization of this Work

In Chapter 2 the fundamentals and backgrounds for the subsequent chapters are provided. These fundamentals are divided into three topics which provide the

needed knowledge in the fields of the SW, Reconfigurable Computing (RC) and index structures.

The Chapters 3, 4 and 5 contain the main contributions of this work as outlined in the former section. Each of these chapters starts with a section (3.1, 4.1 and 5.1) preparing the reader for the topic of the chapter. Each of these sections has a subsection (3.1.1, 4.1.1 and 5.1.1) connecting the introduced fundamentals with the current needed knowledge of the chapter, therefore the reader can refresh his knowledge if needed before continuing to the main part of the chapter. Further, scientific contributions, that are related to the research field handled in the current chapter, are presented in the next subsection (3.1.2, 4.1.2 and 5.1.2). This gives the reader the opportunity to broaden his or her knowledge about the topic with external works if interested. After this introduction the main concepts of the chapter are presented in Section 3.2, 4.2 and 5.2. Thereafter, in Section 3.3, 4.3 and 5.3 the experimental setup and results are presented and evaluated. Each chapter finishes with a summary and conclusion (see Section 3.4, 4.4 and 5.5).

The last Chapter 6 sums up the contributions of the former chapters and gives an outlook on possible future steps.

The back matter provides a list of acronyms used in this work, references and a list of used figures, listings and tables. Further, the Curriculum Vitae of the author and his personal publications are presented.

2 Fundamentals

In this chapter many basic procedures, that will reappear in later chapters, are explained in detail. Therefore certain techniques or algorithms will only refresh main points and refer to this chapter for further knowledge. There are three wide fields that are treated in this work.

Namely the Semantic Web with its context sensitive information for workable interpretation by machines.

The reconfigurable computing that allows specialized hardware for custom applications without hardwired single-use solutions.

The last part are the title giving indices that are explained from the very basic ideas to different, complexer structure types.

2.1 Semantic Web

> *This section introduces the concept of the Semantic Web (SW) which is important for the Chapters 3, 4 and 5.*

In this section the Semantic Web (SW) is introduced. After giving a motivation and an overview over the technical details, especially about the data representation and query execution of the SW, the SW database LUPOSDATE is presented, which is the commonly used DBMS in this work.

2.1.1 Introduction

> *This section introduces the basic idea behind the SW and is important to understand decisions for the Layer Cake in Section 2.1.2.*

With the world wide connection of computers within a network, known as the internet, information could be spread globally in a short amount of time. Still the World Wide Web (WWW) [16] had to be developed to allow more common users than technical experts to use these systems since the ability to share information

was partly bound to certain routines and commands of technological aspects. Over time the hurdle was lowered so far that even children can post and share information in the WWW.

But with this easy to use access to the WWW comes the problem to reacquire knowledge out of the given information. For a human it is easy to follow certain links in the WWW to find an answer to a certain question. Even though the links only connect one web resource with another, for example a website with a video, a person can interpret the semantic connection between the two resources. This is possible because the creator of the resources and the person that is searching for something, share a certain amount of knowledge to the connection and background of the resources acquired in daily life and behavior.

As an example a person is looking for a picture of the 'Queen'. The first step to find such data is using a search engine since a manual search within the amount of possible web resources exceeds the capabilities of an individual in the form of patience and concentration. The search engine has to decide whether the keyword 'queen' belongs to the British rock band 'Queen' or the current queen of the united kingdom and other countries Elizabeth II. Since a clear distinction is impossible with only one word, the search engine delivers pictures of both. An result of such a search can be seen in Figure 2.1 and even though in this case `google.com` was used, a similar result can be seen in other search engines like `duckduckgo.com`, `yahoo.com` or `bing.com`.

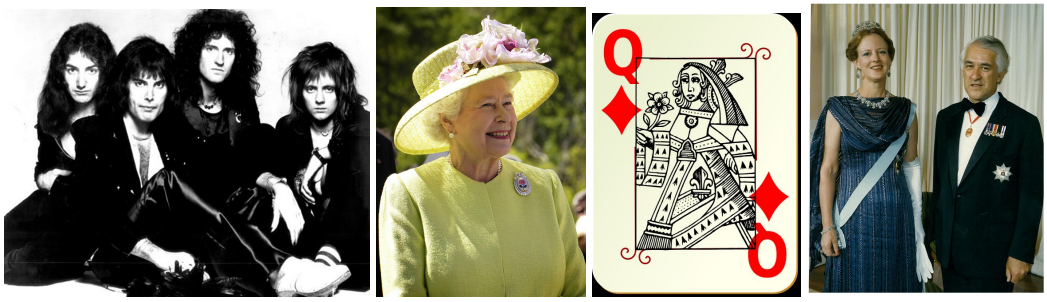


Figure 2.1: A search on google for a picture with the keyword 'Queen' (sources from left to right: [17–20])

These results show the efforts of search engines to deliver relevant data. On the one hand all the other queens (like Margrethe II of Denmark) that exist or existed are filtered out, even though it is not clear that Elizabeth II was searched, but because of her popularity and corresponding searches she seems to be a good match. On the other hand the music band 'Queen' is also very popular in such a way that they are competing with Elizabeth II in the resulting ranking. With the addition

of 'the' to the word 'Queen' the results only show pictures of Elizabeth II (see Figure 2.2).

Figure 2.2: A search on google for a picture with the keyword 'the Queen' (sources from left to right: [18, 21–23])

Still, the phrase 'the Queen' never mentions Elizabeth II, but all results picture her. This means, there is a direct connection between Elizabeth II and calling her 'the Queen' without directly mentioning it. For the creator of the data and the person that is searching this connection is known to some extent while for a machine this connection is not directly processable. Simple ways to make this connection visible to machines are the accompanying text with the words 'Elizabeth II' and 'the Queen' or direct tags as meta-data on the picture itself. These methods still are very limited in the sense that humans can directly identify the person on the picture or have further information what it means to be a queen.

This is one of the things the Semantic Web (SW) wants to correct and extend. Adding a direct reference from the picture to meta-data of Elizabeth II will help to identify this file as a picture of a person, a woman, a monarch and many other useful information which a human can abstract from the picture and is based on his own knowledge, but also can be accessible to machines. In order to achieve this, the SW needs a standardized meta-data representation which makes machine processing possible. Still, a whole new system would make the WWW and the SW direct competitors which would not be a good solution. Instead the SW is meant more like an extension to the WWW than a whole new system.

In the next parts of this section the different basic technologies of the SW are introduced and explained to give a better view over the features of the SW.

2.1.2 Layer Cake of the Semantic Web

This section introduces the SW Layer Cake and is the framework for the data representation in Section 2.1.3 and the query execution in Section 2.1.4.

As the Semantic Web (SW) is intended as an extension of the World Wide Web (WWW) it is not easy for the end-user to recognize if he or she is using SW technologies or not. The basic idea described by Tim Berners-Lee [24] is to enable machines to process a web of data in a more context aware way to interpret not only the data but also the connection to other data and resulting implications. This means that the processing unit of a search can produce more accurate and context-sensitive results for the user, but the user can only evaluate the results and not the way these were produced. Still, the user has an advantage if the used web application uses SW technologies although he or she does not know. In order to achieve the unified use of these technologies, the World Wide Web Consortium (W3C) promotes the standards for data formats and exchange protocols in the WWW. The W3C constantly updates standards and is working on different aspects of the SW which means that the technologies have not reached their final state. In Figure 2.3 the layer system of the SW is represented which is also often called 'Layer Cake'.

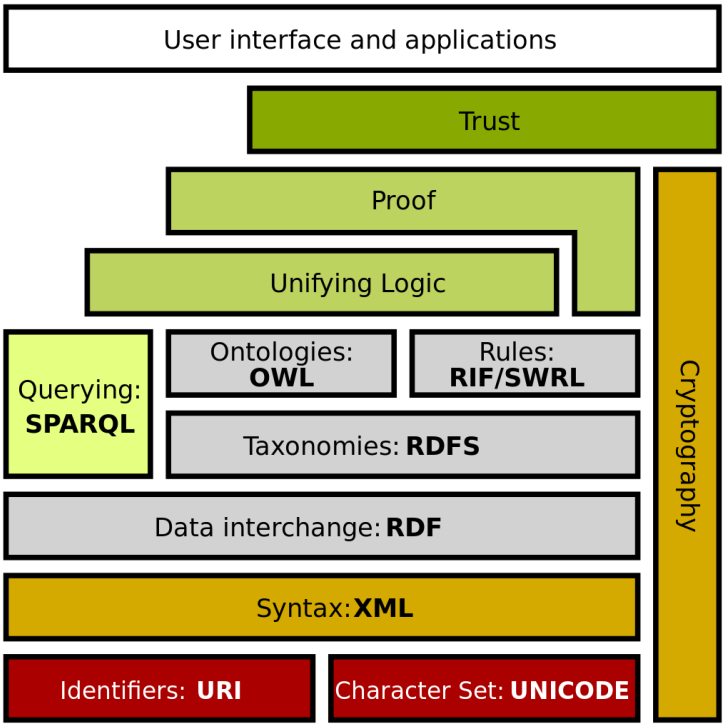


Figure 2.3: The SW Layer Cake from [25]

Starting from the bottom and going to the top the layer system describes well-defined standards up to proposal that are in a working progress. The starting

point is the data that needs to be stored. Since the SW uses string representations to describe resources and their connections to each other the bottom layer ensures the comparability of data from different sources. The Universal Character Set (UCS) guarantees that the encoding of different language sets and special characters is standardized and usable for each application. With the UCS the resources can be described by Internationalized Resource Identifier (IRI) which basically are strings of characters to identify unique resources, like the Uniform Resource Identifier (URI). The Extensible Markup Language (XML) layer ensures that the documents that describe the resources are human- and machine-readable. The final step to ensure a unified data format is the Resource Description Framework (RDF) layer. Statements and connections between different resources can be described in this layer as triples which are explained in more detail at a later point in this section. With a coherent view on the data there are three different layers that provide different functionalities. The SPARQL Protocol And RDF Query Language (SPARQL) is a query languages that allows different operations like searches on RDF data sets. In order to provide taxonomies and ontologies the middle section consists of the RDF Schema (RDFS) and the Web Ontology Language (OWL) layers. The Rule Interchange Format (RIF) layer allows the exchange of different, defined rule sets. All the so far mentioned layers will be described further in this section, while the rest of the layers will only be described at this point since these layers are not finalized and their basic ideas are given. The unifying logic layer uses the given data and rules to produce inferences that extend the data set with new data that is only indirectly given. Further the proof layer tries to prove certain circumstances that are described by facts in the data set. Of course finding a proof is a much harder task than verifying that a proof is correct, which explains why this layer is more an idea than finalized. In the trust layer the used data sets are evaluated by their sources and if these sources can be trusted. This layer tries to prevent the usage of intentionally corrupted data or allows the user to trust or distrust a certain source. Another layer that spans over most of the layer cake addresses the security aspect of the SW technologies and handle the cryptography between the layers. The last layer represents only the User Interface (UI) and applications which are not standardized but give an idea that a SW application should integrate all layers below (or at least the layers with a standard).

2.1.3 Data Representation

This section introduces how data in the SW is described, which influences the query handling (see Section 2.1.4), especially the practical handling in a SW

DBMS, like LUPOSDATE (see Section 2.1.5). Further this data is handled as keys and/or values in the introduced index structures in Section 2.3.

The concept of the SW is to describe any possible resources in the most accurate way possible. This makes a highly flexible data representation model necessary. The first three layers of the introduced SW Layer Cake allow such a flexible model. This section starts with the encoding of the data in Section 2.1.3.1 and the identifier of a resource in Section 2.1.3.2. Further, Section 2.1.3.3 introduces the syntactical description of the data. At last, the common way to describe a resource in the SW is shown in Section 2.1.3.4.

2.1.3.1 Universal Character Set (UCS)

The standard format for statements in the SW is the RDF format which will be introduced in Section 2.1.3.4. Its string representation is introduced in this section which will be important for the resource identifiers in Section 2.1.3.2 and the used markup language in Section 2.1.3.3

The UCS is a standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. The long term goal is a collection of all used meaningful symbols or letters in any text form. With such a universal encoding other local or regional encodings can be replaced, preventing these encodings from overlapping each other and producing incorrect translation into different letter. The first version was released in October 1991 supporting 24 scripts and 7,161 characters [26]. These scripts are collections of letters or other signs that belong to one or multiple languages. For example the Latin script covers next to Latin itself also languages like English, German, French and other languages that mostly use Latin letters. The current version 10.0.0 of the UCS [27] supports 136,690 characters and a total of 139 scripts [28]. It is divided in 17 different planes. From these planes 4 planes are currently used. The first plane (0) is called Basic Multilingual Plane (BMP) and is used for characters of almost all modern languages, and a large number of symbols. The structure of BMP consist of 2^{16} code points which therefore can support up to 65,536 characters. In Figure 2.4 the structure of the BMP is shown and the distribution of the different language sets.

The most part of the BMP consists of Chinese, Japanese and Korean (CJK) characters. The second plane (1) is the Supplementary Multilingual Plane (SMP) that consist of characters of old, mostly unused languages. As an example the Egyptian hieroglyphs are encoded in this plane. The third plane (2) is the Supplementary Ideographic Plane (SIP) which contains CJK ideographs, that were not included in

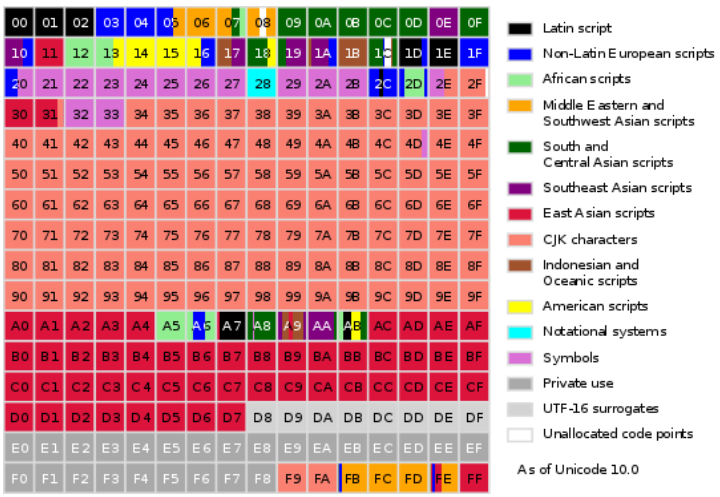


Figure 2.4: The first plane (plane 0 or the BMP) of the Unicode from [29]

earlier character encoding standards. After these three planes the rest of the planes mostly provide space for further encodings and are therefore unassigned (planes 3 to 13). The plane 14 is the Supplementary Special-purpose Plane (SSP) and contains non-graphical characters, like an alternate glyph for a character that cannot be determined by context. The last two planes are the Private Use Area (PUA) planes named PUA-A and PUA-B which allow parties outside the International Organization for Standardization (ISO) [30] and the Unicode Consortium their own character assignments. This comes with the price that such characters have a limited interoperability and support by other parties.

2.1.3.2 Internationalized Resource Identifier (IRI)

> *The statements in the SW are about resources. This section introduces how to give these resources identifiers which are used in the RDF (see Section 2.1.3.4) but also in queries (see Section 2.1.4).*

The IRI [31] can identify resources with a unique chain of characters. Its origin comes from the Uniform Resource Identifier (URI) which is widely used in the World Wide Web (WWW) in form of Uniform Resource Locators (URLs) which are often called 'web addresses'. A URI has a certain syntax and the generic form can be seen in Listing 2.1

Listing 2.1: The generic form of a Uniform Resource Identifier (URI)

```
1 scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

A URI is divided in multiple parts:

- The **scheme** is mostly a combination of lowercase letters ending with a colon. Famous schemes in the web are protocols like the Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP) which are represented as 'http:' and 'ftp:'. Although the schemes can be registered at the Internet Assigned Numbers Authority (IANA) using custom unregistered schemes is no problem.
- The double slashes (//) can be necessary or obsolete depending on the chosen scheme.
- The **authority** part contains three parts:
  - Optional **user**name and **password** separated by a colon and ending with @.
  - A **host** is either an Internet Protocol (IP) address or a registered hostname.
  - An optional **port** number, separated from the hostname by a colon.
- A **path** has an hierarchical order from left to right, separated by a slash for each level. This representation comes close to a file system path but does not need to be one.
- A **query** is optional and introduced with a question mark (?). Depending on the scheme this part can look pretty diverse, an example would be key-value pairs where the key identifies the resource that should be searched and the value defines what is searched for.
- The last part is the optional **fragment**. With this fragment identifier a certain subsection of the resource can be identified and set in focus. This is often used for URLs to force the browser to scroll to a certain location of a website.

The difference between an IRI and a URI is the used encoding. For example the following string is an IRI:

https://en.wiktionary.org/wiki/パン

This IRI links to the wiktionary web page of the Japanese word パン which translates to 'bread'. With the use of percent-encoding the IRI can be translated into a URI resulting into the following string:

https://en.wiktionary.org/wiki/%E3%83%91%E3%83%B3

In this case the two Katakana characters are each represented as three bytes indicated by the % character as an escape sign. This processes can also be reverted, which makes a conversion from IRIs to URIs and back possible. This is a better method than converting Katakana characters directly into Latin characters since the translation of パ is 'pa' and for ン it is 'n'. This would result into the following string:

https://en.wiktionary.org/wiki/pan

Of course this URI would refer to a cooking device and not to the type of food original intended.

2.1.3.3 Extensible Markup Language (XML)

> *Statements about resources in the SW are divided into three parts. This section show how to arrange these statements, embedding the ground work of the Section 2.1.3.1 and the Section 2.1.3.2, as a last step before the RDF can be introduced in Section 2.1.3.4.*

With the possibility of the correct encoding in the UCS and the string representation of resources by IRIs the next step is the syntactical arrangement of the data. In order to support human- and machine-readability the Extensible Markup Language (XML) is used for this purpose. The name of XML already implies that it is a markup language which are typically designed to distinguish keywords or annotations from the original text. An example of an XML file can be seen in Listing 2.2.

Listing 2.2: A small XML example describing an e-mail.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE email_list SYSTEM "email.dtd">
3  <mail>
4    <from email="heinrich@ifis.uni-luebeck.de">Dennis</from>
5    <to email="werner@ifis.uni-luebeck.de">Stefan</to>
6    <heading>Just a short reminder</heading>
7    <body>I wanted to remind you that tomorrow it will  rain.</body>
8    <attachment/>
9  </mail>
10 <!-- This is a comment at the end of the document -->
```

This XML file describes a short Electronic Mail (e-mail) from one person to another. The file starts with the XML declaration which gives information about the document, like in this case the XML version and the character encoding. Every meta information is encapsulated between chevrons ('<', '>') and is called a 'tag'. For the declaration extra exclamation marks ('?') are added at the beginning and the end of the tag. The second line consists of a 'start-tag' which marks the beginning of an 'element' of the same name. In this case the 'book' element starts at line 3 and ends on line 9 with an 'end-tag' which shares the same name with the start-tag but also has an additional slash (/) at the beginning of the tag. With this addition the start- and end-tags are easily recognizable. Inside the two tags from the 'book' element there are 5 additional elements (lines 4 to 8). These elements are called 'child' elements of the 'book' element since they are on a lower level in the documents hierarchy. The indented text and the new lines for each element only support the understanding and readability of the human observer. For a machine all text could be written in one line and would still represent the current structure correctly. The elements 'from', 'to', 'heading' and 'body' all have a number of characters between their tags (line 4 to 7). These strings are called 'content' and are often the actual representation of the element. For example the 'body' element contains a message that should be displayed to the user in a specific window of an e-mail client. The 'heading' element is also a message but is mostly just a few words introducing the topic of the actual message, here the 'body' element. The e-mail client can easily distinguish the two elements from each other and present their content in a suitable way. Beneath the content of an element there are also 'attributes'. In this example the elements 'from' and 'to' have both the attribute 'email' inside the start-tag (line 4 and 5). Although both elements share the same name 'email' as an attribute, these are not necessary the same and can be distinguished by the element they belong to. This also means that one element can only have one attribute with a specific name but multiple with different names divided by a comma (','). Additional to their names, all attributes have a value indicated by an equals sign ('=') and surrounded by quotation marks ('"'). Attributes often represent data that are unique to a certain element and give context to an element (but do not need to). The e-mail client for example could show the content of the elements to the user since humans normally use the name of a person (in this case 'Dennis' and 'Stefan') as an identification and not the e-mail address. But for the client it is important to know to which address it should send a message and can not identify a recipient by the content of an element. Of course the attributes 'email' could also be 'email' elements with the attributes name as the tag of the element and the attributes value as the elements content. But this would make the two attributes that are distinguishable by their elements to the same type of element since they share the same tag 'email'. This is also true no matter on which level of the document hierarchy the elements appear. Further the uniqueness is not

guaranteed by default since an element can have multiple child elements with the same tag. Still it is possible to make an element unique by using a set of markup declarations that define a document type. A Document Type Definition (DTD) can describe a certain structure for an XML document either directly inside this document or with a reference to an external `.dtd`-file. In the example, the line 2 shows a link to an external `.dtd`-file with the name 'email.dtd' that describes the document type 'email_list'. The tag starts with an exclamation mark ('!') to indicate that this is a special element like the XML declaration in the first line. While DTD is the oldest schema language for XML, there are newer alternatives like XML Schema Definition (XSD). With such a definition the appearance of a certain element can be mandatory but sometimes does not contain any data. While other such elements are simply left out, mandatory empty elements can be presented by an 'empty-element' tag. The 'attachment' element on line 8 is an example for an empty-element tag which consist of only one tag with an slash ('/') before the ending chevron ('>').

XML 1.0 (Fifth Edition) and XML 1.1 support the direct use of almost any Unicode character in any part of an XML document with a few exceptions, like characters with special meaning for example the chevrons ('<','>'). This means an element like the following is possible:

<word translation="bread">パン</word>

2.1.3.4 Resource Description Framework (RDF)

After the introductions of needed concepts in the Sections 2.1.3.1, 2.1.3.2 and 2.1.3.3 statements about resources in RDF are possible. These statements enable the execution of queries on them (see Section 2.1.4), are handled by SW DBMS (see Section 2.1.5) and indexed by index structures introduced in Section 2.3.

Giving a statement about a resource with the RDF [32] is the next step in the SW layer cake after using XML for Syntax and IRIs as identifiers for these resources. A statement to a resource in the Web is always an RDF triple consisting of a subject (s), predicate (p) and object (o) resembling a simple sentence that describes the subject. The formal definition of a triple is $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ where I represents the IRIs, B are blank nodes and L are literals. Since IRIs were already described in Section 2.1.3.2 only blank nodes and literals are examined further.

Blank Nodes do not have an IRI and can therefore not be identified globally (meaning over multiple RDF documents). Still it is possible to give blank nodes

local identifiers for the current document. Because of this, it is possible to describe something that is so far not identified or does not need to be identified. An example would be a movie that is normally identified by the title, but right at the beginning there are only ideas for the screenplay. Making the title of the movie a blank node gives the opportunity to name the genre (comedy), the audience (teenagers) and other details like the budget ($100,000,000$\$) without knowing the identifier the film will later have. While the blank nodes are unique in one document using the same blank node identifier over multiple documents can lead to misinformation. If one document describes another movie with the same identifier as another document the identifiers need to be renamed when merged. This prevents the two locally distinct blank nodes from becoming only one blank node in the merged document and therefore linking all statements to one node instead of describing two different objects.

Literals [32] are values such as a string, a date or a number. A literal can consist of two or three of the following elements:

- the **lexical form** is a Unicode string like `"This is Text"`. In order to ensure the equality of two lexical forms the string should be in Normalization Form C (NFC). The NFC checks for canonical equivalence (strict sequence of characters, like ä→ $a+\ddot{}$) and compatibility equivalence (weaker type of equivalence, like $\mathbf{a}$→a, $\boldsymbol{a}$→a). Still a certain value can have different lexical forms. The integer value '1' can be represented by the strings `"1"` and `" 1"` since the application will most likely ignore the space character while the lexical forms are determined as different.

- the **datatype IRI** maps the string of the lexical form to a certain datatype. This is necessary, since the lexical form `"1"` does not determine if the number or the letter is meant. With this method it is also possible to use own datatypes encoded in the lexical form. It is important to know that the combinations of lexical form and datatype IRI are not checked. For example, the mapping of the string `"XIII"` as a numerical XML Schema Definition (XSD) datatype is possible, while the application will fail at runtime to translate the characters to a number. On the other hand, an own datatype for roman numbers would allow the application to translate the string `"XIII"` to the integer value 13.

- the last possible element is the **language tag**. It is only possible to add this element to a literal if the chosen datatype IRI is set to a certain type[1]. With this datatype a non-empty, predefined language tag [33] can be added to the

[1] `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`

literal to identify the used language in the lexical form. For example, the tag `"de-AT"` represents German (`'de'`) as used in Austria (`'AT'`).

While a literal should consist of at least a lexical form and a datatype IRI, concrete syntaxes sometimes leave the standard datatype for a string[2] out. Most applications assume this datatype anyway, if no datatype is given. In this case such literals are called **simple literals** [34]. Literals with a given datatype are called **typed literals**. A **language-tagged string** is a literal with a language tag. Similar to the simple literals, syntaxes can leave the datatype out, since there is only one datatype that can have language tags. This means a language tag already implies the datatype and the datatype is not written out. Examples for this behavior are the SPARQL or Terse RDF Triple Language (TURTLE) grammar where only the abbreviation is allowed [35, 36].

IRIs, literals and blank nodes are collectively known as RDF terms and are distinct sets. This means that the same group of characters can be an IRI, literal string and blank node identifier without causing confusion. The use of an RDF term in an RDF triple is limited to certain parts of the triple.

Subjects are resources with a certain characteristic that need to be described. If the resource is known, an IRI is used to identify the resource directly, otherwise a blank node is used as a placeholder. A subject cannot be a literal since the value of a literal is used to describe a resource but is not a resource itself. Still a certain value can be described as a resource but is than again an IRI and not a literal. For example the letter A can be described with an IRI and statements over this resource are possible while the literal `"A"` is not a resource. Because subjects can only be IRIs and blank nodes, these are represented as nodes in a graphical representation. In order to represent their uniqueness a node only appears once in a graph which can be problematic for a resource with many statements

Predicates can only be IRIs. Since the triple should represent a statement about the subject a blank node as a predicate would not make sense. A so far undefined connection between a subject and an object could be anything and therefore validate a blank node connection between each possible subject and object. Further predicates are represented as edges in a graph representation of triples pointing from the subject node to the object. Blank node predicates would overload the graph presentation with undefined statements about resources that might exist. The same is true for a literal since the value is not an identifiable connection between the subject and the object. For example the strings `"isParentOf"`,

[2] `http://www.w3.org/2001/XMLSchema#string`

"hasTheChild" and "isParenting" all describe a parental connection between the subject and the object. Still an application would have to interpret these string value in the right manner while an IRI predicate is easily identifiable.

Objects can be all different types of RDF terms. An object represented by an IRI is another unique resource which has a connection to the subject. Further this object can itself be a subject to another statement which makes it possible that an IRI node can have in- and outgoing edges in a graphical representation. The same is true for a blank node object. For example a graph about heredity could use a blank node as an object for a subject representing an ape. The same blank node could be a subject to a human object which makes the blank node a 'missing link' in the heredity from ape to human. Since the literals can only be objects and never be subjects, all edges point to these nodes with no outgoing connections in the graph presentation. In order to present this uniqueness, often these literal nodes are represented in a different style than blank or IRI nodes in the graph presentation. Further the literal nodes are often duplicated even if they have the same values to represent these nodes local to unique resources in the graph.

In Listing 2.3 an example document with RDF triples is given. Further the Figure 2.5 is the corresponding graph representation of this document. The document starts with the XML declaration in the first line (see Section 2.1.3.3). This statement identifies the file as an XML document but not as an RDF document. Therefore the next two lines (line 2 and 3) declare that this is an RDF document ('rdf:RDF') and define two XML Name Spaces (XMLNSs) ('xmlns:rdf' and 'xmlns:own'). The XMLNS are shortcuts for long IRIs. For example the term 'rdf:RDF' starts with the shortcut 'rdf', is separated by a colon (':') and ends with a string that is added to the end of the shortcut forming an IRI[3]. These shortcuts are not necessary and only help human readers by avoiding long strings or save memory space because of the lesser number of characters. As an example the graph does not use the XMLNSs although a graph legend would be possible to identify the shortcuts. Still, graph and document represent the same resources and statement about them. In the document are two resources described. The first resource is the person Elizabeth II. from the lines 4 to 20 while the second resource is a picture of this person from lines 21 to 29. Both resources use the element type 'Description' in the 'rdf' XMLNS to mark a certain description to a resource while the 'rdf:about' attribute defines an IRI to this resource. Important at this point is that even though there is the 'own' attribute 'name' in the 'Description' element, it is not an identifier for Elizabeth II. as a resource which can be seen in the graph. Inside the graph both resources are represented as an ellipse node with the identifier in the middle. For Eliza-

[3] http://www.w3.org/1999/02/22-rdf-syntax-ns#RDF

beth (Person456) her name is a statement with a literal as an object. Therefore this attribute is represented as a node with the literal in the form of a rectangle. The same representation is used for the other elements describing the queen (`'own:title'` and `'own:alsoKnown'`). The difference between the literals as attribute or element is that the elements can directly describe their datatype by an attribute (`'rdf:datatype'` in lines 5, 8, 11, 14, 17, 22 and 25) while literals as an attribute can not directly be described and need a schema that guarantee the datatype of this attribute. Since the graph presentations purpose is the easy explanation of an RDF document in a visual way, the datatypes are mostly left out in order to prevent huge nodes with too much text. Even without their datatypes the literals can be identified by the different graphical presentation.

Listing 2.3: An example XML document with RDF triples describing two resources, a person and a picture of the person.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3       xmlns:own="http://localhost/predicates#">
4    <rdf:Description rdf:about="http://localhost/person#person456" own:name="Elizabeth
       Alexandra Mary">
5      <own:title rdf:datatype="http://www.w3.org/2000/01/rdf-schema#string">
6        Queen of the United Kingdom
7      </own:title>
8      <own:title rdf:datatype="http://www.w3.org/2000/01/rdf-schema#string">
9        Queen of Australia
10     </own:title>
11     <own:title rdf:datatype="http://www.w3.org/2000/01/rdf-schema#string">
12       Queen of Canada
13     </own:title>
14     <own:alsoKnown rdf:datatype="http://www.w3.org/2000/01/rdf-schema#string">
15       Elizabeth II.
16     </own:alsoKnown>
17     <own:alsoKnown rdf:datatype="http://www.w3.org/2000/01/rdf-schema#string">
18       The Queen
19     </own:alsoKnown>
20   </rdf:Description>
21   <rdf:Description rdf:about="http://localhost/pictures#picture234">
22     <own:name rdf:datatype="http://www.w3.org/2000/01/rdf-schema#string">
23       Elizabeth II. visiting Berlin
24     </own:name>
25     <own:URL rdf:datatype="http://www.w3.org/2000/01/rdf-schema#string">
26       https://en.wikipedia.org/wiki/File:Elizabeth_II_in_Berlin_2015.JPG
27     </own:URL>
28     <own:PictureOfPerson rdf:resource="http://localhost/person#person456"/>
29   </rdf:Description>
30 </rdf:RDF>
```

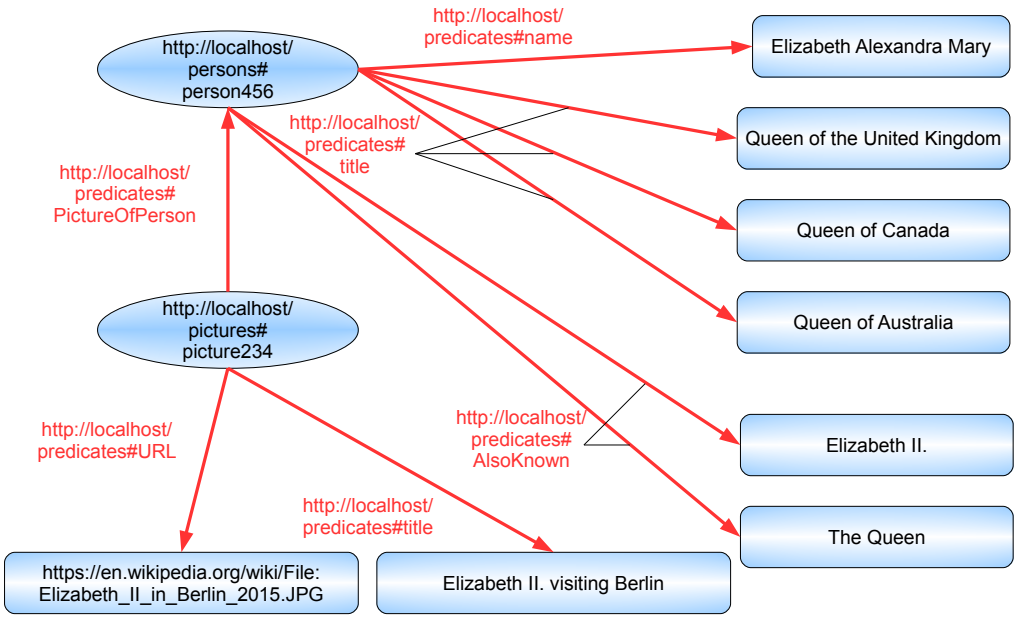


Figure 2.5: A graph representation of the XML document from Listing 2.3

For example the URL of the picture (line 25 to 27) looks in the graph presentation familiar to an IRI for a resource but is surrounded by an ellipse which identifies it as a literal. In the given example many differences are shown to make the graphical presentation easier understandable for humans. Still the RDF document is hard to read even with indented lines and new lines for XML elements.

Serialization formats allow the representation of the same RDF document in a different string representation than XML. Some examples for these formats are JavaScript Object Notation (JSON) for Linked Data (JSON-LD) [37], N-triples [38], N-quads [39], Notation 3 (N3) [40] and Terse RDF Triple Language (TURTLE) [36]. The JSON-LD format already implies the involvement of JSON in this format. Since JSON is used to describe data objects in an open-standard file format that uses human-readable text, it can also be used in JSON-LD to describe linked data, in this case the triples. The goal of JSON-LD is an easy transition from JSON to JSON-LD with little overhead for the developer. The other formats are connected to each other and focus more on the human-readability of the XML than on the easy conversion from a different file format. Starting with the N3 format most XML tags are shortened which makes this representation more compact and therefore better readable. Since the N3 format supports features that go beyond RDF-Serialization it is a superset for the TURTLE format which focuses only on the RDF representation. Therefore any application which can handle N3 can also

support TURTLE. The N-Triples format is a subset of the TURTLE format that focused on a simpler design to allow easier generation and parsing for applications. Because of this some shortcuts are not supported as in the other formats. This leads to longer IRIs since namespaces are not possible which also leads to more text to read and write for the human user. Still, if an application supports N3 or TURTLE, it also supports N-Triples. The last mentioned format is the N-Quad format. It is a superset for the N-Triple format that can extend a triple with a reference to its graph. In this case an RDF graph means a set of RDF triples and not its graphical presentation. Therefore triples of different origins can be addressed by their graph labels forming a quadruple with the other three RDF terms. A graph label is not necessary since triples without it are assumed to be part of a default graph.

The Listing 2.3 is converted from its XML/RDF format into the N3 format in Listing 2.4.

Listing 2.4: The Notation 3 (N3) representation of Listing 2.3.

```
1  @prefix own: <http://localhost/predicates#> .
2  @prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
3
4  <http://localhost/person#person456>
5    own:name "Elizabeth Alexandra Mary" ;
6    own:title  "Queen of the United Kingdom"^^rdfs:string,
7         "Queen of Australia"^^rdfs:string,
8         "Queen of Canada"^^rdfs:string  ;
9    own:alsoKnown "Elizabeth II."^^rdfs:string,
10          "The Queen"^^rdfs:string  .
11
12 <http://localhost/pictures#picture234>
13   own:name "Elizabeth II.  visiting  Berlin"^^rdfs:string  ;
14   own:URL "https://en.wikipedia.org/wiki/File:Elizabeth_II_in_Berlin_2015.JPG"^^rdfs:string  ;
15   own:PictureOfPerson <http://localhost/person#person456> .
```

Overall can be seen that the start- and end-tags normally used by XML are gone. The first two lines of Listing 2.4 are comparable to the XMLNS declarations in Listing 2.3 (lines 2 and 3). Starting with the string 'prefix' a new prefix is introduced followed by its shortcut (`own`, `rdfs`) ending with a colon (':'). The part of the IRI that is added as a prefix is surrounded by chevrons ('<','>'). After the prefixes are declared, the two resources of the original listing are described. Again Elizabeth II. as a person is represented as the IRI ending with '`person456`' while the picture of her uses the IRI ending with '`picture234`'. In the N3 format these IRIs are only surrounded by chevrons (lines 4 and 12) and indicate that the corresponding resource will be described further. In line 5 the name of the resource is

described by a literal which are surrounded by quotation marks ('"'). The line ends with a semicolon (';') indicating that the resource as the subject, the 'own:name' as the predicate and the literal as the object form a triple. Further this indicates that the resource is described by other statements but not with the same predicate again. The next line starts directly with the predicate 'own:title' followed by a literal as the object still describing the same resource as the subject. In this case the literal is directly followed by its datatype which is represented by two leading carets ('^^'). The end of the line is a comma (','), which indicates that the predicate and the subject is reused and only the object changes. Therefore, the next two lines only consist of a literal with its datatype and the last ending with a semicolon marking that the same resource is described with another predicate and object. The next two lines (lines 9 and 10) repeat the process from the former tree lines (lines 6, 7 and 8) using 'own:alsoKnown' as the predicate with two different literals. This time the last line (line 10) ends with a colon ('.') which signalizes that another resource is described in the following lines. The lines 12 to 15 consist of statements to the picture as a resource. So far there are no further formatting rules in this example and we used the N3 format only to describe RDF triples. This makes this example not only valid in N3 but also in TURTLE. But since prefixes are used the example is invalid in N-Triple and therefore also invalid in the N-Quad format.

Dictionary So far, the serialization formats only redefined the string representation of the RDF document. This can help the human reader and the machine interpretation alike. Another way to aid the machine processing is the usage of a dictionary mapping strings to integer values in exchange for a harder readability for human readers. The usage of a dictionary is not required for a SW DBMS to process RDF data but has certain advantages. Mapping RDF terms to integer Identifiers (IDs) lowers space requirements in the evaluation indices storing the input RDF triples each of which with three integers instead of possibly large strings. Using difference encoding [41] and avoiding to store leading zero bytes additionally saves space. Furthermore, using IDs enables space-efficient representations of (intermediate) solutions lowering the memory footprint: more solutions can be processed before swapping to Hard Disk Drives (HDDs)/Solid State Drives (SSDs) starts increasing the overall performance. For example, Listing 2.5 contains the ID triples of Listing 2.4 according to the dictionary in Table 2.1.

On the other hand using IDs causes high costs for the materialization of the RDF terms for (more seldom) operations like sorting or relational comparisons like $<$, $\leq$, $\geq$ and $>$, because these operations require the string representations of the RDF terms and not the IDs. Whenever the query result is large, displaying the final *textual* query result is also a costly operation. However, especially for large-scale datasets, the advantages typically outweigh the disadvantages of using IDs. Hence,

ID	RDF term
0	own:alsoKnown
1	own:name
2	own:PictureOfPerson
3	own:title
4	own:URL
5	"Elizabeth␣Alexandra␣Mary"
6	"Elizabeth␣II."^^rdfs:string
7	"Elizabeth␣II.␣visiting␣Berlin"^^rdfs:string
8	"https://en.wikipedia.org/wiki/File:Elizabeth_II_in_Berlin_2015.JPG"^^rdfs:string
9	"Queen␣of␣Australia"^^rdfs:string
10	"Queen␣of␣Canada"^^rdfs:string
11	"Queen␣of␣the␣United␣Kingdom"^^rdfs:string
12	"The␣Queen"^^rdfs:string
13	<http://localhost/person#person456>
14	<http://localhost/pictures#picture234>

Table 2.1: Possible dictionary for the RDF terms in Listing 2.4

Listing 2.5: ID triples of Listing 2.4 according to the dictionary in Table 2.1

```
13  1  5 .
13  3 11 .
13  3  9 .
13  3 10 .
13  0  6 .
13  0 12 .
14  1  7 .
14  4  8 .
14  2 13 .
```

many Semantic Web query evaluators such as RDF3X [41, 42] and Hexastore [43] as well as LUPOSDATE [44] use dictionary indices to map RDF terms into integer IDs.

2.1.4 Queries

Even though the Chapters 3, 4 and 5 focus more on the index structures handling RDF data, the query execution and therefore the handling of the

data itself gives a deeper insight into the design decisions in the former Section 2.1.3. Further, this section outlines the possibilities of SW DBMSs, like LUPOSDATE which is introduced in Section 2.1.5.

After introducing the data representation of the Semantic Web (SW) in the last section, the next step is the possibility to retrieve this data from Resource Description Framework (RDF) graphs. At the moment the most common language for this is the SPARQL Protocol And RDF Query Language (SPARQL) [45] which originated 2008 in Version 1.0 [46]. Since the RDF data representation was first specified in 1999 [47] and reached Version 1.0 in 2004 [48] there are predecessors and competitors to SPARQL. Some of the RDF query languages can be grouped into language families that related to each other closely [49]. In Figure 2.6 some of these families are shown.

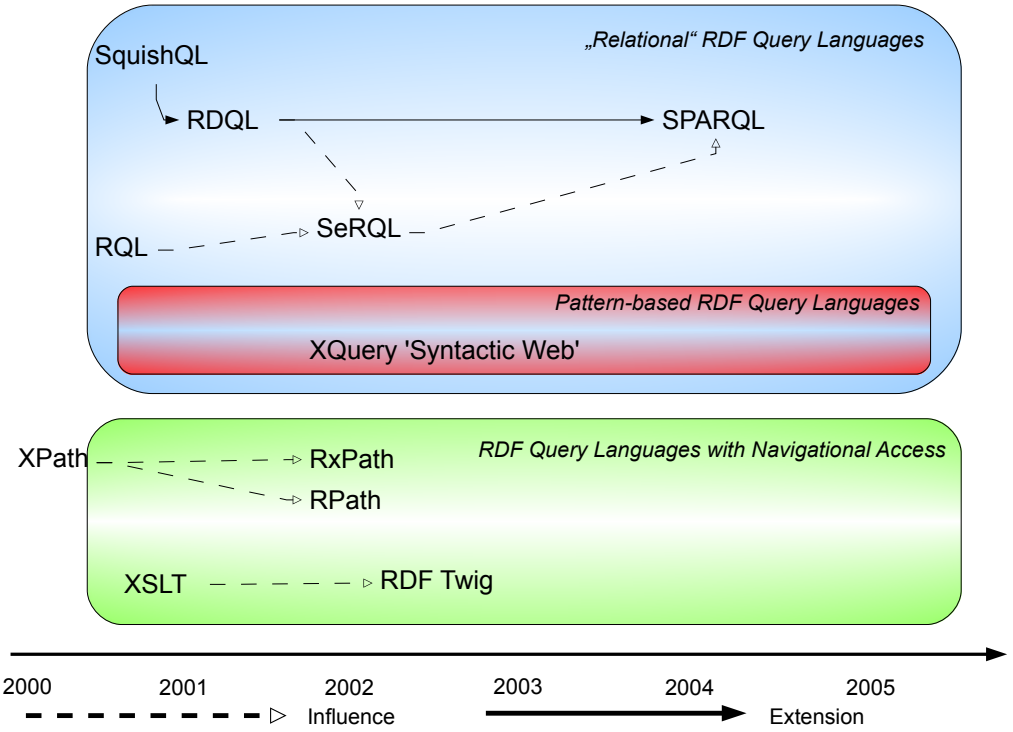


Figure 2.6: RDF query languages grouped into different families (inspired from [50])

The SPARQL family consists of languages which treat RDF data stores as triples that do not necessarily have ontology or schema information associated with them

and are considered relational query languages because they provide relational or pattern-based operations [50]. The first member of this family is the Squish Query Language (SquishQL). In the early days of RDF Query Languages (QLs), SquishQL focused on the already existing standard in relational Database Management Systems (DBMSs) which in this case is and was the Structured Query Language (SQL). This is also what the name tries to imply that this is a "SQLish" QL [51]. The idea was a language that could be picked up by SQL users in a very easy way. We will later come back to SQL and the similarities and differences between this language and SPARQL, but first we will stick to the RDF QL families. The RDF Data Query Language (RDQL) [52] derived from SquishQL and was submitted by Hewlett-Packard [53] as a standard to the W3C. But the standardization of RDQL never happened and instead it was extended to SPARQL which was introduced with the promise that "*Trying to use the Semantic Web without SPARQL is like trying to use a relational database without SQL*" by Tim-Bernes Lee [54].

The next family is the RDF Query Language (RQL) [55] family which influenced the SPARQL family. Starting with RQL this family focuses not only on the direct use of the triples in queries but also includes the RDF Schema (RDFS). This makes this family more expressive than the SPARQL family [4] by combining schema and data queries which makes classifications of resources, like being a subclass of another class possible. However, RQL uses stricter rules than the RDF and RDFS datamodel. As an example, the predicate '`own:PictureOfPerson`' needs two properties which describe the valid subject and the object. In this case, the '`domain`' property would limit the subject of the predicate to the class '`Picture`'. On the other hand the '`range`' property would limit the object of the predicate to the class '`Person`'. This means a resource described by '`own:PictureOfPerson`' is either of the picture class or the person class. These properties are normally optional but necessary in RQL. This and other restrictions, further the pure expressive strength make RQL a heavy-weight query language. In order to provide a light-weight and easy to access query language Sesame RDF Query Language (SeRQL) [57] was introduced which is influenced by RQL and RDQL [58].

The last family consists of languages reusing concepts of already existing XML query technologies. Since triples are XML based, an extension of an XML query language to an RDF query language seemed a reasonable alternative. One common language for XML queries is the XML Path Language (XPath) which became a standard in 1999 with version 1.0 and is currently at version 3.1 [59]. XPath

[4] The used 'family' term only references to SPARQL 1.0 and its ancestors (SquishQL, RDQL) of the monitored time period from around 2000 to 2005. The later SPARQL 1.1 [35] standard from 2013 allows so called 'Entailment Regimes' [56] enabling the usage of RDFS or OWL. This enhances the expressiveness of SPARQL 1.1 comparable to the RQL family.

allows the navigation inside a graph of an XML document. In Section 2.1.3.4 the transformation of an RDF document into its graphical presentation is introduced (see Listing 2.3 and Figure 2.5) which can be achieved in a similar matter for XML documents. Languages, like RDF Path (RPATH) [60] and RDF for XPath (RxPATH) [61], enable the navigation in these graphs to search certain nodes or values. However, circles inside the graph need to be handled otherwise infinite duplicates can occur by revisiting nodes multiple times. RxPATH solves this by revisiting the so far taken path in search for the same IRI as the current IRI [61]. Still, the characteristics of XPath limit the structure of the query results. It is only possible to receive either single values, like a string or a boolean, or an unordered set of nodes. In order to structure and manipulate the results of an XML Query, languages like XML Query (XQuery) [62] or Extensible Stylesheet Language Transformations (XSLT) [63] can be used. Both integrate XPath but allow further functions than graph navigation. While XQuery focuses more on the order and the presentation of the query results in another XML document, XSLT is used to transfer given data into different formats like converting Hypertext Markup Language (HTML) into a Portable Document Format (PDF) file. Robie et al. suggested the direct use of XQuery on the RDF data, calling this the 'syntactic web' [64] [65]. They propose how XQuery can be used to imitate special features normally only special-purpose QLs for RDF data do support. Still, they admit that type safety can be lost in this process. For the XSLT approach, similar problems to the XPath approach occur. Transferring an RDF graph into XML needs a root and should avoid a cycle in the graph to build a tree. RDF Twig [66] allows the generation of a tree or subtree (in this context called 'twig') from a certain starting node with the additional dissolving of any cycles.

After this history of possible QLs for the RDF data we will focus on the SPARQL family and its connection to SQL. Since the RDF for SPARQL is already introduced, the data model of SQL is examined. SQL uses the Relational Model (RM) where all data is represented in terms of tuples with a certain number of attributes and grouped into relations. In Figure 2.7 multiple examples of relations are given.

There are five different relations in this picture which are graphically represented by tables, namely name, title, alsoKnown, URL and PictureOfPerson. Each of these relations have two attributes in these examples represented by the columns subject and object. So far, the relations with their attributes only describe how the information is structured but not how the actual data looks like. The tuples are the direct shaping of the data represented by the rows of each table. For example, the relation name has two tuples, giving the entities `person456` and `picture234` their corresponding names. These entities can be identified by making the subject attribute of the name relation a primary key, which means for a tuple that this attribute cannot be empty or exist multiple times in the same relation. With this

name	
subject	object
person456	Elizabeth Alexandra Mary
picture234	Elizabeth II. visiting Berlin

title	
subject	object
person456	Queen of the United Kingdom
person456	Queen of Canada
person456	Queen of Australia

alsoKnown	
subject	object
person456	Elizabeth II.
person456	The Queen

URL	
subject	object
picture234	https://en.wikipedia.org/wiki/File:Elizabeth_II_in_Berlin_2015.JPG

PictureOfPerson	
subject	object
picture234	person456

Figure 2.7: Multiple relations portrayed as tables depicting the predicates of Listing 2.3. Further, the subjects and objects are represented as columns of the tables.

primary key it is possible to connect information of other relations with each other describing the same entity. Together with the title relation the `person456` can be named and all three titles can be identified using the subject attribute as a foreign key in the title relation which originated from the name relation. This technique makes it possible to connect all data from all relations to describe the two given entities just like in Listing 2.3 or Listing 2.4. Of course, this transformation of RDF to the RM needs sometimes quite an effort. While in Figure 2.7 the predicates are used as relations and the subject and the object are attributes, it is possible to create just a triple relation with three attributes as in Figure 2.8.

triples		
subject	predicate	object
person456	name	Elizabeth Alexandra Mary
person456	title	Queen of the United Kingdom
person456	title	Queen of Canada
...	...	...
picture234	PictureOfPerson	person456

Figure 2.8: A single relation portrayed as a table named 'triples'. The columns subject, predicate and object contain parts of the data from Listing 2.3

In this example transformation errors are easy to be spotted. The data type of the object attribute cannot be limited to a certain type. The first three tuples contain a string as a data type while the tuple with the `PictureOfPerson`-predicate references an actual resource and not a literal. Even though the first example in Figure 2.7 has more opportunities to limit or support the corresponding datatype special constructs like the blank nodes or language-tagged strings are hard to address.

For a small comparison of the two Query Languages (QLs) the example of Figure 2.7 is still sufficient. The Listing 2.6 is an SQL query on the relations in Figure 2.7.

Listing 2.6: An SQL query to request the name of a person and the corresponding URL of a picture of this person. The used relations are the same as depicted in Figure 2.7

```
1 SELECT n.object, u.object
2 FROM name n, URL u, PictureOfPerson p
3 WHERE
4   n.subject   = p.object  AND
5   p.subject   = u.subject;
```

For SPARQL, the Listing 2.7 contains a query on the RDF graph in Listing 2.3.

Listing 2.7: A SPARQL query to request the name of a person and the corresponding URL of a picture of this person.

