Master's thesis ICA-3492206

Bridging the gap between Big Genome Data Analysis and Database Management Systems

Supervisors:
Ying Zhang (CWI)
Hans Philippi (UU)

Additional supervisors:
Tobias Marschall (CWI)
Yağız Kargın (CWI)

Author:
Robin Cijvat

February 3, 2014

Abstract

The bioinformatics field has encountered a data deluge over the last years, due to increasing speed and decreasing cost of DNA sequencing technology. Today, sequencing the DNA of a single genome only takes about a week, and it can result in up to a terabyte of data. The sequencing data are usually stored in files, and specialized tools have been designed to analyze and manage them. Despite of these tools, bioinformaticians are still exposed to many data management hurdles when analyzing these files, which often leads to excessively time consuming tasks.

In this thesis, we accurately map the needs of bioinformaticians by defining a set of use cases that reflect the everyday analysis that is applied on genetic data. We propose a modern-DBMS based approach, to analyze and manage genetic data file repositories. We identify the pros and cons of this method compared to the traditional file-based approach.

Additionally, we experimented with a novel in-situ approach, where the DBMS applies Just-In-Time ETL (Extract-Transform-Load) on the original files instead of loading all data from these files up front. A major advantage of this approach is that it greatly reduces the data-to-query time, since not all data are loaded in the DBMS during initialization. Other advantages include the decrease in storage requirements and the reduced data duplication.

With this project, we have taken the first step towards the adaptation of the state-of-the-art database technology to accelerate genetic data analytics. The preliminary results presented in this thesis are highly promising and they open up a plethora of new research opportunities.

Contents

1 Introduction **4**
1.1 Research scope . . . 5
1.2 Outline . . . 6

2 Related work **7**

3 DNA sequence alignment data **8**
3.1 DNA sequencing and alignment . . . 8
3.2 The BAM file format . . . 9
3.2.1 Alignment fields . . . 10
3.2.2 Sortedness of BAM files . . . 12

4 Practical use cases **13**
4.1 Notation . . . 13
4.1.1 General . . . 13
4.1.2 Set notation . . . 14
4.1.3 Result . . . 14
4.1.4 Sorting . . . 14
4.1.5 Grouping . . . 14
4.2 Functionality over alignments . . . 15
4.3 Simple use cases . . . 15
4.4 Advanced use cases . . . 17

5 Implementation **24**
5.1 A DBMS implementation . . . 24
5.1.1 Database schemas . . . 24
5.1.2 Loading data into a storage schema . . . 26
5.1.3 User defined functions . . . 28
5.1.4 Solving use cases . . . 28
5.2 The traditional approach . . . 31
5.2.1 Elementary data structures . . . 31
5.2.2 Functionality defined over data structures . . . 34
5.2.3 Sortedness of the input . . . 36
5.2.4 Use case implementations . . . 36

6 Experiments **40**
6.1 Setup . . . 40
6.1.1 Hardware and software . . . 40
6.1.2 Our file repository . . . 40
6.1.3 Initialization . . . 42
6.1.4 Solving use cases . . . 42
6.1.5 Output . . . 43
6.2 Results . . . 44
6.2.1 Initialization . . . 44
6.2.2 Solving use cases using the full-output approach . . . 45
6.2.3 Solving use cases using the minimal-output approach . . . 49
6.2.4 Correlations in the results . . . 52
6.2.5 Pros and cons of our implementations . . . 56

7 In-situ processing **57**
7.1 The Data Vault Framework (DVF) . . . 57
7.2 Applying the DVF to BAM data . . . 58
7.2.1 Initializing the Data Vault . . . 58
7.2.2 Mounting data from BAM file repository . . . 60
7.3 Experimental results . . . 61
7.3.1 Initialization . . . 61
7.3.2 Solving use cases . . . 62

8 Conclusions and future work **65**
8.1 Future work . . . 65
8.1.1 Improving the Data Vault implementation . . . 66
8.1.2 Experimenting . . . 66
8.1.3 Compressed file systems . . . 67
8.1.4 Querying *EXTRA* data . . . 67
8.1.5 Data visualization . . . 67
8.1.6 Distributed file systems . . . 67

A Pseudocode for functionality over alignments **70**

B SQL code for initialization of DBMS implementations **72**
B.1 Schema to store BAM header data . . . 72
B.2 Schema to store alignment data of BAM file with file ID i . . . 73
B.3 Schema to store alignment pairs of BAM file with file ID i . . . 74
B.4 SQL code for dividing alignments over the paired schema . . . 77

C SQL queries for the straightforward storage schema **79**

D SQL queries for the pairwise storage schema **87**

E Plots showing implementation performance **93**
E.1 Small files – full-output approach 93
E.2 Small files – minimal-output approach performance increase 99
E.3 Big files – ‘minimal-output’ approach 104
E.4 Small files – Straightforward storage schema versus Data Vault implementation 105

Chapter 1

Introduction

In the last years, the bioinformatics field has encountered a data deluge, where the rate at which data is generated far exceeds the conventional query processing capabilities for many applications. This is mainly due to the Human Genome Project [15], in which DNA sequencing was industrialized. As a result of this project, it nowadays costs only about 2000 USD and less than one week on a single sequencing device to sequence a genome. These new high-throughput sequencing methods, also known as Next Generation Sequencing (NGS), are applied by several projects across the world. These projects together already sequenced tens of thousands of genomes [14], with a storage requirement of hundreds of gigabytes per genome.

The sequencing of a genome is a process that results in millions of short sequences. Most NGS applications therefore rely on sequence alignment techniques, where the millions of sequences are aligned to a reference genome, as a first analysis step. The result of sequence alignment is stored in files that are often on a terabyte scale. The reason for these huge file sizes is that they need to store the millions of sequences that follow from the sequencing process, including meta data for every sequence. Bioinformaticians use these files as the input for various analyses, ranging from simple consistency checking to complex data analysis algorithms. An example of such algorithms is analyzing if there exists a correlation between some mutations in the sequences and whether or not the organism in question is diagnosed with cancer. Another example is searching for and analyzing the differences in the sequences of two genomes. To efficiently conduct their daily research work, bioinformaticians need easy to use and fast exploration tools to access and analyze the sequence alignment data.

There are several reasons why a Database Management System (DBMS) would be beneficial for doing bioinformatics data analysis. First of all, alignment data can be easily stored in a relational database schema, since alignment data is stored in a tabular form already in the BAM file. Furthermore, communication with a DBMS goes through a declarative query language, such as SQL, which is widely known and heavily used in many fields. Even those who do not have much experience with such a language will have no problem updating their knowledge to a reasonable extent within only a few hours. A big advantage of communicating with scientific data through a DBMS is the significant reduction of data-to-knowledge time and the ease of maintenance. With a declarative query language, the DBMS users only need to specify what they want to analyze, instead of programming how the analysis should be done exactly, which is

extremely time consuming and error-prone. Additionally, a DBMS provides users with a transaction engine that makes it possible for multiple users to work on the same data simultaneously, without corrupting the data.

However, applying DBMS technology on bioinformatical data analysis is not a trivial task. The most straightforward approach is to load all genetic data directly into a DBMS. This comes with a high data-to-query time and storage requirements, especially if the original data is compressed and the DBMS has to store the decompressed data. Another difficulty that rises is the consistency of the duplicated data. If something changes in the original data, the DBMS should be updated accordingly and vice versa. To overcome these problems, a DBMS could be set up to work directly on the external files without loading any of this external data into its own storage structures, as is done in [1] for CSV files. This is however not a trivial task either when working with compressed genetic data. Parsing and decompressing large amounts of genetic data during query time slows data analysis down tremendously. Therefore, a hybrid approach that combines the pros and cons of both solutions would be a good aim.

1.1 Research scope

The most challenging data management issues in bioinformatics are identified in [6, 14, 9]. These challenges include coming up with proper data structures that enable easy and efficient data management and data parallelization. This thesis will focus on using a DBMS to tackle these data management and data parallelization challenges, by making use of and extending the highly optimized data management techniques that DBMSs offer. These will do the data management for the user and will parallelize the users algorithms where possible.

The main research question of this thesis is: *How can a DBMS be exploited to better support analysis on DNA sequence alignment data?* In order to answer the research question, we first have to have an idea of the concrete use cases that exist in bioinformatics. What are bioinformaticians doing with genetic data and what methods do they use to perform these tasks? This information can then be used to design an implementation on a DBMS. An important part of this design focuses on dealing with the Extract-Transform-Load (ETL) hurdle. From where and how does the DBMS *extract* genetic data? How does the DBMS *transform* this data before it uses it? What portions of the transformed data are *loaded* into the DBMS and how is this performed exactly? Another question that needs to be addressed is how the DBMS implementation compares to traditional approaches, such as using third party software like Samtools [11] or Bamtools.[1] What are the pros and cons of using a DBMS? This thesis focuses on answering these questions. These answers contribute to many fruitful insights into applying DBMS techniques to bioinformatics.

Genetic data are often stored in SAM or BAM files [11], for instance in the Genome of the Netherlands (GoNL) project [3]. SAM and BAM files contain the exact same data, however BAM files are compressed using BGZF compression. For this thesis, we work exclusively with BAM files. Therefore, we omit further mentioning of the SAM format.

[1] Bamtools is a library similar to Samtools, implemented in C++. It is available on `http://sourceforge.net/projects/bamtools`.

Everything that applies to BAM files, applies trivially to SAM files as well. There are other file formats for storing genetic data, like the FASTQ and VCF formats, which can be derived from BAM files. BAM files can however not be derived from these other files, as they store less data.

1.2 Outline

Chapter 2 gives an overview of other work that is related in some way with the work presented here. Chapter 3 then gives some basic insight into how DNA sequencing is done. It focuses mainly on explaining the terminology that is used in the remainder of this thesis. To test any of the solutions we propose, a broad range of practically relevant use cases are a necessity. Therefore, Chapter 4 defines such use cases textually and formally. Chapter 5 presents the implementations that solve the earlier presented use cases, followed by experiments done with these implementations in Chapter 6. A possibility for improvement of our DBMS implementation is then presented in Chapter 7. Finally, Chapter 8 mentions the conclusions that can be drawn from the work presented here and gives an overview of interesting research areas.

Chapter 2

Related work

There are already several tools, with which one can analyze alignment data. The most popular tool is Samtools [11], which is a C library providing a command line interface and an API for operations on BAM files. Using the API, Samtools provides users with C data structures and functions that enable iterating over the alignments of a BAM file, without requiring the users to know exactly how these are stored on disk. Therefore, users don't have to worry about parsing and possible data compression. Furthermore, using these tools allows users to do some simple filtering steps when querying SAM and BAM files. However, relying on Samtools or Bamtools is sub-optimal for more complex needs. Users receive a collection of alignments and have to manage these alignments themselves. If they, for example, want to group data, they will have to do so manually in some programming or scripting language.

Another possible solution is described in [1], which extends a DBMS with the possibility to efficiently query raw data files. This technique is designed for CSV files, but would be well applicable to SAM files since SAM files are TAB delimited files. BAM files however are stored in a compressed manner and this breaks the assumption that every line in the file represents a tuple. Furthermore, individual entries are not accessible in BAM files, as all data is stored in a compressed manner.

Another possibility to work with BAM files is an extension to the popular system Hadoop, BAM-Hadoop [13]. It enables users to express their algorithms in a map-reduce fashion. However, translating complex use cases to a map-reduce algorithm is in many cases not a trivial task. This will therefore require a much practice for users that are not experienced with the map-reduce paradigm. Another downside to this approach is that BAM files have to be split up among several nodes. This disables users from using any other tools than Hadoop-BAM, since the original BAM files are lost.

Oracle has also been working on technology to support bioinformatics data in their DBMS [2]. We could however not find performance results from BAM data that was loaded into this implementation and hence this doesn't help us in answering our research question.

So far, we haven't found any contributions to the bioinformatics field that answers our research question.

Chapter 3

DNA sequence alignment data

Section 3.1 gives a brief overview of the DNA sequencing and alignment process. The data that result from this process and the way they are stored in BAM files is then discussed in Section 3.2.

3.1 DNA sequencing and alignment

Today it is still impossible to simply sequence whole chromosomes. Instead, many copies are made of every chromosome, which in turn are randomly cut into small pieces and sequenced individually. These small pieces are called *templates*. Since the partitioning of chromosomes happens randomly, heavy oversampling is required to ensure that the entire chromosome is covered with sufficient probability. Due to this oversampling, sequencing even the shortest human chromosome requires the generation of ~800,000 templates [14].

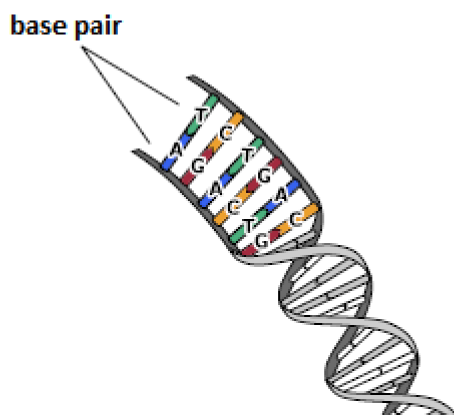


Figure 3.1: Part of a template, containing base pairs that are represented by letters. Every base pair can be represented by one character. Courtesy of T. Marschall, CWI.

Every template contains an unknown number of *base pairs*, illustrated in Figure 3.1. Every base pair can be represented by one character, since one character implies the other. Modern sequencing techniques enable bioinformaticians to read the first and the last part of every template. This results in two *reads*, or one *read pair*, for every template. The area in between the two reads is referred to as the *internal segment*. Due to the stochastic

nature of the partitioning process, the size of the internal segment is unknown. Figure 3.2 visualizes these terms.

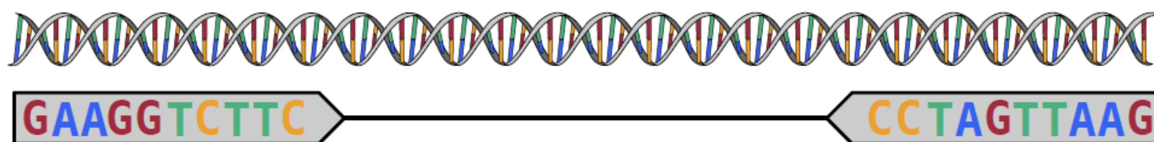


Figure 3.2: A template (top) and its digital representation (bottom). The digital representation consists of two reads (one read pair) and an internal segment of unknown size. Courtesy of T. Marschall, CWI.

The resulting read pairs are then run through one or more alignment algorithms. Such algorithms try to map every read pair to its original position in a reference string, taking into consideration the different uncertainties that the sequencing process is subject to. Thanks to the Human Genome Project, such a reference string exists for human genomes. The DNA of any human, which consists of ~3.2 billion base pairs, will only differ from the reference string on a few positions. Therefore, the alignment algorithms can use the reference string to map all alignments to a position where their *SEQ* match the reference string at that position as good as possible. It is highly possible for a single read to be aligned to multiple locations, because the reference string is long (~3.2 billion base pairs for human genomes), while the reads are short (~100 base pairs) and typically only four distinct characters are used to denote a read pair. Whenever a read pair gets mapped to multiple positions, one of these positions typically is considered to be the best position. The alignment on this position is then referred to as the *primary alignment* for this read pair, while the others are referred to as *secondary alignments*.

Figure 3.3: The reference string (top) and the aligned read pairs. The red circles indicate base pair mismatches. Courtesy of T. Marschall, CWI.

The resulting data from the alignment algorithm are visualized in Figure 3.3 and are usually stored in BAM files, since BAM files store all alignment information.

3.2 The BAM file format

This section gives an overview of the information stored in BAM files.

BAM files contain a header and a body. However, most use cases in the GoNL project do not use the information contained in the header. Therefore, it is sufficient to know only the most important things stored in such a header:

- The sorting order of the file, which can be one of the following:
  - Unsorted
  - Sortedness on the unique names of the templates of the alignments
  - Sortedness on the coordinates of the alignments
- Sequence dictionary, where all chromosomes that are stored in this file are denoted. In case this file is sorted on coordinate, the sequence dictionary gives the order in which the different chromosomes appear within the file.
- Grouping information about the reads that are present in the file.
- Information about the programs that were used to construct the BAM file.

The body of a BAM file consists of consecutive read alignment records. Every read alignment record stores information about a single alignment of one read. Alignment records are stored per read instead of per read pair and hence, alignment pairs are not stored explicitly. Instead, additional information is stored in the alignment records that can be used to reconstruct the read pairs.

In the remainder of this thesis, we will use *alignment* to refer to an alignment record, which contains data about an alignment.

3.2.1 Alignment fields

The fields contained in every alignment are described in this section.

QNAME

The *QNAME* field contains the unique name of the template where this alignment belongs to.

FLAG

The *FLAG* field contains an integer of which 11 bits are used as bitwise flags. These flags contain additional information about this alignment. The flags that are relevant for this thesis are described in Table 3.1.

RNAME

The *RNAME* field contains the name of the chromosome to which this alignment belongs. This field is set to '*' if the chromosome is unknown.

POS

The *POS* field contains the integer position where the *SEQ* string got aligned to within its chromosome. This field is set to 0 if the *SEQ* string is unmapped.

Bit	***Abbreviation***	***Description***
$0x4$	segm_unma	0 if this read is mapped, 1 if this read is unmapped. If a read is unmapped, the alignment algorithm that was used could not find an appropriate position in the reference string for this read and hence, when this flag is set no assumptions can be made on the *RNAME*, *POS*, *CIGAR* and *MAPQ* fields.
$0x10$	segm_reve	1 if the reverse complement of *SEQ* is stored, 0 otherwise.
$0x40$	firs_segm	1 if this is the first read from the template, 0 otherwise.
$0x80$	last_segm	1 if this is the last read from the template, 0 otherwise.
$0x100$	seco_alig	0 if this alignment is primary, 1 if this alignment is secondary.

Table 3.1: All the relevant bitwise flags that are present in the *FLAG* field of an alignment.

MAPQ

The *MAPQ* field contains the mapping quality of this alignment. This field is set to 255 if the mapping quality is not available.

CIGAR

The *CIGAR* field contains a string that consists of the characters {'M', 'I', 'D', 'N', 'S', 'H', 'P', 'X', '='}. These characters give information about the mapping to the reference string. For instance, the 'M' stands for 'alignment match', meaning that the corresponding character in the *SEQ* field matches the reference string. Other examples are the 'I' and 'D', which stand for an insertion into and a deletion from the reference string respectively.

Since a *CIGAR* string often contains many consecutive pieces of the same characters, *CIGAR* is encoded by run-length encoding. It therefore fulfills the regular expression /([0-9]+[MIDNSHPX=])+/.

An example *CIGAR* string would be $3M2D2I$. This means that the first three characters of the sequence string match with the position it is mapped to in the reference string. The next two characters exist in the reference string but are deleted (non existent) in the sequence string. The last two characters are inserted in the sequence string, meaning they don't exist in the reference string.

In this thesis we only use the *CIGAR* field to determine the length of a *SEQ* field if it is placed in the reference string.

RNEXT

The *RNEXT* field contains the value of the *RNAME* field of the mate of this alignment. The mate is defined as the other read in the same read pair as this alignment. This field

is set to '*' if the mate is unknown.

PNEXT

The *PNEXT* field contains the value of the *POS* field of the mate of this alignment. This field is set to 0 if the mate is unknown.

TLEN

The *TLEN* field contains an integer that denotes the observed template length. The exact definition of the meaning of this field is omitted, as the field is not used any further in this thesis.

SEQ

The *SEQ* field contains the actual sequence string. Every character in this string represents one base pair. This field is set to '*' if it is unavailable.

QUAL

The *QUAL* field contains the quality string. The quality string assigns to every character in *SEQ* a quality (encoded by a character) that describes the quality of this character. This field is set to '*' if it is unavailable. Otherwise, its length must equal the length of the accompanying *SEQ* field.

EXTRA

The *EXTRA* field contains extra information about an alignment. It contains substrings of the form TAG:$TYPE$:$VALUE$, where TAG is a 2-character string, $TYPE$ is a single character and $VALUE$ is a string of any length. There are some predefined TAGs, but the *EXTRA* string can be used to store any extra needed information.

3.2.2 Sortedness of BAM files

The header of a BAM file tells if and how a BAM file is sorted. It can indicate that it is unsorted, sorted on query name or sorted on coordinate. In case a BAM file is sorted on query name, it means that its alignments are ordered on their *QNAME* field. Whenever a BAM file is ordered on coordinate, its alignments are ordered on ($RNAME$, POS), meaning that they are first ordered on their *RNAME* fields and then on their *POS* fields.

BAI files are index files that allow random access on a BAM file. These BAI files are only defined over BAM files that are ordered by coordinate. They make it possible to do random access in a BAM file to extract exactly the alignments that lie in a given region.

The data that were explained in this section are enough to understand the remainder of this thesis. For more elaborate information on BAM files, the SAM/BAM specification can be consulted at `http://samtools.sourceforge.net/SAM1.pdf`.

Chapter 4

Practical use cases

To sketch a realistic view of the needs of bioinformaticians, a broad range of different use cases is a necessity. These use cases must cover a wide enough area of the most common operations on BAM files. Since no benchmarks exist that describe use cases that fulfill these properties, we collaborated with bioinformaticians of the GoNL project [3] to construct useful use cases. They gave us insights into the every day analysis they are doing on their BAM file repositories. We subdivided the resulting use cases into two categories: simple and complex use cases. They are described in Sections 4.3 and 4.4 respectively.

Giving formal definitions of these use cases requires specific notation. Therefore, Section 4.1 elaborates on the notation that we use. Also, the presented use cases have some commonalities in the functionalities they need. We prettified the definitions of the use cases by identifying these functionalities and putting them in function definitions, presented in Section 4.2.

In some of the use cases, variables are used in filtering steps. For example, $rname_2_10$ is a string variable that contains an rname string for use case 2.10. These variables will obtain concrete values in the experiments section.

4.1 Notation

Due to the formal nature of this chapter, many different notations are used. Therefore, this section introduces all notations used throughout the remainder of this chapter.

4.1.1 General

In many cases we need to extract individual fields from given alignments. We do this by using dot-notation, i.e. if we have an alignment a and we want to extract its $QNAME$ field, we write $a.QNAME$.

Furthermore, we will define several functions that can take arguments. We will use the notation that is used most often in writing pseudo code, i.e. $z = f(x, y)$ means we evaluate function f by passing it arguments x and y and we store the result of the function evaluation in a variable z.

4.1.2 Set notation

For the formal definitions of our use cases we will use set notation. We will always apply the notion of a strict set, meaning that a set always represents a collection of distinct elements. To indicate the cardinality of a set S, we use the notation $|S|$.

Furthermore, we use both square brackets and parentheses to denote ranges. Square brackets imply an inclusive and parentheses an exclusive range. For example, $[2, 9)$ is the equivalent of the set $\{2, 3, 4, 5, 6, 7, 8\}$.

In this chapter, we will use A to denote sets containing alignments and P to denote sets containing alignment pairs. A_f denotes a set containing all alignments from BAM file f.

4.1.3 Result

We will use $R_{i.j}$ to denote the result list of use case $i.j$. Such a result list uses notation similar to a set, although a result list has a given order and can in theory contain duplicates.

4.1.4 Sorting

Almost all of our use cases sort their final result set on some of the data fields, denoted by $R_{i.j} = sort_{(a,b)}(A)$, where the set A is sorted on sorting attributes a and b. Hence, the input to the sorting function is a set and the output is a result list. In some cases, a sorting attribute is ambiguous. For example, $sort_{(QNAME)}(P)$ where P contains alignment pairs, can not know if it has to sort on the $QNAME$ of the first or the second alignment. In that case, we use a subscript to distinguish between those two possibilities, as in for example $sort_{(QNAME_l)}(P)$ to indicate that we are sorting on the $QNAME$ field of the left alignments.

4.1.5 Grouping

Some of our use cases create data groups. We use the notation $G_{(i,..,k)}$ to denote a grouping operator that groups a set of alignments by the fields i until k. Grouping on any number of attributes is possible but in most cases a grouping will be done just on $QNAME$, yielding the notation $G_{(QNAME)}$. Given a set of alignments A,

$$G_{(i,..,k)}(A) = \{A_{i',..,k'} \mid A_{i',..,k'} \text{ contains all alignments from } A \text{ that have } i = i',..,k = k'\}$$

Such a group $A_{i',..,k'}$ has two properties, the first of which is the set of values of the grouping attributes: $i', .., k'$ The other property is the set of alignments that belong to the group, meaning they have exactly the values $i', ..k'$ for their grouping attributes. For example, if we have a set of alignments A, $G_{(QNAME)}(A)$ contains p groups of alignments $A_{q_1}, A_{q_2}, .., A_{q_p}$, where every group A_{q_j} contains all alignments a with $a.QNAME = q_j$ for $j \in [1..p]$. We extend the earlier defined *sort* function to also work on the result of a grouping by sorting all data inside the groups as if they were stored in a regular set. For instance, $sort_{(QNAME)}(G_{(QNAME)}(A))$ sorts all alignments in the grouping as if they were stored in A and results in a sorted version of A. This is useful when groups are filtered out of a grouping and the result has to be sorted afterwards.

4.2 Functionality over alignments

Some of the use cases require calculations over the attributes of one or more alignments. Table 4.1 defines functions for these calculations that will be used in the remainder of this chapter. Some of the implementations of these functions are given in pseudo code in Appendix A. When there exists such an implementation of a function, a reference to the algorithm is included in the last column of the table.

4.3 Simple use cases

The simple use cases are executed to explore parts of a BAM file f. The following sections present the simple use cases. They all give both a textual and a formal definition.

Use case 1.1

Select all primary alignments and sort the result set on $QNAME$.

Formal definition

$$A = \{a \mid a \in A_f \land \texttt{PrimaryAlignment}(a)\}$$
$$R_{1.1} = sort_{(QNAME)}(A)$$

Use case 1.2

Select all primary alignments and sort the result set on $(RNAME, POS)$.

Formal definition

$$A = \{a \mid a \in A_f \land \texttt{PrimaryAlignment}(a)\}$$
$$R_{1.2} = sort_{(RNAME,POS)}(A)$$

Use case 1.3

Select all alignments with $RNAME = rname_1_3$ and their POS field in the region $[pos_1_3_1, pos_1_3_2]$ and sort the result set on POS.

Formal definition

$$A = \{a \mid a \in A_f \land a.RNAME = rname_1_3 \land a.POS \in [pos_1_3_1, pos_1_3_2]\}$$
$$R_{1.3} = sort_{(POS)}(A)$$

Use case 1.4

Select all alignments with $QNAME = qname_1_4$ and sort the result set on $(RNAME, POS)$.

Formal definition

$$A = \{a \mid a \in A_f \land a.QNAME = qname_1_4\}$$
$$R_{1.4} = sort_{(RNAME,POS)}(A)$$

Function	*Explanation*	*Alg*
`ReverseSequence(`*String seq*`)`	Calculates the reverse complement of sequence string *seq*.	4
`ReverseQual(`*String qual*`)`	Computes the reverse of quality string *qual*.	5
`SequenceLength(`*Alignment a*`)`	Use the *CIGAR* string of *a* to calculate the actual length of the sequence string, which is defined as the length of the piece of the reference string it is mapped to.	6
`Distance(`*Alignments* a_1, a_2`)`	Calculate the distance between alignments a_1 and a_2, which is defined as the difference of the starting position of the rightmost alignment and the ending position of the leftmost alignment. The starting position of the rightmost alignment is stored explicitly in its *POS* field. For the computation of the end position of the leftmost alignment, the `SequenceLength` function can be used.	7
`InInternalSegment(`*Alignments* a_1, a_2, *position*`)`	Returns *True* if *position* lies in the internal segment of alignments a_1 and a_2.	8
`PrimaryAlignment(`*Alignment a*`)`	Returns *True* if *a* is flagged as primary alignment.	-
`FirstSegment(`*Alignment a*`)`	Returns *True* if *a* is flagged as first segment.	-
`LastSegment(`*Alignment a*`)`	Returns *True* if *a* is flagged as last segment.	-
`SegmentReversed(`*Alignment a*`)`	Returns *True* if *a* is flagged as being a reversed segment.	-
`SegmentUnmapped(`*Alignment a*`)`	Returns *True* if *a* is flagged as being an unmapped segment.	-

Table 4.1: Functions that define calculations over the attributes of one or more alignments. The algorithms this table refers to can be found in Appendix A.

Use case 1.5

Select all alignments with $MAPQ > mapq_1_5$ and sort the result set on the *MAPQ* values of its alignments.

Formal definition

$$A = \{a \mid a \in A_f \land a.MAPQ > mapq_1_5\}$$
$$R_{1.5} = sort_{(MAPQ)}(A)$$

4.4 Advanced use cases

In this section, we define the advanced use cases both textually and formally.

For the formal definition of the advanced use cases, we define all the functions that return $True$ if some flag is set on a set of alignments on sets of alignments. Instead of returning a single boolean, this function will return a set containing a boolean result for every alignment in the input set. We also define the additional function sum that sums the values of its input set. Suppose, for example, that we have a set of alignments A. Then, `PrimaryAlignment(`a`)` results in a set of booleans and sum(`PrimaryAlignment(`a`)`) results in the number of $True$ values in the set `PrimaryAlignment(`a`)`, assuming that the boolean values $True$ and $False$ are represented by a one and a zero respectively.

Furthermore, many use cases put the filtering constraint `FirstSegment(`a`)` $\neq$ `LastSegment(`a`)` on all their input alignments a. Since both functions are boolean functions (see Section 4.2), this implies that the qualified alignments must be flagged either as first segment or as last segment. Alignments that fulfill this filter will be referred to as being *flag-consistent*.

Use case 2.1

Select all flag-consistent alignments with $MAPQ < mapq_2_1$. Reconstruct the primary alignment pairs that exist in the resulting alignments and write the resulting left and right reads to two aligned FASTQ[1] files, sorted on $QNAME$.

Formal definition
Let A contain all flag-consistent primary alignments from A_f that have the value of their $MAPQ$ field below $mapq_2_1$.

$$A = \{a \mid a \in A_f \land \texttt{PrimaryAlignment}(a) \land \texttt{FirstSegment}(a) \neq \texttt{LastSegment}(a) \land a.MAPQ < mapq_2_1\}$$

Then, group A on $QNAME$ and only select the groups of size two, that have exactly one alignment flagged as first segment and the other as last segment. Store the result of the grouping in G.

$$G = \{A_q \mid A_q \in G_{(QNAME)}(A) \land |A_q| = 2 \land sum(\texttt{FirstSegment}(A_q)) = 1 \land sum(\texttt{LastSegment}(A_q)) = 1\}$$

Now, define A' as all alignments in A that have a $QNAME$ that exists in G.

$$A' = \{a \mid a \in A \land A_{a.QNAME} \in G\}$$

[1]For a definition of the FASTQ file format, see [4]

A' can then be split into A'_f and A'_l.

$$A'_f = \{a \,|\, a \in A' \wedge \texttt{FirstSegment}(a)\}$$
$$A'_l = \{a \,|\, a \in A' \wedge \texttt{LastSegment}(a)\}$$

The result lists are simply the sorted versions of A'_f and A'_l.

$$R_{2.1_1} = sort_{(QNAME)}(A'_f)$$
$$R_{2.1_2} = sort_{(QNAME)}(A'_l)$$

$R_{2.1_1}$ and $R_{2.1_2}$ can then both be written to their own FASTQ file. In case $\texttt{SegmentReversed}(a) = True$ for alignment a, $\texttt{ReverseSequence}(a.SEQ)$ and $\texttt{ReverseQual}(a.QUAL)$ are written instead of $a.SEQ$ and $a.QUAL$.

Use case 2.2

Calculate the distance between every flag-consistent primary alignment pair and create a histogram that, for every occurring distance, displays the number of alignment pairs with that distance. The output should be sorted on the number of alignment pairs.

Formal definition
Let A contain all flag-consistent primary alignments from A_f.

$$A = \{a \,|\, a \in A_f \wedge \texttt{PrimaryAlignment}(a) \wedge \texttt{FirstSegment}(a) \neq \texttt{LastSegment}(a)\}$$

Then, use A to construct G in the exact same way as we did in use case 2.1.

$$G = \{A_q \,|\, A_q \in G_{(QNAME)}(A) \wedge |A_q| = 2 \wedge sum(\texttt{FirstSegment}(A_q)) = 1$$
$$\wedge sum(\texttt{LastSegment}(A_q)) = 1\}$$

Now, again as in use case 2.1, use A and G to construct A'.

$$A' = \{a \,|\, a \in A \wedge A_{a.QNAME} \in G\}$$

Now let P be a set that contains triples. Every triple contains a pair of alignments from A', where every pair has the same $QNAME$ and $RNAME$, and the distance between these two alignments.

$$P = \{(a_1, a_2, d) \,|\, a_1, a_2 \in A' \wedge a_1.QNAME = a_2.QNAME \wedge a_1.RNAME = a_2.RNAME,$$
$$d = \texttt{Distance}(a_1,\ a_2)\}$$

Then build the histogram H by grouping the triples in P on their distance and inserting a record in the histogram for every group.

$$H = \{(d, count) | A_d \in G_{(d)}(A''), count = |A_d|\}$$

The result is then obtained by sorting the histogram.

$$R_{2.2} = sort_{(count)}(H)$$

Use case 2.3

Templates that have alignments neither flagged as their first segment nor as their last segment (or both) are considered to be inconsistent. Select all alignments that belong to such templates and sort them on *QNAME*.

Formal definition
Let G be the result of grouping the alignments in A_f on their *QNAME* and only store the groups that are inconsistent, i.e. have either no alignment flagged as first segment or no alignment flagged as last segment.

$$\begin{gathered} G = \{A_q \mid A_q \in G_{(QNAME)}(A_f) \\ \wedge(sum(\texttt{FirstSegment(}A_q\texttt{)}) = 0 \vee sum(\texttt{LastSegment(}A_q\texttt{)}) = 0)\} \end{gathered}$$

The result set is now obtained by sorting the alignments in G.

$$R_{2.3} = sort_{(QNAME)}(G)$$

Use case 2.4

All flag-consistent alignments from a template that agree on their first read and last read flag must either be unmapped, or there should exist exactly one primary alignment. If this doesn't hold, this template is considered to be inconsistent. Select all alignments that belong to such templates and sort them on *QNAME*.

Formal definition
Let A be defined as follows.

$$A = \{a \mid a \in A_f \wedge \texttt{FirstSegment(}a\texttt{)} \neq \texttt{LastSegment(}a\texttt{)}\}$$

Now, let G be the result of grouping the alignments in A on *QNAME* and on `FirstSegment`. Since the alignments in A are flag-consistent, grouping on `FirstSegment` implies grouping on `LastSegment`. G only contains groups that are considered to be inconsistent.

$$\begin{gathered} G = \{A_{q,f} \mid A_{q,f} \in G_{(QNAME\,\texttt{FirstSegment(}a\texttt{)})}(A) \wedge sum(\texttt{SegmentUnmapped(}A_{q,f}\texttt{)}) < |A_{q,f}| \\ \wedge sum(\texttt{PrimaryAlignment(}A_{q,f}\texttt{)}) \neq 1\} \end{gathered}$$

The result set is the sorted variant of G.

$$R_{2.4} = sort_{(QNAME)}(G)$$

Use case 2.5

Given two different BAM files, compute the number of *QNAMEs* they have in common and the number of *QNAMEs* that exist in one file but not in the other (both ways).

Formal definition
Let Q_1 and Q_2 contain the distinct *QNAME* fields from file A_{f1} and A_{f2} respectively.

$$\begin{gathered} Q_1 = \{q \mid \exists a \in A_{f1} \text{ with } a.QNAME = q\} \\ Q_2 = \{q \mid \exists a \in A_{f2} \text{ with } a.QNAME = q\} \end{gathered}$$

The result is the following triple:

$$R_{2.5} = \{(|Q_1 \cap Q_2|, |Q_1 - Q_2|, |Q_2 - Q_1|)\}$$

Use case 2.6

Given two different BAM files, select all alignments from these files that belong to a template that exists in both files and sort the result on *QNAME*.

