Efficient Storage and Analysis of Genome Data in Relational Database Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Sebastian Dorok

geb. am 09.10.1986 in Haldensleben

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Jens Teubner
Prof. Dr. Ralf Hofestädt

Magdeburg, den 27.04.2017

Dorok, Sebastian:
Efficient storage and analysis of genome data in relational database systems
Dissertation, University of Magdeburg, 2017.

Abstract

Genome analysis allows researchers to reveal insights about the genetic makeup of living organisms. In the near future, genome analysis will become a key means in the detection and treatment of diseases that are based on variations of the genetic makeup. To this end, powerful variant detection tools were developed or are still under development.

However, genome analysis faces a large data deluge. The amounts of data that are produced in a typical genome analysis experiment easily exceed several 100 gigabytes. At the same time, the number of genome analysis experiments increases as the costs drop. Thus, the reliable and efficient management and analysis of large amounts of genome data will likely become a bottleneck, if we do not improve current genome data management and analysis solutions.

Currently, genome data management and analysis relies mainly on flat-file based storage and command-line driven analysis tools. Such approaches offer only limited data management capabilities that can hardly cope with future requirements such as annotation management or provenance tracking. On the other hand, we have advanced and sophisticated relational database management systems that are well researched and already provide advanced data management functionality. However, existing approaches that use relational database technology in the context of genome analysis mainly leverage the data integration capabilities for result data. A holistic integration of genome-specific analysis functionality is still missing as relational database system are said to perform bad on genome analysis tasks.

In this thesis, we examine the design space to integrate variant detection into a relational database system. To ensure optimal performance, we take care to integrate the genome-specific analysis task using relational database operators only. Our evaluation on three real-world data sets confirms that our careful design allows us to use a relational database systems as competitive alternative for specialized analysis tools with regard to analysis runtime. In addition, we propose genome-specific compression and query optimization techniques to improve the performance of a database-driven analysis pipeline further. Using our techniques, we can outperform existing analysis tools by up to a factor of five with regard to runtime and reduce the storage requirements compared to a standard relational database system by up to 50%.

Inhaltsangabe

Die Genomanalyse ermöglicht es Forschern den genetischen Code von Lebewesen zu ermitteln. In Zukunft wird die Genomanalyse ein wichtiges Mittel zur Behandlung von Krankheiten sein, die auf Variationen des genetischen Codes beruhen. Zu diesem Zweck wurden und werden effiziente Tools zur Erkennung von Genomvariationen entwickelt.

Allerdings steigt die Menge an verfügbaren und zu analysierenden Genomdaten kontinuierlich an. Die Datenmengen, die in einem typischen Genomanalyseexperiment erzeugt werden, übersteigen leicht mehrere 100 Gigabyte. Gleichzeitig erhöht sich die Zahl der Genomanalyseexperimente, da diese kostengünstiger werden. Daher wird in der Zukunft neben der effizienten Analyse auch die zuverlässige und effektive Verwaltung von Genomdaten immer wichtiger werden.

Derzeit werden Genomdaten in Dateien mit speziellen Formaten verwaltet und hauptsächlich mit Kommandozeilen-Tools verarbeitet. Solche Ansätze bieten nur einfache Möglichkeiten zur Datenverwaltung, die kaum den zukünftigen Anforderungen wie der Verwaltung von Annotationen oder der Rückverfolgbarkeit von Analyseergebnissen gerecht werden können.

Relationale Datenbanksysteme dagegen verfügen bereits heute über effiziente und effektive Mittel zur Datenverwaltung. Allerdings werden sie in der Genomanalyse meistens nur zur Verwaltung von Ergebnisdaten eingesetzt, da sie die Integration der Ergebnisse mit anderen Datenquellen vereinfachen. Eine umfassende Integration genomspezifischer Analysefunktionalität fehlt hingegen.

In dieser Arbeit wird untersucht, wie die Erkennung von Genomvariationen mit Hilfe eines relationalen Datenbanksystems durchgeführt werden kann. Um eine optimale Verarbeitung der Genomdaten im Datenbanksystem zu gewährleisten, werden existierende, hoch optimierte Datenbankoperatoren wiederverwendet. Damit ist es möglich die Analyse in einem Datenbanksystem vergleichbar schnell durchzuführen wie mit spezialisierten Analysewerkzeugen. Um die datenbankgestützte Analyse weiter zu verbessern, werden außerdem genomspezifische Kompressions- und Datenverarbeitungstechniken in ein relationales Datenbanksystem integriert. Mit diesen Techniken kann die Analyse im Vergleich zu bestehenden Analysewerkzeugen um bis zu Faktor fünf beschleunigt werden. Gleichzeitig kann der Speicherverbrauch gegenüber dem Einsatz von herkömmlichen Kompressionsverfahren für Datenbanksysteme um bis zu 50% reduziert werden.

Acknowledgements

I thank my advisors Gunter Saake and Horstfried Läpple, who have guided me since my Master's thesis. We had many fruitful discussions about the content and the direction of this thesis leading to its current state. Furthermore, I thank Karsten Tittmann from Bayer Healthcare AG for his trust in me and the topic that made this work possible. I also thank Uwe Scholz and Matthias Lange from IPK Gatersleben for inspiring discussions about my work and support with evaluation data.

Special thanks to the complete CoGaDB team around Sebastian Breß for delivering a great piece of software that allowed me to conduct my research. As always, not everything works as expected, but with the help of the team, I was able to implement and evaluate my ideas successfully. Thank you.

Usually, you do not do a PhD alone. At some point in time, you need someone to discuss your ideas. Fortunately, I found nice and encouraging fellows in the DBSE working group in Magdeburg. Special thanks to David Broneske, Andreas Meister and Marcus Pinnecke for feedback on this thesis and especially for the productive team meetings and constant pressure to finish the work. Thanks guys.

I especially thank Sebastian Breß. He taught me what it means to be a scientist, that it can be hard and frustrating, but also fun and satisfying. He kept my motivation up more than once. Although he moved several times during the last three years, he was always there to support me and my work. I thank him for his trust in my work and capabilities to do the work. Thank you so much!

Last but not least, I thank my family and in particular my partner Vicki and my little son Paul for their constant support and patience, especially during the Christmas holidays in 2016. Thank you!

Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

API	application programming interface
BITDICT	bit-packed dictionary encoding
DBMS	database management system
DICT	dictionary encoding
DNA	deoxyribonucleic acid
GPU	graphics processing unit
NGS	next-generation sequencing
RLE	run-length encoding
SAM	sequence alignment/ map
SNV	single nucleotide variant
SQL	structured query language
UDA	user-defined aggregation function
UDF	user-defined function

1. Introduction

In recent years, genome analysis experienced a big hype due to the possible positive impact on human life. For example, the analysis of genetic variations is a promising method to improve the detection, prediction and prevention of diseases [19]. The decreasing cost and time to sequence whole genomes amplify this trend. The initial human genome project [62] required over 2.5 billion dollars and took over ten years to complete a first version of the human genome. Since then, next-generation sequencing (NGS) techniques led to a decrease of sequencing costs to several thousand dollars per genome and the sequencing process finishes within days [82]. Consequently, data generation is not the bottleneck anymore, but managing, analyzing and assessing large amounts of genome data in general and NGS data in particular [88].

The analysis of NGS data can be separated into two phases: detecting genetic variations and investigating their consequences. Typically, researchers use specialized analysis tools to detect genetic variations. Then, the found genetic variations must be integrated with information from other data sources to investigate their consequences [19]. Since relational database management systems (DBMSs) provide excellent data-integration capabilities, researchers use them to facilitate the analysis of found genetic variations [70, 72, 115, 128]. Detecting genetic variations directly within the database system is not possible, because off-the-shelf DBMSs do not provide the required domain-specific functionalities [67].

The separation between detecting genetic variations and investigating their consequences complicates provenance tracking of analysis results within the complete analysis process. Involved analysis tools and the database system have to cooperate and provenance information has to be collected and exchanged partly manually [112]. In contrast, extending DBMSs with domain-specific functionality and enabling researchers to analyze genome data directly within the database system would allow for reliable genome data management and analysis. Researchers would be equipped with comprehensive data-management features, such as provenance tracking [40] or annotation management [10], within the complete analysis process. Furthermore, researchers would be able to perform variant detection on demand and in a declarative fashion improving comprehensibility of analysis results [108]. Within the discussion about limited reproducibility in life-sciences [9], such data-management features become more and more important.

1.1 Goal of this thesis

Goal. The goal of this thesis is to show that we can detect genetic variants efficiently in a relational DBMS. This is a first step to make advanced data management capabilities of relational DBMSs, such as provenance tracking [40] or annotation management [10], available within the complete genome analysis process. To this end, it is necessary to investigate approaches for storing, querying and analyzing genome data within a database system and to evaluate their efficiency in comparison to state-of-the-art storage and analysis approaches.

Challenge. The integration of genome analysis tasks into relational DBMSs is said to be hardly possible, because the relational database model *appears* to not fit well for modeling genome data and expressing genome analysis queries [67].

Of course, integrating genome-specific analysis functionality into DBMSs requires functional extensions via stored procedures or user-defined functions (UDFs), otherwise an integration would not be possible. Nevertheless, current approaches limit the benefits of declarative query languages, because they integrate most of the analysis-related functionality into one single stored procedure or UDF [108, 109]. In addition, such approaches are hard to optimize and parallelize by the DBMS, because the DBMS does not know about the specifics of the encapsulated functionality. Thus, it is very likely that DBMSs do not compete with specialized analysis tools regarding analysis performance due to the use of UDFs. Nevertheless, the reason for using UDFs is not always the inability to perform analyses using conventional database operators, but to avoid that intermediate results exceed available main-memory [108].

Considering modern computer architectures with increasing amounts of main-memory, main-memory consumption is not the limiting factor anymore. Additionally, new computer architectures led to a renaissance of column-oriented main-memory DBMSs. These DBMSs offer tremendous performance improvements especially for analytical workloads [44, 60] compared to traditional disk-based DBMSs, because they are designed to leverage the full potential of increasing amounts of main memory and the growing number of CPU cores [125]. For this reason, we focus on the use of column-oriented main-memory DBMSs to achieve our goal.

Benefits. Extending database systems with genome-specific analysis functionality and allowing for genome data analysis directly within the database will lead to following benefits:

On the one hand, we expect to improve the comprehensibility of the genome analysis process [35]. Since the outcome of variant detection highly depends on the chosen approaches and parametrizations [96, 99], it is important to document the chosen configuration. Using the structured query language (SQL) to express analyses inherently provides an analysis description for documentation purposes. Moreover, we can use different approaches and parametrizations during analysis, because we compute analysis results on demand and do not rely on precomputed analysis results. To be able to compute analysis results on demand, we have to store the underlying raw data, which directly improves reproducibility.

On the other hand, we expect to speed up the analysis of genome data, if it is stored using a database system. First, we can perform analysis directly at the data site avoiding costly data exports. Second, main-memory database technology should allow us to provide competitive analysis performance compared to specialized analysis tools. Moreover, we expect to benefit from future improvements of DBMSs such as co-processor acceleration, if we can perform genome analysis tasks using standard database operators as a basis.

Research questions. Pursuing our goal, we answer the following four research questions:

1. *Which steps of variant detection should we integrate into a DBMS?*

   Variant detection comprises several subtasks. It is not clear yet whether all of these tasks can or should be integrated into a DBMS. Therefore, we first have to describe the essential subtasks and investigate whether an integration is reasonable considering the advantages and disadvantages of computing the analysis result on demand or storing a static copy of the analysis result. We answer this question in Chapter 2.

2. *How can we express variant detection using relational DBMS operators as a basis?*

   Variant detection is not a typical database task. Still we want to use relational DBMS operators as a basis to perform the actual analysis, which allows us to benefit from the optimized processing stack of a DBMS. Moreover, we reduce implementation and maintenance effort, ensure optimal processing also on future hardware and facilitate declarative querying via SQL. We answer this question in Chapter 3.

3. *How can we process genome data sets as efficient as specialized analysis tools using relational DBMSs?*

   Genome data sets can have a size of up to hundreds of gigabytes. In order to make database-integrated variant detection a competitive alternative to specialized analysis tools, we must be able to process these gigabytes of genome data within the database system as efficient as specialized analysis tools. Therefore, we have to apply advanced processing techniques to fully exploit all available resources. We answer this question in Chapter 4.

4. *How can we store genome data sets using a relational DBMS as efficient as state-of-the-art flat-file approaches without sacrificing analysis speed?*

   Even though main memory capacities increase, it is necessary to compress genome data to keep more genome data sets in main memory and better utilize memory bandwidth. We use the storage requirements of state-of-the-art flat-file formats as baseline and focus our research on lightweight compression schemes to mitigate the negative impact of compression on query runtime due to computational overhead. However, genome data mainly consists of unique strings making it hard to compress it using standard lightweight compression schemes. Genome-specific compression schemes promise better compression ratios, but it is not clear yet whether and how we can integrate them into database systems. Moreover, we have to investigate whether and how we can still process compressed genome data as efficient as specialized analysis tools. We answer this question in Chapter 5.

1.2 Outline and contributions

In the following, we outline and highlight our contributions. The thesis is structured as follows. In Chapter 2, we introduce necessary basics about genome analysis. Furthermore, we provide an overview of a typical variant detection and analysis process and discuss related data management challenges. Moreover, we present related work that uses DBMSs or database technology to perform genome analysis tasks in general and variant detection in particular. We use this knowledge to contribute:

A concept for variant detection using DBMSs. The concept describes which parts of the variant detection process should be integrated into a relational DBMS. To this end, we discuss the single analysis steps in detail and base our decision which step to integrate on the trade-off between analysis throughput and the outreach of improved data management capabilities. We published parts of the material in the following papers:

[35] Sebastian Dorok, Sebastian Breß, Horstfried Läpple, and Gunter Saake. Toward efficient and reliable genome analysis using main-memory database systems. In *SSDBM*, pages 34:1–34:4, 2014.