```
1 PREFIX own:<http://localhost/predicates#>
2 SELECT ?name ?url
3 FROM <http://website.own/example>
4 FROM <http://a.different/example>
5 WHERE {
6   ?picture  own:PictureOfPerson ?person.
7   ?picture  own:URL ?url.
8   ?person own:name ?name.
9 }
```

Both queries are searching for a person that was taken a picture of and the corresponding URL of this picture. The main keywords of both QLs are capitalized. The **`PREFIX`** keyword is similar to the prefixes in the N3 example in Listing 2.4 shortening the long IRIs to a smaller string. Similar to that, a relation can be renamed in an SQL query by an alias, shortening the relations `name`, `URL` and `PictureOfPerson` to `n`, `u` and `p` (line 2 of Listing 2.6). The **`SELECT`** keyword in both queries defines the search result. Although the result of the **`SELECT`** clause

is the same for both QLs, the representation differs because of the different data models. SQL directly addresses certain relations and corresponding attributes to extract data of matching tuples, for example the names of the searched persons are inside the `name` relation (shortened with `n`) in the `object` attribute. The direct addressing of these attributes cannot be done in SPARQL because relations do not exist. Instead variables indicated by a question mark (`'?'`) are used. While the data type in SQL can be directly identified by the relation and the attribute, the data type of the SPARQL variable can be unclear depending on the position in the triple. For example a variable that is searched at the object position can return IRIs, blank nodes or literals, the latter each with a possible different data type, in any combination. Here, the first difference is in the output of the result, since SQL produces a new result relation which allows the nesting of multiple SQL results. Instead the result of the SPARQL query with the **`SELECT`** clause only returns the variable bindings without reproducing an RDF graph. The next keyword is the **`FROM`** clause. In SQL there is and needs to be only one clause listing all relations that are involved in the search process. As already mentioned a renaming of the relations can be done at this point and the different relations are separated by a comma (`','`). The SPARQL query can contain multiple **`FROM`** clauses or none at all, then working on a predefined default graph. Each **`FROM`** clause has one IRI which together form the graph for the query. In the example of Listing 2.7 are two different graphs used (lines 3 and 4). This shows an important difference in the conception of these QLs. SQL expects the relations to be locally stored and directly accessible on the same host while SPARQL expects a web of data sources that needs to be addressed. Of course distributed databases are also possible with SQL but the addressing of a remote relation can be done in multiple ways [67]. For example in Oracle [68] remote resources can be addressed by an `'@'` followed by a database link [69] (`relation_name@database_link`). The last important clause in these examples is the **`WHERE`** clause. In this clause the conditions for a match are represented. For the SQL query two statements must be fulfilled. First, the ID of the person must be the same as the person that is on the picture, so the name is correct (line 4). Second, the ID of the picture must be the same as the picture that is portraying a person, so the URL is correct (line 5). These two requirements are connected by the **`AND`** keyword. The SPARQL query instead uses three different patterns that need to match with the given triples in the graph. The first pattern in line 6 of the Listing 2.7 is indirectly included in the SQL example by the relation PictureOfPerson. This relation p, as it is shortened, binds the object in the first condition with the subject of the second condition, since these attributes must belong to the same tuple. On the other hand, the RDF does not have such indirect context between the triples, therefore the pattern is important to connect the variables `?picture` and `?person` with each other. Otherwise the result would be all pictures with an URL and every person with a name without

connecting both entities.

After the short comparison of SQL and SPARQL, the language constructs of SPARQL are examined further without direct comparison due to simplicity of presentation. There are four types of SPARQL queries, namely the **ASK**, **CONSTRUCT**, **DESCRIBE** and **SELECT** queries. The **SELECT** query was already introduced in Listing 2.7 declaring the searched variables and giving back the matching bindings in the graph. Still the result of the query does not resemble a graph which makes nesting multiple **SELECT** clauses impossible. In order to be able to nest queries and build new RDF graphs the **CONSTRUCT** can generate new RDF triples which form a new graph. The Listing 2.8 shows an example of a **CONSTRUCT** query.

Listing 2.8: A SPARQL query using the **CONSTRUCT** keyword to generate a new graph.

```
1 BASE <http://localhost/predicates#>
2 CONSTRUCT {
3   ?person  <PersonInPicture> ?picture
4 }
5 WHERE {
6   ?picture  <PictureOfPerson> ?person.
7 }
```

The example query generates a new predicate `<PersonInPicture>` that connects a person to multiple pictures of themselves (line 3). Since the predicate `<PictureOfPerson>` gives a similar statement about pictures which portraise one or multiple persons the subject and object can be switched (line 6). This is done by using the **WHERE** clause in a similar way like in the Listing 2.7 example with the **SELECT** clause and binding the variables in a certain pattern. These bindings are used in the **CONSTRUCT** clause to fill the missing variables therefore generating new triples. A small addition in Listing 2.8 is the **BASE** keyword. It has a similar function like the **PREFIX** keyword that was already introduced in Listing 2.7. In Listing 2.9 the original strings are presented without using the **PREFIX** or **BASE** keyword.

Listing 2.9: Three possible prefixes as example data

```
1 <http://localhost/pictures#Picture>
2 <http://localhost/pictures/blackAndWhite#Picture>
3 <http://otherLocation.org/photos#Picture>
```

The first two strings are similar and look like they are ordered in a hierarchical

way. The string in the last line is instead very different to the other two. So far, the solution only using the **PREFIX** keyword would look like in Listing 2.10

Listing 2.10: The usage of the **PREFIX** keyword on the same three prefixes from Listing 2.9

```
1 PREFIX pic1:<http://localhost/pictures#>
2 PREFIX pic2:<http://localhost/blackAndWhite#>
3 PREFIX pic3:<http://otherLocation.org/photos#>
4 pic1:Picture
5 pic2:Picture
6 pic3:Picture
```

Since this example should only show the possibilities of representing an IRI in different ways, there is only the definition in the first three lines and then the usage of the prefix in the last three lines without a proper query. The usage of these three prefixes also hide the similarities between the first and the second IRIs which is not necessary intended. The **BASE** keyword in Listing 2.11 gives a solution for that problem.

Listing 2.11: The usage of the **BASE** and **PREFIX** keyword on the same three prefixes from Listing 2.9

```
1 BASE <http://localhost/pictures>
2 PREFIX pic:<http://otherLocation.org/photos#>
3 <#Picture>
4 </blackAndWhite#Picture>
5 pic:Picture
```

With the addition of a common path in the first line, the first two IRIs can use this path. In comparison to the **PREFIX** keyword where a shortcut has to be defined the path of the **BASE** keyword is unique. Further, the usage of a prefix or a common path can be seen directly on the IRI that is described. In order to use the common path the differences are surrounded by chevrons ('<','>') to resemble the original IRI. On the other hand, a colon (':') is used to divide the shortcut of the prefix and the rest of the IRI. Since the last IRI does not share a path with the other two, a prefix is used. Here is the difference between the **BASE** and the **PREFIX** keyword, while multiple prefix shortcuts can be defined the **BASE** keyword is limited to be used only once.

After this excursion into the IRI presentation of SPARQL there are two different query types left. The third query type is the **ASK** query. An example of this query type can be seen in Listing 2.12.

Listing 2.12: A SPARQL query using the **ASK** keyword to answer the question whether a certain triple statement exists.

```
1 PREFIX own:<http://localhost/predicates#>
2 ASK {?picture own:PictureOfPerson ?person}
```

In the **ASK** query variables are used but never specified as a result as in the **SELECT** query. Further no triples are generated as in the **CONSTRUCT** query, rather a direct Boolean value is given back indicating if a solution exists or not. If the given ASK query would be used on Listing 2.4 the result would be '`true`', since in line 15 a matching triple exists. On the other hand, if the **?person** variable would be replaced by the literal **'john'** there would not be a matching triple and the answer would be '`false`'. Therefore the **ASK** query just indicates that at least one matching triple exists or not, but does not give further information like the amount of results or any other indication about the result set.

The last query is the **DESCRIBE** Query that returns relevant triples on certain variables. In Listing 2.13 a small example is given, how a **DESCRIBE** query looks like.

Listing 2.13: A SPARQL query using the **DESCRIBE** keyword to get all triples connected to a certain resource.

```
1 PREFIX own:<http://localhost/predicates#>
2 DESCRIBE ?person
3 WHERE {?picture own:PictureOfPerson ?person}
```

Like in a **SELECT** query the keyword **DESCRIBE** is followed by one or more variables indicated by a question mark ('?') that should be described. In this case 'describe' means all triple statements that include the current binding of the variable are solutions. This also means that **DESCRIBE** queries, like the **CONSTRUCT** queries, assemble a new RDF graph, unlike **SELECT** queries where only the bindings of the variables are presented. If the query from Listing 2.13 would be executed on the Listing 2.4 the only binding of `?person` would be `person456` [5] since the optional **WHERE** clause limits the possible bindings. The result set would be at least the triples from line 4 to 10 of the Listing 2.4 and the last line 15. Still, the result set could be enhanced by the information publisher. This is possible since the **DESCRIBE** keyword is only marked as 'informative' in SPARQL Version 1.0 [46] and was not updated in Version 1.1 [35] which allows different interpretations in the direct implementations. In the given example the additional information of

[5] actually <http://localhost/person#person456>

the picture portraying the searched person could help the questioning person to understand what the statement in line 15 means, since the IRI of the resource does not necessarily describe what this resource is. Another example for the use of a **DESCRIBE** Query is the direct use of an IRI to an RDF graph, like in Listing 2.14.

Listing 2.14: A minimal SPARQL query using the **DESCRIBE** keyword to retrieve a whole RDF graph.

```
1 DESCRIBE <http://localhost/predicates#>
```

As a result set the whole RDF graph could be returned to the user which at the beginning had no other information than the IRI and can now comprehend the structure of the graph. But this also produces the problem which information is necessary. Nokia [70] made a submission about Concise Bounded Description (CBD) for RDF graphs to address this problem for SW agents [71]. While the human reader can still understand inconsistencies or ambiguity in the result set, an SW agent is not so flexible and can deal better with a simple, clear structure in the RDF graph. For this reason, the CBD reduces the original graph to a subgraph intended to be used by an SW agent. As already mentioned, this also contributes to the variety of the result sets of **DESCRIBE** queries. While the intention of the keyword is clear, the result set depends highly on the implementation of the information provider.

After the introduction of the different query types the next step is the handling of different RDF graphs in SPARQL. In the example comparison between SQL and SPARQL the keyword **FROM** was introduced. This keyword allows the redefinition of the *default RDF graph* which is normally defined by the implementation the query is running on. In order to understand this keyword the definition of an RDF dataset G_{set} is presented.

$$G_{set} = \{G_{def}, (< IRI_1 >, G_{IRI_1}), (< IRI_2 >, G_{IRI_2}), \cdots, (< IRI_n >, G_{IRI_n})\}$$

The G_{def} is the default graph that is always existent and can be changed by the **FROM** keyword. In Listing 2.7 the use of two FROM keywords changes the system predefined Graph G_{def} to the new default graph G'_{def} which is a union of the two by the IRIs described graphs. The G_{set} changes from

$$G_{set} = \{G_{def}\}$$

to the new set G'_{set}

$$G'_{set} = \{G'_{def}\} = \{G_{IRI_1} \cup G_{IRI_2}\}$$

through the use of the keyword. In this case the graphs G_{IRI_1} and G_{IRI_2} are the graphs described by the IRIs in the lines 3 and 4. Next to the default graph G_{def} the dateset G_{set} can contain multiple, so called *named graphs* which consist of an IRI ($< IRI_i >$) which is also the 'name' and the corresponding graph (G_{IRI_i}). This makes it possible to switch the *active graph* within a query. The *active graph* is the graph from the dataset used for basic graph pattern matching. If a query does not use the **FROM** keyword and has no *named graphs* it falls back on the default graph G_{def} predefined from the system and sets it the *active graph*. If the G_{set} is changed by the **FROM** keyword, only the default graph G_{def} is altered but stays the active graph. In order to define a *named graph* the **FROM** keyword is enhanced by the keyword **NAMED**.

Listing 2.15: A SPARQL query using the **FROM** and **FROM NAMED** keywords.

```
1  PREFIX own:<http://localhost/predicates#>
2  SELECT ?name ?namedGraph ?date
3  FROM <http://localhost/graphList>
4  FROM NAMED <http://localhost/persons>
5  FROM NAMED <http://localhost/pictures>
6  WHERE {
7    ?namedGraph own:changed ?date.
8    GRAPH ?namedGraph {
9    ?x  own:name ?name.
10   }
11 }
```

In Listing 2.15 an example query is shown. In line 3 the default graph is set to the corresponding IRI:

$$G_{def} = G_{<http://localhost/graphList>}$$

Further, lines 4 and 5 introduce two named graphs:

$$\begin{gathered}(< http://localhost/persons >, \\ G_{<http://localhost/persons>}), \\ (< http://localhost/pictures >, \\ G_{<http://localhost/pictures>})\end{gathered}$$

These named graphs are set as active graphs by the **GRAPH** keyword. In this case the variable `?namedGraph` is limited to the IRI of named graphs:

$$?namedGraph \in \{< http://localhost/persons >, \\ < http://localhost/pictures >\}$$

Further, the pattern matching of line 9 is executed on the named graphs $G_{<http://localhost/persons>}$ and $G_{<http://localhost/pictures>}$ but not against G_{def} since this graph is not active at that point. On the other hand, the pattern matching in line 7 is evaluated on the default graph G_{def} but not on the named graphs $G_{<http://localhost/persons>}$ and $G_{<http://localhost/pictures>}$. Hence, the bindings of the result variables are limited to certain graphs. The binding of the variable `?name` is part of the named graphs $G_{<http://localhost/persons>}$ and $G_{<http://localhost/pictures>}$. Further, the variable `?namedGraph` will be either $< http://localhost/persons >$ or $< http://localhost/pictures >$ depending on the source graph ($G_{<http://localhost/persons>}$ or $G_{<http://localhost/pictures>}$) of the binding of `?name`. The bindings of the last variable `?date` are limited to the default graph G_{def}.

2.1.5 LUPOSDATE

After the introduction of the main SW concepts in Section 2.1.2, 2.1.3 and 2.1.4, a DBMS that fulfills these requirements is introduced. Each of the Chapters 3, 4 and 5 use LUPOSDATE as the main SW DBMS.

The LUPOSDATE project [44] [72] is a SW DBMS developed by the Universität zu Lübeck. It uses SPARQL [46] engines with logical and physical optimization and supports full SPARQL 1.1. In this work LUPOSDATE will be the main SW DBMS used for any evaluation. Further, in Chapter 5 LUPOSDATE will be the software part of a hybrid system with an FPGA. In this scenario LUPOSDATE provides the major functionalities for SW operations while the FPGA, as the hardware part, is used for specialized tasks. LUPOSDATE uses a full B^+-tree structure for indexing. In this B^+-tree each key in a node has a pointer to its child. The data inside LUPOSDATE is stored as RDF triples. The triples are stored in all six different combinations (*spo*, *sop*, *pos*, *pso*, *osp*, *ops*) to maximize the number of possible merge joins [43] [41]. To avoid large strings while searching, the SPARQL engine of LUPOSDATE uses a dictionary like Hexastore [43] or RDF-3X [41]. In this dictionary the RDF-terms are mapped to integer IDs (using 96 bits per triple, 32 bit each component) and vice versa. By using IDs the intermediate results and the result consume less memory. On the other hand the mapping from IDs to strings is time consuming for large query results. In Big Data scenarios the dictionary indices typically do not fit into main memory. LUPOSDATE uses a B^+-tree for storing the mapping of RDF terms into integer IDs (and hence the key of the B^+-tree is the string representation and the value is the integer ID). The

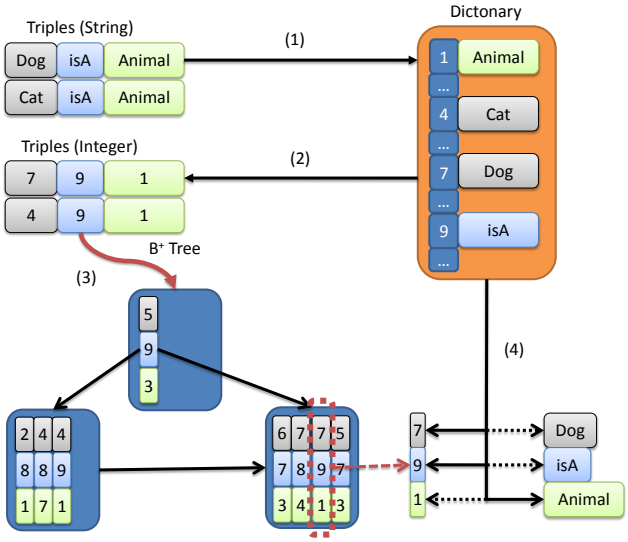


Figure 2.9: Two triples in string representation are transformed into their integer ID representation with the use of a dictionary. ID triples are stored in a B$^+$-tree and results are transformed back into string representation.

other mapping direction is maintained in a file-based array of pointers addressing the strings of RDF terms in a second file. For looking up the string representation of an ID the position of the pointer in the first file is calculated by multiplying the ID value with the pointer size. Then the string can be directly accessed in the second file according to the retrieved pointer. Hence only two disk accesses are necessary for each lookup.

A simplified example is presented in Figure 2.9. In this scenario there are two triples in string representation giving the information that cats and dogs are animals. The first step (1) is the generation of the dictionary giving each component of the statements a corresponding integer ID. In the second step the original statements are transformed into an integer representation using the IDs from the dictionary. These ID triples are indexed in six B$^+$ trees but in order to keep the example simple only the tree for the *spo* combination is presented in step (3). For a search the fixed components are mapped from strings to integer. In step (4) all statements about **'dog'** as a subject and **'isA'** as a predicate are queried. Therefore, the string are mapped by the dictionary to the IDs **7** and **9**. Inside the B$^+$ tree the ID **7** of the **'dog'** is bigger than the ID **5** of the root key leading to a continued search in the right child node. There, only one triple matches the subject and predicate of the query. At last, the object is mapped to its string representation **'animal'** using again the dictionary.

Further details about LUPOSDATE are presented in [44].

2.2 Reconfigurable Computing

This section introduces the concept of the reconfigurable computing which is important for the Chapter 5.

This section gives an overview over different aspects of reconfigurable computing architectures, especially about Field-Programmable Gate Arrays (FPGAs). First, the historical context of reconfigurable computing is examined, giving the reason why reconfigurable hardware, like FPGAs exist next to Central Processing Units (CPUs) and Application-Specific Integrated Circuits (ASICs).

2.2.1 Origins of reconfigurable Hardware

This section sums up the history about Integrated Circuits (ICs) before presenting FPGAs in Section 2.2.2. This summary is shared by the history of CPUs in Section 2.2.1.1 and the history of ASICs in Section 2.2.1.2.

Reconfigurable computing evolved between the two principals of Central Processing Units (CPUs) and Integrated Circuits (ICs). While CPUs are limited to direct the control flow without options to alter the datapath in a significant way, reconfigurable hardware, like FPGAs are able to do so. Further, the possibility of adapting the hardware to a new task on runtime makes reconfigurable hardware more flexible than hardwired ICs. Still all mentioned techniques and architectures share their origin in the history of digital electronics.

Before the direct start into digital electronics it is important to know what *'digital'* means. A *digital* signal can only represent specific discrete values instead of continuous changes of an *analog* signal [73]. This is comparable to a water tap with hot and cold water supply. The analog way of operating the tap would be the opening or closure of the valves in order to get the desired temperature of the water. In this scenario the hot water is 40°C and the cold water is 10°C which makes a direct, analog selection of the water temperature between those two values possible. For the digital representation of the tap the mixture of the differently tempered water would still happen as in the analog scenario but the steps are limited. As an example the steps would be `'hot'` between 40°C and 30°C, `'lukewarm'` between 30°C and 20°C and `'cold'` between 20°C and 10°C. A desired change from `'hot'` to `'cold'` water would result in a longer time period in the `'lukewarm'` step before the temperature is below the 20°C mark and the `'cold'` state is reached.

This masking of changes can be desired to represent easier models, in this case a child would not need the understanding of the Celsius temperature scale and just the understanding of states hot and cold. Limiting the digital signal to only two possible values allows the usage of the signal for a *Binary Digit* (Bit). Depending on the purpose of the bit the representation of the values can differ, for example `true/false` as logical definitions or `on/off` for activation state of a machine. Further on in this work all bit values are represented by `0/1`.

With the bit as a basic unit for information the next step to advance into digital technology are the devices working with it. The first ancestors of the modern computer were the mechanical calculators of the 17th century. Different designs of Wilhelm Schickard [74] ('Rechenuhr' <ger>, calculation clock) or of Blaise Pascal [75] ('Pascaline' or Pascal's calculator) allowed the subtraction or addition of two numbers. This happened, as the term 'mechanical' already states, without an electric power source and was realized by entering the two numbers over metal wheel dials that operated different gears, similar to a pocket watch. After the input was finished the result was displayed by a cylinder per digit from zero to nine. This also means that these calculations were performed at the base of 10. The *'Analytical Engine'* [76] of Charles Babbage can be seen as the next step towards modern computers. Even though the machine was never finished before his death the idea of using so called *'punched card'* to give instructions and input data into the machine made it usable for different tasks. These *punched cards* were pieces of paper with predefined positions for wholes, which either existed or not. This resembles a bit representation of a statement defined by the position on the paper. As a result of this flexibility Ada Lovelace published an additional note on how to calculate Bernoulli numbers with the *Analytical Engine* while translating a description of the *Analytical Engine* from Luigi Federico Menabrea from French to English [77]. Still, the pure mechanical design of the machine made a realization at that time hard to achieve.

Through new developments in electrical engineering pure mechanical parts were replaced by electrical compounds. Especially, relays, basically electrical operated switches, allowed better control over the functions of the machine. The basic concept of a relay was an electromagnet that could be activated or deactivated to trigger a mechanical closure or opening of another circuit. Konrad Zuse started in 1935 with the design of a mechanical computer which he built from 1936 to 1938 that was called Z1 [78]. The Z1 was only electronically operated by a motor to move the otherwise mechanical parts getting instructions by a 'punched tape', similar to a punched card but in tape form. Often the mechanical parts jammed leading to long maintenance phases. In the next iterations, Zuse switched the arithmetic and control logic parts with electrical relay circuits in the Z2 and later replaced the mechanical memory by relay driven memory in the Z3 [79]. For the Z3 it can

be shown that it is turing-complete under specific circumstances since conditional branches in the instruction set are absent [80]. But not only in Europe calculation machines were developed. At around the same time in 1937 Howard Aiken began the development of the Mark I [81] at the Havard University and finished it in 1944 [82]. Like the Z3 the Mark I used relays as a key component but was using arithmetic to the base of 10 instead of base 2.

Another technology to resemble an electronic switch were vacuum tubes. The first version, the diode, was not a switch but a rectifier. A rectifier converts Alternating Current (AC), which periodically reverses direction, into Direct Current (DC), which has only a constant direction. The early diode consisted of a glass tube and two conductors inside. It was sealed in such a way that a vacuum could exist inside the tube. The cathode, in this case the negative conductor, is heated to emit electrons through the vacuum to the anode. The triode is built from the diode by adding a grid between the cathode and anode to direct the flow of the electrons. Placing a negative voltage onto the grid will have the effect of repelling some electrons back to the cathode and thereby reducing the number of electrons traveling to the anode. In this way the voltage on the grid acts as a controlling voltage that controls the amount of current that flows in the anode circuit. While there are multiple further types of vacuum tubes the importance for computers began with the triode as an alternative to relays.

Eckert and Mauchley [83] developed the Electronic Numerical Integrator and Computer (ENIAC) at the University of Pennsylvania. The ENIAC was presented in 1946 to the public and used a little less than 18,000 triodes to calculate results using the decimal numeral system. The advantage of triodes compared to relays is the faster execution of changing the state from one to another. Relays perform a physical movement to close or open the controlled circuit while triodes only change the voltage of the grid. This is one of the reasons [6] the Z3 from 1941 had only a frequency of 5-10 Hz [84] while the ENIAC had a frequency of 100 kHz [85]. Still, vacuum tubes had their own issues while the Z3 had a power consumption of about 4,000 Watt [84] the ENIAC had a consumption of about 174 kW [86]. The reason for this is the heating of the cathode which needs between 1,100 to 2,500 K [87] depending on the material of the cathode to operate.

Another change came when Bardeen, Brittain and Shockley invented the transistor [88] in 1947 [89]. Introduced as a "*semi-conductor triode*" [88] the basic structure looked similar to a triode. There are three contacts, namely *emitter*, *collector* and *base*, where the electron flow between *emitter* and *collector* is controlled by the *base*. The difference between triodes and transistors is the used material, in this case a semi-conductor material instead of a vacuum. Such material falls between

[6] Another reason would be the motor moving the mechanical parts.

conductors, such as copper, and isolators, such as glass, in terms of electrical conductivity. It can be '*doped*' by placing impurities into the source material which leads to positive or negative areas depending if the introduced material is lacking or adding electrons to the semi-conductor material. The *emitter* and *collector* are doped in the same polarity while the *base* controls an area between the two contacts of opposite polarity. In order to control the flow between *collector* and *emitter* the circuit between *base* and *emitter* must be closed otherwise the flow of electrons between *collector* and *emitter* is blocked. With this invention the size and power consumption of calculators could shrink further as it already happened with the use of vacuum tubes.

So far the used 'switches' were single entities in a complex design. The next step was the combination of multiple circuit components, like transistors or diodes, into a single semi-conductor element. These elements are called Integrated Circuits (ICs) and the first functional IC was developed by Jack Kilby in 1958 [90]. Three years later, in 1961 James L. Buie invented the Transistor–Transistor Logic (TTL) [91]. The TTL ICs used only Bipolar Junction Transistors (BJTs) and resistors to perform logic and amplifying functions. Instead of rebuilding a complete circuit with single components resembling a logical gate, like an AND function, the TTL ICs could directly support this feature and therefore save wiring space. Still, many ICs were needed to build a more complex systems. Therefore, these ICs became later known to be Small-Scale Integration (SSI) with a small number of transistors and minor complexity [92]. The used BJT still consumed power while idle which made a higher density in the ICs harder to achieve. The Field-Effect Transistor (FET) solved this problem, since it is voltage controlled instead of current controlled. FETs have three terminals, namely *source*, *drain* and *gate*, which are similar in function to the *emitter*, *collector* and *base* of the BJT, although the physical background differs. The *gate* can control the flow of electron from the *source* to the *gate*. The patent for an FET was entered in 1925 by Lilienfeld and published in 1930 [93] but the production cost were higher than the costs for BJTs. After the introduction of the Metal–Oxide–Semiconductor (MOS) technology new MOSICs with FETs replaced the TTLICs with BJTs. A MOSFET has an added *body* which is a silicon substrate topped by silicon dioxide. The ICs are manufactured by using so called 'wafers' made of silicon and building components and connection of transistors in different layers. This so called 'planar process' was patented by Hoerni in 1962 [94] and solved the problem of the high production costs.

With this production technique and better production facilities the ICs could handle an increasing number of transistors which lead to two twigs in the micro processing field, the Central Processing Unit (CPU) and the Application-Specific Integrated Circuit (ASIC).

2.2.1.1 Von Neumann architecture and the general-purpose Central Processing Unit (CPU)

After the history of ICs in Section 2.2.1 this section presents the history of the CPUs in contrast to ASICs in the next Section 2.2.1.2.

The first publicly available microprocessor, the 'Intel 4004', was released by the Intel Corporation [95] in the year 1971 [96]. A total number of 2,300 transistors were held by the Intel 4004. It was build in Harvard architecture, which originated from the already mentioned Mark I [81] of the Harvard university. This architecture type separates the programmed instructions from the data by physically independent storage and signal pathways. Therefore the custom programs for the Intel 4004 could be stored in cheaper Read-Only Memory (ROM) while the processing data could be stored in Random-Access Memory (RAM). One advantage of this separation was the parallel handling of instructions and reading/writing data at the same time. Further the used memories could not only differ by their technology but also in word width, timing and addressing. Still, this could also be a disadvantage depending on the desired program. A program with few instructions that processes a huge amount of input data wastes memory space in the instruction memory while the data memory cannot hold the data. Similar, a program with many instructions and a minimal amount of input would not fit into the instruction memory while the data memory is mostly empty. Another architecture type therefore was the Von Neumann architecture. This architecture was proposed by John von Neumann in 1945 for the Electronic Discrete Variable Automatic Computer (EDVAC) [97]. He was inspired by the works of Alan Turing [98] [99] and his idea of a Universal Turing Machine (UTM) [7] which he described as following:

> *"It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $\mathcal{U}$ is supplied with a tape on the beginning of which is written the S.D [8] of some computing machine $\mathcal{M}$, then $\mathcal{U}$ will compute the same sequence as $\mathcal{M}$."*
>
> from [98], pages 241 & 242

The UTM therefore describes a machine that expects a sequence of instructions that resemble the same behavior as a machine dedicated to only one purpose and provides the same results. For example a calculator is limited to arithmetic functions and a clock is limited to keep and indicate the current time. An UTM could process a Standard Description (S.D) of a calculator $S.D_{calc}$ to perform arithmetic

[7] first introduced as '*universal computing machine*' in [98]

[8] defined as '*standard description*' of an action table that resembles actions of computing machine in a standardized way

operations while an $S.D_{clock}$ would allow the UTM to behave like a clock. One important fact of the UTM was the mentioned 'tape' which would hold the S.D, the input and the computed output. Von Neumann transferred this idea of a 'tape' by holding the instructions and the data in the same memory. The commands and the data were in this case limited to the same specifications of the memory, like word width and addressing, but allowed a flexible usage of the memory space for both. Further this design limited the access to instructions and data in contrast to the Harvard architecture. This made the program flow in a Von Neumann architecture deterministic and therefore avoided race conditions of signals that were possible in the Harvard architecture. Still, this also meant that loading an instruction and reading a datum would take two cycles instead of one cycle in the Harvard architecture. At that time this was not a big disadvantage in exchange for easier debugging of the software. Further new generations of microprocessors could hold a bigger number of transistors and increase their clock speeds in comparison to their predecessors which made the mentioned disadvantages less noticeable for a certain time period. The 'Intel 8008', as an example, was introduced 5 months after the 'Intel 4004' and added 1,200 transistors on the chip area reaching 3,500 transistors in total and nearly doubled the clock speed from 108 KHz to 200 KHz [100]. This growth continued and was described by Gordon Moore in 1965 [101] which became later known as 'Moore's Law'. Moore's Law stated that the complexity of ICs with minimal costs will double in between 1 to 2 years [9]. In 1978 John W. Backus stated that although programming languages added more features, their principles focused heavily on the 'word-at-a-time' thinking of the Von Neumman architecture [104]. He refers with this term to the connection between the Central Processing Unit (CPU) and the store of instruction and data, which is only a single 'tube' [10]. Words, either data or instructions, in the store need to be transferred from and to the CPU one at a time through the tube. This is why Backus calls this tube the *Von Neumann bottleneck* since most of the words need to be addressed and therefore more addresses are sent through the tube instead of actual data. Another problem for the general-purpose CPU is the so called 'memory wall' [105]. Wulf and McKee coined this term in 1995 that addresses the rate of improved speed of microprocessors and their correspondent off-chip-memory, in this case Dynamic Random-Access Memory (DRAM). The growth of the processor speed is higher than the memory speed which leads to a growing frequency gap between both components. In this case, the used memory becomes the leading performance factor to the system since the CPU is depending on incoming data for tasks that exceed its own resources. While the DRAM fetches the data in a certain amount of cycles

[9] first, only 1 year in [101], corrected to 2 years in [102], later estimated around 18 Months by Dave House, a colleague of Moore [103]

[10] basically meaning a Bus

the higher clocked CPU spends a multiple of this amount idle in the worst case. In order to bridging this gap caches are used. A cache is a high-speed memory located close to the CPU to minimize latency. Caches can be arranged in multiple levels with high clock and small memory space caches close to the CPU and dropping clock rates and higher space caches closer to the main memory. Another approach to alleviate the gap are Three-Dimensional Integrated Circuits (3DICs) which stack multiple silicon wafers on top of another. This allows caches to be stacked on top of a CPU to generate buses that have a higher bandwidth than non-three-dimensional ICs [106]. Further, the Von Neumann architecture can be mixed with the Harvard architecture to create a 'modified Harvard architecture'. The degree of the modifications on the Harvard architecture has a wide range, as a simple example the separation of the instruction and data memory could be weakened to allow modifications on instructions in the run time. This so called 'self-modifying code' is possible in Von Neumann architecture while separated memories like in the Harvard architecture allow parallel access on instructions and data. Still, all approaches and many more not mentioned here only mitigate the influence of the Von Neumann bottleneck on the performance but never solved the problem entirely.

2.2.1.2 Custom Hardware and Application-Specific Integrated Circuits (ASICs)

After the history of ICs in Section 2.2.1 and of the CPUs in Section 2.2.1.1, the history of ASICs is presented before introducing FPGAs in Section 2.2.2.

While the general purpose CPU could emulate any given program walking in the footsteps of the UTM, there is still a need for specialized hardware only built for one application. These Application-Specific Integrated Circuits (ASICs) are hardwired circuits with a special design for a specific task often just shipped to one costumer. The advantages of ASICs are based on their focus on one special task which allows full customization of the circuit. The clock frequency, word width of buses or parallel executions in different branches of the circuit outperform the capabilities of a CPU in most cases, since they are chosen to fit the task. Further the power consumption and/or chip area is lower and instructions are not necessary since ASICs do not need to be an UTM. On the other hand, ASICs have the disadvantage of a high amount of produced units to lower the price per unit since they need long design and testing phases. An example task for an ASIC is the decoding of the MPEG-1 or MPEG-2 Audio Layer III (MP3) [107] audio coding format. While the decoding of an MP3 file can be performed by a CPU the power consumptions is too high which makes ASICs a better choice for mobile devices, like MP3 players or mobile phones, in order to extend the time before the battery runs out of power

and has to be recharged. Since the decoding of MP3 files is a pretty common task an ASIC can be reused and therefore sold to different costumers which makes these ASICs often referred to as Application-Specific Standard Products (ASSPs). Therefore, ASSPs help to lower the costs per unit but the overall design costs stay fixed. The reason for the high design efforts lies in the manufacturing of physical chips instead of just a virtual program for a CPU performing the task. Of course, a mass-produced ASIC with flaws would be a financial disaster for a producer because of the high pre-production investment which is why the design is simulated at different states. An early way of not only simulate but also test a design in real conditions were Programmable Logic Devices (PLDs). Instead of using a fixed set of logic gates to directly resemble a fixed logical function, PLDs are undefined in their logical function at manufacturing. There are different types of PLDs using different techniques. On the one hand, Programmable Logic Arrays (PLAs) use a set of programmable AND gate planes, which link to a set of programmable OR gate planes. 'Programmable' in this case means that connections can be cut inside the AND/OR planes to get the desired output. On the other hand, there are Programmable Array Logics (PALs) and Generic Array Logics (GALs) devices that could be programmed once (PAL) or multiple times (GAL). They use a Programmable read-only memory (PROM) with a number of i input bits (address width of the PROM) and a number of o output bits (word width of the PROM). The PROM could be programmed once [11], also called 'reconfigured', to a simulated circuit with i or less different input signals and o or less different output signals. Since the output is dependent on the address, an address change results in the same output change just as the intended circuit would produce. Therefore, a PLD can be reconfigured to test certain aspects of the desired ASIC design.

2.2.2 Field-Programmable Gate Arrays (FPGAs)

After the historical background of the FPGA in Section 2.2.1, the technical structure of an FPGA is explained before the programming of this hardware is introduced in Section 2.2.3.

The Field-Programmable Gate Arrays (FPGAs) originated from the PLDs. While the different PLDs types (see Section 2.2.1.2) supported smaller circuits, an FPGA should handle complex circuits in the last step of the design of an ASIC. The 'Field' part in the names references to 'in the field' programming of the FPGA, meaning that the customer could program or reprogram the PLD. For other PLDs the manufacturer often produced a certain amount of chips and configured them before

[11] later different types, like Erasable Programmable Read-Only Memory (EPROM) or Electrically Erasable Programmable Read-Only Memory (EEPROM), allowed multiple reconfigurations

shipment to the costumer, which made these tailored to the needs of the costumer but limited the possibility of the costumer to reconfigure them (depending on the PLD type). Since Moore's Law [101] applies to all ICs, the possible complexity of circuits on PLDs was and is rising. Therefore FPGAs and ASICs slowly became competitors in the sense that the design focus shifted on the execution on an FPGA rather than producing an ASIC. Reasons for this shift are smaller businesses that could not effort the high costs of ASICs design and/or did not need the high production rate. FPGAs allowed custom solutions with a small number of devices that could also be reused. The circuit designs on FPGAs could handle much more variety in the performed tasks. Further, no longer produced ASICs could be replaced by FPGAs with the equivalent configuration which supports systems that otherwise would not have any spare parts left. Still, ASICs are better in the fields of area, delay and power consumption in comparison to FPGAs, but the gap is slowly closing [108].

2.2.2.1 Architecture

This section provides an overview how the FPGA is integrated into the board design before the resources are highlighted in Section 2.2.2.2, 2.2.2.3 and 2.2.2.4. Especially, Section 2.2.2.3 introduces the connector to the components on the board.

CPUs, ASICs and FPGAs are all ICs. These ICs are normally connected to a Printed Circuit Board (PCB) which holds the different components in place and connects them with conductive tracks. The most common PCB is perhaps the 'motherboard' which connects one or multiple CPUs with other peripherals in the home Personal Computer (PC) market. These PCBs are designed in such a way that components can be exchanged to some degree in order to support a custom configuration of the whole system. Especially the CPU socket allows the exchange of the CPU, while peripheral card slots, such as Peripheral Component Interconnect Express (PCIe), allow the exchange of different peripherals without soldering the components onto the PCB. Therefore, CPUs have a Pin Grid Array (PGA) on one side of the chip that allows connectors to fit into holes of the PCB closing the connection. On the other hand, ASICs are commonly directly soldered to the PCB to build a system that can not be altered without a certain skill and even if the ASIC is removed the new ASIC needs the same pins and must behave in the same fashion the old one did. FPGAs are in between these two archetypes, even though sometimes CPUs are directly soldered to the PCB or ASICs are changeable, the variety seems greater for FPGAs. Starting from the CPU side, there are plans from Intel [95] that include a CPU and an FPGA on

the same IC [109] [110]. The advantage of this architecture is the direct access to the caches and exchange between the CPU and the FPGA. Further the FPGA has the same connections to other peripherals connected to the motherboard just like a CPU. Since these CPU/FPGA-ICs are currently in development the most common architecture are FPGAs on their own PCB. These PCBs have many different manufacturers which make a general description of the capabilities of a PCB next to impossible. Important in this case is perhaps the presence of a PCIe connector on the PCB to divide the PCBs into two groups. The PCBs with a PCIe connector allow the communication between the CPU and the FPGA by exchanging data in the main memory. This method is of course slower than the direct access to the cache but allows a direct addition or exchange of an FPGA board for a PC. Just like an addition or exchange of a Graphics Processing Unit (GPU) board would add more processing power for graphical computations on a PC. The group of PCBs without a PCIe connector can still communicated with PCs through other connectors, like the Universal Serial Bus (USB) connector, but their purpose is more like a standalone system then a support card for the PC.

In this work all FPGA related experiments are executed on a `DNPCIe_10G_HXT_LL` [111] board manufactured by the Dini Group [112]. The used IC is a Virtex-6 FPGA (`XC6VHX380T-2FF1923` [113] to be exact) manufactured by Xilinx [114]. The Figure 2.10 depicts the front of the `DNPCIe_10G_HXT_LL` board without the cooler unit.

On the right half of the board in the middle is the FPGA. In this case a `XC6VHX565T` [113] is pictured instead of the `XC6VHX380T` but the overall board design stays the same. At the bottom is the PCIe connector. The board can be plugged into a PCIe 16 lane slot, but as it can be seen there are only pins on the left half. Therefore the board supports a PCIe connection with 8 lanes of the PCIe 2.0 Standard [116].

2.2.2.2 Logic Blocks

> *This section introduces the smallest unit reconfigurable on the FPGA which is not only important as connector and controller of Hard Blocks (see Section 2.2.2.3) and clocking (see Section 2.2.2.4) but the design and programming of circuits also depends highly on them (see Section 2.2.3)*

Since the name FPGA refers to an array there are elements in this IC that are ordered as an array. Logic blocks are the most common kind of these elements even so there are different namings for them. The vendor Xilinx [114] for example calls them Configurable Logic Blocks (CLBs), while the vendor Altera [117] calls them Logic Array Blocks (LABs) [118]. Since in this work a Xilinx IC is used,

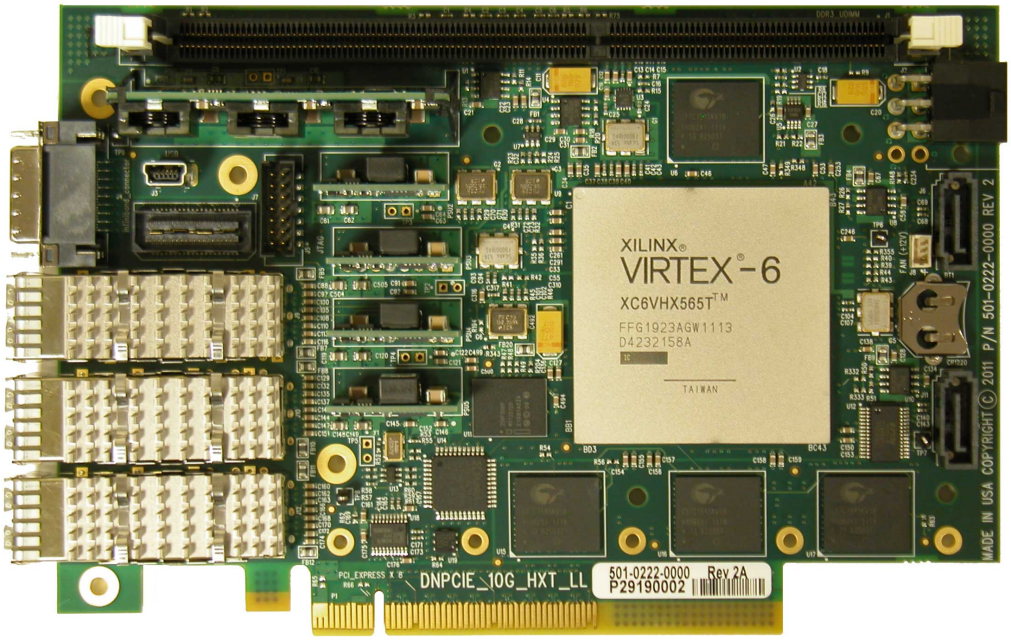


Figure 2.10: Front picture of the DNPCIe_10G_HXT_LL board from [115]

the namings of Xilinx are used, but the namings of Altera are given in brackets. The purpose of these CLBs is the containment of different smaller logical cells, while the block itself is surrounded by routing channels. These routing cells allow the connection between different CLBs or other blocks which makes more complex circuits possible. The smaller logic cells inside a CLB are called 'slices' (Adaptive Logic Modules (ALMs) in LABs [118]) and contain further logic. Typical logic in a CLB are Lookup Tables (LUTs), Full Adder (FA) and Flip-Flops (FFs). LUTs have a similar function like the mentioned PROM in Section 2.2.1.2. For any given input address a LUT provides the desired output like a fixed logic function would (AND, OR, NOR, etc.). FA can add two bits while a carry in and carry out can transfer a carry over from one FA to another. FFs can store state information temporarily.

In Figure 2.11 one CLB is shown.

The CLBs of the Virtex-6 Series all consist of two slices [119]. The CLB in this example is called CLBLM_X25Y312 and contains the two slices SLICE_X40Y312 and SLICE_X41Y312 (represented by squares). Around and between CLBs are blue lines which represent the routing channels. These channels have Programmable Interconnect Points (PIPs) at the corners which can be reconfigured to lead a sig-

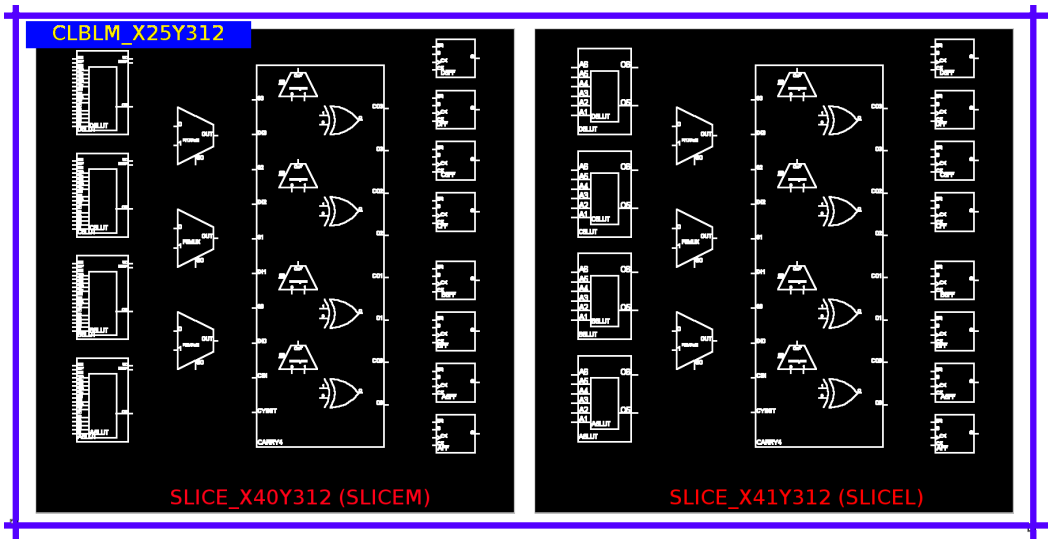


Figure 2.11: Schematic of a CLB with two slices

nal to another block. Inside one slice are four LUTs on the outer left side. These LUTs have two modes. Either one function with six input bits and one output bit can be implemented or two functions with five input bits each and two independent output bits. Right of the LUTs are three Multiplexers (MUXs) that can select different input signals and forward the selected input into an output signal. Further to the right is a carry chain which is a variation of a Carry-Lookahead Adders (CLAs) [120] that uses shared logic to lower the transistor count. In this case the CLA consists of four FAs and logic to transfer the carry over. On the outer right side of a slice are eight FFs that can store the result bits from the other components of the slice. In this case the example shows a slice of the type `SLICEL` which only support combinatorial functions on the right. The other type of slices on the left is a `SLICEM`, which allows the use of the LUTs as either distributed 64-bit RAM or 32-bit shift registers. In total there are 29,880 CLBs on the `XC6VHX380T` which makes 59,760 slices. 18,240 of these slices can be used as distributed RAM.

2.2.2.3 Hard Blocks

This section introduces Hard Blocks that can be integrated into the user-specific circuit by CLBs (see Section 2.2.2.2) if the architecture of the board supports it (see Section 2.2.2.1)

Although, the CLBs are built to resemble any circuit in theory a practical rebuild of a memory or processor would waste a lot of CLBs while the desired application runs out of chip area. Therefore FPGAs do not just consist of CLBs but also have other types of blocks, so called Interlectual Property (IP) cores. As the name already states are these cores not necessary from the IC vendor or the board vendor but the designs of these cores belong to other parties. Since many functions are standardized the own design of a function can cost more than the licensing fees for an already existing design. Further the own designs can give some extra earnings by licensing the use to other parties. There are two types of IP cores, the soft cores and the hard cores. Soft cores provide a greater flexibility in design, allowing multiple configurations, then hard cores, which can be limited to the connections and remain a black box otherwise. The reason for the limited configuration possibilities is the direct building out of transistors instead of LUTs which give these cores a similar performance and power consumption as ASICs. Both core types can be directly integrated into the chip area or just redirect the signals over connectors to the IC on the board.

For the used Virtex-6 FPGA in this work there are many IP cores. One of the smallest is the Block RAM (BRAM) Block which is a small memory directly integrated into the chip area. In Figure 2.12 a BRAM surrounded by multiple CLBs is shown.

The BRAM of the Virtex-6 can store 36Kbit of data. With the height of five slices and a width smaller than a CLB the BRAM utilizes the chip area more efficient in case of storage capacity. Five CLBs can only provide 320bit against the 36,000bit of the BRAM. The BRAMs of the FPGA are arranged into multiple columns on the chip and can be separated into two 18Kbit independent blocks [121]. Furthermore, they are capable of true dual-port access by providing dedicated signal ports for data, address and clocking. In order to minimize the amount of used CLBs, the BRAMs contain dedicated logic to resemble a First-In First-Out queue (FIFO). Additionally, the BRAMs have logic to connect multiple BRAMs to a seemingly bigger memory. This makes the BRAMs the choice for middle ranged buffers or FIFOs that would otherwise require an excessively amount of `SLICEM` type slices.

Other IP cores can be found off-chip. In Figure 2.13 a block diagram of the different components of the board can be seen.

On the right side of the figure are four Quad Data Rate (QDR) Static Random-Access Memorys (SRAMs). While two memories are independent, the other two are connected to constitute a bigger memory than the two single ones. The four memory chips can also be seen in the Figure 2.10 where one QDR chip is located directly above and the other three directly below the FPGA. One memory has a capacity of 144Mb which makes the two connected memories a storage with 288Mb

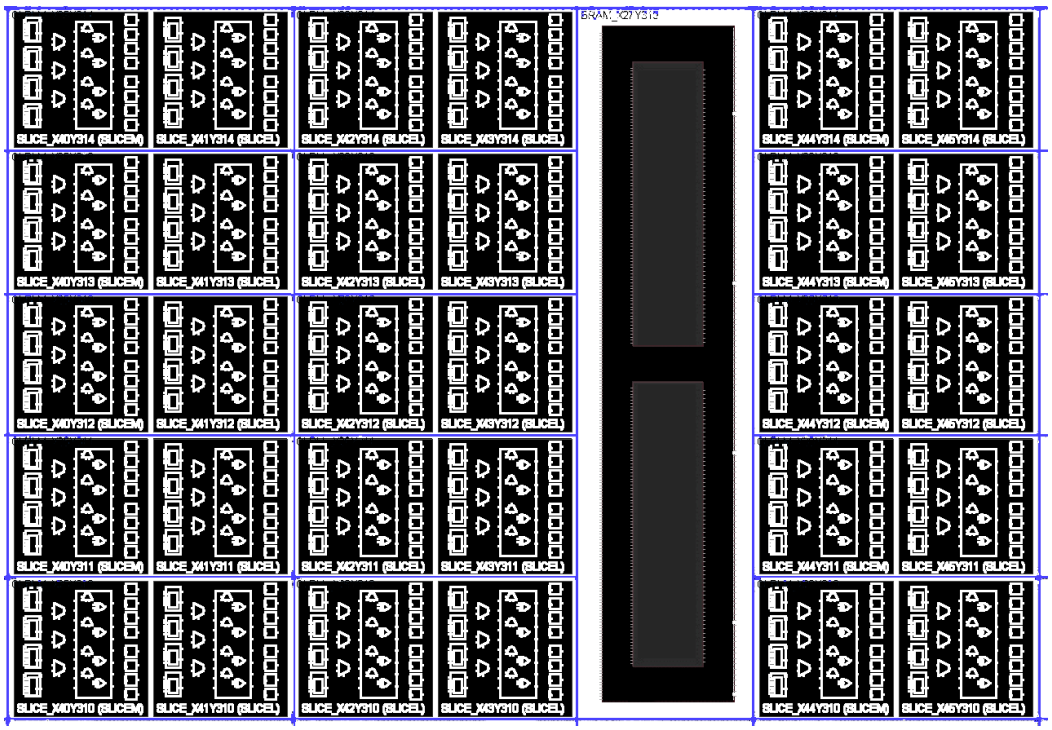


Figure 2.12: Schematic of a BRAM surrounded by multiple slices

capacity. [123] These components represent the next level in the memory hierarchy slower than the distributed memory or BRAM but with a higher capacity. Another memory level is the Double Data Rate (DDR) Dual Inline Memory Module (DIMM) connector that allows the extension of the board with DDR DIMMs. The connector can also be seen in Figure 2.10 at the top edge of the board without a module inserted. It supports modules with up to 16GB storage capacity of the DDR3-Standard. This gives the FPGA the opportunity to work with a memory size similar to the main memory of a PC, but also with the same disadvantages, like high latency compared to QDR or BRAM. The last level in the memory hierarchy are the two Serial AT Attachment II (SATA-II) connectors that allow the connection of Hard Disk Drives (HDDs) or Solid State Drives (SSDs). In Figure 2.10 these connectors are on the right edge of the board. Again, this level is slower than the levels before but can support a much higher storage capacity.

After these memory components there are also many different components for communication. The board has three 10 Gigabit Ethernet Local Area Network (LAN)/Wide Area Network (WAN) connections consisting of Enhanced Small Form-Factor Pluggable (SFP+) [124] transceivers. Further, there is a CX4 socket

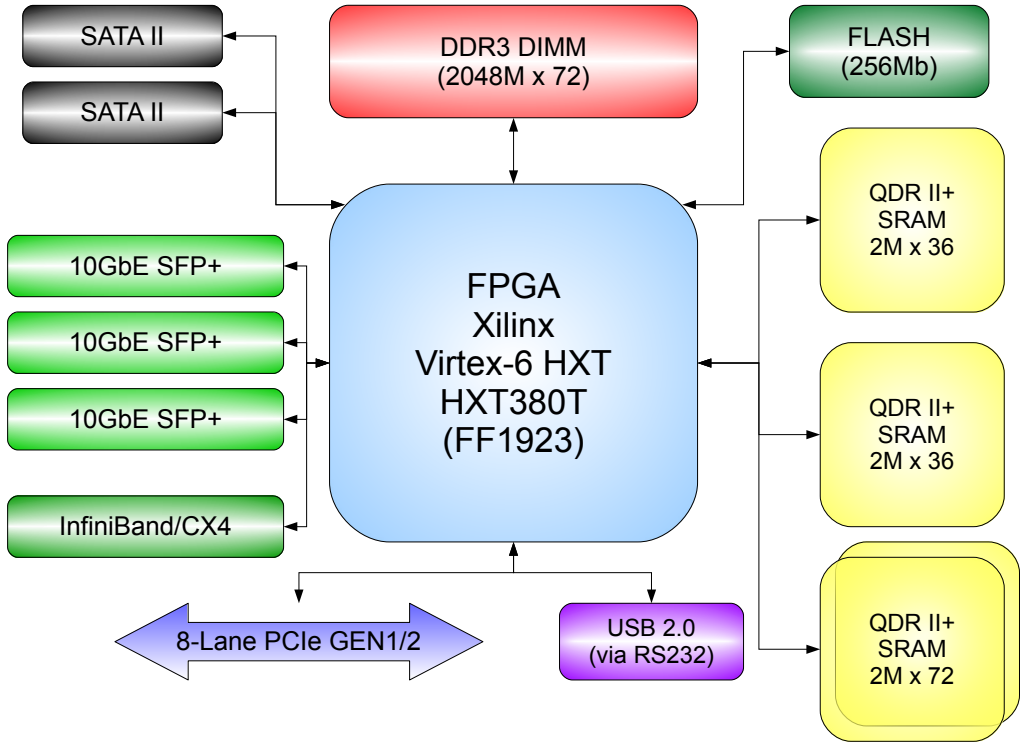


Figure 2.13: Block diagram of the `DNPCIe_10G_HXT_LL` board (inspired from [122])

for low latency InfiniBand networking. As the board can be plugged into a PCIe slot, the PCIe connector supports 8 Lanes using the PCIe 2.0 standard. The last connector is the USB 2.0 connector which is the slowest connection to and from the FPGA.

Also, as a special feature, the board has an additional flash memory that can store a general instantiation of a configuration. After powering up the board an automatically reconfiguration is performed on the FPGA. This way the FPGA can restore its configuration after a power shortage.

2.2.2.4 Clocking

This section explains the clocking of an FPGA which can handle multiple frequencies for different blocks (see Section 2.2.2.2 and 2.2.2.3). Further, this possibility has impact on the design and programming in Section 2.2.3.

The handling of clock rates of an FPGA is different than the clocking of a CPU. Single-core processors have just one CPUs with one variable clock rate up to a certain maximum [12]. The CPUs of a multi-core processor can have different clock rates, but are limited to one rate each. FPGAs on the other hand have multiple clock regions that allow a wide variety of different clocking rates. Even though these clock rates are much lower than the clock rates of a CPU the intended pipelining and/or parallelism of an FPGA can outperform the CPU. Furthermore, a shorter duration for a cycle leads to a direct performance boost for the FPGA circuit while a multi-core processor would 'lose' a fraction of the performance due to the scheduling of the tasks to the different cores.

The chip area of a Virtex-6 series FPGA can be split into 6 to 18 so called 'clock regions' [125]. In the case of the model `XC6VHX380T` that is used in this work there are 18 clock regions separated into two horizontal columns with 9 regions each on the chip area. There is a slim *center bank* between the two clock region columns where the 9 Clock Management Tiles (CMTs) are located next to other dedicated configuration pins. These CMTs have two Mixed-Mode Clock Managers (MMCMs) that allow adjusting the frequency of the cycle between 10 and 800 MHz. Each clock region has a height of 40 CLBs and is divided horizontal by a Horizontal Clock Row (HROW). These HROWs go through the western clock region column over the center bank with a corresponding CMT to the eastern clock region column. This means that above and below a HROW are always 20 CLBs. Each clock region has further clocking logic and routing resources. Further, there is the possibility to use regular routing resources if the dedicated routing resources are used up. However, this possibility can not guarantee a certain frequency as these resources are not optimized for clocking. A user-specific IC can have multiple so called *clocking trees* starting at the central bank at a CMT. The clocking from the CMT can be adjusted by the MMCMs and reach the corresponding clock region over the HROW. Further, local adjustments in the clock region can be made which leads to further twigs in the tree each with an own possible frequency.

2.2.3 Design and Programming

> *After the historical background (see Section 2.2.1) and the architecture of the FPGA (see Section 2.2.2) the next steps is the design and programming of a user-specific circuit onto an FPGA.*

The design and programming of a application-specific circuit for an FPGA differs very much to a program designed for a CPU. On the other hand is the design

[12] Exceeding the maximum is possible, but can lead to irreparable damage due the higher power consumption and resulting heat development.

process very similar to the development of an ASIC. This is no coincidence since the connection between ASICs and FPGAs is made clear in Section 2.2.1.2 where FPGAs can help testing ASIC designs before these are finalized. Since there are multiple stages in the development of circuits for an FPGA, vendors tend to support developers with tools similar to Integrated Development Environments (IDEs) for software development. These Electronic Design Automation (EDA) tools support the different development stages in different ways and to different extents. Examples for EDA tools are Intel Quartus Prime [126] of Intel [117] [13], ModelSim [127] as one of many tools of Mentor Graphics [128], Vivado [129] and Integrated Synthesis Environment (ISE) Design Suite [130] of Xilinx [114]. Some of the examples combine a variety of tools for all design steps while others concentrate on some or just one step in the design. For example, Sigasi Studio [131] from Sigasi [132] focuses on the Hardware Description Language (HDL) design but has no features that could apply the written code onto a specific FPGA. The compatibility between the EDA tools depends on the companies providing the tools. As an example the ISE Design Suit allows the execution of third-party tools for certain design steps, like Sigasi Studio for HDL design and ModelSim for simulation, even though own solutions are available. Still, the last design steps for applying the application-specific circuit onto the FPGA are limited to the vendors own EDA tools, limiting for example the Vivado and ISE Design Suit to Xilinx FPGAs or Intel Quartus Prime to Intel FPGAs.

Independent from the used EDA tools the design steps stay the same and a design flow is shown in Figure 2.14.

At first the concept of the application-specific circuit is developed in a top-down manner. This process starts with a top level module (TOP) and divides it into different sub-modules that perform different sub-tasks of the desired design. These sub-modules can also be divided until no further sub-tasks are needed. The implementation and verification follows a bottom-up strategy and starts with the modules on the lowest hierarchy level. The behavior of these modules are described in an HDL and evaluated in a so called 'testbench' in different simulation scenarios. If the modules works as expected the next level is implemented which should avoid errors or malfunctions in the lower modules. At some point the TOP module is reached, implemented and tested. The next step is the translation of the whole design into an application-specific circuit on the FPGA. Simulation happened so far only on the Register Transfer Level (RTL) which is a design abstraction for the flow of digital signals between registers and logical operators. The synthesis connects the desired behavior on the RTL with resources available on FPGAs and generates a netlist. This netlist will be implemented specifically for the used FPGA. There-

[13] formerly Altera which was acquired by Intel in June 1, 2015.

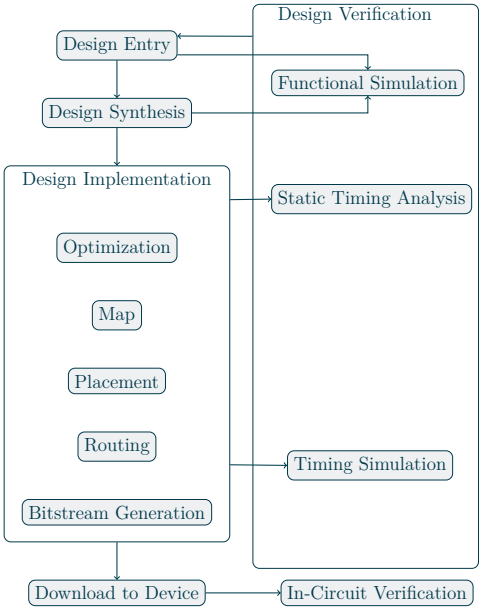


Figure 2.14: Standard design flow for FPGA development inspired by Xilinx User Guide [133]

fore, the needed resources are mapped to actual instances of the FPGA. These are then placed within the chip area and the signal path are routed between them. At the end a binary file [14] is generated that contains all information to configure the FPGA with the desired design. Further, with every step in the design flow the design verification gets more accurate since more implementation details are available.

In the following, the focus is on the ISE Design Suit from Xilinx since in the evaluations of this work a Virtex-6 from Xilinx is used. Still, a general view on the design process is given.

2.2.3.1 Hardware Description Languages (HDLs)

This section introduces the main language set to reconfigure FPGAs and therefore is important for the synthesis (see Section 2.2.3.2). Further, the Section 2.2.3.6 introduces possible ways to translate a programming language into an HDL.

[14] called bitstream by Xilinx

An HDL describes the structure and behavior of electronic circuits in comparison to a programming language that specifies a set of instructions that can be executed in a sequence leading to a certain output. Still, both language types look pretty similar as they are textual descriptions consisting of expressions and statements. The first HDLs were introduced at end of the 1960s [134]. Before most ICs were only described by schematic diagrams that focused more on the structural and less on the behavioral aspect of the design. A first milestone in the history of HDLs was the introduction of the Register Transfer Level (RTL) notation in 1971 by Barbacci [134] and its first use in the Instruction-Set Processor (ISP) descriptive system [135]. The RTL allowed to depict the flow of signals between registers and logical operators. An RTL example can be seen in Figure 2.15.

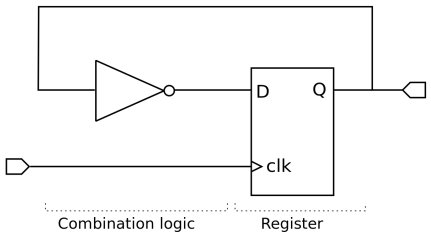


Figure 2.15: A circuit presented in RTL that toggles the output signal (from [136])

In the example only a D-Flip-Flop (FF) is used as an implementation of the register group and an inverter is the only logical operator. The output signal `Q` toggles every time the `clk` signal changes since the input `D` of the FF is always the opposite of `Q`.

Even though the RTL was well accepted the used HDLs were pretty diverse and vendor-specific. A change came at the beginning of the 80s with the Very High Speed Integrated Circuit (VHSIC) program of the United States (US) Department of Defense (DoD). At this time the DoD had the problem that the development cost for electric circuits were very high and still rising. The reason for this were the many different suppliers that used different EDA tools, methods and HDLs which were sometimes only vendor specific or incompatible at all. The target of the VHSIC program was a language that could describe and/or document the desired hardware and further 'simulate' this description for testing purposes. An automatic translation from an HDL to a finished design on an IC was not planned at this point. The VHSIC program had much in common with the efforts of the High Order Language Working Group (HOLWG) formed by the DoD. The HOLWGs goal was a programming language for software development that would supersede the many

different programming languages used at that time. Since the HOLWG started its work in 1975, therefore before the VHSIC program, the resulting programming language Ada could be used as a reference for the planned HDL design [137]. This is the reason why Ada and the resulting VHSIC Hardware Description Language (VHDL) have much in common. At the end of the 80s companies started to develop so called 'synthesis' tools that would allow the translation from RTL to logic gate level. These tools could only support a subset of the original HDL constructs since the only purpose of some constructs are more accurate simulation and can not be mapped on physical components. Thus, with these tools an automated flow from an HDL to a finished design on an IC was now possible.

Another well-known HDL is Verifying Logic (Verilog) [138]. Verilog was developed between 1983 and 1984 [139] as a new HDL for the planned simulator 'Verilog-XL'. This also meant that Verilog was a proprietary HDL and bound to the 'Verilog-XL' simulator as de facto standard simulator for Verilog. Still, Verilog was successful, but VHDL increased its market share slowly due to its open standard. Therefore, Verilog became available for open standardization resulting into the Institute of Electrical and Electronics Engineers (IEEE) Standard 1364-1995 [140] in 1995. Verilog shares similarities in syntax with the programming language C and the control flow keywords (if/else, for, while, case, etc.) are equivalent. Like VHDL, Verilog was designed for simulation and therefore only a subset of statements is synthesizable.

There are many former or newer HDLs on the market, still VHDL and Verilog are the main competitors at the moment. EDA tool vendors reacted to this and usually support both languages in their tools. Therefore, a design consisting of a mixture of VHDL and Verilog files is possible which both are translated into netlists. In Listing 2.16 and Listing 2.17 two code examples in VHDL and Verilog are given which resemble the circuit of Figure 2.15 that toggles its output signal Q and is therefore called 'toggler' in the following.

In both languages the code can be split into two parts for the toggler. The first part is the **entity** which describes the incoming and outgoing signals of the component. In the case of the toggler the `clk` is an input and the `Q` is an output signal. VHDL separates this part (lines 1 to 6) strongly from the **architecture** part which directly describes the behavior of the component (lines 8 to 21). In Verilog the behavior is encapsulated in the module (lines 5 to 10) and the entity is described in the upper part (lines 1 to 3) but not directly separated. The VHDL examples shows the direct naming of entities, architectures and processes in this case *myToggler*, *myArchitecture* and *toggle* while in Verilog only the module is named. The naming of the architectures especially enables multiple architectures for the same entity which can be chosen through inclusion from higher modules. An important

Listing 2.16: Toggler example (VHDL).

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity myToggler is
5    Port (
6      clk : in  std_logic;
7      Q   : inout std_logic
8    );
9  end myToggler;
10
11 architecture myArchitecture of myToggler is
12
13 signal D : std_logic;
14
15 begin
16   D <= not Q;
17
18   toggle:  process(clk)
19   begin
20     if rising_edge(clk)  then
21       Q <= D;
22     end if;
23   end process toggle;
24 end myArchitecture;
```