Formal definition
Let Q_1 and Q_2 contain the distinct *QNAME* fields from file A_{f1} and A_{f2} respectively, formally defined in use case 2.5. Then let Q be the set intersection of Q_1 and Q_2.

$$Q = Q_1 \cap Q_2$$

Now define A to contain all alignments from A_{f1} and A_{f2} that have a *QNAME* that exists in Q.

$$A = \{a \,|\, a \in A_{f1} \cup A_{f2} \wedge \exists q \in Q \text{ with } q = a.QNAME\}$$

The result list is then obtained by sorting A.

$$R_{2.6} = sort_{(QNAME)}(A)$$

Use case 2.7

Given two different BAM files, select all alignments from a file that belong to a template that exists only in this file and sort the result on *QNAME*.

Formal definition
Let Q_1 and Q_2 contain the distinct *QNAME* fields from file A_{f1} and A_{f2} respectively, formally defined in use case 2.5. Then let Q be the set intersection of Q_1 and Q_2.

$$Q = Q_1 - Q_2$$

Now define A to contain all alignments from A_{f1} that have a *QNAME* that exists in Q.

$$A = \{a \,|\, a \in A_{f1} \wedge \exists q \in Q \text{ with } q = a.QNAME\}$$

The result list is again obtained by sorting A.

$$R_{2.7} = sort_{(QNAME)}(A)$$

Use case 2.8

Join flag-consistent primary alignments from two files if they have the same *QNAME* but are mapped to different positions and sort the result on *QNAME*.

Formal definition
Let A_l and A_r contain flag-consistent primary alignments from file $f1$ and $f2$ respectively.

$$A_l = \{a \,|\, a \in A_{f1} \wedge \texttt{PrimaryAlignment}(a) \wedge \texttt{FirstSegment}(a) \neq \texttt{LastSegment}(a)\}$$
$$A_r = \{a \,|\, a \in A_{f2} \wedge \texttt{PrimaryAlignment}(a) \wedge \texttt{FirstSegment}(a) \neq \texttt{LastSegment}(a)\}$$

Now let P contain alignment pairs, formed from A_l and A_r. The alignments in an alignment pair in A have the same $QNAME$, they agree on their `FirstSegment` and `LastSegment` value and they have either a different $RNAME$ or a different POS.

$$P = \{(a_1, a_2) \,|\, a_1 \in A_l \wedge a_2 \in A_r \wedge a_1.QNAME = a_2.QNAME$$
$$\wedge \texttt{FirstSegment}(a_1) = \texttt{FirstSegment}(a_2)$$
$$\wedge (a1.RNAME \neq a2.RNAME \vee a1.POS \neq a2.POS)\}$$

The result list is obtained by sorting P.

$$R_{2.8} = sort_{(QNAME)}(P)$$

Use case 2.9

Select all alignments with $RNAME = rname_2_9$ that have overlap with pos_2_9 and sort the result on POS.

Formal definition
Let A contain all alignments from A_f that have their $RNAME$ equal to $rname_2_9$ and that contain pos_2_9.

$$A = \{a \,|\, a \in A_f \wedge a.RNAME = rname_2_9$$
$$\wedge a.POS \leq pos_2_9 < a.POS + \texttt{SequenceLength}(a.CIGAR)\}$$

The sorted version of A now represents the result list.

$$R_{2.9} = sort_{(POS)}(A)$$

Use case 2.10

Reconstruct flag-consistent primary alignment pairs, output a pair if its internal segment overlaps pos_2_10 and sort the result on POS.

Formal definition
Let A contain all flag-consistent primary alignments from A_f.

$$A = \{a \,|\, a \in A_f \wedge \texttt{FirstSegment}(a) \neq \texttt{LastSegment}(a) \wedge \texttt{PrimaryAlignment}(a)\}$$

Then, let G be a grouping of A, similar to use case 2.1.

$$G = \{A_q \,|\, A_q \in G_{(QNAME)}(A) \wedge |A_q| = 2$$
$$\wedge sum(\texttt{FirstSegment}(A_q)) = sum(\texttt{LastSegment}(A_q)) = 1\}$$

Now, define A' as all alignments in A that have $rname_2_10$ as their $RNAME$ and have a $QNAME$ that exists in G.

$$A' = \{a \,|\, a \in A \wedge a.RNAME = rname_2_10 \wedge A_{a.QNAME} \in G\}$$

Now let P contain pairs of alignments from A', where every pair has the same $QNAME$ and every pair has pos_2_10 in its internal segment. Furthermore, every pair has to contain an alignment flagged as first segment and an alignment flagged as last segment.

$$P = \{(a_1, a_2) \mid a_1, a_2 \in A' \land a_1.QNAME = a_2.QNAME$$
$$\land \texttt{InInternalSegment}(a_1, a_2, pos_2_10) \land \texttt{FirstSegment}(a_1) \land \texttt{LastSegment}(a_2)\}$$

The result list is obtained by sorting the alignment pairs in P by the position of the left alignments.

$$R_{2.10} = sort_{(POS_l)}(A)$$

Use case 2.11

Pair flag-consistent secondary alignments, based on the information contained in the $RNEXT$ and $PNEXT$ fields and sort the result on $QNAME$.

Formal definition
Let A contain all flag-consistent secondary alignments from A_f that have a known value for their $RNAME$, POS, $RNEXT$ and $PNEXT$ fields.

$$A = \{a \mid a \in A_f \land \texttt{FirstSegment}(a) \neq \texttt{LastSegment}(a) \land \neg\texttt{PrimaryAlignment}(a)$$
$$\land a.RNAME \neq' *' \land a.POS > 0 \land a.RNEXT \neq' *' \land a.PNEXT > 0\}$$

Now let P contain pairs of alignments from A that have the same $QNAME$ and that have their $RNEXT$ and $PNEXT$ fields pointing to each other. Furthermore, as in use case 2.10, every pair contains an alignment flagged as first segment and an alignment flagged as last segment.

$$P = \{(a_1, a_2) \mid a_1, a_2 \in A \land a_1.QNAME = a_2.QNAME$$
$$\land \texttt{FirstSegment}(a_1) \land \texttt{LastSegment}(a_2)$$
$$\land ((a_1.RNEXT =' =' \land a_1.RNAME = a_2.RNAME) \lor a_1.RNEXT = a_2.RNAME)$$
$$\land a_1.PNEXT = a_2.POS$$
$$\land ((a_2.RNEXT =' =' \land a_1.RNAME = a_2.RNAME) \lor a_2.RNEXT = a_1.RNAME)$$
$$\land a_2.PNEXT = a_1.POS\}$$

Sorting P now yields the result list.

$$R_{2.11} = sort_{(QNAME_l)}(P)$$

Use case 2.12

Search for possible flag-consistent secondary alignment pairs based on the distance between two alignments and sort the result on the $RNAME$ of the left alignments.

Formal definition
Let A contain all flag-consistent secondary alignments from A_f that have a known value

for their *RNAME* and *POS* fields and have an unknown value for their *RNEXT* and *PNEXT* fields and sort the result on *RNAME*.

$$A = \{a \mid a \in A_f \wedge \texttt{FirstSegment}(a) \neq \texttt{LastSegment}(a) \wedge \neg\texttt{PrimaryAlignment}(a) \\ \wedge a.RNAME \neq' *' \wedge a.POS > 0 \wedge a.RNEXT =' *' \wedge a.PNEXT = 0\}$$

Now let P contain pairs of alignments from A, where every pair has the same *QNAME* and the same *RNAME*.. Only include a pair in P if the distance between the alignments in the pair is smaller than $distance_2_12$.

$$P = \{(a_1, a_2) \mid a_1, a_2 \in A \wedge a_1.QNAME = a_2.QNAME \wedge a_1.RNAME = a_2.RNAME \\ \wedge \texttt{Distance}(a_1, a_2) < distance_2_12\}$$

The result list is now obtained by sorting P.

$$R_{2.12} = sort_{(RNAME_l)}(P)$$

Chapter 5

Implementation

This section presents two implementations that solve the use cases presented in Chapter 4. For the first implementation, explained in Section 5.1, a DBMS was applied to work with data from BAM files. The second implementation, explained in Section 5.2, is meant for comparison with our DBMS implementation and is based on traditional techniques used for solving use cases on BAM files.

Throughout this chapter, use cases 1.1, 1.3 and 2.2 will be used to illustrate the implementations.

5.1 A DBMS implementation

We developed a DBMS implementation to solve the use cases presented in 4. Section 5.1.1 shows the storage schemas that we designed for this purpose. Section 5.1.2 then elaborates on how BAM files are loaded into these storage schemas. In order to solve our use cases, some user defined functions need to be implemented in the DBMS, based on Table 4.1. Section 5.1.3 shows these user defined functions. Finally, Section 5.1.4 explains how our use cases can be solved with the loaded storage schemas.

5.1.1 Database schemas

In order to apply a DBMS to data from BAM files, a storage schema has to be designed. Since BAM data is already stored in tabular form, we chose to design our schemas for relational database systems. The header data of the BAM files is stored in a way that is straightforward according to the BAM specification. We won't go into detail on the storage schema for header data, since the use cases that we need to solve do not use it. Those who are interested in this storage schema can find its SQL definition in Appendix B.1.

Since most analysis in the GoNL project is done on only one or in some cases two BAM files, we chose to maintain separate database tables for the alignments of every BAM file. This makes the number of tables in the schema proportional to the number of BAM files that are stored in the database. We designed two different storage schemas for storing the alignment data of a single BAM file: a straightforward design and a pairwise design. Both designs store the virtual offset of every alignment and use this as a primary key. The virtual offset is an unsigned 64-bit integer, where the first part points

to a compressed block inside the BAM file and the last part points to the start of an alignment in the decompressed version of the block. For an exact definition of the virtual offset, `http://samtools.sourceforge.net/SAM1.pdf` can be consulted.

Straightforward storage schema

The straightforward storage design simply stores all alignments of a BAM file in a single database table. The only effort that is taken is that the *EXTRA* field of every alignment is parsed and put into a separate table. Figure 5.1 shows a diagram of the tables that are created for a single BAM file with file id i. Underlined entries denote primary keys and entries between angle brackets denote foreign keys. Furthermore, entries are preceded by a symbol. A filled symbol means that the field has to have a non-empty value, an empty symbol means that the field may be empty. The symbol is a circle for normal fields and a diamond for primary key fields.

The SQL definition of this storage schema can be found in Appendix B.2.

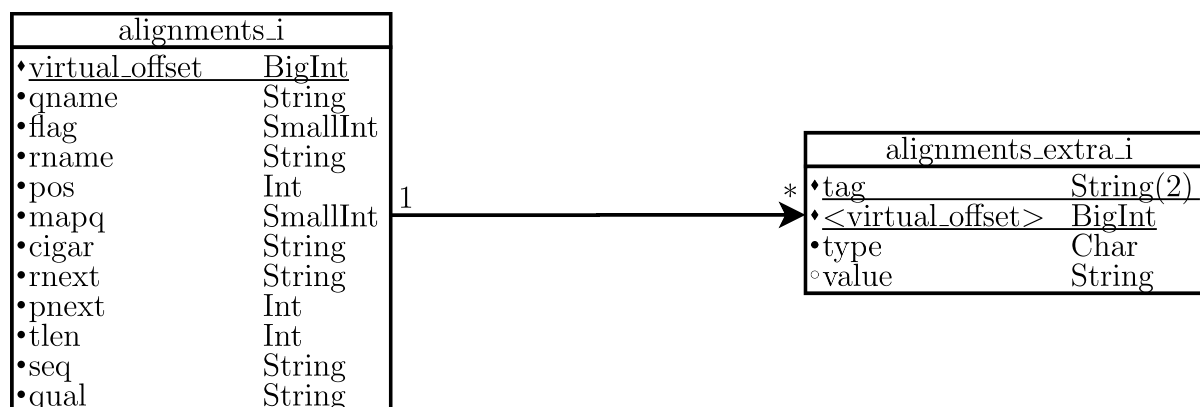


Figure 5.1: Straightforward storage schema.

Pairwise storage schema

Many of our use cases require some sort of alignment pair reconstruction. In most of these cases the primary alignment pairs need to be reconstructed, but in use case 2.11 secondary pairs need to be retrieved. If a bioinformatician mainly wants to operate on such alignment pairs, the previously presented storage schema is far from optimal, since reconstruction of the pairs needs to be done every time. Therefore, we designed a storage schema that explicitly stores primary and secondary alignment pairs. All the alignments that are not part of such an alignment pair are stored in a separate table. To simplify the required queries needed to solve our use cases, we included some views over the database tables that extract data from the paired tables and present them in an unpaired fashion. Figure 5.2 shows a diagram of the tables and views that are created for a single BAM file with file id i. In this storage schema, the table that stores the extra information from the alignments is exactly the same as before. Its foreign key relation can now however not be connected to a physical table since the alignments are scattered across multiple physical tables. Therefore, the foreign key relation is connected to the view that contains

the exact same data as the *alignments_i* table in Figure 5.1. The SQL definition of the pairwise storage schema can be found in Appendix B.3.

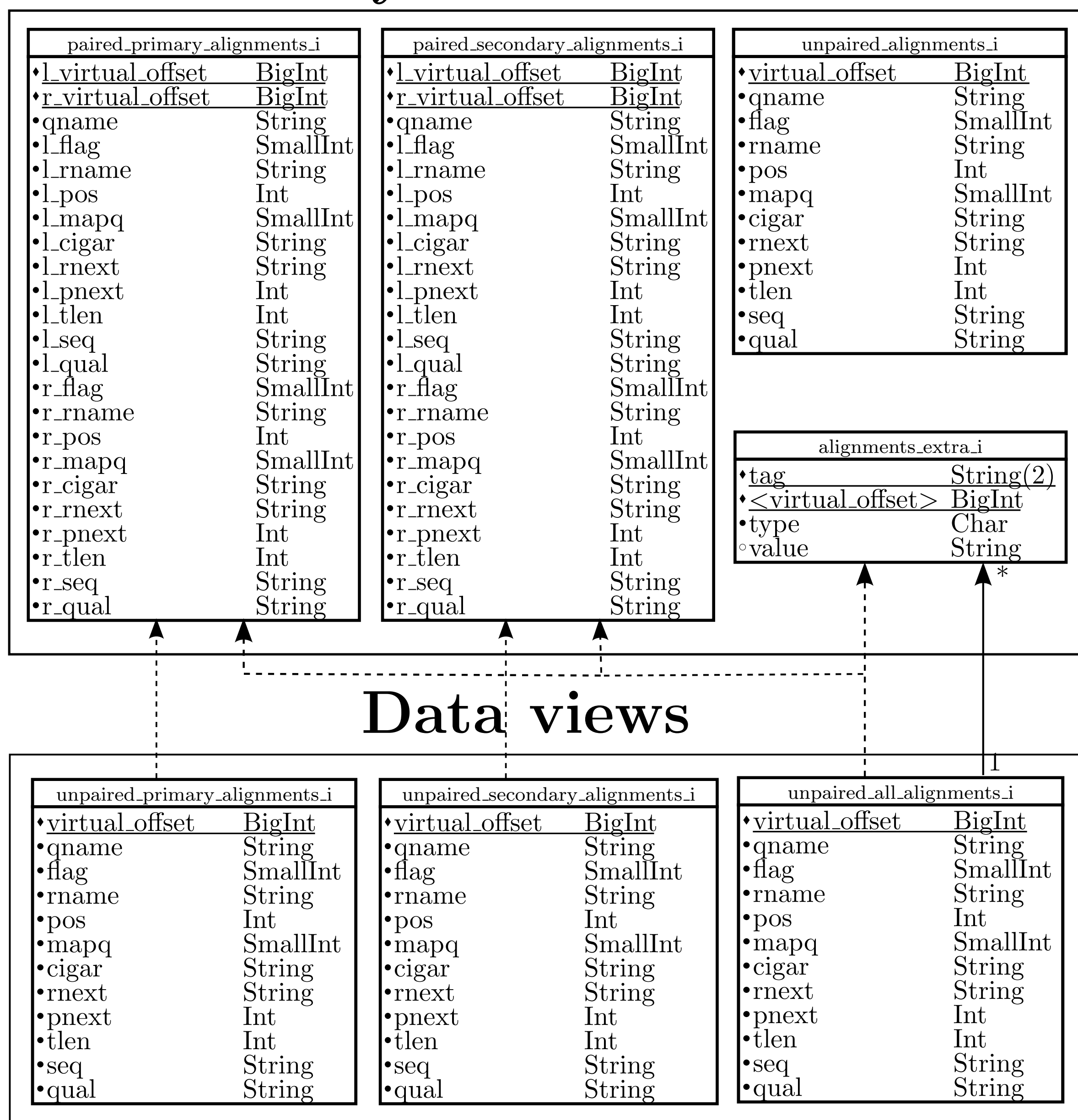


Figure 5.2: Pairwise storage schema. The solid arrow indicates a foreign key relationship, while the dotted arrows indicate the physical data sources of the data views.

5.1.2 Loading data into a storage schema

Since the BAM file format is highly specific, we developed a BAM loader that reads a BAM file and writes its contents to the appropriate database tables. This loader uses the Samtools API for the extraction of data from the BAM file and is able to load data in

both the straightforward and the pairwise storage schema. For both schemas, the loader automatically creates the appropriate database tables automatically.

The BAM loader is implemented as a user defined function an can therefore be called from a SQL interface. It is implemented by two user defined functions. The function definition for the first function is *bam_loader_file(string path, smallint schema, string mask)*. The *path* argument contains the absolute path to a BAM file and the *schema* argument contains a small integer, where a zero indicates that the BAM file has to be loaded into the straightforward storage schema and a one indicates that it has to be loaded into the pairwise storage schema. The *mask* argument should contain a 12-character string of '0's and '1's and they indicate which alignment fields have to be loaded, by the order in which they appear in Section 3.2.1. Note however that this functionality is only available for the straightforward storage schema. The function definition for the second function is *bam_loader_repos(string path, smallint schema, smallint nrthreads, string mask)*. This function is meant to load a whole repository of BAM files into the DBMS. The *path* argument this time has to point to a text file that contains a list of absolute paths to BAM files. It then loads all of these BAM files into the storage schema indicated by the *schema* argument, using as many threads as indicated by the *nrthreads* argument. Behind the scenes, every thread calls *bam_loader_file*.

Loading the straightforward storage schema

Every alignment that is encountered in a BAM file with file id i is inserted into the *alignments_i* table. The *EXTRA* field of the alignment is parsed and interpreted and is inserted into the *alignments_extra_i* table.

Loading the pairwise storage schema

We implemented two approaches for loading data from a BAM file with file id i into the *paired_primary_alignments_i*, *paired_secondary_alignments_i*, *unpaired_alignments_i* and *alignments_extra_i*. The second approach turned out to be significantly faster and therefore, if we talk about the pairwise storage schema, we always assume that the data is loaded using this approach. However, first approach is still presented here since it uses SQL code that gives a definition of primary and secondary alignment pairs that is used in the second approach.

The first approach starts with inserting every alignment in the *unpaired_alignments_i* table as if every alignment is unpaired. The techniques developed for loading data into the straightforward storage schema is reused for this purpose. When that is done, the primary and secondary alignment pairs can be computed with only SQL code. Appendix B.4 shows this SQL code. After executing this SQL code, the pairwise storage schema is filled correctly.

The second approach avoids expensive SQL operations on the data of the entire BAM file by inserting alignments directly into the appropriate tables. The SQL code in Appendix B.4 gives a definition of what can be considered a primary or secondary alignment pair. This approach has to respect this definition in order for the result to be the same. As can be seen from the SQL code in Appendix B.4, constructing primary and secondary alignment pairs can be done as soon as all alignments for a *QNAME* are known. Therefore, the second approach expects the input to be sorted on query name since this

greatly simplifies the internal data management that has to be performed in our loader. In case the BAM file isn't sorted accordingly yet, this can be performed relatively fast by Samtools. This approach then collects all alignments from some *QNAME* until another *QNAME* is encountered. At that point, the collected alignments are evaluated. The primary and secondary alignment pairs are detected and written to the appropriate tables and the remaining alignments are written to the *unpaired_alignments_i* table. Both approaches still simply parse, interpret and directly insert the *EXTRA* field of the alignments into the *alignments_extra_i* table upon traversing the alignments.

5.1.3 User defined functions

In order to solve the use cases, some user defined functions were implemented, based on the functions mentioned in table 4.1. These functions are mentioned in table 5.1.

Function	***Explanation***
bam_flag(smallint flag, string s)	Return a boolean indicating whether or not the flag indicated by *s* is set in the given *flag*
reverse_seq(string seq)	Reverse sequence string *seq* according to algorithm 4
reverse_qual(string qual)	Reverse quality string *qual* according to algorithm 5
seq_length(string cigar)	Calculate sequence length using CIGAR string *cigar*, according to algorithm 6

Table 5.1: User defined functions in DBMS. The algorithms this table refers to can be found in Appendix A.

5.1.4 Solving use cases

With the data from a BAM file loaded into one of our storage schemas, solving our use cases is simply a matter of writing SQL queries. Obviously, the SQL code for our different storage schemas will be different, as we are querying different physical storage architectures.

Listings 5.1, 5.2 and 5.3 give SQL code for solving use cases 1.1, 1.3 and 2.2 respectively. Every listing gives exactly two queries. The first query solves the corresponding use case for the straightforward storage schema and the last query solves it for the pairwise storage schema. The SQL code contains references to tables and views ending with the variable i. This variable should be substituted with the file ID of the file that is being queried, which can be found in the *bam.files* table.

It can be seen from these SQL queries that it depends on the purpose of the use case and the desired form of its output whether or not the pairwise storage schema will contribute to writing shorter SQL code. For instance, the SQL code for use case 1.1 is more complicated for the pairwise than for the straightforward storage schema.

It could be simplified by writing it exactly as the query that solves this use case for the straightforward storage schema, replacing the table *bam.alignments_i* with the view *bam.unpaired_all_alignments_i*. This is possible for every use case, since *bam.unpaired_all_alignments_i* is a view over all alignment data, thereby making it equivalent to the *bam.alignments_i* table in the straightforward storage schema. Doing this would however impose a less efficient execution in many cases, since the separation between primary and secondary alignments and the pairing information that is already present in the pairwise storage schema would then be neglected. The SQL code for use case 2.2 experiences an enormous decrease in complexity thanks to the pre-processing that was done when initializing the pairwise storage schema. For the straightforward schema, this SQL code has to filter out consistent alignments grouped on their *QNAME*, followed by creating valid primary alignment pairs from these alignments whereas these steps can be skipped by the SQL code for the pairwise schema.

Use cases 2.1, 2.2 and 2.5 are the only use cases that define how the output should be projected. Use case 2.1 writes its output to FASTQ files, use case 2.2 writes its output to a histogram and use case 2.5 writes exactly three fields. The output of the other use cases is a set of (paired) alignments. Since further analysis on these (paired) alignments might be desirable, we project all data except for the virtual offset and the extra alignment information for every query, as can be seen in Listings 5.1 and 5.2.

Listing 5.1: SQL queries that solve use case 1.1 for the straightforward and the pairwise storage schema respectively.

```
-- straightforward storage schema
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE bam_flag(flag, 'seco_alig') = False
ORDER BY qname;

-- pairwise storage schema
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_primary_alignments_i
UNION
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_alignments_i
WHERE bam_flag(flag, 'seco_alig') = False
ORDER BY qname;
```

Listing 5.2: SQL queries that solve use case 1.3 for the straightforward and the pairwise storage schema respectively.

```
-- straightforward storage schema
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE rname = rname_1_3
  AND pos >= pos_1_3_1
```

```
  AND pos <= pos_1_3_2
ORDER BY pos;

-- pairwise storage schema
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_all_alignments_i
WHERE rname = rname_1_3
  AND pos >= pos_1_3_1
  AND pos <= pos_1_3_2
ORDER BY pos;
```

Listing 5.3: SQL queries that solve use case 2.2 for the straightforward and the pairwise storage schema respectively.

```
-- straightforward storage schema
WITH alig AS (
    SELECT qname, flag, rname, pos, cigar
    FROM bam.alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = False
      AND qname IN (
        SELECT qname
        FROM bam.alignments_i
        WHERE bam_flag(flag, 'firs_segm') <>
              bam_flag(flag, 'last_segm')
          AND bam_flag(flag, 'seco_alig') = False
        GROUP BY qname
        HAVING COUNT(*) = 2
           AND SUM(bam_flag(flag, 'firs_segm')) = 1
           AND SUM(bam_flag(flag, 'last_segm')) = 1
     )
)
SELECT
    CASE WHEN l.pos < r.pos
         THEN r.pos - (l.pos + seq_length(l.cigar))
         ELSE l.pos - (r.pos + seq_length(r.cigar))
    END AS distance,
    COUNT(*) AS nr_alignments
FROM (
    SELECT qname, rname, pos, cigar
    FROM alig
    WHERE bam_flag(flag, 'firs_segm') = True
) AS l JOIN (
    SELECT qname, rname, pos, cigar
    FROM alig
    WHERE bam_flag(flag, 'last_segm') = True
) AS r
    ON l.qname = r.qname
```

```
   AND l.rname = r.rname
GROUP BY distance
ORDER BY nr_alignments DESC;

-- pairwise storage schema
SELECT CASE WHEN l_pos < r_pos
            THEN r_pos - (l_pos + seq_length(l_cigar))
            ELSE l_pos - (r_pos + seq_length(r_cigar))
       END AS distance,
       COUNT(*) AS nr_alignments
FROM bam.paired_primary_alignments_i
WHERE l_rname = r_rname
GROUP BY distance
ORDER BY nr_alignments DESC;
```

Appendices C and D give the SQL queries that solve all of our use cases for respectively the straightforward and the pairwise storage schemas, assuming that BAM files with file id i and j are loaded into these schemas.

5.2 The traditional approach

Bioinformaticians typically use some programming language, combined with an API for BAM files to solve use cases that involve BAM files. To discover the pros and cons of a DBMS implementation, we first solved the use cases mentioned in Chapter 4 using such a traditional approach. We chose to develop a program in C that uses the Samtools API to handle BAM file access. The reason we chose to do this in C instead of e.g. Python is that an implementation in C will in general be more efficient, since it is a low level programming language which gives us exact control over the data operations.

When implementing our C program, we decided that all tasks must be done as efficient as possible, i.e. operations that could be done within some time bound should be implemented accordingly. However, we did not implement parallel algorithms to solve our use cases since this would greatly complicate the required algorithms.

In order to implement all operations that are necessary as efficient as possible, many questions had to be answered. What intermediate data do we need to store, what updates are done on the data, do we need deletions, will we use dynamic memory allocations? Answering such questions finally lead to two important data structures that form the foundation of our program. These data structures are introduced in Section 5.2.1. Section 5.2.2 describes the most important functionality that is defined over the data structures to be able to implement our use cases. Section 5.2.3 will then go into more detail on how the sortedness of a BAM file influences the performance of our C program. Finally, Section 5.2.4 describes the implementation of some of our use cases.

5.2.1 Elementary data structures

To satisfy the efficiency norm mentioned earlier, we designed data structures that enable us to manage alignment data efficiently. We designed two such data structures: data tables and group tables.

Data table

A data table is used to store arbitrary tabular data. A data table is implemented as a doubly linked list, where every item in the list represents one tuple. A tuple contains one or more data fields, where every data field has a certain type (e.g. integer or string) and a value. A data table structure that stores alignment data is illustrated in Figure 5.3.

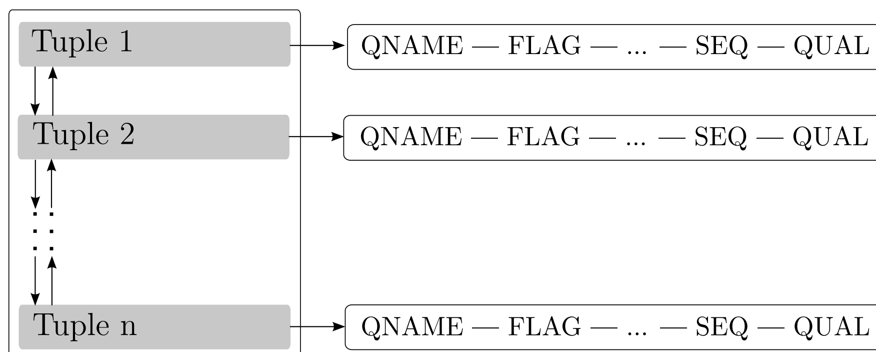


Figure 5.3: Visualization of data table structure.

Look up
Efficient look up of tuples is not possible in a data table structure. If one would look for a particular tuple, the whole linked list should be traversed. This is however not a problem for the efficiency norm. since individual tuple look ups are never performed in our algorithms.

Insertion
Insertion of a new tuple into a data table is in our case always done at the end of the linked list, since we do not maintain an ordering on the tuples. This operation therefore takes only $O(1)$.

Deletion
Deletion of a tuple can be done in $O(1)$ time as well, since the neighboring nodes in the linked list can be simply linked to each other.

Group table

In many of our use cases, data needs to be grouped according to some grouping attributes and aggregate functions need to be applied to the different groups. For this purpose, we introduced the group table as an elementary data structure. A group table is in fact just an array of buckets that implements dictionary encoding using a hash function. Attributes on which grouping is done are passed to this hash function, that hashes them to a bucket. In this bucket a doubly linked list is maintained that stores all group attributes that map to this bucket. Every node in this linked list remembers its grouping attributes and maintains a data table that stores all tuple-wise data for these grouping attributes. For further explanation in this chapter, let n be the number of distinct groups that are

stored in a group table and let m be the number of buckets. Furthermore, we assume that our hashing function evenly divides the groups over the buckets, yielding an expected linked list length of n/m for the linked list in any bucket. The group table structure is presented in Figure 5.4.

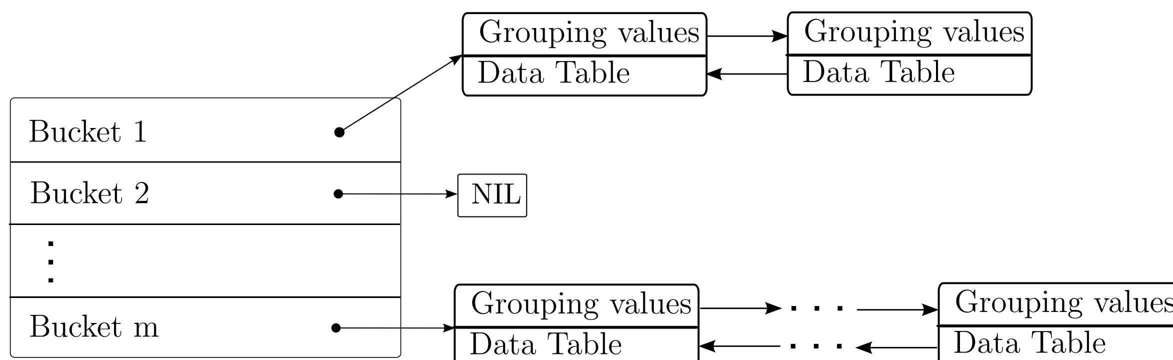


Figure 5.4: Visualization of group table structure with m buckets and a variable number of nodes in every linked list. The number of data tables stored in this group table is n. An empty linked list inside a bucket is represented by a NIL-pointer.

Look up
We often need to retrieve the data table that corresponds to some set of group attributes. The hashing mechanism can do this very efficiently, since only the data tables in one linked list have to be traversed. The efficiency then depends on the length of this linked list, since determining which linked list to traverse is as easy as computing the hash function, which takes $O(1)$ time. In the worst case, a look up has to search through an entire linked list, which has expected length n/m. Therefore, the expected worst case running time of a look up equals $O(1 + n/m)$.[1] A look up returns either a node from a linked list, or nothing if no node for the given group attributes was found.

Insertion
To insert a tuple, the group attributes are extracted from the tuple and used to perform a look up into the group table. If this look up results in a node from a linked list, the tuple can be simply inserted into the data table of this node. Otherwise, this tuple is inserted into the data table of a newly created node and this node is prepended to the linked list of the bucket where the group attributes of this tuple map to. The expected running time of an insertion into a group table is dominated by the look up that is performed, yielding an expected running time of $O(1 + n/m)$.

Deletion
Our algorithms require the possibility to delete an entire node from a linked list of a group table, including the clearance of the associated data table. Deleting a node from a linked list only requires some pointer restructuring and can therefore be done in $O(1)$. Clearing the associated data table is done by deleting every tuple in it. This can be done in $O(q)$ if the data table that has to be deleted contains q tuples. Hence, the actual

[1] The notion of expected (worst case) running times is taken from [5].

deletion takes $O(q)$. If the node to remove however first has to be found in the group table, the expected running time becomes $O(q + 1 + n/m)$.

5.2.2 Functionality defined over data structures

This section describes the functionality that is defined over the elementary data structures that is necessary to solve our use cases.

Transforming a data table

This functionality transforms a data table by applying a transformation on every tuple. For example, use case 2.2 transforms every tuple in a data table by calculating the distance and storing this in an extra field. A transformation traverses the entire data table and applies the transformation on every tuple. If we assume the mapping operator to run in $O(1)$, which is always the case in solving our use cases, this operation runs in $O(n)$, where n is the number of tuples in the data table.

Joining data

Joining is an operation that is performed in many of our use cases. This functionality takes two data tables as its input and produces one data table with tuples that contain data from both input data tables. A tuple from one data table and a tuple from the other data table form a tuple in the join result if the join condition is fulfilled. In our use cases, there is always at least one attribute that has to be equal in both tuples. Therefore, we chose to implement our join algorithm as an equi join; join two tuples from different input tables together whenever they have the same value for a predefined set of attributes. Since some of our join conditions are somewhat more complicated, we extended our join algorithm to a theta join by adding the possibility to pass an additional filter to the join algorithm. This filter determines whether or not a pair of tuples should be in the join result. The equi join is implemented in two ways: as a hash join and a merge join. The merge join is used when one or more of the join attributes are sorted in the input data tables.

Hash join
Let d_s be the data table with the least tuples of the two input data tables and d_b the one with the most tuples. The hash join assumes that both d_s and d_b are unsorted. It inserts all tuples from d_s into a group table g_s, where the join attributes are selected as the group attributes. This enables us to do efficient look ups when performing the equi join. We then iterate over all tuples in d_b. For every tuple t_b in d_b, we do a look up in g_s using the join attributes in t_b. If these join attributes aren't found in g_s, t_b doesn't join with any tuple from d_s and hence isn't inserted into the result. If however the join attributes are found, we obtain a data table d_{g_s} containing all tuples from d_s that join with t_b. Hence, t_b can be joined with every tuple in d_{g_s} and the resulting tuples are inserted into the result.

Now let $|d_s|$ and $|d_b|$ denote the number of tuples in d_s and d_b respectively. We will use n_s to denote the number of distinct join attributes in d_s and m to denote the number of buckets used in g_s. Furthermore, the resulting data table is denoted by r and $|r|$ denotes the number of tuples in r. Building g_s will then take an expected running time

of $O(|d_s|(1 + n_s/m))$. Then doing the actual join takes $|d_b|$ look ups in g_s, yielding an expected running time of $O(|d_b|(1 + n_s/m))$. Traversing all the tuples in the matching groups from g_s and inserting these tuples in the result can then be done in $O(|r|)$. The total expected running time of our hash join then becomes $O((|d_s|+|d_b|)(1+n_s/m)+|r|)$.

Merge join
The merge join assumes that both input tables are sorted on all join attributes. This makes the joining process much easier, as the two data tables can just be merged together without reordering any of them. This is done by traversing both input tables and constructing output tuples whenever we find a match between two tuples from the input tables. This operation runs in $O(|d_1| + |d_2|)$, where $|d_1|$ and $|d_2|$ denote the number of tuples in the first and in the second data table respectively.

Set operations

For several of our use cases, we have to perform set operations on two data tables. The set operations that we have implemented are set intersection and set minus. Both of them work similarly. They expect two data tables that are both sorted and contain only distinct values. The set operations can then be performed by simultaneously traversing both input data tables and add tuples to the result set accordingly. Therefore, this operation takes only $O(|d_1|+|d_2|)$, where $|d_1|$ and $|d_2|$ again denote the number of tuples in the first and in the second data table respectively.