[33] Sebastian Dorok. The relational way to dam the flood of genome data. In *SIGMOD/PODS Ph.D. Symposium*, pages 9–13, 2015.

In [35], we discussed data management challenges within genome analysis and presented an initial concept to address them using a database system. In [33], we presented more details about the concept.

In Chapter 3, we refine our high-level concept from Chapter 2 and describe how we can perform variant detection using relational database operators as a basis. We put our primary focus on the detection of single nucleotide variants (SNVs). SNVs are variations of the individual genome at single genome positions and may indicate a predisposition for diseases [133] or have impact on drug efficacy [46]. Thus, their efficient and reliable detection is important for medical and pharmaceutical purposes. We explain how we store the required genome data and perform SNV detection using SQL. We contribute:

SNV detection as aggregation task. We model the detection of SNVs as aggregation task over genome data. To this end, we introduce a user-defined aggregation function (UDA) and a genome-specific database schema resembling a star schema. Moreover, we show how we perform SNV detection via SQL. We published the material in the following papers:

[36] Sebastian Dorok, Sebastian Breß, and Gunter Saake. Toward efficient variant calling inside main-memory database systems. In *BIOKDD-DEXA*, pages 41–45, 2014.

[39] Sebastian Dorok, Sebastian Breß, Jens Teubner, and Gunter Saake. Flexible analysis of plant genomes in a database management system. In *EDBT*, pages 509–512, 2015.

In [36], we introduced the general concept and a first assessment using MonetDB [60]. In [39], we presented a prototype using CoGaDB [14] and showed how to integrate further analysis tasks that accompany SNV detection.

In Chapter 4, we explain how we integrate variant detection efficiently into a relational DBMS. Our contribution is:

Efficient declarative SNV detection. Besides describing SNV detection logically as an aggregation task, we also have to ensure to process genome data efficiently. Otherwise, long waiting times would sacrifice the benefits of detecting SNVs on demand via SQL. Therefore, we compare different join and aggregation implementations and examine their advantages and disadvantages. Using advanced processing techniques such as the invisible join [3], we are able to improve the runtime by up to a factor of four allowing a relational DBMS to keep pace with specialized analysis tools regarding analysis runtime. We published parts of the material in the following paper:

[34] Sebastian Dorok. Memory efficient processing of DNA sequences in relational main-memory database systems. In *GvDB*, pages 39–43, 2016.

In Chapter 5, we introduce genome-specific extensions for relational database systems to improve storage consumption and to speed up the analysis especially of compressed genome data. Our contributions are:

Genome-specific compression in column stores. Genome data consists mainly of unique strings that are hard to compress using lightweight compression schemes. However, we rely on the use of lightweight compression schemes, since these allow for efficient processing of compressed data. To this end, we propose to integrate genome-specific compression schemes such as reference-based compression into a database system. We show that the database schema that allows us to perform SNV detection via SQL also facilitates to integrate reference-based compression. In combination with a modified run-length encoding, which allows us to compress sequences of incremented numbers, we are able to reduce the storage increase and improve overall storage consumption compared to a straightforward database-storage approach by up to a factor of two.

Genome-specific query optimization. Efficient processing of genome data is a prerequisite to provide on demand SNV detection. Additionally, we propose a filtering technique that uses knowledge about the new UDA for SNV detection to further speed up the analysis. Furthermore, we show how we can benefit from reference-based compressed genome data to speed up the computation for this optimization itself. Using our optimization, we are able to detect SNVs in a human genome data set more than five times faster than using a specialized external tool. Compared to a straightforward database approach, we achieve a speedup of up to a factor of eight.

We published both optimizations in the following papers:

[37] Sebastian Dorok, Sebastian Breß, Jens Teubner, et al. Efficient storage and analysis of genome data in databases. In *BTW*, pages 423–442, 2017.

[38] Sebastian Dorok, Sebastian Breß, Jens Teubner, et al. Efficiently storing and analyzing genome data in database systems. In *Datenbank-Spektrum*, 2017. doi:10.1007/s13222-017-0254-9.

In [37], we introduce the basic concepts of our genome-specific compression schemes and query filtering technique. In [38], we provide further details about the inner mechanics of our compression schemes. In particular, we discuss efficient access and retrieval approaches for our compression schemes.

In Chapter 6, we conclude our work and provide an overview on future work.

2. Genome analysis and relational database technology

Genome analysis is a broad field of research involving many different techniques to use, perspectives to consider and questions to answers. In this chapter, we start with a brief overview on genome analysis in Section 2.1 and provide basic concepts and terms. Then, we discuss the detection of genetic variants based on next-generation sequencing (NGS) data, which is an emerging topic within genome analysis demanding for efficient and reliable storage and processing infrastructures. In Section 2.2, we provide details about the variant detection process, which essentially consists of two steps: detecting genetic variations and investigating their consequences. We also present related data-management challenges. In Section 2.3, we present our concept for integrated variant detection based on a DBMS and explain how it can address the data-management challenges within the variant detection process. These considerations lead us to an answer to our first research question: *Which steps of variant detection should we integrate into a DBMS?* Furthermore, we introduce related work on managing and analyzing genome data using DBMSs.

2.1 Genome analysis

In this section, we give an overview on the broad field of genome analysis to classify the topic of this thesis, variant detection, and to introduce basic terms and concepts.

Genome analysis is the study of organisms' genetic information. Genetic information encodes the instructions that are used in living organisms to control their growth, reproduction and cell functioning. This *code of life* is stored in deoxyribonucleic acid (DNA) molecules.

The term *genome analysis* refers not only to one single analysis process or definition, but is a collective term for diverse analyses within the research field of genomics. For

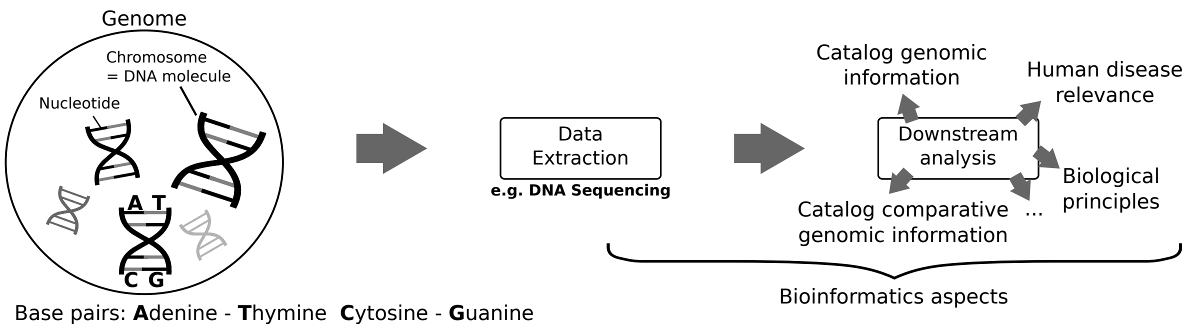


Figure 2.1: A general genome analysis process: Bioinformatics aspects are crosscutting concerns to enable, facilitate and improve genome analysis.

example, Pevsner introduces five different perspectives on genomics that all relate to the analysis of genomes [105, p. 701]:

1. **Catalog genomic information** such as size, number of chromosomes, chemical composition, number of genes or coding and noncoding regions. The genomic information is the base for the further perspectives on genomics.

2. **Catalog comparative genomic information** of genomes, which allows for deducing about the function of parts of yet unknown genomes by comparing them with known genomes. In addition, relationships between species can be established based on genome divergence.

3. Investigate **biological principles** to reveal what functions an organism has and how the genome encodes these functionality. Part of this perspective is also how information stored within DNA comes to life.

4. Investigate the **human disease relevance** of variations of the genome. This also includes studies about how variations can occur and to detect and document variations that are related to diseases.

5. Consider **bioinformatics aspects** such as efficient storage and processing of genome-related data. Moreover, tools for visualizing and navigating genome(-related) data are of interest.

The five perspectives introduced by Pevsner can be mapped to the general genome analysis process that we depict in Figure 2.1. In the following, we define basic terms and explain the general genome analysis steps.

The genome. It all starts with the physical *genome* that is the complete collection of an organism's *deoxyribonucleic acid (DNA)*. DNA is a molecule that consists of two strands made up of a sequence of *nucleotides*. These *nucleotides* are molecules each consisting of a deoxyribose molecule, a phosphate group and one of four nucleo*bases*, either cytosine (C), guanine (G), adenine (A), or thymine (T). *Chromosomes* are DNA molecules that carry the specific traits of living organisms. *Chromosomes* consist of coding regions, i.e., *genes*, and non-coding regions, i.e., sequences of bases with no or yet unknown function. The sequence of bases of *genes* represents the active genetic code. Always three consecutive bases, called codon, can be translated into an amino acid. Amino acids are the basic building blocks of proteins that enable growth, functioning and reproduction of living organisms. Organisms can have chromosome sets, i.e., there is more than one chromosome encoding the same gene. Each of these chromosomes in a set can have a different DNA sequence. In case organisms such as humans have sets of two chromosomes, they are called *diploid* organisms. Each chromosome in a set can have a differing base sequence resulting in different encodings of the same gene. The *genotype* describes the exact encoding of all chromosomes in a set.

Data extraction. To process and analyze the genetic code stored in the genome, we have to extract it from DNA molecules. One technique to extract data from DNA is *DNA sequencing*. DNA sequencing is the process of making the genetic information encoded within DNA digitally readable. Literally spoken, a DNA sequencer reads the sequence of nucleotides of a DNA strand and returns a sequence of the four characters A, C, T and G indicating the respective base of the read sequence of nucleotides. Other techniques are DNA microarrays that allow for detecting whether a specific base sequence is present in a DNA molecule or not.

Downstream analysis. Finally, downstream analyses start that comprise tasks from the first four perspectives introduced by Pevsner. For example, the genome of different species can be compared to find mutual genetic ancestors. Other analyses aim to reveal the functionality of yet unknown genes. Another emerging topic is the detection of genetic variations, which is an essential preprocessing step for the analysis of disease relevance of genome variations. Moreover, it is used to determine an organism's or cell's genotype and allows for comparisons with other DNA sequences. The fifth perspective introduced by Pevsner, bioinformatics aspects, is a crosscutting concern and provides tools and methods to enable, facilitate or improve the actual analysis. For example, efficient tools for variant detection were developed to reduce the analysis runtime.

2.2 Variant detection based on NGS techniques

Genome analysis comprises various research directions. A promising topic is to investigate the disease relevance of genetic variations based on NGS techniques to better detect and treat diseases [19, 50, 110]. In this section, we describe the variant detection process based on NGS data and present data-management-related challenges that we want to address with the use of DBMS technology.

2.2.1 The variant detection process

The variant detection process comprises four steps that we depict in Figure 2.2. The first three steps comprise the extraction of DNA and the actual detection of variants. Finally, the consequences of found variations will be investigated in a downstream

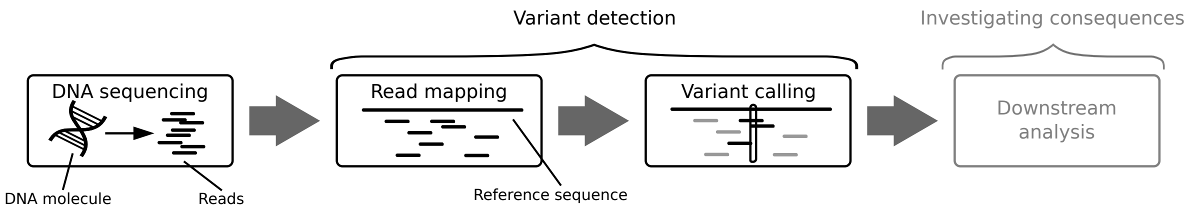


Figure 2.2: Variant detection and analysis process: In this thesis, we focus in the integration of variant detection steps read mapping and variant calling into DBMSs.

analysis step. In this thesis, we aim at integrating the process steps of variant detection into a DBMS. In the following, we give details about DNA sequencing with special focus on NGS techniques, read mapping and variant calling.

2.2.1.1 Next-generation DNA sequencing

The goal of DNA sequencing is to read the base sequence of a given DNA molecule, e.g., a chromosome. Current DNA sequencing methods cannot determine the base sequence of a complete DNA molecule, but only smaller stretches of 100 to several thousand bases depending on the used DNA sequencing technique. A human genome comprises more than 3 billion base pairs. Consequently, it is not possible to sequence whole genomes at once. Thus, a technique called shotgun sequencing is used. We depict the idea in Figure 2.3. In a first step, the DNA molecule under investigation, e.g., a chromosome, is broken up randomly into smaller fragments. Then, these single fragments are sequenced and their base sequence is determined. The small base sequences are called *reads*. Since the DNA fragments are created randomly, the reads must be organized to reconstruct the actual DNA molecule's base sequence.

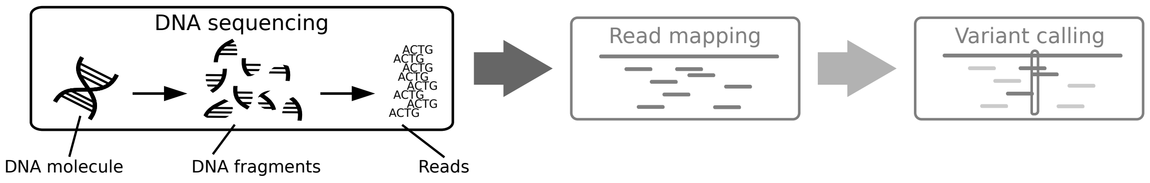


Figure 2.3: DNA shotgun sequencing: Since DNA sequencers can only sequence small DNA stretches, DNA molecules are broken up and the single fragments are sequenced.