Listing 2.17: Toggler example (Verilog).

```
1  module myToggler(clk, Q);
2    input clk;
3    output reg Q;
4
5    wire D = !Q;
6
7    always @ (posedge clk)
8      begin
9          Q <= D;
10     end
11 endmodule
```

construct in the two HDLs are processes introduced by the keyword **process** in VHDL (line 15) and **always @** in Verilog (line 7). Both keywords are followed by a so called *sensitivity list* that consists of signals the process reacts to (in this case the signal `clk`). There are two types of processes. Either the sensitivity list does not contain a clock signal which leads to a '*combinational process*' or the sensitivity list consists of a clock signal and sometimes a reset signal leading to a *'clocked process'*. In the case of the examples the processes are clocked, especially indicated through functions that react to the rising edge [15] of the clock signal (VHDL, line 17: `rising_edge()`, Verilog, line 9: `posedge`). This guarantees that the statements inside the process, in this case the adoption from the value of the signal `D` as the value of signal `Q` (`Q<=D`), are executed once each cycle. The processes could be converted into combinational processes by removing `clk` and adding `D` to the sensitivity lists. This would lead to a loop since the signals `Q` and `D` affect each other (VHDL lines 13 and 18, Verilog lines 5 and 9) and the `clk` signal is not considered anymore. Changes of the signal values would happen multiple times inside the same cycle which makes the prediction of the value at the end of a cycle harder since the time for a signal depends highly on the actual placement inside the IC. In case of this scenario a combinational process does not make sense but both processes types are important depending on the desired task. A difference between both HDLs is the explicit typing in VHDL (lines 3, 4 and 10) while Verilog renounces direct typing. In Verilog the signals have a value range of `0` and `1`, a `Z` for high-impedance or `X` for an unknown value. The `std_logic` type in VHDL shares these values and adds `U` for uninitialized values in simulation, `W` for weak signals that can either be `0` or `1`, `L` for low signals close to `0`, `H` for high signals close to `1` and – when the value of the signal does not matter. Both languages can combine these single data types to a vector to constitute registers or a set of related signals. Verilog just adds a range to a signal while VHDL has the own data type `std_logic_vector`. This again shows the importance of data types in VHDL which has further types, like `Integer`, `Character` or `String` and even allows the creation of individual data types. Still, the resulting synthesised netlist of code based either on VHDL or Verilog is not distinguishable since the data types are mapped to the same resources at gate level.

In the following, VHDL is used as the main HDL since the circuits design is written in VHDL and only modules belonging to IPs are provided in Verilog.

2.2.3.2 Logic Synthesis

[15] when the signal changes from a low value ('`0`' or '`L`') to a high value ('`1`' or '`H`').

After the introduction of the HDLs in the last Section 2.2.3.1, the logic synthesis is presented in this section. The synthesis is important to define the components used in the user-specific circuit before they can be placed and routed in the next Section 2.2.3.3.

If the desired circuits design is described in an HDL the next step is the logic synthesis. This step is further divided into multiple steps [16]:

Parsing The given HDL code is parsed which means the syntax and hierarchy of the design is analyzed. Typing and grammatical errors in the code are highlighted depending on the used HDL. Signals with multiple drivers are highlighted. Since this step is also performed for the generation of a simulation model, non-synthesiseable code persists.

Elaboration This step translates the given code to generic hardware resources. The process of this translation is called 'inference'. The detected modules/units are inferred into two types. Primitives are described by using common elements that are expected to be available on the FPGA, like inverters or multiplexers. The concrete instantiation on the IC is still left out, for example an FF is detected but not the type, like a D-FF. This makes the resulting description of the elaboration technology independent. The second type are macros that basically describe black boxes since the description is too complex for primitives. At a later point the used IC either has a direct counterpart in hardware for the macro or needs an own circuit design with the resources available. This guarantees again the independence of the fundamental technology at this point. Further, sequential logic is detected. These are often hardware implementations of Finite State Machines (FSMs). They are also recognized as macros.

HDL Synthesis The technology independent description of the elaboration is mapped to the actual resources available on the used FPGA. Thus, all resulting implementations are technology-specific to the used device. The used tool attempts to recognize 'basic macros' from the inferred

[16] The steps follow the designs steps of the Xilinx Synthesis Technology (XST) tool [141]. The synthesis procedure of other vendors is similar but steps may differ.

macros that represent registers, adders or multiplexers. Further, the detected FSMs are inferred to device-specific sequential logic.

Advanced Synthesis The basic macros are combined to macro blocks, which can represent counters, pipelined multipliers or multiply-accumulate functions. Further, the encoding scheme for each inferred FSM is chosen.

Low Level Synthesis The macros that are so far not handled get an implementation based on the available timing information of the other macros leading to more contextual decisions. Especially FFs are either removed because they are redundant or optimized if they are constant. Further, registers are replicated if they are needed in different areas of the design allowing better timings.

As an example, we synthesize the code of Listing 2.16 and Listing 2.17. The resulting RTL schematic can be seen in Figure 2.16.

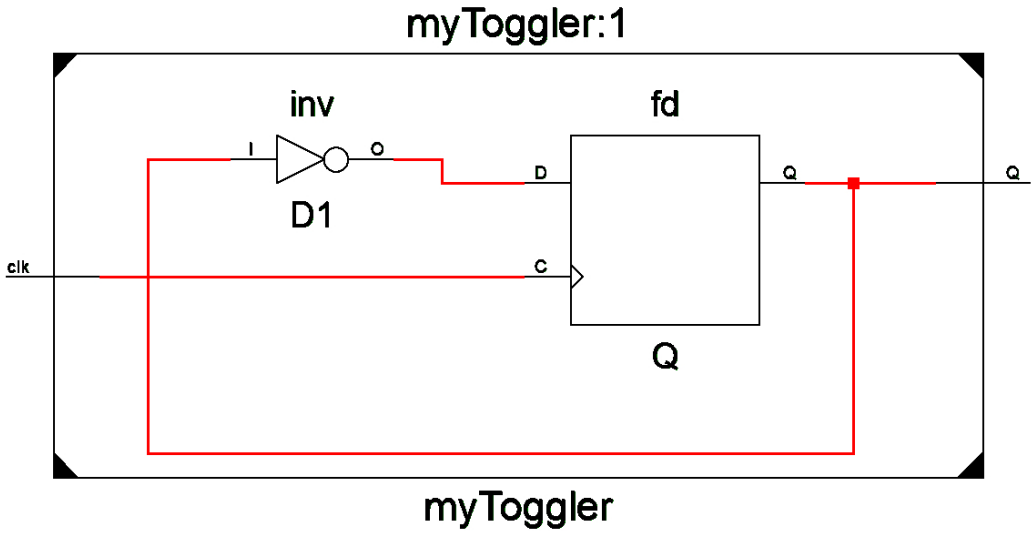


Figure 2.16: The resulting RTL schematic after synthesizing the VHDL/Verilog code of Listing 2.16/Listing 2.17

There are certain interesting points in this example. First, the VHDL and Verilog code both produce the same RTL schematic. This must not be the case since in more complex designs the different language constructs of each language can be hard to match by the programmer. But this is not necessary, since normally

the behavior is only described in one language, the behavior should stay the same even if the used components differ and the used synthesis tool can also take an effect on the whole design. Second, the resulting Figure 2.16 shows the same schematic as the schematic in Figure 2.15. This is also owed to the simplicity of the example. There is no guarantee that an intended behavior described in an HDL leads to a certain RTL schematic, especially in more complex designs. Sometimes the described behavior must be rewritten since the synthesis tool interpreted the instructions as a suboptimal design.

The RTL schematic only depicts a technology independent view on the design. Therefore, the Xilinx Synthesis Technology (XST) tool also provides a technology schematic view as seen in Figure 2.17.

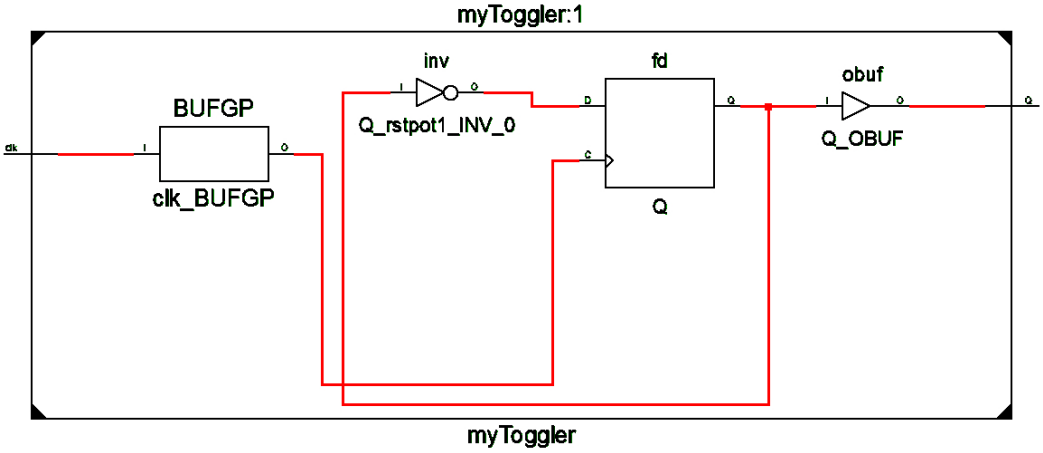


Figure 2.17: The resulting technology schematic after synthesizing the VHDL/Verilog code of Listing 2.16/Listing 2.17

Again, the changes are minor and the D-FF and the inverter stay the same. The only additions are the buffers on the clock signal. The Primary Global Buffer for Driving Clocks (BUFGP) ([142], page 78) connects the pin of a clock signal with a CLB while a Output Buffer (OBUF) ([142], page 438) does the same the other way around.

2.2.3.3 Implementation of the logical design

After the synthesis (see Section 2.2.3.2) the user-specific circuit needs to be placed and routed on the chip area of the FPGA. This step is further needed for the generation of a programming file in Section 2.2.3.4.

The synthesis results in a netlist of the desired behavior. Still, the needed resources are not assigned to the resources on the chip area of the FPGA. Therefore, multiple steps have to be passed through in order to achieve this goal.

Translation: The resulting file format of the synthesis is the Electronic Data Interchange Format (EDIF) [143]. This format was chosen as a vendor-neutral format that can be used for EDA purpose. Different vendor tools can translate this neutral format into their own proprietary format. For example, the NGDBuild [133] tool of Xilinx translates EDIF files into Xilinx Native Generic Database (NGD) files which is a format only suitable for Xilinx FPGAs and its tools. Important is also the User Constraint File (UCF) format that is an American Standard Code for Information Interchange (ASCII) file. It contains timing and layout constraints from the user that affect how the logical design is implemented in the target device. These constrains are also added to the NGD file.

Mapping: After a vendor-specific format is created, the resources on the FPGA can be assigned accordingly. In case of the Xilinx tool flow, the MAP tool fulfills this task. Special macros, described by the Xilinx Native Macro Library (NMC) format, are predefined circuits best suited for the physical resources of the FPGA. Other logic is mapped to Xilinx components (Input/Output Blocks (IOBs), CLBs, etc.). Further, logic that is not used in the design is removed. The MAP tool generates a Native Circuit Description (NCD) file and a Physical Constraints File (PCF).

Placement and Routing: The mapping to the physical resource is not necessary optimal especially if timing constrains of the user are not met. The PAR tool of Xilinx takes the NCD and replaces components and routes new connections between them. With this, the PAR tool tries to meet the requirements in the PCF. It can be that no requirements were given and therefore the PCF is empty. In this case, the PAR tool switches from its timing-driven mode into a cost-based mode. In this mode the PAR tool evaluates the connections length on a weight model rather than meeting a certain timing. The output is a routed NCD file instead of a mapped NCD file. However, it is not guaranteed that a certain timing can be meet and finding a possible optimal solution can take quiet a while because of the many combinations in placing and routing resources.

2.2.3.4 Bitstream Generation and Configuration

The last step for a user-specific circuit on an FPGA is the generation of a programming file. It is the result of all former steps (see Section 2.2.3.1 to 2.2.3.3).

The generation of a Bitstream (BIT) file to configure the FPGA with the user-specific circuit is the last step in the design process. The Bitstream Generator (BitGen) tool of Xilinx takes the mapped, placed and routed NCD file and generates a BIT file. This file can be downloaded into the memory of the FPGA and applied to the resources on the chip area. The user-specific circuit on the FPGA is then ready but it is not persistent. Shutting down the host system or a power shortage will reset the FPGAs configuration. Therefore, the BIT file can be used to generate a PROM file. As the name already states, this file can be uploaded into persistent memory that is accessed by the FPGA on start-up.

2.2.3.5 Dynamic Partial Reconfiguration (DPR)

This section introduces the DPR which allows reconfiguring parts of the circuit at run time. Therefore, several changes have to be taken into account for all former steps (see Section 2.2.3.1 to 2.2.3.4).

The so far described design process only leads to one user-specific circuit on the FPGA. In order to apply another user-specific circuit the FPGA would be reconfigured entirely. This can be acceptable if the downtime of the FPGA is not critical. Still, some parts of the circuit stay the same, for example when all communicate by the PCIe connection. A reconfiguration of the components belonging to the PCIe connector is therefore avoidable. The DPR makes it possible to reconfigure only a small part of the FPGAs chip area. Therefore, the design is split into two parts. The static, main part of the design that is not reconfigured and at least one or more Reconfigurable Partitions (RPs) [144]. There can be one or multiple Reconfigurable Modules (RMs) for each of these RPs. The static part of the circuit has its own BIT file generation incorporating all RPs. Since it is not clear which RM is loaded in one of the RPs these areas are handled as 'black boxes'. This means that the configuration of the RPs chip area is left out and only an interface (inputs, outputs) is provided. The interface must apply to all RMs of one RP otherwise the static part and the RM cannot be connected properly. Since each RPs is handled as a black box each RM needs its own BIT file which is synthesized independently of the static part. First, the static design is loaded to the FPGA and then the RMs can be reconfigured while the rest of the circuit is unaffected. This allows the independent, uninterrupted calculation in one part of the chip area while input data for the RP can be buffered until the reconfiguration is done. Although DPR means an always at least partially functional design is possible, the DPR has also some disadvantages. The design of the user-specific circuit cannot optimize over the borders of the RPs. This seems obvious for the static part since a black box cannot be optimized in any way. But also the RMs are stuck to the resources

present in the RP. This also means that one RM which has a small amount of needed resources and a RM with a dense design and a hugh need for resources are not suited for the same RP. If the area of the RP is chosen too small, only the first RM can fit into the area while the second RM is impossible due to the lack of resources in the area. On the other hand a bigger area can support both RMs but the smaller RM uses only a small amount of the available resources that could otherwise be used by the static part of the design but are left unused instead. Therefore, DPR is a powerful optimization technique but the optimal use is hard to achieve.

2.2.3.6 High Level Synthesis (HLS)

This section introduces the HLS which is a step that leads to an HDL description (see Section 2.2.3.1) of a former programming language description. Alternatively, it skips the HDL part and directly jumps into synthesis (see Section 2.2.3.2).

The HDLs presented in Section 2.2.3.1 have a different look and feel than programming languages a software programmer is used to. Therefore, a longer training period is necessary to understand the differences between the design of a software application and a user-specific circuit. The HLS shortens this trainings period or at least gives an easier entry into the thematic by providing another abstraction layer. There are many different different tools [145] that have a 'higher', meaning a non-HDL, language as an input and can output files as an HDL or on RTL. For example, the tool Sea Cucumber [146] can translate Java classes into the EDIF which means that the user does not see any HDL in this case. Still, many HLS tools translate from a C-based language (C, C++, SystemC) into an HDL (VHDL, Verilog), like eXCite [147], Catapult [148] or VivadoHLS [129]. On the one hand, smaller companies do not necessary have the knowledge or working power to design a tool that also synthesizes the design. On the other hand, this is perhaps also not desired by the user since he/she uses the HLS only as a first draft and then fine tunes the design in an HDL. Further, companies with long experience in the field of reconfigurable computing have already established tools for the design process. Designing a new tool that skips over the older tools makes these obsolete, but only if the new tool supports all functions the old tools had. Therefore, it is easier to integrate the HLS tools as a new entry in a tool chain rather than reworking all functions into the new tool.

2.3 Index Structures

This section introduces different index structures which are important for the Chapters 3, 4 and 5.

The last important field of this work are index structures. This section starts with the basic operations that are performed in databases and the importance of index structures to the performance of these operations. After this closer look onto the reasons for index structures, relevant index structures are presented, ranging from simple, widely used structures to more complex, specialized structures.

2.3.1 Operations in DBMSs and why Index Structures are important

Before different index structures are presented in Section 2.3.2 or 2.3.3 the need for them is explained in this section.

Most databases support certain operations that can be performed on the data they are holding. In this section the operations are broken down in very simple schemata that will be revisited in later parts of this section to highlight certain aspects of an index structure. The three main operations are to search, insert and delete certain data. Further there is the sorting of the data and updates replacing an old datum with a new one.

2.3.1.1 Search

The search operation is one of the main operations in this section, next to insertion (see Section 2.3.1.3) and deletion (see Section 2.3.1.4). Further, sorting (see Section 2.3.1.2) can accelerate the search operation.

The first and seemingly most important task of a database is to provide the user with the information he or she needs. In most cases the available amount of information exceeds the amount of currently needed amount of information. A real world analogy would be a library with a huge amount of books. The person in need of information mostly needs only a small subset of these books or even only a specific book. This leads to the first difference in a search:

- *Point Search* - The searched object either exists or not. This means the amount of results is one or zero.
- *Range Search* - The maximum number of result objects are unknown because the search is not specific enough to limit the results to a upper end.

Further, the person needs help since he or she doesn't want to look at all the books. The same is true for the database context. Since a human is incapable of processing a huge amount of data in a short period of time the database can not output every information it has onto the computer screen. Therefore a search algorithm has to be implemented to perform a search for certain data to limit the presented results.

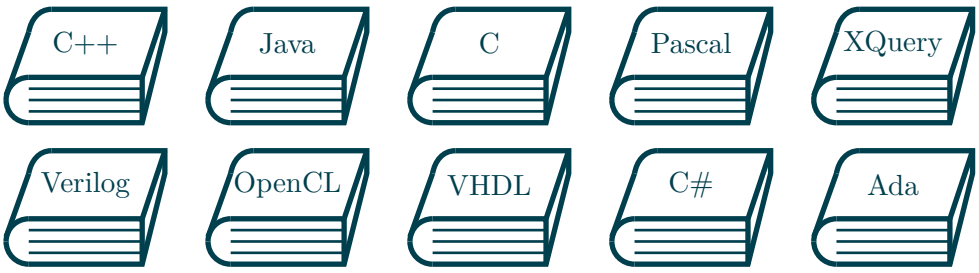


Figure 2.18: Different unsorted books of programming languages

Lets start with a very simple scenario, pictured in Figure 2.18, that is seemingly impractical but shows the importance of an index structure. The library is unsorted and the target is a certain book. One of the more naive search algorithms is the linear search [149] that can be used in this scenario. The simple idea behind this algorithm is to look at every record that is stored and either the data is found or all records are looked at and the searched information does not exist. For the point search, a positive result is the break point of the search, like if the book is found, the search in the library is stopped. Of course when the search starts from top to bottom and left to right there are certain books that are faster found than others. The $C++$ book is found just after the first step while the book for *Delphi* is found after 15 steps. There is no way to avoid that since there is no structure which implies certain rules that can be used. This also means that the negative result where no book is found, always takes n steps, where n is the number of books. This is necessary since the result that a book is missing is only available after all books are looked at. The same is true if a range search is performed. For example if multiple books starting with C in the title are searched, the search needs to continue till all books with this condition are found. Although there are already results available there can still be more. Stopping at the book C is not an option, even though the books $C++$ and C are found, there is further the third book $C\#$. And there could be still more results even if in this example there are only three books. Further the search for duplicates always has to read every single book with the same explanation as the range search. In the first step, to find the first occurrence of a book and after that, the search continues to look for

duplicates until all books are inspected. Looking at this scenario it becomes clear that operating on unsorted data will most likely touch every data for each search operation performed. This is because there is no way to predict the following data, so that rules can be applied to stop just like the point search. The solution for this problem is to sort the data.

2.3.1.2 Sorting

> *Sorting can accelerate search operations (see Section 2.3.1.1) and therefore influences the way data is inserted (see Section 2.3.1.3) or deleted (see Section 2.3.1.4) in an index structure.*

Sorting is the construction of an ordered listing of a set by a certain attribute of the sorted objects. This operation seems unimportant for the user at first, since he or she is mostly interested in certain information. Back to the library example it is clear that the user just wants the books no matter if the library is sorted or not.

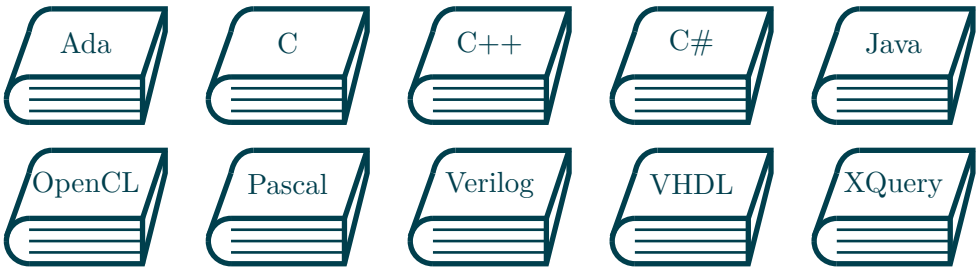


Figure 2.19: Different sorted books of programming languages

The Figure 2.19 shows the books of Figure 2.18 sorted alphabetically by their title. So far the sets of books are the same and each search operation would return the same results in both cases. The difference in searching in a sorted data set is that the existence of further results can be easier limited compared to an unsorted data set. If a point search for the book *Delphi* is performed, it is clear that it does not exist in the data set. Still, a linear search would go from left to right and then top to bottom through each book and check if it is the *Delphi* book. In the sorted data set the search can stop after the *Java* book, since the *Delphi* book should be between the *C#* and the *Java* book. Since this is not the case, the result is that there is no *Delphi* book. This saves half the amount of steps the unsorted linear point search needs since it can not make any assumptions where the *Delphi* book could be. For this small example the linear search on sorted and unsorted

books does not differ much in the steps the search needed, but with the growing number of books the difference gets significant. Further, there are more advanced search algorithms, like the binary search, or tree structures that depend on sorted data that can perform better than the linear search. But since these only work on sorted data a comparison between searches on a sorted and unsorted data set is not possible. For this section it is just important to understand that there is a trade off. The sorting of the data takes time while the search operations will be performed faster and will hopefully compensate the time loss and even save time in the late end. This trade off sounds more reasonable when it is taken into account that the data only needs to be sorted once and search operations can be performed multiple times. However, this is only possible if the data set does not change, which is typically not the case. This leads to the insertion of data.

2.3.1.3 Insert

> *The insert operation is one of the main operations in this section, next to searching (see Section 2.3.1.1) and deletion (see Section 2.3.1.4). Further, insertion is also important for updates in Section 2.3.1.5.*

The second main operation performed by data bases is the addition of new data to the data set of the database. Like the search operation, the insert operation can be divided into two different modes:

- *Point Insert* - Inserting a single data into the data set
- *Bulk Insert* - Inserting multiple data into the data set

The library example is continued by the addition of one or many books to the library as the insertion process. In the unsorted library the insertion is accomplished by adding all new books to the next shelve with a free spot. Figure 2.20 presents a library with three books where two further books will be added. In the case of the unsorted insertions, the library does not need to maintain its order, therefore the books are just added to the end (highlighted in green). So far there is no big difference between a point insert and a bulk insert into an unsorted set, except the ordering of the inserted books (*XQuery* before *C#*, or vice versa) at the end. But since the set is unsorted this is not important. This behavior can be translated to a data set in a database. With a pointer to the end of the set new data can be easily added to the end and the end pointer just needs to be incremented by the amount of inserted values. This behavior leads to a growing block of unsorted data which makes the linear search operations take longer as stated in the search section. To prevent this, the data should be sorted to speed up the search as stated in the sorting section. Every time new data or books are added, the data set or

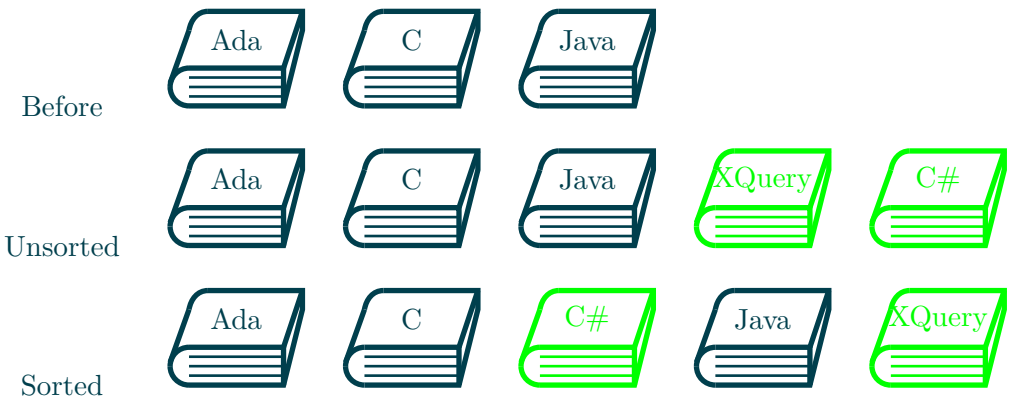


Figure 2.20: Sorted and unsorted insertion of two books

library needs to be sorted again. At the bottom of Figure 2.20 the result of the insertion of the books is pictured. The *C#* book is between the *C* and the *Java* book and the *Xquery* book is at the end. It would be possible to simply insert the books in the unsorted method above and then again sort the whole library. But the order of the already sorted data is not changed. The book *Ada* is still before *C* and this is before *Java*. This means that every time a datum is inserted into a sorted data block the order before and after the inserted datum stays the same. Therefore the insert operation into a sorted set can be achieved by first searching for and then adding the datum into the insert position without sorting the whole set. Again, like at the search operation, there is a trade off. The search for the correct position takes time against the insert direct into the end of the data block, but for each search operation on sorted data the lost time is hopefully reclaimed.

2.3.1.4 Delete

> *The delete operation is one of the main operations in this section, next to searching (see Section 2.3.1.1) and insertion (see Section 2.3.1.3). Further, deleting data is also important for updates in Section 2.3.1.5.*

The deletion of data is the last main operation that is performed in a database. While the insert of new data gives the user new possible search results, the deletion takes results away. There can be different reasons for this behavior that not necessarily need the interaction of the user. Often there are timing restrictions that let data get out of date or physical restrictions of the hardware, like deleting the oldest entry before inserting new data because of restricted memory space. There are two

common methods to delete a datum. Either the space the datum consumes is freed and can be taken by another datum or the datum is invalidated and therefore is not accessible by the user but still by the database system. In Figure 2.21 the books $C++$ and *Java* are deleted in these two ways.

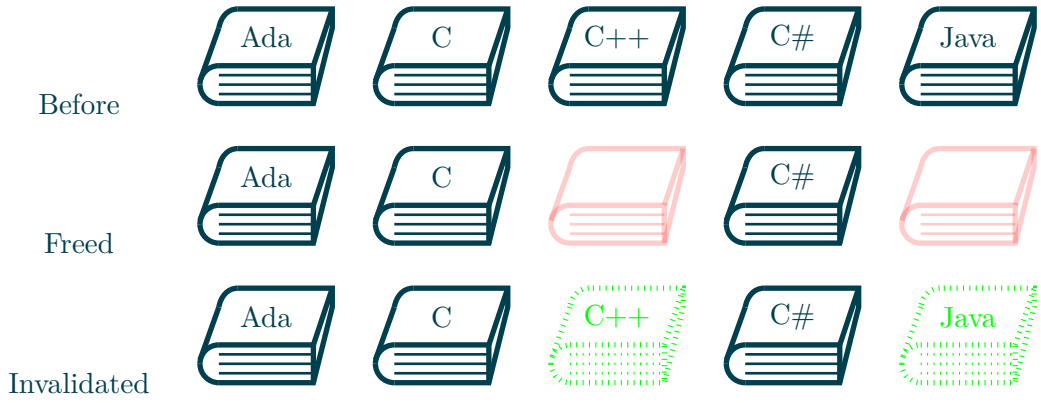


Figure 2.21: Different methods to delete two books of programming languages

Technically the data of the two books can still be stored on the storage medium (e.g. hard drive) but from the view of the DBMS it is unable to retrieve the data. The intention is to reuse the restored memory space for new data entries. For an unsorted list of books, the reusage of the deleted entries is easy as long as the DBMS keeps track where the data can be entered. In a sorted list the datum must fit, therefore the book *VHDL* could not be entered into the two resulting gaps. Many delete operations can therefore lead to a widely fractured data set. This makes a reordering of the data necessary either directly after deletion or when inserting new data. By invalidating the data the fracturing of the data can be prevented at the cost that the memory space can't be reused. Even though the accessible data is diminished, the required memory space stays the same. In this case, the database system has the advantage that it can still restore the data and therefore undo the delete operation at a later point.

2.3.1.5 Update

Update operations delete old data (see Section 2.3.1.4) and replace it with new data (see Section 2.3.1.3), therefore avoiding an additional search (see Section 2.3.1.1).

Updates are, like searches, inserts and deletes, common operations in a DBMS. An update operation can be substituted by a delete operation followed by an insert operation. Hence the update operation is often not seen as main operation. As the name implies updates bring a certain datum to the newest version and replace the old version. By deleting the old data and then inserting the new data an update can be performed. This method fulfills all necessary tasks for an update but can still be optimized. Both operation need to search for the datum that needs to be deleted and then reinserted. Searching only once and delete old data while inserting new data avoids another search operation.

2.3.2 Graphs and Binary Tree Variants

After the introduction of the main operations in a DBMS in Section 2.3.1 the fundamentals and vocabulary of trees are explained in this section. Therefore, graphs (see Section 2.3.2.1) and binary trees (see Section 2.3.2.9) are presented before more complex trees, like B-trees, are introduced in Section 2.3.3.

This section presents several different data structures which starts with the origin in graph theory and gradually presents different tree structures. After introducing the main operations performed on the data in a database system in the last section the data structures are examined under these aspects.

2.3.2.1 Graphs

Graphs are detailed mathematical descriptions of the index structures that will be introduced in Section 2.3.2.9 or 2.3.3. Further, one of the common terms used for describing trees are coined in this section (terms reach from Section 2.3.2.1 to 2.3.2.8)

Graphs are mathematical structures that combine different objects by a certain relation. A graph is commonly an ordered pair of a set of objects and a set of relations between those objects. One of these objects is mostly described as vertex or node while the relations are represented by edges addressing a pair of vertices. This leads to the mathematical representation of a graph G as $G = (V, E)$ where V is the set of vertices and E is the set of edges. Often graphs are not only described in their mathematical representation but also in their graphical presentation which can help to explain a graph without using excessive mathematical formulas.

2.3.2.2 Vertices

Graphs consists of vertices and edges (see Section 2.3.2.3).

A vertex is an element of a set. This set consists of definite, distinct objects that are or are not related to each other. But without the set of edges this is not possible to determine. As mentioned earlier graphs are not limited to their mathematical description and are often portrait in pictures as graphical representation. In these cases a vertex is mostly pictured as a geometrical form (e.g. a circle or a rectangle) containing a description that helps to distinct this vertex from all other. Figure 2.22 gives an example of the set V_1 with the two elements 1 and 2 in its mathematical and graphical representation.

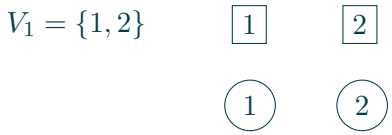


Figure 2.22: The set V_1 in its mathematical presentation on the left and two possible graphical presentations of the vertices 1 and 2 on the right

2.3.2.3 Edges

Graphs consists of vertices (see Section 2.3.2.2) and edges.

An edge is a pair of exactly two vertices. The edge therefore describes an abstract relation between the two elements. This relation is mostly only understandable with a certain context of the scenario a graph represents since the mathematical and graphical presentation can only describe a certain connection between two objects but not the meaning of this connection. Therefore an edge in its graphical presentation is a line between two vertices. Multiple edges can be distinct by the nodes the lines are connecting and mostly do not need further description in certain graph forms. An example of the mathematical and graphical presentation of the graph V_2 can be seen in Figure 2.23

Although the three lines between the nodes look graphically the same, the distinguishable vertices make it possible to determine which element of the set E is represented. In this example the edges are presented as unordered pairs (using braces instead of round brackets) which is usual in an undirected graph, which is a special form of a graph. The degree of a vertex can be determined by the number of edges that are connected to it and can be further specified in certain graph forms.

$$V_2 = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{2, 3\}, \{3, 4\}\})$$

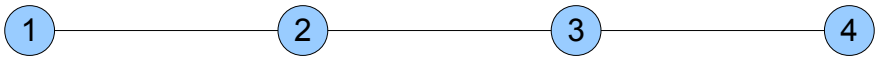


Figure 2.23: Undirected graph V_2 with 4 vertices and 3 edges

Therefore, two common graph forms are presented, the undirected and directed graph.

2.3.2.4 Undirected Graphs

Graphs can be undirected or directed (see Section 2.3.2.5) depending on the used edges (see Section 2.3.2.3).

Undirected Graphs only have edges without a certain orientation. This means that a relation between two elements always applies to both elements. In Figure 2.23 the edges can be seen as a "neighborhood" relation. If 1 is the neighbor of 2 then 2 is also the neighbor of 1. This leads to the presentation of an undirected edge as an unordered pair. Here the degree of a vertex is not dependent by the edge direction, which means in this example that the degrees of vertex 1 and 4 is one and the degrees of 2 and 3 is two.

2.3.2.5 Directed Graphs

Graphs can be undirected Section 2.3.2.4 or directed depending on the used edges (see Section 2.3.2.3).

Directed graphs only have edges with a certain orientation. In this case the relation between two elements is limited and applies only from a specific start vertex to a destination vertex. Two simple examples are represented in Figure 2.24.