Sorting a table

All of our use cases need to output sorted data. Therefore, we implemented a sorting algorithm that is able to sort a data table d on a given set of sorting attributes. For this purpose we use a merge sort approach that works on a linked list.[2] This approach does $O(log\,|d|)$ passes over the linked list, where $|d|$ is the number of tuples in our data table. Since every pass runs in $O(|d|)$, the total running time of the sorting functionality is $O(|d|\,log\,|d|)$.

Filtering

Many use cases need some kind of filtering operators on their data. For example, use cases 2.1, 2.2, 2.3, 2.4 and 2.10 require that as soon as a group table is filled with all the data, groups that do not fulfill a given predicate are deleted from the group table. This filtering results in a group table that contains a subset S of *QNAMEs*. The mentioned use cases then all have to delete all alignments from a data table that do not have a *QNAME* that exists in S. More formally, we have defined two different filtering operations.

The first operates on a group table g and takes a filter operator that operates on data tables. The entire group table is then traversed and for every data table that it contains, the filter operator is applied. In case the filter operator eliminates a data table, this data table will be removed from the group table. The filter operators used in our algorithms all run in $O(1)$. Let $|g|$ denote the number of tuples in g. The number of groups is never

[2] We adapted the approach described on `http://www.chiark.greenend.org.uk/~sgtatham/algorithms/listsort.html` to sort our linked list

bigger than $|g|$, i.e. $n \leq |g|$. Furthermore, no more than $|g|$ tuples can be deleted and hence, this operation runs in $O(|g|)$.

The other filtering operation takes a data table d and a group table g. It traverses the entire data table and does a look up in g for every tuple in d. In case no match is found in g, the tuple is removed from the data table. This filtering operation has an expected running time of $O(|d|(1 + n/m))$, where $|d|$ denotes the number of tuples in d.

5.2.3 Sortedness of the input

Our program considers every BAM file to have four different versions:

1. Unsorted version of the BAM file
2. BAM file is sorted by query name
3. BAM file is sorted by coordinate
4. BAM file is sorted by coordinate and is accompanied by a BAI index file

Our program takes these different versions into account by making all implementations of the use cases aware of the current kind of input they are dealing with. The performance of solving a use case depends on the version of the files it is dealing with. For example, if in use case 2.2 the input would be sorted by *QNAME*, the performance could be improved in the following ways:

- The grouping on *QNAME* now knows when the last tuple of a group has been inserted. Hence, the filtering that is done on every group can now be done while traversing the alignments of the BAM file. This saves on required storage space.
- A merge join can be done on *QNAME* when the second joining attribute, *RNAME*, is considered as an additional filter.

5.2.4 Use case implementations

This section illustrates how the presented data structure and the functionality that is defined over them can be used to solve our use cases. We present pseudo code for two of the use cases: use case 1.1 and use case 2.2. Both implementations assume unsorted input. Table 5.2 shows some functions that are used in the algorithms in terms of the aforementioned functionalities. We chose to show the implementation of use case 1.1 due to its simplicity. It selects all primary alignments from a BAM file and sorts the result on *QNAME*. The implementation is given in algorithm 1. Use case 2.2 is much more complex with regard to the required data management techniques. Its implementation can be found in algorithm 3. On lines 1-13, two functions are defined that are used in the main algorithm as $\oplus$ operators for the functions from Table 5.2 that require such an operator. On lines 15-18, empty data structures are created that will be filled by the algorithm. Lines 20-30 display the traversal of the BAM file and the storage of its alignments in d_{first}, d_{last} and g_{qnames}. The filter function is then applied to every data table in g_{qnames} on line 32. Lambda notation is used to indicate that the callback function receives one argument, namely a data table upon which the filtering should be

based. On lines 33 and 34, all alignments with a *QNAME* that doesn't exist in g_{qnames} are eliminated from both d_{first} and d_{last}. A join is then performed on line 35 on the *QNAME* and *RNAME* attributes. The result from the join is then transformed on line 36 using the predefined transform function to calculate and store the distance of every joined alignment pair. On line 37, the transformed join result is added into a group table, which represents the final grouping that we need. This grouping can be traversed and the result can be built from it, which is done on lines 39-41. $|d|$ is used here to denote the number of tuples in data table d. Finally, the result is sorted on line 43.

Function	***Explanation***
`DInsert`($d, fields$)	Insert all fields in $fields$ into data table d. If a whole alignment is given, all fields of the alignment will be stored.
`GInsert`(g, gv, t)	Insert (gv, t) in group table g, where gv contains the group values and t contains the values that will form a tuple in the data table.
`Sort`($d, attr$)	Sort data table d on sorting attributes $attr$.
`GFilter`($g, \oplus$)	Apply the first filter from the filtering explanation in Section 5.2.2 on group table g, using $\oplus$ as the transformation operator.
`DFilter`(d, g)	Apply the second filter from the filtering explanation in Section 5.2.2 on data table d, using g as the group table.
`Join`($d1, d2, attr$)	Join data tables $d1$ and $d2$, based on the joining attributes in $attr$.
`Transform`($d, \oplus$)	Transform data table d, using $\oplus$ as the transformation operator.
`DataTableToGroupTable`($d, attrs$)	Create and return a group table and insert all tuples from data table d in it, based on group attributes $attrs$.

Table 5.2: Functions used in algorithms 1 and 3

Algorithm 1: Implementation of use case 1.1.

Input : One BAM file f
Output : One CSV file

```
1 d ← empty data table
2 foreach alignment a in f do
3 |   if PrimaryAlignment(a) then
4 |   |   DInsert(d, a)
5 |   end
6 end
7 Sort(d, {QNAME})
8 return d
```

Algorithm 2: Implementation of use case 1.3.

Input : One BAM file f
Output : One CSV file

```
1 d ← empty data table
2 foreach alignment a in f do
3 |   if a.RNAME = rname_1_3 ∧ pos_1_3_1 ≤ a.POS ≤ pos_1_3_2 then
4 |   |   DInsert(d, a)
5 |   end
6 end
7 Sort(d, {POS})
8 return d
```

Algorithm 3: Implementation of use case 2.2.

```
   Input  : One BAM file f
   Output : One CSV file

 1 Filter(Data table d) begin
 2 |  if |d| = 2 ∧ sum(firs_segm(d)) = sum(last_segm(d)) = 1 then
 3 |  |  return True
 4 |  end
 5 |  else
 6 |  |  return False
 7 |  end
 8 end
 9
10 GetDistance(Tuple t) begin
11 |  d ← distance between alignments in t
12 |  add d as a data field to t
13 end
14
15 d_first ← empty data table
16 d_last ← empty data table
17 g_qnames ← empty group table
18 d_result ← empty data table
19
20 foreach alignment a in f do
21 |  if ¬seco_alig(a) ∧ firs_segm(a) ≠ last_segm(a) then
22 |  |  if firs_segm(a) then
23 |  |  |  DInsert(d_first, a)
24 |  |  end
25 |  |  else
26 |  |  |  DInsert(d_last, a)
27 |  |  end
28 |  |  GInsert(g_qnames, {a.QNAME}, {a.FLAG})
29 |  end
30 end
31
32 GFilter(g_qnames, λd →Filter(d))
33 DFilter(d_first, g_qnames)
34 DFilter(d_last, g_qnames)
35 d_joined ← Join(d_first, d_last, {QNAME, RNAME})
36 Transform(d_joined, λt →GetDistance(t))
37 g_distance ← DataTableToGroupTable(d_joined, {distance})
38
39 foreach tuple (distance, datatabled) in g_distance do
40 |  DInsert(d_result, {distance, |d|})
41 end
42
43 Sort(d_result, |d|)
```

Chapter 6

Experiments

To discover the pros and cons of the approaches presented in Chapter 5, we ran experiments that enable us to compare these approaches. Section 6.1 presents the details of these experiments and Section 6.2 shows and explains the results obtained through these experiments.

6.1 Setup

This section elaborates upon the different things that we did in order to set up our experiments. The hardware and software that we used to run our experiments are described in Section 6.1.1. To be able to do experiments, sample BAM files were needed. Section 6.1.2 presents the statistics of the BAM file repository that we used. The implementations from Chapter 5 require different kinds of initialization, elaborated upon in Section 6.1.3. After initializing, the use cases can be solved, which is explained in Section 6.1.4. Finally, Section 6.1.5 gives details about the output that is generated when the use cases are solved.

6.1.1 Hardware and software

All experiments were done on a machine with 32 Intel® Xeon® E5-2650 0 @ 2.00GHz processors and 256 GB DIMM DDR3 1600 MHz (0.6 ns) RAM.

We decided to do the experiments using MonetDB.[1] Thanks to its possibility to develop User Defined Functions (UDFs), we could easily extend it with the appropriate code for loading the BAM files into the database.

6.1.2 Our file repository

The Life Sciences group of the CWI provided us with a BAM file repository which we could use to run our experiments with. To obtain this repository, a read simulator was used to generate many reads. Different read mappers were then used to obtain different BAM files. Furthermore, some BAM files only contain part of the original data, which

[1] MonetDB is a column-store DBMS, available at http://www.monetdb.org/

results in a BAM repository with different file sizes. For more information on how the repository was initially obtained, see [12].

All the BAM files that we work with are stored in an unsorted fashion, since this is how they are essentially generated by the alignment process. We have run our experiments on a total of 17 files, which can be subdivided into two groups: a group containing relatively small BAM files and a group containing big BAM files. The first group contains 15 BAM files, the last group contains 2 BAM files. Some statistics of both groups are shown in Table 6.1.

ID	***Size***	***SAM size***	***Compression***	***# Alignments***	***Primary***	***Unmapped***	***Reversed***
1	22M	61M	1:2.7	200,000	100.00%	0.04%	49.99%
2	22M	59M	1:2.7	197,870	100.00%	0.37%	49.82%
3	24M	67M	1:2.8	200,000	100.00%	0.16%	49.92%
4	24M	69M	1:2.9	200,000	100.00%	0.16%	49.94%
5	32M	122M	1:3.8	605,908	66.02%	0.50%	49.71%
6	115M	301M	1:2.6	831,930	100.00%	0.42%	49.79%
7	126M	464M	1:3.7	2,288,284	73.39%	0.50%	49.75%
8	1145M	3,183M	1:2.8	10,354,336	100.00%	0.04%	49.98%
9	1242M	3,468M	1:2.8	10,354,336	100.00%	0.17%	49.91%
10	1255M	3,515M	1:2.8	10,354,336	100.00%	0.05%	49.97%
11	1257M	3,560M	1:2.8	10,354,336	100.00%	0.17%	49.93%
12	1352M	3,714M	1:2.7	10,354,336	100.00%	0.14%	49.93%
13	1442M	3,793M	1:2.6	10,242,048	100.00%	0.38%	49.81%
14	1444M	3,794M	1:2.6	10,242,048	100.00%	0.38%	49.81%
15	1660M	6,353M	1:3.8	31,510,492	65.72%	0.50%	49.75%
16	92G	253G	1:2.8	848,596,934	100.00%	0.91%	49.55%
17	100G	276G	1:2.8	848,596,934	100.00%	0.99%	49.51%

Table 6.1: Some statistics of the individual BAM files in our file repository. A horizontal line is added between the small files and the big files.

The first column shows the file identifier, which will be used to refer to individual BAM files throughout the remainder of this chapter. The second and third column respectively display the file size of the BAM file and the equivalent SAM file (an uncompressed version of the BAM file). The fourth column than shows the compression ratio, which can be calculated from the file sizes of the BAM file and the equivalent SAM file. The fifth column shows the number of alignments. Furthermore, the sixth until the last column show the percentage of alignments that is flagged as primary, unmapped and reversed respectively. The percentage of secondary alignments is easy to determine, since an alignment is always either primary or secondary. Hence, the percentage of secondary alignments can be expressed as 100%−percentage of primary alignments.

One thing that can be seen from Table 6.1 is that there is a relation between the compression ratio and the percentage of primary alignments. All three files that contain secondary alignments have a much higher compression ratio. This is easily explained by noting that secondary alignments contain the exact same *SEQ* and *QUAL* string as their primary versions. Since the primary alignments and their corresponding secondary alignments are often stored shortly after each other, the BGZF compression can take advantage of this redundancy.

Another thing that can be seen from Table 6.1 is that many BAM files have the same number of alignments. This can be explained by the fact that all the BAM files origin from the same set of read pairs.

6.1.3 Initialization

Before the use cases can be solved, the data in the BAM files need to be initialized.

In the case of the DBMS implementation the initialization is straightforward: all data from the BAM file repository needs to be loaded into the storage schemas of the database.

The traditional implementation works directly on BAM files. Its running time however depends on the sortedness of the input BAM files. Since the BAM files in our repository are all stored in an unsorted fashion, we consider different initialization times for this implementation. For some use cases, sorting the input will pay off, for others it won't. The following initialization types will be considered for our experiments:

- No sorting (will always have an initialization time of 0)
- Sorting on *QNAME*
- Sorting on (*RNAME*, *POS*)
- Sorting on (*RNAME*, *POS*) plus the creation of a BAI file

6.1.4 Solving use cases

When the systems are initialized, our use cases can be solved.

Solving the use cases using the DBMS implementation is simply done by executing the SQL queries for both storage schemas, displayed in Appendices C and D. We picked the use case variables such that the queries would generate a non-empty output for at least some BAM files. For instance, if for use case 1.3 we pick an *RNAME* that doesn't exist in any of the BAM files, the output will always be empty, which is something that we would like to prevent. The use case variables that we used for our BAM repository are shown in Table 6.2.

Var	***Value***
rname_1_3	chr10
pos_1_3_1	1,000,000
pos_1_3_2	2,000,000
qname_1_4	sim_Venter_chr1_1_1
mapq_1_5	200
mapq_2_1	100
rname_2_9	chr1
pos_2_9	73,796,782
rname_2_10	chr1
pos_2_10	80,000,000
distance_2_12	10,000,000

Table 6.2: Use case variables used for the experiments on our BAM file repository.

For the traditional approach, we solve every use case for all different possible orderings of the BAM files. All use cases are applied on unsorted BAM files to get a base line execution time. Only the orderings which possibly contribute to a faster running time will then also be run. For instance, solving use case 1.1 on a BAM file that is ordered by coordinate doesn't have an advantage over solving it on an unsorted BAM file. Therefore, we only solved use case 1.1 on an unsorted BAM file and on a BAM file that is sorted by query name. Table 6.3 displays for every use case the different orderings that were used as input to the use case.

Use case	***U***	***Q***	***C***	***B***
1.1	✓	✓		
1.2	✓		✓	
1.3	✓		✓	✓
1.4	✓	✓	✓	
1.5	✓			
2.1	✓	✓		
2.2	✓	✓		
2.3	✓	✓		
2.4	✓	✓		
2.5	✓	✓		
2.6	✓	✓		
2.7	✓	✓		
2.8	✓	✓		
2.9	✓		✓	✓
2.10	✓	✓	✓	✓
2.11	✓	✓		
2.12	✓	✓	✓	

Table 6.3: The orderings that were used as input to our use cases. Meaning of the column names: 'U' → unsorted, 'Q' → query name, 'C' → coordinate and 'B' → coordinate + BAI file.

To obtain statistically sound results, we have ran the experiments on the small files ten times. The results in this chapter will present the averages of these ten runs, minus the two outliers. For the big files however, running times of the experiments are quite long. Therefore, we only ran our experiments on big files once and these results are thus less accurate. Furthermore, we only solved use cases 1.1 - 1.5 for the big files.

6.1.5 Output

The SQL queries in Appendices C and D project almost all of the information in the alignments. This could be desired, when for instance further analyses must be done on all of these data. However, the user might just be interested in one column of the result set or in the number of tuples that it contains. Therefore, we also experimented with a version of the queries that outputs only columns on which sorting is performed, plus the virtual offset for every tuple. These virtual offsets could then be used to retrieve the remaining information of the tuples, if needed. This is implemented in both the DBMS and the traditional implementation. For the DBMS implementation, this means that it will have to do the same filtering and grouping operations, but it doesn't have to collect

all the columns from the alignment tables in the end. Furthermore, the amount of data that it has to send over a TCP socket reduces. For the traditional implementation, this means that the data structures do not have to contain all the alignment data anymore and thus have to deal with less data. We will refer to the full projection approach as the full-output approach and to the other approach as the minimal-output approach. Due to the huge size of the output files for some queries, we chose to apply the full-output approach only on the small files. The use cases on the big files are only solved using the minimal-output approach.

To verify the results of the different implementations, the use case results are written to CSV files. The CSV files of all different implementations are then checked for similarity in two ways. It is checked whether or not both files contain exactly the same rows. Furthermore, it is checked whether or not both files are ordered on the same columns by extracting the sorting columns from the CSV files and checking them for similarity. In case both of these checks show no differences, the two outputted CSV files are considered to be equal. This comparison method however doesn't work for the minimal-output approach, since virtual offsets are included in the output then and the virtual offset of an alignment may change whenever the ordering of a BAM file changes. Therefore, when comparing the output files from the minimal-output approach, the virtual offset column is removed from the result as a first step.

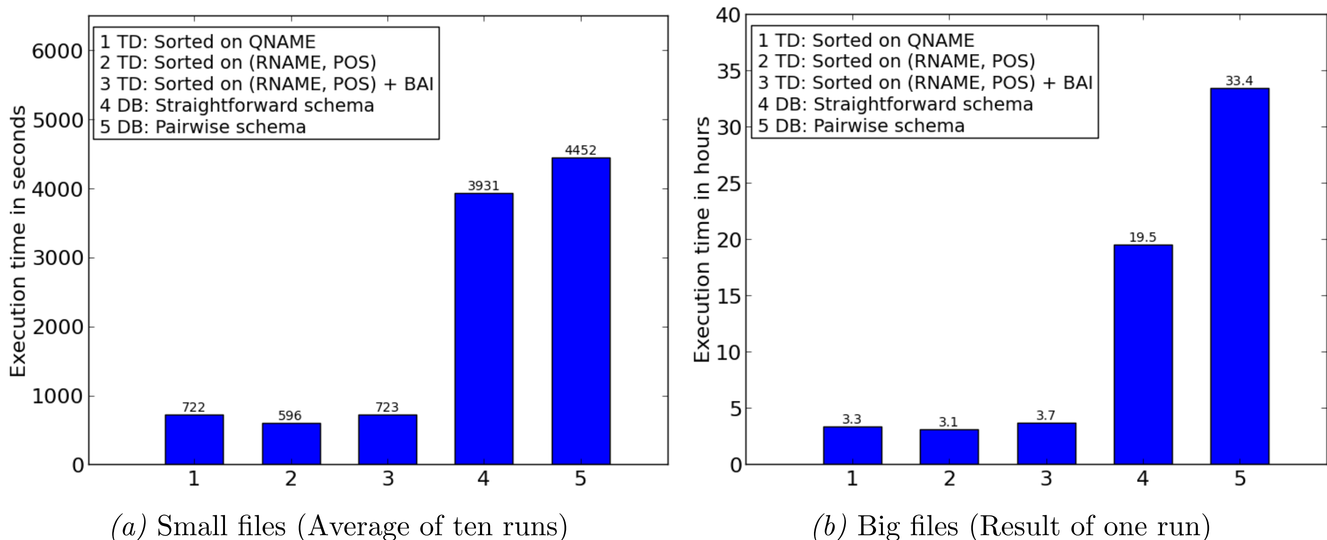
6.2 Results

Our experiments gave rise to many interesting results for the implementation methods we have presented thus far. The DBMS implementation (Section 5.1) presents two of these methods: the methods with the straightforward and the pairwise storage schema respectively. The traditional implementation (Section 5.2) presents the remaining four of these methods, where every possible sorting order is interpreted as a method. Section 6.2.1 presents the results for initializing these different methods. Sections 6.2.2 and 6.2.3 both present the running times obtained when solving the use cases, which focus on the full-output approach and the minimal-output approach respectively. Section 6.2.4 presents some details on correlations that we have identified in our data. Finally, Section 6.2.5 lists the pros and cons of our implementations that can be concluded from the results presented in this section.

6.2.1 Initialization

The timing statistics for initializing our implementations for both the small files and the big files from our repository are plotted in Figure 6.1. Note that the presented results in Figure 6.1a are the averages from ten runs, while Figure 6.1b presents data from a single run.

The most important conclusion that can be drawn from these figures is that inserting all data from the BAM files in any of our storage schemas takes significantly longer than sorting the BAM file in any way. This is caused by several factors, one of which is that the DBMS must store all data in a decompressed manner, causing a higher I/O load when writing the result to disk than when another BAM file has to be written. Another



(a) Small files (Average of ten runs)

(b) Big files (Result of one run)

Figure 6.1: Required running time by different methods to initialize all BAM files in our repository.

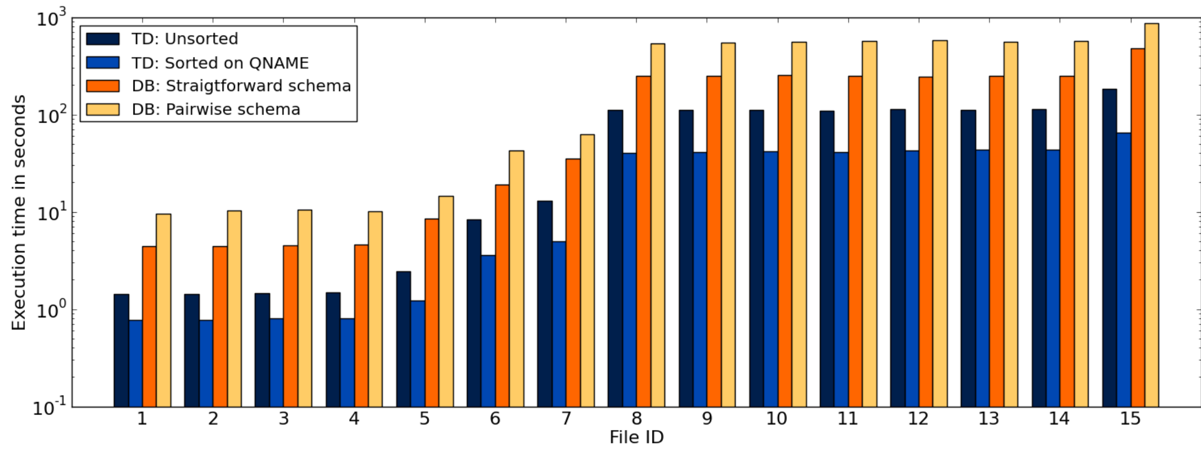
factor is that the DBMS has to parse every field, whereas the sorting algorithm only has to interpret the fields on which it is sorting and some minor things needed to traverse and rebuild the BAM file.

Another thing that can be seen is that loading the data into the pairwise storage schema takes significantly more time than loading it into the straightforward schema, especially for the big files. This difference in loading time makes sense, since loading these different storage schemas works in approximately the same way, although the loader has to perform more data management tasks when loading the pairwise storage schema. These data management tasks create an overhead on loading the alignments from every single template, imposing a difference between loading the two storage schemas that is proportional to the number of templates that are stored within a BAM file. This gives rise to the much bigger difference between loading the storage schemas for big files.

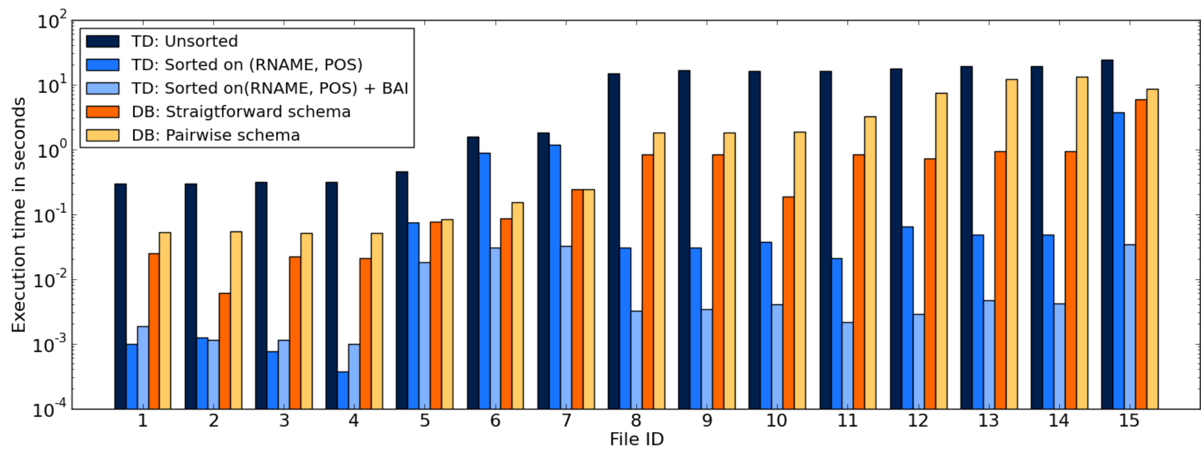
6.2.2 Solving use cases using the full-output approach

Figure 6.2 presents bar charts that illustrate the average running times for solving use cases 1.1, 1.3 and 2.2, using the full-output approach. Every chart shows the results for all its relevant orderings, conforming Table 6.3. Furthermore, every chart shows the results for both storage schemas of the DBMS implementation.

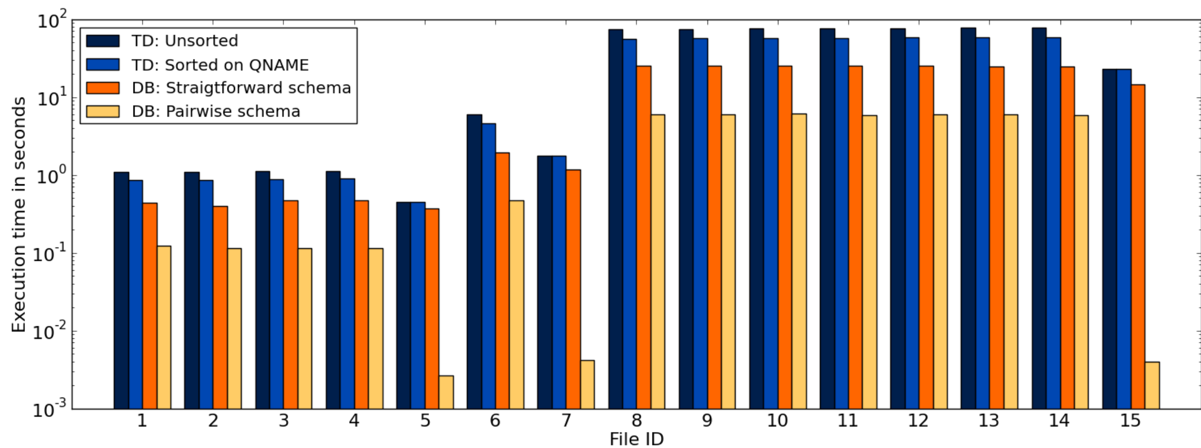
The results for use case 1.1, shown in Figure 6.2a, show clearly that the DBMS implementation performs significantly worse than the traditional implementation. This is due to the difference in how these systems write the result sets, as the result sets for use case 1.1 are huge. The traditional implementation builds the result set and then writes everything from main memory directly to a file, whereas the DBMS implementation has to do an additional serialization and deserialization step, since it has to send this entire result set over a TCP socket to the client process. Figure 6.3 visualizes this overhead by showing the time between having calculated the result set and having the result set written to disk for every file when executing use case 1.1. This time is significantly higher for the DBMS implementation. Another thing that can be seen from Figure 6.2a is that sortedness on



(a) Use case 1.1



(b) Use case 1.3



(c) Use case 2.2

Figure 6.2: Running times of solving use cases on small files using our different methods. These are the results for the full-output approach and are the averages of ten runs.

QNAME significantly improves the performance of the traditional implementation for every input file. This makes sense, since every file gives rise to a big output set and when the input is already sorted by *QNAME*, the traditional implementation can skip

the final sorting step. Furthermore, this figure clearly shows the overhead that is caused by the unpairing step that needs to be done for the pairwise storage schema of the DBMS implementation. It also shows that this overhead is somewhat smaller for files 5, 7 and 15, as these files contain secondary alignments. Thanks to the separation between primary and secondary alignments in the pairwise storage schema, the DBMS implementation can discard the secondary alignment data for these files immediately.

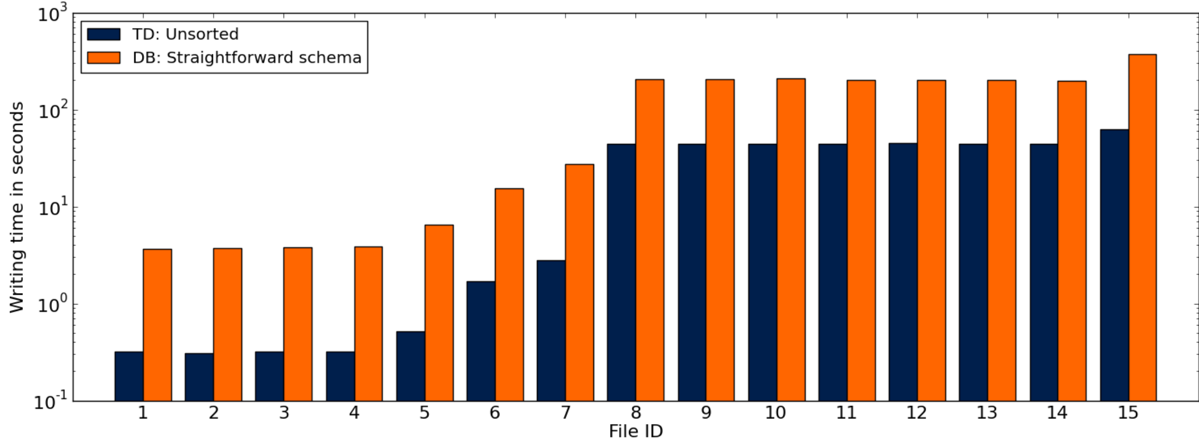


Figure 6.3: Time required to write the result to disk for use case 1.1 using the full-output approach

Figure 6.2b shows the results for use case 1.3. It shows big differences between the different methods. Most importantly, for all files both storage schemas from the DBMS implementation perform better than the unsorted traditional implementation. However, when there is an ordering on $(RNAME, POS)$, the traditional approach outperforms both storage schemas of the DBMS implementation. This is due to the fact that, whenever a BAM file is sorted by $(RNAME, POS)$, the traversal of a BAM file can be stopped as soon as the right $RNAME$ is found. The differences in the performance increase for this ordering reflect how early in the BAM file the right $RNAME$ occurs. In case the right $RNAME$ occurs really early in the BAM file, this method even outperforms the equivalent method where a BAI index is used to extract the right alignments. Furthermore, it can be easily seen from the figure that using the pairwise storage schema for this use case is suboptimal, since it is always outperformed by the straightforward storage schema.

Finally, Figure 6.2c shows the results for use case 2.2. The DBMS implementation outperforms the traditional implementation for every file. This is partly due to the complex data management techniques that need to be done to solve this use case, for which a DBMS is highly optimized. Another thing that plays a role is the relatively small output size for this use case. As mentioned in the explanation of Figure 6.2a, writing results using the DBMS implementation creates an overhead due to the need to send all output over a TCP socket first. Furthermore, the pairwise storage schema imposes a significant performance gain for this use case, since the pairing does not need to be done anymore during query execution time. Especially for files 5, 7 and 15, the pairwise storage schema greatly outperforms the straightforward storage schema. This can again be explained by the fact that exactly these files contain secondary alignments. For the pairwise storage schema, the DBMS can immediately discard the secondary alignment data, while for the straightforward storage schema it has to consider every alignment.

We ran these experiments on all of the use cases, which resulted in a bar chart for every use case, which are located in Appendix E.1. Some general conclusions that can be drawn from these charts are mentioned here.

BAM files containing secondary alignments

The patterns that can be seen in the results of files 5, 7 and 15 often differ from the results of the other files, due to the fact that these files contain secondary alignments. The DBMS implementation often takes advantage of the pairwise storage schema for these files whenever it receives a query that needs to operate on paired data. However, if a query filters out all primary or secondary alignments, the DBMS implementation can also take advantage of the pairwise schema, since this schema already separated the primary from the secondary alignments on initialization.

BAM files that contain secondary alignments can however also slow the DBMS implementation down. Figure 6.4 shows such an example, where the straightforward storage schema of the DBMS implementation performs significantly worse when secondary alignments are involved. The reason for this performance loss is again the overhead caused by writing the result set. Files 5, 7 and 15 are the only files that contain inconsistencies, yielding a result set for only these use cases.

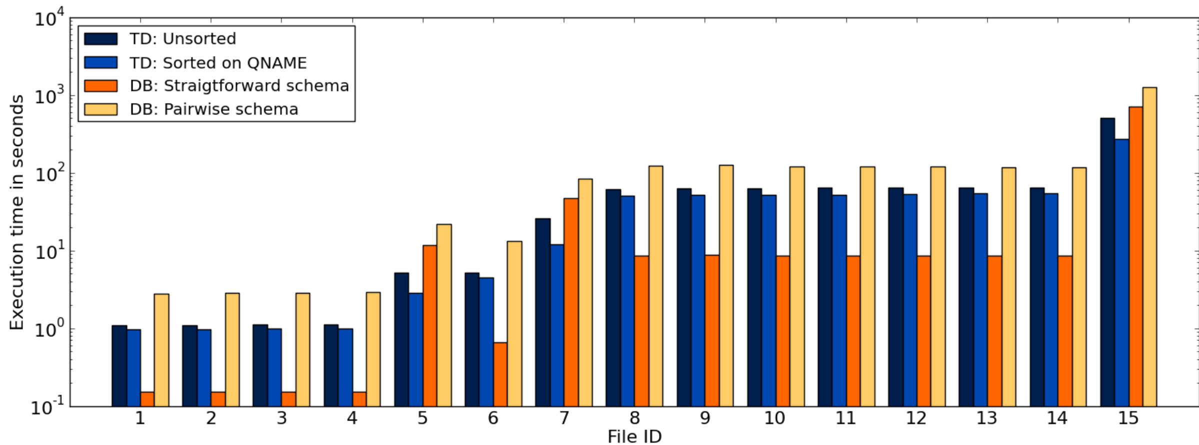


Figure 6.4: Execution time of use case 2.3 using the full-output approach

Pairwise storage schema

The pairwise storage schema obviously improves query performance whenever the query operates on paired alignment data. However, in some other cases the pairwise storage schema yields better performance too, for example for use case 1.4, as can be seen in figure 6.5. A cause for these differences are the increased possibilities to operate on the data in a multithreaded fashion, since the data is subdivided over more columns.