Within the initial Human Genome Project [62], DNA sequencing based on Sanger sequencing [113] was used. Nevertheless, Sanger sequencing requires too much time and is too costly to sequence complete genomes on a large scale. For example, the human genome project cost over 2.5 billion dollars and took over ten years[1] to complete a first version of the human genome.

In the mid 2000's, faster and cheaper sequencing techniques were introduced, so called next-generation sequencing techniques [94]. These techniques allow for sequencing DNA

[1]https://www.genome.gov/11006943/

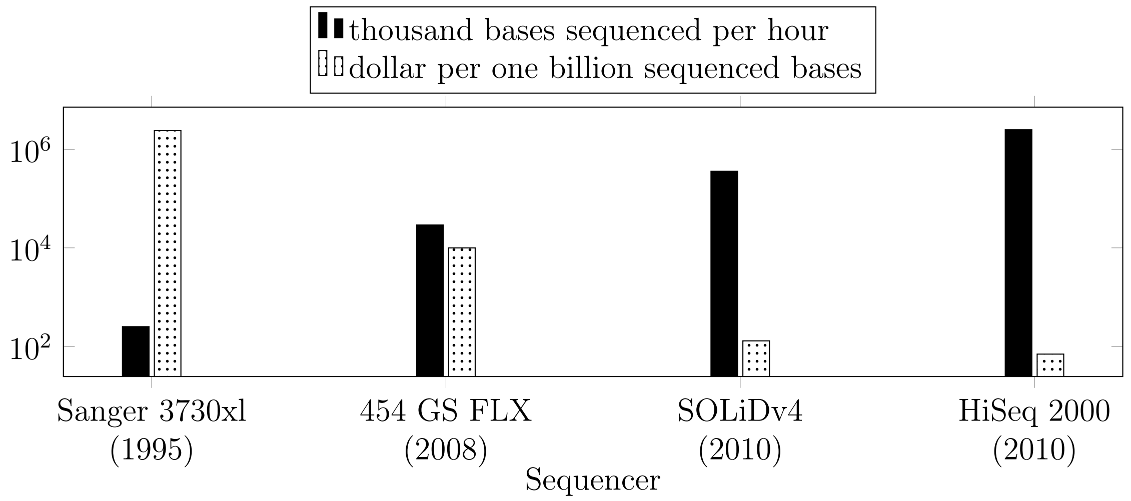


Figure 2.4: Sequencing throughput and costs by sequencer (derived from [82])

molecules in a high-throughput manner [82]. In Figure 2.4, we show a comparison of an automated Sanger sequencing machine (Sanger 3730xl) with selected NGS systems. For example, an Illumina HiSeq2000 sequencer is able to sequence 2.5 billion bases per hour, which is 10,000 times more than the 252 thousand bases per hour of the automated Sanger sequencing machine. The high throughput also allows to decrease the costs of sequencing. Nowadays, systems are available that can sequence a complete genome at the cost of $1,000 [130].

Between the different NGS techniques, we also observe differences regarding throughput and costs. The reason for these differences are the different technologies used within the systems. The used technology also has impact on the characteristics of the generated reads. For example, some approaches are able to generate longer reads or make less errors during the reading process. Other improvements of NGS techniques are paired-end reads. A paired-end read consists of two DNA sequences representing the two ends of the original DNA stretch to be sequenced. It is possible to fix the distance between these two DNA sequences, which can be leveraged to decide about a proper read mapping that we describe in the next section. A detailed discussion of different NGS approaches and their advantages and disadvantages can be found in [95].

2.2.1.2 Read mapping

Read mapping is the process of reorganizing reads generated by DNA sequencers to reconstruct the base sequence of the original DNA molecule. The term *mapping* refers to the procedure of *mapping* reads to a reference sequence. Thus, read mapping requires a reference sequence such as the human reference sequence that was created by the Human Genome Project [62]. Without a reference, reads are processed to find overlaps which can be assembled to bigger base sequences until the complete sequence is reconstructed. This approach is called DNA assembly.

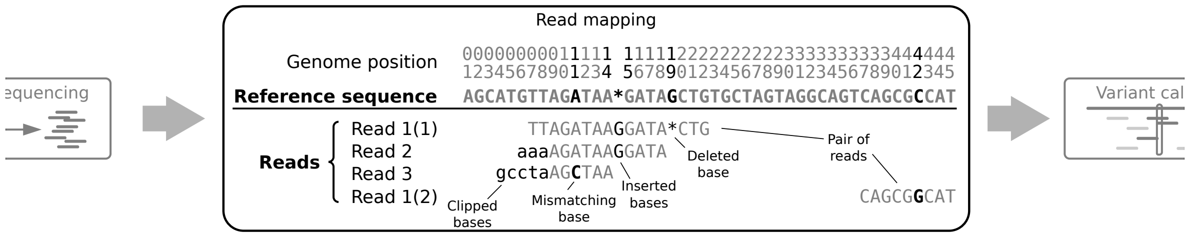


Figure 2.5: Result of a read mapping: Gray bases and positions match completely. At other genome positions, the reads and the reference sequence differ. These differences can be real or result from wrong read mapping or erroneous bases from DNA sequencing.

The challenge of read mapping is to find the correct position of a read, despite erroneous bases within the read, genome variations that exist in the original sample genome and are not present in the reference sequence, and multiple matching sites. To find mappings of one sequence (read) within another reference sequence, local alignment algorithms such as the Smith-Waterman [120] algorithm can be used. The standard algorithm finds the optimal local alignment, but has a runtime complexity of $\mathcal{O}(nm)$, where n indicates the read length and m the reference sequence length. Thus, searching for alignments of millions or billions of reads is inefficient. More efficient approaches such as the Basic Local Alignment Search Tool (BLAST) use heuristics to provide better runtime performance [5]. Nevertheless, the tool is intended to search for single query sequences in a set of reference sequences (database) to identify those reference sequences that match best. To this end, it indexes the query sequence and searches within the database. Since the use case for read mapping is to find a mapping of millions or billions of small reads within a much larger reference sequence, read mapping using BLAST is not a good choice with regard to reasonable runtime [129]. Consequently, special read mapping approaches were developed that explicitly tackle the problem of read mapping. Many of them index the large reference sequence to speed up the search for the mapping of individual reads [78].

In Figure 2.5, we show an example of a read mapping of 3 reads. *Read 1* is a paired-end read. *Read 1(1)* has a single inserted base (between position 14 and 15) and a single deleted base (at position 19) compared to the reference sequence. *Read 1(2)* representing the other end of the read has a mismatching base at position 42. The inserted base of *Read 1(1)* is also present in *Read 2*. In addition, *Read 2* has three clipped bases in the beginning. These were ignored by the read mapper as it found enough matching bases in the rest of the read. *Read 3* also has clipped bases in front. Additionally, *Read 3* has a mismatching base at position 11.

The result of read mapping is used to derive information about the genome. Usually, a first step is to analyze whether genetic variations exist compared to the reference sequence. This step is called variant calling.

2.2.1.3 Variant calling

In the variant calling step, we determine variances within genomes compared to a reference sequence. Variations can be detected for single individuals as well as populations. Population analysis is used to detect variations that are common and appear in several individuals. According to Haraksingh and Snyder, we distinguish three general types of variations within a genome [51]:

- **Structural variations**
  Structural variations refer to variations of the structure of DNA molecules such as chromosomes. These variations are large in size and comprise copy number variations or inversions:
  - Copy number variations: A copy number variation consists of deletions or insertions of parts of a DNA molecule.
  - Inversions: An inversion is present if the sequence of bases within an DNA molecule such as a chromosome is reversed.
- **Small insertions and deletions**
  In contrast to structural variations, small insertions and deletions have a size of 1 to 50 base pairs. In case that the event is not a multiple of 3 and located within a gene coding region, the variation leads to a frame shift having severe impact on the expression of genes (cf. Section 2.1).
- **Single nucleotide variants (SNVs) and Polymorphisms (SNPs)**
  SNVs are exchanges of bases at single genome positions in one genome compared to another and can have a severe impact on organisms life. SNVs can indicate a predisposition for diseases such as cancer [91, 116, 133] or may have impact on drug efficacy by interfering with the metabolism [46]. A special class of SNVs are SNPs. SNPs are exchanges of bases at single genome positions that occur quite frequently within a population. Thus, a SNV can be a SNP but does not have to be. Moreover, it is likely that SNPs are better researched, because they occur more often.

We focus our research on the detection of SNVs within single genomes, because their impact as well as their detection is well researched and the process is established [21, 41]. Moreover, the basic mechanisms behind SNV detection are also needed for SNP detection. In the following, we explain common strategies to detect SNVs from NGS read data.

Details on SNV calling

The task of SNV calling comprises to determine the genotype of a sample (cf. Section 2.1) and to compare it with the respective base in a given reference sequence [96]. In the case that the genotype and the reference base differ, a SNV is found that could

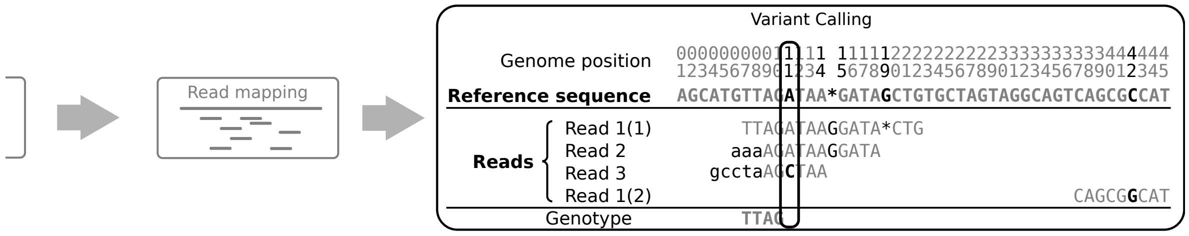


Figure 2.6: During SNV calling, we process all bases that map to the same genome position to derive a genotype. The genotype is compared with the reference sequence.

be a mutation with severe impact. The challenge is to determine the genotype from multiple NGS reads that cover a genome position. In Figure 2.6, we depict the general concept of SNV calling. The bold bar highlights the current genome positions that we want to analyze regarding SNVs. All bases that are mapped to this genome position have to be processed to call a genotype. The term *call* is used to emphasize that the detection of a genotype and finally a SNV depends on chosen heuristics and parameters. Two general approaches exist to derive a genotype from NGS reads [96]:

Frequency approach

The frequency approach simply counts all bases at a genome position and uses simple cut-off rules to decide on a genotype. In Figure 2.6, position 11 is of interest, because the bases of two reads are equal to the reference and one base of one read is different. We assume a diploid organism. Thus, we have pairs of chromosomes and the possible genotypes indicated by the given bases could be AA, AC or CC. The first and the last genotype are called homozygous, because both chromosomes of the pair have the same base at the genome position. AC is called heterozygous, because the chromosomes have different bases at the genome position. Using simple cut-off rules, we could decide for one of the possible genotypes. Common cut-off rules are 20% and 80%. If a base has a frequency of more than 80%, we call a homozygous genotype. If a base has a frequency below 20%, we do not consider it at all. If the frequencies are between 20% and 80%, we call a heterozygous genotype. Especially, homozygous genotypes can be easily called as variant or not by simply comparing them with the reference base. In case of heterozygous genotypes, the decision is not that easy and depends on the actual genotype. In our example, we would call a heterozygous genotype.

A frequency approach is especially useful in case of high coverage regions, i.e., regions that have many overlapping reads. Nielsen states that a coverage of 20 is sufficient [96].

Probabilistic approach

A probabilistic approach takes further information than base frequencies into account. Each base value within a read is associated with an error value indicating the chance that the base is wrong. Usually these error probabilities are in the range below 1%. An error probability of 1% is already a reason to not trust this base and to not consider it during SNV calling. This is one way to improve the results of frequency callers by filtering for high quality bases before genotype detection.

A probabilistic approach would not remove this base completely, but weights the impact of each base according to its error probability. The idea is to compute a posterior probability $p(G|B)$ for every possible genotype G at a specific genome site given a set of bases B aligned to this site. For diploid organisms, we can assume genotype $G \in AA, AC, AG, AT, CC, CG, CT, GG, GT, TT$. The genotype with the highest posterior probability is called to be the resulting genotype. Afterwards the comparison with the reference sequence is done similar to the frequency approach. Moreover, the posterior probability can be used as quality measure for the SNV call.

Nevertheless, we can not directly compute the depending probability $p(G|B)$. However, we can use Bayes' theorem [8] to compute it:

$$p(G|B) = \frac{p(B|G) * p(G)}{p(B)}$$

$p(B|G)$ is the genotype likelihood, the probability to call a set of bases B indicating a genotype assuming a specific genotype G. To determine this probability, we can use the error values of each base. The probability $p(G)$ is the prior probability for genotype G. There are different strategies how to initialize these probabilities. One approach is to use equal probabilities for all genotypes. For example, having 10 possible genotypes, every genotype has a prior probability of $^1/_{10}$. More sophisticated strategies estimate genotype and allele frequencies from multiple genome data sets [74] or incorporate knowledge about the probability of base exchanges in combination with the given reference base to derive prior probabilities [79]. For example, genotypes containing the reference base get higher probabilities than those that do not contain it.