The elements in the vertices set have not changed from V_2 of Figure 2.23 but the set of the edges. There are two variants. Variant (a) can be seen as the "successor" relation. In this example this means that the next natural number is reached by adding 1 to the current number and therefore is the successor. This means 2 is the successor of 1, 3 is the successor of 2, and so on. This relation

$$V_{2(a)} = (\{1,2,3,4\}, \{(1,2),(2,3),(3,4)\})$$
$$V_{2(b)} = (\{1,2,3,4\}, \{(2,1),(3,2),(4,3)\})$$

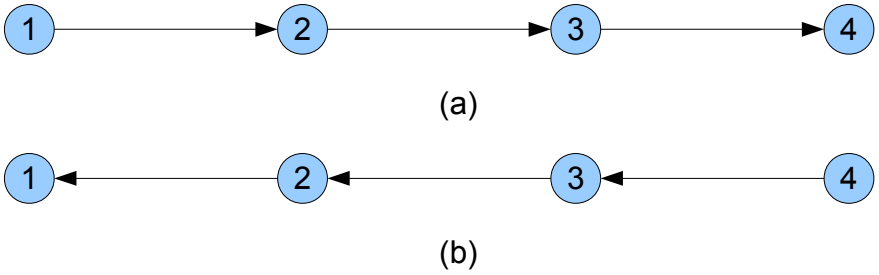


Figure 2.24: Directed graphs with (a) the "successor" relation and (b) the "predecessor" relation

works only in one direction because the other direction would be the "predecessor" relation (b) which is the number reached by subtracting 1 from a number. In the mathematical representation the edges are therefore represented as ordered pairs in round brackets instead of braces. This is the reason for the different spellings between the edge sets of $V_{2(a)}$ and $V_{2(b)}$ since the sequence of the vertices in an edge is important. For the graphical presentation of this behavior mostly an arrow head is added to the line end between two vertices, pointing to the destination vertex. The degree of the vertices in a directed graph can be further specified. All edges that point to a certain destination vertex count as indegree for this vertex. All edges that start at a certain vertex count as outdegree for this vertex. In the example the indegrees for variant (a) the indegree of vertices 2, 3, 4 is one while no edge points towards vertex 1, which means the indegree of this vertex is zero. On the other hand, outdegree of the vertices 1, 2 and 3 is one while 4 has the outdegree zero in variant (a).

2.3.2.6 Paths

Paths are subgraphs of a original graph (see Section 2.3.2.1) described by a sequence of alternating vertices (see Section 2.3.2.2) and edges (see Section 2.3.2.3).

A path is a sequence of vertices that can be "visited" by using connected edges between the different vertices. Since a path P uses a set or subset of vertices V_G and edges E_G of a given graph, a path P is a subgraph or the same graph as the original graph G. This means that the sets of P are smaller or equal to

the sets of G where for a subgraph one of the sets must be really smaller than the original. The "visits" of the vertices can help to represent and solve specific problems by using the relations between the vertices. In Figure 2.24 the edges of $V_{2(a)}$ represent the the direct successor relation between two vertices. A problem that can be solved with this relation could be if a specific number comes after another number. For example, is the number 4 a direct or indirect successor of the number 2? A path can be described as an altering sequence of vertices and edges (e.g. $v_0, e_0, v_1, e_1, \ldots, e_n, v_{n+1}$). This representation style can be reduced to only the vertices if the distinction of the edges is not necessary or simply not wanted. In the following sections only the vertices are used and represented as a path by a line over all vertices of the path. The path that proofs that 4 is an successor of 2 is a path P_{result} that starts at the vertice 2 and ends at the vertice 4. The direct connections can be checked by a search for the edge $(2, 4)$ which is not the case. Therefore, there is no path $P_{(2,\ a,\ direct)}$ between vertices 2 and 4. Still there can be a path $P_{(2,\ a,\ indirect)} = \overline{2 \cdots 4}$ that starts with 2 and ends with 4. In this case $P_{(2,\ a,\ indirect)} = \overline{2\ 3\ 4}$ is possible by using the edges $(2, 3)$ and $(3, 4)$ to build a path between those vertices. Therefore it is clear that 4 is a successor of 2. The edges of $V_{2(b)}$ describe the predecessor relation between two vertices. The description of the example problem can be switched into: Is the number 2 a direct or indirect predecessor of the number 4? Again, there exists no direct path $P_{(2,\ b,\ direct)}$ between the vertices 4 and 2. Still, there is an indirect path $P_{(2,\ b,\ indirect)} = \overline{4\ 3\ 2}$ that makes 2 a predecessor of 4. This further shows the connection between a path and its graph while path $P_{(2,\ a,\ indirect)}$ is possible in graph $V_{2(a)}$ it is impossible in graph $V_{2(b)}$. The same is true for the path $P_{(2,\ b,\ indirect)}$ and the graphs $V_{2(a)}$ and $V_{2(b)}$ vice versa.

2.3.2.7 Trees

With the knowledge of graphs (see Section 2.3.2.1), vertices (see Section 2.3.2.2), edges (see Section 2.3.2.3) and paths (see Section 2.3.2.6) it is possible to describe a tree. This section coins one of the common terms used for describing trees (terms reach from Section 2.3.2.1 to 2.3.2.8)

A tree is a special form of a graph [150]. It still consists of vertices and edges but adds more restrictions to these groups. A graph G has to fulfill the following conditions to count as a tree:

- *The graph G has to be connected* - A graph G is connected if there is a path between every combination of two vertices. This means that each node in a tree has the minimum degree of one. An exception is a tree consisting of only a single vertex, because the single vertex counts as connected.

- *The graph G has no cycles* - A graph G is without cycles if there are no paths that start and finish at the same vertex. Further applies for the path that an edge can not be used twice, which is important for undirected graphs.

At this point the vertices are renamed into nodes, which is the more common term used in the context of index structures. This marks a small shift from the mathematical context to the practical use [17]. In the following, some main terms connected to trees are introduced and the orientation of a tree is described to identify them.

Root

A root is the starting point in directed trees to build a path to each other vertex possible. The reason for this is that the root has no incoming edges, which means that the indegree is zero. Therefore starting at any other node in the tree makes it impossible to build a path to the root. In Figure 2.25 are four different trees represented.

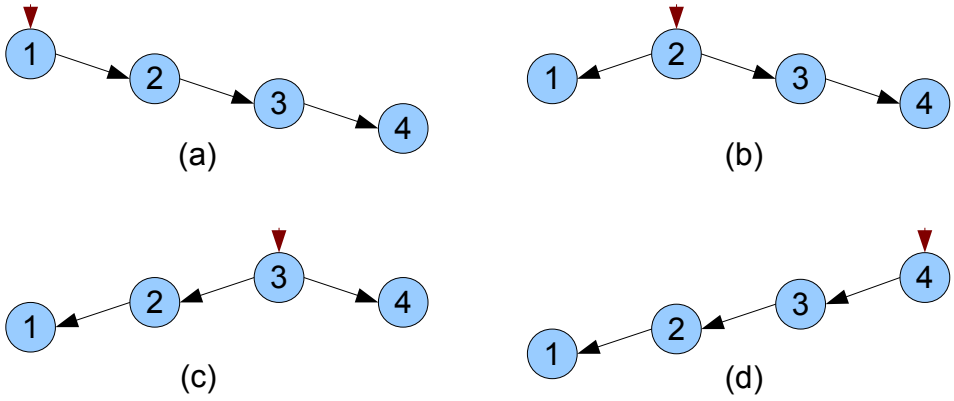


Figure 2.25: Directed trees with the following vertices as root: (a) node 1, (b) node 2, (c) node 3 (d) node 4

In each tree the root node is marked with a red arrow. Therefore the roots are node 1 in tree (a), node 2 in tree (b), node 3 in tree (c) and node 4 in tree (d). Only from these nodes it is possible to build a path to each other node. In tree (a) the paths $\overline{1\ 2}$, $\overline{1\ 2\ 3}$ and $\overline{1\ 2\ 3\ 4}$ make it possible to visit each other node from the root 1.

Axis Orientation in a Tree

[17] Especially in the later Chapters 3, 4 and 5 only the term 'node' is used.

There are several terms for nodes inside a tree that correspond to a specific node or node group from the view of a specific node. This helps to describe certain dependencies that are present between the current node and other entities in a tree. Figure 2.26 contains a simple version of tree with the root node r.

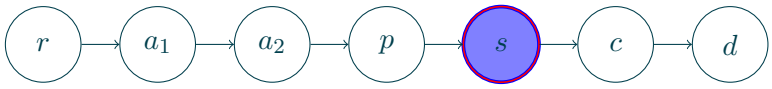


Figure 2.26: Directed graph with orientation for the node s

In this example the node s (for $self$) is the starting point. The child of s is the node c. The requirement for a node to be a child of another node is that it is directly reachable by only one edge. Therefore a path between parent and child contains no other nodes than these two. The path $\overline{p\ s}$ describes that p is parent of s and s is child of p while the path $\overline{s\ c}$ describes that s is parent of c and c is child of s. Further there are ancestors and descendants of a node, which are defined by a path from the root r to the last possible node l without revisiting a node. For a node x in a path $\overline{r\ \cdots\ b\ x\ a\ \cdots\ l}$ all nodes in $\overline{r\ \cdots\ b}$ are ancestors of x and all nodes in $\overline{a\ \cdots\ l}$ are descendants of x. In the example the path $\overline{r\ a_1\ a_2\ p\ s\ c\ d}$ is given which results in r, a_1, a_2, p being the ancestors of s making p not only the parent but also an ancestor of s. The other nodes c and d are descendants of s

Leaves

Leaves are kind of the opposite of the root. Instead of the indegree the outdegree is zero which makes leaves the last nodes in paths through a tree. While the outdegree of the root can be greater one and therefore have multiple children, the indegree of a leaf is always one and therefore has only one parent node. In Figure 2.25 the example trees (a) and (d) have only one leaf (either node 1 or 4) while the trees (b) and (c) have two (both nodes 1 and 4).

Internal Nodes

All nodes that have a child node are internal nodes. Therefore the set of leaves and the set of internal nodes are disjoint. In all tree vaiants of Figure 2.25 the nodes 2 and 3 are internal nodes.

2.3.2.8 Balanced Trees

After the introduction of trees (see Section 2.3.2.7), this section coins the term 'balanced tree'. This term is important to describe the many different types of trees that will be further introduced (see Section 2.3.2.9, 2.3.2.8, 2.3.3).

A balanced tree can be divided into subtrees of equal size. In Figure 2.27 are two trees given with the same node and only the edges differ.

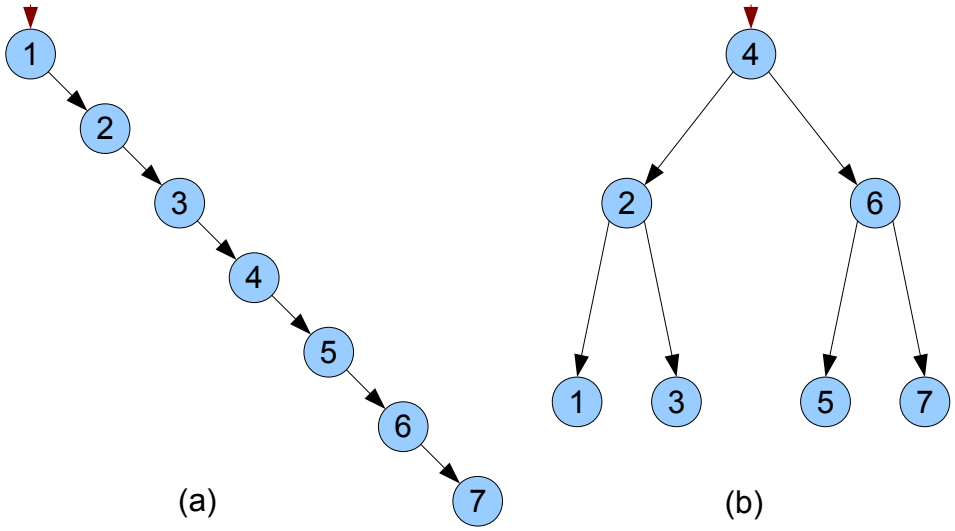


Figure 2.27: Directed tree with 7 nodes: (a) unbalanced (b) balanced

In example (a) it is impossible to build a non-empty left subtree since no node has a left child. On the other hand all right subtrees only have one path from its subtree root to the leaf 7. In the later part of this section we will see that this limitation of choice can lead to unsatisfying performance of a tree as an index structure. The example tree (b) can be divided into two subtrees that contain the same amount of nodes and the same amount of edges. In order to compare these subtrees there are two further parameters needed, the depth of a node and the height of the tree.

Depth of a Node

The depth of a node depends on the number of edges used from the root to reach the node. This means the length of the path is important. Since the length of a path l is determined by the number of nodes visited, the number of edges e is $l-1$; The root has always the depth of zero while all other nodes have a positive depth. As an example the path $p = \overline{a\ b\ c\ d\ e\ f\ g}$ describes the path from the root a to the node g. The length of the path l_p is 7, which leads to the depth of the node g (d_g) of $d_g = e_p = l_p - 1 = 6$. Looking back at Figure 2.27 the depth of the node 5 is 4 in variant (a) and 2 in variant (b).

Height of a tree: The heigth of a tree h is determined by the maximum depth

of a leaf. In Figure 2.27 the tree variant (a) has only one leaf node which has the depth 6 and therefore the height is also 6. In contrast, the variant (b) has the leaves 1, 3, 5 and 7 which all have the depth of 2. Therefore the height of the tree (b) is 2.

Coming back to the subtrees, dividing the tree (b) on the root into two subtrees generates two trees with equal height of 1. On the other hand, the tree in (a) would be divided into an empty tree and a subtree of height 5. Balanced trees have all their leaves in the same depth while the height of the subtrees from one parent node are equal. This is only possible for trees that consist of a specific number of nodes since adding a node 8 to the balanced tree of Figure 2.27 would result in an unbalanced tree. Therefore, the requirements for balanced trees are weakened allowing the depth of leaves and the height of the subtrees to differ in a certain range depending on the tree type.

2.3.2.9 Binary Tree

In this section the binary search tree is introduced which is a special variant of a binary tree. Both are unbalanced trees (see Section 2.3.2.8), therefore balanced variants are introduced in Section 2.3.2.10.

The first tree index structure that gets introduced in this work is the binary search tree. A binary tree is a rooted, directed graph. Each node can have up to two children which means the outdegree of the nodes is between 0 and 2. The indegree is always 1 except for the root having the indegree 0.

The difference between a binary tree and a binary search tree are the optimizations for database operations (see Section 2.3.1), especially the search operation. In Figure 2.27 both trees are binary trees but only variant (b) performs well for searches. There are two important factors for this. The first factor is the arrangement of the values inside the tree. Both trees are sorted in a way that numbers with higher values than the current value are stored in the right leaf while numbers with smaller values are stored in the left leaf. This is not the case for all binary trees, in Figure 2.28 the values of the tree are symbols that can not be easily compared without a defined order.

Searching for the value 6 in the trees of Figure 2.27 and 2.28 shows the difference between the tree variants (a), (b) and (c). Every search in a binary tree starts in the root. In the trees (a) and (b) a comparison between the value in the root and the searched value is possible. In both cases the searched value is higher than the node value, therefore the next node visited is the right child. In the tree (c)

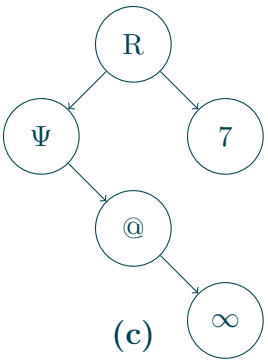


Figure 2.28: A binary tree but not a binary search tree

a direct comparison between R and 6 is not possible[18]. Therefore a calculated decision which child should be visited next is not possible. Both directions can lead to the searched value and in this case all nodes of the tree are visited, and the value is not found in the tree (c). The other important factor for a binary search tree is that the tree is balanced. This characteristic allows a faster limitation of the search space. While the decision to take the right child is the same in both trees (a) and (b) the consequences are different. In (a) only the root is removed from the search space while there are still 6 nodes to look at. On the other hand, in (b) the left subtree with the nodes 1, 2 and 3 can be ignored since the searched value is either inside the right subtree or not in the tree at all.

While the data remains sorted by first searching the position inside the tree and then either generating a new node or deleting the searched node, the balance of the tree is ignored by these operations. In Figure 2.29 the tree (A) consist of the root with the value 1.

Assuming the values are defined in the natural numbers $\mathbb{N}$ there are no values left that can be smaller than 1. This means that every other value that is inserted belongs into the right subtree. Further, if the next inserted value is the 2, this behavior continues as seen in step (B). The sequential insertion of an already sorted list of values results in a highly unbalanced tree, often called a degenerated tree because of the similarity to a linked list. In order to prevent such behavior the values are often inserted as a bulk instead. In the step (C) there is an array of element from 0 to n. Taking the element in the middle of the array and define it to be the new root, divides the array into two parts that will be the left and right subtree of the root. This recursive generation of a binary search tree stops

[18]Theoretically, an order could be defined on the given symbols and then the values could be sorted and compared, but for the simplicity of the example this possibility is left out.

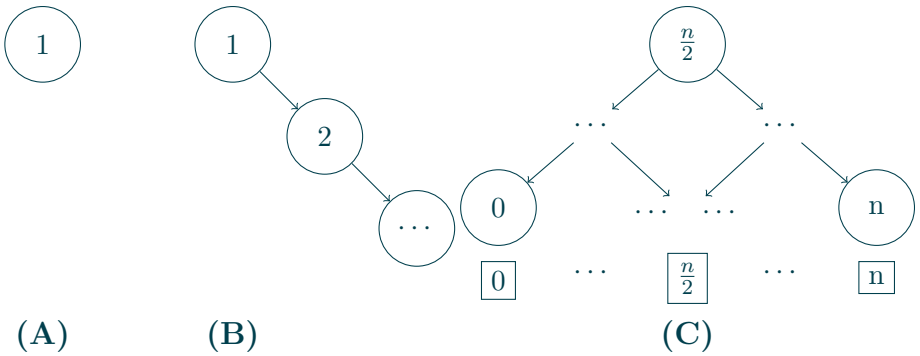


Figure 2.29: Insertion in binary tree

when all subarrays contain only a single value. Although the tree is close to a perfect balance depending on the number of elements inserted, each further insert operation can still degenerate the tree again.

2.3.2.10 Balancing Binary Search Tree Variants

This section introduces balanced (see Section 2.3.2.8) variants of the binary search tree (see Section 2.3.2.9) to show how balancing in trees can be handled. This is important for B-trees and B^+-trees introduced in Section 2.3.3.

The flaw of a possible degeneration of the search tree lead to new types of self-balancing binary search trees.

The AVL tree (named after Georgi Maximowitsch **A**delson-**V**elski und Jewgeni Michailowitsch **L**andis [151]) is a binary search tree that introduces balance factors for nodes. These balance factors are calculated by the height of the right child subtree minus the height of the left child subtree as represented in Equation 2.1.

$$\text{BalanceFactor(x)} = \text{Height(RightSubtree(x))} - \text{Height(LeftSubtree(x))} \quad (2.1)$$

For the node x the functions RightSubtree(x) and LeftSubtree(x) return the corresponding subtree while the Height(N) function expects a tree N to calculate its height. A binary search tree is an AVL tree if each node n in the tree satisfies the requirement that BalanceFactor(n) $\in \{-1, 0, 1\}$. An example AVL tree is represented in Figure 2.30.

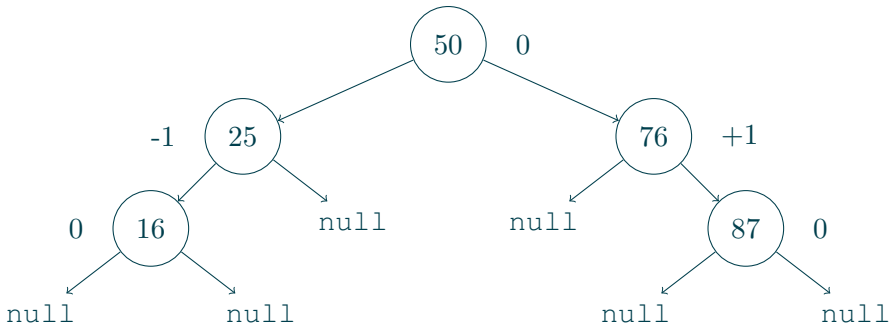


Figure 2.30: An AVL tree with balance factors added to the nodes

Starting from the outer left node with the value 16 its balance factor is 0 since neither the left subtree nor the right subtree contain any node. This means that both heights of the subtrees are 0 and therefore are balanced.

In order to fulfill this requirement a rebalance phase is added to the insertion or deletion operation of a node if the resulting tree does not hold the requirement. The rebalance phase involves 'rotations' which is an exchange of a child node switching place with an ancestor node. A simple rotation is the direct exchange between child and parent node, making the child the new parent and the parent the child, while a double rotation also involves the parent of the parent node.

A red-black tree is less restrictive than an AVL tree. Although still a balanced search tree, the height of one subtree to another can differ in a wider range for the red-black tree than in an AVL tree. In order to maintain the balance, to each node an extra bit is added that provides a balancing function to the tree. This bit is often interpreted as two colors which are black and red and the origin of the trees name. There are requirements that must be fulfilled by a binary search tree to count as a red-black tree:

1. Each node is either red or black
2. The root is black
3. All leaves are black (In this case all `null` nodes)
4. A red node has always two black children
5. Every path from a given node to any of its descendant `null` nodes visits the same number of black nodes.

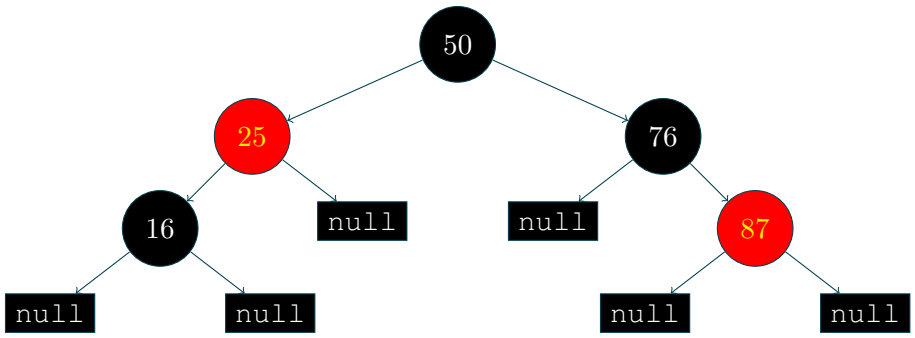


Figure 2.31: A red-black tree with nodes marked in red and black.

2.3.2.11 Heap

This section introduces heaps as a variant of binary trees (see Section 2.3.2.9) to retrieve the smallest/biggest item of a set. A heap is not only used in the replacement selection algorithm (see Section 2.3.4.2) but is further important for PatTrieSort introduced in Chapter 3.

The smallest item from a collection can be efficiently retrieved by using a (min-) heap [152], which supports inserting as well as removing an item in logarithmic time (in comparison to the items stored in the heap). The internal organization of the heap is a tree, most often a complete binary tree memory-efficiently stored in an array. The heap condition requires the root of each subtree to contain the smallest item of the subtree. Hence adding an item inserts the item as leaf to the heap tree and performs a *bubble-up* operation, which swaps the item with its parent as long as it is smaller than its parent. For removing the smallest item from the root of the heap, the item in the most-right leaf of the bottom level is moved to the free space in the root. Afterwards in order to reestablish the heap condition, during a *bubble-down* operation the root item is recursively swapped with its minimum child if the minimum child is smaller than it. An example of the bubble-up/down operation can be seen in Figure 2.32.

This minimum heap starts with 6 values where the minimal value is 25 at the root of the tree in step (a). If the smallest item is retrieved, the 25 is removed from the root and the value 40 from the most right leaf is entered into the root in step (b). The value in the root is not necessarily the smallest item inside the heap therefore a bubble-down operation needs to be performed. The value 40 is higher than both child values 33 and 39. In this case the smallest of both values is chosen and switched with the root value in step (c). Another check on the child nodes reveals that 40 is smaller than 45 and 50. Therefore no further bubble-down operation is

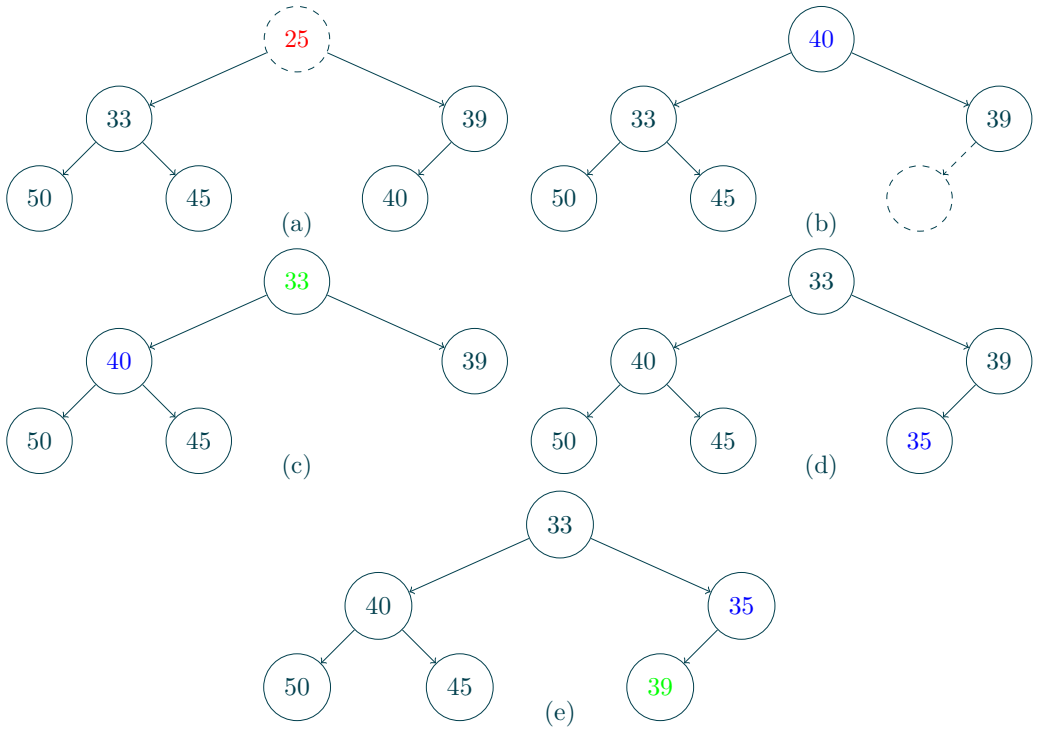


Figure 2.32: A minimum heap starting with 6 values. The smallest element is removed (a) and replaced with a value in a leaf (b). Thereafter a bubble-down operation is performed (c). Further the value 35 is added (d) and a bubble-up operation performed (e).

needed and the removal of the smallest item is finished. In the next step (d) the value 35 is entered into the heap as the most right leaf. Since 35 is smaller than the parent value 39 a bubble-up operation is performed switching the parent and the child values in step (e). The insertion of 35 is finished at that point since the father node has a smaller value than 35.

2.3.3 B-Trees and Variants

After the introduction of the terms in context of several binary tree variants in Section 2.3.2, this section introduces more advanced trees, like the B-tree and its variants. Especially the B^+-tree is important for the Chapters 4 and 5.

The trees introduced so far all resemble a binary tree having one key per node and

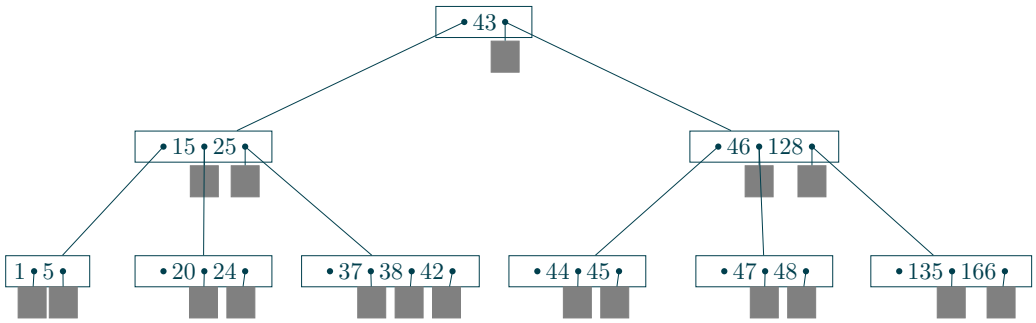


Figure 2.33: An example of a B-tree with order 2. The keys are numbers and the values are gray boxes.

two children. This is only halves the solution space per visited node while multiple keys in a node would reduce the solution space faster with less nodes looked up. Therefore, this section introduces the B-tree and its variants the B^+-tree and the CSB^+-tree.

2.3.3.1 B-Tree

B-trees are the fundamental tree structure for the index structures used in later chapters. They are necessary for the B^+-tree in Section 2.3.3.2 and the CSB^+-tree in Section 2.3.3.3.

B-trees [153] allow the usage of multiple keys with corresponding values in the same node. Therefore, the node can be loaded as a block of data from external memory, like a HDD. Since each node should have a similar amount of keys the so called '*order*' of a tree is introduced. The order k of a tree guarantees that at least k keys are located in a node. Further, no more than $2 \cdot k$ keys are allowed per node otherwise it is split which divides the keys and values between the original and a new node. Each key in an inner node has a pointer to a child with keys less or equal to its own. The amount of pointers is defined by the order k of a B^+-tree as follows: a node can have $k + 1$ to $2 * k + 1$ pointers because there is an additional pointer to a child that contains all keys that are bigger than the keys in its father node. The only exception is the root which has no minimum. An example B-tree can be seen in Figure 2.33. It has three interior nodes and six leaves.

Each inner node except the root has at least two keys since in this example the order $k = 2$ applies. Each integer key has a corresponding value (gray box) and a pointer to a child node. The leaves are not connected therefore parent nodes have to be revisited if a range search is performed. For example a search for all values

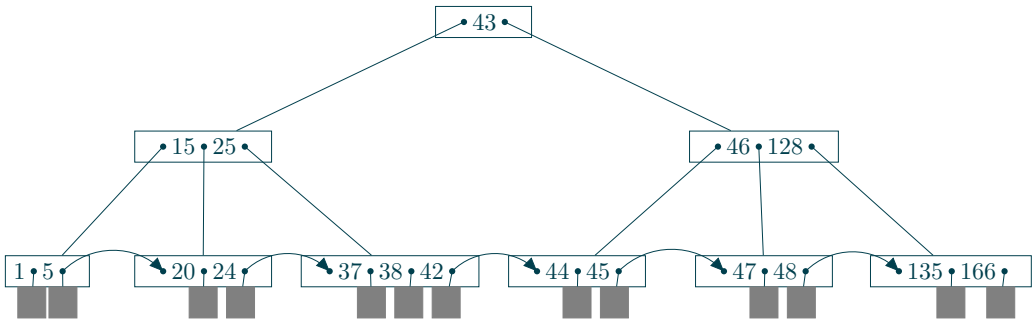


Figure 2.34: An example of a B^+-tree with order 2. The keys are in the nodes (numbers) and the values are only in the leaves (gray boxes).

with keys ranging from 1 to 24 would revisit the interior node with the keys 15 and 25.

2.3.3.2 B^+-tree

B^+-trees derived from B-trees (see Section 2.3.3.1) and are important indices in LUPOSDATE (see Section 2.1.5) and therefore important for the Chapters 3, 4 and 5. They are further necessary for the the CSB^+-tree in Section 2.3.3.3.

B^+-trees [154] are widely used in database indices and are derived from B-trees [153]. In contrast to its ancestors a B^+-tree saves values only in leaves which are nodes that do not have children. All other nodes (interior nodes) only contain keys. An example of a B^+-tree can be seen in Figure 2.34. It looks familiar to Figure 2.33 with some exceptions.

In this example, the numbers inside the nodes are keys while the values are represented by gray boxes beneath the leaves. This also means that there are less values stored in this tree than in the B-tree example before. But this is only to match the structures of the two trees otherwise the structure of the B^+-tree would look slightly different. Still, this example also shows that the B^+-tree can have keys in interior nodes that have no corresponding value in the leaf nodes, like the keys 15, 25, 43, 46 and 128. The B^+-tree has some advantages for search operations. The lack of values inside the interior nodes makes it possible to hold more keys per inner node and therefore less nodes are needed to be loaded while searching. Furthermore, the connected leaves make it easier to perform a range search since the parent nodes do not need to be revisited to find the next leaf node. The same rules for the order k apply as in case of a B-tree but only for interior nodes. The

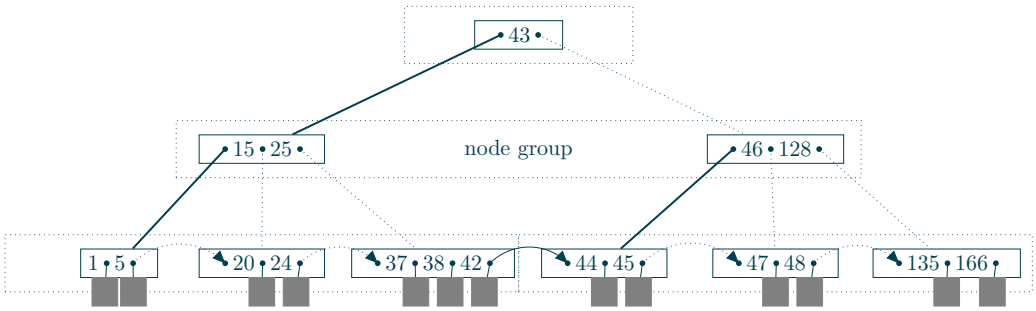


Figure 2.35: A CSB$^+$-tree with the same keys and values as the B$^+$-tree seen in Figure 2.34.

leaves in a B$^+$-tree can have their own order k'. k and k' put the same limitations and requirements on the nodes, but make it possible that interior nodes can have a different number of keys than the leaf nodes. This can be necessary to fit the nodes better into the block size (typical 8 KB) of a HDD to make a block wise processing of operations possible.

2.3.3.3 CSB$^+$-tree

CSB$^+$-trees are a special variant of B$^+$-trees (see Section 2.3.3.2) and are used as index structure on the FPGA in Chapter 5.

CSB$^+$-trees [155] are modified B$^+$-trees with only one pointer per node to the children. The basic idea behind this is that a B$^+$-tree has per node one more pointer than keys (see Figure 2.34). If the child nodes are scattered in different address spaces it makes sense to address each node independent but also limits the capacity of the parent node by holding many pointers. CSB$^+$-trees (see Figure 2.35) avoid this problem by putting child nodes into *node groups* and storing nodes in a node group contiguously. While searching inside the tree the eliminated pointers can easily be restored by adding an offset to the one leftover pointer.

2.3.3.4 Searching inside a B$^+$-tree Node

In Chapter 5 the search operation inside a B$^+$-tree is optimized, therefore it is important to highlight how these search operations can be performed.

The basic concept of searching inside a B$^+$-tree is divided into two steps. First, search inside the node for the correct key. Second, follow the pointer of the key and continue this search recursively for the new node until a leaf is reached. Especially

the first step is interesting since it gives the opportunity to parallelize the search for the correct key. Due to the architecture this is most suitable for the FPGA. Figure 2.36 shows three different search methods inside a sorted array of numbers. These numbers can be seen as keys in a B^+-tree inner node. Thus, a search inside this array corresponds to the search to find the correct child and continue the search inside a B^+-tree from there. The linear search (a) starts at the beginning of the array and inspects every item sequentially. If it is the search key, in this case 76, it takes 13 steps to find the correct value with this method. The second method, called binary search (b), also works sequential, but uses the sorting of the array to its advantage. It starts in the middle and decides if the searched value is bigger or smaller than the inspected element. This gives the opportunity to halve the solution space each time an element is inspected. The binary search needs only four steps to find the correct value. Still the linear search has one advantage against the binary search even though it needs more steps. The linear search can handle keys that are compressed. The different values of the array can be seen as integer values taking four bytes of space each. In this example the difference between one value and its successor is never bigger than 11 which means that with four bits each number can be described by its predecessor. Instead of 16 times 32 bit the compressed array only needs the starting 32 bits (which also can be further compressed, not going into details at this point) and 15 times the difference for only four bits. So there is a trade off between compression and fast search. But we can search even faster than the binary search. Assuming we could compare all numbers in the array at the same time (c) the search would be over in just one step. For sequential working processors this task is hard to achieve but with logic in hardware it is possible to realize such search behavior.

2.3.4 Initial Index Construction from sorted Data

> *Performing individual insert operations (see Section 2.3.1.3) for a large set can be time consuming for the tree variants presented so far (see Section 2.3.2 and 2.3.3). Therefore the possibility of creating an index on sorted data is introduced, since Chapter 3 focuses on the sorting while Chapter 4 on the construction aspect.*

In Section 2.3.1.3 two different insertion modes were introduced. The *single insertion* for a single datum and the *bulk insertion* for data. Assuming there exists an unsorted data set with N key-value pairs there are two different ways to construct an index. First, the single insertion for each pair can be used. This would lead to N searches for the correct position of the pair in the index and possible resulting balancing operations on the index (see Section 2.3.2.8). On the other hand, the

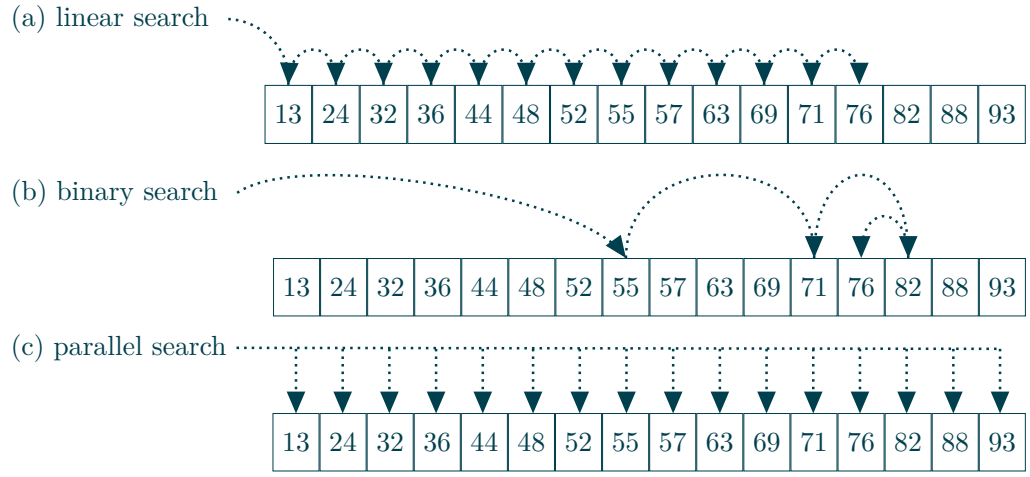


Figure 2.36: The sequential (a) linear and (b) binary search methods and (c) a parallel approach in an array with 16 values. The search key is 76.

data set could be sorted and a matching tree could be calculated, which comes close to the bulk insertion method on an empty tree although we are technically constructing and not inserting. This means that a bulk insertion would cause balancing operations, while in the index construction no balancing is needed because of precalculations. Depending on the size of N one of the methods can be faster than the other. Especially, if the data set is big, the N searches and the multiple balancing actions can take more time then sorting the data and directly construct the index. In Figure 2.37 a construction of a B^+-tree with the orders $k = k' = 2$ is depicted.

In layer **A** the already sorted data set is presented with the keys as numbers and the values as red rectangles. Starting from left to right the leaf level (**B**) can be constructed. Since the leaf order k' is 2 up to 4 keys can be stored in one leaf node. In **B.1** the first four key-value pairs are inserted into the first newly created leaf and the highest keys is also duplicated into a new parent node (**C.1**) at the layer **C**. This key (4) references the child node by a pointer. After that a new leaf is created and the next 4 key-value pairs are inserted (**B.2**). Again, the highest key is duplicated into the already existing parent node with a pointer to the new leaf (**C.2**). The same happens in the steps **B.3** and **C.3**. At the end, the last 4 key-value pairs are added to a new leaf (**B.4**) but since there is no data left only a pointer is added to the parent node but no further key (**C.4**).

In this example, the data set was already sorted but still this is a step that has to be done before the index can be constructed. There are many different sorting

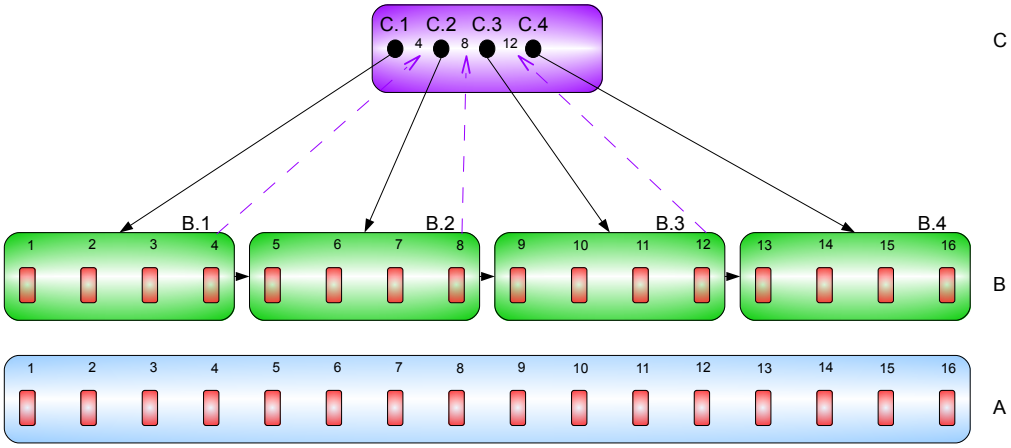


Figure 2.37: Initial construction of a B$^+$-tree with orders $k = k' = 2$ from sorted data (A). The keys are numbers and the values are red rectangles.

algorithms for different scenarios which makes it hard to introduce each one of them. Since this work has a focus on SW DBMSs with large data sets, we will only present some sorting algorithms optimized for that purpose.

2.3.4.1 (External) Merge Sort

> *Since the index construction (see Section 2.3.4) can happen only on sorted data, it is important to introduce some sorting algorithms. External Merge Sort is commonly used for huge dataset and is important for its variant called replacement selection in Section 2.3.4.2.*

(External) merge sort [149] is known to be one of the best sorting algorithms for external sorting, i.e., where the data is too large to fit into the main memory. The merge sort algorithm first generates several initial runs, which contain already sorted data. The initial runs are afterwards merged to generate a new round of runs. A new run contains the sorted data of its merged runs, so that the number of new runs becomes less while the size of each new run increases. This process is repeated until all the data is sorted.

Instead of merging only two runs, it is more efficient to merge several runs. In order to merge a new run from several runs, we always need to find the smallest items from these runs. Thus, a heap (see Section 2.3.2.11) is the ideal data structure to perform this task.

The runs can be generated by reading as much data into main memory as possible, sorting this data and write this run to external storage. For sorting the data in main memory, any main memory sort algorithm can be chosen [44], e.g. quicksort [156], (main memory) merge sort (and its parallel version) or heapsort [157], which are well-known to be very fast.

In Figure 2.38 an example for merge sort is presented.

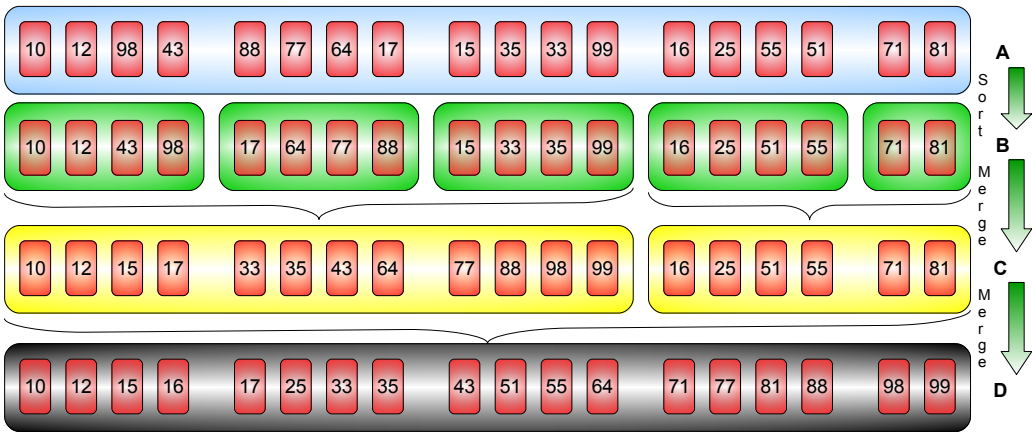


Figure 2.38: Using merge sorting on a set of 18 elements (A) by creating initial runs (B). Thereafter the runs are merged in multiple steps (C, D).

Initial, there is a set of 18 unsorted values in step A. First, the initial runs are created by setting a run length, dividing the original set by this length and sort these subsets. In this case the run length is set to 4 elements which leads to 5 sorted initial runs in step B. The first three initial runs are merged to a new runs thereafter. The same happens to the fourth and fifth initial runs leading to only two runs in step C. In the last step D the two runs are merged into one run which means that the set is sorted.

2.3.4.2 Replacement Selection

> *Replacement selection is a variant of the external merge sort algorithm (see Section 2.3.4.1) which both are competitors to PatTrieSort in Chapter 3.*

Another variant of external merge sort, called replacement selection [158], uses a heap to increase the length of the initial runs on average by a factor of 2. Whenever the heap is full, its root item is retrieved and written to the current run. In the heap, items, which still can be written into the current run (i.e., which are greater

than or equal to the last item written into the current run), are ordered before those items, which must be written into the next run (i.e., which are smaller than the last item written into the current run). The values of the items are the second order criterion in the heap. If the value of the root item is smaller than the last item of the current run, the current run is closed, and a new run is created and becomes the current run.

2.3.5 Tries and Variants

The PatTrieSort algorithm is introduced in Chapter 3 and relies on the use of patricia tries (see Section 2.3.5.2). The patricia tries are compressed versions of tries presented in Section 2.3.5.1.

So far, the introduced graph structure and its special form, the tree, in Section 2.3.2 are only used for index structures, like the B-tree and its variants in Section 2.3.3. The only exception are heaps introduced in Section 2.3.2.11 that allow the fast retrieval of the smallest/biggest item of a set but are not optimized for other operations, like searching the set. In this section, the introduced tries have a similar purpose as the heap. While binary search trees, B-trees and their variants allow optimized indexing, tries and heaps fulfill special purposes. Tries are especially used for strings and are further limited to this data type, but are very well optimized for this data type, just like the heap is optimized for retrieving the smallest/biggest item. In this section we look at the common tries and their compressed variants the patricia tries.

2.3.5.1 Tries

Tries are efficient in storing strings of characters and lead to other variants, like the patricia tries in Section 2.3.5.2.

Tries (e.g., [159]) serve as efficient data structure for storing strings of characters with common prefixes (e.g., see Figure 2.39 a)). Common prefixes of all strings, which are contained in the trie, are stored only once in the trie. For this purpose, the trie is a tree structure, in which each edge is labeled with one character, and the concatenation of the characters along the path from the root to a leaf is one of the stored strings in the trie. The edges of a parent node must contain different characters as labels, and are ordered according to the lexicographical order of their labels.

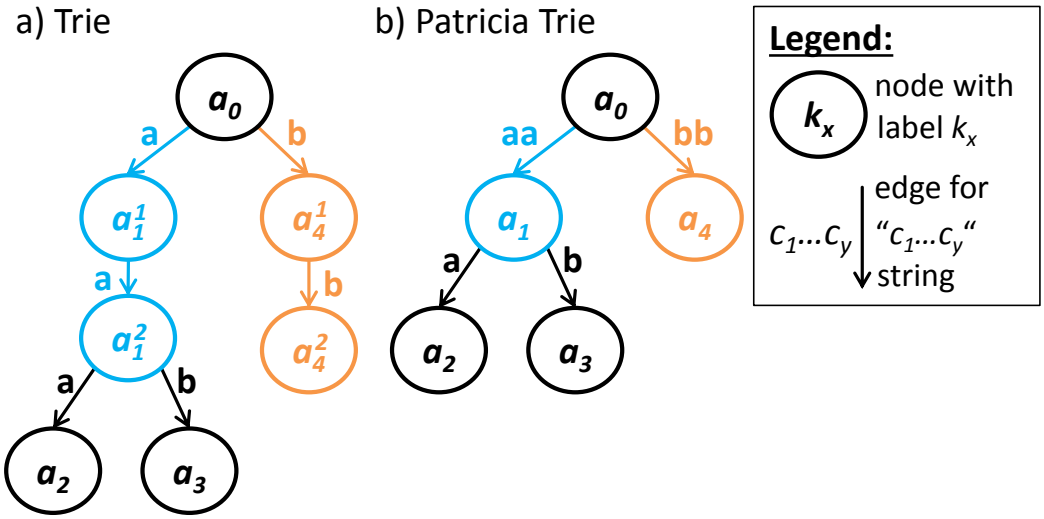


Figure 2.39: a) Trie and an equivalent b) patricia trie containing the strings "aaa", "aab" and "bb"

2.3.5.2 Patricia Tries

Patricia tries are important for the PatTrieSort algorithm in Chapter 3 and are compressed versions of tries introduced in Section 2.3.5.1.

Patricia tries are compressed tries (see Figure 2.39), where the edges can contain not only one character as label, but a string of several characters. In comparison to tries all nodes (except of the root node) with only one child are therefore melt together with their single child (and the edge between them is removed). The label of the incoming edge of the parent node is set to the concatenation of its previous label and the label of the old edge to the child. The order of the edges corresponds to the lexicographical order of their labels, such that searching within the patricia trie and therefore also update operations are more efficient. Note that the label can be an empty string (denoted by $\emptyset$) occurring if the patricia trie contains a string, which is a substring of another one in the patricia trie.

SW data typically consist of many strings with common prefix, as often IRIs (see Section 2.1.3.2) are used. Thus, patricia tries are the ideal data structure to store SW data in main memory.

3 A new String-based Sorting Approach - PatTrieSort

Sorting is one of the fundamental problems of computer science [149]. Sorting data not fitting into main memory is called *external sorting*. Although the sizes of the main memories of computers increase continuously, the data sets also become larger and larger (categorized as big data trend).

In the area of the Semantic Web (SW), there are masses of data with over 30 billions triples in nearly 300 datasets with over 500 million links between these datasets freely available to the public - thanks to the efforts of the linked data initiative [160]. Most of these datasets are too large to fit into main memory. Efficient processing of these data sets requires indexing approaches (e.g., [42, 43]), and sorting the data is one of the basic steps, which are typically done for index construction [161]. The most widely used index type in databases is the B^+-tree. B^+-trees can be built very efficiently from sorted data by avoiding costly node splitting (see [162] and extend its results to B^+-trees). Thus, the performance of index construction from scratch relies heavily on the techniques of data sorting.

The *external merge sort* [149] first generates initial runs of sorted data. An initial run is typically computed by reading as much data as possible from input into main memory, and sorting these data using main memory sorting algorithms. The alternative approach replacement selection [158] uses a heap to generate longer initial runs. Runs are written into external storage and merged afterwards to larger sorted runs until all the data is sorted.

Patricia tries (e.g., [159]) are space-efficient data structures for storing strings. Common prefixes of strings are stored only once in this data structure. We can retrieve the contained strings in sorted order by just one left-order traversal through its internal data structure in form of a tree. Especially the terms of Semantic Web data consist of many long common prefixes, such that using patricia tries for SW data offers obviously a good compression and low memory consumption.

Hence we propose a new external sorting algorithm *PatTrieSort* based on patricia tries: We generate the initial runs by inserting the input strings into patricia tries and we are swapping these patricia tries as initial runs to disk if they are

not fitting into main memory any more. For the merging phase, we developed a merging algorithm, the input of which are the swapped patricia tries. There are advantages of PatTrieSort in the phase of initial run generation as well as in the merging phase: Initial runs can be quite large because of the good compression of patricia tries, such that more strings can be held in main memory. In the merging phase, merging patricia tries instead of strings avoids the comparison of common prefixes, which is significantly more efficient.