Comparison between the DBMS and the traditional implementation

One of the most important conclusions that have to be drawn is how the DBMS implementation performs compared to the traditional implementation. In general, there

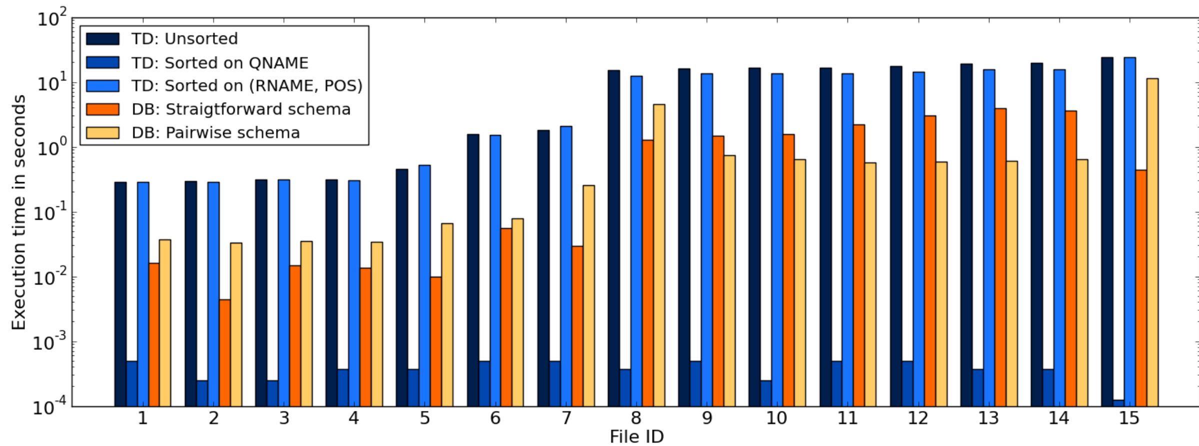


Figure 6.5: Execution time of use case 1.4 using the full-output approach

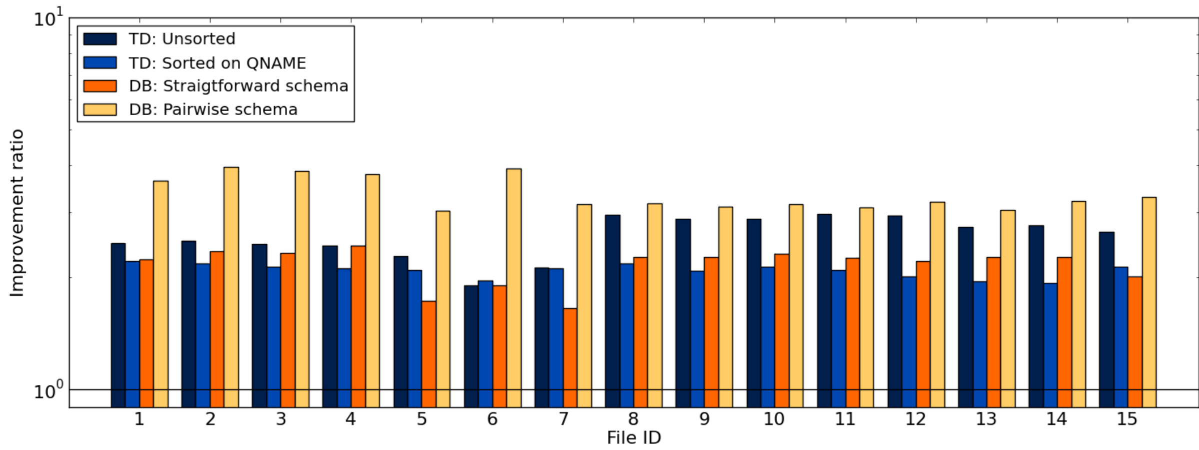
is often a traditional method that solves a use case faster than both DBMS methods. However, this requires a BAM file to be ordered in a specific way. Whenever the ordering of a BAM file can not be exploited by the traditional method, the DBMS implementation is often faster. This is especially true for the pairwise storage schema whenever a use case needs to operate on paired alignment data.

6.2.3 Solving use cases using the minimal-output approach

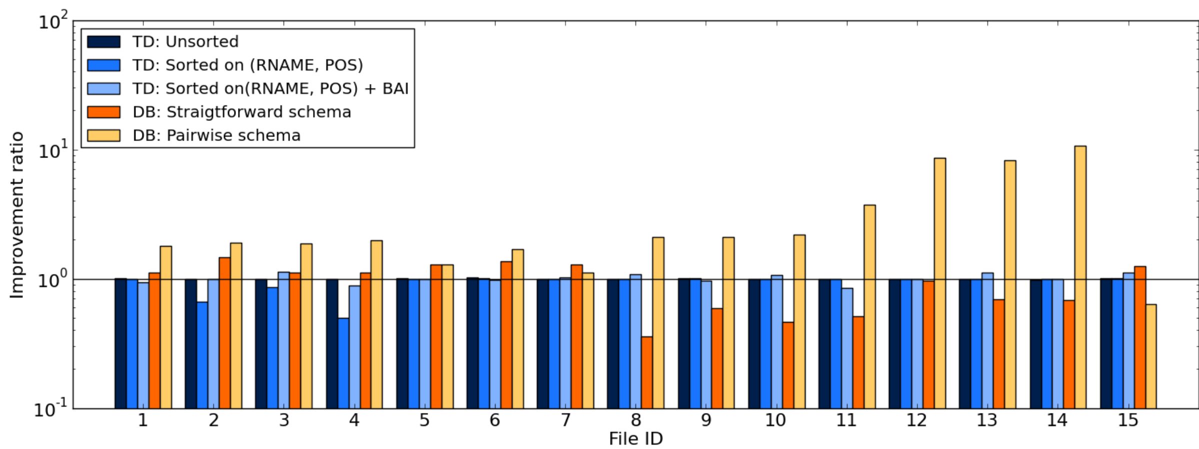
Since we are mostly interested in the performance of the minimal-output approach compared to the previous, full-output approach, we only plotted comparative bar charts to show the results for the minimal-output approach. Figure 6.6 shows these charts for respectively use case 1.1 and 1.3. Such a bar chart displays the improvement factor for using the minimal-output approach for the running times of all different methods. So if for a certain triple (use case, file, method) the running time is x_f using the full-output approach and x_m using the minimal-output approach, the comparative bar chart shows x_f/x_m for this triple. Therefore, a value higher than one indicates an improvement, since the running time decreased for the minimal-output approach. In the same way, a value lower than one indicates a regression. All comparative plots contain a horizontal line at $y = 1$, to easily see which running times have improved.

The first thing that can be concluded from Figure 6.6a is that all running times decrease when the minimal-output approach is used, since every bar ends above the $y = 1$ line. The pairwise storage schema of the DBMS implementation benefits the most from this approach, which is caused by the reduced complexity of the union operations that have to be done, since the number of columns that are involved in this union are heavily decreased. For most files, the traditional method that works on input files sorted by their *QNAME* has a smaller improvement ratio than the traditional method that works on unsorted files.

Figure 6.7 shows the effect on the difference in writing times when the minimal-output approach is used. From comparing this figure with figure 6.3, we can easily see that the overhead for writing is now not as big as it was for the full-output approach. This leads to the observation that small result sets obtained by the DBMS contribute to a good



(a) Use case 1.1



(b) Use case 1.3

Figure 6.6: Running times of solving use cases on small files using our different methods. These are the results for the minimal-output approach and are the averages of ten runs. Results for use case 2.2 are not presented, since the minimal-output approach is not defined for use case 2.2

performance.

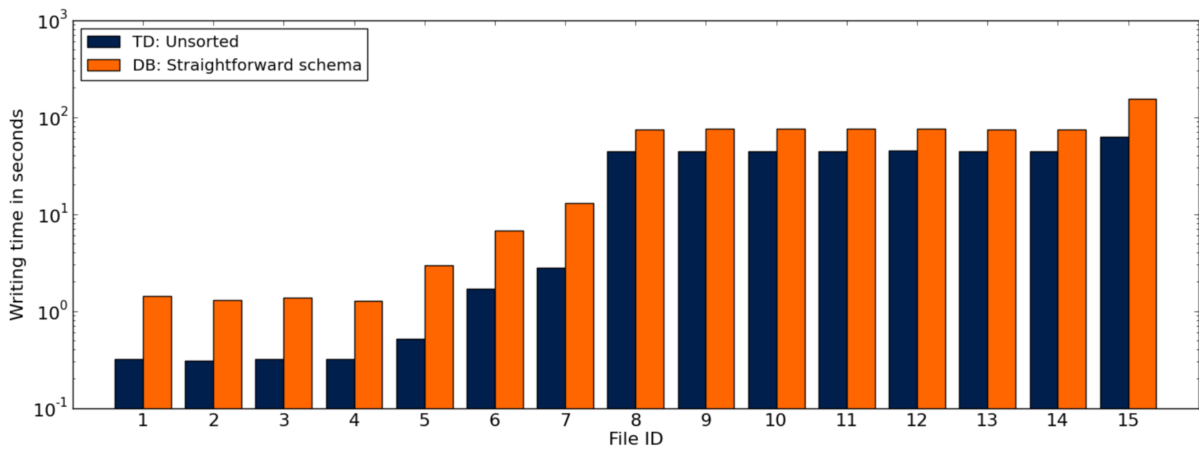
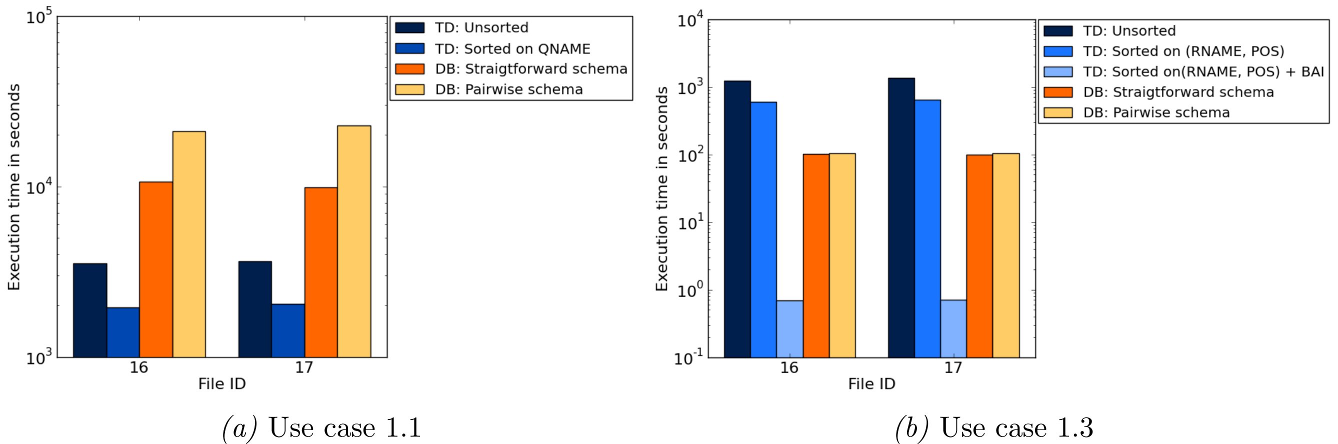


Figure 6.7: Time required to write the result to disk for use case 1.1 using the minimal-output approach

Figure 6.6b shows less consistency in the comparison patterns for the different files. The first things that stands out from this figure is that for some files, there are methods that perform worse than for the full-output approach. For the DBMS implementation, this can be explained by the smaller amounts of data that have to be collected, giving rise to fewer possibilities to exploit multithreading. For the traditional implementation however, these events occur only if the original running time was already in the order of milliseconds and therefore the increase in running time is still negligible. Furthermore, the pairwise storage schema for the DBMS implementation improves the most for many files. A cause for this improvement is the decrease in data projection needs, which frees resources for parallelization of filtering operators.

No bar chart is included for use case 2.2, since the minimal-output approach is not defined for use cases 2.1, 2.2 and 2.5. For all other use cases, a bar chart is included in Appendix E.2. Many charts show that the DBMS implementation regresses for bigger files, while the traditional implementation still improves. The charts for use cases 2.11 and 2.12 however show an enormous improvement for the DBMS implementation. In general, we can say that all of our implementations mostly improve their running times when they apply the minimal-output approach. However, these trends do not show as drastically as we had hoped for.

Since we did not apply the full-output approach to the big files in our repository, we plotted the results for the minimal-output approach on the big files similar to the way we plotted the results for the full-output approach on the small files. Figure 6.8 shows these plots. Note that again, results for use case 2.2 are not included, since only simple use cases were solved on the big files.



(a) Use case 1.1

(b) Use case 1.3

Figure 6.8: Running times of solving use cases on big files using our different methods. These are the results for a single run using the minimal-output approach. Results for use case 2.2 are not presented, since only the simple use cases were solved on the big files of our repository.

The pattern that we see in Figure 6.8a is exactly the same as the pattern that occurs in Figure 6.2a and hence the same explanations hold for the big files.

Figure 6.8b also has many similarities with Figure 6.2b, although there are some subtle differences as well. Both storage schemas of the DBMS implementation still outperform the unsorted traditional implementation in every case. However, for big files they also outperform the traditional implementation that operates on files sorted by *QNAME*.

Furthermore, the traditional method now outperforms all other methods if it is provided with a BAI file.

Bar charts for all simple use cases, solved on big files, can be found in Appendix E.3. One thing that really stands out is the outstanding performance of the pairwise storage schema for use cases 1.4 and 1.5 compared to its performance for small files. Again, this can be explained by the increased opportunities for exploiting parallel data operations, due to the distribution of the alignment data over more columns than for the straightforward storage schema.

6.2.4 Correlations in the results

Investigating the correlation between different possibly interesting sets of variables is a useful tool to compare the scalability of the different methods we presented in this thesis. Let $X = \{(x_1, y_1), (x_2, y_2), ...\}$. X is positively correlated if an increase in the x values implies an increase in the y values. It is negatively correlated if an increase in the x values implies a decrease in the y values. It is not correlated if such a relation doesn't exist. We searched for correlations by creating scatter plots and calculating the Pearson correlation. [2] This yields a value in the range $[-1, 1]$, where negative outcomes yield a negative correlation and positive outcomes yield a positive correlation. The closer the outcome is to -1 (1), the stronger the negative (positive) correlation is. Furthermore, an outcome of 0 implies no correlation at all. In the scatter plots presented in this section, the corresponding Pearson correlation is displayed as ρ.

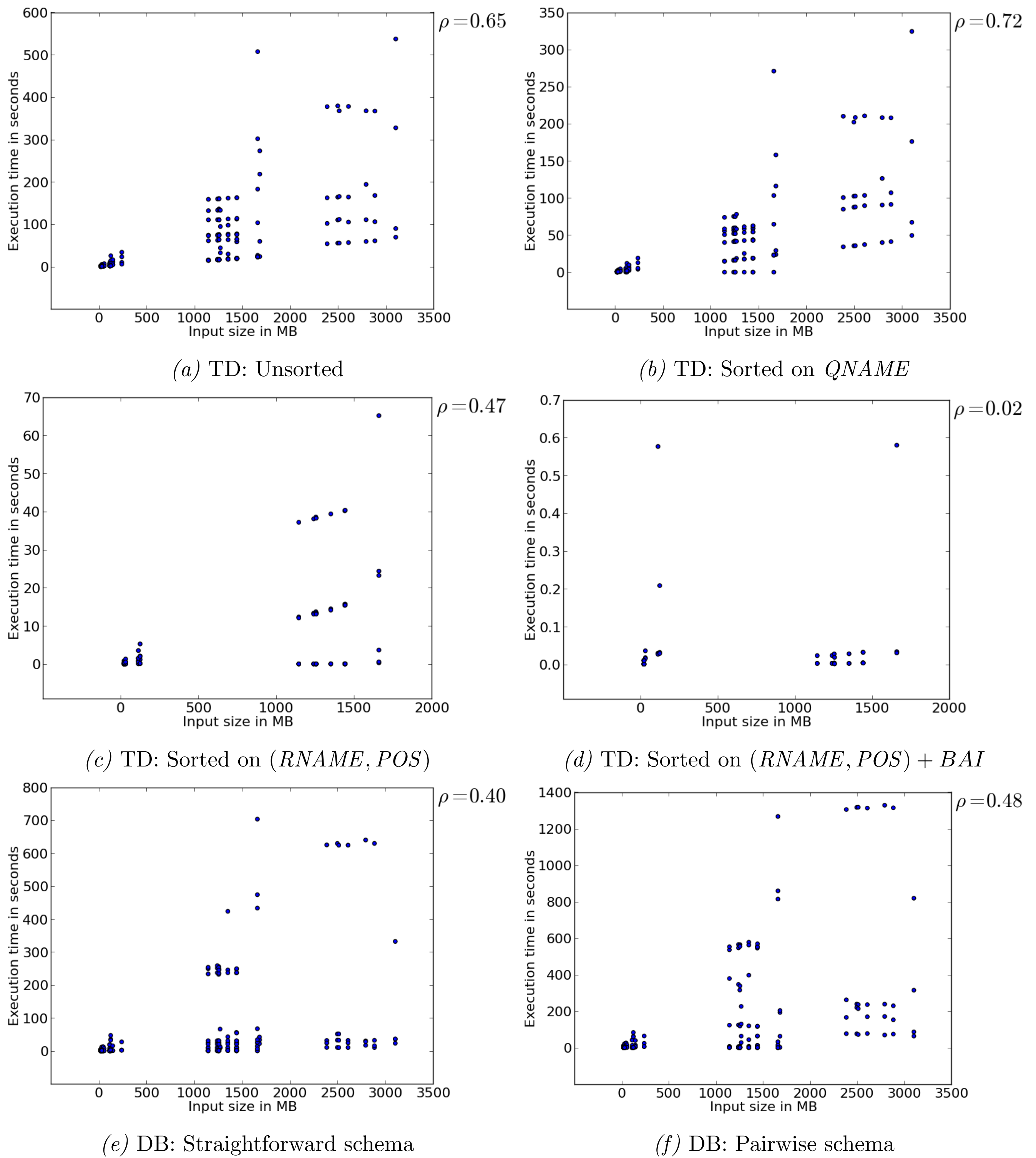
BAM file size versus running time of use cases

An interesting relationship is that between the size of the BAM files and the running times of the use cases. We expect to see a positive correlation, since a use case will generally need more time to complete if it has to process more data. We created scatter plots for all different methods, displayed in Figure 6.9. Note that there are measurements in these plots with a file size bigger than the file size of any file among the small files in our repository. This is caused by the fact that we chose to sum the file sizes of two BAM files if they are both input to a single use case, which is the case for use cases 2.5 until 2.8.

Every scatter plot contains data from exactly one solving method, yielding $m_i \cdot 15$ (x, y)-tuples per scatter plot, where m_i equals the number of use cases that are solved using method i, according to Table 6.3. For the traditional implementation, this yields 255 (x, y)-tuples for the unsorted variant, 195 for the *QNAME* sorted variant, 90 for the $(RNAME, POS)$ sorted variant and 45 for the variant that includes a BAI file. Since every use case is solved on both storage schemas of the DBMS implementation, the scatter plots for these methods also include 255 points.

One thing that can be seen from comparing Figures 6.9a and 6.9b is that they have approximately the same pattern. However, Figure 6.9b contains faster running times, as can be seen on the scale of the y-axis. This means that the traditional approach speeds up every use case with approximately the same factor when operating on files that are

[2] Weisstein, Eric W. "Correlation Coefficient." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/CorrelationCoefficient.html`



(a) TD: Unsorted

(b) TD: Sorted on *QNAME*

(c) TD: Sorted on $(RNAME, POS)$

(d) TD: Sorted on $(RNAME, POS) + BAI$

(e) DB: Straightforward schema

(f) DB: Pairwise schema

Figure 6.9: Scatter plots that shows BAM file size versus running times of use cases. Every plot mentions its Pearson correlation ρ in the right top.

sorted by *QNAME*. The same relation exists between Figures 6.9e and 6.9f. They have approximately the same pattern, but the results for the pairwise storage schema are on a larger y-scale. So although the pairwise storage schema speeds the DBMS

implementation up tremendously for some of the queries, it slows down all other queries with approximately the same factor.

Almost all the scatter plots show a convincing positive correlation, meaning that the bigger the input becomes, the longer the use case takes to finish. This is especially true for the traditional implementation that works on either unsorted files or on files sorted by $QNAME$, shown in Figures 6.9a and 6.9b. Both storage schemas for the DBMS implementation and the traditional implementation on files sorted by $(RNAME, POS)$, respectively shown in Figures 6.9e, 6.9f and 6.9c, also show this correlation, although it is not as emphatic as for the first two methods.

Figure 6.9d shows near to no correlation, meaning that the use cases that are able to exploit a BAI file benefit so much from this that the size of the input files doesn't influence the running time anymore. This can be easily explained by noting that a BAI file allows the traditional implementation to do random access in a BAM file to extract the alignments in a certain region, thereby eliminating the size of the BAM file as a factor.

So unless a use case can exploit a BAI file, these plots show that the straightforward storage schema of our DBMS implementation scales the best, as its running times depend the least on the size of the input.

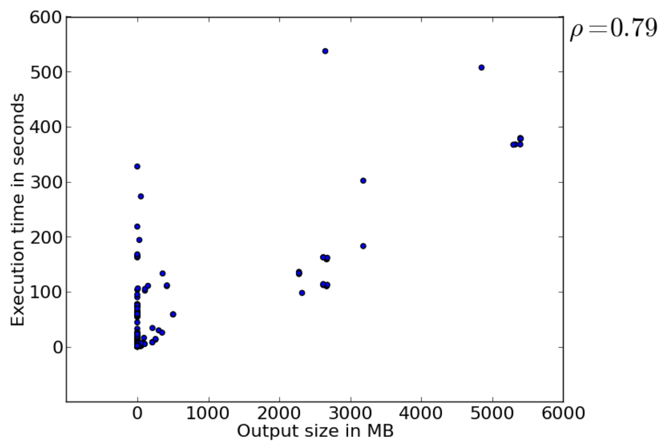
Output file size versus running time of use cases

Another interesting relationship is that between the size of the output that is generated by a use case and the time it took the use case to conclude to this output. Like for the previous relationship, we plotted a scatter plot for every of our implementation methods. These plots are shown in Figure 6.10. The plots contain exactly as many (x, y)-tuples as in Figure 6.9 and every plot again displays the corresponding Pearson correlation.

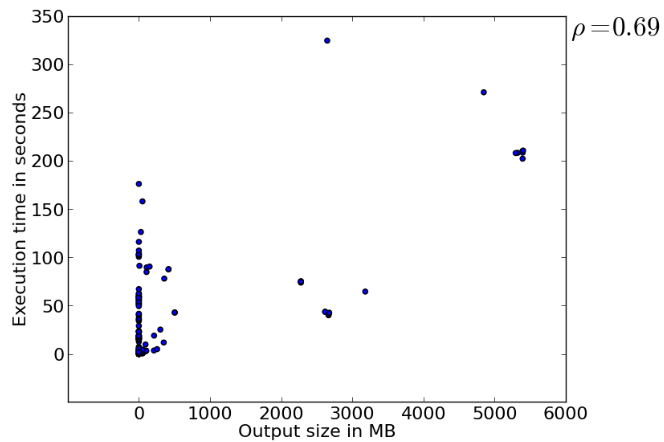
The most important thing that stands out in these plots is the high correlation that this relation has. So use cases that have a big result set take a long time to execute. This correlation is the highest for both storage schemas of the DBMS implementation, shown in Figures 6.10e and 6.10f, meaning that the performance of the DBMS implementation regresses faster than the performance of the traditional approach if big output sets must be written. This matches our expectations, since the DBMS implementation has to send its entire result set over a TCP socket before a client application can write it.

Furthermore, the Pearson correlation is undefined for the traditional implementation that exploits a BAI file, shown in Figure 6.10d, since all of the output sizes for this implementation equal zero and this would yield a division by zero. We would however expect use cases that exploit the BAI file to be positively correlated, since writing of results would be the bottleneck of this operation due to the tiny use case execution times thanks to the BAI file.

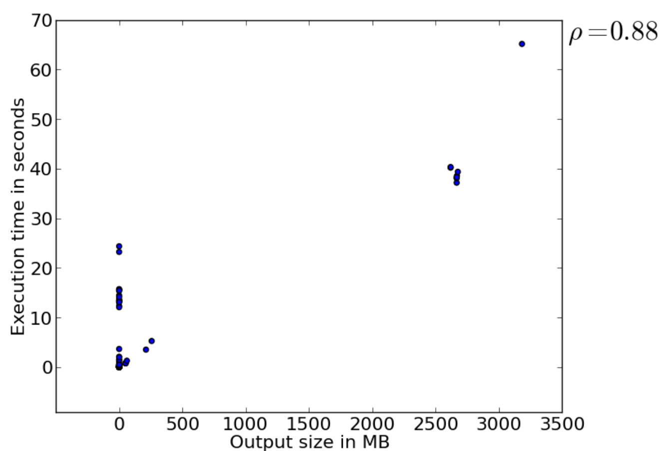
An overall conclusion that can be drawn from these plots is that when the DBMS implementation is used, high selectivity of use cases tend to significantly improve the performance. This is to a more limited extent also the case for the different methods of the traditional implementation.



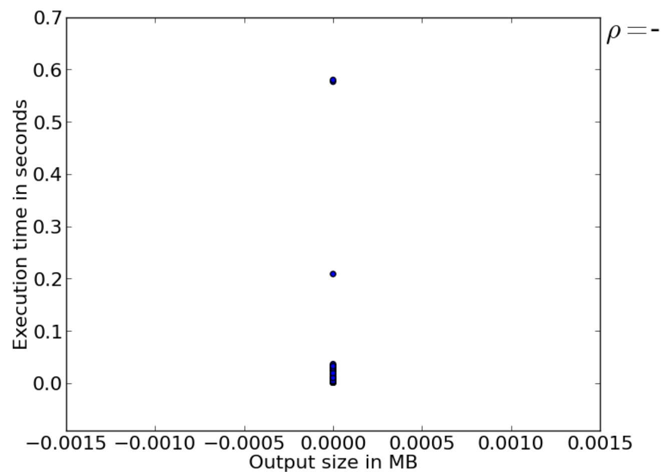
(a) TD: Unsorted



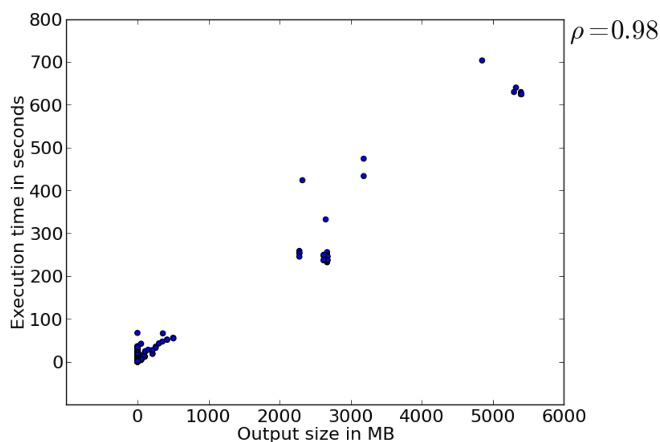
(b) TD: Sorted on $QNAME$



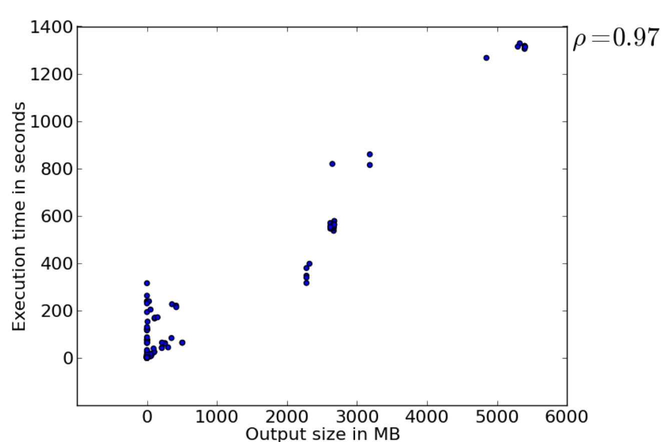
(c) TD: Sorted on $(RNAME, POS)$



(d) TD: Sorted on $(RNAME, POS) + BAI$



(e) DB: Straightforward schema



(f) DB: Pairwise schema

Figure 6.10: Scatter plots result set output size versus running times of use cases. Every plot mentions its Pearson correlation ρ in the right top.

6.2.5 Pros and cons of our implementations

This section discussed the results of many experiments. The main aim of our experiments was however to identify the pros and cons of the implementations presented in Chapter 5. Therefore, this section interprets the results that we just presented to conclude to these pros and cons.

The most obvious con of the DBMS implementation is its initialization time, as is illustrated in figure 6.1. However, a con of the traditional implementation is that it often needs its input file sorted on some attribute in order to perform good. Whenever such an ordering can not be exploited by the traditional approach, the DBMS often performs better. Furthermore, the correlation plots in figure 6.9 show that the DBMS implementation generally scales better with the size of the input files. The only exception is if the use cases predominantly can exploit a BAI file to obtain the required results, which is certainly not the case for the broad range of our use cases. Figure 6.10 shows another con of the DBMS implementation. It shows that the correlation between the size of the output and the running time of the use cases is much higher for the DBMS implementation than it is for the traditional implementation. Thus, in order for the DBMS implementation to perform good compared to the traditional implementation, the use cases that are solved must have a relatively small output. This correlation is also illustrated by comparing the writing times required for use case 1.1 using the full-output and the minimal-output approach, as is done by figures 6.7 and 6.3.

Another pro of the DBMS implementation might be that it performs better than the traditional implementation when the data in main memory becomes bigger than the actual physical main memory of the system, since MonetDB might handle such queries better than the traditional approach, which relies on the swapping mechanism of the OS in such cases. We however didn't have the time to perform these tests, since our machine had 256G of main memory, which is plenty to perform our experiments in main memory.

The results presented in this section, together with the other reasons to use a DBMS mentioned in the introduction (Chapter 1), show that a DBMS is favorable over the traditional approach.

Chapter 7

In-situ processing

A big disadvantage of our DBMS implementation (Section 5.1) is the required initialization time when doing a full database load, since it takes significantly more time than initializing the file repository for the traditional approach (Section 5.2). Furthermore, our DBMS implementation has a storage requirement of up to five times the size of the original BAM file repository, since the alignment data is stored uncompressed in the DBMS. For these reasons, applying some form of in-situ processing is worth considering, where part of the alignment data will be loaded from the BAM files only when it is needed. We chose to adapt the Data Vault Framework (DVF), proposed in [8, 7, 10], to our DBMS implementation. Section 7.1 gives a brief summary of the DVF. Section 7.2 then elaborates on how the DVF can be applied to our DBMS implementation, since this is far from trivial. Finally, Section 7.3 presents some results that we obtained with our DVF implementation.

7.1 The Data Vault Framework (DVF)

The DVF is profiled as a symbiosis between database technology and external file-based repositories. It maintains the original data, while at the same time opening it up for analysis and exploration through database technology [7]. It is known to largely reduce data-to-query times and avoid redundant storage of data. Only the very basic idea of the DVF will be explained here. For further details, [8, 7, 10] can be consulted.

The Data Vault is initialized by providing it with a link to a file repository. Every encountered file in this repository provides the DVF with meta data, which it interprets and stores inside a pre-loaded database schema. The DVF proposes that the majority of a file is untouched when meta data is extracted from it, which greatly reduces initialization time.

When a query is fired at an initialized Data Vault, it is parsed and compiled into a relational algebra plan as is done for any regular query. At that point, the query optimizer identifies for which parts of the relational algebra data from the original files is necessary and adds appropriate *mount* functions to the relation algebra plan. Such a mount function reads data from the original files and 'mounts' them to the database for as long as the query runs, thereby providing Just-In-Time loading. The query plan can then be executed. Hence, by executing a query on the initialized Data Vault it seems as if all data is resident in the database.

Figure 7.1 illustrates the DVF, applied to MonetDB. An application can use the initialized Data Vault by using any query language supported by the DBMS engine, for instance SQL. This query is then parsed and compiled by the DBMS (MonetDB) and the Data Vault makes sure that the necessary external data is mounted Just-In-Time.

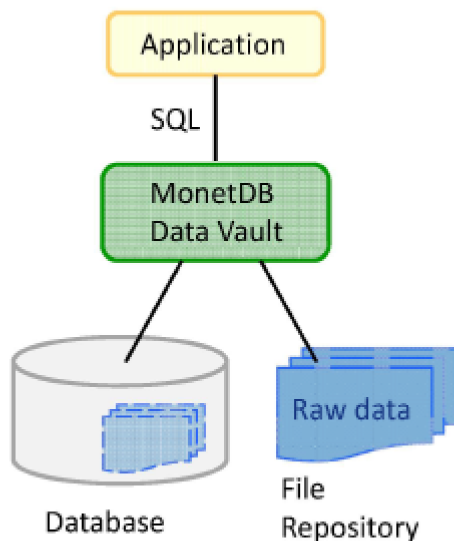


Figure 7.1: Simple illustration of the architecture of the DVF, applied to MonetDB [7].

7.2 Applying the DVF to BAM data

The DVF has already proven useful for several file formats, like mSeed, GeoTIFF and FITS [8]. Applying the DVF to BAM data however comes with two major problems.

1. The initialization step of the DVF proposes loading only meta data in the database, without touching the other data, thereby minimizing initialization time. Since the header of a BAM file does not contain useful information for solving our use cases, we have to access the alignment data to get some meaningful meta data. However, to extract fields from alignments in a BAM file, the whole BAM file has to be read, since alignments are compressed as a whole. This makes it impossible to directly apply the existing DVF on BAM files.

2. The DVF proposes that the mount operation operates on a per-file basis, which for BAM files obviously is sub-optimal due to the potentially huge file sizes.

How we dealt with these problems is presented in Sections 7.2.1 and 7.2.2 respectively.

7.2.1 Initializing the Data Vault

Unfortunately, the header of a BAM file does not contain information that would make useful meta data. Therefore, we chose to use fields from the alignments as meta data. In

order to extract these fields, there is nothing that we can do but read all BAM files in the repository in their entirety.

To make a deliberate choice on which fields to store in the database, we consulted the storage requirements for every column in the alignments tables in the database implementation mentioned in Chapter 5. Figure 7.2 shows the percentage of storage taken by the different alignment fields. This percentage was calculated over the average storage requirements of the small BAM files in our repository. The following can be seen from this figure:

- The fields *FLAG*, *RNAME*, *MAPQ* and *RNEXT* all have very low storage requirements. For the *FLAG* and *MAPQ* fields this makes sense, since these fields are stored as 16-bit integers. The *RNAME* and *RNEXT* fields however contain chromosome names, encoded as variable length strings. The small storage requirements can however be easily accredited to the dictionary encoding mechanism that MonetDB uses for strings. The *RNAME* and *RNEXT* fields always contain only a few distinct values (mostly < 30). MonetDB will create a string heap for these values. Every alignment will then maintain a pointer to a string inside this string heap.

- The *POS*, *CIGAR*, *PNEXT* and *TLEN* fields also have low storage requirements. This can be easily explained for the *POS*, *PNEXT* and *TLEN* fields, as they are stored as 32-bit integers. The *CIGAR* field again exploits the dictionary encoding from MonetDB, as most *CIGAR* strings turn out to equal iM, where i equals the length of the sequence string.

- The *QNAME* needs a fair amount of storage space, which seems reasonable since many distinct string values exist inside a BAM file.

- The *SEQ* and *QUAL* fields require much storage space, which can be explained by their many characters (somewhere between 50 and 100) and the fact that most of these strings will be distinct.

- The *EXTRA* field occupies almost 50% of the total storage requirements, since many alignments contain much extra information.

Based on these observations we chose to consider all fields except for the *SEQ*, *QUAL* and *EXTRA* fields as meta data, as these three fields together make up 89.5% of the storage requirements and they are never used in any filtering step of our use cases. Hence, they are not even needed during execution of the queries mentioned in Appendix C until the result set is known. The *EXTRA* field is even never projected when a result set is known, so not loading this field is an obvious choice. Another field that we considered to not load during initialization is the *QNAME* field, since it accounts for another 6.3% in the storage requirements. However, since many of our use cases heavily depend on this field during the filtering, this would decrease performance dramatically. Therefore, we chose to accept the additional storage requirement for this field.

Figure 7.3 illustrates the storage schema that we designed for our DVF implementation. It is separated into two sections: a meta data part and an external data part. The meta data part will be filled with all the data from the BAM file repository upon initialization, while the external data part will remain empty. The Data Vault will apply

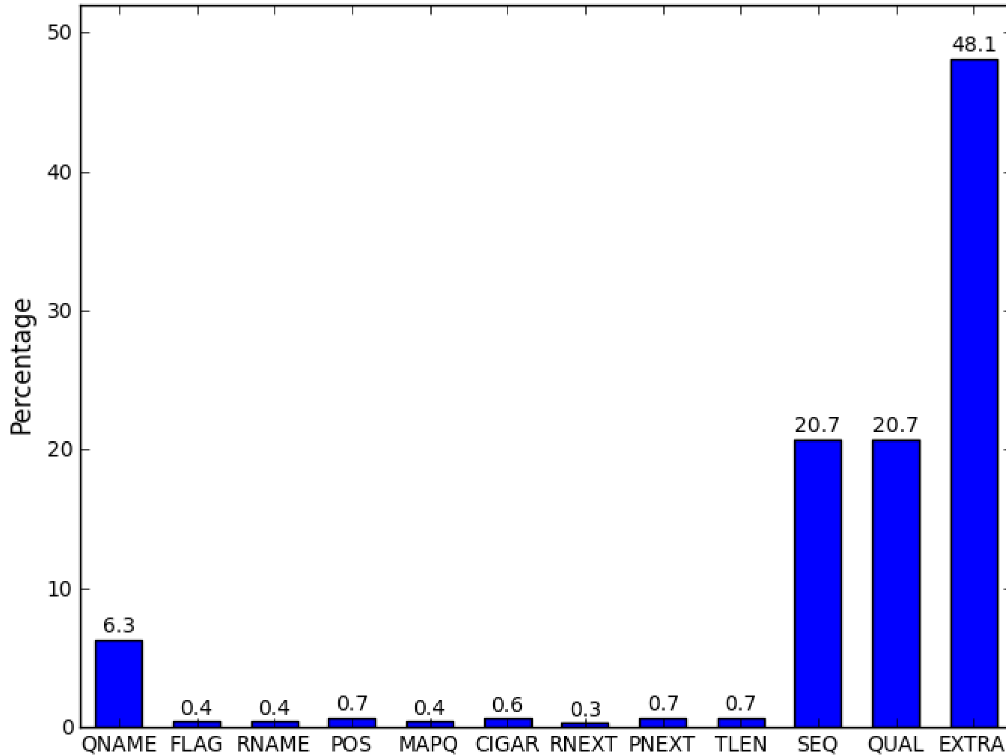


Figure 7.2: Distribution of storage requirements of the different alignment fields.