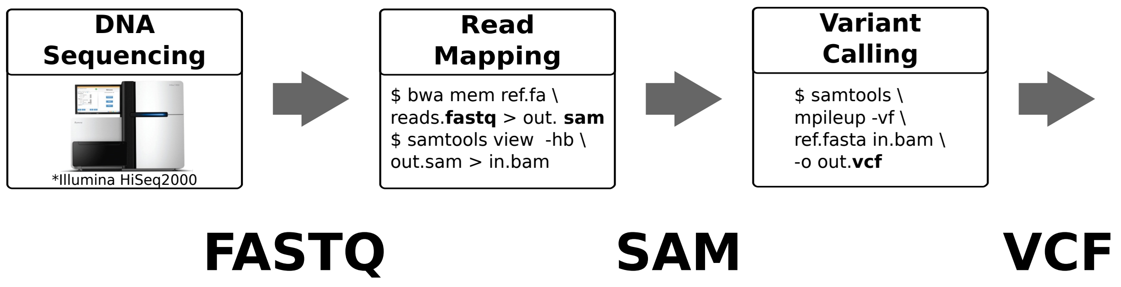
2.2.2 Data-management-related challenges

In this section, we discuss challenges within the variant detection process due to NGS techniques and the current state-of-the-art of managing and processing NGS data. We focus on challenges related to data management and do not consider technical challenges such as how to detect variants in noisy data sets.

Efficient and effective data management

Current variant detection processing using NGS data is flat-file based and commandline driven. In Figure 2.7, we depict typical flat-file formats used within variant detection. FASTQ files are used to store raw reads. The sequence alignment/ map (SAM) format encodes read mappings. Variant Calling Format (VCF) files store the result of variant detection.

Although flat files provide several advantages such as easy sharing of data and require low effort to store the data, they complicate efficient and effective data management:



```
$ bwa mem ref.fa \
reads.fastq > out. sam
$ samtools view  -hb \
out.sam > in.bam
```

```
$ samtools \
mpileup -vf \
ref.fasta in.bam \
-o out.vcf
```

Figure 2.7: The state-of-the-art variant detection pipelines are based on flat-file storage and rely on commandline-driven processing.

High effort for efficient data access
Specialized analysis tools have to manage efficient access to data items within flat files. Therefore, they have to incorporate indexing strategies and must be aware of compressed data. This increases the effort to access large data sets efficiently.

Inefficient file-based processing
Usually, flat-file based analysis incorporates different tools that operate sequentially on the data. In worst case, one tool writes its complete output to another file that is consumed by another tool [32]. *Streaming* could mitigate the effort, because results do not have to be written to disk but just handed over to the next tool. Nevertheless, tools have to be able to work on streams.

Limited data integrity and consistency
File systems provide limited capabilities to check data integrity and consistency. Checksums can be computed for complete files, but a more fine-grained control is not possible, because the file system has only knowledge about files not data stored within the files. For example, inconsistency between related data fields cannot be tracked.

In order to address these challenges, advanced data management approaches are required that provide convenient and efficient access to the data and keep control over data access as well as integrity at all time.

Analysis reliability and reproducibility

The broad use of genome analysis techniques within research and practice leads to an increased demand for reliability and reproducibility of analysis results [9, 112]. This requires to document used parameters, tools and filtering criteria of every analysis step. For example, many different specialized analysis tools exist to perform the steps of read mapping and variant calling. Established tools for read mapping are bowtie [71] or bwa [76]. Both use Burrows-Wheeler indexes to speed up read mapping. Typical variant calling tools are SAMTOOLS/ BCFTOOLS [77] and the Genome Analysis

Toolkit (GATK) [92]. Besides these tools, many other tools exist that put special focus on calling specific variations or use different approaches to determine mappings and variations [99, 100]. For example, the basic SNV detection approaches introduced in Section 2.2.1.3 already provide many options to influence the outcome of the analysis. Consequently, studies have shown that the results from different variant detection pipelines, i.e., combinations of specialized analysis tools, lead to results that have low concordance [99]. Thus, a comprehensive documentation of the analysis process is required to track the analysis results and to make use of them.

Integrated storage and analysis on a large scale

NGS technologies allow for generating vast amounts of genome data fast and cheaply [82]. This requires efficient storage, access and processing techniques. Moreover, the data must be integrated with other data sources to increase its value. For example, variant calls must be combined with gene annotation data to find out which genes are affected by a variation. At the same time, it would be helpful to verify the variant call by drilling down to the actual raw sequencing data. Furthermore, the computation of alternative analysis results based on a different parametrization or approach could help to reject or accept hypothesis on demand. Currently, analysis steps are bulk-oriented and separated. Moreover, many approaches are available that consider the data integration aspect, but only a few focus on analysis integration (cf. Section 2.3.3).

2.3 A concept for DBMS-based variant detection

In this section, we discuss how DBMSs could address the challenges within the variant detection process presented in Section 2.2.2. Then, we introduce a general concept for variant detection using a DBMS.

2.3.1 Why a DBMS?

DBMSs aim to introduce data independence between storage and application layer. This allows them to provide a logical view on the data and optimize the physical data representation for efficient access or reduced storage consumption. Since a DBMS is placed between applications and the actual data, it can offer further services such as integrity and consistency control, access management or declarative data access. In the following, we explain how these features could address the single challenges from Section 2.2.2.

Efficient and effective data management

The decoupling of storage and application layer enables the use of declarative query languages such as SQL. Using a declarative data-access interface frees analysis applications to manage efficient data access themselves. Instead, the application declares what data it needs and the DBMS manages the efficient data access. Moreover, the DBMS has control over the data at any time to ensure its consistency and integrity.

Analysis reliability and reproducibility

DBMSs provide data provenance features that allow for tracking information about the origin of data items, the steps leading to an intermediate data item and the parameters used to generate the analysis result. In case that all analysis functionality is available within the DBMS, a comprehensive tracking would be possible. In the case that we can express the analysis task using a declarative query language such as SQL, we can inherently provide a description for documentation purposes.

Integrated storage and analysis on a large scale

DBMSs are designed to provide access to large amounts of data. Moreover, data integration features are available allowing to combine and access data from different data sources. In combination with a declarative query language and efficient data processing, on-demand analysis workflows could be supported.

There are two directions to introduce the DBMS idea into the variant detection process. On the one hand, we could build it from scratch based on existing techniques and approaches (option A). On the other hand, we could adapt the variant detection process and integrate it into an existing DBMS (option B). In this thesis, we investigate *option B* and aim to integrate variant detection into a relational DBMS.

Especially relational DBMSs are well researched and offer techniques and mechanisms required to address the data-management-related challenges within variant detection. Nevertheless, bioinformatics and scientists claim that relational DBMSs are not capable to allow for efficient analysis of genome data and in particular NGS data. Moreover, it is said that the relational database model and genome analysis do not fit well [67] likely leading to suboptimal performance. However, the renaissance of column-oriented main-memory storage for relational DBMSs that lead to tremendous speedups within analysis applications [60] makes us confident to successfully challenge these prejudgments, especially regarding analysis performance. These physical optimizations in combination with an appropriate database design should enable us to provide competitive performance compared to specialized analysis tools. Nevertheless, we acknowledge that not every analysis step is suited for integration into a database system. In the following, present our concept to integrate variant detection into a DBMS.

2.3.2 Integration concept

To use a relational DBMS within the variant detection process, we have to answer our first research question: *Which steps of variant detection should we integrate into a DBMS?* Integration of an analysis task into a DBMS means to us that we are able to perform the analysis task on demand using a declarative query language such as SQL, instead of externally computing and storing static results. This should inherently provide an analysis description for documentation purposes improving reproducibility of the analysis process complemented by the data integrity and integration features of a

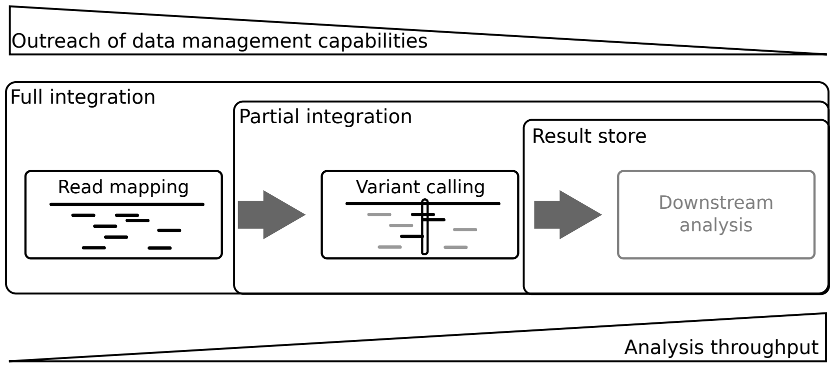


Figure 2.8: Trade-off between providing analyses on demand or storing analysis results. If we just store the results, we gain no improvement regarding data management throughout the complete analysis process. In contrast, if we provide all analysis steps on demand, we likely reduce the analysis throughput.

DBMS. Moreover, the DBMS can leverage the knowledge about data characteristics and processing steps to improve the query execution and analysis runtime. Furthermore, the computation of results on demand enables greater flexibility for the user, since the user can decide about parameterizations or algorithms to use at query time. We can consider three different scenarios to integrate an analysis process into or with a DBMS:

Result store We just store the results of an analysis process in a database and use the DBMS to provide efficient access.

Full integration We map all analysis tasks to the processing engine of the DBMS and just store the raw data.

Partial integration In case that not all tasks can be integrated into the DBMS or storing the raw data is not reasonable, we only integrate parts of the analysis tasks into the DBMS.

2.3.2.1 Trade-off

Deciding for one of the scenarios is a trade-off between analysis throughput and the outreach of improved data management capabilities. If we integrate all analysis steps into the DBMS, we benefit from advanced data management capabilities throughout the complete analysis process, but we are also required to compute analysis results inside the DBMS at runtime, which likely lasts longer than just storing the result. On the other hand, if we just store the final results, e.g., the variant calls, we gain fast access to the results, but we do not gain any improvement with regard to data management throughout the complete analysis process.

In the context of variant detection, we have to consider two analysis steps: read mapping and variant calling (cf. Section 2.2.1). In the following, we assess the three different scenarios in the context of the variant detection process. We depict the trade-off in Figure 2.8.

Result store. In this scenario, we store externally generated variant calls in a database and use the DBMS only to provide efficient access to these variant calls. These variant calls can be integrated with further information, e.g., about gene coding regions, ontologies and molecular pathways, to facilitate downstream analyses such as gene annotations. Furthermore, the DBMS provides efficient access via declarative query languages over an integrated database schema. However, such an approach relies on external information, since we generate variant calls outside the database. Consequently, reliability and reproducibility of analysis results rely strongly on the availability of metadata and the ability to exchange it between external tools and the DBMS.

Full integration. To improve the reliability and reproducibility of the overall analysis process, we could think of integrating all variant detection related tasks into the DBMS. Thus, we would benefit from the data integration capabilities of the DBMS and would also use it to perform analyses internally. However, considering the task of read mapping, we observe that the processing of a read mapping always requires to process the complete underlying data set. Depending on the data set size, which easily reaches several gigabytes, we will not be able to provide analysis results on demand in reasonable time if we would always compute the read mapping based on raw sequencing data. Moreover, raw sequencing data has little use for other analysis tasks than read mapping. Thus, read mapping is a necessary preprocessing step.

Partial integration. Since a full integration of variant detection, which requires to perform read mapping inside the DBMS, does not allow for on demand analysis, we could think about storing mapped read data in a database and integrate the process step of variant calling into the DBMS. Considering the concept behind variant calling, we can use the DBMS to efficiently retrieve the data, e.g., a genome region of interest or a specific sample genome, to perform the actual calling. Enabling researchers to call variants on demand using a specific parametrization instead of importing the variant calls facilitates the analysis process. It allows for what-if analyses as well as the combination of different results. Applying this scenario, we would still benefit from the data management capabilities of DBMSs to provide efficient and effective data management for integrated genome data. Moreover, we would perform an integral step of variant detection inside the DBMS improving analysis reliability and reproducibility.

2.3.2.2 Design decision

Based on our assessment in the previous section, we choose the partial integration scenario as best trade-off between reduced analysis throughput and gained data management improvements. In Figure 2.9, we depict the concept. We aim to integrate variant calling as internal DBMS functionality complementing and leveraging existing DBMS functionality such as access control, data compression or query optimization. Read mapping is done externally and the mapped reads are stored within the database. Thus, we can manage *and* analyze the mapped reads using the DBMS. In particular,

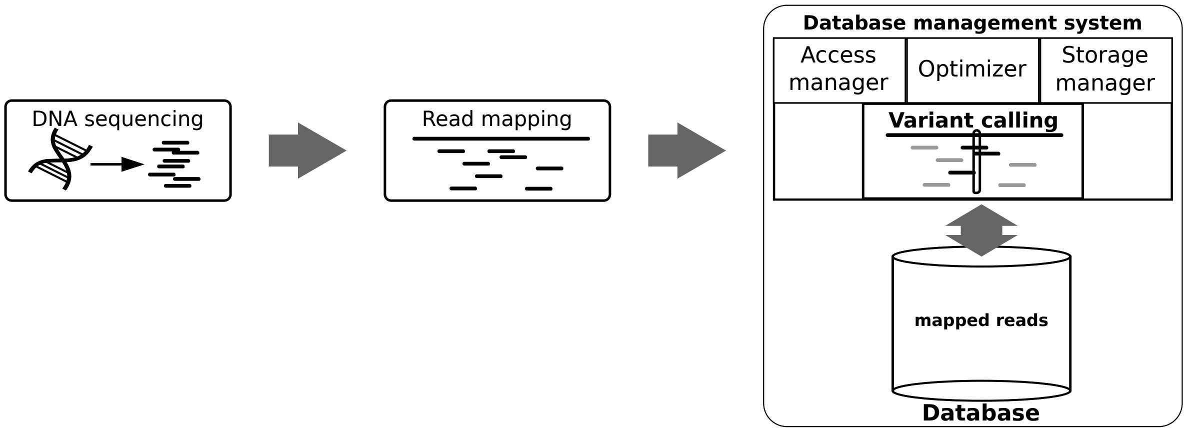


Figure 2.9: Next-generation sequence analysis using DBMSs: We aim to store mapped read data within the database and integrate variant calling as internal functionality.

we focus on SNV calling using relational database operators as a basis allowing us to express the analysis via SQL. In the following section, we provide an overview on related work for storing, integrating and analyzing genome data using DBMSs and show how our work complements existing approaches or integrates with them.