The main contributions of this chapter include:

- a new sorting approach *PatTrieSort* as variant of external merge sort for sorting strings, where the initial runs are generated by using patricia tries. The initial runs are swapped in form of patricia tries to disk and are merged in a later processing step.
- a new algorithm for merging patricia tries used within the merge phase of the new sorting approach.
- a complexity analysis for the new merging algorithm in terms of runtime, I/O costs and memory consumption.
- a comprehensive performance evaluation and analysis of PatTrieSort compared to the other external merge sort variants using large-scale datasets with over 1 billion strings.

3.1 Basic Data structures and Sorting Algorithms

In this section we recall the used fundamentals and describe relevant related work.

3.1.1 Used Fundamentals

In Section 2.3.2.11 we introduce the heap as an efficient data structure to retrieve the smallest (or biggest) item from a set stored in the heap. The key operations in a tree are so called *bubble* operations. A new value, inserted into the most right leaf of the heap, will *bubble-up* every time it is smaller than the value of the father node and therefore switching place with it. On the other hand, by removing the smallest item and thereby replacing the root value with the right most leaf value a *bubble-down* process starts. Every time the new value is bigger than one or both of its child values this value is swapped with the smaller child value. After the smallest element in the heap is taken away, a succeeding insertion of a new element can be optimized by first placing the new element in the root and then performing a bubble-down

operation. This approach avoids the bubble-up operation by optimizing a pair of removing and insertion operations. We use this improvement during initial run generation of the sorting approach replacement selection (see Section 2.3.4.2).

The external sorting algorithm called (external) merge sort is introduced in Section 2.3.4.1. It is known as one of the best algorithms when the data exceeds the capacity of the main memory. First, subsets of the whole data set are sorted in main memory building so called *initial runs* and written back to the HDD/SSD. Thereafter, runs are merged into bigger runs until a single run holds the whole data set. In the merge process it is important to find the smallest item of the current runs, therefore a heap is suitable for this task.

A heap is also used in the replacement selection algorithm introduced in Section 2.3.4.2. This algorithm can increase the length of the initial runs on average by a factor of 2 by using a heap [158].

Tries are efficient data structures for storing strings of characters and are introduced in Section 2.3.5.1. These tree-like structures label each edge between two nodes with a unique character. A path from the root to a leaf therefore generates a string by linking the passed characters. Further, patricia tries are introduced in Section 2.3.5.2 which can add a character sequence to the edges instead of just single characters. This makes the storing of common, longer prefixes easier and more efficient. SW data typically consist of many strings with common prefix, as often IRIs (see Section 2.1.3.2) are used. Thus, patricia tries are the ideal data structure to store SW data in main memory. We show in the experiments that patricia tries are also ideal for sorting "normal" strings, not only SW data.

3.1.2 Further Related Work

While [163, 164] introduce basic sorting algorithms in more detail, [165, 166] are appropriate as surveys on external string sorting.

Some contributions utilize tries already for sorting (e.g., burstsort and its variants [167, 168]). In burstsort, a trie is dynamically constructed as strings are sorted, and is used to allocate a string to a bucket. For full buckets new nodes of the trie are constructed the leaves of which are again buckets. However, these algorithms work only in main memory for the purpose of lowering the rate of cache miss and are not developed for external sorting.

The main idea (and conclusion) of [169] is that it is faster to compress the data, sort it, and then decompress it than to sort the uncompressed data. This approach reduces disk and transfer costs, and, in the case of external sorts, cuts merge costs

by reducing the number of runs. The authors of [169] propose a trie-based structure for constructing a coding table for the strings to be sorted. In comparison, we do not use codes, but we also store *compressed runs* by storing the patricia trie containing all the entries of the run, which reduces the space on disk and in memory, too.

The contributions in [170] lay the foundations for a complexity analysis for I/O costs for the string sorting problem in external memory. Its contribution covers the discussion of optimal bounds for this problem under different variants of the I/O comparison model, which allow or not allow strings to be divided in single characters in main memory and/or on disk.

3.2 PatTrieSort

We propose a new sorting algorithm PatTrieSort as variant of external merge sort, where patricia tries are extensively used.

As external merge sort, PatTrieSort has two phases (see Figure 3.1): In the first phase initial runs (in form of patricia tries) are generated and swapped to disk. In the second phase the initial runs are merged until only one run (in form of a patricia trie) remains, which contains the sorted result. An optional step may be used to retrieve the sorted list of strings from the final run.

Within the first phase, the initial runs are generated by inserting the strings to be sorted into a patricia trie. If the main memory is full, the patricia trie is swapped to disk. It is important that the patricia tries are swapped to disk in a format, where the structure of the patricia tries remains, and the nodes of the patricia trie are stored by a left-order traversal through the patricia trie.

Within the second phase, the initial runs are merged and stored in a merged patricia trie. We present the merge algorithm in the next Section 3.2.1. The merge algorithm can have an arbitrary number of patricia tries as input, reads and processes all these input patricia tries by a left-order traversal, and stores the resultant merged patricia trie again in a left-order traversal. Hence the merge algorithm can merge as many patricia tries at once, as many nodes of patricia tries can be intermediately held in main memory. Typically there is only one merging step necessary even for huge data sets to be sorted, which further improves the speed of the overall sorting algorithm.

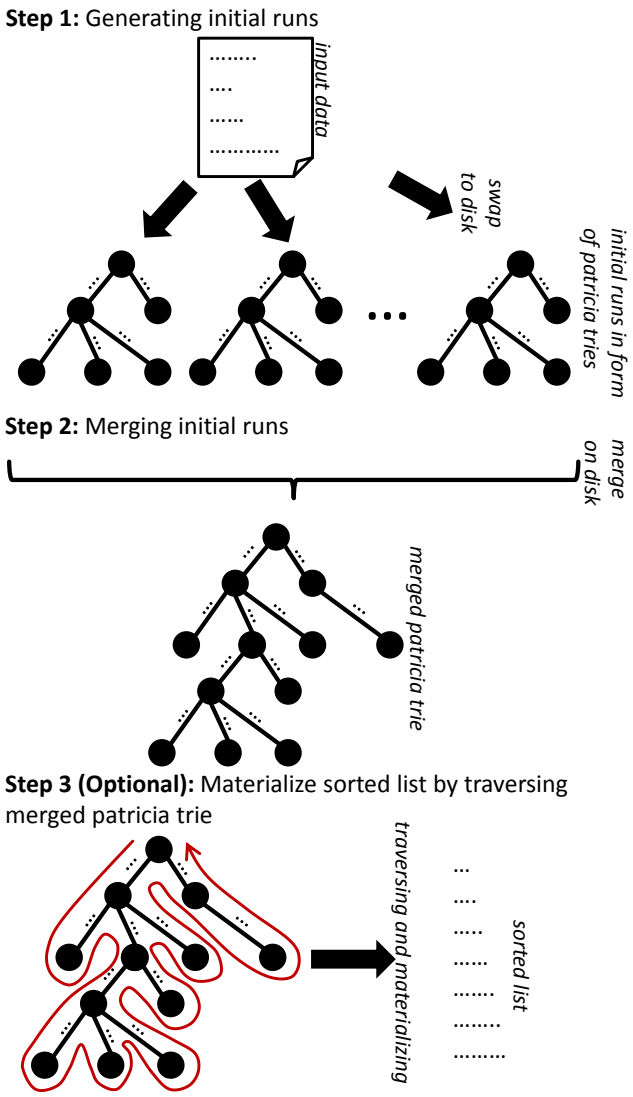


Figure 3.1: Overview of the main phases of PatTrieSort

3.2.1 Merging Patricia Tries

While the algorithm for inserting in a patricia trie is well-known [171], merging patricia tries, which we extensively use in PatTrieSort for merging the initial runs in form of patricia tries, has not been investigated to the best of our knowledge.

3.2.1.1 Examples of Merging Patricia Tries

We will start with two examples in Figure 3.2 and Figure 3.3 for merging patricia tries. Based on the examples, we will afterwards formulate the algorithm for merging patricia tries.

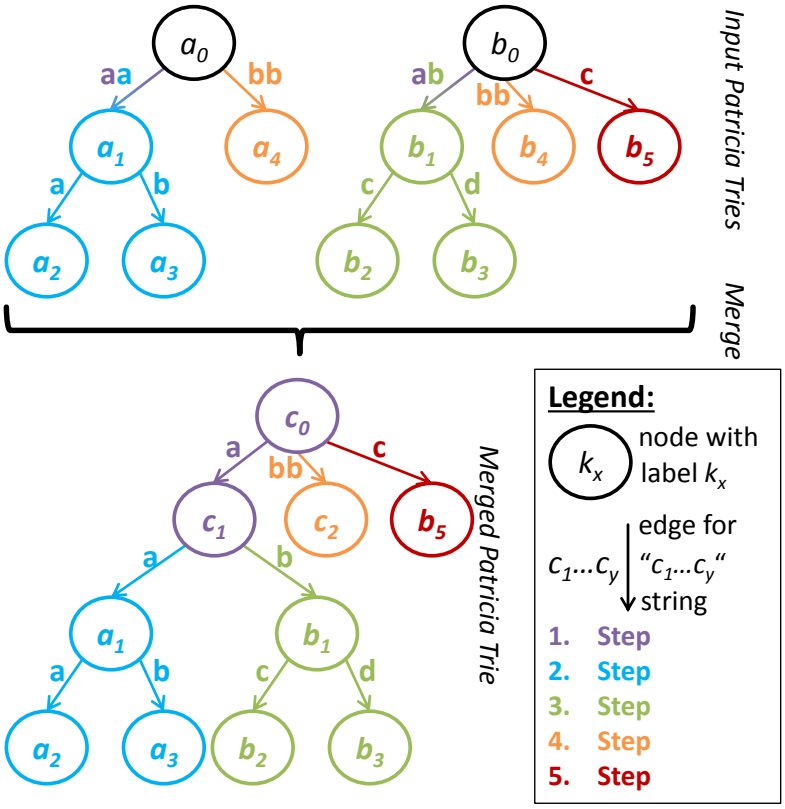


Figure 3.2: Example of merging 2 patricia tries

In Figure 3.2, two patricia tries are merged. The different nodes and edges to be considered in the input patricia tries as well as the nodes and edges created (or copied from an input patricia trie respectively) in the different steps are marked by different colors. In the first step the common prefix 'a' of the labels of the first edges (between a_0 and a_1 as well as between b_0 and b_1) in both input patricia tries are considered and lead to the nodes c_0 and c_1 and the edge labeled with 'a' between them. Because of the distinct postfixes 'a' and 'b' of the labels of the edges between a_0 and a_1, and between b_0 and b_1 respectively, the subgraph with nodes a_1, a_2 and a_3 is copied from the first input patricia trie in the second step, and the subgraph with nodes b_1, b_2 and b_3 is copied from the second input patricia

trie in the third step. The string 'bb' is contained in both input patricia tries. Hence, the nodes and edges for 'bb' are created only once in the merged patricia trie. If we want to support duplicates in our sorting algorithm, we need to hold a counter at each leaf node for representing the number of occurrences. In the latter case, we would need to compute the sum of the occurrences of 'bb' in both input patricia tries, and just store the sum in the counter of the merged patricia trie. Our merge algorithm will have only few additional steps when duplicates should be considered. In the last step, the remaining edge and node b_5 for the string 'c' is copied into the merged patricia trie.

If the patricia tries are held in main memory with a pointer structure and the input as well as the merged patricia tries are not modified any more after merging, we can speed up performance: we can avoid the costly operation of copying whole subtries and just use references to the subtrie in the corresponding input patricia trie. If we would allow to modify an input or the merged patricia trie, then the modification would lead to side-effects in the other patricia trie.

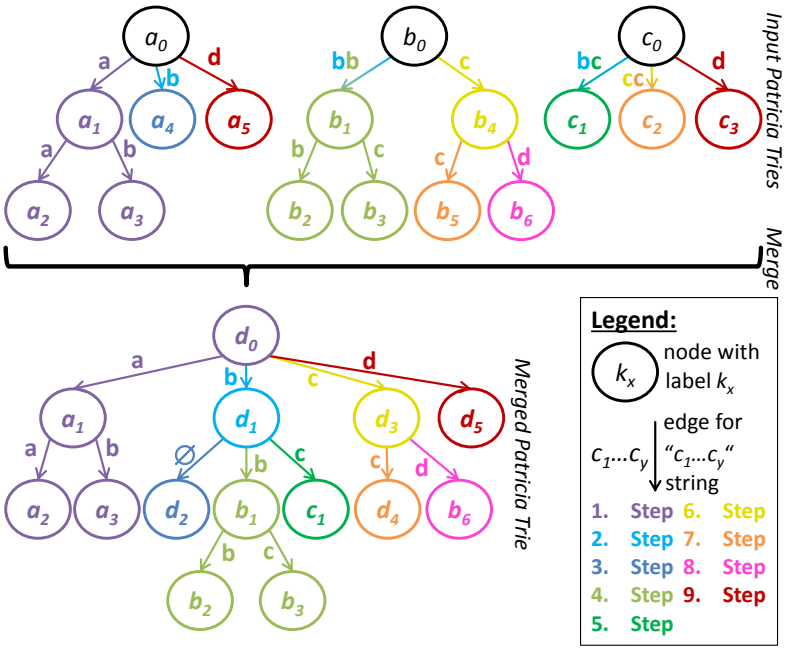


Figure 3.3: Example of merging 3 patricia tries

In Figure 3.3, three patricia tries are merged. There are analogous steps as for merging two patricia tries, we just have to consider the nodes and edges of an additional patricia trie. Merging even more patricia tries is also possible.

3.2.1.2 Merge Algorithm

One can imagine that the merge algorithm can be easily generalized to merge an arbitrary number of patricia tries. We only present the generalized merge algorithm in the following.

Algorithm 1: MainMergePatTries

Input : patricia tries $T_1 \dots T_n$, where $\forall i \in \{1, ..., n\} : T_i = (r_i, V_i, E_i)$ with r_i root node of the patricia trie i, V_i its set of nodes and E_i its set of edges

Output: Merged patricia trie

1 Create node r
2 return $MergePatTries(T_1 \dots T_n, (r, \{r\}, \emptyset), r)$

A patricia trie is represented in the merge algorithms by a triple (r, V, E), where r is its root node, V its set of nodes and E its set of edges. An edge $e \in E$ is represented by a triple (v_s, v_e, l), where $v_s, v_e \in V$. This edge is a directed edge from node v_s to node v_e and is labeled with a string $l = c_1 \dots c_m$, where c_i are characters. We use the notation $l[k]$ for the k-th character in l. We define $l[k]$ to return the empty character for $k \in \mathbb{N}$ if l is the empty string. We define the empty character to be the smallest character and use a function min to retrieve the smallest character from a given set of characters. The notation $\mid l \mid$ represents the number of characters in l. Hence, $l[k+1] \dots l[\mid l \mid]$ represents the substring of l after the first k characters (and is the empty string in the case that l contains only k characters). A node v is a leaf node if $\nexists v_e, l : (v, v_e, l) \in E$. For a leaf node v, $count(v)$ represents the number of occurrences of the string represented by the leaf node v. We extend the standard definition of patricia tries at this point in order to deal also with duplicates during sorting.

The main algorithm is Algorithm 1, which just creates a dummy patricia trie with one node for holding the resultant merged patricia trie later and calls Algorithm 2 with it (additionally with its root node and the input patricia tries).

Algorithm 2 first considers the left-most (unmarked) edges of all input patricia tries in line 3. Already considered edges will be marked later (in line 6).

The minimum first character (or the empty character respectively) among the labels of the left-most (unmarked) edges in E is computed in line 4. In line 5 all those edges are filtered from E, the labels of which start with the minimum character (or which labels are the empty string respectively), and stored in M. In line 7 the longest possible common prefix $c_1 \dots c_k$ of the labels in M are determined. Lines

Algorithm 2: MergePatTries

Input : • patricia tries $T_1 \dots T_n$, where $\forall i \in \{1, ..., n\} : T_i = (r_i, V_i, E_i)$ with r_i root node of the patricia trie i, V_i its set of nodes and E_i its set of edges
• patricia trie $R = (r_R, V_R, E_R)$ for the result
• root node r of the current subtrie in R

Output: Merged patricia trie

1 $jobs \leftarrow \emptyset$ // for collecting remaining pat. tries to store the merged patricia trie according to a left-order traversal
2 **while** *any* T_i *contains unmarked edge(s)* **do**
3 $E \leftarrow \{e \mid \exists v \in V_i, l : e = (r_i, v, l) \in E_i \wedge e$ is left-most unmarked edge in $T_i\}$
4 $c_1 \leftarrow min(l[1] \mid \exists v \in V_i, l : (r_i, v, l) \in E)$
5 $M \leftarrow \{e \mid \exists v \in V_i, l : (r_i, v, l) = e \in E \wedge l[1] = c_1\}$
6 Mark all edges of M
7 $P \leftarrow c_1 \dots c_k$, where $\forall (r_i, v, l) \in M : c_1 \dots c_k = l[1] \dots l[k] \wedge k$ is maximal
8 Create new node w
9 $V_R \leftarrow V_R \cup \{w\}$
10 $E_R \leftarrow E_R \cup \{(r, w, P)\}$
11 $Y \leftarrow \emptyset$ // for holding the remaining pat. subtries to be recursively merged
12 **foreach** $e = (r_i, v, l) \in M$ **do**
13 **if** *v is a leaf node* $\wedge \exists t^*, v^* : (t^*, \{t^*, v^*\}, \{(t^*, v^*, l[k+1] \dots l[|\ l\ |])\}) \in Y$ **then**
14 $count(v^*) \leftarrow count(v^*) + count(v)$
else
15 Create new node t
16 $V_X \leftarrow \{t\}$
17 $E_X \leftarrow \{(t, v, l[k+1] \dots l[|\ l\ |])\}$
18 Copy subtrie with root node v to V_X and E_X
19 $Y \leftarrow Y \cup \{(t, V_X, E_X)\}$
20 $jobs \leftarrow jobs \cup \{(Y, R, w)\}$
21 **foreach** $(Y, R, w) \in jobs$ **do**
22 $R \leftarrow mergePatTries(Y, R, w)$
23 **return** R

8 to 10 add a new node w and an edge to this new node with the longest possible common prefix $c_1 \dots c_k$ as label.

Line 11 initializes a data structure for holding the remaining patricia subtries to be recursively merged (later in line 22) and added as subtrie to w in the final merged patricia trie.

For each edge e in M, in line 13 our algorithm checks whether or not its target node is a leaf node and there exists already a patricia trie in Y, which contains only one edge with the same postfix as label: original label minus the common prefix $c_1 \dots c_k$. If it is the case, then the number of occurrences are added together to be the new number of occurrences for the string represented by this leaf node (see line 14). If duplicates should not be considered, this line 14 just may do nothing. Otherwise in lines 15 to 19, a patricia trie is added to Y containing an edge from a new node to the target node v of the considered edge e with the postfix (original label minus the common prefix) as label, as well as the whole patricia subtrie of v. Actually in a more efficient implementation, copying subtries in line 18 is not necessary and the same can be achieved by storing references to these subtries and accessing the original subtries when needed. Line 20 stores a job for merging the sub-patricia tries of Y. Afterwards, the algorithm continues to handle the remaining unmarked edges at line 3.

If all edges have been considered (and all edges are marked, line 2), the sub-patricia tries stored in *jobs* are merged (lines 21 and 22). Note that instead of holding the sub-patricia tries to be merged in *jobs*, they could already be merged in line 20. However, first collecting the sub-patricia tries in *jobs* has the advantage, that the nodes of the merged patricia trie can be stored in the order of a left-order traversal (by just storing the nodes and edges of the merged patricia trie in lines 9 and 10, and storing a mark for the start of a next node after leaving the loop of line 2 to 20). This has considerable benefits for stream processing, which requires the patricia trie in a stream to be read in the order of a left-order traversal (see next Section 3.2.1.3).

Finally, the merged patricia trie R is returned in line 23.

3.2.1.3 Dealing with Streams

Investigating Algorithm 2, we notice that the nodes and edges of the input patricia tries are accessed in the order of a left-order traversal. Hence, if the input patricia tries can be accessed in streams in left-order traversal, the merge algorithm works correctly also with streams. Furthermore, the merged patricia trie can be stored in a stream in left-order traversal.

These properties have several benefits:

- The merge algorithm can efficiently handle large patricia tries, which are serialized on disk in left-order traversal. Furthermore, the merged patricia trie can be easily serialized on disk in the order of a left-order traversal (and be the input for further merge steps).
- The merge algorithm can be deployed (as merge service) in distributed scenarios, where the patricia tries are sent and consumed in streams. The output stream of one merge service can be the input of another, leading to possibly complex merge trees in a distributed fashion.

3.2.2 Complexity Analysis

We will discuss the complexity of the merging algorithm in Algorithm 1 in terms of memory consumption, I/O costs and runtime in the following subsections. Let n be the number of input patricia tries, x the number of strings contained in all input patricia tries, l the maximum length of contained strings and c the size of alphabet used within the strings.

3.2.2.1 Memory Consumption

We have already observed that copying subtries in line 18 is not necessary and the same can be achieved by a delayed access to these subtries in succeeding recursion steps. With this observation, we can conclude the following memory consumption: Because of the recursion step in line 22 of Algorithm 2 and for each input patricia trie, only the nodes of a complete path from the root to a leaf node must be held in main memory. The maximum number of nodes in such a path from the root to a leaf node is l (but is typically much smaller for real-world data). However, for a more precise analysis we consider the number r of nodes of a complete path from the root to a leaf node. The maximum size of a single node is the maximum number of edges c of a single node multiplied with the maximum size p of the labels of the edges: $O(c \times p)$. For the merged final patricia trie, only one node must be temporarily held in main memory before it is written out. Hence, altogether the upper bound of the memory consumption is $O(n \times c \times p \times r)$, but is much less if we consider the properties of real-world data (see Section 3.2.2.4).

3.2.2.2 I/O Costs

Each node of the input patricia tries must be loaded only once into main memory under the condition that the nodes are held in main memory until the recursion

(in line 23 of Algorithm 2) is left again. This means that not whole subtries must be held in main memory, but only the ancestor nodes of the currently processed node, which has low memory footprint even for large datasets. Each node of the merged patricia trie is stored only once (lines 8 to 10). Under the assumption that optimal I/O costs are *loading* the input tries only *once* and *storing* the resultant patricia trie only *once*, we have optimal I/O costs for merging patricia tries.

3.2.2.3 Runtime

We will first consider each non-trivial step in Algorithm 2 before we discuss the overall complexity.

As all edges in a patricia trie node are ordered according to their labels, not all edge labels of the current nodes in the input patricia tries must be compared with each other, but only a part of them. Hence, the check of the loop condition in line 2 as well as the determination of the set E in line 3 can be done in time linear to the number of input patricia tries: $O(n)$. As at most one edge of each input patricia trie is added to E, the size of E is at most n. The determination of the minimum first character c_1 of the edge labels in E in line 4 is therefore also $O(n)$. For the same reason, the number of edges in M as well as the determination of M in line 5 are $O(n)$ (but for real-world data often much smaller). Marking all edges of M in line 6 is obviously in $O(n)$. The determination of the longest common prefix in line 7 is restricted by the size of M and the maximal size p of the labels of the edges and thus is $O(n \times p)$. Note that by intelligent coding, lines 3 to 7 could be done by iterating only once through the current edges of the input patricia tries (but this does not affect the complexity in O notation).

The loop from line 12 to 19 is iterated at most n times. Checking if an edge with the currently considered postfix to a leaf node in line 13 already exists in the patricia tries Y still to be merged, can be done in $O(p)$ (by choosing a good data structure for searching for the postfix, e.g. a hash table with the postfix as key). As already mentioned, copying subtries in line 18 is not necessary and the same can be achieved by a delayed access to these subtries in succeeding recursion steps. Hence, line 18 can be done in constant time (by storing a reference). Thus, the loop from line 12 to 19 is in $O(n \times p)$.

The loop from line 2 to 20 is iterated at most $O(c)$, as the single characters of the strings are from the alphabet with c different characters, and there are therefore at most c different common prefixes. Altogether the loop from line 2 to 20 is in $O(n \times p \times c)$.

We assume to have r recursion steps. The final merged patricia trie has at most x leaf nodes.

Hence, the overall runtime complexity is $O(n \times x \times c \times p \times r)$ and much less for real-world data (see Section 3.2.2.4).

3.2.2.4 Complexities for Real-World Data

For typical real-world data, c is not too large and can be seen as constant. Note that p and r depend on each other: as larger p is, as smaller is r and the other way around. Actually we can assume that l is in $O(p \times r)$ for typical real-world data.

Memory Consumption: For typical real-world data, the upper bound of the memory consumption is hence $O(n \times l)$.

Runtime: Assuming the properties of real-world data, the upper bound of the runtime is $O(n \times x \times l)$. Assuming that the sum L of the sizes of all strings is in $O(x \times l)$ for typical real-world data, we achieve a runtime complexity of $O(n \times L)$.

3.3 Experimental Analysis

We compare different variants of external merge sort: Our proposed approach *PatTrieSort*, *string merging*, *external merge sort* and *replacement selection*. The implementations of these sorting approaches are open source and publicly available as part of the LUPOSDATE project [44, 72].

3.3.1 Implementation Details

We varied the number of elements after which initial runs are swapped in *PatTrieSort*. For example, PatTrieSort with paramater 100 000 means that after 100 000 entries have been inserted into the patricia trie in main memory, this patricia trie is swapped to disk as initial run.

String merging is very similar to PatTrieSort. However, instead of writing the patricia trie as initial run, string merging writes the sorted list of strings as run. String merging writes the sorted list in a compressed way by leaving out common prefixes and writing instead an integer number for the number of characters in the common prefix with the previous entry. In experiments, this achieved a much better performance in comparison to writing the whole strings. In the merging phase, strings need to be merged instead of patricia tries. A heap is used for

merging the strings of typically a huge number of initial runs. Again we have chosen as parameter the limit of entries after which an initial run is swapped to disk.

External merge sort just uses an in-memory sorting algorithm to generate the initial runs and stores the initial runs in the same way as string merging. This approach has also benefits for parallel sorting of data streams [172], not only for sorting large data sets on a local machine. We have done experiments with several in-memory sorting algorithms like quicksort, LSD radixsort (specialized to string merging) and several variants of merge sort, and present here only the results with the best one of our experiments, a parallel merge sort algorithm with 8 threads for merging. Also for external merge sort we have chosen as parameter the number of entries, which are sorted in main memory and afterwards swapped as initial run to disk.

Replacement selection uses a heap for increasing the size of the initial runs (in typical cases by a factor of 2 [158] on average). Replacement selection with the parameter x means here that the sorting algorithm reserves a heap of height x (containing therefore $2^{x+1} - 1$ entries) for generating the initial runs. We use an optimized heap, which avoids a bubble-up operation in the two succeeding operations retrieving the smallest item of the heap and adding a new item to the heap: After the root item is taken away, instead of the standard operations (i.e., moving the last leaf node to the root, performing a bubble-down operation, adding the new item as the last leaf node and performing a bubble-up operation), we directly insert the new item at the root, and perform just one single bubble-down operation. This optimization significantly speeds up the replacement selection by avoiding a bubble-up operation.

Although we have used input data with over 1 billion entries, we only use 1 merging step for all approaches.

3.3.2 Configuration of the Test System

The test system for the performance analysis uses an Intel Xeon X5550 2 Quad CPU computer, each with 2.66 Gigahertz, 72 Gigabytes main memory, Windows 7 (64 bit) and Java 1.6. We have used a 500 GBytes SSD for reading in the input data and writing out the runs. The input data is read asynchronous using a bounded buffer. For saving space on the SSD, we compressed the input data by using BZIP2 [173]. Decompression is done on-the-fly during reading in the input data. For all approaches, we write the final run to disk and iterate once through the final run. We have run the experiments ten times and present the average execution times.

Table 3.1: String Length Statistics for Sort Benchmark

Number of Strings:	1,000,000,000
Average String Length:	98
Standard Deviation of the Sample:	0
Minimum String Length:	98
Maximum String Length:	98

3.3.3 Sort Benchmark

We have used the Sort Benchmark [174] for testing the performance of PatTrieSort in comparison to the other external merge sort algorithms. More concretely, we have used the input of PennySort [175], which is part of the Sort Benchmark, with 1 billion entries and measured the time for sorting these entries. The statistics of the string length (see Table 3.1) show that the dataset is homogeneous, as all strings have the same length of 98 characters.

It is obvious that larger initial runs lead to a faster merging phase. However, the generation of larger initial runs itself slows down performance: For PatTrieSort and string merging, inserting an entry into a larger patricia trie is slower. For external merge sort the in-memory sorting of more entries takes more time as well as for replacement selection inserting an entry into a larger heap.

Hence, there is an optimum for the amount of main memory reserved for initial run generation, after which sorting becomes slower again. For PatTrieSort and for string merging, this optimum is swapping after 8 million entries (see Figure 3.4), for external merge sort sorting of 2 million entries in-memory and for replacement selection using a heap of height 16 (with space for 131 071 entries).

Overall, PatTrieSort is the fastest among the external merge sort variants, followed by the traditional external merge sort, then string merging and finally replacement selection. Due to today's larger main memories, replacement selection does not save merging steps in comparison to the other sorting approaches, as all initial runs can be merged within one merging step. Merging a large number of patricia tries avoids comparing common prefixes, such that PatTrieSort is much faster than string merging and even gets ahead of external merge sort. PatTrieSort is at least 30 % faster than the other approaches (see Table 3.2).

The number of initial runs is the same and relatively small for PatTrieSort, string merging and external merge sort (see Figure 3.5). For their optimal parameters, we have 500 initial runs for external merge sort and 125 initial runs for PatTrieSort and string merging. Figure 3.6 shows the number of initial runs for replacement

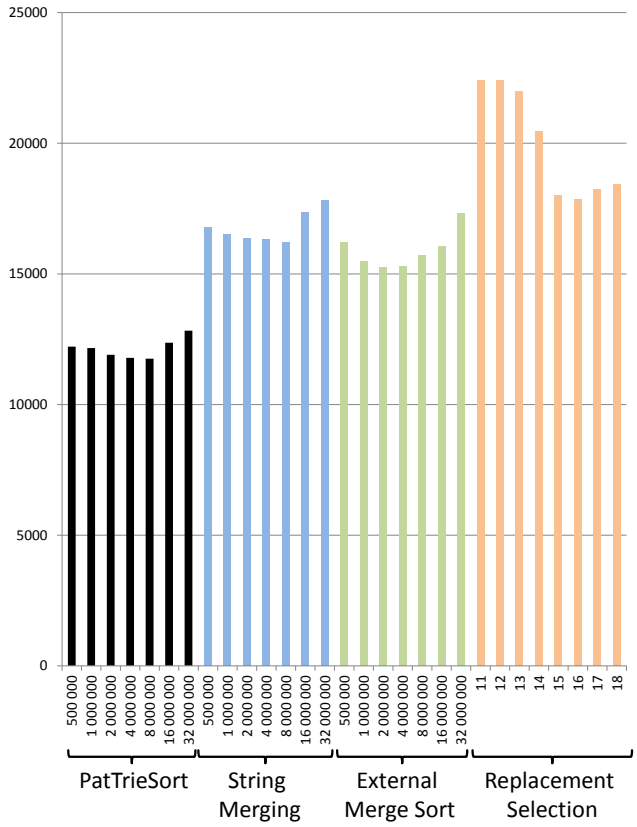


Figure 3.4: Results of Sort Benchmark in seconds

selection in relation to their parameters. Replacement selection with the best chosen height 16 of the heap generates 3817 initial runs. Although the merge phase is much slower for replacement selection for the optimal parameters, using larger heap heights becomes slower, as the slower insertion of entries in these larger heaps outweighs faster merge phases.

Storing patricia tries as initial runs results in some overhead in comparison to storing a sequence of sorted strings. This is reflected in the I/O-costs: See Figure 3.7 for the number of read bytes during sorting the data of the Sort benchmark, Figure 3.8 for the number of written bytes and Figure 3.9 for the total I/O-costs as sum of the read and written bytes. Hence, the I/O-costs are not completely the dominant factor in our considered sorting approaches. Not surprisingly is the number of read bytes a little bit higher than the number of written bytes, and the I/O-costs are (only slightly) lower when consuming more memory. Replacement Selection has competitive I/O costs only with high memory consumption.

Table 3.2: Speed Comparison of PatTrieSort with the other approaches for Sort Benchmark (only best chosen parameters)

x	$\frac{\text{Time of } x}{\text{Time of PatTrieSort}}$
String Merging	1.38
External Merge Sort	1.3
Replacement Selection	1.52

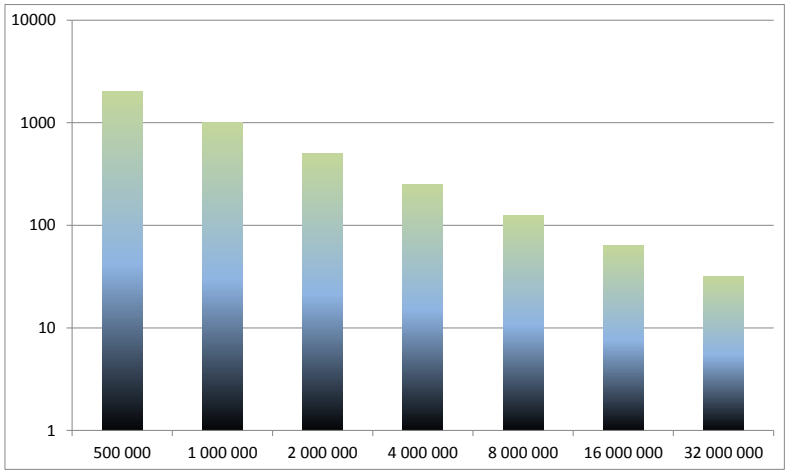


Figure 3.5: Sort Benchmark: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for PatTrieSort, string merging and external merge sort, which all have the same number of initial runs

3.3.4 Billion Triples Challenge

The overall objective of the SW challenge is to apply SW techniques in building online end-user applications that integrate, combine and deduce information needed to assist users in performing tasks [176]. For this purpose, in last years large-scale datasets were crawled from online sources which are used by researchers to showcase their work and compete with each other. The Billion Triples Challenge (BTC) dataset of 2012 [177] consists of 1 436 545 545 triples crawled from different sources like Datahub, DBpedia, Freebase, Rest and Timbl. BTC is perfectly suited as example for large-scale datasets consisting of real world data with varying quality and containing noisy data.

We have sorted the string representations of the three components (called subject, predicate and object) of the triples of the BTC data resulting in 4 309 636 635

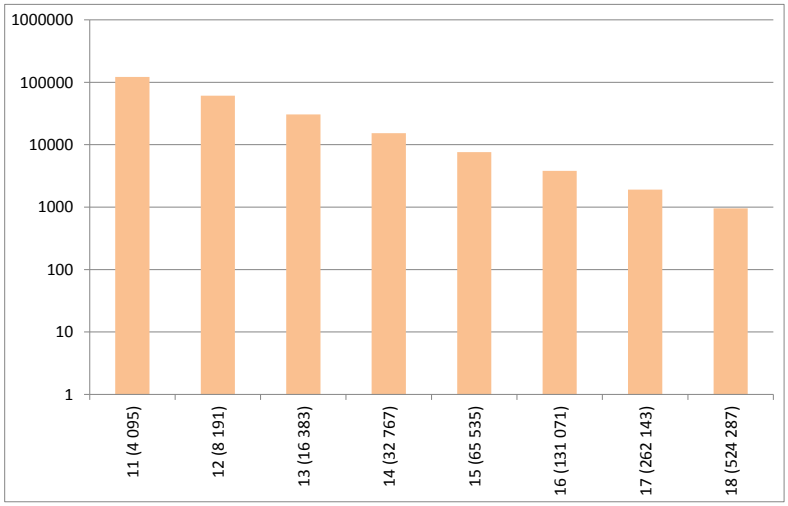


Figure 3.6: Sort Benchmark: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for replacement selection. h (s) at the x axis means reserving a heap of height h containing s $(= 2^{h+1} - 1)$ entries.

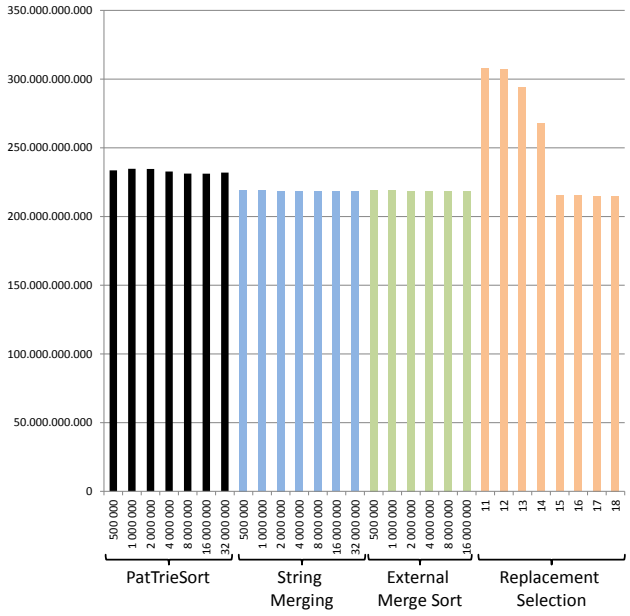


Figure 3.7: Sort Benchmark: Number of read bytes

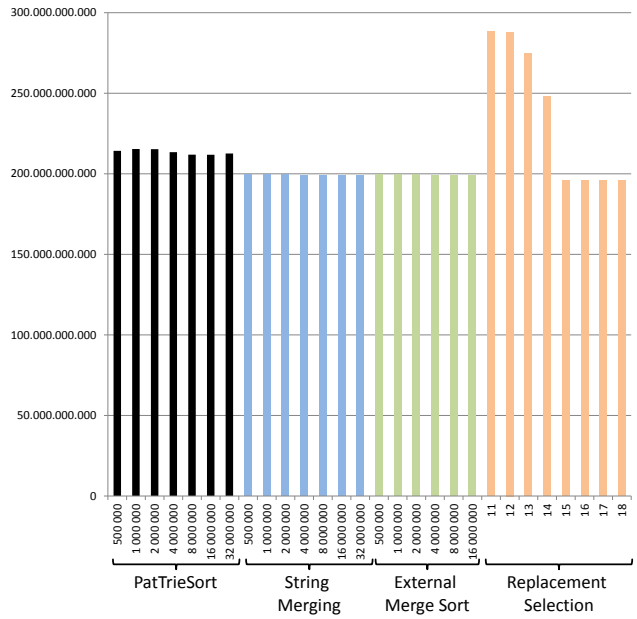


Figure 3.8: Sort Benchmark: Number of written bytes

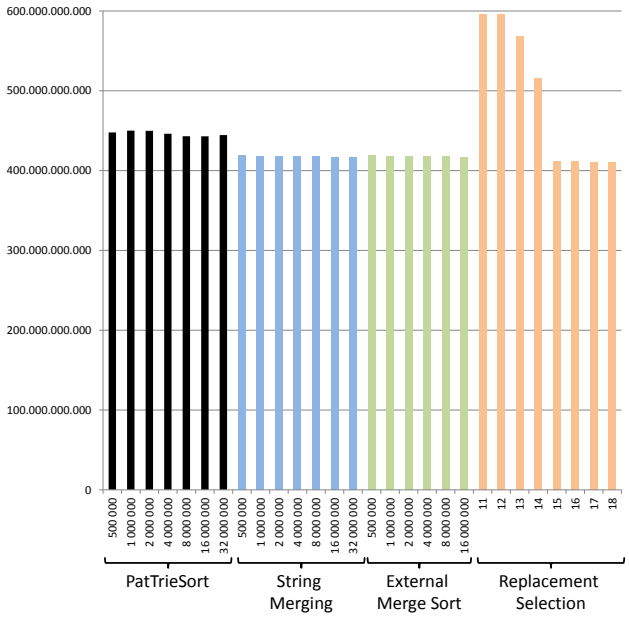


Figure 3.9: Total I/O costs of Sort Benchmark: Sum of read and written bytes

Table 3.3: String Length Statistics for Billion Triples Challenge of 2012

Number of Strings:	4,309,636,635
Average String Length:	55
Standard Deviation of the Sample:	73
Minimum String Length:	2
Maximum String Length:	335,085

Table 3.4: Speed Comparison of PatTrieSort with the other approaches for sorting BTC data (only best chosen parameters)

x	$\frac{\text{Time of } x}{\text{Time of PatTrieSort}}$
String Merging	3.76
External Merge Sort	4.05
Replacement Selection	6.13

strings to be sorted. The string length statistics (see Table 3.3) reflects the noisy nature of the input: While having an average length of 55 characters, the lengths vary between 2 and 335 085 characters. Sorting is one basic step when constructing a dictionary for the BTC data, which maps each component of a triple to a unique number. Using unique numbers instead of the space-consuming string representations greatly reduces space used for indices on disk, improves performance and lowers the memory footprint [41]. There are many duplicates among the strings to be sorted: Only 14 669 339 unique strings are among the strings of the BTC data.

SW data consists mainly of IRIs [31]. The syntax of IRIs corresponds to the one of URLs, i.e., they consist of many characters and many of them have a long common prefix (For more information on IRIs and URLs see Section 2.1.3.2). Hence, patricia tries are the ideal data structure to space-efficiently store IRIs in main memory. For this reason our proposed approach PatTrieSort performs extremely well (see Figure 3.10): The string merging approach is already 3.76 times slower than PatTrieSort, external merge sort 4.05 times and replacement selection 6.13 times slower (see Table 3.4).

Real-world data does not have a regular structure like synthetic data as in the case of the Sort Benchmark has. Hence the development of the execution times dependent on the main memory consumption is not so regular as well. However, the tendencies remain similar to those with synthetic data.

Because of the huge number of duplicates and their space-efficient storage in Pa-

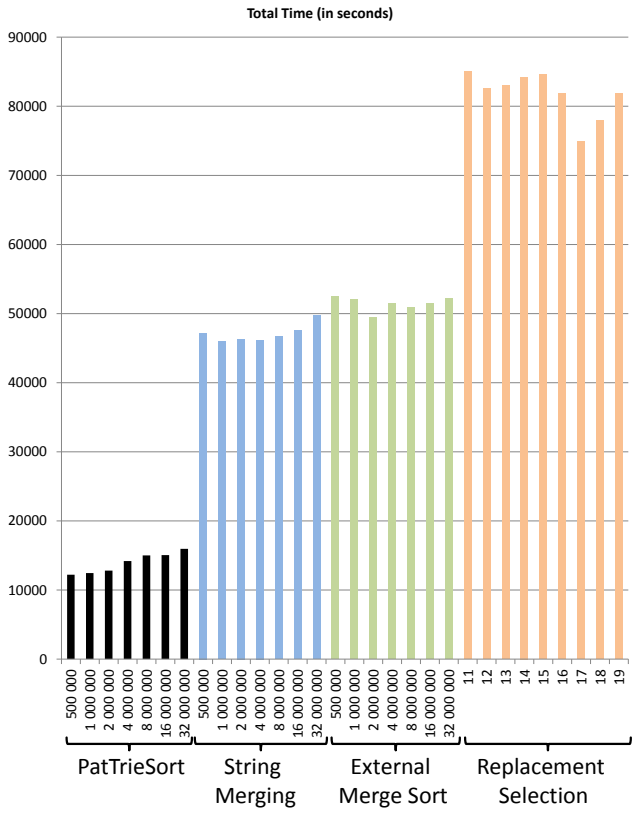


Figure 3.10: Results of sorting BTC data in seconds

tricia Tries, the number of initial runs is much lower for the PatTrieSort and string merging approaches compared to external merge sort (see Figure 3.11) and replacement selection (see Figure 3.12). Not surprisingly the factor

$$f := \frac{\#\text{Initial runs of PatTrieSort and String Merging}}{\#\text{Initial Runs External Merge Sort}}$$

even increases from about 7.5 to 9.6 when more memory is reserved for generating the initial runs (see Table 3.5). This is another reason for PatTrieSort beating the other approaches, although it is not the dominant factor (as string merging has the same number of initial runs as PatTrieSort). However, in the results of the Sort benchmark (see Section 3.3.3) external merge sort was faster than string merging, for BTC data it is the other way around.

Many duplicates lower the I/O-costs of the PatTrieSort approach, as fewer bytes need to be transferred between main memory and external storage: See Figure 3.13

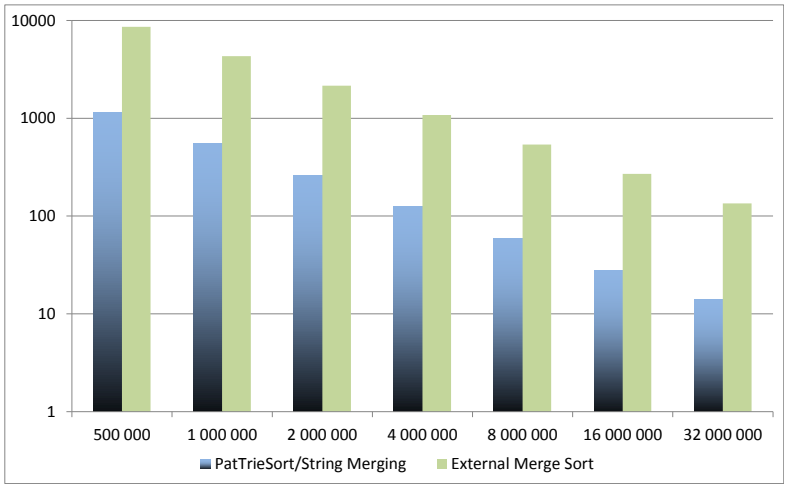


Figure 3.11: BTC: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for PatTrieSort, string merging and external merge sort

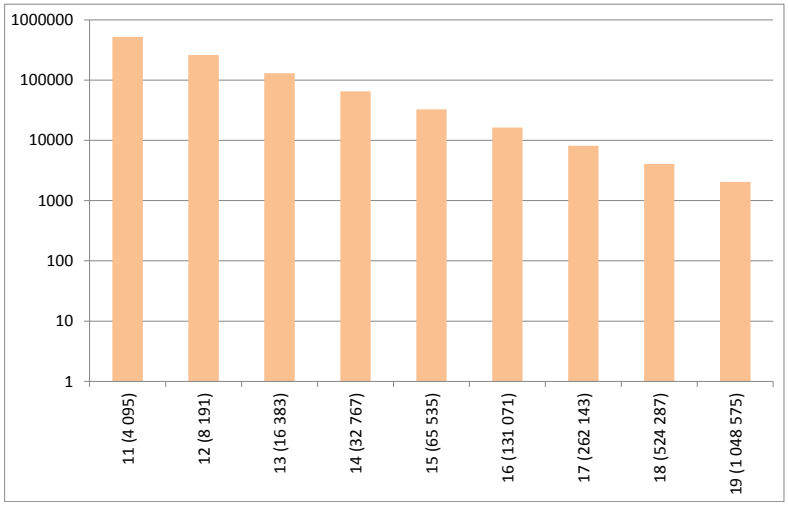


Figure 3.12: BTC: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for replacement selection. h (s) at the x axis means reserving a heap of height h containing s ($= 2^{h+1}-1$) entries.

for the number of read bytes during sorting the data of the BTC benchmark, Figure 3.14 for the number of written bytes and Figure 3.15 for the total I/O-costs

Table 3.5: BTC data: Comparing approaches by factor
$f := \frac{\#\text{Initial runs of PatTrieSort and String Merging}}{\#\text{Initial Runs External Merge Sort}}$

Entries inserted before swapping	Factor f
500 000	7.5
1 000 000	7.8
2 000 000	8.3
4 000 000	8.7
8 000 000	9.1
16 000 000	9.6
32 000 000	9.6

as sum of the read and written bytes. For real data with more irregular properties higher memory consumption leads to much less I/O-costs. Replacement Selection has again competitive I/O costs only with high memory consumption, but achieves the second best I/O-costs (after the PatTrieSort approach).

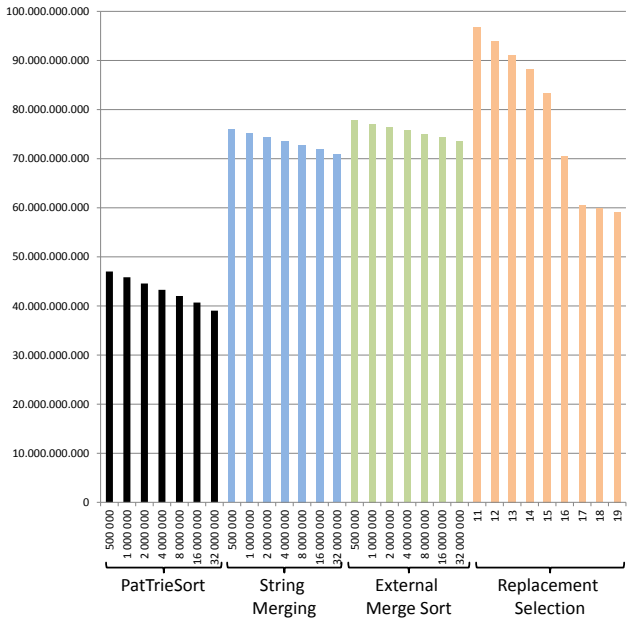


Figure 3.13: BTC: Number of read bytes

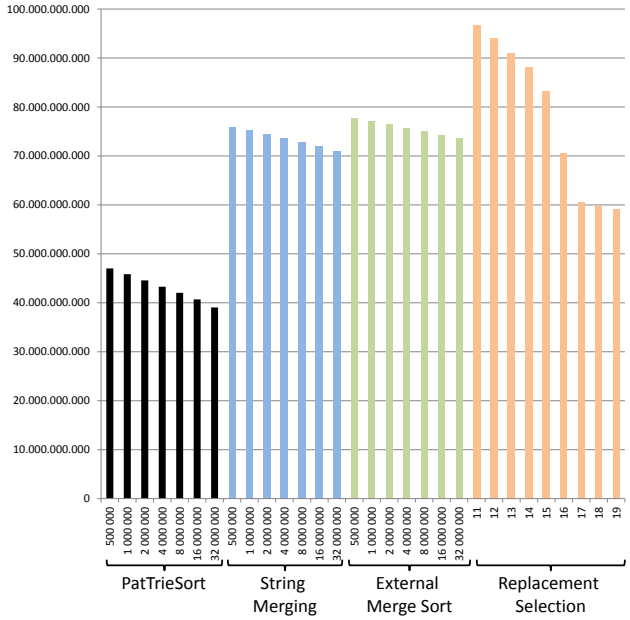


Figure 3.14: BTC: Number of written bytes

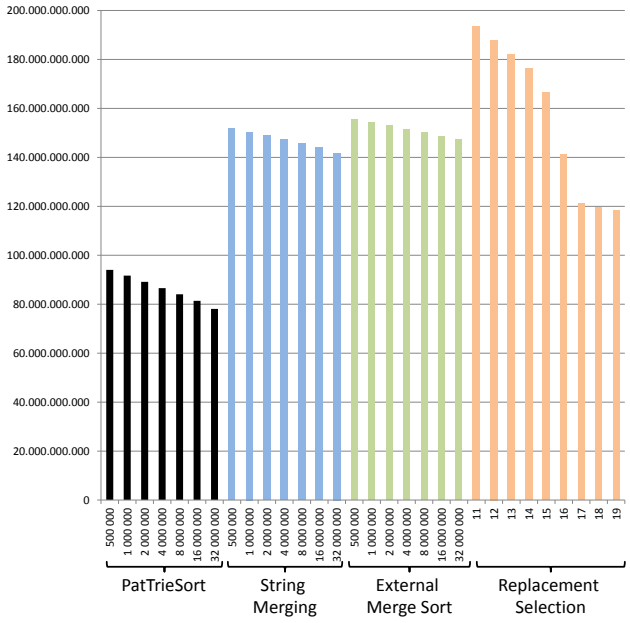


Figure 3.15: Total I/O costs of BTC Benchmark: Sum of read and written bytes

3.4 Summary and Conclusions

Patricia tries are one of the most space-efficient data structures for strings. Considering the size of main memory as limit we can store much more strings in main memory than just adding strings to lists or arrays. Adding a string to a patricia trie is efficient as well as we can iterate over the contained entries of a patricia trie in sorted order by traversing the tree of the patricia trie. Hence, the first idea is to utilize patricia tries for generating large initial runs of an external merge sort variant.