Just-In-Time loading to populate the external data tables when they are required by the queries.

In order to initialize the Data Vault, we adapted our BAM loader, described in 5.1.2, to be able to load BAM files into the schema of our Data Vault.

7.2.2 Mounting data from BAM file repository

When a query is executed, the DBMS parses and compiles this query and then executes it. Using the meta data that is already present in the database, some queries can be answered directly. Most queries from Appendix C however, will at some point need the external data. This is performed by the Data Vault by executing a mount operation that retrieves the required external data and inserts it into the DBMS. The original DVF expects that a file location is passed as an argument to the mount operation. The mount operation will then open the file on this file location and insert all the external data from this file into the database. This is however not a suitable approach for our DVF implementation for BAM files, due to their potentially huge file size. Therefore, our mount operation works with a file location, combined with a list of virtual offsets. The signature of the mount operation is *mount(string file_location, bigint voffset[])*, where the string *file_location* has to be an absolute path to a BAM file. During query time, all tables that contain alignment data have a title that ends with the file id of the corresponding

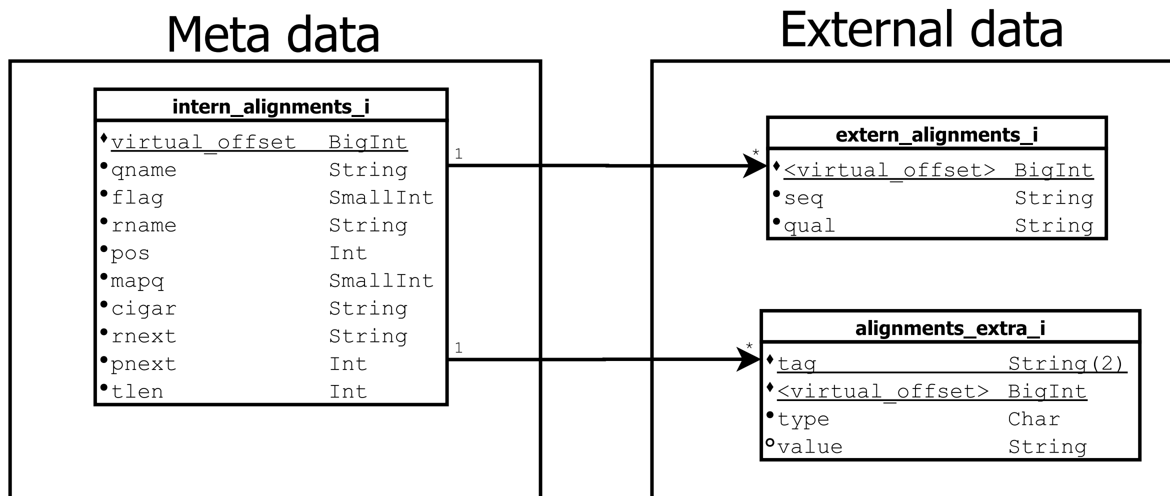


Figure 7.3: Storage schema for our DVF implementation for BAM files.

BAM file. Hence, this file id can be used to retrieve the correct absolute path from the files table. The other argument represents an array of virtual offsets, where every virtual offset is stored in a big integer.

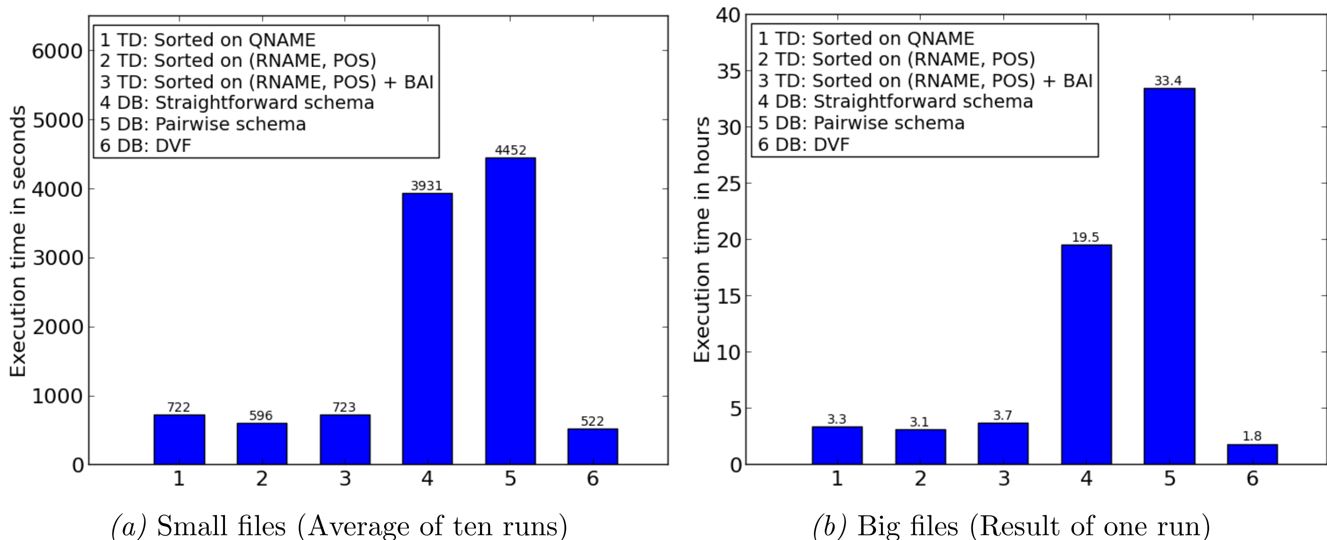
The mount operation then uses Samtools [11] to extract the alignments on the locations indicated by the list of virtual offsets. The resulting alignments are then inserted into the DBMS, and the execution of the query can finish.

7.3 Experimental results

We extended Figure 6.1 with the initialization time of our Data Vault for both the small and the big files in our repository. The bar charts including these initialization times are shown in Figure 7.4.

7.3.1 Initialization

What can be seen in these charts is that initializing our Data Vault takes less time than initializing any of the other implementation methods that we presented (except for the traditional implementation on unsorted files, since that method doesn't need to initialize). Furthermore, Table 7.1 shows the storage requirements of the original BAM repository, the initialized straightforward storage schema and the Data Vault. The storage requirements shown for the straightforward storage schema excludes the file size of the original file repository, since the BAM files become superfluous when all their data are stored in the DBMS. However, the presented storage requirements for the Data Vault implementation include the file size of the original file repository, since the original files are still needed to perform Just-In-Time loading. Table 7.1 clearly shows that the storage requirements are much more acceptable for the Data Vault implementation. The storage requirements could be reduced even further by using a compressed file system.



(a) Small files (Average of ten runs)

(b) Big files (Result of one run)

Figure 7.4: Required running time by different methods to initialize all BAM files in our repository.

	Small files	***Ratio***	***Big files***	***Ratio***
Original BAM files	11,162M	100%	192G	100%
Straightforward storage schema	54,021M	484%	944G	492%
Data Vault + original files	17,171M	154%	288G	150%

Table 7.1: Storage requirements for the original files, straightforward storage schema and the Data Vault

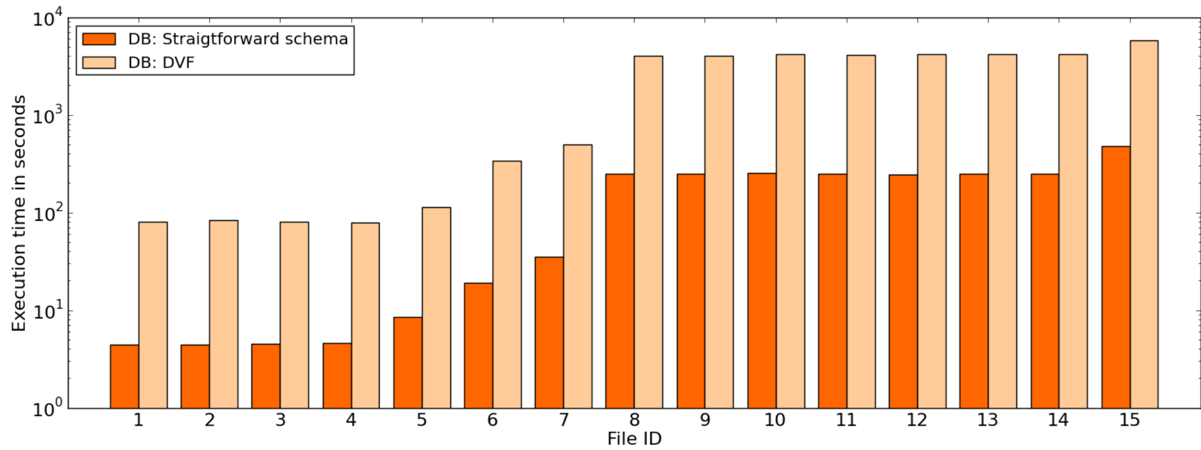
7.3.2 Solving use cases

Although the major problems of the DBMS implementation have been solved by the Data Vault implementation, the Data Vault has to apply Just-In-Time loading whenever it needs to access data that only resides in the original BAM files. This Just-In-Time loading will generally take longer than retrieving the data from an already loaded database table. Therefore, we expect the running times of the queries to be significantly higher for our Data Vault implementation compared to the straightforward storage schema of the DBMS implementation. We solved all of our use cases using the Data Vault implementation, using the full-output approach. Due to limited time, we could however only do a single run and thus the results aren't as reliable as the results for the small files of the DBMS implementation. Figure 7.5 shows bar charts that compare the running times for the original DBMS implementation and the Data Vault implementation for use cases 1.1, 1.3 and 2.2.

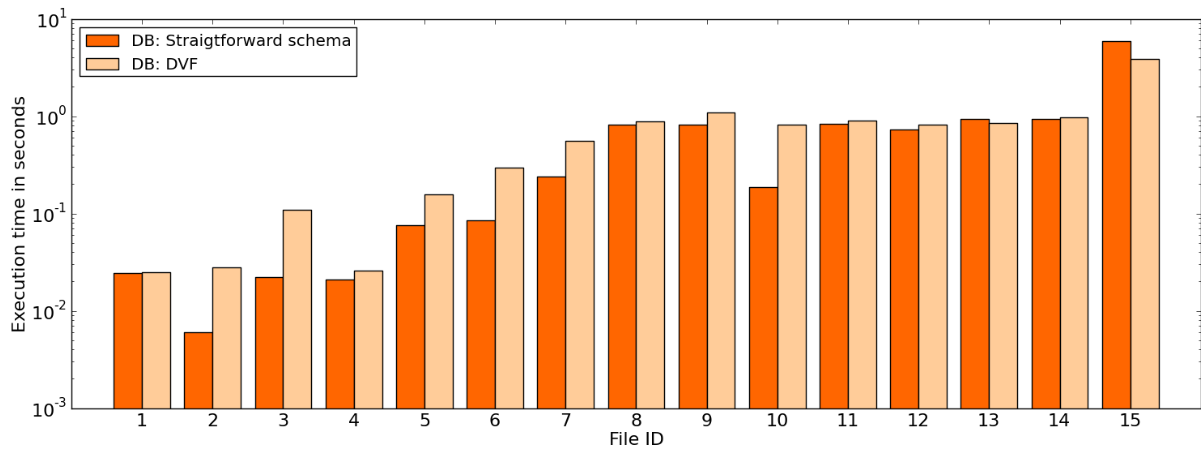
The running time for the Data Vault implementation is significantly higher for use case 1.1, as can be seen from Figure 7.5a. This can be explained by the relatively big size of the result sets for this use case. Due to this, the Data Vault has to load many *SEQ* and *QUAL* strings into the database during query execution.

For use case 1.3, the running time of the Data Vault implementation does not differ much from that of the original DBMS implementation.

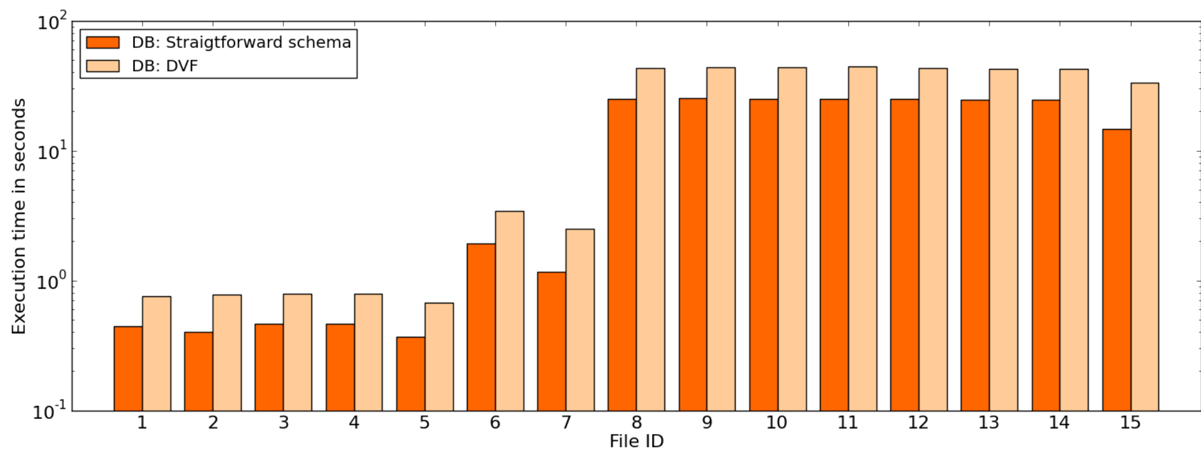
For use case 2.2, the running time of the Data Vault implementation is again somewhat bigger for every file, while use case 2.2 never uses the *SEQ* and *QUAL* fields and thus no Just-In-Time loading occurs. This regression can be explained by the difference in



(a) Use case 1.1



(b) Use case 1.3



(c) Use case 2.2

Figure 7.5: Running times of solving use cases on small files using the straightforward storage schema and using the Data Vault.

the query plan optimizer pipelines. For the Data Vault implementation, we use another optimizer pipeline that enables us to perform the Just-In-Time loading. This causes MonetDB to use somewhat different physical query operators and this can result in a

performance gain or loss.

Bar charts for all use cases can be found in Appendix E.4. Most of these charts show similar patterns as the ones already discussed. However, for use case 2.3 and 2.4 the performance of the pairwise storage schema is much worse than the performance of the straightforward storage schema, while the result set is always relatively small. This could be due to the difference in the optimizer pipeline. Our Data Vault implementation is just in an experimental state however and it could be the case that the custom made optimizer unnecessarily loads external data for these use cases. Most other differences can be ascribed to the different optimizer pipelines, irrespective of whether it is an improvement or a regression.

Chapter 8

Conclusions and future work

For this thesis, we looked at the possibilities of using a DBMS for analyzing and managing big genome data. We formally defined a broad range of use cases that are applied on an everyday basis by bioinformaticians on BAM files. We then developed both a DBMS implementation, based on MonetDB, and a traditional implementation, written in C, to solve these use cases and we compared the results of solving the use cases on both implementations. The initialization time of the DBMS implementation is really high compared to that of the traditional implementation, because all data have to be loaded into the DBMS. However, in terms of running times for solving the use cases, the traditional implementation is often outperformed by the DBMS. Furthermore, experiments show that solving use cases using the DBMS implementation scales better to bigger BAM files than the traditional implementation. Experiments have also shown that the DBMS implementation performs best compared to the traditional implementation when the size of the result is relatively small.

To reduce both the initialization time and the storage requirements of the DBMS implementation drastically, we applied the Data Vault Framework to our DBMS implementation. The query performance of our Data Vault implementation is already close to that of the original DBMS implementation for most queries. However, our implementation is still experimental and work needs to be done to really expose the power of the DVF.

The research question for this thesis was 'How can a DBMS be exploited to better support analysis on DNA sequence alignment data?'. The results of our experiments, combined with the many advantages of using a DBMS to analyze data, show that using the DBMS implementation is already favorable and hence the research question is answered by our DBMS implementation.

The Data Vault implementation for BAM files would even be a better answer to the research question, since it solves the main problems that exist for the DBMS implementation. Therefore, more research on the Data Vault implementation has the potential to have a huge impact on the everyday analysis of bioinformaticians.

8.1 Future work

This paper presents a good start for bridging the gap between big genome data and DBMSs. There is however much more work that can be done in addition to this start.

This section gives an overview of things that could be done to continue this research.

8.1.1 Improving the Data Vault implementation

The most straightforward and promising future work that can be done is the improvement of the Data Vault implementation, since right now there is just an experimental implementation. Some ideas for further work on the Data Vault include:

- Create a stable version of the Data Vault implementation, that works on any SQL query. The current implementation could for example be made more robust by moving functionality from the physical plan optimizer to the relational algebra optimizer.
- For the straightforward storage schema in our DBMS implementation, the user can choose which alignment fields he/she wants to store. Do something similar for the Data Vault implementation.
- The Data Vault implementation currently performs Just-In-Time loading as a single operation. There is a possible performance gain by dividing this operation in multiple independent sub-operations, that can be executed concurrently.
- Incorporate the Variant Call Format (VCF) file into the Data Vault implementation as an extra layer of meta data. If a query is executed, the Data Vault then first sees which information it can extract from its meta data, then it consults the VCF file for data that it doesn't have and only as a last resort the BAM file is consulted.
- Add a synchronization mechanism that synchronizes changes between the data in the DBMS and in the BAM files.
- Implement a caching mechanism for the Data Vault implementation that temporarily stores Just-In-Time loaded data.
- Implement a feedback mechanism that asks the user for permission whenever Just-In-Time loading is about to load a huge amount of external data.

8.1.2 Experimenting

It would be useful to do more experiments on big BAM files, since right now we can only speculate about the scalability of our methods. More experiments would also be very useful for our Data Vault implementation. It would also be interesting to see what happens to the performance of both the traditional and the DBMS implementations when they run out of main memory. We would expect MonetDB to behave better in such an event, since the traditional implementation has to rely on the swapping mechanism of the OS.

Yet another possibility for experimenting would be to try to use Oracle's technology for bioinformatics [2] to solve our use cases and compare the results with the results presented in this thesis.

8.1.3 Compressed file systems

All data that we load in the DBMS is decompressed first. Therefore, data blows up whenever it is inserted in the DBMS to almost five times its compressed size. To prevent this, research could be done on running a DBMS on a compressed file system.

8.1.4 Querying *EXTRA* data

In our BAM repository, ~50% of the data is contained in the *EXTRA* field of the alignments. Currently, all of this data is stored a strings in a separate table. Whenever use cases arise that want to use this data as a filtering step, performance might drop drastically. Therefore, a more clever solution could perhaps be implemented and tested.

8.1.5 Data visualization

Extracting data from a DBMS is one thing, presenting it in a useful way for analyses is another. There exist several software solutions that visualize alignment data using a GUI, of which one of the most prominent is IGV. [1] It might be an interesting additional feature to be able to render the output of the DBMS to such a GUI. To do this efficiently, more research needs to be done in how query results can be outputted more efficiently.

8.1.6 Distributed file systems

Whenever a user uses a distributed file system, BAM files are scattered across multiple nodes. It might be interesting to develop a DBMS implementation that exploits the multithreading possibilities that a distributed file system gives rise to.

[1] IGV: `http://www.broadinstitute.org/igv/`

Bibliography

[1] I. Alagiannis, R. Borovica, M. Branco, I. Stratos, and A. Ailamaki, *NoDB: Efficient Query Execution on Raw Data Files*, Proceedings of the ACM SIGMOD International Conference on Management of Data, 2012.

[2] B. Blackwell and S. Ravada, *Oracle's Technology for Bioinformatics and Future Directions*, First Asia-Pacific Bioinformatics Conference (APBC2003) (Yi-Ping Phoebe Chen, ed.), vol. 19, 2003, pp. 35–42.

[3] D.I. Boomsma, C. Wijmenga, E.P. Slagboom, M.A. Swertz, L.C. Karssen, and A. Abdellaoui et al., *The Genome of the Netherlands: design, and project goals*, European Journal of Human Genetics **22** (2014).

[4] P.J.A. Cock, C.J. Fields, N. Goto, M.L. Heuer, and P.M. Rice, *The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants*, Nucleic Acids Research **38** (2010).

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, 3 ed., MIT press, 2009.

[6] N. Goodman, S. Rozen, and L. Stein, *A Glimpse at the DBMS Challenges Posed by the Human Genome Project*, 1994.

[7] M. Ivanova, M. Kersten, and S. Manegold, *Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories*, Lecture Notes in Computer Science **7338** (2012).

[8] M. Ivanova, M. Kersten, S. Manegold, and Y. Kargın, *Data Vaults: Database Technology for Scientific File Repositories*, Computing in Science & Engineering **15** (2013).

[9] H.V. Jagadish and F. Olken, *Database management for life sciences research*, SIGMOD Rec. **33** (2004).

[10] Y. Kargın, *Turning scientists into data explorers*, SIGMOD'13 PhD Symposium, 2013.

[11] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup, *The Sequence Alignment/Map format and SAMtools*, Bioinformatics **25** (2009).

[12] T. Marschall, I.G. Costa, S. Canzar, M. Bauer, G.W. Klau, A. Schliep, and A. Schönhuth, *CLEVER: clique-enumerating variant finder*, Bioinformatics **28** (2012).

[13] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko, *Hadoop-BAM: directly manipulating next generation sequencing data in the cloud*, Bioinformatics **28** (2012).

[14] S. Wandelt, A. Rheinländer, M. Bux, L. Thalheim, B. Haldemann, and U. Leser, *Data Management Challenges in Next Generation Sequencing*, Datenbank-Spektrum **12** (2012).

[15] J.D. Watson, *The human genome project: past, present, and future*, Science **248** (1990).

Appendix A

Pseudocode for functionality over alignments

Algorithm 4: Calculate the reverse complement of a sequence string. Simplified version, since in practice more characters can occur, which happens if, due to uncertainty in the sequencing process, multiple characters are feasible to be in a certain position. Code assumes that a String can be accessed as a regular zero-based array.

```
1  ReverseSequence(String seq) begin
2  |   rev ← empty string of size Length(seq)
3  |   for i in [0, length(seq)) do
4  |   |   switch seq[i] do
5  |   |   |   case 'A' rev[Length(seq)−i − 1] = 'T'
6  |   |   |   case 'T' rev[Length(seq)−i − 1] = 'A'
7  |   |   |   case 'C' rev[Length(seq)−i − 1] = 'G'
8  |   |   |   case 'G' rev[Length(seq)−i − 1] = 'C'
9  |   |   endsw
10 |   end
11 |   return rev
12 end
```

Algorithm 5: Compute the reverse of a quality string.

Input : Quality string *qual*
Output : The reverse of *qual*

1 `ReverseQual(`*String qual*`)` **begin**
2     $rev \leftarrow$ empty string
3   **for** i *in* $[0, length(qual))$ **do**
4         $rev[$`Length(`seq`)`$-i-1] = qual[i]$
5   Prepend c to rev
6   **end**
7   **return** *rev*
8 **end**

Algorithm 6: Calculate the actual length of the sequence string of an alignment, which is defined as the length of the piece of the reference string it is mapped to.

1 `SequenceLength(`*Alignment a*`)` **begin**
2     $count \leftarrow 0$
3   **foreach** *tuple (count i, character c) in a.CIGAR* **do**
4   **if** $c \in$ 'M','D','N',' $='$,'X' **then**
5             $count = count + i$
6   **end**
7   **end**
8   **return** *count*
9 **end**
10

Algorithm 7: Calculate the distance between two alignments.

1 `Distance(`*Alignments* a_1, a_2`)` **begin**
2   **if** $a_1.POS \geq a_2.POS$ **then**
3   Swap a_1 and a_2
4   **end**
5   **return** $a_2.POS - (a_1.POS+$`SequenceLength(`a_1`)`$)$
6 **end**

Algorithm 8: Return *True* if *position* lies in the internal segment of alignments a_1 and a_2.

1 `InInternalSegment(`*Alignments* a_1, a_2, *position*`)` **begin**
2   **if** $a1.POS \geq a2.POS$ **then**
3   Swap a_1 and a_2
4   **end**
5   **return** $(a_1.POS+$`SequenceLength(`$a_1.CIGAR$`)`$\leq position < a_2.POS)$
6 **end**

Appendix B

SQL code for initialization of DBMS implementations

B.1 Schema to store BAM header data

```
CREATE SCHEMA bam;

CREATE TABLE "bam"."files" (
    "file_id"           SMALLINT    NOT NULL,
    "file_location"     STRING      NOT NULL UNIQUE,
    "format_version"    VARCHAR(7),
    "sorting_order"     VARCHAR(10),
    "comments"          STRING,
    CONSTRAINT "files_pkey_file_id"
               PRIMARY KEY (file_id)
);

CREATE TABLE "bam"."sq" (
    "sn"                STRING      NOT NULL,
    "file_id"           SMALLINT    NOT NULL,
    "ln"                INT         NOT NULL,
    "as"                INT,
    "m5"                STRING,
    "sp"                STRING,
    "ur"                STRING,
    CONSTRAINT "sq_pkey_sn_file_id"
               PRIMARY KEY (sn, file_id),
    CONSTRAINT "sq_fkey_file_id"
               FOREIGN KEY (file_id)
               REFERENCES bam.files (file_id)
);

CREATE TABLE "bam"."rg" (
    "id"                 STRING      NOT NULL,
    "file_id"            SMALLINT    NOT NULL,
```

```
    "cn"                 STRING,
    "ds"                 STRING,
    "dt"                 TIMESTAMP,
    "fo"                 STRING,
    "ks"                 STRING,
    "lb"                 STRING,
    "pg"                 STRING,
    "pi"                 INT,
    "pl"                 STRING,
    "pu"                 STRING,
    "sm"                 STRING,
    CONSTRAINT "rg_pkey_id_file_id"
               PRIMARY KEY (id, file_id),
    CONSTRAINT "rg_fkey_file_id"
               FOREIGN KEY (file_id)
               REFERENCES bam.files (file_id)
);

CREATE TABLE "bam"."pg" (
    "id"                 STRING      NOT NULL,
    "file_id"            SMALLINT    NOT NULL,
    "pn"                 STRING,
    "cl"                 STRING,
    "pp"                 STRING,
    "vn"                 STRING,
    CONSTRAINT "pg_pkey_id_file_id"
               PRIMARY KEY (id, file_id),
    CONSTRAINT "pg_fkey_file_id"
               FOREIGN KEY (file_id)
               REFERENCES bam.files (file_id)
);
```

B.2 Schema to store alignment data of BAM file with file ID i

```
CREATE TABLE bam.alignments_i (
    virtual_offset       BIGINT      NOT NULL,
    qname                STRING      NOT NULL,
    flag                 SMALLINT    NOT NULL,
    rname                STRING      NOT NULL,
    pos                  INT         NOT NULL,
    mapq                 SMALLINT    NOT NULL,
    cigar                STRING      NOT NULL,
    rnext                STRING      NOT NULL,
    pnext                INT         NOT NULL,
    tlen                 INT         NOT NULL,
```

```
    seq                 STRING      NOT NULL,
    qual                STRING      NOT NULL,
    CONSTRAINT alignments_i_pkey_virtual_offset
               PRIMARY KEY (virtual_offset)
);

CREATE TABLE bam.alignments_extra_i (
    tag                 CHAR(2)     NOT NULL,
    virtual_offset      BIGINT      NOT NULL,
    type                CHAR(1)     NOT NULL,
    value               STRING,
    CONSTRAINT alignments_extra_i_pkey_tag_virtual_offset
               PRIMARY KEY (tag, virtual_offset),
    CONSTRAINT alignments_extra_i_fkey_virtual_offset
               FOREIGN KEY (virtual_offset)
               REFERENCES bam.alignments_i (virtual_offset)
);
```

B.3 Schema to store alignment pairs of BAM file with file ID i

```
CREATE TABLE bam.paired_primary_alignments_i (
    l_virtual_offset    BIGINT      NOT NULL,
    r_virtual_offset    BIGINT      NOT NULL,
    qname               STRING      NOT NULL,
    l_flag              SMALLINT    NOT NULL,
    l_rname             STRING      NOT NULL,
    l_pos               INT         NOT NULL,
    l_mapq              SMALLINT    NOT NULL,
    l_cigar             STRING      NOT NULL,
    l_rnext             STRING      NOT NULL,
    l_pnext             INT         NOT NULL,
    l_tlen              INT         NOT NULL,
    l_seq               STRING      NOT NULL,
    l_qual              STRING      NOT NULL,
    r_flag              SMALLINT    NOT NULL,
    r_rname             STRING      NOT NULL,
    r_pos               INT         NOT NULL,
    r_mapq              SMALLINT    NOT NULL,
    r_cigar             STRING      NOT NULL,
    r_rnext             STRING      NOT NULL,
    r_pnext             INT         NOT NULL,
    r_tlen              INT         NOT NULL,
    r_seq               STRING      NOT NULL,
    r_qual              STRING      NOT NULL,
    CONSTRAINT paired_primary_alignments_i_pkey_l_vo_r_vo
```

```
            PRIMARY KEY (l_virtual_offset , r_virtual_offset)
);

CREATE TABLE bam.paired_secondary_alignments_i (
    l_virtual_offset    BIGINT      NOT NULL,
    r_virtual_offset    BIGINT      NOT NULL,
    qname               STRING      NOT NULL,
    l_flag              SMALLINT    NOT NULL,
    l_rname             STRING      NOT NULL,
    l_pos               INT         NOT NULL,
    l_mapq              SMALLINT    NOT NULL,
    l_cigar             STRING      NOT NULL,
    l_rnext             STRING      NOT NULL,
    l_pnext             INT         NOT NULL,
    l_tlen              INT         NOT NULL,
    l_seq               STRING      NOT NULL,
    l_qual              STRING      NOT NULL,
    r_flag              SMALLINT    NOT NULL,
    r_rname             STRING      NOT NULL,
    r_pos               INT         NOT NULL,
    r_mapq              SMALLINT    NOT NULL,
    r_cigar             STRING      NOT NULL,
    r_rnext             STRING      NOT NULL,
    r_pnext             INT         NOT NULL,
    r_tlen              INT         NOT NULL,
    r_seq               STRING      NOT NULL,
    r_qual              STRING      NOT NULL,
    CONSTRAINT paired_secondary_alignments_i_pkey_l_vo_r_vo
            PRIMARY KEY (l_virtual_offset , r_virtual_offset)
);

CREATE TABLE bam.unpaired_alignments_i (
    virtual_offset      BIGINT      NOT NULL,
    qname               STRING      NOT NULL,
    flag                SMALLINT    NOT NULL,
    rname               STRING      NOT NULL,
    pos                 INT         NOT NULL,
    mapq                SMALLINT    NOT NULL,
    cigar               STRING      NOT NULL,
    rnext               STRING      NOT NULL,
    pnext               INT         NOT NULL,
    tlen                INT         NOT NULL,
    seq                 STRING      NOT NULL,
    qual                STRING      NOT NULL,
    CONSTRAINT unpaired_alignments_i_pkey_vo
            PRIMARY KEY (virtual_offset)
);

CREATE TABLE bam.alignments_extra_i (
```

```
    tag                 CHAR(2)     NOT NULL,
    virtual_offset      BIGINT      NOT NULL,
    type                CHAR(1)     NOT NULL,
    value               STRING,
    CONSTRAINT alignments_extra_i_pkey_tag_vo
               PRIMARY KEY (tag, virtual_offset)
);

CREATE VIEW bam.unpaired_primary_alignments_i AS
    SELECT l_virtual_offset AS virtual_offset, qname,
           l_flag AS flag, l_rname AS rname, l_pos AS pos,
           l_mapq AS mapq, l_cigar AS cigar, l_rnext AS rnext,
           l_pnext AS pnext, l_tlen AS tlen, l_seq AS seq,
           l_qual AS qual
    FROM bam.paired_primary_alignments_i
    UNION ALL
    SELECT r_virtual_offset AS virtual_offset, qname,
           r_flag AS flag, r_rname AS rname, r_pos AS pos,
           r_mapq AS mapq, r_cigar AS cigar, r_rnext AS rnext,
           r_pnext AS pnext, r_tlen AS tlen, r_seq AS seq,
           r_qual AS qual
    FROM bam.paired_primary_alignments_i;

CREATE VIEW bam.unpaired_secondary_alignments_i AS
    SELECT l_virtual_offset AS virtual_offset, qname,
           l_flag AS flag, l_rname AS rname, l_pos AS pos,
           l_mapq AS mapq, l_cigar AS cigar, l_rnext AS rnext,
           l_pnext AS pnext, l_tlen AS tlen, l_seq AS seq,
           l_qual AS qual
    FROM bam.paired_secondary_alignments_i
    UNION ALL
    SELECT r_virtual_offset AS virtual_offset, qname,
           r_flag AS flag, r_rname AS rname, r_pos AS pos,
           r_mapq AS mapq, r_cigar AS cigar, r_rnext AS rnext,
           r_pnext AS pnext, r_tlen AS tlen, r_seq AS seq,
           r_qual AS qual
    FROM bam.paired_secondary_alignments_i;

CREATE VIEW bam.unpaired_all_alignments_i AS
    SELECT *
    FROM bam.unpaired_primary_alignments_i
    UNION ALL
    SELECT *
    FROM bam.unpaired_secondary_alignments_i
    UNION ALL
    SELECT *
    FROM bam.unpaired_alignments_i;
```