2.3.3 Related work

DNA sequencing became widely applicable using the Sanger method at the end of the 1970's [113]. In the beginning, researchers had to familiarize themselves with the technology, define data formats and methods for analysis. With the start of more sophisticated sequencing projects such as the Human Genome Project [62] in 1990 more sophisticated data management and analysis technologies were required to achieve the goal to decode the complete human genome. Since the theory of relational database systems was already introduced in 1970 [25] and matured within the next decades, relational DBMSs were and are still a method of choice for managing genome data.

In this section, first, we provide an overview on approaches that mainly store and integrate genome data, such as raw sequencing data or mapped read data, with other data sources storing annotations such as gene positions or functions. We distinguish application-driven approaches and data warehouse toolkits. In particular, the data warehouse toolkits rely on relational DBMSs to facilitate the integration and analysis of preprocessed data. These toolkits are highly related to the idea of a *result store* discussed in Section 2.3.2. Then, we describe approaches that perform preprocessing, such as read mapping, and the analysis of genome data, such as variant calling, directly within a DBMS or use database technology to do so. We distinguish approaches for sequence similarity search or read mapping and mapped read analysis including variant calling. Finally, we show how our work complements and integrates with existing approaches for storing, integrating and analyzing genome data using relational DBMSs.

2.3.3.1 Approaches for data integration

DBMSs offer data integration functionality that is required in many application scenarios. In this section, first, we consider approaches designed for specific applications within the context of genome analysis. Then, we present an overview on general data warehouse approaches for managing and integrating genome(-related) data.

Application-driven approaches

The *Nucleic Acids Research database issue* provides an overview on existing databases available within genome research and related fields [107]. It currently lists 1685 biological databases that are designed for specific analysis use cases. Some systems are dedicated to the integration of data of a specific genome such as the human genome (genome databases) or the storage of a specific type of data such as sequencing data. Not all of these databases rely on a DBMS. Instead they are simple flat file repositories allowing for online access to retrieve information. Here, we present a brief overview on sequence databases and genome databases and discuss how database technology is used within these use cases.

Management of sequencing data. To provide access to publicly available DNA sequences, systems to collect and manage sequence data from various sequencing experiments were established. For example, the *Gen*etic Sequence Data *Bank* (GenBank) [20] was designed to store genome sequence data and associated annotation information. In the early days, the data was maintained in a relational DBMS to benefit from advantages such as schema modeling and data integrity checking. While the relational DBMS provides a solid solution to store and maintain the data, several special-purpose tools were built around the database system that allow for data exchange, online access and interaction with the data. For example, the export of data was and is mainly implemented using flat files allowing users to convert the data in any needed format, e.g., importing it into a local relational database system [29].

Current releases of GenBank indicate that no relational DBMS is used anymore to maintain the sequencing data [24], but a specialized tool suite was built around the existing flat files. This system setup is similar to other databases such as the European Molecular Biology Laboratory's Nucleotide Sequence Database [124].

These systems focus on the storage of DNA sequences of genomes, but not DNA sequences from NGS technologies, i.e., reads. To this end, the Sequence Read Archive (SRA) was established [66]. It allows to submit different types of data such as raw sequences and mapped reads using flat-file formats and manages these files.

Genome databases. Genome databases manage the data of specific genomes to facilitate their analysis. For example, the Genome Data Base (GDB) [102] developed in the 1980's was used to facilitate the process of gene mapping that allows for insights about which genes are involved in diseases that are transmitted from parents to children. Furthermore, gene mapping can provide information about which chromosome contains which gene and where this gene lies on that chromosome. The software was based on the relational DBMS Sybase. To allow for easy integration of new methods for gene mapping, the GDB was designed to allow for easily adding new types of information without redesigning the entire database. The database did not store sequence data itself, but referred to data items stored in GenBank.

While GDB focused on the complete human genome, other databases exist that store data about microbial genomes [89] or consider only single chromosomes [73]. For example ACeDB [122] (*A C*aenorhabditis *e*legans *D*ata *B*ase) was developed to store and maintain sequencing data and annotation information for worms. The main requirement was to create a database system that has a flexible data structure to adjust it if new experience requires it. In contrast to GDB, ACeDB is an object database system developed from scratch. The object model allows users and developers to flexibly create individual objects and add or remove attributes. Relational DBMSs have a rather static model making it difficult to achieve this goal. On top, ACeDB provides sophisticated graphical views on the data that allow for navigation and manipulation of the data. The unique combination of a visual interface and a flexible data storage backend might be the reason why ACeDB is still listed as active genome database.

Nevertheless, technology behind relational DBMSs evolved and the systems matured. For example, the Ensemble project [58] relies on a relational DBMS such as MySQL or PostgreSQL, because flat-file storage requires deep knowledge about how to access data efficiently [121]. Moreover, the limited scalability of ACeDB led to the choice for using a relational DBMS. Ensemble provides a comprehensive overview on genomes as well as an automatic annotation pipeline. The project stores various kinds of information such as assembled DNA sequences and sequencing project related information, but also computed features, e.g., alignment results, and predicted genes from the annotation pipeline. The user has the possibility to access data via web, can download it as flat files or use special software libraries that provide access to the database by encapsulating SQL commands.

Data warehouse toolkits

Inspired by application-driven databases and fostered by the ongoing standardization process within genome analysis, data warehouse toolkits were proposed. These approaches aim to provide a general infrastructure for managing, accessing and integrating biological data. In Figure 2.10, we depict the general architecture of data warehouse toolkits especially for biological use cases. The goal is to provide uniform access to different biological data sources. The data sources comprise sequence databases as well as databases about known variations [118] or genome information such as gene coding regions. Since genetic variations affect gene products and interfere metabolism [46], databases providing protein information can be integrated allowing researchers to analyze these effects. To integrate these different kinds of data, a uniform schema is used and data-source-specific loaders extract, transform and load the data into the data warehouse. Finally, the user can access the integrated database using plain SQL, application programming interfaces (APIs) or specific retrieval tools that can also provide data format conversion.

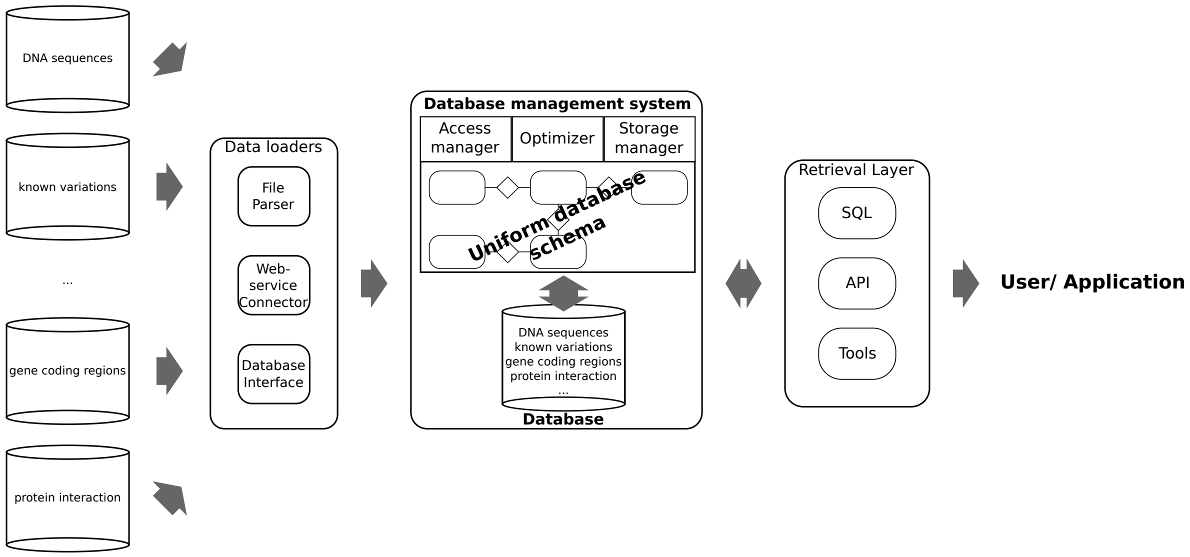


Figure 2.10: Data warehouse toolkits provide the necessary infrastructure such as data loaders and access mechanisms to integrate biological data from different data sources. The schema and tools are generic to support a wide range of projects or applications.

An example for a data warehouse toolkit is the Genome Information Management System (GIMS) that uses an object database and integrates genome sequence data with functional information [101]. Initially, the system was developed for analyzing yeast genomes, but the developers took care to model the schema without too much genome-specific characteristics allowing to apply it to other projects. Besides data integration, the system design focuses on data analysis. To this end, specialized analysis routines that directly incorporate the object schema were developed [27].

Other approaches, such as BioWarehouse [72], Atlas [115] or BioDWH [128], focus on using relational database systems to facilitate the data integration task and to allow for multi-database queries via SQL. While BioWarehouse focuses on expressing analysis tasks using SQL, Atlas also provides a set of advanced retrieval applications that allow for querying data and also provide data conversion functionality into common flat-file formats. Additionally, Atlas provides an API for programming languages such as C++ and Java. The API encapsulates SQL commands and allows for convenient access from within programs. BioDWH provides similar functionality using object-relational mappers such as the Hibernate[2] framework. Instead of adapting the API to different database systems, Hibernate encapsulates the characteristics to communicate with different relational databases facilitating the exchange of database systems.

2.3.3.2 Approaches for DBMS-integrated genome analysis

So far, we considered approaches that store and integrate genome data and (partly) use DBMSs for this purpose. These approaches have shown to provide new insights by

[2]http://www.hibernate.org/

allowing researchers to query an integrated database. Another advantage of DBMS-integrated analysis, is that the analysis is performed directly at the data site reducing export and transfer overhead. The data integration approaches considered so far focus on downstream analysis steps. In this section, we focus on approaches that integrate primary analysis steps of NGS data into DBMSs, e.g., read mapping and variant calling. We focus our overview on approaches using relational DBMSs or technology.

Read mapping and sequence similarity search. The task of read mapping is to find the best matching position of a short DNA sequence, a read, within a large DNA sequence, the reference (cf. Section 2.2.1.2). The problem is known as local alignment and an algorithm to compute the best matching position is the Smith-Waterman Algorithm [120]. Nevertheless, the complexity of this algorithm is too large to scale to large data sets. An optimized approach is the Basic Local Alignment Search Tool (BLAST) [5] using small seeds that get extended to find optimal local alignments. Besides exporting data from a relational DBMS to benefit from its query capabilities and calling BLAST as external commandline tool [83], approaches were proposed that directly integrate the BLAST functionality as table-valued UDFs into a relational DBMS [57, 109, 123]. While BLAST provides sufficient performance for sequence similarity use cases such as finding homologous sequences between species, the alignment of millions and billions of NGS reads in reasonable time would be challenging [129].

To this end, more specific approaches were proposed that are highly tuned for the mapping of short reads against a long reference sequence of the same species and implemented within specialized commandline tools [78]. A straightforward approach to integrate such functionality into a DBMS was proposed by Schapranow and Plattner in the High-performance In-memory Genome (HIG) project [114]. The HIG platform uses bwa [76] to perform the alignment task on (chunks of the) unmapped reads that reside in the database. In contrast, the Wisconsin's High-throughput Alignment Method (WHAM) provides read mapping functionality using indexing and optimization techniques inspired by database systems [81].

Mapped read analysis and variant calling. After reads have been mapped, they are analyzed. A typical first analysis is variant calling. We are only aware of one approach by Fähnrich et al. using a DBMS that addresses this task. Fähnrich et al. store mapped read data in a main-memory DBMS and perform SNV calling [43]. They report competitive runtime performance compared to state-of-the-art tools. To achieve this, Fähnrich et al. propose a special database schema storing the sequences of single chromosomes of a reference genome in separate tables. The actual processing is a MapReduce-like approach. To the best of our knowledge, approaches using only a relational processing stack have not incorporated variant calling so far. Cijvat et al. introduce an approach that allows to compute coverages within mapped read data sets [23]. Röhm and Blakeley use the commercial DBMS SQL Server 2008 that allows to process genome data stored within flat-files using the file wrapper functionality of SQL Server 2008. In addition, the authors show how to query unique reads and consensus sequences [108].