In a second phase, external merge sort merges already sorted initial runs until only one sorted run remains (which is the result). If strings with many common prefixes are merged, these common prefixes are compared unnecessarily often. Patricia tries store common prefixes only once. Hence the second idea for sorting strings is to store the (initial) runs as patricia tries and to integrate a merging algorithm based on patricia tries into external merge sort.

The complexity analysis shows best results for the new merging algorithm for patricia tries in terms of memory consumption, I/O costs and runtime. While we have optimal I/O costs, the used memory is linear to the number of patricia tries to be merged multiplied with the maximum length of contained strings, and the runtime depends on the factor of number of patricia tries and the total size of all strings.

The performance analysis highlights the new sorting algorithm for strings as the best one in its family of external merge sort algorithms. Especially sorting SW data like the large-scale BTC data consisting of many string with common prefixes benefits extremely from merging patricia tries in external merge sort with speed-ups higher than 3.7 compared to other external merge sort variants.

4 Constructing Large-Scale Semantic Web Indices for the Six RDF Collation Orders

In order to realize the vision of the Semantic Web (SW) [178], the World Wide Web Consortium (W3C)) recommends a number of standards. Among them are recommendations for the data model Resource Description Framework (RDF) [48] of the SW and the ontology languages RDFS [179] and OWL [180]. Ontologies serve as schemas of RDF and contain implicit knowledge for accompanying datasets. Hence using common ontologies enable interoperability between heterogeneous datasets, but also proprietary ontologies support the integration of these datasets based on their contained implicit knowledge. Indeed one of the design goals of the SW is to work in heterogeneous Big Data environments. Furthermore, the SW community and especially those organized in the Linking Open Data (LOD) project [160] maintain a Big Data collection of freely available and accessible large-scale datasets (currently containing approximately 150 billion triples in over 2,800 datasets [181,182]) as well as links (of equivalent entities) between these datasets.

Douglas Laney [183] coined the 3 V's characteristics of Big Data: Volume, Velocity and Variety[1]. Considering LOD datasets and the SW technologies, variety of data is basically dealt with the support of ontologies. The velocity of data can be only indirectly seen in the ongoing increasing growth of the dataset sizes in LOD, which are snapshots and are not typically processed in real-time. Hence there is a need for speeding up the processing of these SW datasets addressing especially the volume characteristics.

Some (but not all) of the LOD datasets are also available via SPARQL endpoints which can be queried remotely [181]. However, also for these valuable large-scale datasets (as well - of course - for those datasets which are only available as dumps) it is often desirable to import them to local databases because of performance and availability reasons. An incremental update to databases (inserting one triple after the other) will be too time-consuming for importing large datasets and hence

[1] Later the community proposed an increasing number of V's (e.g. [184] for 7 V's and [185] for the top 10 list of V's).

will be impractical. Instead, it is essential to construct indices from scratch in an efficient way. Efficient index construction approaches are also needed in many other situations, e.g. whenever databases need to be set up from archived and previously made dumps, because of e.g. a recovery (based on a backup) after a hardware crash or reconfigurations of the underlying hardware for upgrading purposes. Hence we propose an efficient approach to speed up the index construction for large SW datasets, which is one of the keys for paving the way for the success of SW databases in the Big Data age.

Today's efficient SW databases [41–44] use a dictionary to map the string representations of RDF terms to unique integer IDs. Using these integer IDs instead of space-consuming strings, the indices of those SW databases do not waste much space on external storage like SSDs and HDDs. Furthermore and more important, processing integer IDs instead of strings is more efficient during atomic query processing operations like comparisons and lowers the memory footprint such that more intermediate results can be hold in memory and less data must be swapped to external storage especially for Big Data.

SW databases like [41–44] further use 6 indices (called *evaluation indices*) according to the 6 collation orders of RDF triples. This allows a fast access to the data for any triple pattern having the result sorted as required by fast database operations like merge joins.

Overall this type of SW databases hence needs to construct the dictionary and 6 evaluation indices, which is very time-consuming. Our idea is to use a very fast algorithm to construct the dictionary, but to also smoothly incorporate the construction of the 6 evaluation indices into the generation of the dictionary. Furthermore, we want to use the parallelism capabilities of today's multi-core CPUs to further speed up the index construction.

The main contributions of this chapter include:

- a new index construction approach for SW data smoothly incorporating dictionary construction and evaluation indices construction by taking advantage of parallelism capabilities offered by today's multi-core CPUs, and

- a comprehensive performance evaluation and analysis of our proposed index construction approach using large-scale real-world datasets with over 1 billion triples.

4.1 Basic Data structures, Indices and Sorting Algorithms

In this section we recall the used fundamentals and introduce the further related work.

4.1.1 Used Fundamentals

In Section 2.3.3.2 the widely used database index structure known as B^+-tree is introduced. It is a variant of a B-tree (see Section 2.3.3.1) where all values are stored in the leaves. In this way the interior nodes maintain more keys lowering the height of the overall search tree. Since processing unsorted data leads to expensive node splitting operations the method of index construction using sorted input data is introduced in Section 2.3.4.

The B^+-tree index structure is used in a SW context which is introduced in Section 2.1. Therefore the data that is stored in the index structures are RDF triples consisting of *subject*, *predicate* and *object* (see Section 2.1.3.4). Also in Section 2.1.3.4 we show that mapping RDF terms to integer IDs lowers space requirements in the evaluation indices. This dictionary is storing the input RDF triples each of which with three integers instead of possibly large strings. Further, using difference encoding [41] and avoiding to store leading zero bytes additionally saves space. Furthermore, using IDs enables space-efficient representations of (intermediate) solutions lowering the memory footprint: more solutions can be processed before swapping to HDDs/SSDs starts increasing the overall performance. A simple example about how LUPOSDATE handles the connection between dictionary and index structures is presented in Section 2.1.5.

In Chapter 3 PatTrieSort is introduced and will be used in this chapter for sorting the RDF terms as preparing step for generating the dictionary. Therefore, also many fundamentals introduced in the former chapter are important in the current chapter. PatTrieSort uses patricia tries (see Section 2.3.5.2) for initial run generation in an external merge sort (see Section 2.3.4.1) variant for strings. These initial runs are usually larger compared to traditional external merge sort approaches consuming the same main memory size because of the compact representation of strings in patricia tries. Furthermore, PatTrieSort stores the initial runs as patricia tries instead of lists of sorted strings, as patricia tries can be efficiently merged in a streaming fashion having a superior performance in comparison to merging runs of sorted strings. We further propose to smoothly integrate mapping the RDF terms

of triples to IDs and sorting the triples according to the six collation orders of RDF in PatTrieSort in order to speed up the overall index construction.

4.1.2 Further Related Work

For the research fields of patricia tries and sorting algorithms many references in the previous chapter also apply in this chapter as related work for these fields (see Section 3.1.2). Therefore, they are not repeated in this section.

There are further engines besides LUPOSDATE that use 6 collation orders and dictionaries, like Hexastore [43] or RDF-3X [41]. Both extend the work of Abadi et al. [186]. In this work vertical partitioning is used to create tables of RDF data in a relational DBMS. For each predicate a table is created and sorted after the subject. This partitioning results in less unions and more but faster merge joins. Both, Hexastore and RDF-3X, reuse the vertical partitioning but extend it to the full 6 collation orders. This results in better performance for queries where the subject is not bound.

In [161], we already propose approaches to construct indices for the 6 collation orders of RDF. However, the proposed approaches in [161] do not construct a dictionary and the triples are stored with the string representations of RDF terms instead of IDs.

4.2 Constructing Indices according to 6 RDF Collation Orders

We focus on constructing indices for SW databases supporting dictionaries for mapping RDF terms to integer IDs (in order to lower space requirements and memory footprint resulting in higher performance for most query types) and retrieving presorted data according to the six RDF collation orders (in order to support as many merge joins as possible). The main tasks for index construction are hence a) dictionary construction, b) mapping the RDF triples to ID triples using the integer IDs of the dictionary and c) constructing the evaluation indices according to the six different collation orders of RDF. The naive way is to separate these three main tasks in different phases during index construction, and (even worse) to separate the construction of the six evaluation indices into 6 different sub-phases.

In contrast to this naive way, we propose to smoothly integrate all these tasks in order to avoid unnecessary I/O workload and computations.

We propose to apply a sophisticated process of 9 steps, which we describe in the following subsections in more detail. Figure 4.1 contains an overview of the overall process including an example.

4.2.1 Building patricia tries and mapping triples to temporary IDs

According to the former Chapter 3 utilizing patricia tries for sorting the RDF terms of large-scale datasets is highly efficient. In more detail in Chapter 3 we propose the PatTrieSort approach, which constructs patricia tries in main memory, rolls the full patricia tries out into external storage (but storing them as patricia tries) and finally merges them by a merge algorithm specialized to patricia tries.

Our main idea is to smoothly integrate the remaining tasks into PatTrieSort: the mapping of the RDF triples to ID triples and sorting them according to the six collation orders. We also want to save as much memory space as possible, such that more triples can be processed block-wise in main memory leading to larger runs of sorted ID triples stored in external storage. Hence we propose to use ID triples as early as possible: After reading a triple according to PatTrieSort we first add the string representations of its RDF terms (i.e., subject, predicate and object) into the main-memory patricia trie (see step 1 in Figure 4.1).

In order to avoid several reads of the RDF data, we propose to hold also the triples in main memory, sort them before the main-memory patricia trie is rolled out according to the six collation order and finally write these sorted runs according to the six collation orders to external storage. However, storing the string representations of the RDF terms for each triple is too space-consuming, which abolishes the advantage of PatTrieSort of space-efficiently storing strings. Hence we propose to transform the RDF triple to an ID triple right after reading a triple, which greatly saves main-memory by storing only 3 integers instead of 3 strings for each triple. However, right after reading a triple we have not the IDs of its RDF terms (because the dictionary has not been built so far). Hence we propose to map the RDF terms first to a temporary ID, which we maintain in the currently constructed patricia trie acting as main-memory key-value store with the RDF terms as keys and the temporary IDs as values: We just check if the RDF term to be mapped is already included in the currently constructed patricia trie. In the case that the RDF term is included we use its previously assigned temporary ID. In the other case we assign a new temporary ID (corresponding to the current number of entries in the patricia trie) for the RDF term and store this temporary ID at the leaf of the RDF term in the patricia trie.

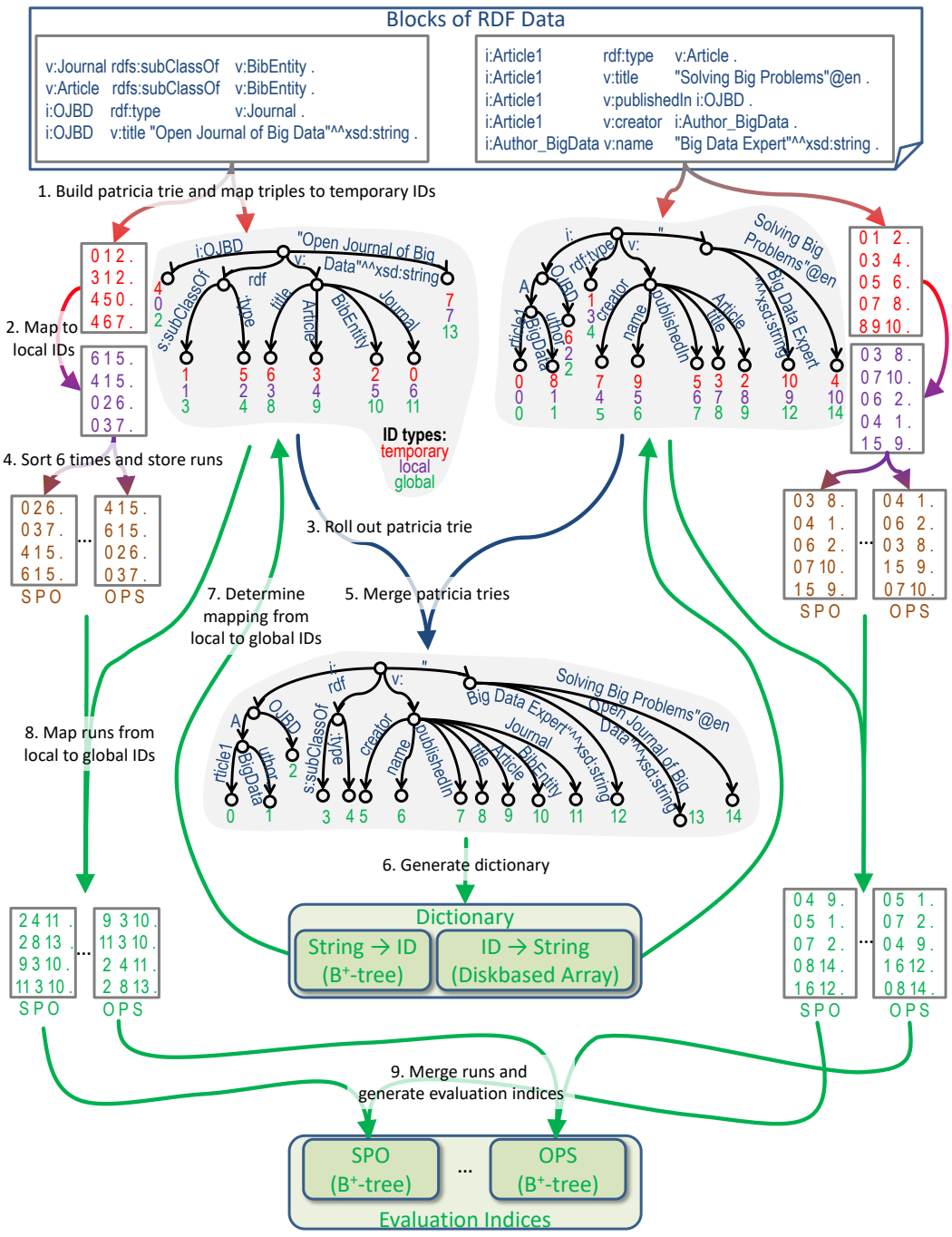


Figure 4.1: Overview of the index construction process with an example

4.2.2 Mapping to local IDs

Once the current block of triples is completely read in, our approach maps the temporary IDs (built according to the occurrences of the RDF terms) of the loaded triples to local IDs (built according to the lexical order of the RDF terms) (see step 2 in Figure 4.1). For this purpose, we determine a mapping of the temporary IDs to the local IDs by just constructing a one-dimensional array, where the index in the array corresponds to the temporary ID and the array value at the index to the local ID (which corresponds to the position in the sorted sequence of RDF terms). This one-dimensional array can be constructed during one in-order traverse through the patricia trie of RDF terms.

In one of the following steps, we want to already generate initial runs after sorting the loaded triples, which are later (mapped to global IDs and) merged for determining a complete sorted sequence of triples. For this purpose, the relative order between local IDs and global IDs must be the same, i.e., if a local ID id_1 is smaller than another one id_2 ($id_1 < id_2$), then the same order-relation must hold for the corresponding global IDs ($global(id_1) < global(id_2)$ with $global$ is a mapping from local to global IDs). The simplest way is to use the lexical order of the original RDF terms (which is always implicitly given and can be determined in different blocks of triples independently from each other), although in general it is only important that the dictionary is constructed according to any order. Indeed this initial order of IDs according to the lexical order of the original RDF terms will be destroyed after updates on the constructed indices, which typically introduce new IDs without considering the lexical order of original RDF terms in order to avoid a costly renumbering of the old IDs.

4.2.3 Rolling out patricia trie

Before we sort the triples with local IDs according to the six collation orders of RDF, which consumes 6 times more main memory for maintaining the triples, we roll out the current patricia trie to free up main memory (see step 3 in Figure 4.1). It is important that the patricia trie is swapped to external memory (like HDD or SSD) in a format, where the structure of the patricia tries remains, and the nodes of the patricia trie are stored by a left-order traversal through the patricia trie. In this way the patricia trie does not need to be constructed again and can be directly reused (in the later step for mapping the local IDs to global ones). Furthermore, besides having a very compact representation of the contained strings, the patricia trie in this form can be processed in a streaming fashion [1], which is especially important for a later merging of all the patricia tries for generating the dictionary.

4.2.4 Sorting runs of triples according to 6 collation orders

In this 4^{th} step (see Figure 4.1) the loaded triples with local IDs are sorted according to the 6 collation orders. Our idea is to use the properties of RDF and of the local ID triples for further improving the processing speed. We observe the following properties:

- We have to sort triples composed of integers with a limited, relatively small domain (from 0 to $n-1$, where n is the number of distinct RDF terms in the current block of RDF triples, i.e. the number of contained RDF terms in the previously rolled out patricia trie).
- The 6 collation orders of RDF have 3 primary collation order criteria (subject, predicate and object), each of which having 2 secondary collation order criteria (subject, predicate and object without the primary collation order criterion).

Among the sorting algorithms Counting Sort [149, 187] is a specialized linear runtime algorithm working on keys with small domain. Hence we propose to use Counting Sort for sorting the local IDs triples according to the 3 primary collation orders (subject, predicate and object). Counting Sort has another advantage: It already determines the borders of blocks of triples with the same primary key in the sorted output. Hence we can use these borders to sort blocks of triples with the same primary key according the secondary and tertiary keys with a fast standard sorting algorithm like quicksort [156].

We can easily parallelize sorting by

- dealing with the 3 primary collation orders in parallel, and
- sorting the blocks of triples with the same primary key in parallel.

Afterwards the 6 different runs for the six collation orders can be swapped to external memory (like HDD or SSD). Inspired by [41,42] we use difference encoding in order to compress the data for saving space and increasing the speed of I/O operations. Difference encoding does not store common components of the current triple compared to the previous one. Furthermore, for the different components of the triple, we only store the difference to the previous triple, which is a smaller number. We also only store as many bytes of an integer ID (its difference to the previous triple respectively) as necessary (avoiding to store leading zeros). This lightweight compression scheme is fast to compute, but saves many I/O operations, which overall hence saves also computation costs. All the other blocks of RDF data are now processed in the same way (steps 1 to 4 in Figure 4.1).

4.2.5 Merging patricia tries

In order to construct a uniform mapping from RDF terms (in form of strings) to global IDs (in form of integers) and vice versa (i.e., the dictionary), we need to merge all the generated patricia tries (see step 5 in Figure 4.1). We propose to use the merge algorithm of Chapter 3 working directly on patricia tries, being very efficient and having some extraordinary advantages: According to Chapter 3, the merge algorithm for patricia tries can have an arbitrary number of patricia tries as input, reads and processes all these input patricia tries by a left-order traversal, and stores the resultant merged patricia trie again in a left-order traversal. Hence the merge algorithm can merge as many patricia tries at once, as many nodes of patricia tries can be intermediately held in main memory. Typically there is only one merging step necessary even for huge datasets to be sorted, which further improves the speed of the overall algorithm.

4.2.6 Generating Dictionary

After the patricia tries have been merged, the dictionary can be generated (see step 6 in Figure 4.1). The dictionary consists of two indices: The first index is a B^+-tree for the mapping of the RDF terms (in form of strings) to the global IDs (in form of integer values). The second index maintains the other mapping direction from the global IDs to the RDF terms. For fast access, this index is stored in a file-based array of pointers addressing the strings of RDF terms in another file. In this way there are only two disk accesses necessary for retrieving the RDF term of a global ID: We can look up the RDF term in the second file at the position addressed by the pointer stored at the position calculated by multiplying the global ID value with the pointer size in the first file.

Both indices, the B^+-tree (see [162] and extend its results to B^+-trees, or see [44]) as well as the file-based array can be generated by iterating one time through the sorted sequence of RDF terms.

Generating the dictionary indices can be parallelized by generating the two indices in parallel.

4.2.7 Determining mapping from local to global IDs

After the dictionary has been generated the next step is to determine a mapping from the local IDs of the runs to the global IDs of the dictionary (see step 7 in Figure 4.1). As dictionary lookups are expensive especially for masses of lookups,

we want to avoid dictionary lookups as much as possible. For this reason, we load the rolled out patricia tries for each run and traverse one time through the patricia tries. For each element in a considered patricia trie, we look up one time in the dictionary to retrieve its global ID. At the same time we build the mapping by using an one-dimensional array, where the index corresponds to current position in the patricia trie (which is the local ID) and we set its value to the retrieved global ID. Note that the one-dimensional array has a range from 0 to $n-1$, where n is the number of elements contained in the patricia trie. It is hence a very compact representation of the mapping. After traversing the patricia trie, we can free up its resources as we will only work with the mapping (stored in the one-dimensional array) and do not need the patricia trie of the run any more.

4.2.8 Mapping runs from local to global IDs

In the 8^{th} step of Figure 4.1 we use the mapping (in form of an one-dimensional array) determined in the previous step to go through the corresponding run and replace all local IDs with their global ones by looking up the mentioned one-dimensional array at the index position of the local ID. As not all mappings and runs fit into main memory, we again swap the runs (this time containing global IDs) to external memory.

4.2.9 Merging runs and generating Evaluation Indices

In the last step and for each collation order of RDF, we merge all its runs containing global IDs (see step 9 in Figure 4.1). As for large-scale datasets many runs must be merged, we use a heap of the current triples of each run (i.e., the heap has the size of the number of runs). Determining the next smallest triple can be done in logarithmic time (in relation to the number of runs) when using a heap in comparison to a linear time for a linear search through all current triples of the runs. During merging, we always remove the smallest triple of the remaining ones located in the root of the heap and insert the next triple of that run, which is the same as the one of the removed triple. We optimize this pair of removing and insertion operations by first placing the new triple in the root and then performing a bubble-down operation in the heap, which avoids one bubble-up operation.

For each collation order of RDF while merging, we can build the evaluation index (in form of a B^+-tree) in one pass through the corresponding final merged run (see [162] and extend its results to B^+-trees, or see [44]).

We can parallelize this step by merging and generating the evaluation indices in parallel for each of the six collation orders.

4.3 Experimental Analysis

We compare the runtime of our proposed index construction approach for different parameters. The implementation of the index construction approach is open source and publicly available as part of the LUPOSDATE project [44,72]. We describe the configuration of the test system in Section 4.3.1, the used dataset in Section 4.3.2 and the experimental results and analysis in Section 4.3.3.

4.3.1 Configuration of the Test System

The test system for the performance analysis uses an Intel Xeon X5550 2 Quad CPU computer, each with 2.66 Gigahertz, 72 Gigabytes main memory, Windows 7 (64 bit) and Java 1.8. We have used a HDD for reading in the input data and a 500 GBytes SSD for writing out the runs. The input data is read and parsed asynchronously (by 8 threads) using a bounded buffer. For saving space, we compressed the input data by using BZIP2 [173]. Decompression is done on-the-fly during reading in the input data. We have run the experiments ten times and present the average execution times.

4.3.2 Billion Triples Challenge

The overall objective of the SW challenge is to apply SW techniques in building online end-user applications that integrate, combine and deduce information needed to assist users in performing tasks [176]. For this purpose, in last years large-scale datasets were crawled from online sources which are used by researchers to showcase their work and compete with each other. The Billion Triples Challenge (BTC) dataset of 2012 [177] consists of 1 436 545 545 triples crawled from different sources like Datahub, DBpedia, Freebase, Rest and Timbl. BTC is perfectly suited as example for large-scale datasets consisting of real world data with varying quality and containing noisy data.

Although we have used input data with over 1 billion entries, we only use one merging step for merging the patricia tries.

4.3.3 Results

Figure 4.2 contains the total construction times for importing the whole BTC dataset of 2012 [177]. As explained in Section 4.2, the RDF triples are block-wise processed during index construction. We have varied the sizes of RDF blocks and present the results when using block sizes of 1, 5, 10, 25, 50 and 100 million triples. Overall and for our test system we achieve best results for an RDF block size of 10 million triples.

For investigating why using smaller or larger block sizes slows down index construction, Figure 4.3 contains the single times for the different phases of index construction [2]. While the processing times of most phases remain about the same for different block sizes or are not significantly compared to the total index construction time, building the patricia tries becomes slower for larger block sizes and mapping initial runs to global IDs faster, i.e. we have two contrary trends for the processing times of index construction phases for larger block sizes. We also observed in [1] that larger block sizes increase the times for building the patricia tries, which may be explained by more complex computations necessary to add a string to a fuller patricia trie in comparison to the insertion into emptier patricia tries. For mapping initial runs to global IDs we need more lookups in the global dictionary (for determining the global ID of a given RDF term) for smaller RDF blocks, as often used RDF terms must be looked up for many RDF blocks containing these RDF terms. Larger block sizes hence decrease the number of lookups for duplicated RDF terms, as each RDF term is only looked up at most once for each block (see Section 4.2.7).

4.4 Summary and Conclusions

The SW with its large-scale datasets (e.g., those collected in the Linking Open Data (LOD) project [181, 182]) cries for efficient index construction approaches in order to build indexes from scratch for succeeding big data analytics.

In this chapter we propose an efficient index construction approach in order to generate a dictionary (i.e., a mapping from the string representation of RDF terms

[2] In addition to the evaluation indices, LUPOSDATE constructs indices (called *histogram indices*) with the help of which LUPOSDATE efficiently determines histograms of triple pattern results [44]. Like constructing the evaluation indices, the construction of the histogram indices can be done in one pass through the corresponding final merged run, and roughly takes about the same time as constructing the evaluation indices. Please note that in our presentation of the experimental results the times to construct the histogram indices are included in the times to construct the evaluation indices.

to unique integer IDs and vice versa) and evaluation indices according to the 6 collation orders of RDF, which are widely used indices in SW DBMS. We describe a sophisticated process, where the generation of the evaluation indices is smoothly integrated into the dictionary construction in order to save I/O costs as well as computation costs. Our proposed approach is parallelized in many phases of the index construction to take advantage of the today's multi-core CPUs. Furthermore, the data is compressed as early as possible in order to lower the main memory footprint and process bigger blocks of triples in main memory (by using patricia tries and IDs as early as possible), and transfer less bytes to and from external storage (by natively storing patricia tries and applying difference encoding during storing the runs). The proposed index construction approach is also designed to apply very efficient algorithms like Counting Sort working on small domains, which we especially created for this purpose by introducing local IDs for each block of triples.

A comprehensive experimental analysis shows the practical feasibility of the proposed approach by analyzing the execution times for importing over 1 billion triples of the overall construction approach.

Our major contribution to the big data area is hence on volume since our proposed approach can manage, process and organize a large quantity of data, validated by system design and experimental results.

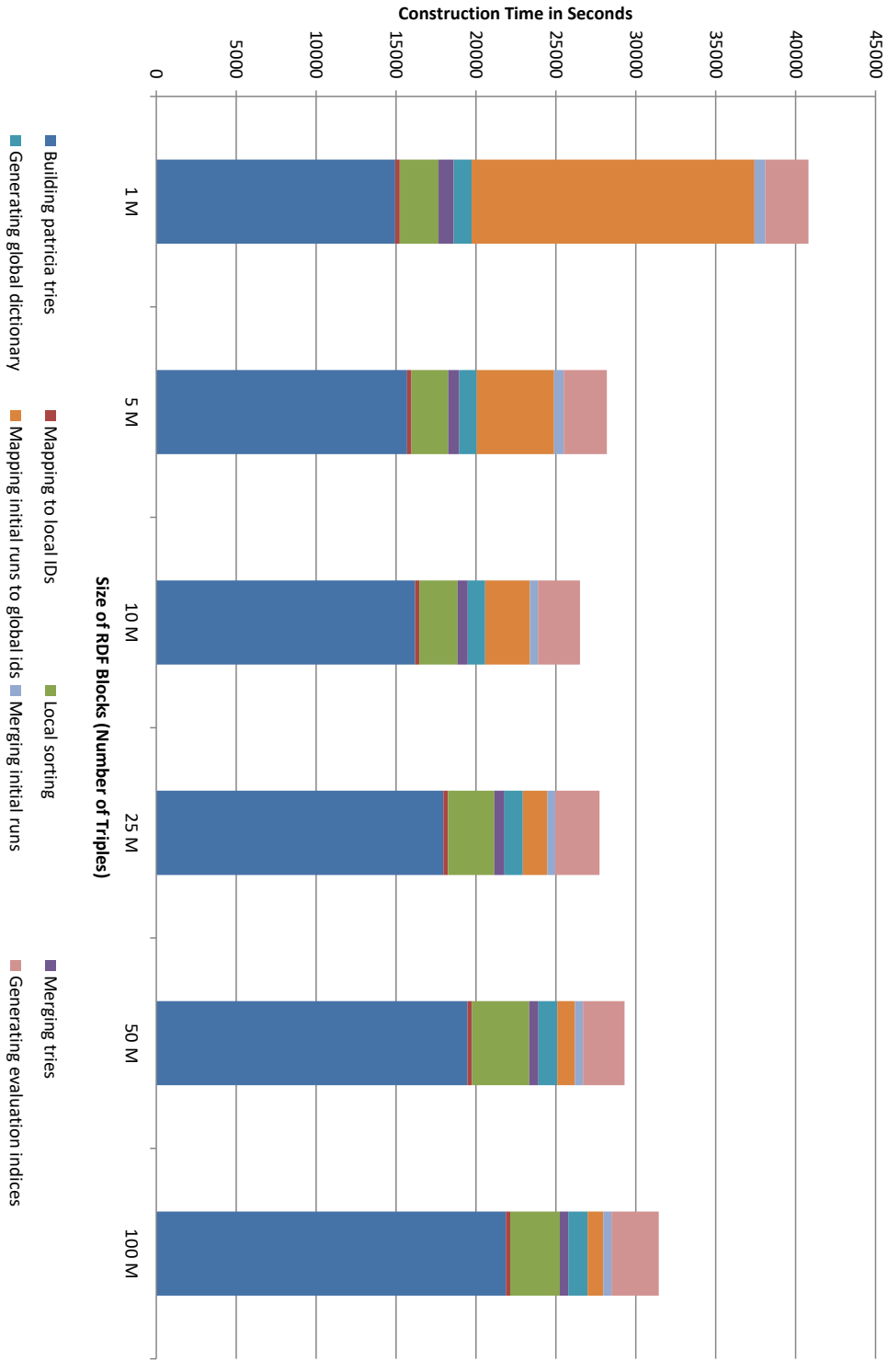


Figure 4.2: The total index construction times for different sizes of RDF blocks

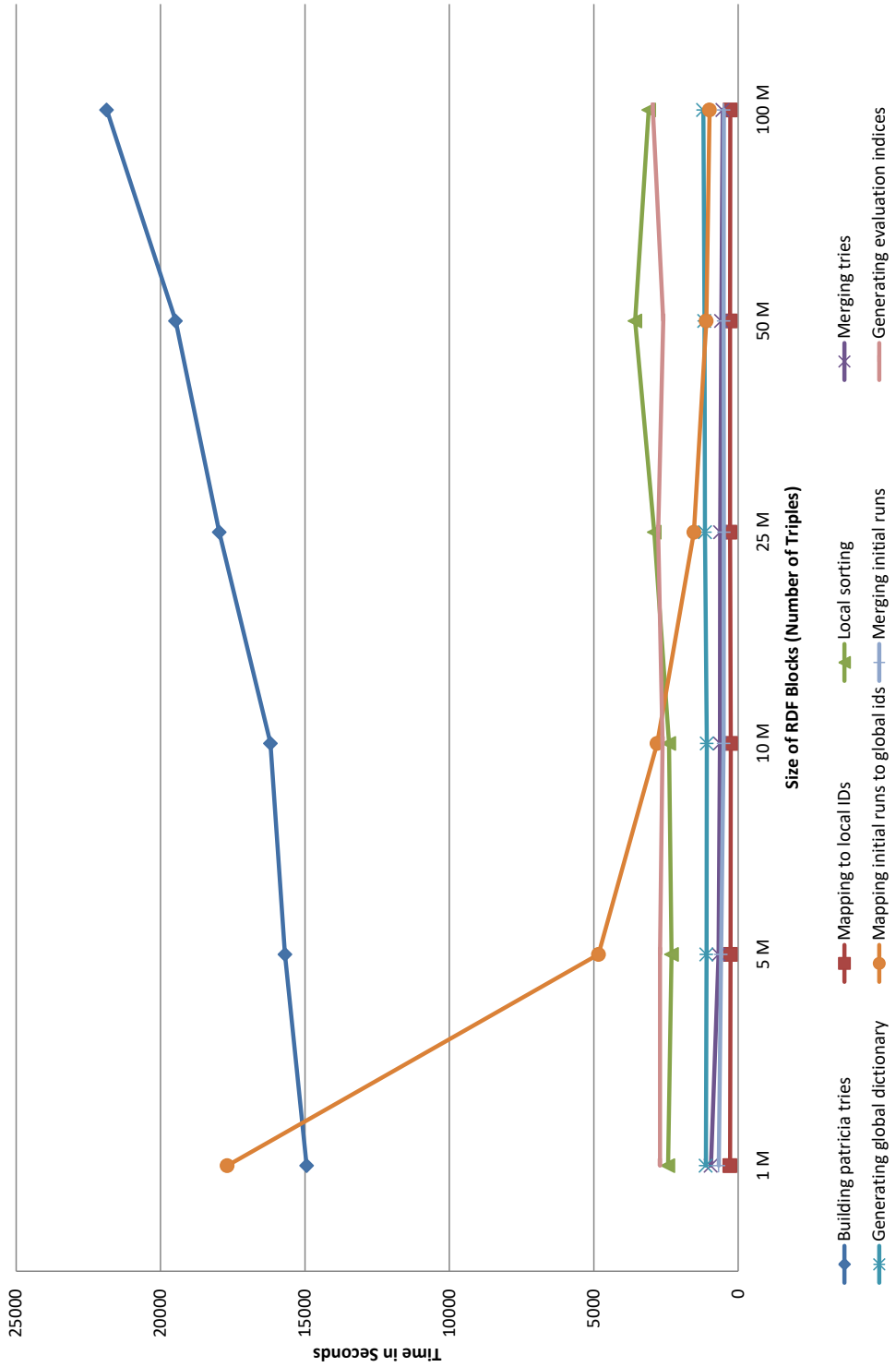


Figure 4.3: The processing times of the different phases of index construction for different sizes of RDF blocks

5 Parallel Search inside B$^+$-tree Nodes in a CPU-based Host System with an FPGA Accelerator Card

In this chapter we focus on search and update optimizations in a SW DBMS aided by an FPGA. The chosen SW system LUPOSDATE utilizes B$^+$-tree index structures for querying data. In our proposed system the B$^+$-tree is divided into two parts, therefore it is called a hybrid system. The lower levels of the B$^+$-tree are stored in the host system with values and keys. The root and other inner nodes can be stored in the FPGAs internal memory. This way we can perform a parallel search inside the inner nodes of the FPGA by exploiting its parallel nature. Since update operations presuppose a search for the correct position inside a B$^+$-tree leaf, these operations can benefit from the parallel searches performed in the FPGA. We present our scheduler ideas to estimate the expected benefit against the setup of the system and further adjustments made necessary by performed updates. In a best, average and worst case scenario we show how our scheduler would calculate the possible acceleration of our hybrid system.

The main contributions of this chapter include:

- a parallel search that can be performed inside the B$^+$-tree nodes of the FPGA resulting in shorter search times
- a fully functional SW DBMS by utilizing the PCIe connection for the communication between CPU and FPGA
- an estimation for a scheduler handling the setup of the system for update heavy scenarios

5.1 Basic Data structures, Indices and Reconfigurable Computing

In this section we recapitulate about the basic ideas that build the fundamentals of this chapter. Since this work is based in the field of the SW, we focus on the data

representation and the storage in index structures. Further, LUPOSDATE and its connection to the index structure and FPGAs are explained.

5.1.1 Used Fundamentals

The focus of the current chapter lies on the B$^+$-tree index structure introduced in Section 2.3.3.2. It is the main index structure of the chosen SW DBMS LUPOSDATE (see Section 2.1.5) for storing SW data. The common data representation of such data is the RDF format introduced in Section 2.1.3.4. This string data is mapped by a dictionary to integer IDs which are stored in B$^+$-trees. In this chapter part of a B$^+$-tree with ID triples are transferred to an FPGA. In order to save memory space the B$^+$-tree is transformed into a CSB+-tree introduced in Section 2.3.3.3. Further, on the FPGA we use a parallel search (see Section 2.3.3.4) to improve the performance of the system.

5.1.2 Further Related Work

In the following we present several works that are related to our proposed system. Starting with the use of FPGAs in DBMS we continue with the possible compression methods of trees by pointer elimination. After this we come to two hardware types that compete with the usage of FPGA in database context. First there are the smart SSDs that use internal processing power that optimize the transferred data and second we introduce related work in the area of General Purpose Computing on Graphics Processing Units (GPGPU) utilizing GPU processing power in a nonstandard way.

5.1.2.1 FPGAs in DBMS

FPGAs are well-known for their good performance in parallelizable tasks in such a way that they are used in many scientific areas to exploit this behavior. Although there is a wide range of research that considers FPGAs we will focus on researches that try to accelerate databases with the help of FPGA. First attempts to accelerate databases with the help of specialized hardware were made in the late 70's of the past century.

De Witt [188] presented the DIRECT design which is a multiple-instruction multiple-data stream architecture using microprocessors and Charge-Coupled Device (CCD) memories to access relational data in parallel and provide concurrent updates by locks. In these times hardware limits were reached quite fast and even nowadays

it is easy to exceed the limits of FPGAs with resource excessive designs or inadequate tasks. Still there is much more research around databases focusing on sorting [189, 190], merging [191] and searching [192] the data in a database with the help of an FPGA.

Casper and Olukotun [191] focus in their paper on three primitive database operations: selection, merge join, and sorting. They propose new designs for these operations and combine them in an FPGA cluster to perform a join on two tables by first sorting each table on a separate FPGA and then merging and selecting the results on another FPGA.

IBM [193] presented its Netezza Data Appliance Architecture [194] which consists of distributed computing nodes, so called Snippet-Blades (S-Blades). Inside each of these servers there is a multi-core CPU, gigabytes of RAM and an FPGA. The S-Blades are mainly used for compression and filtering of data which comes from physical hard drives. The FPGA supports the CPU in this task and forms the FPGA Accelerated Streaming Technology (FAST). Still, Netezza only supports relational databases.

Mueller and Teubner [189,190,192,195–197] cover a wide field of topics on FPGAs in database context. First, they proposed some database designs with FPGA support [195,196]. Furthermore, they focused on sorting [189], especially on sorting networks [190] which is an efficient way to sort data parallel on an FPGA. With Glacier [192,197] they proposed a library and compiler that transforms continuous queries into logic circuits.

Werner et al. [12] present a hardware accelerated query engine. At system runtime, the proposed approach dynamically generates an optimized hardware accelerator in terms of an FPGA configuration for each individual query and transparently retrieves the query result to be displayed to the user.

Blochwitz et al. [14] optimized the data structure of a radix tree for an FPGA and exploited its characteristics to accelerate the dictionary generation for SW databases.

5.1.2.2 Pointer elimination in trees

Sometimes data structures need to be transferred from one medium to a different one. The communication costs of the transfer can be lowered by using compression techniques. A possible compression of a trees data is a pointer elimination. The number of pointers is reduced in this process and eventually recalculated at the target destination. There are several approaches for pointer elimination in a tree.

Torp, Mark and Jensen [198] used an Insertion tree (I-tree), which is a degenerated B^+-tree specialized in append-only data and eliminated its pointers due to the I-trees regular structure to create a Pointer-Less Insertion tree (PLI-tree). Since this tree focuses on logging, it expects data always to be newer than the data in the tree. This can lead to a temporarily unbalanced index structure where a different number of levels is run through for search operations.

Rao and Ross [155, 199] introduced the Cache-Sensitive Search tree (CSS-tree) which is an index structure optimized for good search and lookup performance. The nodes inside a CSS-tree are optimized to match the cache line size in the computer system. Furthermore child nodes can be located by arithmetic operations. Both authors later optimized their CSS-tree which lead to a Cache Sensitive B^+-tree (CSB^+-tree) [155]. This B^+-tree variant has only one pointer per node and determines the rest of the pointers by arithmetic operations. Therefore the node must be put in *nodegroups* where they are stored contiguously.

5.1.2.3 Utilization of smart storages

FPGAs are not the only hardware platforms that can be utilized to improve database operations. The access of the CPU to the memory is also an important part in a DBMS. Not only is the size important but also the time to retrieve the data. This latency at main-memory access leads to the suboptimal phenomenon of the *memory wall* [200]. Going down the line from smaller, faster memories to bigger, slower memories the problem grows larger. In the field of persistent memories there is currently a shift between Hard Disk Drives (HDDs) and the newer Solid State Drives (SSDs) which tones down the latency problem while a low bandwidth remains [201]. To further utilize the bandwidth in an optimized way *Smart SSDs* are used to lower the amount of unnecessary or unused data sent from the SSDs. This is possible because they provide embedded processors and can use multiple I/O-channels which leads to a high performance on concurrent access patterns [202].

Do et al. [203] use these integrated processing abilities of SSDs to filter out unnecessary data and transmit only data that is needed.

Sesahdri et al. [204] use the internal processing resources of the SSD for a prototype which allows data access specialized for certain applications.

5.1.2.4 General Purpose Computing on Graphics Processing Units (GPGPU)

Another competitor in the field of parallel hardware are Graphics Processing Units (GPUs). Normally they are used for processing and manipulating images inside the GPUs memory to be displayed on a two dimensional surface. But the GPUs highly parallel internal processing structures can be used to solve problems in other application areas as well. Utilizing a GPU for another task than image processing is called General Purpose Computing on Graphics Processing Units (GPGPU).

Govindaraju et al. [205] present GPUTeraSort which uses a GPU for sorting tasks in a large databases and come to the conclusion that utilizing GPUs as co-processor can improve the performance of sorting algorithms.

Further, He at al. [206–210] extend their executions of relational queries by using a GPU as a coprocessor.

To reduce the amount of data transferred from and to the GPU, Fang et al. [211] use data compression techniques on GPUs and avoid unnecessary overhead.

5.2 Architectural Overview

In the following a introduction of the entire system with its concept and ideas is given. Starting from the software side the required steps to reach the hardware part are introduced and explained. Thereafter the best, worst and average case scenarios of the update operations are described, focusing on the possible acceleration in each case.

5.2.1 Using GPU or FPGA Processing in the proposed system

At first, we want to outline why we choose to accelerate our system with an FPGA and how an acceleration by a GPU would have changed our system. In both cases, either using an FPGA or a GPU card, the communication to the CPU would happen over the PCIe connection. Therefore, both have the disadvantage of relatively slow connection to the CPU by exchanging data over the main memory. Still, both have certain advantages [212]. GPUs can accelerate applications that have a huge amount of data parallelism or that can run threads with the same unique control flow. Further, problems that need to access memory in repeated patterns or are based on small, constant working sets without dynamic memory allocation, are best suited to be solved by a GPU. The native floating-point arithmetic of a GPU

can also be important for certain tasks. FPGAs on the other hand are chosen if an application is designed with a data flow in mind often using processing pipelines. Especially, if streams of data need to be processed FPGAs have an advantage. Therefore, a system based on FPGA acceleration was chosen since the database should handle a stream of queries. Further, FPGAs have a lower power consumption than GPUs. A special case are the triples handled in the SW scenario. As pointed out in Section 2.1.3.4, RDF triples consist of three components (subject s, predicate p, object o) and can be compressed with the help of a dictionary. Therefore, triples have a bit width of 96 bits (32 bits per s, p, o). Current CPUs and GPUs have only a word width of either 32 or 64 bits which means that triples can not be handled optimal. On the other hand, FPGAs are not bound to a certain bit width and can assemble circuits handling 96 bits at the same time and more efficient than CPUs or GPUs.

5.2.2 Basic Concept

The basic concept of this hybrid design is that the software side uses partial solutions for search operations precomputed by the hardware. On a simplified view the basic concept is a B+-tree in LUPOSDATE whose upper levels are transferred to the FPGA. With these levels, the FPGA can perform a search in the upper levels of the tree. The result can be delivered to LUPOSDATE as an entry point inside the tree rather than starting from the root. Figure 5.1 shows the normal search process on software on a CPU-based host system on the left side. The search S_1 starts at the root of the tree and handles its way through the tree structure finally ending in a leaf to finish the search. On the right side the FPGA starts its search S_F at the root and finishes it when a certain amount of levels is passed through. The FPGA then delivers an entry point for the software search S_2 that is an inner node or even a leaf from where the search will continue processed by software. Both components of the system benefit from each other. The FPGA can perform parallel searches (see subsection 2.3.3.4) inside the upper levels. Still the hardware resources on the FPGA are limited and storing all values on the FPGA is not necessary if the host system already stores them. As a consequence, only the parallel search task in the upper levels of the tree is assigned to the FPGA. The remaining operations (insert, delete, etc.) are handled by the LUPOSDATE system.

5.2.3 Pointer Elimination for Communication

To give an overview over the whole hybrid system the point of view starts from the software side of LUPOSDATE translating a B+-tree into a more suitable version

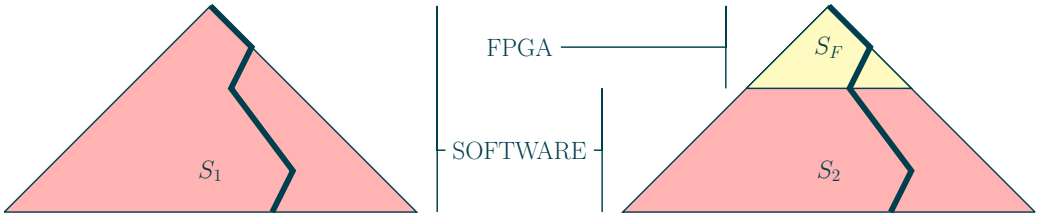


Figure 5.1: Concept of the hybrid system.

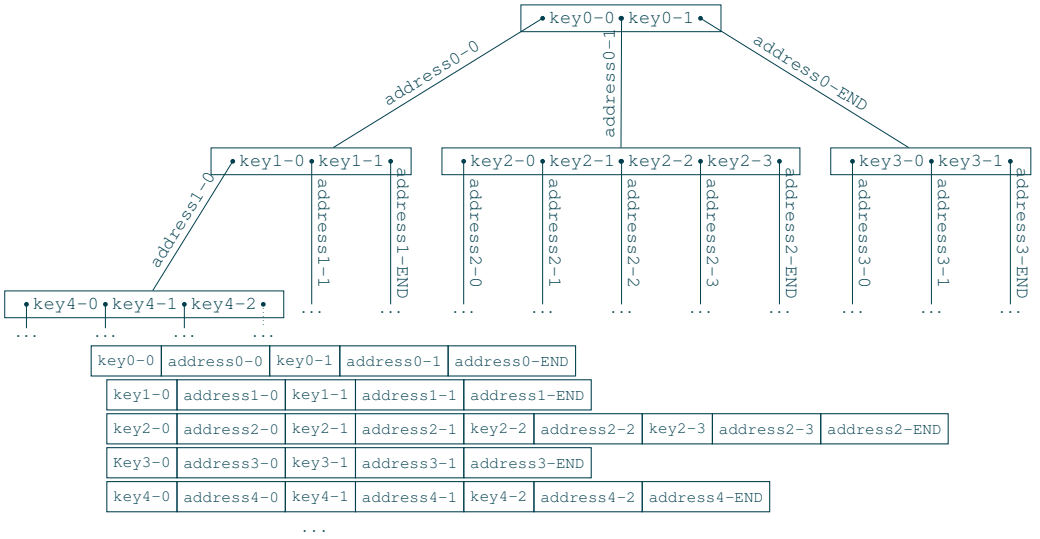


Figure 5.2: Example for a graphical representation of a B$^+$-Tree with $k = 2$ and the corresponding data representation. Each key in a node has a pointer (`AddressX` = pointer to the childnode).

for the communication with the hardware side. Two steps will be explained to show how the pointer elimination effects the transmitted data to the FPGA.

We start with an example B$^+$-tree which is located in the LUPOSDATE system. In Figure 5.2 there is a graph representation of a B$^+$-tree on the left side and a data representation on the right side. The data representation will be data that is sent to the FPGA. There are five nodes of a B$^+$-tree with the order $k = 2$. The node in the first row of the data representation is the root of the tree and the following three nodes are its child nodes. This means that the addresses `address0-0`, `address0-1` and `address0-END` point to the child nodes of the root. Usually, in main memory applications these pointers can lead to addresses far beyond the scope of this example and can even be further fragmented. This is suboptimal

because there are from $k+1$ to $2*k+1$ addresses per node that block the space for keys in the memory. The idea of the hybrid system between LUPOSDATE and an FPGA leads to possible optimization steps. The upper levels of the B$^+$-tree will be sent to the FPGA, which means each node that is sent can be rearranged in its address to suit a compression of the pointers. The children of a node will get consecutive addresses. In Figure 5.2, this means that the addresses `address0-0`, `address0-1` and `address0-END` are coherent and can easily be replaced by a starting address (in this case `address0-0`). With an arithmetic operation the resulting addresses for each key can be computed. In this example the index of the key inside the node will be added to the starting address to determine the correct address. Certainly the used memory space for the addresses is traded for an amount of processing time. For the FPGA this is most suitable because it has only a limited memory space but can perform the arithmetic operation in parallel to other tasks like receiving the data. In Figure 5.3 the result of the pointer reduction is shown. In order to transmit this generated CSB$^+$-tree to the FPGA each node starts with the address of its first child. The next two bits determine if the node is full (`00`) or how many keys are missing (in this case `01` for one key and `10` for two keys missing). The number of bits is determined by the number of possibilities a node has to hold keys, so the order of the node is important. The rest of one node are the different keys. In the B$^+$-tree the number of keys determines the number of addresses to be sent while in the CSB$^+$-tree it is the number of nodes.