B.4 SQL code for dividing alignments over the paired schema

```
INSERT INTO bam.paired_primary_alignments_i (
    SELECT l.virtual_offset, r.virtual_offset, l.qname, l.flag, l.rname,
           l.pos, l.mapq, l.cigar, l.rnext, l.pnext, l.tlen, l.seq, l.qual,
                                                         r.flag, r.rname,
           r.pos, r.mapq, r.cigar, r.rnext, r.pnext, r.tlen, r.seq, r.qual
    FROM (
        SELECT *
        FROM bam.unpaired_alignments_i
        WHERE bam_flag(flag, 'firs_segm') <> bam_flag(flag, 'last_segm')
          AND bam_flag(flag, 'seco_alig') = False
          AND bam_flag(flag, 'firs_segm') = True
          AND qname IN (
            SELECT qname
            FROM bam.unpaired_alignments_i
            WHERE bam_flag(flag, 'firs_segm') <>
                  bam_flag(flag, 'last_segm')
              AND bam_flag(flag, 'seco_alig') = False
            GROUP BY qname
            HAVING COUNT(*) = 2
               AND SUM(bam_flag(flag, 'firs_segm')) = 1
               AND SUM(bam_flag(flag, 'last_segm')) = 1
          )
    ) AS l JOIN (
        SELECT *
        FROM bam.unpaired_alignments_i
        WHERE bam_flag(flag, 'firs_segm') <> bam_flag(flag, 'last_segm')
          AND bam_flag(flag, 'seco_alig') = False
          AND bam_flag(flag, 'last_segm') = True
          AND qname IN (
            SELECT qname
            FROM bam.unpaired_alignments_i
            WHERE bam_flag(flag, 'firs_segm') <>
                  bam_flag(flag, 'last_segm')
              AND bam_flag(flag, 'seco_alig') = False
            GROUP BY qname
            HAVING COUNT(*) = 2
               AND SUM(bam_flag(flag, 'firs_segm')) = 1
               AND SUM(bam_flag(flag, 'last_segm')) = 1
          )
    ) AS r
        ON l.qname = r.qname
);

DELETE FROM bam.unpaired_alignments_i
WHERE virtual_offset IN (
```

```
    SELECT l_virtual_offset
    FROM bam.paired_primary_alignments_i
) OR virtual_offset IN (
    SELECT r_virtual_offset
    FROM bam.paired_primary_alignments_i
);

INSERT INTO bam.paired_secondary_alignments_i (
    SELECT l.virtual_offset, r.virtual_offset, l.qname, l.flag, l.rname,
           l.pos, l.mapq, l.cigar, l.rnext, l.pnext, l.tlen, l.seq, l.qual,
                                                          r.flag, r.rname,
           r.pos, r.mapq, r.cigar, r.rnext, r.pnext, r.tlen, r.seq, r.qual
    FROM (
        SELECT *
        FROM bam.unpaired_alignments_i
        WHERE bam_flag(flag, 'firs_segm') <> bam_flag(flag, 'last_segm')
          AND bam_flag(flag, 'seco_alig') = True
          AND rname <> '*'
          AND pos > 0
          AND rnext <> '*'
          AND pnext > 0
          AND bam_flag(flag, 'firs_segm') = True
    ) AS l JOIN (
        SELECT *
        FROM bam.unpaired_alignments_i
        WHERE bam_flag(flag, 'firs_segm') <> bam_flag(flag, 'last_segm')
          AND bam_flag(flag, 'seco_alig') = True
          AND rname <> '*'
          AND pos > 0
          AND rnext <> '*'
          AND pnext > 0
          AND bam_flag(flag, 'last_segm') = True
    ) AS r
        ON l.qname = r.qname
       AND ((l.rnext = '=' AND l.rname = r.rname) OR l.rnext = r.rname)
       AND l.pnext = r.pos
       AND ((r.rnext = '=' AND l.rname = r.rname) OR r.rnext = l.rname)
       AND r.pnext = l.pos
);

DELETE FROM bam.unpaired_alignments_i
WHERE virtual_offset IN (
    SELECT l_virtual_offset
    FROM bam.paired_secondary_alignments_i
) OR virtual_offset IN (
    SELECT r_virtual_offset
    FROM bam.paired_secondary_alignments_i
);
```

Appendix C

SQL queries for the straightforward storage schema

Use case 1.1

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE bam_flag(flag, 'seco_alig') = False
ORDER BY qname;
```

Use case 1.2

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE bam_flag(flag, 'seco_alig') = False
ORDER BY rname, pos;
```

Use case 1.3

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE rname = rname_1_3
  AND pos >= pos_1_3_1
  AND pos <= pos_1_3_2
ORDER BY pos;
```

Use case 1.4

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE qname = qname_1_4
ORDER BY rname, pos;
```

Use case 1.5

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE mapq > mapq_1_5
ORDER BY mapq DESC;
```

Use case 2.1

```
WITH alig AS (
    SELECT qname, flag, seq, qual
    FROM bam.alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = False
      AND mapq < mapq_2_1
), alig_proj AS (
    SELECT qname, flag,
        CASE WHEN bam_flag(flag, 'segm_reve')
             THEN reverse_seq(seq)
             ELSE seq  END AS seq,
        CASE WHEN bam_flag(flag, 'segm_reve')
             THEN reverse_qual(qual)
             ELSE qual END AS qual
    FROM alig
    WHERE qname IN (
        SELECT qname
        FROM alig
        GROUP BY qname
        HAVING COUNT(*) = 2
           AND SUM(bam_flag(flag, 'firs_segm')) = 1
           AND SUM(bam_flag(flag, 'last_segm')) = 1
        )
)
SELECT l.qname, l.seq, l.qual, r.seq, r.qual
FROM (
    SELECT *
    FROM alig_proj
    WHERE bam_flag(flag, 'firs_segm') = True
) AS l JOIN (
    SELECT *
    FROM alig_proj
    WHERE bam_flag(flag, 'last_segm') = True
) AS r
    ON l.qname = r.qname
ORDER BY qname;
```

Use case 2.2

```
WITH alig AS (
    SELECT qname, flag, rname, pos, cigar
    FROM bam.alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = False
      AND qname IN (
        SELECT qname
        FROM bam.alignments_i
        WHERE bam_flag(flag, 'firs_segm') <>
              bam_flag(flag, 'last_segm')
          AND bam_flag(flag, 'seco_alig') = False
        GROUP BY qname
        HAVING COUNT(*) = 2
           AND SUM(bam_flag(flag, 'firs_segm')) = 1
           AND SUM(bam_flag(flag, 'last_segm')) = 1
     )
)
SELECT
    CASE WHEN l.pos < r.pos
         THEN r.pos - (l.pos + seq_length(l.cigar))
         ELSE l.pos - (r.pos + seq_length(r.cigar))
    END AS distance,
    COUNT(*) AS nr_alignments
FROM (
    SELECT qname, rname, pos, cigar
    FROM alig
    WHERE bam_flag(flag, 'firs_segm') = True
) AS l JOIN (
    SELECT qname, rname, pos, cigar
    FROM alig
    WHERE bam_flag(flag, 'last_segm') = True
) AS r
    ON l.qname = r.qname
   AND l.rname = r.rname
GROUP BY distance
ORDER BY nr_alignments DESC;
```

Use case 2.3

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE qname IN (
    SELECT qname
    FROM bam.alignments_i
    GROUP BY qname
    HAVING SUM(bam_flag(flag, 'firs_segm')) = 0
        OR SUM(bam_flag(flag, 'last_segm')) = 0
```

```
)
ORDER BY qname;
```

Use case 2.4

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE qname IN (
    SELECT qname
    FROM (
        SELECT qname, bam_flag(flag, 'seco_alig') AS seco_alig,
               bam_flag(flag, 'segm_unma') AS segm_unma,
               bam_flag(flag, 'firs_segm') AS firs_segm
        FROM bam.alignments_i
        WHERE bam_flag(flag, 'firs_segm') <>
              bam_flag(flag, 'last_segm')
    ) AS qnames
    GROUP BY qname, firs_segm
    HAVING SUM(segm_unma) < COUNT(*)
       AND (COUNT(*) - SUM(seco_alig)) <> 1
)
ORDER BY qname;
```

Use case 2.5

```
WITH qnames1 AS (
    SELECT DISTINCT qname
    FROM bam.alignments_i
), qnames2 AS (
    SELECT DISTINCT qname
    FROM bam.alignments_j
)
SELECT count_a_insct_b,
       count_qnames1 - count_a_insct_b AS count_a_minus_b,
       count_qnames2 - count_a_insct_b AS count_b_minus_a
FROM (
    SELECT COUNT(*) AS count_a_insct_b
    FROM (
        SELECT *
        FROM qnames1
        INTERSECT
        SELECT *
        FROM qnames2
    ) AS a_insct_b
) AS insct_sub, (
    SELECT COUNT(*) AS count_qnames1
    FROM qnames1
) AS qnames1_sub, (
```

```
    SELECT COUNT(*) AS count_qnames2
    FROM qnames2
) AS qnames2_sub;
```

Use case 2.6

```
WITH qnames_insct AS (
    SELECT distinct qname
    FROM bam.alignments_i
    INTERSECT
    SELECT distinct qname
    FROM bam.alignments_j
)
SELECT 'f1', qname, flag, rname, pos, mapq, cigar, rnext, pnext,
       tlen, seq, qual
FROM bam.alignments_i
WHERE qname IN (
    SELECT *
    FROM qnames_insct
)
UNION
SELECT 'f2', qname, flag, rname, pos, mapq, cigar, rnext, pnext,
       tlen, seq, qual
FROM bam.alignments_j
WHERE qname IN (
    SELECT *
    FROM qnames_insct
)
ORDER BY qname;
```

Use case 2.7

```
SELECT 'f1', qname, flag, rname, pos, mapq, cigar, rnext, pnext,
       tlen, seq, qual
FROM bam.alignments_i
WHERE qname IN (
    SELECT distinct qname
    FROM bam.alignments_i
    EXCEPT
    SELECT distinct qname
    FROM bam.alignments_j
)
ORDER BY qname;
```

Use case 2.8

```
SELECT f1.qname, f1.flag, f1.rname, f1.pos, f1.mapq, f1.cigar,
       f1.rnext, f1.pnext, f1.tlen, f1.seq, f1.qual,
                 f2.flag, f2.rname, f2.pos, f2.mapq, f2.cigar,
```

```
       f2.rnext, f2.pnext, f2.tlen, f2.seq, f2.qual
FROM (
    SELECT *
    FROM bam.alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = False
) AS f1 JOIN (
    SELECT *
    FROM bam.alignments_j
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = False
) AS f2
    ON  f1.qname = f2.qname
    AND bam_flag(f1.flag, 'firs_segm') =
        bam_flag(f2.flag, 'firs_segm')
    AND (f1.rname <> f2.rname OR f1.pos <> f2.pos)
ORDER BY qname;
```

Use case 2.9

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.alignments_i
WHERE rname = rname_2_9
  AND pos_2_9 >= pos
  AND pos_2_9 < pos + seq_length(cigar)
ORDER BY pos;
```

Use case 2.10

```
WITH alig AS (
    SELECT *
    FROM bam.alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND rname = rname_2_10
      AND bam_flag(flag, 'seco_alig') = False
      AND qname IN (
        SELECT qname
        FROM bam.alignments_i
        WHERE bam_flag(flag, 'firs_segm') <>
              bam_flag(flag, 'last_segm')
          AND bam_flag(flag, 'seco_alig') = False
        GROUP BY qname
        HAVING COUNT(*) = 2
           AND SUM(bam_flag(flag, 'firs_segm')) = 1
           AND SUM(bam_flag(flag, 'last_segm')) = 1
```

```
    )
)
SELECT l.qname, l.flag, l.rname, l.pos, l.mapq, l.cigar,
       l.rnext, l.pnext, l.tlen, l.seq, l.qual,
                r.flag, r.rname, r.pos, r.mapq, r.cigar,
       r.rnext, r.pnext, r.tlen, r.seq, r.qual
FROM (
    SELECT *
    FROM alig
    WHERE bam_flag(flag, 'firs_segm') = True
) AS l JOIN (
    SELECT *
    FROM alig
    WHERE bam_flag(flag, 'last_segm') = True
) AS r ON l.qname = r.qname
       AND CASE WHEN l.pos < r.pos
                THEN (pos_2_10 >= l.pos + seq_length(l.cigar)
                      AND pos_2_10 < r.pos)
                ELSE (pos_2_10 >= r.pos + seq_length(r.cigar)
                      AND pos_2_10 < l.pos)
           END
ORDER BY l_pos;
```

Use case 2.11

```
WITH alig AS (
    SELECT *
    FROM bam.alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = True
      AND rname <> '*'
      AND pos > 0
      AND rnext <> '*'
      AND pnext > 0
)
SELECT l.qname, l.flag, l.rname, l.pos, l.mapq, l.cigar,
       l.rnext, l.pnext, l.tlen, l.seq, l.qual,
                r.flag, r.rname, r.pos, r.mapq, r.cigar,
       r.rnext, r.pnext, r.tlen, r.seq, r.qual
FROM (
    SELECT *
    FROM alig
    WHERE bam_flag(flag, 'firs_segm') = True
) AS l JOIN (
    SELECT *
    FROM alig
    WHERE bam_flag(flag, 'last_segm') = True
) AS r
```

```
    ON l.qname = r.qname
   AND ((l.rnext = '=' AND l.rname = r.rname) OR
         l.rnext = r.rname)
   AND l.pnext = r.pos
   AND ((r.rnext = '=' AND l.rname = r.rname) OR
         r.rnext = l.rname)
   AND r.pnext = l.pos
ORDER BY l.qname;
```

Use case 2.12

```
WITH alig AS (
    SELECT *
    FROM bam.alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = True
      AND rname <> '*'
      AND pos > 0
      AND rnext = '*'
      AND pnext = 0
)
SELECT *
FROM (
    SELECT l.qname, l.rname, l.flag, l.pos, l.mapq, l.cigar,
           l.rnext, l.pnext, l.tlen, l.seq, l.qual,
                               r.flag, r.pos, r.mapq, r.cigar,
           r.rnext, r.pnext, r.tlen, r.seq, r.qual,
        CASE WHEN l.pos < r.pos
             THEN r.pos - (l.pos + seq_length(l.cigar))
             ELSE l.pos - (r.pos + seq_length(r.cigar))
        END AS distance
    FROM (
        SELECT *
        FROM alig
        WHERE bam_flag(flag, 'firs_segm') = True
    ) AS l JOIN (
        SELECT *
        FROM alig
        WHERE bam_flag(flag, 'last_segm') = True
    ) AS r
        ON  l.qname = r.qname
        AND l.rname = r.rname
) AS alig_joined
WHERE distance > 0
  AND distance < distance_2_12
ORDER BY rname;
```

Appendix D

SQL queries for the pairwise storage schema

Use case 1.1

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_primary_alignments_i
UNION
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_alignments_i
WHERE bam_flag(flag, 'seco_alig') = False
ORDER BY qname;
```

Use case 1.2

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_primary_alignments_i
UNION
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_alignments_i
WHERE bam_flag(flag, 'seco_alig') = False
ORDER BY rname, pos;
```

Use case 1.3

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_all_alignments_i
WHERE rname = rname_1_3
  AND pos >= pos_1_3_1
  AND pos <= pos_1_3_2
ORDER BY pos;
```

Use case 1.4

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_all_alignments_i
WHERE qname = qname_1_4
ORDER BY rname, pos;
```

Use case 1.5

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_all_alignments_i
WHERE mapq > mapq_1_5
ORDER BY mapq DESC;
```

Use case 2.1

```
SELECT qname,
    CASE WHEN bam_flag(l_flag, 'segm_reve')
         THEN reverse_seq(l_seq)
         ELSE l_seq  END AS l_seq,
    CASE WHEN bam_flag(l_flag, 'segm_reve')
         THEN reverse_qual(l_qual)
         ELSE l_qual END AS l_qual,
    CASE WHEN bam_flag(r_flag, 'segm_reve')
         THEN reverse_seq(r_seq)
         ELSE r_seq  END AS r_seq,
    CASE WHEN bam_flag(r_flag, 'segm_reve')
         THEN reverse_qual(r_qual)
         ELSE r_qual END AS r_qual
FROM bam.paired_primary_alignments_i
WHERE l_mapq < mapq_2_1
  AND r_mapq < mapq_2_1
ORDER BY qname;
```

Use case 2.2

```
SELECT
    CASE WHEN l_pos < r_pos
         THEN r_pos - (l_pos + seq_length(l_cigar))
         ELSE l_pos - (r_pos + seq_length(r_cigar))
    END AS distance,
    COUNT(*) AS nr_alignments
FROM bam.paired_primary_alignments_i
WHERE l_rname = r_rname
GROUP BY distance
ORDER BY nr_alignments DESC;
```

Use case 2.3

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_all_alignments_i
WHERE qname IN (
    SELECT qname
    FROM bam.unpaired_all_alignments_i
    GROUP BY qname
    HAVING SUM(bam_flag(flag, 'firs_segm')) = 0
        OR SUM(bam_flag(flag, 'last_segm')) = 0
)
ORDER BY qname;
```

Use case 2.4

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_all_alignments_i
WHERE qname IN (
    SELECT qname
    FROM (
        SELECT qname, bam_flag(flag, 'seco_alig') AS seco_alig,
               bam_flag(flag, 'segm_unma') AS segm_unma,
               bam_flag(flag, 'firs_segm') AS firs_segm
        FROM bam.unpaired_all_alignments_i
        WHERE bam_flag(flag, 'firs_segm') <>
              bam_flag(flag, 'last_segm')
    ) AS qnames
    GROUP BY qname, firs_segm
    HAVING SUM(segm_unma) < COUNT(*)
       AND (COUNT(*) - SUM(seco_alig)) <> 1
)
ORDER BY qname;
```

Use case 2.5

```
WITH qnames1 AS (
    SELECT DISTINCT qname
    FROM bam.unpaired_all_alignments_i
), qnames2 AS (
    SELECT DISTINCT qname
    FROM bam.unpaired_all_alignments_j
)
SELECT count_a_insct_b,
       count_qnames1 - count_a_insct_b AS count_a_minus_b,
       count_qnames2 - count_a_insct_b AS count_b_minus_a
FROM (
    SELECT COUNT(*) AS count_a_insct_b
```

```
    FROM (
        SELECT *
        FROM qnames1
        INTERSECT
        SELECT *
        FROM qnames2
    ) AS a_insct_b
) AS insct_sub, (
    SELECT COUNT(*) AS count_qnames1
    FROM qnames1
) AS qnames1_sub, (
    SELECT COUNT(*) AS count_qnames2
    FROM qnames2
) AS qnames2_sub;
```

Use case 2.6

```
WITH qnames_insct AS (
    SELECT distinct qname
    FROM bam.unpaired_all_alignments_i
    INTERSECT
    SELECT distinct qname
    FROM bam.unpaired_all_alignments_j
)
SELECT 'f1', qname, flag, rname, pos, mapq, cigar, rnext, pnext,
       tlen, seq, qual
FROM bam.unpaired_all_alignments_i
WHERE qname IN (
    SELECT *
    FROM qnames_insct
)
UNION
SELECT 'f2', qname, flag, rname, pos, mapq, cigar, rnext, pnext,
       tlen, seq, qual
FROM bam.unpaired_all_alignments_j
WHERE qname IN (
    SELECT *
    FROM qnames_insct
)
ORDER BY qname;
```

Use case 2.7

```
SELECT 'f1', qname, flag, rname, pos, mapq, cigar, rnext, pnext,
       tlen, seq, qual
FROM bam.unpaired_all_alignments_i
WHERE qname IN (
    SELECT DISTINCT qname
    FROM bam.unpaired_all_alignments_i
```

```
    EXCEPT
    SELECT DISTINCT qname
    FROM bam.unpaired_all_alignments_j
)
ORDER BY qname;
```

Use case 2.8

```
SELECT f1.qname, f1.flag, f1.rname, f1.pos, f1.mapq, f1.cigar,
       f1.rnext, f1.pnext, f1.tlen, f1.seq, f1.qual,
                 f2.flag, f2.rname, f2.pos, f2.mapq, f2.cigar,
       f2.rnext, f2.pnext, f2.tlen, f2.seq, f2.qual
FROM (
   SELECT *
   FROM bam.unpaired_primary_alignments_i
) AS f1 JOIN (
    SELECT *
   FROM bam.unpaired_primary_alignments_j
) AS f2
    ON  f1.qname = f2.qname
    AND bam_flag(f1.flag, 'firs_segm') =
        bam_flag(f2.flag, 'firs_segm')
    AND (f1.rname <> f2.rname OR f1.pos <> f2.pos)
ORDER BY qname;
```

Use case 2.9

```
SELECT qname, flag, rname, pos, mapq, cigar, rnext, pnext, tlen,
       seq, qual
FROM bam.unpaired_all_alignments_i
WHERE rname = rname_2_9
  AND pos_2_9 >= pos
  AND pos_2_9 < pos + seq_length(cigar)
ORDER BY pos;
```

Use case 2.10

```
SELECT qname, l_flag, l_rname, l_pos, l_mapq, l_cigar, l_rnext,
       l_pnext, l_tlen, l_seq, l_qual,
              r_flag, r_rname, r_pos, r_mapq, r_cigar, r_rnext,
       r_pnext, r_tlen, r_seq, r_qual
FROM bam.paired_primary_alignments_i
WHERE l_rname = rname_2_10
  AND r_rname = rname_2_10
  AND CASE WHEN l_pos < r_pos
           THEN (pos_2_10 >= l_pos + seq_length(l_cigar)
                 AND pos_2_10 < r_pos)
           ELSE (pos_2_10 >= r_pos + seq_length(r_cigar)
                 AND pos_2_10 < l_pos)
```