Name	Focus	Remarks
Application-driven data integration approaches		
Genbank [20]	DNA sequence storage	flat-file based storage
AceDB [122]	Data integration	uses an object DBMS
Ensemble [58]	Data integration	uses a relational DBMS
SRA [66]	Storage of NGS data	flat files based storage
Data warehouse toolkits		
GIMS [27]	Data integration	uses an object DBMS
Atlas [115]	Data integration	uses a relational DBMS
BioWarehouse [72]	Data integration	uses a relational DBMS
BioDWH [128]	Data integration	uses a relational DBMS
DBMS-integrated genome analysis		
BLAST external [83]	Sequence similarity search	exports data and calls BLAST externally
BLAST internal [109]	Sequence similarity search	calls BLAST via a UDF
WHAM [78]	Read mapping	inspired by DBMSs
HIG [114]	Read mapping	integrates the bwa alignment tool
Consensus calling [108]	Mapped read analysis	allows for querying flat-file data via SQL
SNV calling [43]	Mapped read analysis	MapReduce-like data conversion and processing

Table 2.1: Overview on related work to store, integrate and analyze genome data: Raw sequencing data including read mappings are usually stored using flat file repositories. Data integration is already supported using DBMSs. The integration of analysis tasks already started, but SNV calling is not yet available as relational DBMS functionality.

2.3.3.3 Inference

In Table 2.1, we summarize our literature overview on data integration and analysis approaches using DBMSs, in particular relational ones. The overview on application-driven data integration approaches shows that relational database technology is used for managing and integrating various kinds of information that are required for the analysis of genomes. Nevertheless, storing raw sequencing data, especially data from NGS experiments is mainly done using flat-file repositories. Within our concept, we aim at storing read data directly within a relational DBMS to allow for its efficient processing and storage using the DBMS.

The reviewed data warehouse toolkits show that database systems and especially relational database systems are tools of choice to integrate different data sources for

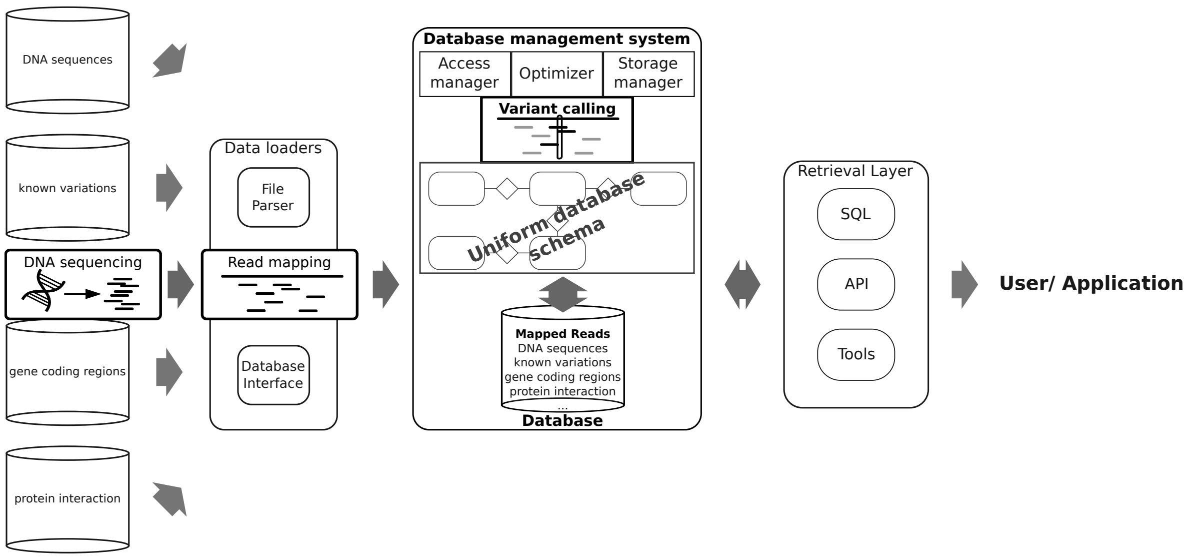


Figure 2.11: Integrated NGS analysis using DBMSs: We want to develop methods and techniques for storing mapped read data and performing variant detection within DBMSs that can be integrated with existing data warehouse approaches.

biological use cases. Our proposed concept for DBMS-integrated variant detection integrates well with these data warehouse approaches. In Figure 2.11, we show a possible integration scenario. Mapped reads are an additional data source that are integrated with other data sources. On top, the variant calling functionality is part of the DBMS and can be executed via different user interfaces.

Furthermore, our review shows that several approaches exist that integrate genome analysis tasks such as similarity search, read mapping or the analysis of mapped reads into a DBMS. Our initial concept from Section 2.3.2 aligns well with these approaches. The HIG [114] and WHAM [78] approach show possible solutions to perform read mapping as preparation tasks, for example during the import of data. Moreover, the approach by Fähnrich et al. shows that SNV calling in a main-memory DBMS is competitive to state-of-the-art analysis variant calling tools such as GATK [92] with regard to runtime performance [43]. However, Fähnrich et al. use a MapReduce-like processing paradigm. In our work, we focus on providing SNV calling functionality via relational database operators to benefit from existing relational DBMS technology (cf. Section 2.3.2).

2.4 Wrap up

In this section, we gave a high-level overview on the field of genome analysis that comprises many research directions. A promising research field is to improve disease detection and treatment by investigating genetic variations. A cost and time-effective method to detect genetic variations is based on next-generation sequencing (NGS). We introduced the necessary analysis steps to derive genetic variations from NGS data and explained related data-management challenges within the process.

A first challenge is the current data management method of NGS data itself, because it mainly relies on flat-file repositories that cannot guarantee consistency and integrity. With increasing amounts of NGS data, reliable data repositories get more and more important. Further challenges are reliability and reproducibility of analysis results. The data processing is mainly based on flat files and involves many different commandline tools. Thus, it is hard to comprehend, which data contributed how to which analysis result. Moreover, the results of variant detection must be integrated with other data sources. Currently, the process is to generate results and integrate them later. In the future, it could be also advantageous to perform variant detection on demand to verify a hypothesis. Of course, this requires access to the respective data and analysis functionality.

DBMSs are known to provide excellent data management capabilities that could be used to address these challenges. Nevertheless, relational DBMSs lack support for scientific and, in particular, genome analysis functionality. To this end, we proposed a concept to integrate variant detection into a relational DBMS and answered our first research question: *Which steps of variant detection should we integrate into a DBMS?* We decide to store mapped read data in a DBMS, and integrate variant calling as internal DBMS functionality. A review of related work has shown that this approach can be integrated into existing data warehouse approaches for biological data to provide an efficient, reliable and integrated analysis platform for the analysis of NGS data.

3. Relational storage and analysis of read mapping data

In this chapter, we investigate approaches to store and analyze mapped reads using a relational DBMS and answer our second research question: *How can we express variant detection using relational DBMS operators as a basis?*

In Section 3.1, we give details about flat-file-based storage and analysis of mapped reads. Then, in Section 3.2, we introduce a basic approach to store and analyze mapped reads using a relational database system. The idea is to store reads as strings. Each character within the string represents a single base. This approach resembles state-of-the-art flat-file-based storage and analysis. However, existing string operators do not provide the necessary functionality to perform genome analysis tasks, which forces users to define their own UDFs. These UDFs tend to encapsulate much of the analysis logic making them specialized programs run inside a database system. Such complex UDFs are less transparent limiting the possibilities to optimize their execution automatically by the DBMS. To this end, in Section 3.3, we present an extended database schema that avoids the need for complex UDFs at all to perform SNV calling within a relational database system. Finally, we present related work regarding the declarative analysis of mapped reads.

3.1 A primer on file-based storage and processing of mapped reads

In this section, we describe how state-of-the-art commandline tools process mapped reads stored in flat-files to detect genetic variations. First, we introduce quasi-standard flat file formats. Then, we briefly explain the general processing approach.

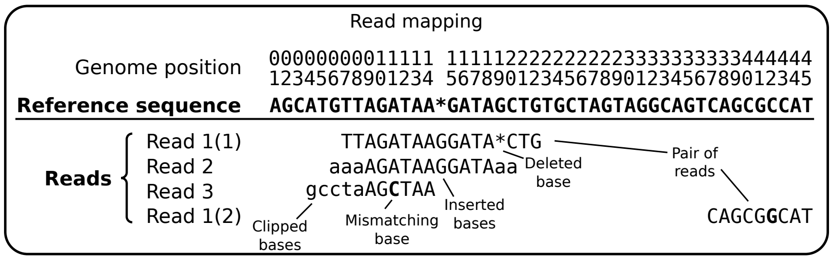


Figure 3.1: To encode the mapping of reads, it is not sufficient to just store the reads as string of characters and the start position of the mapping, because the single bases within a read may be inserted, deleted or clipped influencing the actual mapping position of the single bases with regard to the reference sequence.

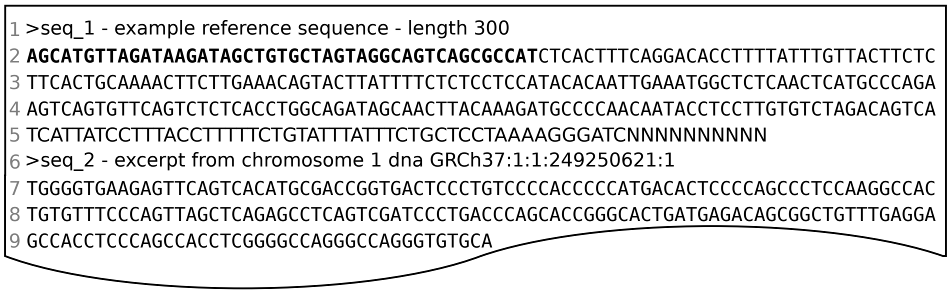
3.1.1 Flat-file formats

In Figure 3.1, we show our running example for this section depicting the mapping of four reads against a reference sequence. Reads are sequences of several hundred to thousand bases. A base is encoded using one of the characters A, C, G or T (cf. Section 2.1). Thus, to store a read mapping, we have to store the read as string encoding the bases and the start position of the mapping. In the case that all bases within a read either match or mismatch (cf. *Read 3* at position 11) their respective reference base, this approach would be sufficient. Nevertheless, the single bases within a read do not have to map properly to the reference sequence. For example, in *Read 2*, base *G* is inserted at position 14 compared to the reference sequence. Other cases are the deletion of bases (cf. *Read 1(1)* at position 19) or the clipping of bases indicating that the read mapper ignored these bases during the read mapping (cf. *Read 3* at position 4 to 8). Thus, these special cases have to be encoded, too. Specific data formats were proposed to efficiently encode read mappings within flat files. In the following, we explain the FASTA format used to store DNA sequences such as reference sequences and the sequence alignment/ map (SAM) format used to encode the mapping of reads.

3.1.1.1 FASTA format

The FASTA format is named after the FASTA program package used for similarity search within protein and DNA sequence databases [103, 104]. In Figure 3.2, we show an example of FASTA encoded DNA sequences. Each sequence has a description line (cf. line 1) giving information about the sequence that follows in the next lines (cf. lines 2 - 5) until another description line (cf. line 6) introduces another DNA sequence (cf. lines 7 - 9). There is no rule what information the description line must contain. Usually, the name of the sequence or the species it belongs to should be reported. Moreover, it is common to report the length of the sequence.

The reference sequence of our running example is part of the first sequence, *seq_ 1*, in the FASTA file. To encode the base sequence, a special set of characters is used. Besides (A)denine, (C)ytosine, (G)uanine and (T)hymine denoting one of the four bases a DNA molecule is made of, the alphabet comprises 12 additional characters [97] denoting cases



```
>seq_1 - example reference sequence - length 300
AGCATGTTAGATAAGATAGCTGTGCTAGTAGGCAGTCAGCGCCATCTCACTTTCAGGACACCTTTTATTTGTTACTTCTC
TTCACTGCAAAACTTCTTGAAACAGTACTTATTTTCTCTCCTCCATACACAATTGAAATGGCTCTCAACTCATGCCCAGA
AGTCAGTGTTCAGTCTCTCACCTGGCAGATAGCAACTTACAAAGATGCCCCAACAATACCTCCTTGTGTCTAGACAGTCA
TCATTATCCTTTACCTTTTTCTGTATTTATTTCTGCTCCTAAAAGGGATCNNNNNNNNNN
>seq_2 - excerpt from chromosome 1 dna GRCh37:1:1:249250621:1
TGGGGTGAAGAGTTCAGTCACATGCGACCGGTGACTCCCTGTCCCCACCCCCATGACACTCCCCAGCCCTCCAAGGCCAC
TGTGTTTCCCAGTTAGCTCAGAGCCTCAGTCGATCCCTGACCCAGCACCGGGCACTGATGAGACAGCGGCTGTTTGAGGA
GCCACCTCCCAGCCACCTCGGGGCCAGGGCCAGGGTGTGCA
```

Figure 3.2: A FASTA file is used to store protein and DNA sequences. The FASTA format distinguishes description lines starting with > from those lines containing the sequence.

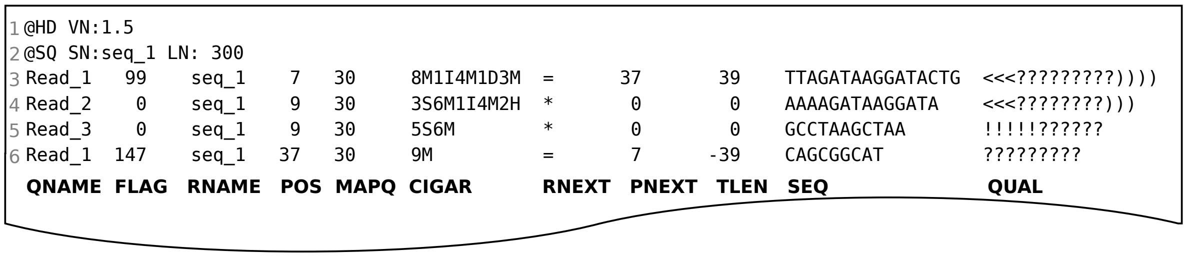
where the actual base is not clear, e.g., “a(N)y”. The actual position of each base is implicitly encoded as position inside the sequence string. The first base is at position 1, the second at position 2, and so forth. But this implicit encoding can be problematic in the case, we store only parts of a complete reference sequence, e.g., the sequence of a gene. The second sequence, *seq_2*, in the FASTA file is an excerpt of chromosome 1 of the human reference genome GRCh37 that has a complete length of 249,250,621 bases. Thus, it is not clear from the implicit encoding whether the first base in the sequence is really the first base in the reference sequence. An additional information in the description line might help but is not mandatory.