Still there is one pointer left per node. But these are not necessary because in the FPGA the memory will be filled consecutively. This means the root will be at address 0 and its children addresses will start at address 1 incrementing further. If every node is filled completely it is easy to determine the actual pointer address as follows: At first the Equation 5.1 presents the maximal numbers of nodes per level of a tree:

$$f(k, L) = (2 \cdot k + 1)^L \tag{5.1}$$

With k as the order and L as the level, the maximum number of nodes can be calculated for each level of a tree. This leads to Equation 5.2 to determine the absolute number of nodes in a tree with a given number of levels L_{max}:

$$f_{absolute}(k, L_{max}) = \sum_{i=0}^{L_{max}} f(k, i) \tag{5.2}$$

The first equation limits a pointer in a level i to a certain range, the last address of the next level $((2 * k + 1)^{i+2}) - 1$ as maximum and a certain minimum, the first address of the next level $(2 * k + 1)^{i+1}$. With fully filled nodes the determination of a pointer is easy. The number of nodes $prev$, before the current node with the pointer, always use $prev * (2 * k + 1)$ addresses and just the position of the pointer in the actual node must be added.

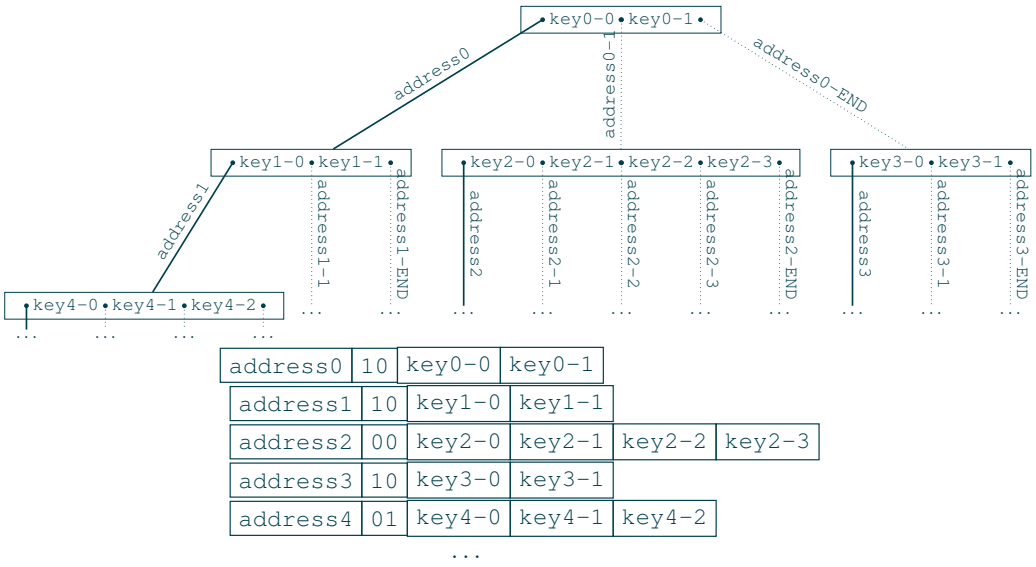


Figure 5.3: The graphical representation of the same B$^+$-Tree from Figure 5.2 with $k = 2$ now as a CSB$^+$-tree and the corresponding data representation. Each node only has one pointer (`Address` = pointer to the first child in the node group).

Since usually a B$^+$-tree is rarely filled with nodes using the maximum number of keys the determination of a pointer is more complex. This leads to the Equation 5.3:

$$A_{res} = ((A_{now} - L_{start}) \cdot (2 \cdot k + 1)) - off + L_{next} \tag{5.3}$$

The starting point to compute the resulting address A_{res} for a pointer is the actual position of the current parent node address A_{now} from the first address of the current level (L_{start}). The result is multiplied with $(2 * k + 1)$ expecting all former siblings are fully filled with keys at first. To correctly calculate the pointer an offset off counts the number of missing keys for a completely filled node and is subtracted from the former ideal multiplication. At last the first address of the next level is added to complete the pointer calculation for a node. In Equation 5.3 A_{now} and off can easily be counted while setting the pointers and $(2 * k + 1)$ is a constant since the order of a tree is variable but fixed at run-time. The most important part are the level boarders L_{start} and L_{next}. In Figure 5.4 the resulting data sent to the FPGA is shown. In contrast to Figure 5.3 not every node has an address: only each level of the tree has a first address from where the level starts. This leads to a logarithmic growth of addresses since Equation 5.1 shows that each new level can hold $(2 \cdot k + 1)$ times more nodes than its predecessor.

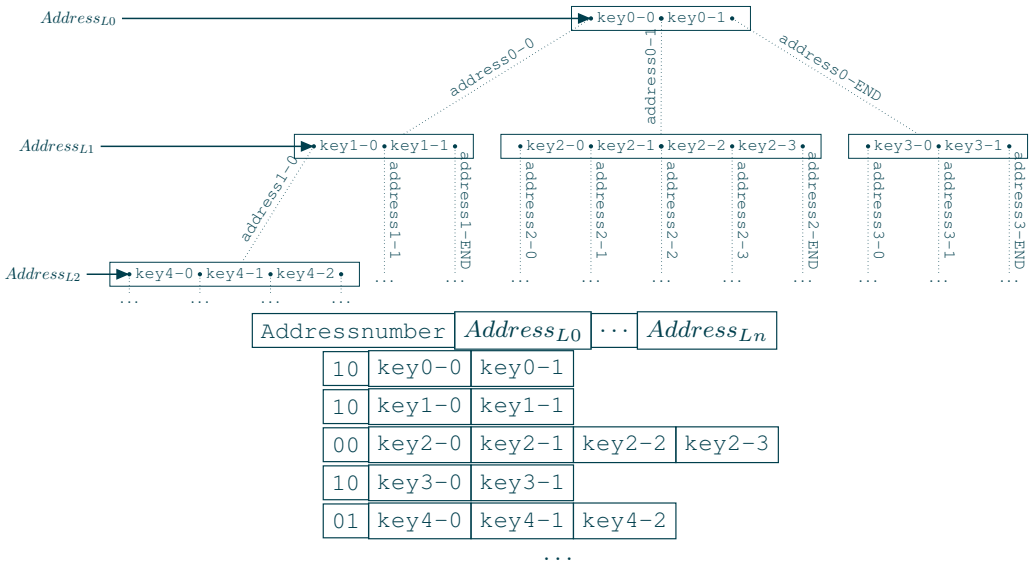


Figure 5.4: The graphical representation of the same B+-Tree from Figure 5.2 and 5.3 with $k = 2$ now without pointers between nodes and the corresponding data representation. Only each level has a pointer ($Address_{Lx}$ = Pointer to the first node of the level x).

5.2.4 Recreating Pointers on the FPGA

After reducing the amount of sent addresses to a logarithmic number, the tree can be sent to the FPGA. Here, one pointer per node will be recreated to construct a CSB+-tree on the FPGA. This is necessary to support better search performance since otherwise the pointers need to be calculated for each search in a non-parallel way. While receiving the data, the FPGA can process these calculations without additional costs since both task are implemented independent from each other and therefore can be executed parallel. Figure 5.5 shows a simplified schematic of the pointer calculation module. This module represents Equation 5.3). As seen in Figure 5.4 the addresses for the levels are transmitted first and saved in a BRAM. With the arrival of the first node the address calculation can start immediately although the first address of the root is rather trivial. The first two bits before the keys of a node are important for the offset off which counts the missing keys. The address for the next level ($Level_{next}$) is loaded from the BRAM and replaced when $Addr_{now}$ has the same value. The old $Level_{next}$ then becomes the new $Level_{now}$. The $2 \cdot k + 1$ box can be seen as a constant since the order of the tree is determined before the synthesis. The $Addr_{result}$ is written back into the BRAM.

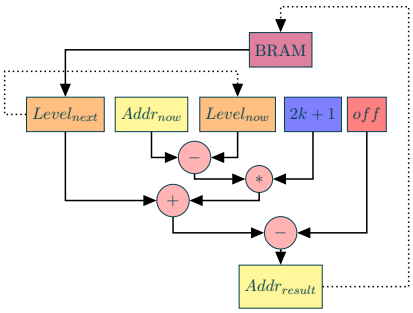


Figure 5.5: Possible arrangement to compute the next address for a search operation with a strong arithmetic approach.

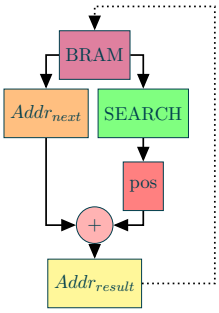


Figure 5.6: Possible arrangement to compute the next address for a search operation with a weak arithmetic approach.

5.2.5 The Search Operation

After the recreation of the pointers, the search process can be explained and is shown in Figure 5.6. The BRAMs provide the information about the triples to the SEARCH module, which will be explained in the next section. It determines the next address to visit. The position *pos* inside the searched node is added to the next address $Addr_{next}$ to calculate the correct child node where the search continues. The search inside a node of a B^+-tree must be modified to take advantage of the FPGAs parallelism, which can be seen in Figure 5.7. Instead of using a binary search or other sequential methods to find the right pointer to the next child node the comparisons between all n triples in the current node and the searched triple take place at the same time where n is the maximum number of triples inside a node $(2 * k)$. The resulting vector of answer bits of these comparisons is evaluated inside the position box which is an abstraction of the real logic at this place. The logic simply looks for the first significant bit in the node to determine the value that will be added to the next address. If all bits are zero it is clear that the searched

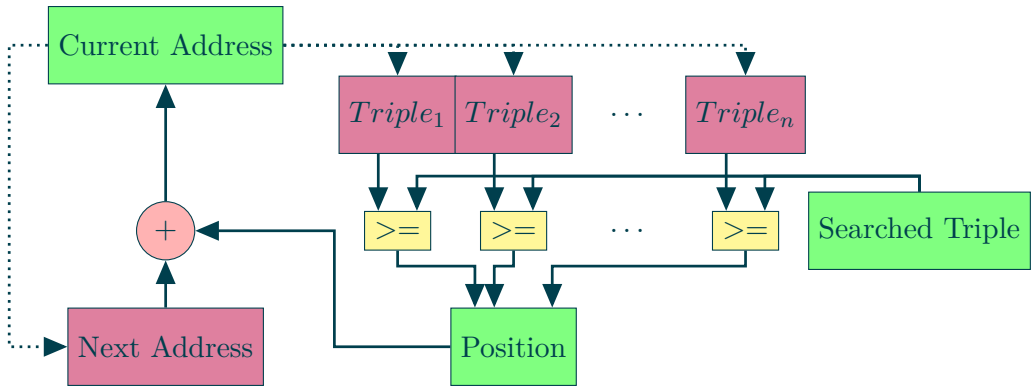


Figure 5.7: Parallel search for a triple.

triple has a greater value than all triples in the node and therefore the highest possible value (n) is added to the next address. Contrarily, if all bits are set to one the searched triple value was smaller than all triples in the node and therefore zero is added to the next address. All other combinations lead to a value in the range 0 to n. The current address can be changed with the determined position. With that the BRAMs for the triples and the next address switch to the new calculated address. This can be seen as the change to a deeper level of the tree since we visit only one node per level in a search operation and this cycle continues until the current address exceeds the number of addresses filled with the tree. This address will be transmitted to LUPOSDATE mapping this address with the correct page value so it can continue its search from a node inside the tree even from a leaf itself instead of starting from the root.

5.2.6 Acceleration of Update Operations

Besides accelerating search operations, our system is optimized to enhance insert and delete performance as well. If a value is inserted/deleted in a B+-tree the first step is always to find the correct position inside a leaf. This means, there is a search operation before any update inside the tree. The structure of the tree will be changed only if a node has more than $2k$ or k' keys and needs to be split in case of insertion. In case of the performed operation removes a key the structure is only changed if there are less keys left in the node than the order k *or* k' and two nodes can be merged. Since we only transfer the root and the most upper inner nodes from the tree to the FPGA, updates inside leaves and the lower inner nodes on the host system do not invalidate the data located on the FPGA. With this in mind, it is possible to create a scheduler that decides if a transfer of the tree to the

FPGA has a certain chance of accelerating the operations of the tree. At first we should look at the number of update operations that are possible. The simplified Equation 5.4 shows the important factors for this:

$$\begin{aligned} possibleNumberOfUpdates = (&UpdatesInLeaves\cdot \\ &UpdatesInInnerNodesSoftware\cdot \\ &EntryPointsFPGAtoHost) \end{aligned} \tag{5.4}$$

We always start update operations in the leaves ($UpdatesInLeaves$). After filling a leaf to its maximum, the leaf is split and a new key is inserted into the inner nodes of the software part ($UpdatesInInnerNodesSoftware$) if it is an insert operation. For a delete operation it is the opposite, since if a leaf has less than the minimum number of keys it is merged with another leaf and a key in an inner node is removed. The last factor is the number of entry points from the FPGA to the host system. This can be seen as multiple B^+-subtrees that are located on the host and be addressed by the FPGA. With Equation 5.4 in mind we can look at the special cases:

5.2.6.1 Worst Case Scenario

The number of times an update operation can be performed before the data in the FPGA needs to be updated is only one ($possibleNumberOfUpdates = 1$). For the insertion case a key is entered into a maximum filled leaf that causes splittings in the inner nodes which are also filled up to the maximum and lead to the point that the data inside the FPGA expires.

Figure 5.8 represents this scenario. The values $k = 2$ and $k' = 2$ are the orders of the leaves and the inner nodes in the host system (definition of k, k' in Section 2.3.3.2). If another key is entered into the leaf it needs to be split. This leads to a leaf with 3 keys and a new leaf with 2 keys. After this split the inner node needs to be updated with a new key resembling the highest key in the node with 3 keys. The inner node has more keys than the order so another split happens where a new inner node is created. But the FPGA can only address the old node while the new node can not be reached by any entry point from the FPGA. In this case the structure on the FPGA needs to be rebuilt to address the new inner node.

In the deletion case a key is removed in a minimal filled leaf causing a merge with another leaf. Therefore the minimum filled inner nodes remove keys until the levels that also reside in the FPGA are altered which invalidates the data inside the FPGA.

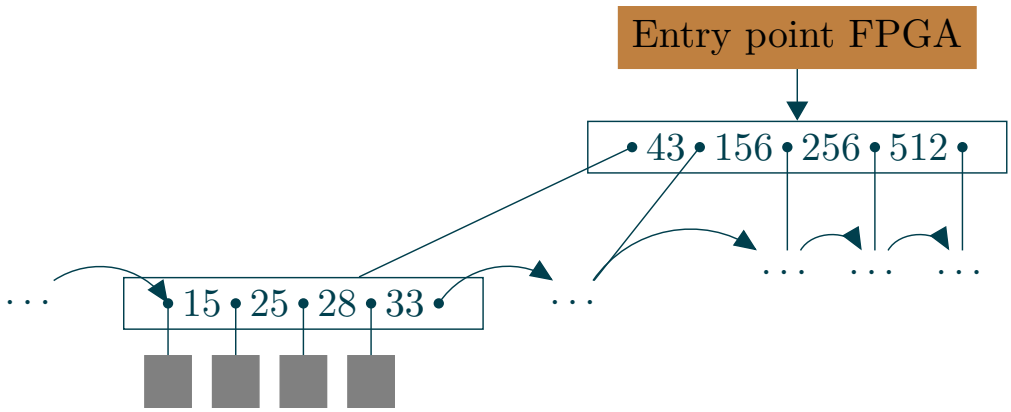


Figure 5.8: An example of a B+-tree with the orders $k' = k = 2$. The keys are in the nodes (numbers) and the values are only in the leaves (gray boxes). This represents a worst case scenario for insert operations and the best case scenario for delete operations

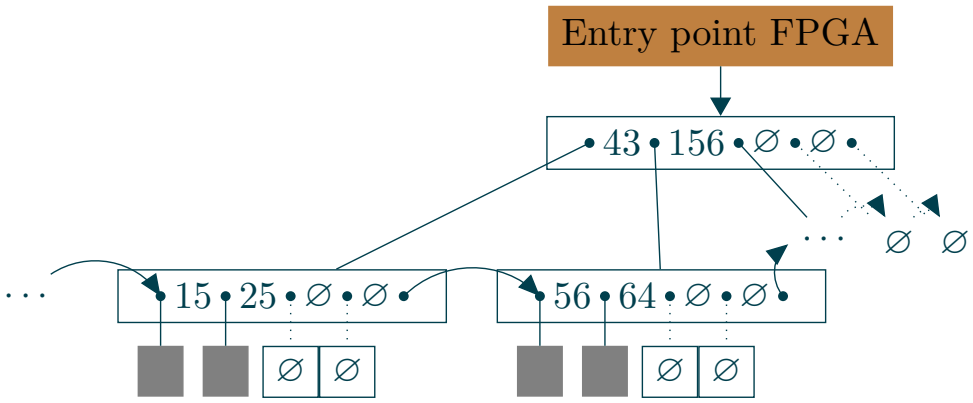


Figure 5.9: An example of a B+-tree with the orders $k' = k = 2$. The keys are in the nodes (numbers) and the values are only in the leaves (gray boxes). This represents a best case scenario for insert operations and the worst case scenario for delete operations

Figure 5.9 shows this worst case scenario for a delete operation. Removing the key 25 and its value leaves the left leaf with only 1 key and is therefore below the leaf order of $k' = 2$. The right leaf also has only the minimum number of keys which leads to a merge of both nodes to only one leave with the keys 15, 56 and 64. The key 43 is not needed anymore since there is only one leaf left that can be addressed by the key 156. By removing the key 43 the inner node has less keys than the order of the inner nodes ($k = 2$) and needs to be merged with another inner node.

At this point one entry point from the FPGA loses its addressed inner node while the software performs further merge actions in the inner nodes of the software part that are also stored in the FPGA.

5.2.6.2 Best Case Scenario

The best case scenarios for updates uses the exact opposite of the starting positions as the worst case scenario. The best case scenario for insert operation is when the inserted keys can be divided in such a way that all leaves can be filled to their maximum capacity. Figure 5.9 shows the best case for insertion operations, assuming that all leaves and inner nodes are only filled to their minimum. On the other hand Figure 5.8 shows the best case for removing keys. Assuming that all leaves and inner nodes are filled to their maximum number of keys, the maximum number of delete operations can be performed before the data on the FPGA does not match the tree in software anymore.

It is possible to derive an equation from Equation 5.4 to calculate the maximum number of updates that are possible. The variable $UpdatesInLeaves$ depends on the number of keys that can be inserted into a leaves. Since we know from the order k' how many keys can be inserted or deleted in a leaf, $UpdatesInLeaves = k'$ applies. The same is true for the inner nodes but since we can have multiple levels of inner nodes we need to adjust $UpdatesInInnerNodesSoftware$. There are multiple levels of inner nodes (L_{tree}) but we can only count the levels that are not located in the FPGA ($UpdatesInInnerNodesSoftware = k^{exponent}$). Therefore we need to adjust the *exponent* by subtracting the leaf level (-1) and the number of inner levels inside the FPGA (L_F) from the total number of levels of the tree (L_{tree}) resulting in $UpdatesInInnerNodesSoftware = k^{L_{tree}-1-L_F}$. At this point the number of connections between the FPGA and the host system are important ($EntryPointsFPGAtoHost$). Every entry point from the FPGA to the host system multiplies the number of subtrees that can be addressed. Therefore, the function $f(k, L_{max})$ from Equation 5.1 can be used with the order k_{FPGA} of the FPGA and the number of levels inside the FPGA (L_F) to compute the maximum number of entry points from the FPGA to the host system. This extends the $possibleNumberOfUpdates$ to Equation 5.5:

$$possibleNumberOfUpdates = k' \cdot k^{L_{tree}-1-L_F} \cdot (2 \cdot k_{FPGA} + 1)^{L_F} \tag{5.5}$$

5.2.6.3 Average Case Scenario

Since the number of possible updates for the worst case (only one update) and best case (Equation 5.5) are known, the average case lies between these two extremes. For the average case we assume that the values that are entered into the tree are chosen random. This is an advantage in the decision process of the scheduler compared to the other two cases where the best case for one operation is the worst case for the other operation where the scheduler takes risk in making a big loss.

5.2.6.4 Making Decisions

When the number of update operations ($possibleNumberOfUpdates$) is determined the scheduler still needs an equation to calculate the potential benefit which leads to Equation 5.6:

$$t_{gain} = possibleNumberOfUpdates \cdot t_{Op} \cdot r_{Op} - t_{setup}(x) \tag{5.6}$$

The important part is the t_{gain} which is the time we gain if we are using the FPGA. If it is negative, this means the setup of the system ($t_{setup}(x)$) takes more time than the acceleration of the operations can compensate before the data in the FPGA expires and needs to be updated. If it is positive, it is expected that the use of the FPGA boosts the database system. Since the setup of the tree depends on the number of triples that are transferred to the FPGA $t_{setup}(x)$ is a function to determine the time for a given number of triples x. Furthermore, there are three factors that influence the time we gain using the FPGA. The first is the $possibleNumberOfUpdate$ we can perform. For each special case we determined the number we can enter here. The time per operation we gain is t_{Op} which needs to be measured first before the scheduler can use heuristics for this factor. The ratio between different operations (search, update) is also important. A pure search gives the hybrid system the opportunity to save time while the structure of the B+-tree will not change. And while an update most likely will add or remove a key in the tree and therefore has the possibility to change the tree structure, there is also a search for the correct position of the key. This leads to Equation 5.7 and the following limitations of the ratio r_{Op}.

$$r_{Op} = \frac{number\ of\ searches}{number\ of\ updates} \tag{5.7}$$

First the value cannot be below 1 because each update operation implies another search operation to find the correct position. If it is exactly 1 ($r_{Op} = 1$) this means there are only update operations i.e., after each search there is an insertion in a

leaf. Between 1 and 2 ($1 < r_{Op} < 2$) there are at least some searches while the most parts are updates. As an example $r_{Op} = 1.5$ means that per two updates there is one search. For a value of two the search and update operations are even, meaning after every second search there is an insertion or deletion in a leaf. Every value of $r_{Op} > 2$ means that there are more search operation than updates. For our worst case scenario it is clear that $r_{Op} = 1$ applies to represent an "insert only" scenario.

5.3 Experimental Analysis

Our approach of hardware accelerated index structures aims on database systems with a high rate of requests in comparison to their updates. A web shop is a good example with many customers requesting data to products while the administrator updates the data on a regular base but only adds or removes products if new releases arrive or products are out of stock and will not be available anymore. With this behavior in mind the following experiments focus on the search itself and in the last part on update operations. The construction, compression and transmission of the tree to the FPGA is therefore seen as a constant cost for the search operation which occurs only once. Thereafter, when we focus on the update operations these steps become more interesting to our scheduler and are examined further.

5.3.1 Experimental Setup

Our experimental setup consists of a Dell Precision T3610 workstation [213] with 40 GB of RAM and an Intel Xeon E5-1600 v2 processor with 3.0 GHz. The FPGA inside the workstation is a Xilinx Virtex-6 XC6VHX380T [113] using only its BRAM to store the triples. The communication between workstation and FPGA takes place via PCIe 2.0 connection with 8 lanes.

There are three different groups of trees we measured. At first a single B^+-tree with an order of 500 for inner and leaf nodes inside LUPOSDATE. Therefore the first group consists only of one tree. This tree is optimized for HDDs, since it expects that the main memory will not hold the complete tree. Therefore the size of the interior nodes is chosen in such a way that if a node is written to disk the size of the node matches the block size of the hard disk. This group will be called L_{B^+} since it is the standard B^+-tree LUPOSDATE uses. The second group consists of trees with different small orders (3, 4, 5, 6, 7, 8, 9) for the inner nodes and an order of 500 for the leaf nodes and like the first group is executed in software. These trees can be seen as main memory variants. Storing them on a HDD will result in

wasted memory since the block size is far greater then the stored node. This group will be called L_{MM} for LUPOSDATE main memory. The last group consists of trees with different orders like the second group but the interior nodes are stored in the FPGA and only the leaves with order 500 are located in LUPOSDATE. This group will be called L_{FPGA}. The reasons for choosing the three groups are the following. The L_{FPGA} group is the design we proposed and it is compared with the L_{B^+} group since these are the standard parameters LUPOSDATE is used with. Still, comparing so strong differing orders seems to be a disadvantage to LUPOSDATE. This led to the L_{MM} group which is rather artificial, but comes close to the architecture of the L_{FPGA} group.

As an indicator of the acceleration by the FPGA the *speed up* for a single search operation is given. This *speed up* is the average from 10,000 performed search operations. These runs were executed 100 times to minimize inaccuracies. In Equation 5.8 the calculation of the *speed up* can be seen. It is the division of a pure software search t_{SW1} and a hybrid search with the FPGA. The time for the hybrid approach consists of the software time t_{SW2} which is the average search time starting inside the tree plus the communication costs to transfer the searched triple and get the answer from the FPGA (t_{FPGA}). With this, a number above 1.0 means an advantage to the hybrid system since its computation time is lower than the pure software solution. On the other hand a number below 1.0 means an advantage for the software solution.

$$speed\ up = \frac{\varnothing t_{SW1}}{\varnothing t_{SW2} + \varnothing t_{FPGA}} \tag{5.8}$$

A tree with a defined order and a given number of levels is limited to a certain amount of triples. If the number of triples is below a minimum the number of levels cannot be hold, because it would ignore the Equation 5.1) and 5.2. The following holds for the L_{MM} and L_{FPGA} groups. In our experiments the number of inner levels of the trees is four since this is the most common supported number of levels per chosen order that fit into the FPGAs BRAM. For the orders five to nine it is also the maximum number the FPGA can handle only supporting BRAM. With four inner levels on the FPGA, LUPOSDATE handles the leave level which in this case has an order of 500. To create the B$^+$-tree index we use a sorted dataset and construct the index from scratch rather than inserting the triples sequentially. The method works bottom to top first dividing the data into a number of leaves equally distributing the triples as far as possible. After the number of leaves is determined the interior levels can be built also equally distributing the keys. To create a suiting tree with four inner levels for each order the maximum triple number a tree with three inner levels is taken and increased by one. Further, Equation 5.2 is used to determine the maximum number. For this purpose L_{max} is three giving

Table 5.1: Number of triples entered into the tree with four inner levels and a leave level for a given order. This table gives the corresponding A, B, C, D, E in the Figures 5.10, 5.11 and 5.12.

order	A	B	C	D	E
3	343,001	857,500	1,372,000	1,886,500	2,401,000
4	729,001	2,187,000	3,645,000	5,103,000	6,561,000
5	1,331,001	4,658,250	7,985,500	11,312,750	14,640,000
6	2,197,001	8,787,750	15,378,500	21,969,250	28,560,000
7	3,375,001	15,187,250	26,999,500	38,811,750	50,624,000
8	4,913,001	24,565,000	44,217,000	63,869,000	83,521,000
9	6,859,001	37,723,000	68,588,000	99,453,000	130,320,000

us the total number of inner nodes and multiplying this number with 1,000 (since the order of the leaves is 500) gives the total number a tree with only three inner levels can handle. This leads to the triple numbers of row A in Table 5.1 which are trees with interior nodes that are not completely filled. On the opposite to reach the maximum number of keys inside each node, Equation 5.2) is used with $L_{max} = 4$ also multiplying the result with 1,000 giving the number of triples in row E. The rows B, C and D are equally taken steps between A and E gradually filling the inner nodes. For the L_{B^+} group there is neither minimum nor maximum to the interior levels. Still, it contains the same amount of triples corresponding to Table 5.1 of the specific order but never changes its order (500), only the groups L_{MM} and L_{FPGA} change orders (3 to 9).

There are three diagrams we use to evaluate the different groups. Figure 5.12 shows the execution times of one search for each group (L_{B^+}, L_{MM}, L_{FPGA}), each order (3 to 9) and each filling state of the interior nodes (A, B, C, D, E). Figure 5.10 shows the *speed up* between the L_{MM} group and the L_{FPGA} group. Figure 5.11 shows the *speed up* between the L_{B^+} group and the L_{FPGA} group.

5.3.2 Fill Ratio of the Tree Levels

This evaluation focuses on the L_{MM} group and the L_{FPGA} group to decide which impact the fill ratio of the interior nodes has. In Figure 5.10 we see the *speed up* of the different orders. It can be seen that for the most orders the minimum of the *speed up* is at point A (order 6, 8, 9) or point C (order 3, 4, 5) except order 7 (D). On the other hand the maximum speed up is located at the points D (order 4, 5, 6) and E (order 3, 7, 8, 9). This leads to the assumption that well filled inner nodes are desired to maximize the acceleration. This can be easily explained when we look

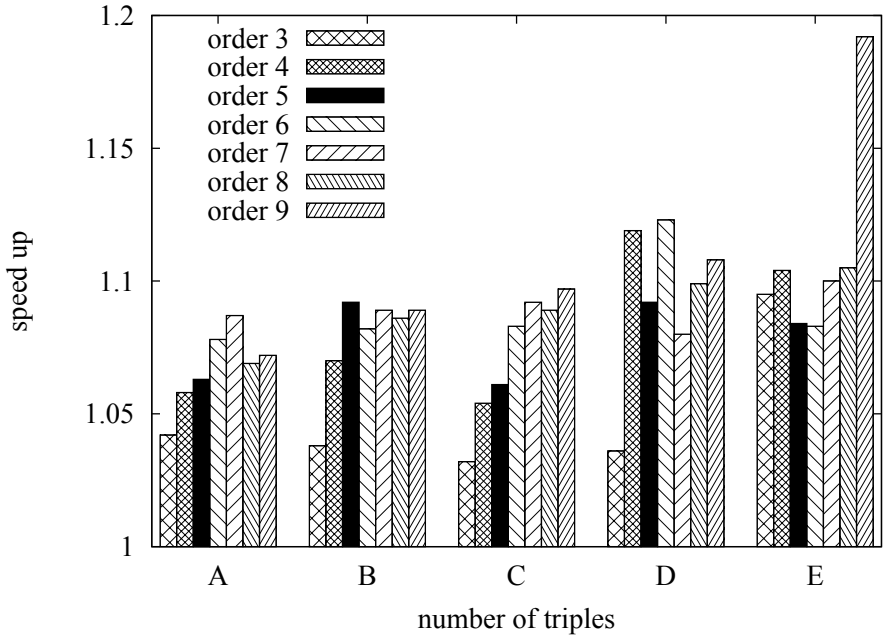


Figure 5.10: Speed up for the hybrid system to perform the searches against LUPOSDATE with various orders (for A to E see Table 5.1).

back at Figure 2.36. Since in this experiment there is compression, LUPOSDATE uses the linear search inside an interior node to find the next corresponding child to continue the search. With an order of three there are only three to six keys in a node. When the inner nodes mostly contain only three keys it is clear that an acceleration is hard to accomplish since the linear search starts at the first element of the node and only needs to perform two more steps in the worst case. This means the linear search only needs three steps at maximum to continue its search in the next node. With a maximum filled node the linear search has six elements to check while for the parallel search of the FPGA it is insignificant how many keys are inside a node.

5.3.3 Best Order of the Tree

Since we made clear in the previous subsection that maximum filled interior nodes are the key to a good acceleration the order of the tree also has an impact. Fig-

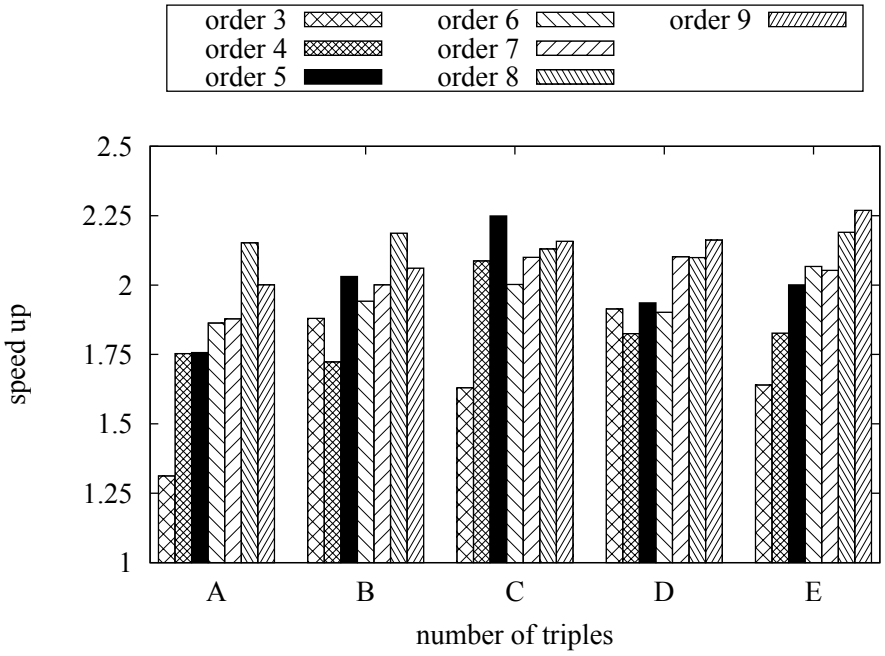


Figure 5.11: Speed up for the hybrid system to perform the searches with various orders against LUPOSDATE with an order of 500 (for A to E see Table 5.1).

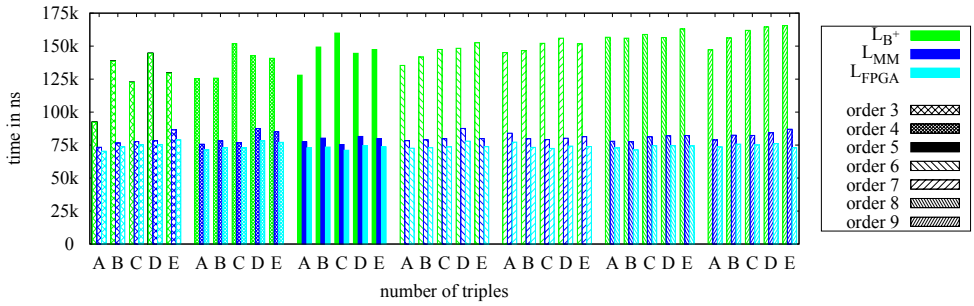


Figure 5.12: The execution times of one search for each group (L_{B^+}, L_{MM}, L_{FPGA}), each order (3 to 9) and each filling state of the interior nodes (A, B, C, D, E) (for A to E see Table 5.1).

ure 5.12 presents all measured execution times. For L_{B^+} the execution times increase from order three to nine. Looking at Table 5.1 it is obvious that the increased number of triples slows down the search operations since LUPOSDATE uses compression and therefore the linear search (see Figure 2.36). For L_{FPGA} this is an advantage since the only linear search is performed in the leaves while the interior nodes are searched in parallel. Figure 5.11 shows therefore a significant *speed up* especially for the orders eight and nine which are always above a *speed up* of 2.0. Compared to the *speed up* from L_{B^+}, the *speed up* from L_{MM} in Figure 5.10 is worse but still significantly over 1.0. Here, the higher orders have a better *speed up* like for example order three. This leads to the conclusion that higher orders are better to reach a higher *speed up* than low orders.

5.3.4 Impact of updates inside the B+-tree

After measuring the times in this evaluation scenario we can get back to the Equation 5.6 in Section 5.2.6 to look if it is even possible for our scheduler to make a decision to transfer data to the FPGA or not.

5.3.4.1 Common Parts

First we will look at the common parts of Equation 5.6 before we come to the special cases. The t_{Op} is the difference between the execution times of the L_{B^+} and the L_{FPGA} group. Since we looked at different filling states of the nodes we will take the best and worst times (A, E) for the corresponding case (worst/best case) and the average for the average case. The setup time of the system t_{setup} is the accumulated time of three successive steps shown in Figure 5.13. First a modified Breadth-First Search (BFS) is performed inside the upper parts of the tree located on the host system. This will also determine the entry point from the FPGA to the host system. After gathering all information the tree is converted (*Convert*) to reduce the data that needs to be transferred to the FPGA. This takes about a quarter of the time the BFS took. The last step is the *transfer* of the data via PCIe to the FPGA which takes the double of the time of the conversion and half the time of the BFS. Adding all three times we get the total amount of time to setup the FPGA with the tree data which we equate with $t_{setup}(x)$ of the Equation 5.6. As it can be seen all times grow linear to the amount of triples that are inserted into the tree. Using linear regression for our experiments $t_{setup}(x)$ can be approximated as:

$$t_{setup}(x) = 2.7501701520384 \cdot 10^{-7} \cdot x \; seconds \tag{5.9}$$

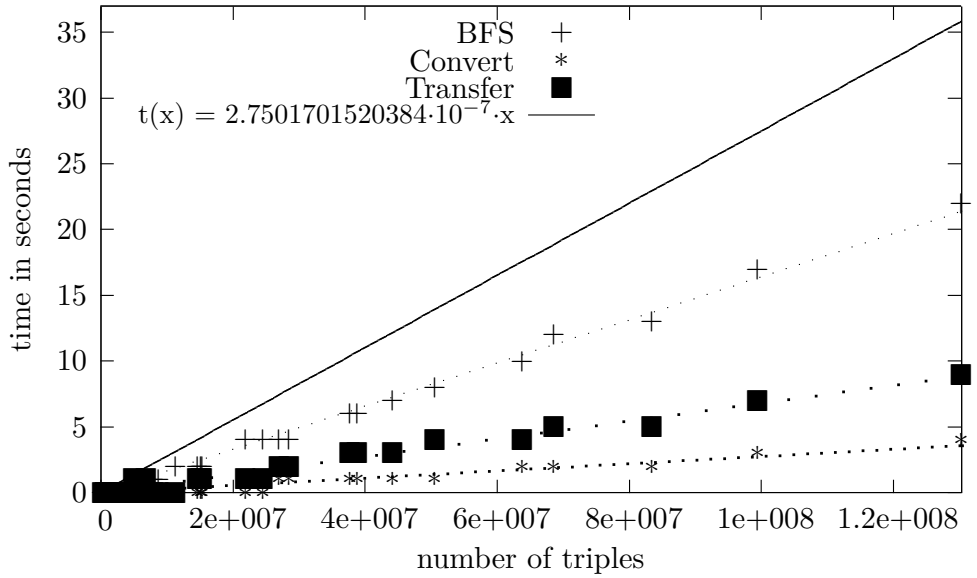


Figure 5.13: Computation time for setting up the hybrid system compared to the number of triples inside the tree

With Equation 5.9 we can calculate the time $t_{setup}(x)$ for a given number of triples x. Further we know the orders of the tree. Since we are looking at the lower levels of the tree in the host system the typical values for LUPOSDATE are $k' = 500$ and $k = 500$. The levels inside the FPGA are four in our experiments, hence $L_F = 4$ applies and the total number of levels is five $L_{tree} = 5$. There are two variables left. The ratio between updates and searches r_{Op} and the time we gained for using our hybrid system (t_{gain}). For our scheduler it is important to know this ratio in order to decide, whether or not it should migrate the B$^+$-tree to the FPGA. To get the minimum worst case ratios for the different numbers of triples in Table 5.1, the value will be set to zero $t_{gain} = 0$ so the hybrid system will be at least as fast as the software solution.

After switching the variables from the Equation 5.6 we get Equation 5.10 .

$$\frac{t_{setup}(x)}{possibleNumberOfUpdates \cdot t_{Op}} = r_{Op} \tag{5.10}$$

The $possibleNumberOfUpdates$ variable will be replaced by the corresponding number of the case.

5.3.4.2 Worst Case

In the worst case we can only perform one update which leads to Equation 5.11.

$$\frac{t_{setup}(x)}{t_{Op}} = r_{Op} \tag{5.11}$$

Therefore the worst case is independent from the orders k_{FPGA}, k' and k and only determines how many searches need to be performed before a single update can take place.

5.3.4.3 Best Case

The best case scenario depends on the order k_{FPGA} of the tree inside the FPGA and the number of levels L_F. Since we only have a leave level in software the $UpdatesInInnerNodesSoftware$ part from Equation 5.4 results in 1 which leads to Equation 5.12.

$$possibleNumberOfUpdates = k' \cdot (2 \cdot k_{FPGA} + 1)^{L_F} \tag{5.12}$$

5.3.4.4 Average Case

For the average case we will choose different values for $possibleNumberOfUpdates$ between the best and the worst case.

5.3.4.5 Comparison

In Table 5.2 the minimum ratio r_{Op} is shown with the restriction to be at least as fast as the software system. In the worst case the ratio is increasing by every order. This is plausible since the order k_{FPGA} has no impact on Equation 5.11 and $possibleNumberOfUpdates$ is constant. With a linear growing setup time $t_{setup}(x)$ the ratio can only grow. A different behavior has the ratio in the best case scenario. Since the number of nodes inside the host we can address, grows by the order we are using and also faster than the setup time, the ratio decreases by every order step. Table 5.2 further shows the maximum number of triples that can be inserted to show the growth which automatically decreases the ratio. The values of the worst case show that we need much more search operations than inserts into the tree. Figure 5.14 shows how fast this condition changes when we move away from worst case. Starting at 500 triples and doubling the number of triples a few times and entering them into Equation 5.10 for not_x shows that the needed

Table 5.2: The needed minimum ratio of search and insert operations in the worst and best case scenario with the maximum number of triples that can be inserted.

order	worst case	best case	$possibleNumberOfUpdates$
3	30,112.9	0.0148414	1,200,500
4	34,210.9	0.0107322	3,280,500
5	73,021.3	0.0082955	7,320,000
6	125,228.9	0.0083823	14,280,000
7	205,270.3	0.0077481	25,312,000
8	280,292.6	0.0064449	41,760,500
9	486,721.4	0.0066672	65,160,000

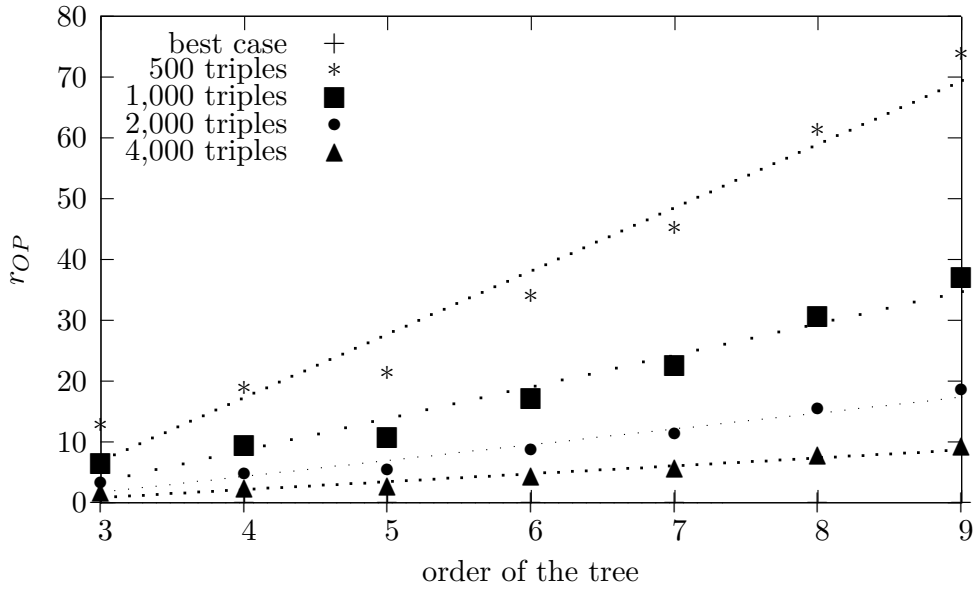


Figure 5.14: Minimum ratio between search and insert operation to the order used on the FPGA.

ratio halves every time. Expecting that the insert operation inserts random values it is very unlikely that from a few thousand inserts one of thousand leaves gets more than its maximum triples. This leads to the assumption that our proposed scheduler is needed to avoid the worst case but it is very likely that in most cases the hybrid system is faster than the software solution even when we only perform insert operations.

5.3.4.6 Delete Opertations

Since we have discussed in Section 5.2.6 that the worst case scenario for inserts operations is the best case scenario for delete operations and vice versa, the given assumptions for insert operations also apply to delete operations. Therefore, our scheduler needs to look for leaves that have a number of triples close to the maximum (worst case for insertions) and close to the minimum (worst case for deletions) to decide whether it is beneficial to transfer the tree to the FPGA or not.

5.4 Discussion

Our scheduler is intended to work in an environment where the continuous flow of search and update operations are unknown. Still, as shown in the worst case scenario, a scheduler is needed in order to avoid long setup times of the system compared to a small gain in performance. One solution to avoid the worst case could be the tracking of the minimal and maximal filled nodes of the tree. If the node/s with the maximum number of keys is/are below the double of the order k, the worst case can only occur by inserting keys to the difference. For example, if the maximum node can receive 5 additional keys the worst case is only possible at the sixth insert operation (which means $possibleNumberOfUpdates = 5$ is the current minimum). The same is true for delete operations where the minimum number of keys shows if a worst case scenario is possible. If the minimum number of keys is the same as the order k/k' a worst case could happen. Otherwise a worst case is not possible since each node in the tree has at least one key it could delete without changing the structure of the tree.

The best case scenario does not help our scheduler answering the question if the tree should be transferred to the FPGA. The answer in this case would always be yes, since even with solely update operations the system would benefit from the FPGA usage. Still, this case helps us to estimated the maximum time t_{gain} our system can safe at a current ratio r_{Op} with a value for $possibleNumberOfUpdates$ from the best case scenario (see Equation 5.6). In the evaluation part t_{gain} was set to zero in order to guarantee that the setup takes the same amount of time as the acceleration. If we want to set a certain acceleration goal t_{goal}, we can use the t_{gain} of the best case scenario to determine if reaching a goal is possible or not. If $t_{goal} > t_{gain}$ holds the goal cannot be met at the current ratio r_{Op}.

For the average case scenario the number of possible updates before a change in the tree structure happens should be monitored. Together with the current ratio r_{Op} the expected acceleration time t_{gain} could be estimated. Further t_{gain} was set

to zero in order to make the FPGA-accelerated system as fast as the software-based system. Since our measurement are grounded on the average there are still deviations that can be compensated by rising t_{gain} to a certain threshold.

A weakness in our evaluation is possibly the ratio r_{Op} that just looks for search and update operations. We assume in a worst case manner that all insert/delete operations lead to a change in the tree structure without affecting each other. But insert operations can be negated by delete operations performed in the same leaf node and vice versa. Therefore, the ratio between insert and delete operations is important to determine how likely these negations are. This also depends on the current number of leaves since the negating operations must happen in the same leaf. A refinement with a new ratio would make our estimations more precise in the average case. Still, the used ratio applies to the worst case while in a best case scenario endless delete/insert operations in the same leaf would be possible without restructuring the B+-tree..

5.5 Summary and Conclusions

In this paper we have shown a hybrid index for big data systems using an FPGA and a traditional computer. The evaluation of the index showed that filling the node with the maximum numbers of keys is important to increase the *speed up*. Furthermore, the order of the tree is important. Trees with high orders have shown a better *speed up* than trees with low orders. This leads to the conclusion that maximum filled nodes in a tree with a high order result in the best system acceleration. In our experiments we reached therefore a maximum *speed up* of 2.3 against the typical B^+tree representation of our SW DBMS LUPOSDATE with a tree order of nine and all nodes completely filled with keys. Further, we have shown that our system can also accelerate update operations. Since the setup of our system takes some time, we need to compensate this loss with our acceleration. We introduced our scheduler which decides whether it is a beneficial to transfer the tree structure to the FPGA or not. For this we evaluated the worst, average and best case scenario and have given equations for our scheduler.

6 Conclusion

In this work we investigate different index and data structures in the context of SW DBMS. Since the SW relies heavily on string representations we presented different techniques to handle them optimal. We introduce a new sorting technique that supports the index construction of a B^+-tree which is later transferred to an FPGA.

The triples used in the SW can consume much memory space, since these consist of three strings (*subject s*, *predicate p*, *object o*). A common approach is the usage of dictionaries that allow the mapping from string to integer IDs and vice versa. These ID triples can be indexed in multiple B^+-tree for each order (*spo*, *pos*, *osp*, *sop*, *ops*, *pso*). Instead of entering the triples with individual operations it is more efficient to sort the triples and construct the index on the sorted data. Patricia tries are data structures that can handle strings in a very space-efficient way. While entering new strings into a patricia trie these are also sorted. Therefore, we introduce PatTrieSort as a new sorting approach as a variant of external merge sort for sorting strings. We construct initial runs of patricia tries in main memory and swap them to HDD if they exceed the capacity of the main memory. The tries only store a common prefix once which allows bigger runs in the main memory. Further, they can be saved in sorted order on HDD by iterating over the contained entries of a trie by traversing its tree. If all data is processed once, the merge process starts creating new and bigger patricia tries from the tries on the HDD. The sorted order of the tries on the HDD allows the loading of the parts of the tries that are currently necessary for the merge process. Thus, all tries avoid repeated comparison of common prefixes and are still processed only once without loading the whole tries. In the end, there is one patricia trie left as the final result. The analysis of PatTrieSort shows the best results in terms of memory consumption, I/O costs and runtime against other external merge algorithms.

After the introduction of PatTrieSort, we use it in a new approach of efficient index construction. The goal is to extend PatTrieSort in a way that the construction of the dictionary and the six evaluation indices can be performed in the most space-efficient and parallel way possible. Therefore, we convert the triples from their string representation into an integer ID representation as soon as possible. Thus, we save main-memory by storing only 3 integers instead of 3 strings for each triple.

Since, the dictionary is one of the results of the whole construction process, there are three types of IDs. The temporary IDs are built according to the occurrences of the RDF terms while still constructing the patricia trie. Therefore, repeated RDF terms can be identified by the temporary ID and no further string representation must be saved. After a patricia trie is finished, the temporary IDs can be mapped to local IDs which are built according to the lexical order of the RDF terms in the trie. The trie is rolled out on HDD/SSD to free space in the main memory. This space is necessary since the local IDs are sorted in parallel after the six collation orders and thus need six times the memory space as before. We use Counting Sort for sorting the 3 primary collation orders (s, p, o) in parallel since the range of the local IDs is limited. Further, blocks of triples with the same primary key sorted by Counting Sort can be sorted after the secondary key by a fast standard sorting algorithm like quicksort. This allows a high parallel execution of the six collation orders. After all tries are generated they are merged into a final patricia trie. This leads to the building of the dictionary allowing the uniform mapping from RDF terms (in form of strings) to global IDs (in form of integers) and vice versa. The local IDs are mapped to the global IDs in the dictionary which allows the final generation of the evaluation indices with IDs. The results proof that our approach is capable of handling large-scale datasets, like the BTC.

The search performed inside a B^+-tree node is either a linear or binary search. Both types do not support a parallel execution of the search. Therefore, we introduce a hybrid system consisting of a CPU-based host system and a PCIe accelerator card with an FPGA. The FPGA allows a user-specific circuit that can perform a parallel search which means that all node keys are compared to a searched key at the same time. Since the memory of the FPGA is limited, only the root and the first inner levels of the tree are handled by the FPGA. The rest of the inner levels and the leaves are handled by the host system. This also means that the communication between the CPU and FPGA happens via PCIe connection. In order to lower the communication cost we use pointer elimination which allows the transfer of keys while pointers are recalculated on the FPGA side. At first, we measured the execution times of search operations with and without FPGA support and show that the times are halved in the hybrid system. Since the system setup needs a certain amount of time, the gained time by parallel searches must at least counteract this loss. With the measured times we propose equations for a scheduler in a scenario in which not only search but also update operations are executed which lead to further readjustment costs.