```
    END
ORDER BY l_pos
```

Use case 2.11

```
SELECT qname, l_flag, l_rname, l_pos, l_mapq, l_cigar, l_rnext,
       l_pnext, l_tlen, l_seq, l_qual,
               r_flag, r_rname, r_pos, r_mapq, r_cigar, r_rnext,
       r_pnext, r_tlen, r_seq, r_qual
FROM bam.paired_secondary_alignments_i
ORDER BY qname;
```

Use case 2.12

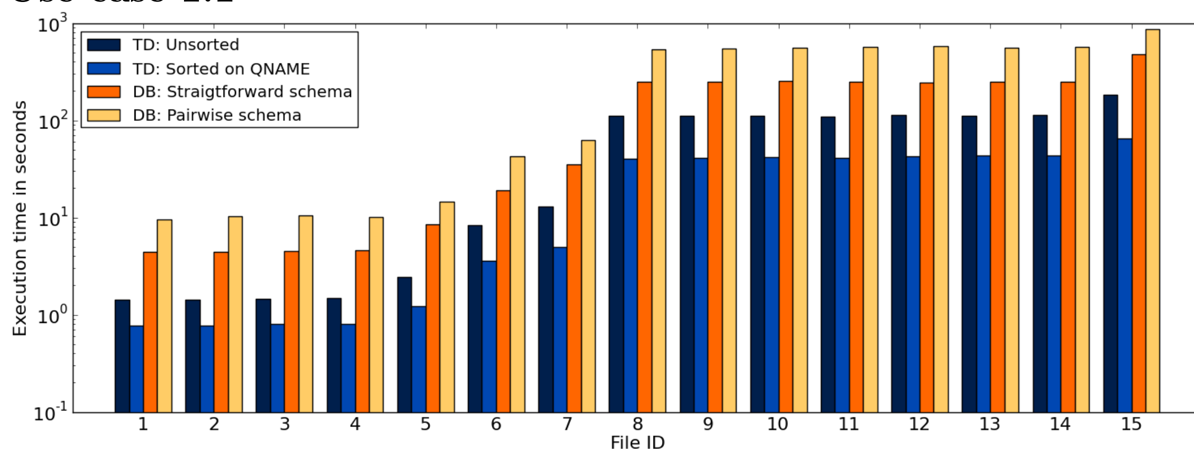
```
WITH alig AS (
    SELECT *
    FROM bam.unpaired_alignments_i
    WHERE bam_flag(flag, 'firs_segm') <>
          bam_flag(flag, 'last_segm')
      AND bam_flag(flag, 'seco_alig') = True
      AND rname <> '*'
      AND pos > 0
      AND rnext = '*'
      AND pnext = 0
)
SELECT *
FROM (
    SELECT l.qname, l.rname, l.flag, l.pos, l.mapq, l.cigar,
           l.rnext, l.pnext, l.tlen, l.seq, l.qual,
                             r.flag, r.pos, r.mapq, r.cigar,
           r.rnext, r.pnext, r.tlen, r.seq, r.qual,
        CASE WHEN l.pos < r.pos
             THEN r.pos - (l.pos + seq_length(l.cigar))
             ELSE l.pos - (r.pos + seq_length(r.cigar))
        END AS distance
    FROM (
        SELECT *
        FROM alig
        WHERE bam_flag(flag, 'firs_segm') = True
    ) AS l JOIN (
        SELECT *
        FROM alig
        WHERE bam_flag(flag, 'last_segm') = True
    ) AS r
        ON  l.qname = r.qname
        AND l.rname = r.rname
) AS alig_joined
WHERE distance > 0 AND distance < distance_2_12
ORDER BY rname;
```

Appendix E

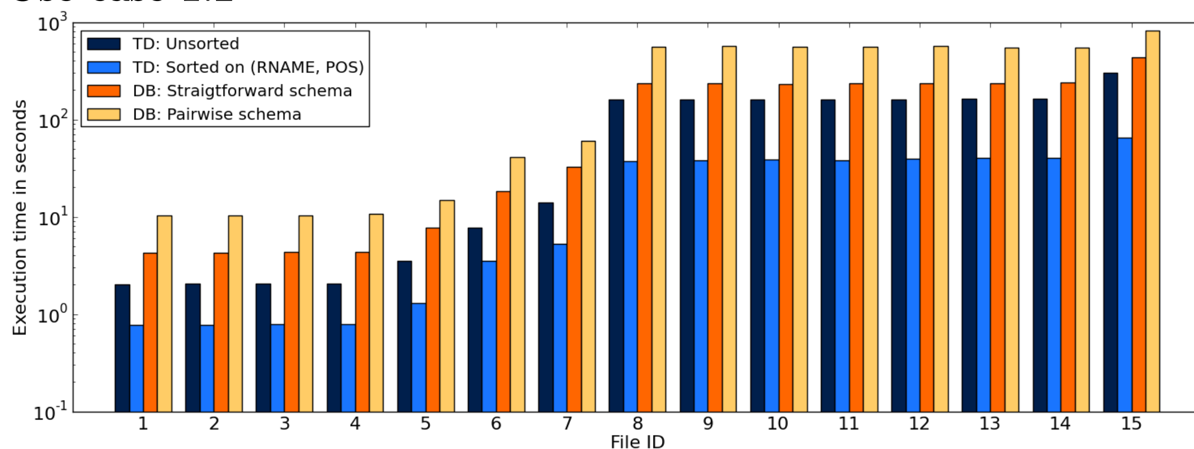
Plots showing implementation performance

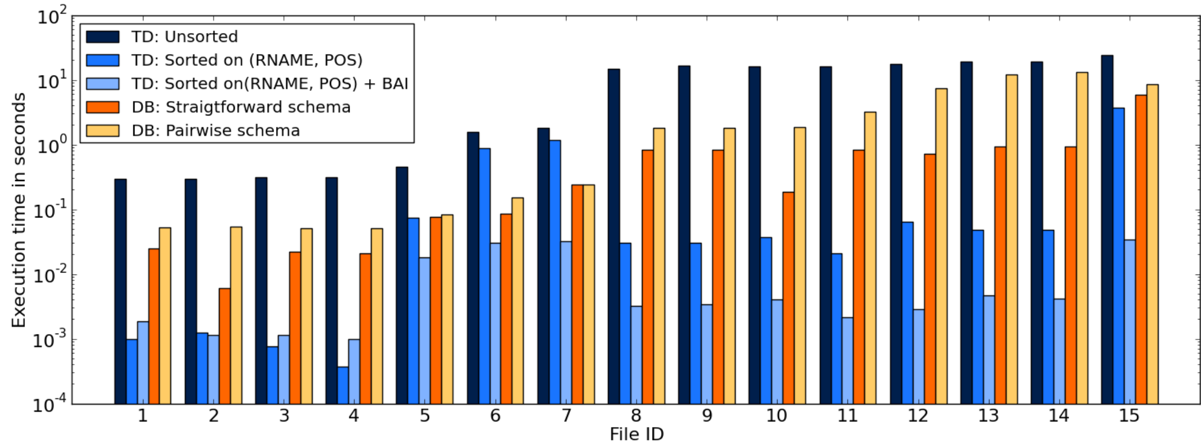
E.1 Small files – full-output approach

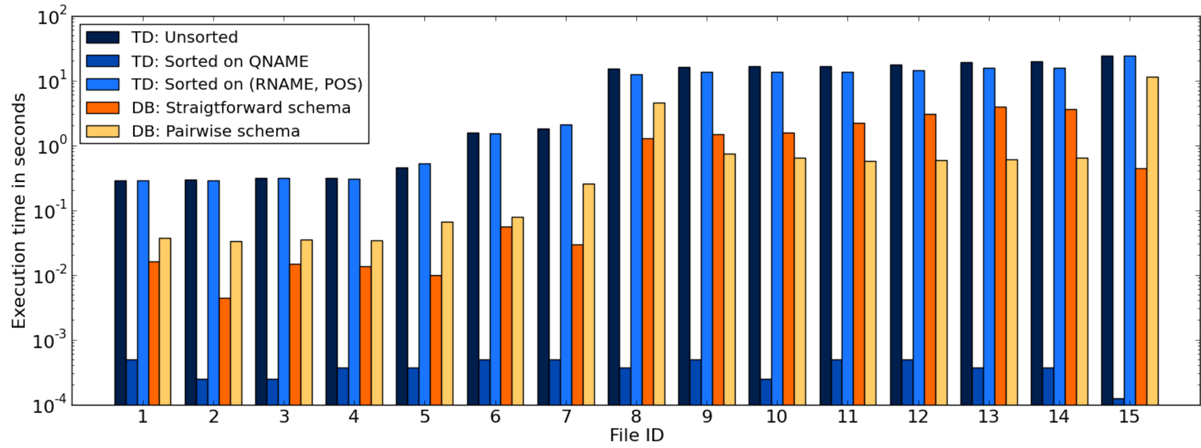
Use case 1.1

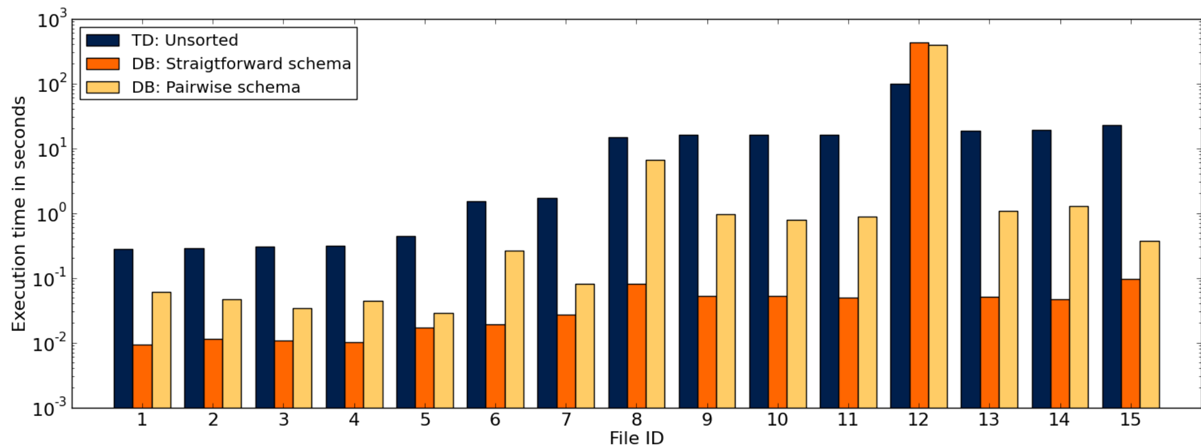


Use case 1.2

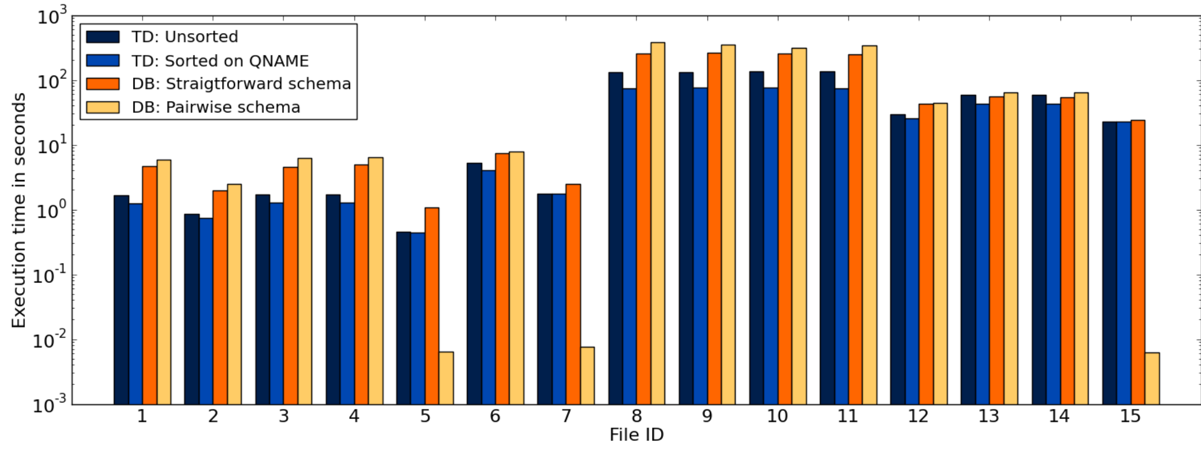



Use case 1.3
TD: Unsorted
TD: Sorted on (RNAME, POS)
TD: Sorted on(RNAME, POS) + BAI
DB: Straigtforward schema
DB: Pairwise schema
Execution time in seconds
File ID

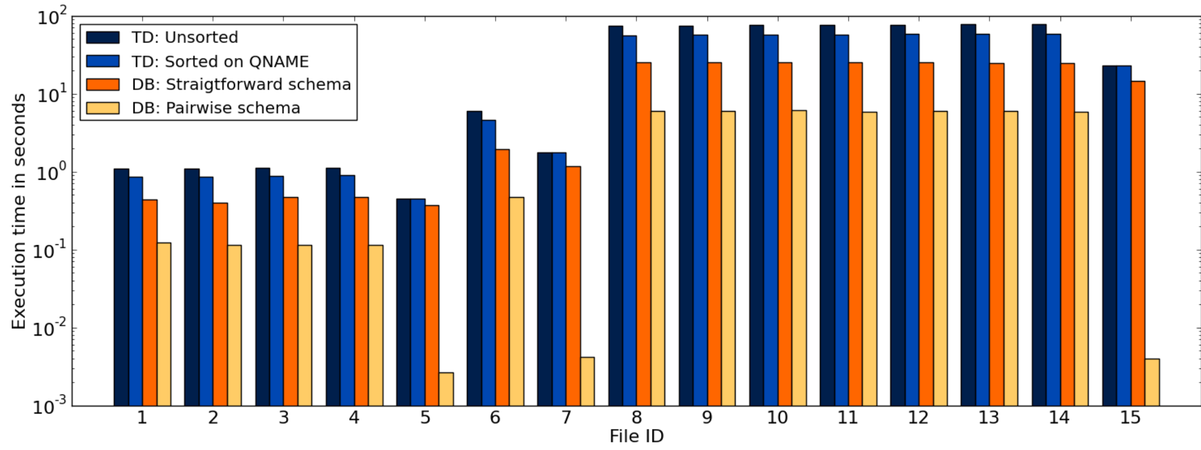

Use case 1.4
TD: Unsorted
TD: Sorted on QNAME
TD: Sorted on (RNAME, POS)
DB: Straigtforward schema
DB: Pairwise schema
Execution time in seconds
File ID


Use case 1.5
TD: Unsorted
DB: Straigtforward schema
DB: Pairwise schema
Execution time in seconds
File ID

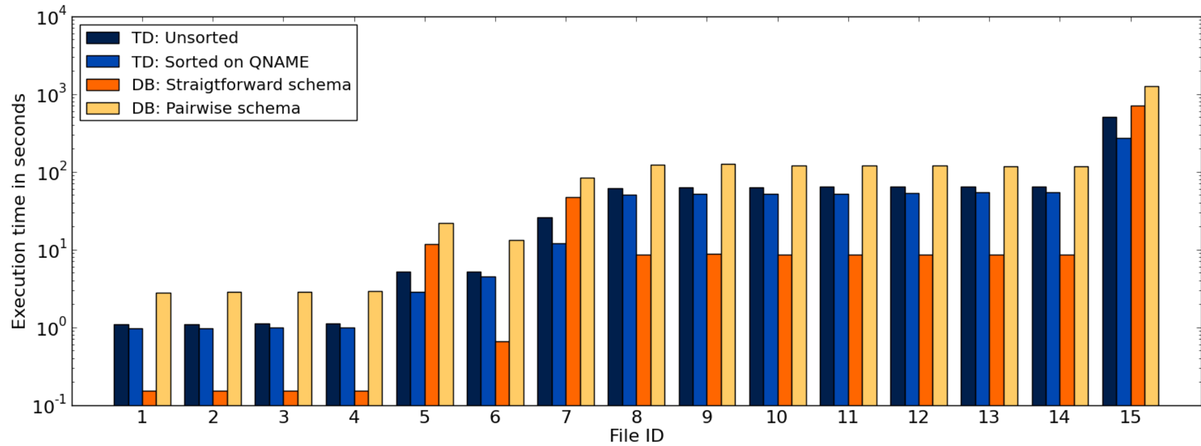
Use case 2.1



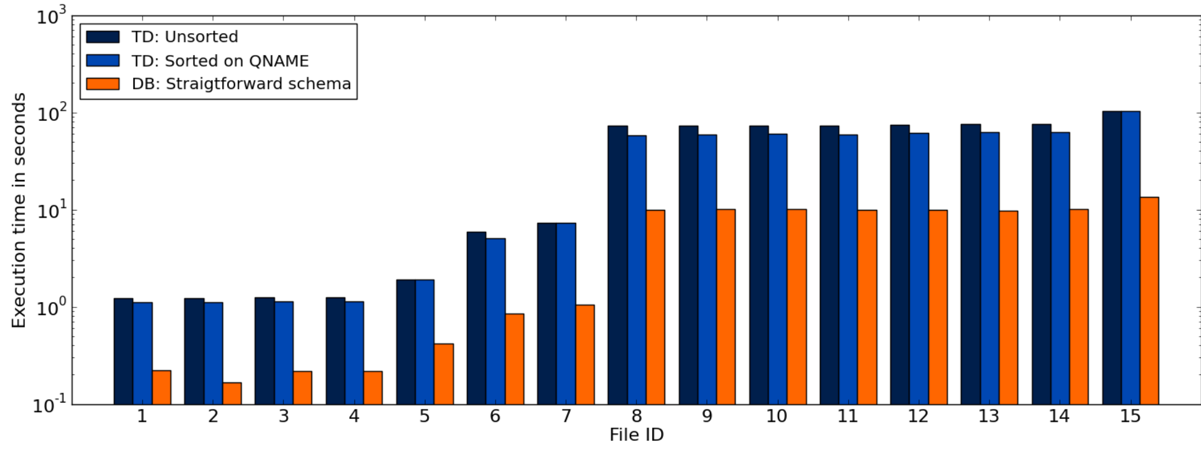
Use case 2.2



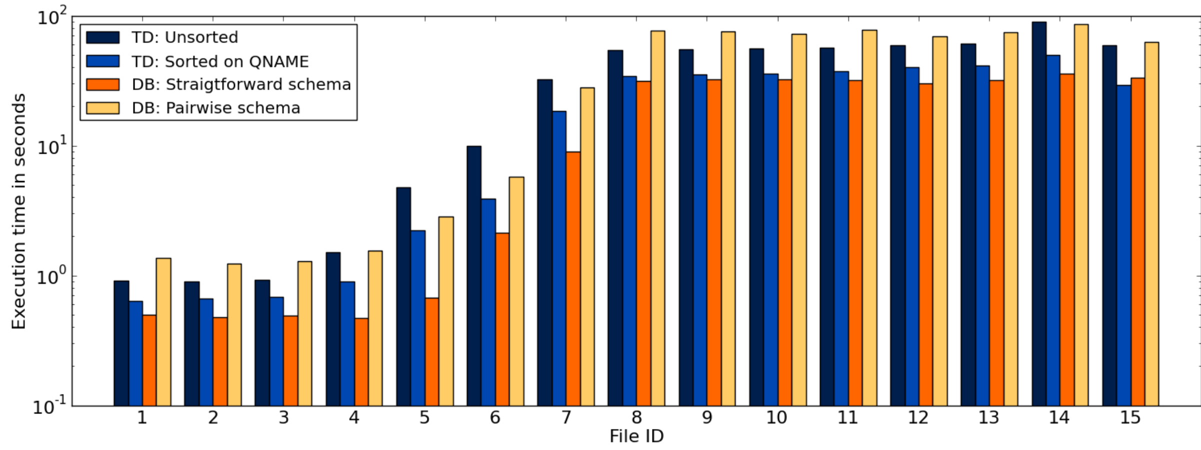
Use case 2.3



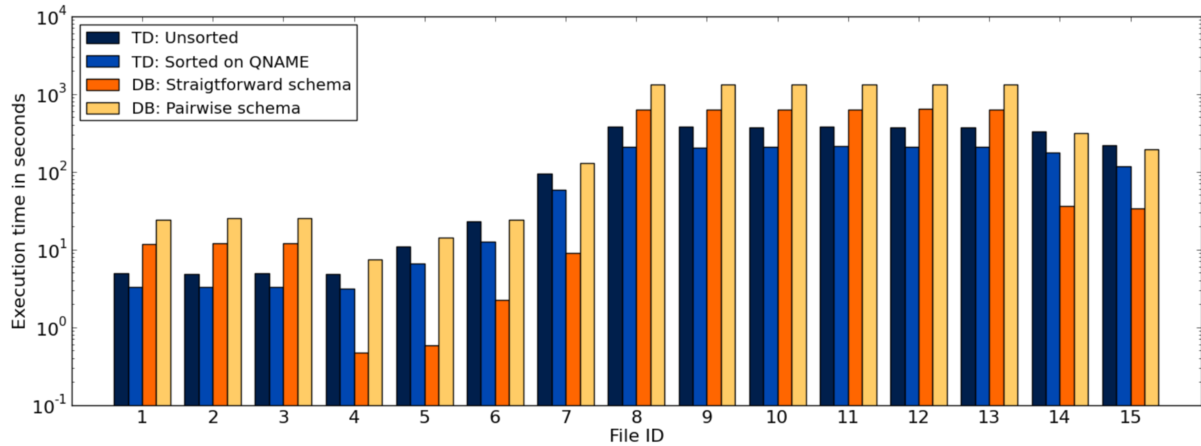
Use case 2.4



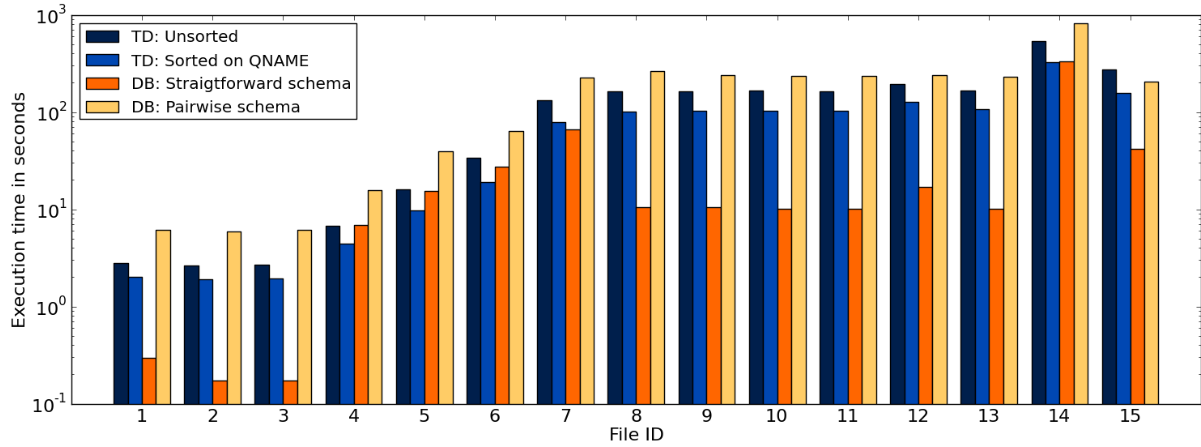
Use case 2.5



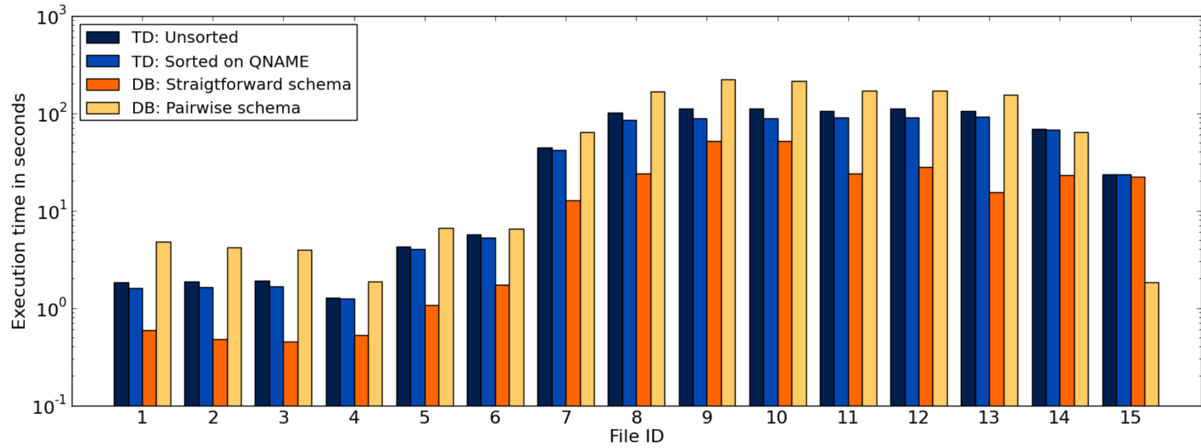
Use case 2.6



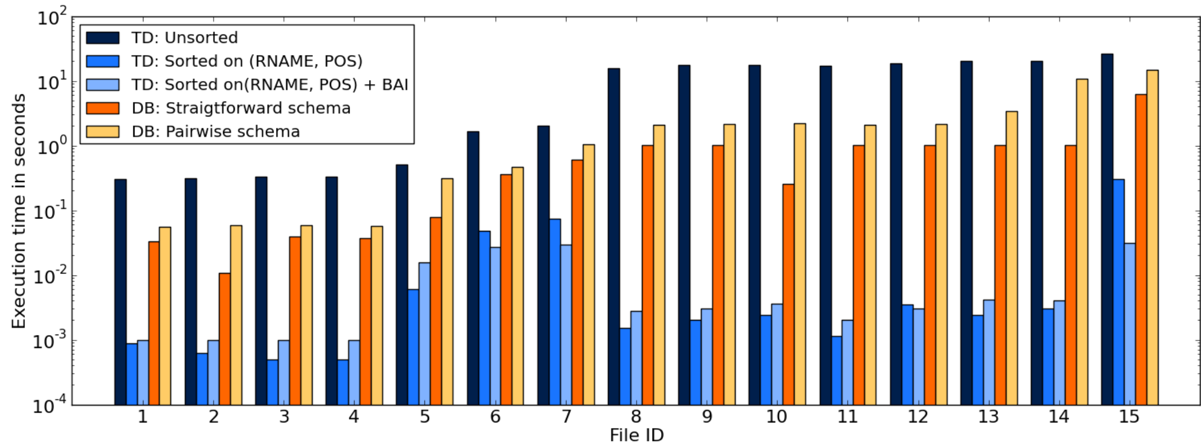
Use case 2.7



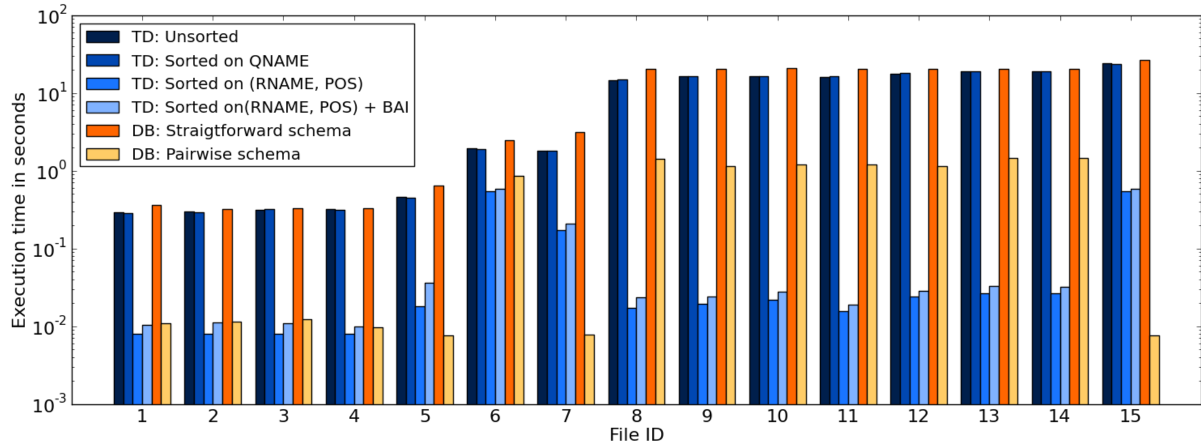
Use case 2.8