3.1.1.2 Sequence alignment/ map (SAM) format

A common file format for storing read mappings is the sequence alignment/ map (SAM) format [77]. In Figure 3.3, we show a SAM file describing the mapping of the four reads of our running example. The file consists of a header (lines 1 - 2) and an alignment section (cf. lines 3 - 6). The header provides meta-information (@HD) such as the used SAM-format version (VN) for compatibility reasons. Furthermore, a reference sequence dictionary (@SQ) lists all used reference sequences by name (SN) and reports their sequence length (LN). It is also possible to assign an optional uniform resource identifier (URI) hinting to a reference sequence file, e.g., a FASTA file on a webserver, or a website containing further details about the reference sequence.

The alignment section stores the actual mappings of each single read. To this end, the SAM format specifies 11 mandatory fields describing the sequence, the mapping and giving information about paired-end reads. These mandatory fields can be complemented with a list of optional fields (not depicted in Figure 3.3). In the following, we describe the fields in the context of NGS read mappings:

General sequence information. The *QNAME* attribute serves as identifier for a read. Note, the *QNAME* is not unique. Reads that have the same *QNAME* belong to the same paired-end read (cf. line 3 and 6 in Figure 3.3). The *SEQ* field contains the base sequence of the read and the *QUAL* field stores a quality value indicating the probability that the base is wrong according to the DNA sequencer. Usually, bases with



```
1 @HD VN:1.5
2 @SQ SN:seq_1 LN: 300
3 Read_1   99    seq_1    7   30     8M1I4M1D3M  =      37       39    TTAGATAAGGATACTG  <<<?????????))))
4 Read_2    0    seq_1    9   30     3S6M1I4M2H  *       0        0    AAAAGATAAGGATA    <<<????????)))
5 Read_3    0    seq_1    9   30     5S6M        *       0        0    GCCTAAGCTAA       !!!!!??????
6 Read_1  147    seq_1   37   30     9M          =       7      -39    CAGCGGCAT         ?????????
  QNAME  FLAG   RNAME   POS  MAPQ  CIGAR       RNEXT  PNEXT  TLEN  SEQ               QUAL
```

Figure 3.3: A SAM file encodes the mappings of reads with 11 mandatory fields.

an error probability below 1% are of interest for further analyses. In order to facilitate the computation with and comparison of rather small error probabilities, the quality values are phred-scaled [42]:

$$Q_{Phred} = -10 * \log_{10} P$$

In the following table, we show the result of the phred scaling for chosen quality values:

phred value	ASCII code	error probability
10	+	10% or 1 in 10
20	5	1% or 1 in 100
30	?	0.1% or 1 in 1000
40	I	0.01% or 1 in 10,000
50	S	0.001% or 1 in 100,000
60	]	0.0001% or 1 in 1,000,000

To make the phred-scaled quality scores human readable, the SAM format adds an offset of 33 allowing for encoding phred-scaled quality values using visible ASCII characters. The second column in the previous table shows the mapping of chosen quality values to ASCII characters. For example, a question mark (*?*) represents a phred-scaled quality value of 30 that is an error probability of 0.1%.

Mapping information. The actual mapping of a single read is encoded using the *FLAG*, *RNAME*, *POS*, *MAPQ* and *CIGAR* field. The *RNAME* field contains the reference sequence the read is mapped to. The given reference name can be looked up in the reference sequence dictionary in the header to get more information about it. *POS* states the start position of the read mapping. *MAPQ* gives the error probability of the read mapping, which is also phred-scaled. The *CIGAR* field encodes the concrete mapping of each base within a read as string. Thus, it accounts for inserted and deleted bases. A *CIGAR* string consists of tuples of a number N and a character C. C indicates the operation that has to be applied to N bases. Important *CIGAR* operations are Insert, Delete, Match and Soft clip. For example, the *CIGAR* string of the first read in our example (cf. line 3) is $8M1I4M1D3M$ meaning that the first 8 bases match the reference sequence. Then, the next (9th) base is inserted followed by four matching, one deleted and three matching bases. A matching base does not mean that the base values are equal to the reference base. For example, the 8th base of the third read in our example (cf. line 5) contains a mismatching base. The *CIGAR* string only states that the first 5 bases are (soft) clipped, i.e., they are ignored (clipped) but not removed from the sequence in *SEQ* (soft), and the last six bases match. Thus, it is not clear whether a matching base is different from the reference or not. The *FLAG* field contains a 16-bit integer used for bitwise encoding of further information about the read mapping. For example, the first bit indicates that the read is part of a paired-end read and the third bit signals whether the read is unmapped. For more details on *FLAG* bits and *CIGAR* operations, we refer the interested reader to the official Sequence alignment/ map Format Specification [111].

Paired-end read information. Finally, there are three fields storing information about paired-end reads that consist of two separately mapped reads having the same *QNAME*. Both reads within a pair are referenced to each other within a SAM file using the *RNAME*, *PNEXT* and *TLEN* fields. *RNAME* indicates the reference sequence that the other part of the paired-end read is mapped to. *PNEXT* gives the start position and *TLEN* the number of bases covered by both ends of the read. *TLEN* is only available, if both reads of a paired-end read are mapped to the same reference sequence.

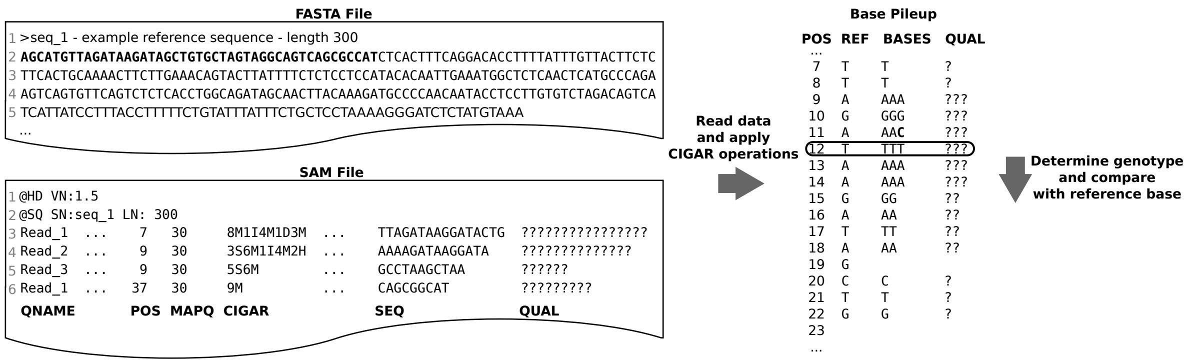
Optional fields. Besides the 11 mandatory fields, a valid SAM row can contain a list of optional fields that are appended at the end. Optional fields allow users and tools to add further information about the read mapping or to store annotations. Optional fields have the format *TAG:TYPE:VALUE*. A *TAG* is a two letter code indicating the meaning of the *VALUE*. *TYPE* is a single letter, encoding the data type of *VALUE*. Several standard tags[1] exist. Some are used to speed up the lookup of information. For example, *R2* is used to store the sequence of the other read within a paired-end read. Thus, the look up of this information does not require to search for the same *QNAME* within the file.

3.1.2 Flat-file based SNV calling

In this section, we consider a typical processing approach to detect genetic variations within mapped reads. In Section 2.2.1.3, we explained two general approaches to perform SNV calling. Both approaches require to have access to the single bases and their related information such as quality scores at a specific genome position. Unfortunately, the reference sequence encoding and the read mapping encoding rely on string representations of DNA sequences. Additionally, the SAM format uses implicit encodings of the concrete mapping. Thus, part of the processing is to convert the data into a suitable format, which allows for efficient access to the single bases of a genome position. In Figure 3.4, we depict a common processing strategy for SNV calling that specialized state-of-the-art analysis tools follow.

Basic processing approach. The sequence data from FASTA and SAM files are combined and converted. The conversion includes to apply CIGAR operations to read

[1]http://samtools.github.io/hts-specs/SAMtags.pdf


FASTA File
SAM File
Base Pileup
Read data and apply CIGAR operations
Determine genotype and compare with reference base

```
>seq_1 - example reference sequence - length 300
AGCATGTTAGATAAGATAGCTGTGCTAGTAGGCAGTCAGCGCCATCTCACTTTCAGGACACCTTTTATTTGTTACTTCTC
TTCACTGCAAAACTTCTTGAAACAGTACTTATTTTCTCTCCTCCATACACAATTGAAATGGCTCTCAACTCATGCCCAGA
AGTCAGTGTTCAGTCTCTCACCTGGCAGATAGCAACTTACAAAGATGCCCCAACAATACCTCCTTGTGTCTAGACAGTCA
TCATTATCCTTTACCTTTTTCTGTATTTATTTCTGCTCCTAAAAGGGATCTCTATGTAAA
...
```

```
@HD VN:1.5
@SQ SN:seq_1 LN: 300
Read_1  ...    7   30     8M1I4M1D3M  ...    TTAGATAAGGATACTG  ????????????????
Read_2  ...    9   30     3S6M1I4M2H  ...    AAAAGATAAGGATA    ??????????????
Read_3  ...    9   30     5S6M        ...    GCCTAAGCTAA       ??????
Read_1  ...   37   30     9M          ...    CAGCGGCAT         ?????????
QNAME         POS  MAPQ   CIGAR              SEQ               QUAL
```

```
POS  REF  BASES  QUAL
...
 7   T    T      ?
 8   T    T      ?
 9   A    AAA    ???
10   G    GGG    ???
11   A    AAC    ???
12   T    TTT    ???
13   A    AAA    ???
14   A    AAA    ???
15   G    GG     ??
16   A    AA     ??
17   T    TT     ??
18   A    AA     ??
19   G
20   C    C      ?
21   T    T      ?
22   G    G      ?
23
...
```

Figure 3.4: To detect SNVs, mapped reads and reference sequences have to be converted into a processing-friendly data layout. Moreover, implicit mapping information encoded within CIGAR strings must be made explicit.

sequences to determine the actual mapping position of each single base. Then, the reads are split and the single bases are stored in a so called *base pileup* that provides direct access to all bases mapped to a specific genome position including their quality scores. This data structure facilitates the computation of genotypes and the comparison with the reference sequence to derive SNVs.

Optimized flat-file processing. Specialized analysis tools that apply this processing approach were developed to operate efficiently on disk-resident data and are optimized for computer systems with small main memory. Working with disk-resident data implies that data is read in chunks from disk and sequential access patterns are required to reduce the disk latency. Having small main memory systems implies that the tools have to reduce the size of intermediate results. Consequently, a straightforward processing approach, i.e., reading data sequentially from disk and creating one large base pileup, will not work due to memory consumption limitations. To overcome this issue, the tools require SAM files to be sorted by reference sequence and mapping position (cf. *RNAME* and *POS* fields in the SAM format). The sorting ensures that mapped reads read from disk in chunks belong to the same genome region. Thus, the tools do not have to maintain a complete base pileup, but only intermediate ones for genome regions that are currently covered by the processed mapped reads. For example, the mapping of the second read in the example SAM file in Figure 3.4 starts at position 9 implying that it does not contribute any bases for genome positions less than 9. Since all reads are sorted by mapping position, this conclusion is true for all subsequent reads in the file. Thus, we can start to determine genotypes for genome position 7 and 8 and compare it with the reference sequence in parallel and do not have to keep this part of the base pileup in main memory.

The SAMTOOLS suite. The SAMTOOLS suite [77] contains the specialized analysis tools SAMTOOLS and BCFTOOLS for variant calling that follow the optimized processing approach from above. SAMTOOLS is responsible for creating the base pileup and computing genotype likelihoods (cf. Section 2.2.1.3). Then, the genotype likelihoods can be streamed to BCFTOOLS that performs the actual SNV calling as well as further statistical analysis [74].

Filtering parameters. The tools provide several options to manipulate the SNV calling result. Besides restricting the analysis to a specific genome region, it is possible to set parameters that have a direct impact on the analysis result. For example, SAMTOOLS allows the user to filter reads that have a mapping quality (MAPQ) below a given threshold. The same principle can be applied to single base values. Moreover, SAMTOOLS filters reads based on the given $FLAG$ information such as reads that are still unmapped, reads that failed previous quality control tests, reads that are marked as duplicate or reads that are part of a read pair that has only one properly mapped read. All these parameters can be adjusted by the user. Moreover, the user can choose whether SAMTOOLS adjusts read mapping qualities or replaces base call quality values by Base Alignment Quality (BAQ) values that have shown to improve the SNV calling quality in the presence of small insertions and deletions [75]. A full list of currently available options in SAMTOOLS can be found at the documentation website[2].

3.1.3 Focus of this thesis

In this thesis, we are interested in the general capabilities of relational DBMSs to store and process read mapping data. The use of a DBMS for storing mapped read data will enhance the data management quality of the genome analysis process. For example, using a relational DBMS, we can directly associate reference sequences and read mappings and avoid the loose coupling present in flat-file-based storage. Our main focus is to provide competitive analysis runtime performance within the DBMS for the core processing functionality behind SNV calling. This core processing functionality comprises the *filtering of reads* based on genome positions and data characteristics and the *processing of bases* per genome position. We do not focus on data manipulation steps, e.g., replacing quality values by BAQs, and other statistical methods as introduced by the SAMTOOLS suite to improve the outcome of SNV calling, because such approaches only complement the core processing functionality. Moreover, according to DePristo et al., adjustments to the data, such as the recalibration of base call quality values in the presence of small insertions and deletions, are not part of the variant detection process, but of the data preprocessing step [30]. Thus, we assume that such data manipulations have already taken place when we query the data.