In summary, this work contributes the new sorting approach PatTrieSort which is at least 30% faster in the Sort Benchmark and 3.75 times faster in the BTC against other external merge sort variants. Further, a paralleled construction of the dictionary and evaluation indices of LUPOSDATE is presented which is capable of

handling the large-scale dataset of the BTC in a space-efficient way. Also, a hybrid system consisting of a CPU-based host system and an FPGA accelerator card enables a parallel search in the upper inner levels of a B^+-tree and therefore halves the execution times of searches. Still, there is potential for some improvements that will be outlined in the next section.

Further Steps

In this section we outline some further suggestions based on this work and provide ideas for research directions that can be investigated in the future.

Hardware acceleration of the index construction

Future work includes research on distributed and hardware-accelerated index construction presented in Chapter 4. Especially the construction of the patricia tries and generating the runs can be independently processed for each block of RDF data and hence these steps seem to be perfectly suitable for distributed processing and hardware-accelerating by FPGAs and/or GPUs. First results for the generation of patricia tries hardware-accelerated by FPGAs are already available [14], which need to be further extended to cover the full index construction process.

Ensuring data quality

Handling large volumes of real data from various sources may cause data quality issues. Especially, synonyms and homonyms can cause unintended results in the merging process. The first group can lead to differentiations that are not necessary or even wrong. For example, if one source uses '`buy`' as predicate in the RDF triples and another sources uses '`purchase`' the resulting merged RDF document should guarantee that all subjects with their corresponding objects that can be '`bought`' can also be '`purchased`' and vice versa. The later group needs differentiations although the writing is the same. For example, '`bow`' could mean a weapon firing arrows or an equipment to play certain string instruments such as the violin. RDF statements to the first kind of '`bow`' do not automatically apply to the second kind of '`bow`'. Future work could cover approaches to ensure data quality directly during data import.

Ensuring data security

Furthermore, in recent years there is an increasing number of attacks based on data hacking and data security breaches. In order to overcome related problems we have to integrate methods to ensure that data to be imported is not altered, manipulated or falsely replicated, which we need to take care of even for publicly freely available datasets such as those of LOD. The development of whole systems like adapting [214] for RDF datasets (also in the context of LOD) is one of the key challenges in the near future.

Extend the memory hierarchy on the FPGA side

In our proposed hybrid system consisting of a CPU-based host and an FPGA accelerator card, all data on the FPGA side is stored in either registers or BRAMs. The integration of external memory present on the accelerator card, like the QDR or DDR Memory, is an important step to maximize the number of levels of the B^+-tree that can be transferred to the FPGA. So far, the user-specific circuit for the FPGA could only support the root level and three inner levels before the memory resources integrated into the chip area of the FPGA are depleted. The use of the QDR and/or DDR Memory could add another one or two levels [1]. Further, the FPGA board has two SATA-II ports that allow the direct connection of SSDs or HDDs.

Partial reconfiguration of the user-specific circuit on the FPGA

Another point is that all evaluated trees on the FPGA were precomputed and the change from one order to another was done by overwriting the complete design on the FPGA. This is not necessary because design parts that do not belong to the tree, like the PCIe connection, never change. Further, if the memory hierarchy is extended, like suggested in the last section, the controlling circuits on the chip area should also stay unaffected by a reconfiguration. Thus, partial reconfiguration would be a good way to change the tree without overwriting the complete design.

[1] Another level means an exponential growth of the number of keys by the order of the inner nodes k'.

Management component in LUPOSDATE

Further the proposed hybrid structure needs to be extended by a management component in LUPOSDATE. This component decides the correct number of levels and the order that will be sent to the FPGA. Further, it should monitor the ratio between updates and searches in order to use this knowledge to calculate estimations if the usage of the FPGA will accelerate the system. Thus, heuristics can be used based on the size of the tree and the corresponding execution times.

Adapt the design for processors with an integrated FPGA

The proposed hybrid system consists of a workstation and an FPGA on a PCIe board. This means that the exchange of data between CPU and FPGA happens through reading and writing data via PCIe in main memory. Therefore, the data has a certain latency and additional communication overhead. Some new processor units integrate an FPGA on the same die as the CPU. An example is the Intel Hardware Accelerator Research Program (HARP) [215] in which Xeon processors are extended with an FPGA. Such processor would allow to process the same data in main memory by the CPU and the FPGA as well in parallel. Hence tasks can be distributed to CPU and FPGA on a more fine-grained-level.

Acronym

3DIC	Three-Dimensional Integrated Circuit
AC	Alternating Current
ALM	Adaptive Logic Module
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
ASSP	Application-Specific Standard Product
BFS	Breadth-First Search
BIT	Bitstream
BitGen	Bitstream Generator
BJT	Bipolar Junction Transistor
BMP	Basic Multilingual Plane
BRAM	Block RAM
BTC	Billion Triples Challenge
BUFGP	Primary Global Buffer for Driving Clocks
CBD	Concise Bounded Description
CCD	Charge-Coupled Device
CJK	Chinese, Japanese and Korean
CLA	Carry-Lookahead Adder
CLB	Configurable Logic Block
CPU	Central Processing Unit
CMT	Clock Management Tile
CSB$^+$-tree	Cache Sensitive B$^+$-tree

CSS-tree	Cache-Sensitive Search tree
DBMS	Database Management System
DC	Direct Current
DDR	Double Data Rate
DIMM	Dual Inline Memory Module
DoD	Department of Defense
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random-Access Memory
DTD	Document Type Definition
EDA	Electronic Design Automation
EDVAC	Electronic Discrete Variable Automatic Computer
EDIF	Electronic Data Interchange Format
EEPROM	Electrically Erasable Programmable Read-Only Memory
e-mail	Electronic Mail
ENIAC	Electronic Numerical Integrator and Computer
EPROM	Erasable Programmable Read-Only Memory
FA	Full Adder
FAST	FPGA Accelerated Streaming Technology
FET	Field-Effect Transistor
FF	Flip-Flop
FIFO	First-In First-Out queue
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FTP	File Transfer Protocol
GAL	Generic Array Logic
GPU	Graphics Processing Unit
GPGPU	General Purpose Computing on Graphics Processing Units

HARP	Hardware Accelerator Research Program
HDD	Hard Disk Drive
HDL	Hardware Description Language
HLS	High Level Synthesis
HOLWG	High Order Language Working Group
HROW	Horizontal Clock Row
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I-tree	Insertion tree
IANA	Internet Assigned Numbers Authority
IC	Integrated Circuit
ID	Identifier
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IOB	Input/Output Block
IOT	Internet of Things
IP	Internet Protocol/Intellectual Property
IRI	Internationalized Resource Identifier
ISE	Integrated Synthesis Environment
ISP	Instruction-Set Processor
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JSON-LD	JSON for Linked Data
LAB	Logic Array Block
LAN	Local Area Network
LOD	Linking Open Data
LUPOSDATE	Logically and Physically Optimized Semantic Web Database Engine

LUT	Lookup Table
MMCM	Mixed-Mode Clock Manager
MOS	Metal–Oxide–Semiconductor
MP3	MPEG-1 or MPEG-2 Audio Layer III
MUX	Multiplexer
N3	Notation 3
NCD	Native Circuit Description
NFC	Normalization Form C
NGD	Xilinx Native Generic Database
NMC	Native Macro Library
OBUF	Output Buffer
OWL	Web Ontology Language
PAL	Programmable Array Logic
PC	Personal Computer
PCB	Printed Circuit Board
PCF	Physical Constraints File
PCIe	Peripheral Component Interconnect Express
PDF	Portable Document Format
PGA	Pin Grid Array
PIP	Programmable Interconnect Point
PLD	Programmable Logic Device
PLA	Programmable Logic Array
PLI-tree	Pointer-Less Insertion tree
PROM	Programmable read-only memory
PUA	Private Use Area
QDR	Quad Data Rate
QL	Query Language

RAM	Random-Access Memory
RC	Reconfigurable Computing
RDF	Resource Description Framework
RDFS	RDF Schema
RDQL	RDF Data Query Language
RIF	Rule Interchange Format
RM	Relational Model/Reconfigurable Module
ROM	Read-Only Memory
RP	Reconfigurable Partition
RPATH	RDF Path
RQL	RDF Query Language
RTL	Register Transfer Level
RxPATH	RDF for XPath
SATA-II	Serial AT Attachment II
S-Blades	Snippet-Blades
S.D	Standard Description
SeRQL	Sesame RDF Query Language
SFP+	Enhanced Small Form-Factor Pluggable
SIP	Supplementary Ideographic Plane/Sideways Information Passing
SMP	Supplementary Multilingual Plane
SPARQL	SPARQL Protocol And RDF Query Language
SQL	Structured Query Language
SquishQL	Squish Query Language
SRAM	Static Random-Access Memory
SSD	Solid State Drive
SSI	Small-Scale Integration
SSP	Supplementary Special-purpose Plane

SW	Semantic Web
TURTLE	Terse RDF Triple Language
TTL	Transistor–Transistor Logic
UCF	User Constraint File
UCS	Universal Character Set
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
US	United States
USB	Universal Serial Bus
UTM	Universal Turing Machine
Verilog	Verifying Logic
W3C	World Wide Web Consortium
WAN	Wide Area Network
WWW	World Wide Web
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Markup Language
XMLNS	XML Name Space
XPath	XML Path Language
XQuery	XML Query
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations
XST	Xilinx Synthesis Technology

References

[1] Sven Groppe, Dennis Heinrich, Stefan Werner, Christopher Blochwitz, and Thilo Pionteck. PatTrieSort - External String Sorting based on Patricia Tries. *Open Journal of Databases (OJDB)*, 2(1):36–50, 2015.

[2] Sven Groppe, Dennis Heinrich, Christopher Blochwitz, and Thilo Pionteck. Constructing Large-Scale Semantic Web Indices for the Six RDF Collation Orders. *Open Journal of Big Data (OJBD)*, 2(1):11–25, 2016.

[3] Dennis Heinrich, Stefan Werner, Marc Stelzner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. Hybrid FPGA Approach for a B+ Tree in a Semantic Web Database System. In *Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, June 29 – July 1 2015. IEEE.

[4] Dennis Heinrich, Stefan Werner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. *Search & Update Optimization of a B^+ Tree in a Hardware Aided Semantic Web Database System*, pages 172–182. Springer Singapore, Singapore, 2018.

[5] Dennis Heinrich, Stefan Werner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. Hardware aided Update Acceleration in a Hybrid Semantic Web Database System. *The Journal of Supercomputing*, 2018. **Note:** accepted for publication after minor revision.

[6] Sven Groppe, Thomas Kiencke, Stefan Werner, Dennis Heinrich, Marc Stelzner, and Le Gruenwald. P-LUPOSDATE: Using Precomputed Bloom Filters to Speed Up SPARQL Processing in the Cloud. *Open Journal of Semantic Web (OJSW)*, 1(2):25–55, 2014.

[7] Sven Groppe, Johannes Blume, Dennis Heinrich, and Stefan Werner. A Self-Optimizing Cloud Computing System for Distributed Storage and Processing of Semantic Web Data. *Open Journal of Cloud Computing (OJCC)*, 1(2):1–14, 2014.

[8] Sven Groppe, Dennis Heinrich, and Stefan Werner. Distributed Join Approaches for W3C-Conform SPARQL Endpoints. *Open Journal of Semantic Web (OJSW)*, 2(1):30–52, 2015.

[9] Stefan Werner, Dennis Heinrich, Marc Stelzner, Sven Groppe, Rico Backasch, and Thilo Pionteck. Parallel and Pipelined Filter Operator for Hardware-Accelerated Operator Graphs in Semantic Web Databases. In *Proceedings of the 14th IEEE International Conference on Computer and Information Technology (CIT 2014)*, Xi'an, China, September 11 - 13 2014. IEEE.

[10] Stefan Werner, Dennis Heinrich, Marc Stelzner, Volker Linnemann, Thilo Pionteck, and Sven Groppe. Accelerated join evaluation in Semantic Web databases by using FPGAs. *Concurrency and Computation: Practice and Experience*, 28(7):2031–2051, May 18 2015.

[11] Stefan Werner, Dennis Heinrich, Jannik Piper, Sven Groppe, Rico Backasch, Christopher Blochwitz, and Thilo Pionteck. Automated Composition and Execution of Hardware-accelerated Operator Graphs. In *Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, June 29 – July 1 2015. IEEE.

[12] Stefan Werner, Dennis Heinrich, Sven Groppe, Christopher Blochwitz, and Thilo Pionteck. Runtime Adaptive Hybrid Query Engine based on FPGAs. *Open Journal of Databases (OJDB)*, 3(1):21–41, 2016.

[13] Stefan Werner, Dennis Heinrich, Thilo Pionteck, and Sven Groppe. Semi-static operator graphs for accelerated query execution on FPGAs. *Microprocessors and Microsystems*, 53:178 – 189, 2017.

[14] Christopher Blochwitz, Jan Moritz Joseph, Thilo Pionteck, Rico Backasch, Stefan Werner, Dennis Heinrich, and Sven Groppe. An optimized Radix-Tree for hardware-accelerated index generation for Semantic Web Databases. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 7 - 9 2015.

[15] Christopher Blochwitz, Julian Wolff, Jan Moritz Joseph, Stefan Werner, Dennis Heinrich, Sven Groppe, and Thilo Pionteck. Hardware-Accelerated Radix-Tree Based String Sorting for Big Data Applications. In *Architecture of Computing Systems - ARCS 2017 - 30th International Conference, Vienna, Austria, April 3-6, 2017, Proceedings*, pages 47–58, 2017.

[16] Tim Berners-Lee. Information Management: A Proposal. `https://www.w3.org/History/1989/proposal.html`, March 1989. Accessed: September 25, 2018.

[17] Electra Records. Publicity photo of Queen. Members from left: John Deacon, Freddie Mercury, Brian May, and Roger Taylor. https://commons.wikimedia.org/wiki/File:Queen_1976.JPG, 1976. Accessed: September 25, 2018.

[18] WikiImages. Portait of Queen Elizabeth II. with hat. https://pixabay.com/de/k%C3%B6nigin-england-elizabeth-ii-63006/, 2012. Accessed: September 25, 2018.

[19] OpenClipart-Vectors. Playing card Queen. https://pixabay.com/de/spielkarte-k%C3%B6nigin-kartenspiel-deck-161491/, 2013. Accessed: September 25, 2018.

[20] The Governor-General. Sir Paul Reeves with the Queen of Denmark. https://gg.govt.nz/images/sir-paul-reeves-queen-denmark, 1987. Accessed: September 25, 2018.

[21] Foreign & Commonwealth Office. Her Majesty Queen Elizabeth II. https://www.flickr.com/photos/foreignoffice/8283384517, 2012. Accessed: September 25, 2018.

[22] UK Parliament. The Queen at the 2008 State Opening. https://www.flickr.com/photos/uk_parliament/2716259749, 2008. Accessed: September 25, 2018.

[23] Herbert James Gunn. Coronation Portrait of Queen Elizabeth II of the United Kingdom. https://en.wikipedia.org/wiki/File:Queen_Elizabeth_II_Coronation_Portrait_Herbert_James_Gunn.jpg, between 1953 and 1954. Accessed: September 25, 2018.

[24] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. In *Scientific American*, volume 284, pages 34–43. Scientific American, a division of Nature America, Inc., 2001.

[25] Wikipedia.org. Semantic Web Layer Cake. https://en.wikipedia.org/wiki/File:Semantic_web_stack.svg. Accessed: September 25, 2018.

[26] Unicode Inc. Unicode version 1.0.0. http://www.unicode.org/Public/reconstructed/1.0.0/UnicodeData.txt, October 1991. Accessed: 2017-08-23.

[27] Unicode Inc. Unicode version 10.0.0. http://unicode.org/versions/Unicode10.0.0/, June 2017. Accessed: 2017-08-23.

[28] Unicode Inc. Unicode scripts. http://unicode.org/charts. Accessed: 2017-08-23.

[29] Unicode Inc. Unicode BMP plane. https://en.wikipedia.org/wiki/File:Roadmap_to_Unicode_BMP.svg. Accessed: 2017-08-23.

[30] International Organization for Standardization. Homepage of the International Organization for Standardization. https://www.iso.org/home.html. Accessed: September 25, 2018.

[31] M. Duerst and M. Suignard. RFC 3987: Internationalized Resource Identifiers (IRIs). *Internet Engineering Task Force (IETF)*, http://www.ietf.org/rfc/rfc3987.txt, 2005. Accessed: September 25, 2018.

[32] World Wide Web Consortium (W3C). RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/rdf11-concepts/, February 2014. Accessed: September 25, 2018.

[33] A. Phillips and M. Davis. Tags for Identifying Languages. https://tools.ietf.org/html/bcp47, September 2009. Accessed: September 25, 2018.

[34] World Wide Web Consortium (W3C). RDF 1.1 Concepts and Abstract Syntax - Section Literals. https://www.w3.org/TR/rdf11-concepts/#section-Graph-Literal, February 2014. Accessed: September 25, 2018.

[35] World Wide Web Consortium (W3C). SPARQL 1.1 Overview. http://www.w3.org/TR/sparql11-overview/, March 2013. Accessed: September 25, 2018.

[36] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 Turtle - Terse RDF Triple Language. https://www.w3.org/TR/turtle/, February 2014. Accessed: September 25, 2018.

[37] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. JSON-LD 1.0 - A JSON-based Serialization for Linked Data. https://www.w3.org/TR/json-ld/, January 2014. Accessed: September 25, 2018.

[38] David Beckett. RDF 1.1 N-Triples - A line-based syntax for an RDF graph. https://www.w3.org/TR/n-triples/, February 2014. Accessed: September 25, 2018.

[39] Gavin Carothers. RDF 1.1 N-Quads - A line-based syntax for RDF datasets. https://www.w3.org/TR/n-quads/, February 2014. Accessed: September 25, 2018.

[40] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. https://www.w3.org/TeamSubmission/n3/, March 2011. Accessed: September 25, 2018.

[41] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-style Engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, August 2008.

[42] Thomas Neumann and Gerhard Weikum. Scalable Join Processing on Very Large RDF Graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.

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

[44] Sven Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011 edition, 5 2011.

[45] Nikos Bikakis, Chrisa Tsinaraki, Nektarios Gioldasis, Ioannis Stavrakantonakis, and Stavros Christodoulakis. *The XML and Semantic Web Worlds: Technologies, Interoperability and Integration: A Survey of the State of the Art*, pages 319–360. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[46] World Wide Web Consortium (W3C). SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/, January 2008. Accessed: September 25, 2018.

[47] World Wide Web Consortium (W3C). Resource Description Framework (RDF) Model and Syntax Specification. https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, February 1999. Accessed: September 25, 2018.

[48] World Wide Web Consortium (W3C). RDF Primer. https://www.w3.org/TR/2004/REC-rdf-primer-20040210/, February 2004. Accessed: September 25, 2018.

[49] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and semantic web query languages: A survey. In *Proceedings of the First International Conference on Reasoning Web*, pages 35–133, Berlin, Heidelberg, 2005. Springer-Verlag.

[50] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. *Reasoning Web*, 4126:1–52, 2006.

[51] Libby Miller, Andy Seaborne, and Alberto Reggiori. *Three Implementations of SquishQL, a Simple RDF Query Language*, pages 423–435. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[52] Andy Seaborne. RDQL - A Query Language for RDF. `https://www.w3.org/Submission/RDQL/`, January 2004. Accessed: September 25, 2018.

[53] HP Deutschland GmbH. German Homepage of Hewlett-Packard . `http://www8.hp.com/de/de/home.html`. Accessed: September 25, 2018.

[54] World Wide Web Consortium (W3C). W3C Opens Data on the Web with SPARQL. `https://www.w3.org/2007/12/sparql-pressrelease`, January 2008. Accessed: September 25, 2018.

[55] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In *Proceedings of the 11th international conference on World Wide Web*, pages 592–603. ACM, 2002.

[56] World Wide Web Consortium (W3C). SPARQL 1.1 Entailment Regimes. `https://www.w3.org/TR/sparql11-entailment/`, March 2013. Accessed: September 25, 2018.

[57] Jeen Broekstra and Arjohn Kampman. SeRQL: A second generation RDF query language. In *SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, 01 2003. `https://pdfs.semanticscholar.org/c77c/d08aeaee144f1d4cbf0cd6186e90d51383c9.pdf`, Accessed: September 25, 2018.

[58] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. *A Comparison of RDF Query Languages*, pages 502–517. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[59] World Wide Web Consortium (W3C). XML Path Language (XPath). `https://www.w3.org/TR/xpath/`, November 1999. Accessed: September 25, 2018.

[60] K. Matsuyama, M. Kraus, K. Kitagawa, and N. Saito. A path-based RDF query language for CC/PP and UAProf. In *IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second*, pages 3–7, March 2004.

[61] Adam Souzis. RxPath: a mapping of RDF to the XPath Data Model. http://conferences.idealliance.org/extreme/html/2006/Souzis01/EML2006Souzis01.html, 2006. Accessed: September 25, 2018.

[62] World Wide Web Consortium (W3C). XQuery 3.1: An XML Query Language. https://www.w3.org/TR/xquery-31/, March 2017. Accessed: September 25, 2018.

[63] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 3.0. https://www.w3.org/TR/2017/REC-xslt-30-20170608/, June 2017. Accessed: September 25, 2018.

[64] Jonathan Robie, Lars Marius Garshol, Steve Newcomb, Michel Biezunski, Matthew Fuchs, Libby Miller, Dan Brickley, Vassillis Christophides, and Gregorius Karvounarakis. The syntactic web. *Markup Languages*, 3(4):411–440, September 2001.

[65] World Wide Web Consortium (W3C). The Syntactic Web - Syntax and Semantics on the Web. https://www.w3.org/XML/2002/08/robie.syntacticweb.html, August 2002. Accessed: September 25, 2018.

[66] Norman Walsh. RDF Twig: Accessing RDF Graphs in XSLT. http://rdftwig.sourceforge.net/paper/index.html, August 2003. Accessed: September 25, 2018.

[67] Ahmad Yaseen. Querying remote data sources in SQL Server. https://www.sqlshack.com/querying-remote-data-sources-in-sql-server/, June 2016. Accessed: September 25, 2018.

[68] Oracle Website. Official Website of the Corporation. https://www.oracle.com/index.html. Accessed: September 25, 2018.

[69] Oracle Help Center. Distributed Queries. https://docs.oracle.com/cd/B19306_01/server.102/b14200/queries010.htm. Accessed: September 25, 2018.

[70] Nokia Oyj. Official Website of Nokia. https://www.nokia.com. Accessed: September 25, 2018.

[71] Patrick Stickler. CBD - Concise Bounded Description. https://www.w3.org/Submission/CBD/, June 2005. Accessed: September 25, 2018.

[72] Sven Groppe. LUPOSDATE Semantic Web Database Management System on Github. https://github.com/luposdate/luposdate. Accessed: September 25, 2018.

[73] Robert K. Dueck. *Digital Design with CPLD Applications and VHDL*. Thomson Delmar Learning, 2. edition, 2005.

[74] Georgi Dalakov. 'History of Computers' Website. http://history-computer.com/MechanicalCalculators/Pioneers/Schickard.html. Accessed: September 25, 2018.

[75] Blaise Pascal. La pascaline, la 'machine qui relève du défaut de la mémoire'. *http://www.bibnum.education.fr/calculinformatique/calcul/la-pascaline-la-machine-qui-releve-du-defaut-de-la-memoire*, 1645. Accessed: September 25, 2018.

[76] Charles Babbage. *Babbage's Calculating Engine - Being a Collection of Papers Relating to Them; Their History, and Construction*. Cambridge University Press, https://monoskop.org/images/4/40/Babbage_Charles_Calculating_Engines.pdf, 1889. Accessed: September 25, 2018.

[77] Luigi Federico Menabrea. Sketch of the Analytical Engine. http://www.fourmilab.ch/babbage/sketch.html, October 1842. Accessed: September 25, 2018.

[78] Friedrich L. Bauer. *Origins and Foundations of Computing*. Springer Science & Business Media, https://books.google.de/books?id=y4uTaLiN-wQC&printsec=frontcover, 2009. Accessed: September 25, 2018.

[79] R. Rojas. Konrad Zuse's legacy: the architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2):5–16, Apr 1997. http://ed-thelen.org/comp-hist/Zuse_Z1_and_Z3.pdf, Accessed: September 25, 2018.

[80] R. Rojas. How to make Zuse's Z3 a universal computer. *IEEE Annals of the History of Computing*, 20(3):51–54, Jul 1998. http://www.inf.fu-berlin.de/users/rojas/1997/Universal_Computer.pdf, Accessed: September 25, 2018.

[81] I. Bernard Cohen. *Howard Aiken: Portrait of a Computer Pioneer*, volume 13 of *History of Computing*. The MIT Press, 12 1999.

[82] Staff of the computation laboratory. *A manual of operation for the automatic sequence controlled calculator*. Havard University Press, 1946. http://www.bitsavers.org/pdf/harvard/MarkI_operMan_1946.pdf Accessed: September 25, 2018.

[83] Paul E. Ceruzzi. *A History of Modern Computing*. MIT Press, Cambridge, MA, USA, 1998. https://doc.lagout.org/science/0_Computer%20Science/0_Computer%20History/A%20History%20of%20Modern%20Computing,%202nd.pdf, Accessed: September 25, 2018.

[84] Marcello Morelli. *Dalle calcolatrici ai computer degli anni Cinquanta*. FrancoAngeli, 2001. https://books.google.de/books?id=p5GszzXR550C&pg=PA177, Accessed: September 25, 2018.

[85] Raúl Rojas and Ulf Hashagen, editors. *The First Computers: History and Architectures*. MIT Press, Cambridge, MA, USA, 2000. https://books.google.de/books?id=nDWPW9uwZPAC&pg=PA177&lpg=PA177, Accessed: September 25, 2018.

[86] Martin H. Weik. *Ballistic Research Laboratories Report No. 971: A Survey of Domestic Electronic Digital Computing Systems*. United States Department of Commerce Office of Technical Services, 1955. http://ed-thelen.org/comp-hist/BRL-e-h.html#ENIAC, Accessed: September 25, 2018.

[87] Leslie William Turner, editor. *Electronics engineer's reference book*. Newnes-Butterworth, 4th edition, April 1976. https://books.google.de/books?id=2N0gBQAAQBAJ&lpg=PP8&hl=de&pg=PP257, Accessed: September 25, 2018.

[88] J. Bardeen and W. H. Brattain. The transistor, a semi-conductor triode. *Phys. Rev.*, 74:230–231, Jul 1948. https://journals.aps.org/pr/pdf/10.1103/PhysRev.74.230, Accessed: September 25, 2018.

[89] W. F. Brinkman, D. E. Haggan, and W. W. Troutman. A history of the invention of the transistor and where it will lead us. *IEEE Journal of Solid-State Circuits*, 32(12):1858–1865, Dec 1997. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.205.6885&rep=rep1&type=pdf, Accessed: September 25, 2018.

[90] Jack S. Kilby. Invention of the integrated circuit. *IEEE Transactions of electron devices*, ED-23(7):648–654, July 1976. http://corphist.computerhistory.org/corphist/documents/doc-496d289787271.pdf, Accessed: September 25, 2018.

[91] J.L. Buie. Coupling transistor logic and other circuits. http://www.google.com.pg/patents/US3283170, November 1 1966. US Patent 3,283,170, Accessed: September 25, 2018.

[92] Engineering Staff of Texas Instruments Incorporated - Semicondutor Group, editor. *The TTL Data Book for Design Engineers*. Texas Instruments Incorporated, second edition, 1976. http://www.smcelectronics.com/DOWNLOADS/1976-TTL%20DATABOOK.PDF, 3-7, Accessed: September 25, 2018.

[93] Julius Edgar Lilienfeld. Method and apparatus for controlling electric currents. https://www.google.com/patents/US1745175 or http://www.freepatentsonline.com/1745175.pdf, January 28 1930. US Patent 1,745,175, Accessed: September 25, 2018.

[94] J.A. Hoerni. Method of manufacturing semiconductor devices. https://www.google.com/patents/US3025589, March 20 1962. US Patent 3,025,589, Accessed: September 25, 2018.

[95] Intel Corporation. Website of the Intel Corporation. www.intel.com, 2018. Accessed: September 25, 2018.

[96] Intel Corporation. Intel's first microprocessor website. https://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html, 2018. Accessed: September 25, 2018.

[97] John von Neumann. First Draft of a Report on the EDVAC. https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf, June 1945. Accessed: September 25, 2018.

[98] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. http://onlinelibrary.wiley.com/doi/10.1112/plms/s2-42.1.230/epdf, Accessed: September 25, 2018.

[99] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1938. http://onlinelibrary.wiley.com/doi/10.1112/plms/s2-43.6.544/epdf, Accessed: September 25, 2018.

[100] Intel Corporation. Microprocessor Quick Reference Guide. https://www.intel.com/pressroom/kits/quickreffam.htm, December 2008. Accessed: September 25, 2018.

[101] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965. http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf, Accessed: September 25, 2018.

[102] G. E. Moore. Progress in digital integrated electronics [Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]. *IEEE Solid-State Circuits Society Newsletter*, 11(3):36–37, Sept 2006. http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf, Accessed: September 25, 2018.

[103] Intel Corporation. Excerpts from A Conversation with Gordon Moore: Moore's Law. http://large.stanford.edu/courses/2012/ph250/lee1/docs/Excepts_A_Conversation_with_Gordon_Moore.pdf, 2005. Accessed: September 25, 2018.

[104] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978. http://doi.acm.org/10.1145/359576.359579, Accessed: September 25, 2018.

[105] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. http://delivery.acm.org/10.1145/220000/216588/p20-wulf.pdf, Accessed: September 25, 2018.

[106] D. H. Woo, N. H. Seong, D. L. Lewis, and H. H. S. Lee. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.604.4875&rep=rep1&type=pdf, Accessed: September 25, 2018.

[107] R. Finlayson. A More Loss-Tolerant RTP Payload Format for MP3 Audio. https://tools.ietf.org/html/rfc5219, February 2008. Accessed: September 25, 2018.

[108] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.

[109] Andy Patrizio. Intel plans hybrid CPU-FPGA chips. https://www.networkworld.com/article/3230929/data-center/intel-unveils-hybrid-cpu-fpga-plans.html, October 2017. Accessed: September 25, 2018.

[110] Chantelle Dubois. Intel to Introduce New CPU-FPGA Hybrid Chip Supported by Acceleration Stack. https://www.allaboutcircuits.com/news/intel-to-introduce-new-cpu-fpga-hybrid-chip-supported-by-acceleration-stack/, October 2017. Accessed: September 25, 2018.

[111] The Dini Group, Inc. Website of the DNPCIe_10G_HXT_LL board. http://www.dinigroup.com/web/DNPCIe_10G_HXT_LL.php. Accessed: September 25, 2018.

[112] The Dini Group, Inc. Official Website of the Dini Group. http://www.dinigroup.com. Accessed: September 25, 2018.

[113] Xilinx Inc. Virtex-6 Family Overview. https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, August 2015. Accessed: September 25, 2018.

[114] Xilinx Inc. Official Website of Xilinx. https://www.xilinx.com/. Accessed: September 25, 2018.

[115] The Dini Group, Inc. Front picture of the DNPCIe_10G_HXT_LL board. http://www.dinigroup.com/product/data/DNPCIe_10G_HXT-LL/images/front.jpg. Accessed: September 25, 2018.

[116] PCI-SIG. PCI Express Base Specification Revision 2.0. http://pcisig.com/specifications/pciexpress?field_technology_value%5B%5D=express&field_revision_value%5B%5D=2&speclib=Base+2006, December 2006. Accessed: September 25, 2018.

[117] Intel Corporation. Official Website of Intel FPGAs. https://www.altera.com/. Accessed: September 25, 2018.

[118] Altera Corporation. Stratix III Device Handbook. Volume 1:1–6, 2011. https://books.google.de/books?id=2N0gBQAAQBAJ&lpg=PP8&hl=de&pg=PP257, Accessed: September 25, 2018.

[119] Xilinx Inc. Virtex-6 FPGA Configurable Logic Block. https://www.xilinx.com/support/documentation/user_guides/ug364.pdf, February 2012. Accessed: September 25, 2018.

[120] Gerald B. Rosenberger. Simultaneous carry adder. https://www.google.com/patents/US2966305, December 27 1960. US Patent 2,966,305, Accessed: September 25, 2018.

[121] Xilinx Inc. Virtex-6 FPGA Memory Resources. https://www.xilinx.com/support/documentation/user_guides/ug363.pdf, February 2014. Accessed: September 25, 2018.

[122] The Dini Group, Inc. Block Diagram of the DNPCIe_10G_HXT_LL board. http://www.dinigroup.com/product/data/DNPCIe_10G_HXT-LL/images/DNPCIe_10G_HXT-LL_blk_v10a.png. Accessed: September 25, 2018.

[123] The Dini Group, Inc. Product Brief of the DNPCIe_10G_HXT_LL board (High Resolution). http://www.dinigroup.com/product/data/DNPCIe_10G_HXT-LL/files/DNPCIe_10G_HXT_LL_v11a_hi.pdf, April 2012. Accessed: September 25, 2018.

[124] SFF Committee. SFF-8431 Specifications for Enhanced Small Form Factor Pluggable Module SFP+. https://ta.snia.org/higherlogic/ws/public/download/268/SFF-8431.PDF, July 2009. Accessed: September 25, 2018.

[125] Xilinx Inc. Virtex-6 FPGA Clocking Resources. https://www.xilinx.com/support/documentation/user_guides/ug362.pdf, January 2014. Accessed: September 25, 2018.

[126] Intel Corporation. Official Website of Intel Quartus Prime. https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html. Accessed: September 25, 2018.

[127] Mentor Graphics Corporation. Official Website of ModelSim. https://www.mentor.com/products/fpga/verification-simulation/modelsim/. Accessed: September 25, 2018.

[128] Mentor Graphics Corporation. Official Website of Mentor Graphics Corporation. https://www.mentor.com/. Accessed: September 25, 2018.

[129] Xilinx Inc. Official Website of Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html. Accessed: September 25, 2018.

[130] Xilinx Inc. Official Website of ISE Design Suite. https://www.xilinx.com/products/design-tools/ise-design-suite.html. Accessed: September 25, 2018.

[131] Sigasi. Official Website of Sigasi Studio. http://www.sigasi.com/products/. Accessed: September 25, 2018.

[132] Sigasi. Official Website of Sigasi. http://www.sigasi.com/. Accessed: September 25, 2018.

[133] Xilinx Inc. Command Line Tools User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/devref.pdf, July 2012. Accessed: September 25, 2018.

[134] Mario R. Barbacci. A comparison of register transfer languages for describing computers and digital systems. Technical report, Carnegie Mellon University, 1973. http://repository.cmu.edu/cgi/viewcontent.cgi?article=2666&context=compsci, Accessed: September 25, 2018.

[135] Chester Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw Hill Book Company, 1971. http://gordonbell.azurewebsites.net/cgb%20files/computer%20structures%20readings%20and%20examples%201971.pdf, Accessed: September 25, 2018.

[136] Wikipedia.org. RTL Example toggler. https://en.wikipedia.org/wiki/File:Register_transfer_level_-_example_toggler.svg. Accessed: September 25, 2018.

[137] M. R. Barbacci, S. Grout, G. Lindstrom, and M. P. Maloney. Ada as a hardware description language : an initial report. Technical report, Carnegie-Mellon Univ., Dept. of Computer Science, 1984. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.958.6770&rep=rep1&type=pdf, Accessed: September 25, 2018.

[138] Steve Golson. Oral History of Philip Raymond 'Phil' Moorby. http://archive.computerhistory.org/resources/access/text/2013/11/102746653-05-01-acc.pdf, April 2013. Accessed: September 25, 2018.

[139] EE Times. Verilog's inventor nabs EDA's Kaufman award. https://www.eetimes.com/document.asp?doc_id=1157349, July 2005. Accessed: September 25, 2018.

[140] IEEE Standards Association. IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language. *IEEE Std 1364-1995*, pages 1–688, Oct 1996.

[141] Xilinx Inc. XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/xst_v6s6.pdf, October 2012. Accessed: September 25, 2018.

[142] Xilinx Inc. Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for Schematic Designs. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_scm.pdf, October 2013. Accessed: September 25, 2018.

[143] H. J. Kahn and R. F. Goldman. The Electronic Design Interchange Format EDIF: Present and Future. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pages 666–671, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[144] Xilinx Inc. Partial Reconfiguration User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/ug702.pdf, April 2013. Accessed: September 25, 2018.

[145] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[146] Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings. Sea Cucumber: A Synthesizing Compiler for FPGAs. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 875–885, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. https://pdfs.semanticscholar.org/c6c3/1c2b1899fc1a02339caacd92433c536fb51f.pdf, Accessed: September 25, 2018.

[147] Y Explorations (YXI). eXCite Product Website. http://www.yxi.com/products.php. Accessed: September 25, 2018.

[148] Mentor Graphics Corporation. Catapult HLS Product Website. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/. Accessed: September 25, 2018.

[149] Donald Knuth. *Sorting and Searching*. Number 3 in The Art of Computer Programming. Addison-Wesley Professional, 2. edition, 1998. ISBN 0-201-89685-0.

[150] P. Widmayer T. Ottmann. *Algorithmen und Datenstrukturen (German Edition)*. Spektrum Akademischer Verlag, 4. edition, 1 2002.

[151] Georgi Maximowitsch Adelson-Velski and Jewgeni Michailowitsch Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[152] Raymond Miller, Nicholas Pippenger, Arnold Rosenberg, and Lawrence Snyder. Optimal 2, 3-trees. *SIAM J. Comput.*, 8(1):42–59, 1979.

[153] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[154] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, June 1979.

[155] Jun Rao and Kenneth A. Ross. Making B+- Trees Cache Conscious in Main Memory. *SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 29(2):475–486, May 2000.

[156] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962. `http://dx.doi.org/10.1093/comjnl/5.1.10`, Accessed: September 25, 2018.

[157] J. W. J. Williams. Algorithms; Algorithm 232 - Heapsort. *Communications of the ACM*, 7(6):347–349, June 1964. `http://doi.acm.org/10.1145/512274.512284`, Accessed: September 25, 2018.

[158] Edward H. Friend. Sorting on Electronic Computer Systems. *Journal of the ACM (JACM)*, 3(3):134–168, 1956. `http://doi.acm.org/10.1145/320831.320833`, Accessed: September 25, 2018.

[159] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data structures and algorithms*. Addison-Wesley, Reading, Mass., 1983.

[160] Linked Data. Official Website of Linked Data. `http://www.linkeddata.org`, Accessed: September 25, 2018.

[161] Sven Groppe and Jinghua Groppe. External Sorting for Index Construction of Large Semantic Web Databases. In *Proceedings of the 25th ACM Symposium on Applied Computing, Vol. II (ACM SAC 2010)*, pages 1373–1380, Sierre, Switzerland, March 2010. ACM.

[162] R. Miller, N. Pippenger, A. Rosenberg, and L. Snyder. Optimal 2-3 trees. In *IBM Research Lab. Yorktown Heights*, NY, 1977.

[163] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4. edition, March 2011.

[164] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3. edition, 2009.

[165] Ranjan Sinha and Justin Zobel. Cache-conscious Sorting of Large Sets of Strings with Dynamic Tries. *Journal of Experimental Algorithmics (JEA)*, 9, December 2004.

[166] R. Angrish and D. Garg. Efficient string sorting algorithms: Cache-aware and cache-oblivious. *International Journal of Soft Computing and Engineering (IJSCE)*, 1, 2011.

[167] Ranjan Sinha and Justin Zobel. Efficient Trie-based Sorting of Large Sets of Strings. In *Proceedings of the 26th Australasian Computer Science Conference - Volume 16*, ACSC '03, pages 11–18, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[168] R. Sinha and J. Zobel. Using Random Sampling to Build Approximate Tries for Efficient String Sorting. *Journal of Experimental Algorithmics (JEA)*, 10:2.10, 2005.

[169] John Yiannis and Justin Zobel. Compression Techniques for Fast External Sorting. *The VLDB Journal*, 16(2):269–291, April 2007.

[170] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 540–548, New York, NY, USA, 1997. ACM.

[171] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960. `http://doi.acm.org/10.1145/367390.367400`, Accessed: September 25, 2018.

[172] Zbyněk Falt, Jan Bulánek, and Jakub Yaghob. On parallel sorting of data streams. In *Advances in Databases and Information Systems*, pages 69–77. Springer, 2013.

[173] Julian Seward. Official Website of bzip2 and libbzip2. `http://www.bzip.org/`. Accessed: September 25, 2018.

[174] Chris Nyberg and Mehul Shah. Sort Benchmark Home Page. `http://sortbenchmark.org/`. Accessed: September 25, 2018.

[175] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. Alphasort: A cache-sensitive parallel external sort. *VLDB J.*, 4(4):603–627, 1995.

[176] Andreas Harth and Sean Bechhofer. A new application award - Semantic Web Challenge. http://challenge.semanticweb.org/, 2014. Accessed: September 25, 2018.

[177] Andreas Harth. Billion Triples Challenge 2012 Dataset. http://km.aifb.kit.edu/projects/btc-2012/, 2012. Accessed: September 25, 2018.

[178] World Wide Web Consortium (W3C). W3C Data Activity - Building the Web of Data. https://www.w3.org/2013/data/, 2016. Accessed: September 25, 2018.

[179] Dan Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation. http://www.w3.org/TR/rdf-schema/, 2004. Accessed: September 25, 2018.

[180] World Wide Web Consortium (W3C). *OWL 2 Web Ontology Language: Document Overview (Second Edition)*. W3C Recommendation, 11 December 2012. http://www.w3.org/TR/owl2-overview/, Accessed: September 25, 2018.

[181] LOD2. Statistics of the Linked Open Data 2 Project. http://stats.lod2.eu/, Accessed: September 25, 2018.

[182] LOD2. Official Website of the Linked Open Data 2 Project. http://lod2.eu, Accessed: September 25, 2018.

[183] Douglas Laney. 3D Data Management: Controlling Data Volume, Velocity and Variety. Gartner, http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf, 2001. Accessed: September 25, 2018.

[184] M. A. Khan, M. F. Uddin, and N. Gupta. Seven V's of Big Data understanding Big Data to extract value. In *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education*, pages 1–5, 2014.

[185] Kirk Borne. Top 10 Big Data Challenges – A Serious Look at 10 Big Data V's. Gartner, https://www.mapr.com/blog/top-10-big-data-challenges-serious-look-10-big-data-vs, 2014. Accessed: September 25, 2018.

[186] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[187] H. H. Seward. *2.4.6 Internal Sorting by Floating Digital Sort, Information sorting in the application of electronic digital computers to business operations, Master's thesis, Report R-232.* Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[188] David J. DeWitt. Direct - a multiprocessor organization for supporting relational data base management systems. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, ISCA '78, pages 182–189, New York, NY, USA, 1978. ACM.

[189] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, August 2009.

[190] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on FPGAs. *The VLDB Journal*, 21(1):1–23, 2012.

[191] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM.

[192] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: A Query Compiler for FPGAs. *Proceedings of the VLDB Endowment*, 2(1):229–240, August 2009.

[193] IBM Corporation. Official Website of IBM. `http://www.ibm.com`, Accessed: September 25, 2018.

[194] Phil Francisco. IBM PureData System for Analytics Architecture: A Platform for High Performance Data Warehousing and Analytics, 2010. `http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf`, Accessed: September 25, 2018.

[195] Rene Mueller and Jens Teubner. FPGA: What's in It for a Database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM.

[196] Rene Mueller and Jens Teubner. FPGAs: A New Point in the Database Design Space. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 721–723, New York, NY, USA, 2010. ACM.

[197] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1159–1162, New York, NY, USA, 2010. ACM.

[198] K. Torp, L. Mark, and Christian S. Jensen. Efficient differential timeslice computation. *Knowledge and Data Engineering, IEEE Transactions on*, 10(4):599–611, Jul 1998.

[199] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[200] Stefan Manegold, A. Peter Boncz, and L. Martin Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[201] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in Flash Memory SSD Technology for Enterprise Database Applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 863–870, New York, NY, USA, 2009. ACM.

[202] Y. Kang, Y. s. Kee, E. L. Miller, and C. Park. Enabling cost-effective data processing with smart SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2013.

[203] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[204] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[205] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[206] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[207] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21:1–21:39, December 2009.

[208] Bingsheng He and Jeffrey Xu Yu. High-throughput Transaction Executions on Graphics Processors. *Proceedings of the VLDB Endowment*, 4(5):314–325, February 2011.

[209] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache Query Co-processing on Coupled CPU-GPU Architectures. *Proceedings of the VLDB Endowment*, 8(4):329–340, December 2014.

[210] Xuntao Cheng, Bingsheng He, and Chiew Tong Lau. Energy-efficient query processing on embedded cpu-gpu architectures. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 10:1–10:7, New York, NY, USA, 2015. ACM.

[211] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, September 2010.

[212] Christian Plessl. Accelerating Scientific Computing with Massively Parallel Computer Architectures, 2012. `http://www.imprs-dynamics.mpg.de/pdfs/Plessl_talk.pdf`, Accessed: September 25, 2018.

[213] Dell Technologies Inc. Product website of Dell Precision T3610 Tower. `http://www.dell.com/de/unternehmen/p/precision-t3610-workstation/pd`, Accessed: September 25, 2018.

[214] Victor Chang, Yen-Hung Kuo, and Muthu Ramachandran. Cloud computing adoption framework: A security framework for business clouds. *Future Generation Computer Systems*, 57:24 – 41, 2016.

[215] Intel Corporation. Official Website of the Hardware Accelerator Research Program. https://software.intel.com/en-us/hardware-accelerator-research-program/. Accessed: September 25, 2018.

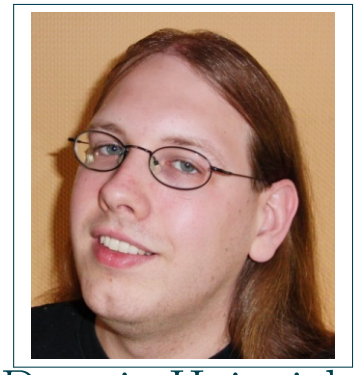
Lists

List of Figures

List of Listings

List of Tables

Curriculum Vitae



Dennis Heinrich

Persönliches

Geburtstag	05. Januar 1987
Geburtsort	Bad Segeberg

Beruflicher Werdegang

seit 09/2018	Elektronikentwickler *Basler AG*
09/2013 – 03/2018	Wissenschaftlicher Mitarbeiter Institut für Informationssysteme *Universität zu Lübeck*

Ausbildung

04/2011 – 07/2013	Masterstudium der Informatik (M.Sc.) mit Nebenfach Medieninformatik *Universität zu Lübeck*
10/2007 – 03/2011	Bachelorstudium der Informatik (B.Sc.) mit Nebenfach Medieninformatik *Universität zu Lübeck*
10/2006 – 06/2007	Grundwehrdienst *Panzergrenadierbataillon 182, Bad Segeberg*
08/2003 – 06/2006	Fachgymnasium Technik (Abitur) *Berufsschule Bad Segeberg, Bad Segeberg*
08/1997 – 06/2003	Mittelschule (Realschulabschluss) *Realschule am Seminarweg, Bad Segeberg*

List of Personal Publications

Dennis Heinrich, Stefan Werner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. Hardware aided Update Acceleration in a Hybrid Semantic Web Database System. *The Journal of Supercomputing*, 2018.

Dennis Heinrich, Stefan Werner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. *Search & Update Optimization of a B^+ Tree in a Hardware Aided Semantic Web Database System*, pages 172–182. Springer Singapore, Singapore, 2018.

Christopher Blochwitz, Julian Wolff, Jan Moritz Joseph, Stefan Werner, **Dennis Heinrich**, Sven Groppe, and Thilo Pionteck. Hardware-Accelerated Radix-Tree Based String Sorting for Big Data Applications. In *Architecture of Computing Systems - ARCS 2017 - 30th International Conference, Vienna, Austria, April 3-6, 2017, Proceedings*, pages 47–58, 2017.

Stefan Werner, **Dennis Heinrich**, Thilo Pionteck, and Sven Groppe. Semi-static operator graphs for accelerated query execution on FPGAs. *Microprocessors and Microsystems*, 53:178 – 189, 2017.

Sven Groppe, **Dennis Heinrich**, Christopher Blochwitz, and Thilo Pionteck. Constructing Large-Scale Semantic Web Indices for the Six RDF Collation Orders. *Open Journal of Big Data (OJBD)*, 2(1):11–25, 2016.

Stefan Werner, **Dennis Heinrich**, Sven Groppe, Christopher Blochwitz, and Thilo Pionteck. Runtime Adaptive Hybrid Query Engine based on FPGAs. *Open Journal of Databases (OJDB)*, 3(1):21–41, 2016.

Christopher Blochwitz, Jan Moritz Joseph, Thilo Pionteck, Rico Backasch, Stefan Werner, **Dennis Heinrich**, and Sven Groppe. An optimized Radix-Tree for hardware-accelerated index generation for Semantic Web Databases. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 7 - 9 2015.

Dennis Heinrich, Stefan Werner, Marc Stelzner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. Hybrid FPGA Approach for a B+ Tree in a Semantic Web Database System. In *Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, June 29 – July 1 2015. IEEE.

Stefan Werner, **Dennis Heinrich**, Jannik Piper, Sven Groppe, Rico Backasch, Christopher Blochwitz, and Thilo Pionteck. Automated Composition and Execution of Hardware-accelerated Operator Graphs. In *Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, June 29 – July 1 2015. IEEE.

Stefan Werner, **Dennis Heinrich**, Marc Stelzner, Volker Linnemann, Thilo Pionteck, and Sven Groppe. Accelerated join evaluation in Semantic Web databases by using FPGAs. *Concurrency and Computation: Practice and Experience*, 28(7):2031–2051, May 18 2015.

Sven Groppe, **Dennis Heinrich**, and Stefan Werner. Distributed Join Approaches for W3C-Conform SPARQL Endpoints. *Open Journal of Semantic Web (OJSW)*, 2(1):30–52, 2015.

Sven Groppe, **Dennis Heinrich**, Stefan Werner, Christopher Blochwitz, and Thilo Pionteck. PatTrieSort - External String Sorting based on Patricia Tries. *Open Journal of Databases (OJDB)*, 2(1):36–50, 2015.

Stefan Werner, **Dennis Heinrich**, Marc Stelzner, Sven Groppe, Rico Backasch, and Thilo Pionteck. Parallel and Pipelined Filter Operator for Hardware-Accelerated Operator Graphs in Semantic Web Databases. In *Proceedings of the 14th IEEE International Conference on Computer and Information Technology (CIT 2014)*, Xi'an, China, September 11 - 13 2014. IEEE.

Sven Groppe, Johannes Blume, **Dennis Heinrich**, and Stefan Werner. A Self-Optimizing Cloud Computing System for Distributed Storage and Processing of Semantic Web Data. *Open Journal of Cloud Computing (OJCC)*, 1(2):1–14, 2014.

Sven Groppe, Thomas Kiencke, Stefan Werner, **Dennis Heinrich**, Marc Stelzner, and Le Gruenwald. P-LUPOSDATE: Using Precomputed Bloom Filters to Speed Up SPARQL Processing in the Cloud. *Open Journal of Semantic Web (OJSW)*, 1(2):25–55, 2014.