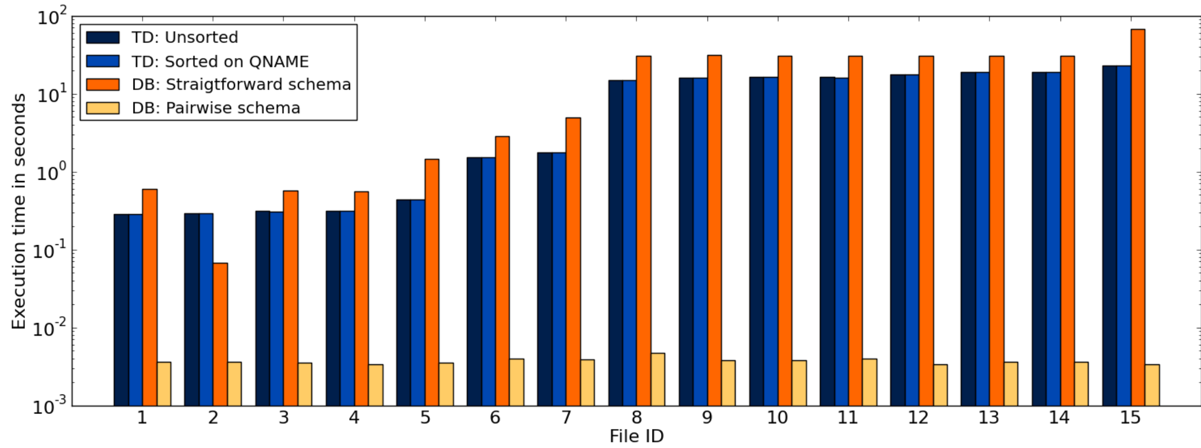
Use case 2.9



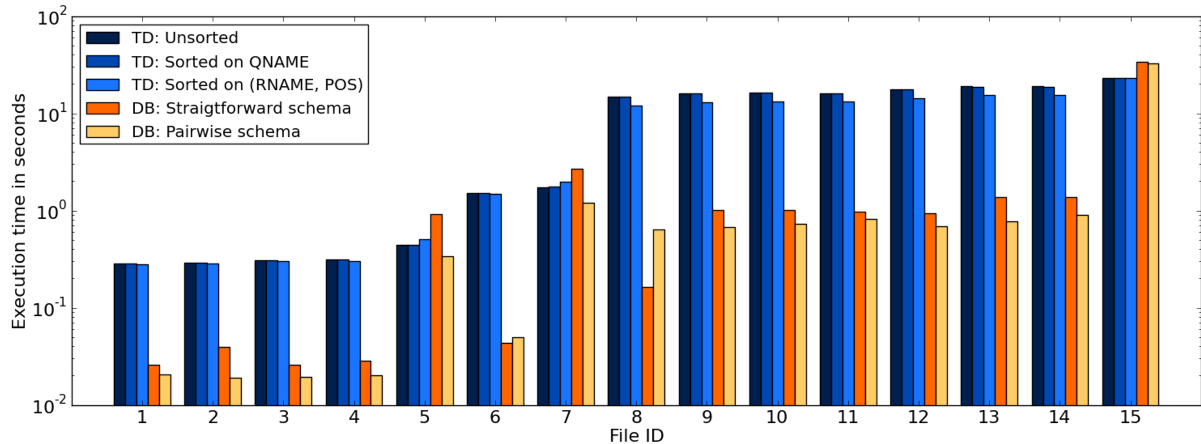
Use case 2.10



Use case 2.11

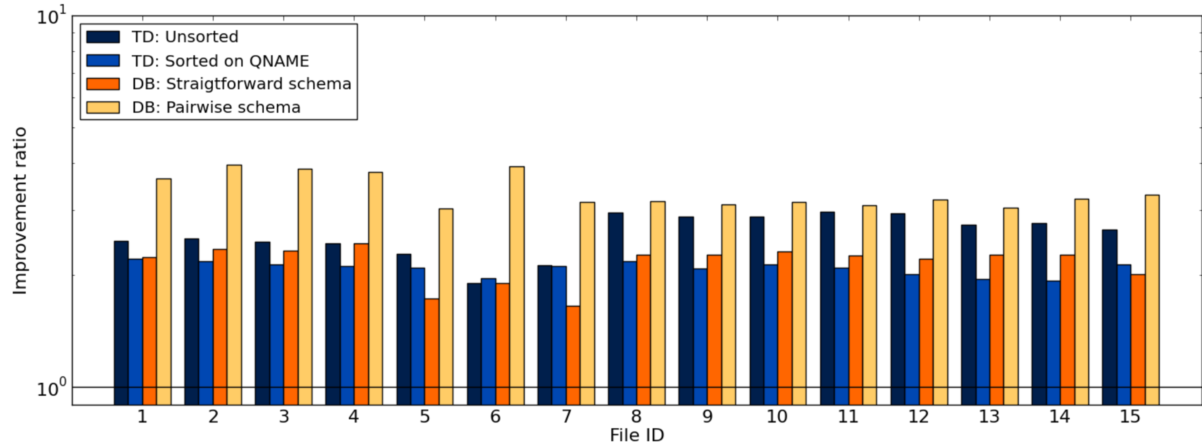


Use case 2.12

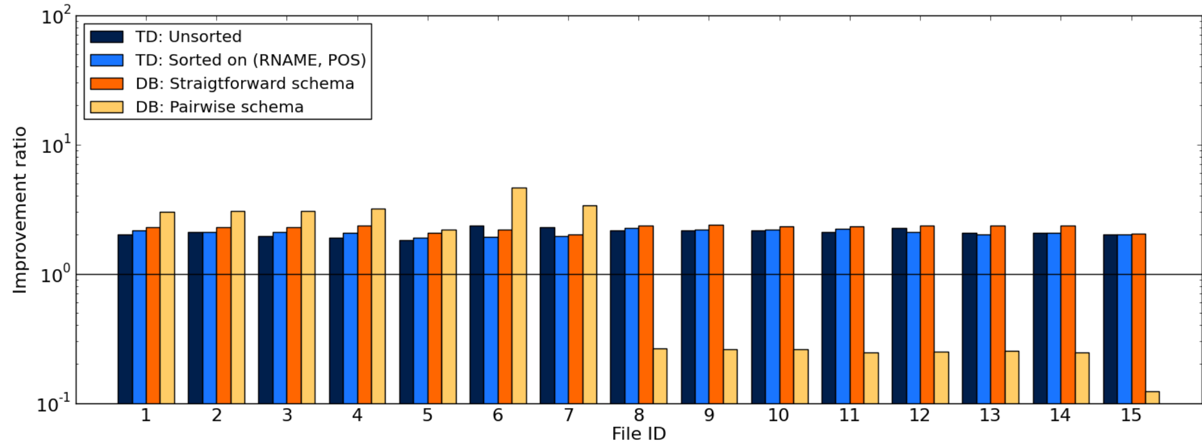


E.2 Small files – minimal-output approach performance increase

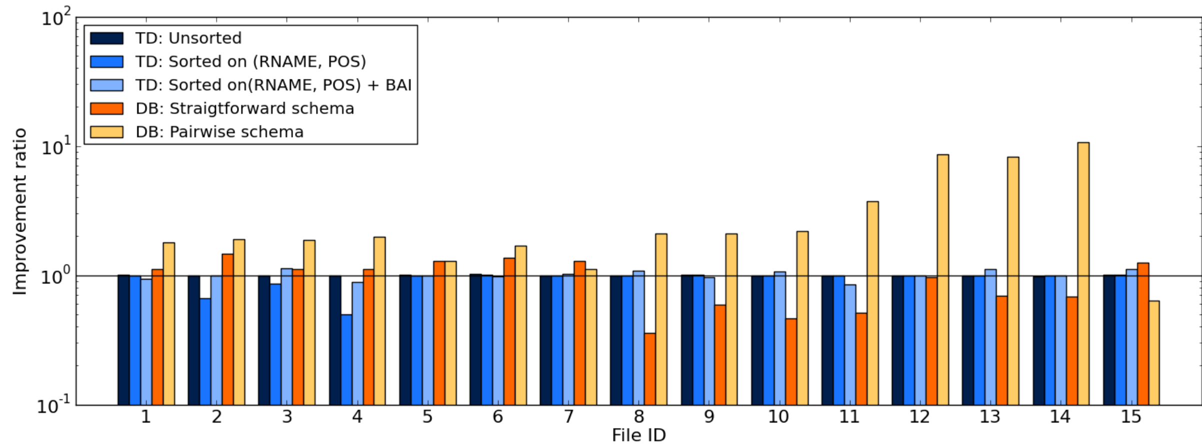
Use case 1.1



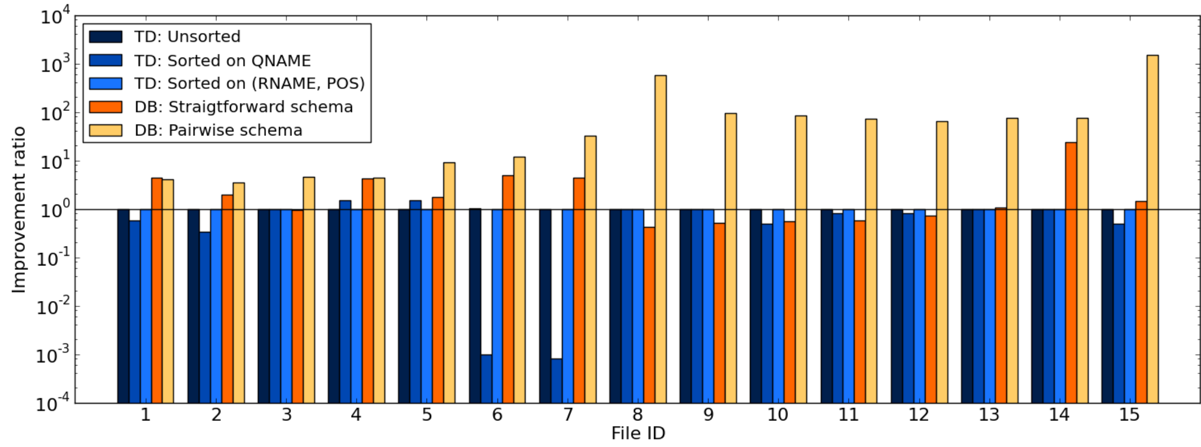
Use case 1.2



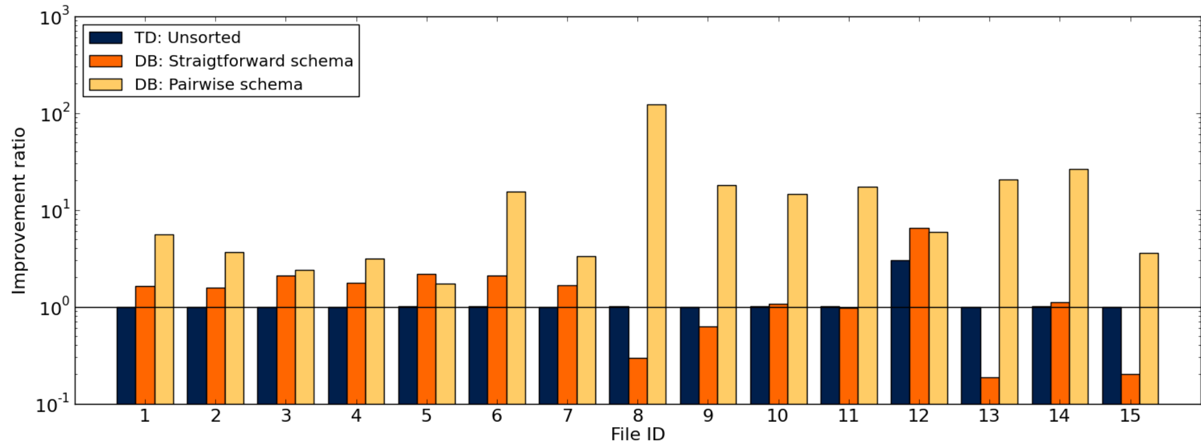
Use case 1.3



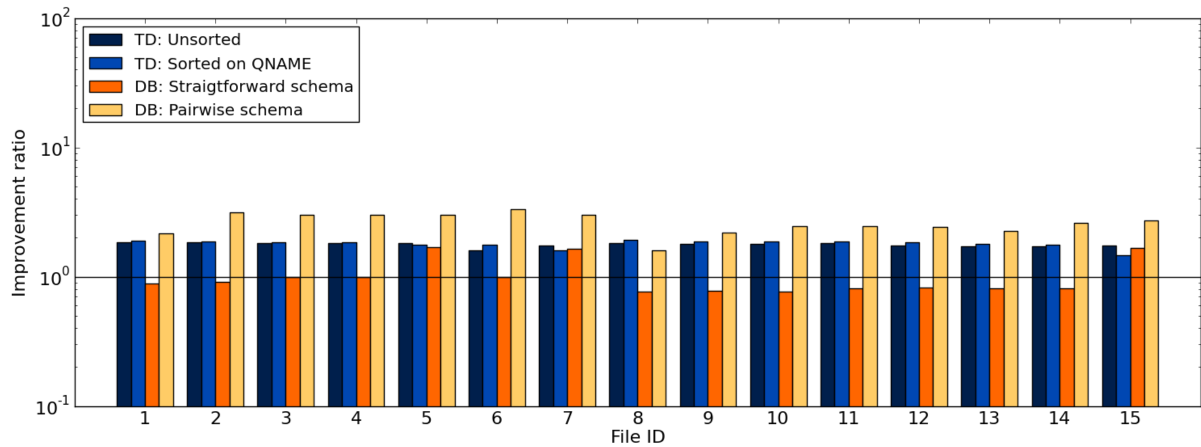
Use case 1.4



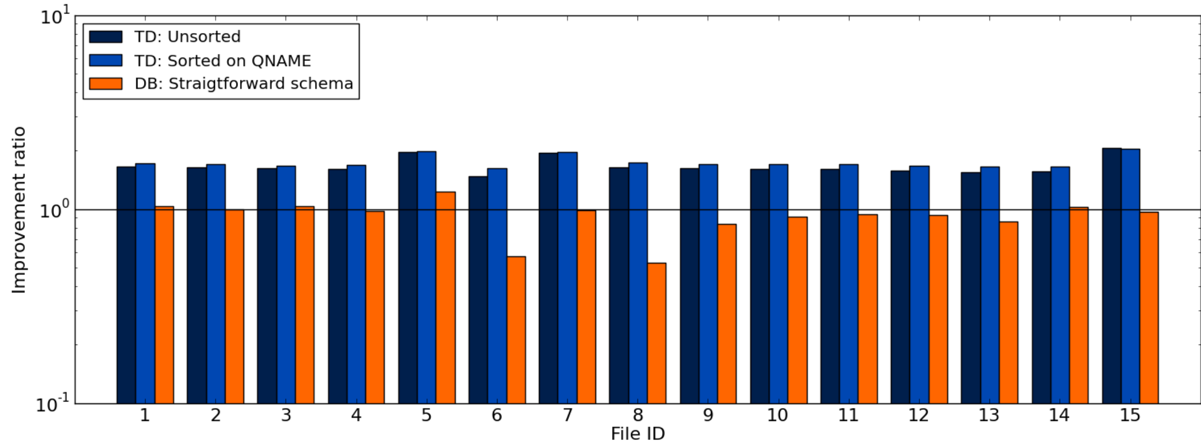
Use case 1.5



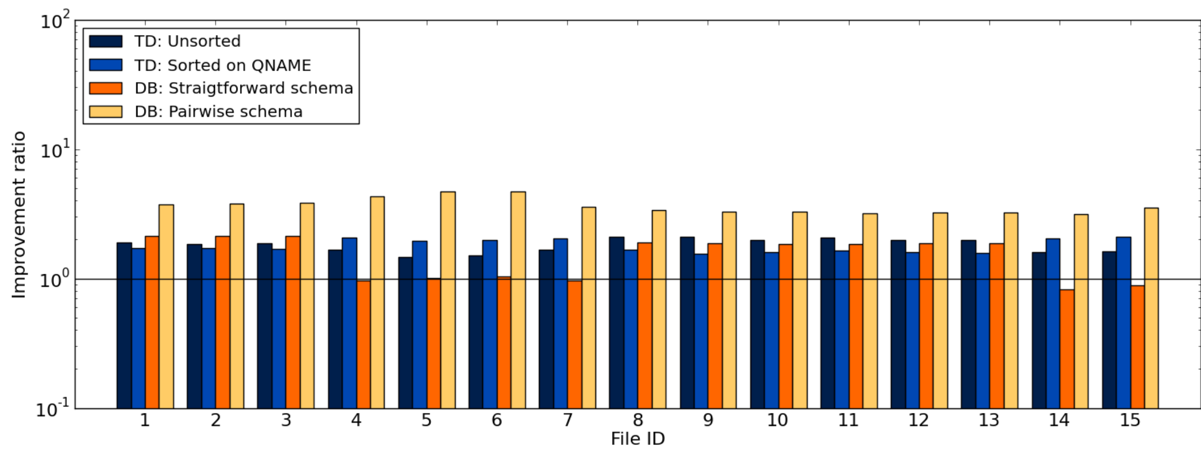
Use case 2.3



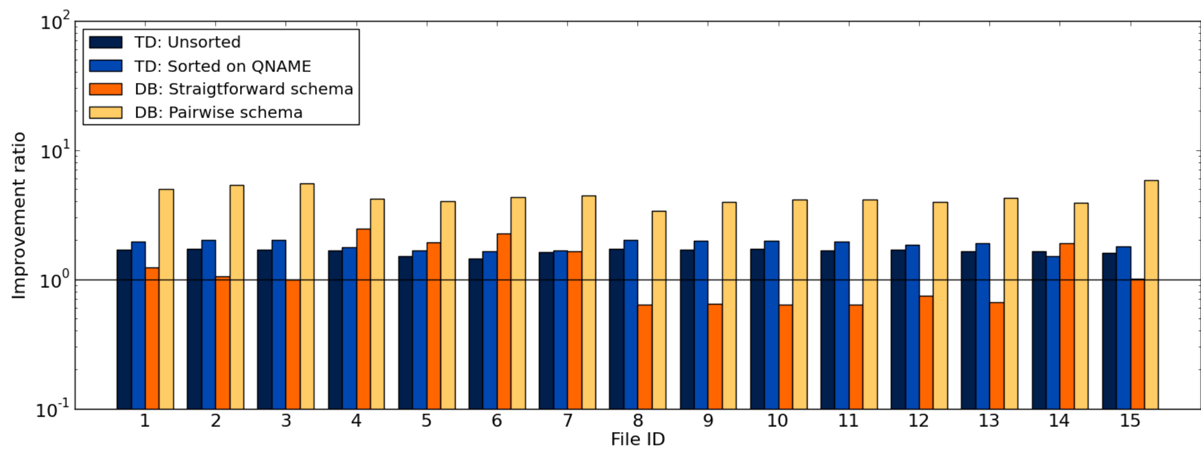
Use case 2.4



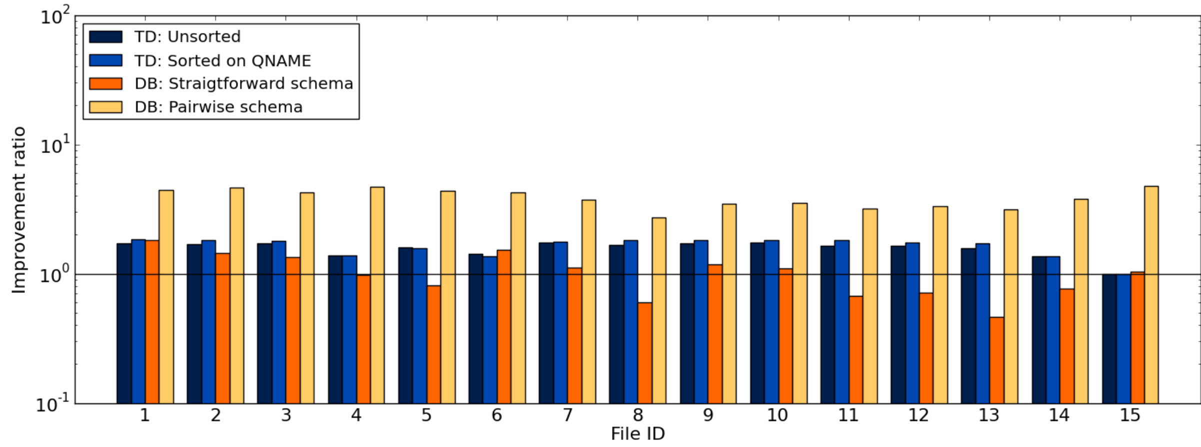
Use case 2.6



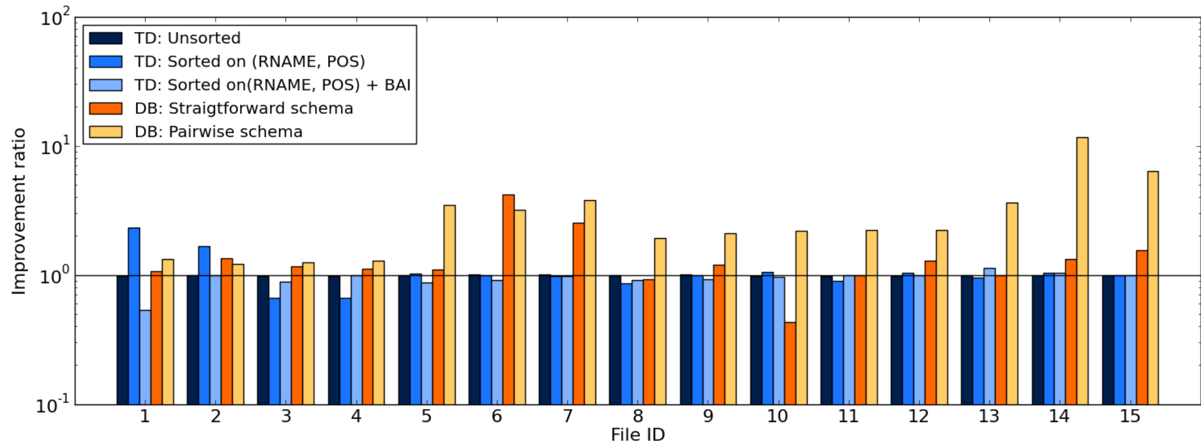
Use case 2.7



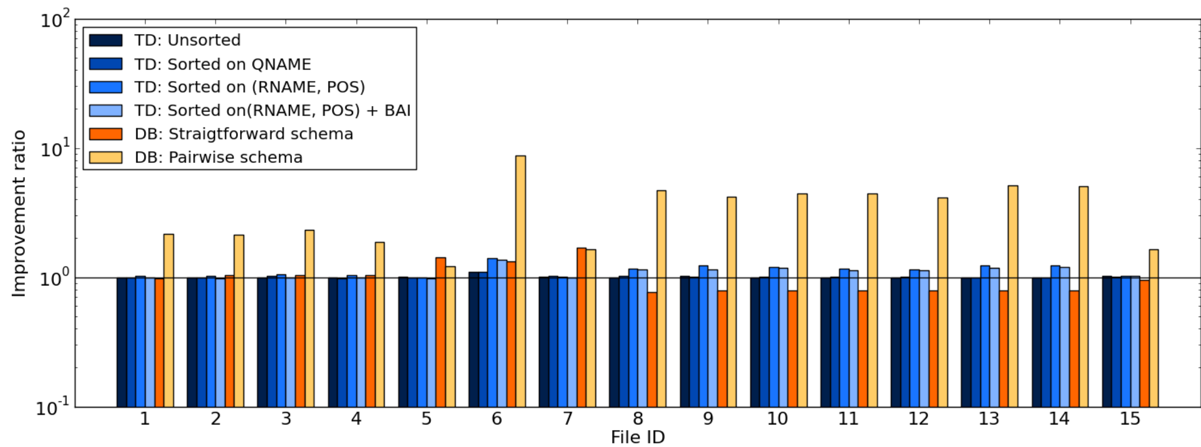
Use case 2.8

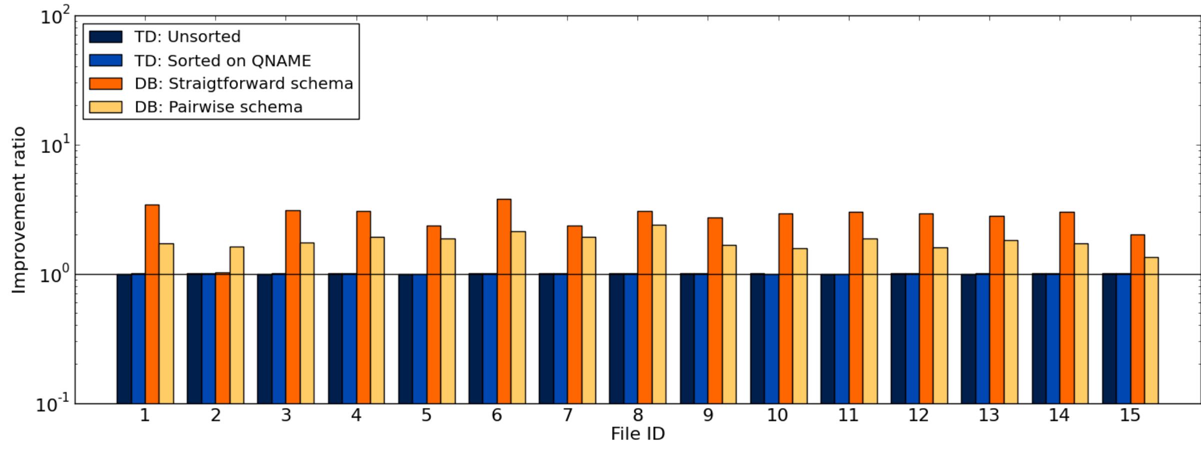


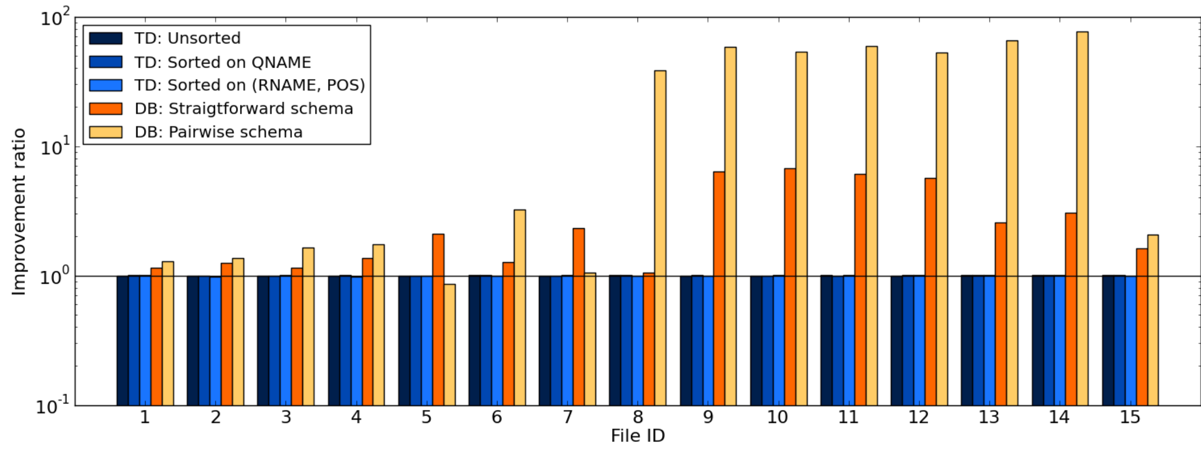
Use case 2.9



Use case 2.10

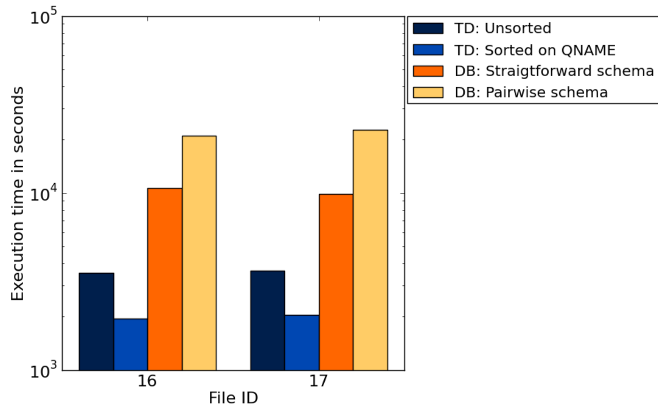



Use case 2.11
TD: Unsorted
TD: Sorted on QNAME
DB: Straigtforward schema
DB: Pairwise schema
Improvement ratio
File ID
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

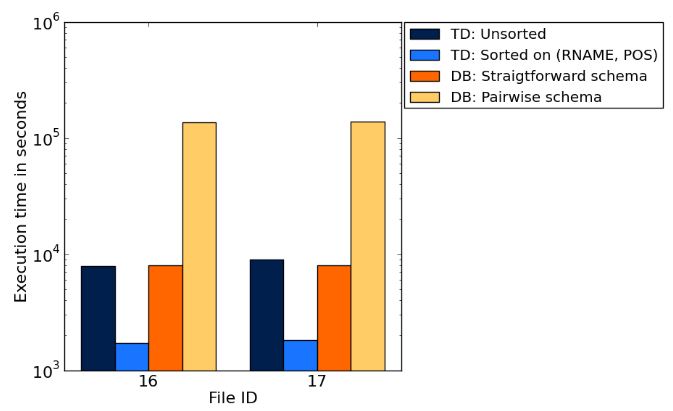

Use case 2.12
TD: Unsorted
TD: Sorted on QNAME
TD: Sorted on (RNAME, POS)
DB: Straigtforward schema
DB: Pairwise schema
Improvement ratio
File ID
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

E.3 Big files – 'minimal-output' approach

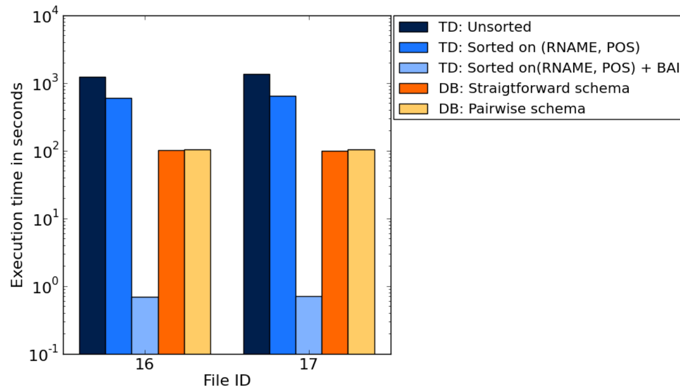
Use case 1.1



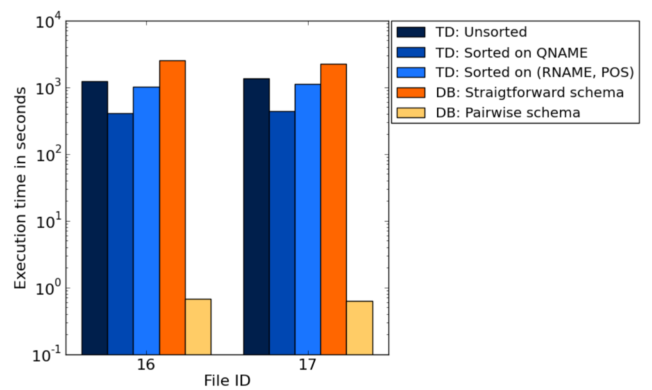
Use case 1.2



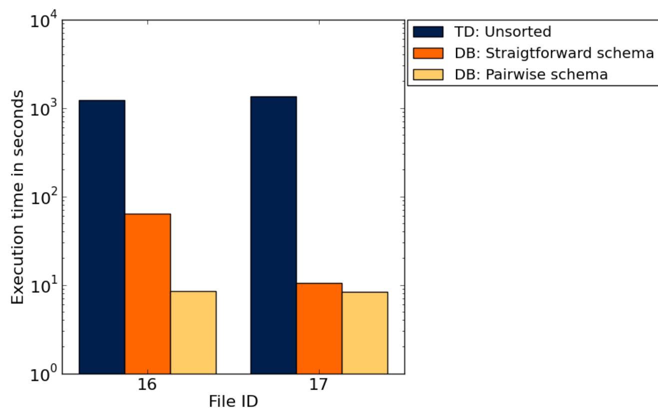
Use case 1.3



Use case 1.4

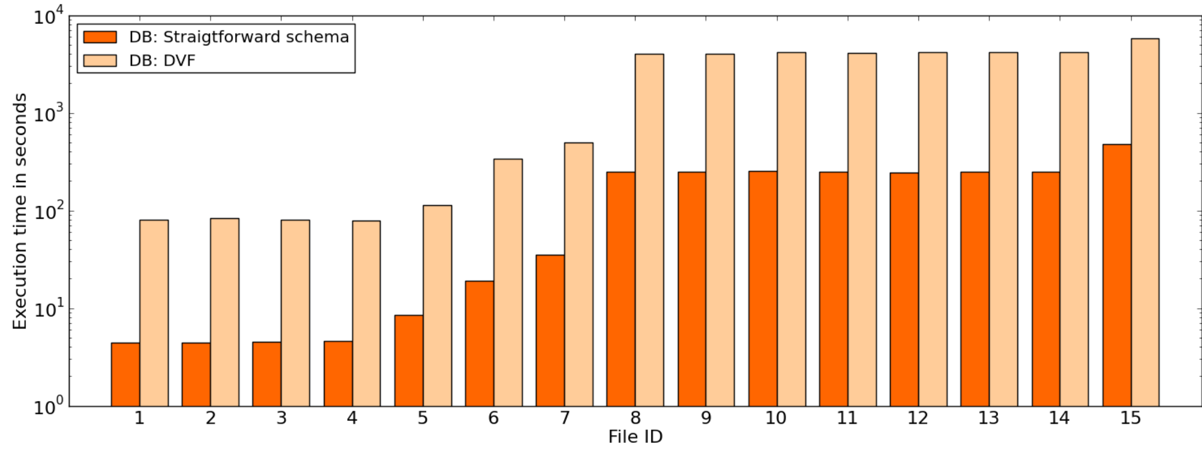


Use case 1.5

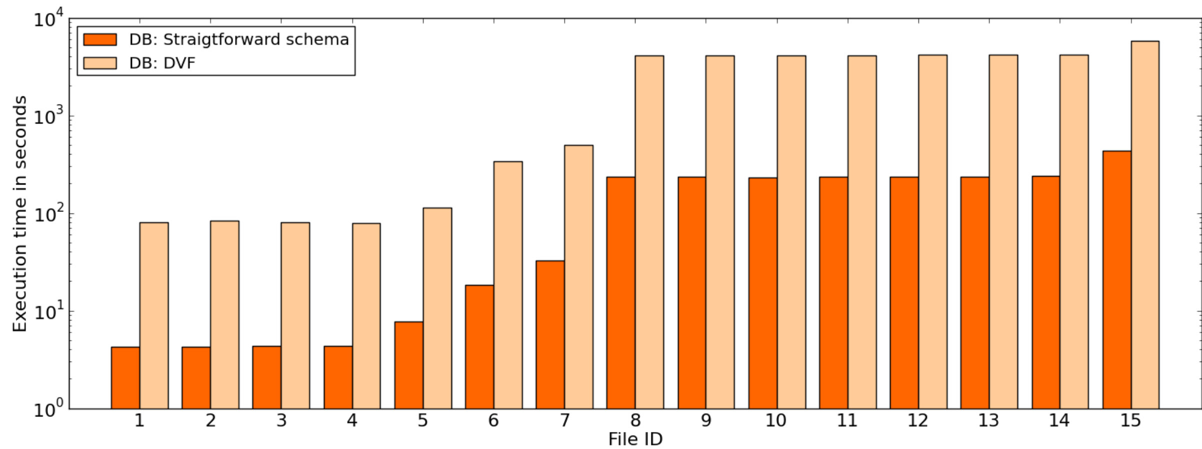


E.4 Small files – Straightforward storage schema versus Data Vault implementation

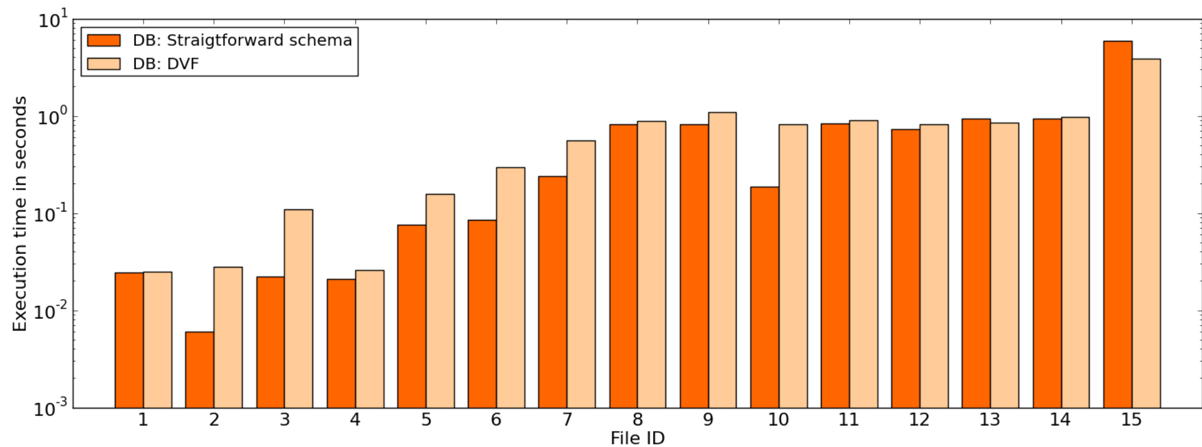
Use case 1.1



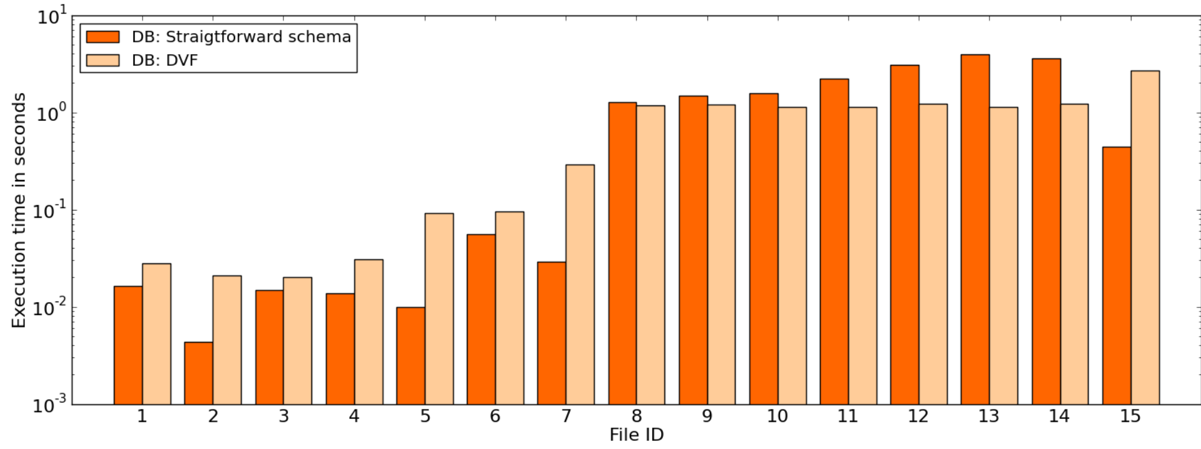
Use case 1.2



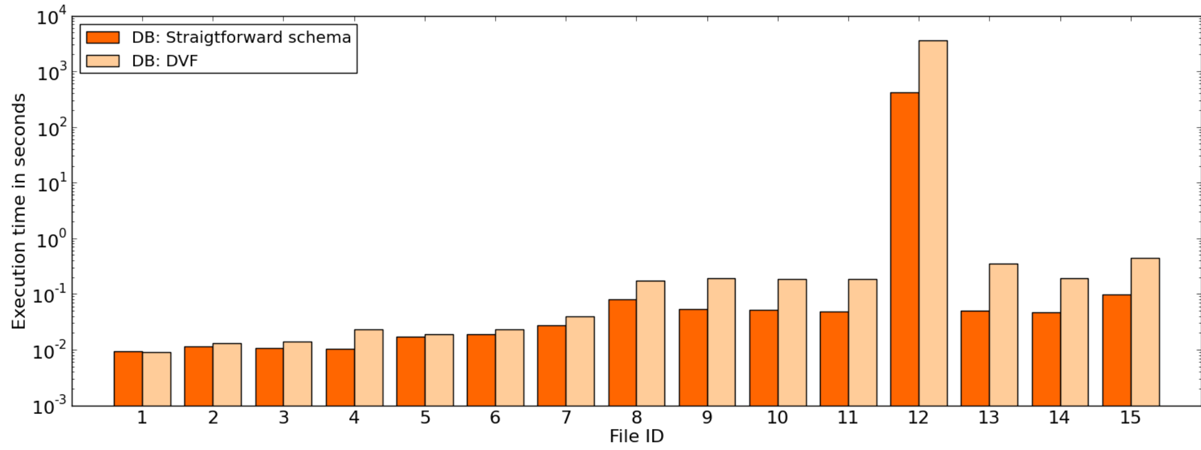
Use case 1.3



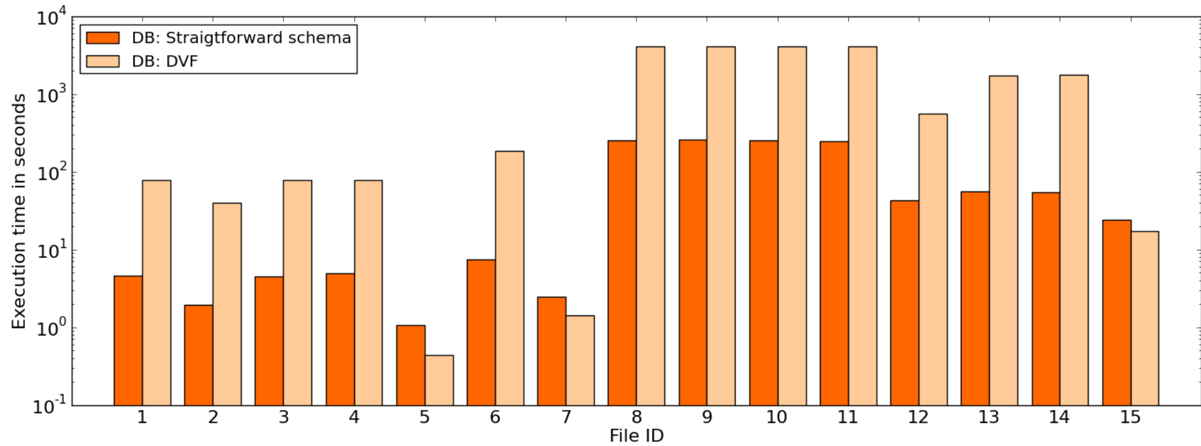
Use case 1.4



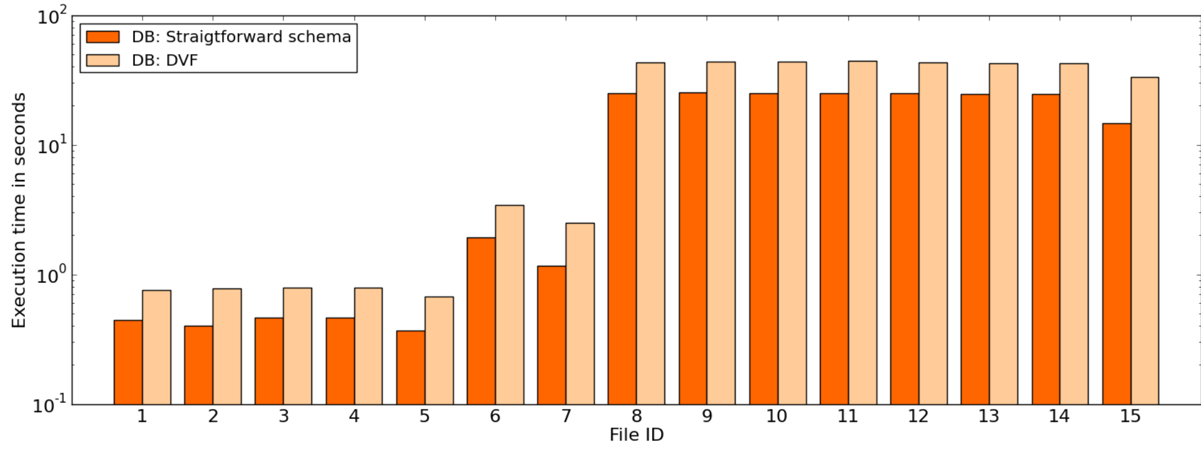
Use case 1.5



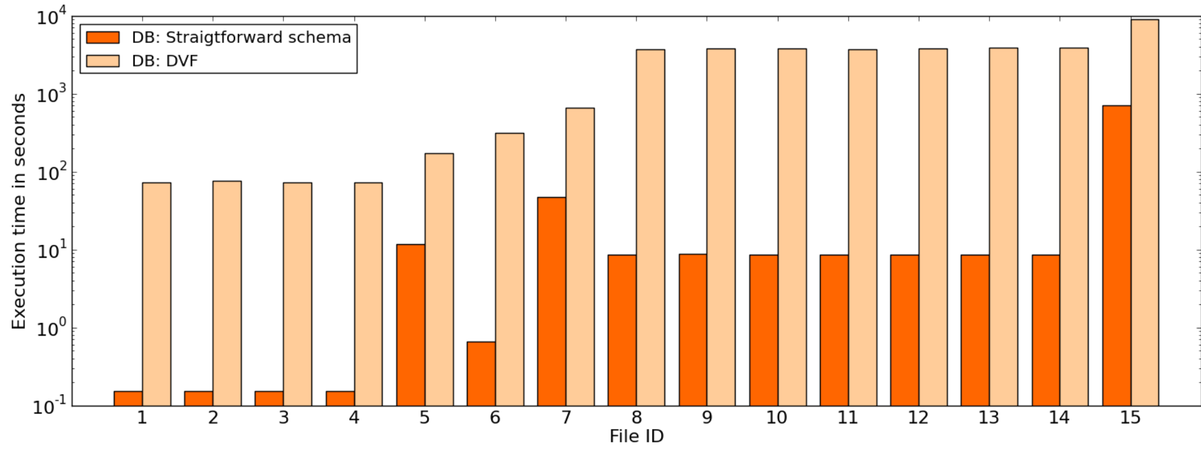
Use case 2.1



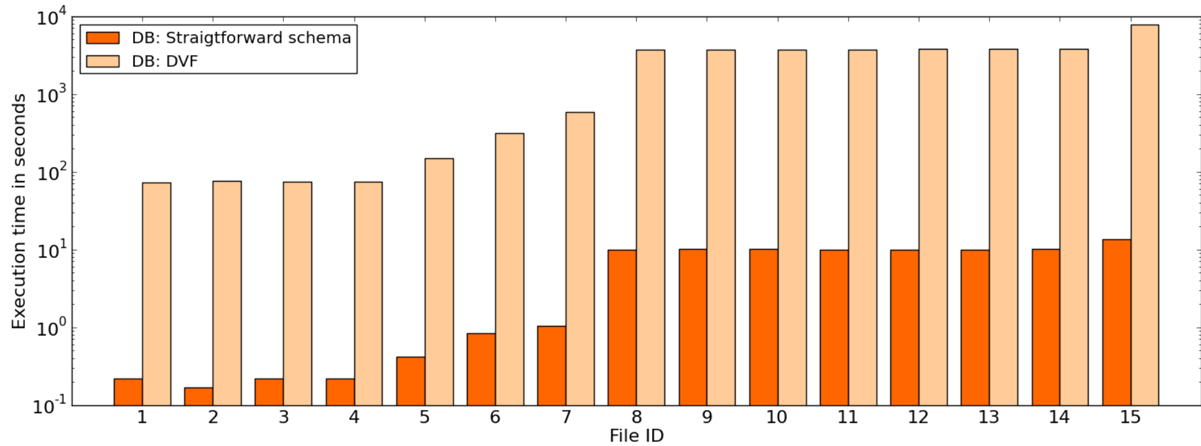
Use case 2.2



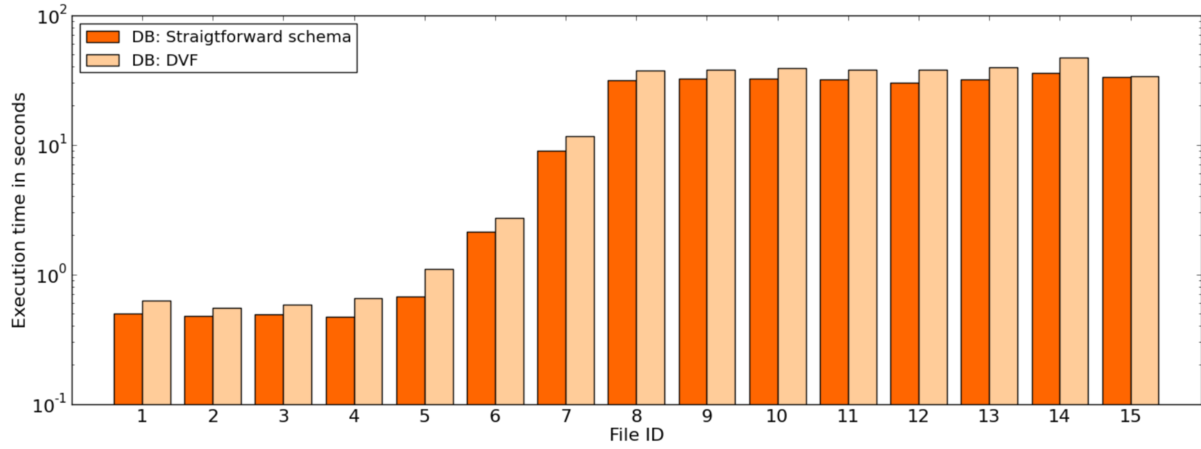
Use case 2.3



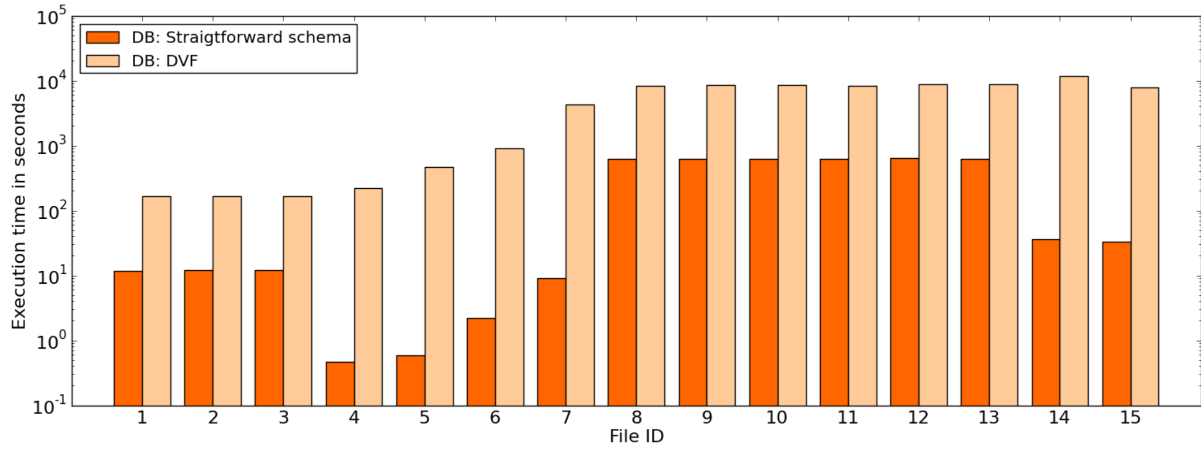
Use case 2.4



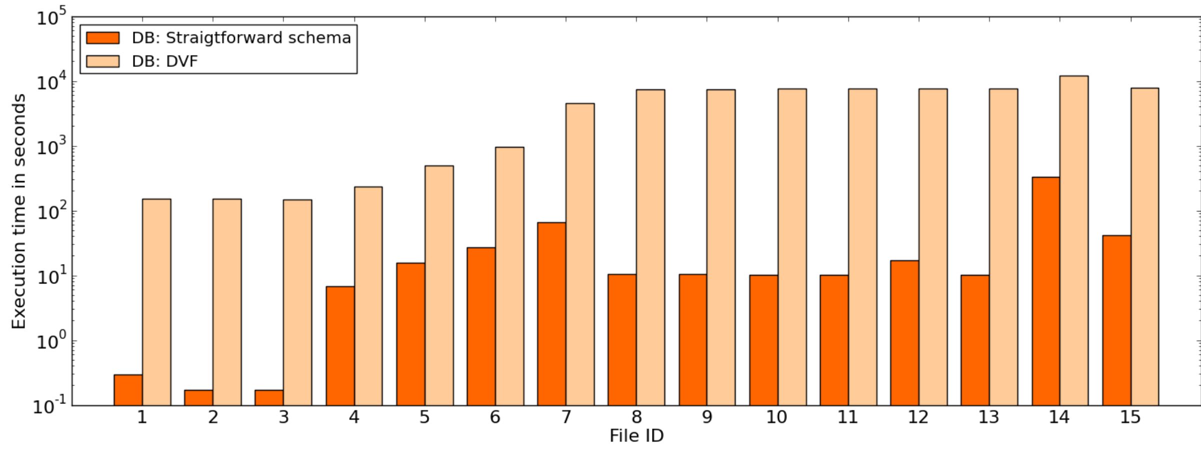
Use case 2.5



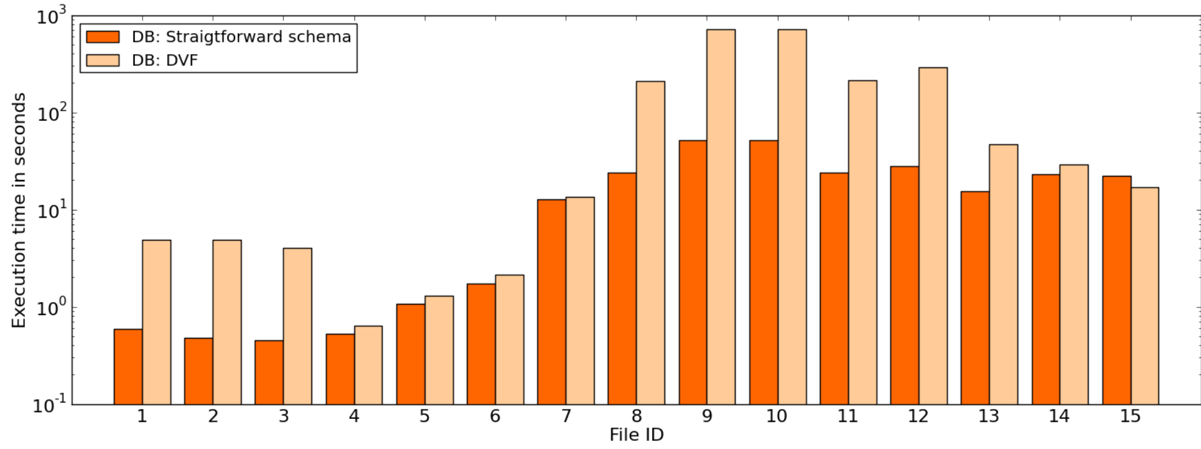
Use case 2.6



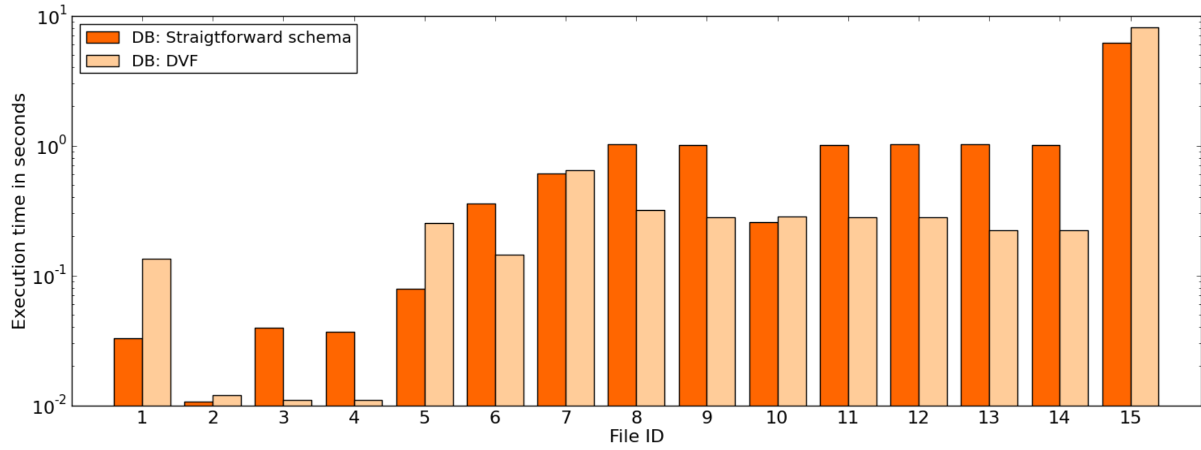
Use case 2.7



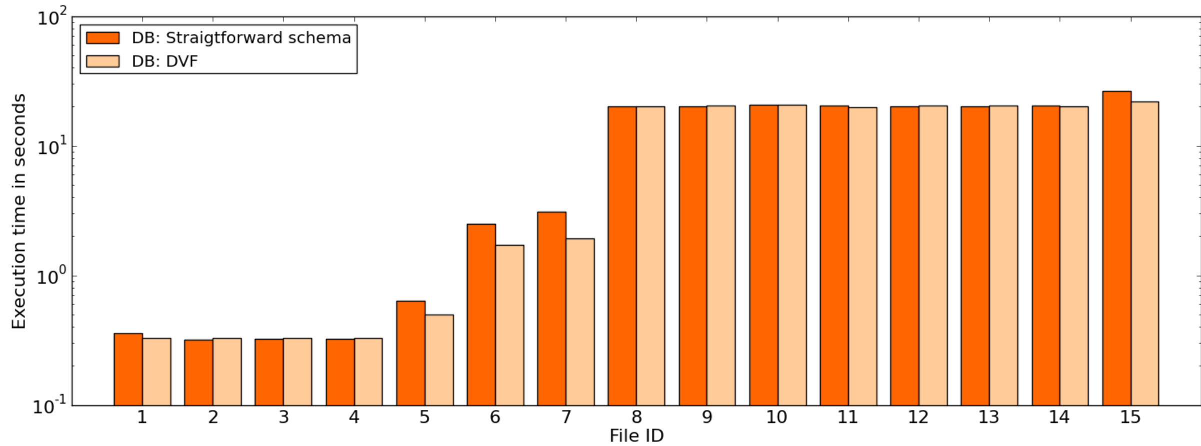
Use case 2.8



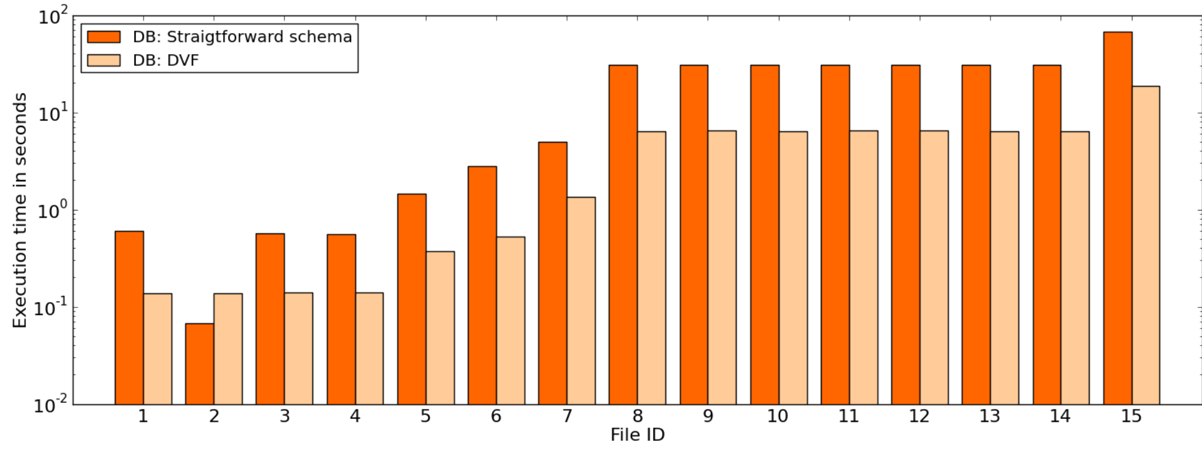
Use case 2.9



Use case 2.10



Use case 2.11



Use case 2.12