3.2 SNV calling using relational DBMSs

Existing approaches that enable users to analyze mapped reads and to call variants using DBMSs do not support all steps required for SNV calling or rely on non-relational processing strategies (cf. Section 2.3.3.2). In this section, we investigate approaches to perform SNV calling using a relational database system.

[2]http://www.htslib.org/doc/samtools.html

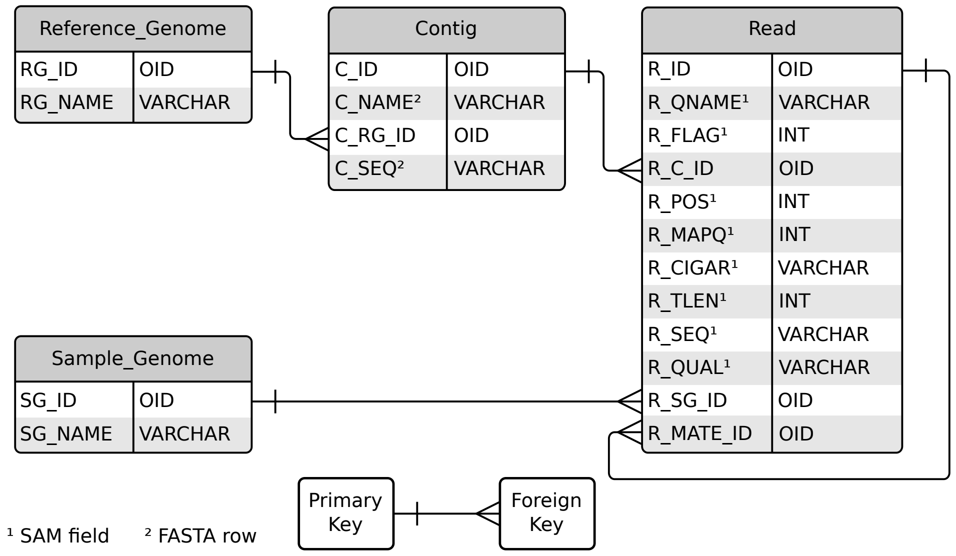


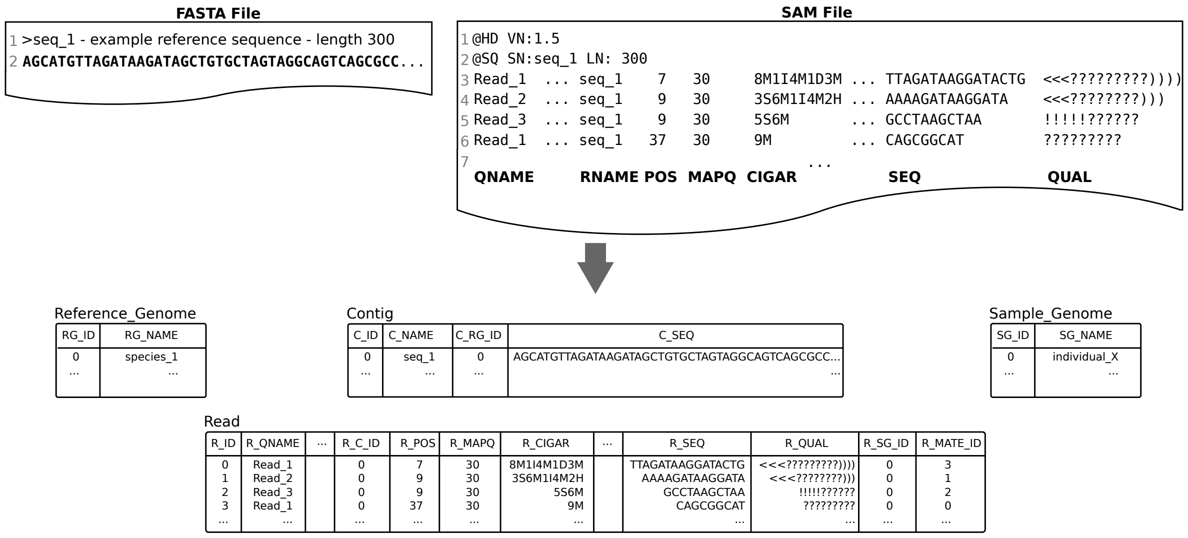
Figure 3.5: The read-centric database schema maps SAM fields and FASTA rows to table attributes facilitating read-centric analyses. In addition, importing and exporting data from and into file formats is straightforward.

First, we introduce a database schema, the *read-centric database schema*, generalized from related work storing data similar to the SAM format. Then, we show how we can analyze mapped reads using SQL based on this database schema. Since the database schema stores mapping information implicit within strings, we cannot directly access single bases per genome position using standard relational database operators. To this end, we require genome-specific UDFs to convert the data. Moreover, existing concepts are not designed to support SNV calling. Thus, we extend the existing concepts on a conceptual level to allow for declarative SNV calling. Finally, we conduct an assessment of the approaches based on the respective literature showing that the processing suffers from large intermediate results and processing overhead introduced by complex UDFs. Increasing main memory capacities mitigate the problems due to large intermediate results. Nevertheless, the processing overhead remains leading us to the idea of a database schema that allows for SNV calling within a relational DBMS without the need for complex conversion UDFs.

3.2.1 The read-centric database schema

A general approach to store mapped read data is to map the SAM fields and FASTA rows to database table attributes. All approaches that we identified in Section 2.3.3.2 for mapped read analysis using a DBMS follow this idea [23, 43, 108]. In Figure 3.5, we depict a generalized database schema. It consists of the four tables *Reference_ Genome*, *Contig*, *Sample_ Genome* and *Read*.

The attributes of table *Read* reflect the mandatory fields of a SAM file. We omit the fields *RNEXT* and *PNEXT* describing information about reference sequence and position of the other read within a *paired-end read* and replace it by a foreign key



```
>seq_1 - example reference sequence - length 300
AGCATGTTAGATAAGATAGCTGTGCTAGTAGGCAGTCAGCGCC...
```

```
@HD VN:1.5
@SQ SN:seq_1 LN: 300
Read_1  ... seq_1    7   30     8M1I4M1D3M ... TTAGATAAGGATACTG  <<<?????????))))
Read_2  ... seq_1    9   30     3S6M1I4M2H ... AAAAGATAAGGATA    <<<????????)))
Read_3  ... seq_1    9   30     5S6M       ... GCCTAAGCTAA       !!!!!??????
Read_1  ... seq_1   37   30     9M         ... CAGCGGCAT         ?????????
                                   ...
```

Reference_Genome

RG_ID	RG_NAME
0	species_1
...	...

Contig

C_ID	C_NAME	C_RG_ID	C_SEQ
0	seq_1	0	AGCATGTTAGATAAGATAGCTGTGCTAGTAGGCAGTCAGCGCC...
...	...	...	...

Sample_Genome

SG_ID	SG_NAME
0	individual_X
...	...

Read

R_ID	R_QNAME	...	R_C_ID	R_POS	R_MAPQ	R_CIGAR	...	R_SEQ	R_QUAL	R_SG_ID	R_MATE_ID
0	Read_1		0	7	30	8M1I4M1D3M		TTAGATAAGGATACTG	<<<?????????))))	0	3
1	Read_2		0	9	30	3S6M1I4M2H		AAAAGATAAGGATA	<<<????????)))	0	1
2	Read_3		0	9	30	5S6M		GCCTAAGCTAA	!!!!!??????	0	2
3	Read_1		0	37	30	9M		CAGCGGCAT	?????????	0	0
...	...		...	...	...	...		...	...	...	...

Figure 3.6: The mapping of genome data stored in FASTA and SAM files is straightforward. Every FASTA line and SAM field is an attribute in the database schema.

relationship to table *Read* itself. Moreover, every read is associated with a tuple from table *Sample_ Genome* containing meta information about the genome's individual or origin such as a describing name. Since we also store reference sequence data in the database, we can directly associate each mapped read with a reference genome. Since a genome consists of separate chromosomes that have their own DNA sequence, we represent the single DNA sequences making up a genome as *Contig*uous sequence. Such a *Contig* is a sequence stored in a FASTA file. We connect table *Read* and *Contig* with a foreign-key relationship to indicate reference sequence the read is mapped to. Each tuple in the *Contig* table belongs to a *Reference_ Genome*. The schema resembles a star schema, where table *Read* is the fact table and all other tables are dimension tables. Thus, most information is centered around read data. For that reason, we call this schema *read-centric database schema*. In the lower part of Figure 3.6, we show how the data of our running example from Figure 3.1 is stored in a read-centric database.

3.2.2 Toward database-integrated SNV calling

We aim to integrate SNV calling into a relational database system to benefit from the internal query optimization and advanced data management mechanisms. Therefore, it is crucial to use existing database operators for analysis, because these are well integrated into the database system stack.

In this section, we describe how we can call SNVs using SQL based on the read-centric database schema. First, we consider how we can filter reads based on mapping characteristics. Then, we explain how we can perform genome-position specific analysis, i.e., processing single bases per genome position. Furthermore, we discuss necessary extension for existing approaches to finally call SNVs.

```
SELECT R_QNAME, R_SEQ
FROM Read JOIN Sample_Genome
        ON R_SG_ID = SG_ID
WHERE SG_NAME = "human1"
    AND R_MAPQ > 29;
```

Listing 3.1 Filter reads of genome *human1* that have a high mapping quality.

```
SELECT R_QNAME, R_SEQ
FROM Read JOIN Sample_Genome
        ON R_SG_ID = SG_ID
WHERE SG_NAME = "human1"
    AND (R_FLAG & 4) != 0;
```

Listing 3.2 Filter reads of genome *human1* that are unmapped.

Read filtering. Since the read-centric database schema is centered around reads, it allows for convenient analysis of read-related data such as genome position, mapping quality or *FLAG* information via SQL. It is also possible to query reads that have a deletion or insertion using the *like* operator. In Listing 3.1 and Listing 3.2, we show two read-centric SQL queries. The first query filters reads of a sample genome called *human1* that have a mapping quality (R_MAPQ) greater 29. These can be considered as high quality reads. The second query filters reads of the same sample genome that are unmapped according to the read mapper. To this end, we perform a check whether the third bit is set within the *FLAG* value using a bitwise AND operation and checking whether the result is not zero [111].

These kinds of queries are interesting for filtering mapped reads and selecting those for further analysis. In case genome annotation information would be available within the database, we could also query for all reads that cover a specific gene for further processing.

Nevertheless, analyses that consider single genome positions are harder to implement, because the actual position information is implicitly encoded. Thus, genome-specific analysis functionality is required. In Listing 3.3, we show an SQL query to filter all reads of genome *human1* that cover the region of 1,000 to 2,000 of chromosome 1 of the human reference genome. To this end, we have to check which reads have at least one end laying inside the interval (cf. lines 6 - 7) or overlap the complete interval (cf. line 8). To enable such a query, the genome-specific UDF *CIGAR_LENGTH* is required that computes the length of a mapped read according to the CIGAR operations. If we would simply determine the length of the sequence string (R_SEQ), we would ignore inserted and deleted bases that impact the overall mapped read length.

Position-specific analysis. Now that we know, how we can query reads that overlap a specific genome region, we might be interested in the read coverage within this region. The coverage indicates the number of reads that overlap a specific genome position. This is important to retrieve further information about the quality of analysis results such as SNVs. Studies have shown that an average coverage of 30 is needed to detect homozygous and heterozygous SNVs reliably [119]. However, coverage can vary over the genome, especially if we filter low quality reads or exclude single bases.

In order to compute the coverage using the read-centric database schema, we have to count how many reads overlap a single genome position. The problem is to determine

```
SELECT R_QNAME, R_SEQ
FROM Read JOIN Sample_Genome ON R_SG_ID = SG_ID
    JOIN Contig ON R_CID = R_ID
    JOIN Reference_Genome ON C_RG_ID = RG_ID
WHERE SG_NAME = "human1" AND RG_NAME = "human" AND C_NAME = "chromosome1"
    AND (R_POS >= 1,000 AND R_POS <= 2,000)
    OR (R_POS + CIGAR_LENGTH(R_CIGAR) BETWEEN 1,000 AND 2,000)
    OR (R_POS < 1,000 AND R_POS + CIGAR_LENGTH(R_CIGAR) > 2,000);
```

Listing 3.3: Filter reads of genome *human1* that overlap region 1,000 to 2,000 of *chromosome1* of the *human* reference genome.

```
1 SELECT s.value AS pos, COUNT(*) AS coverage
2 FROM Read JOIN Sample_Genome ON R_SG_ID = SG_ID
3     JOIN Contig ON R_CID = C_ID
4     JOIN Reference_Genome ON C_RG_ID = RG_ID
5     /* returns a sequence of continuous numbers in the requested range */
6     JOIN generate_series(1000,2000) AS s
7         ON s.value BETWEEN R_POS AND R_POS + CIGAR_LENGTH(R_CIGAR)
8 WHERE SG_NAME = "human1" AND RG_NAME = "human" AND C_NAME = "chromosome1"
9 GROUP BY pos;
```

Listing 3.4: Computing coverages according to [23].

the right grouping attribute. Simply using R_POS does not work as it only indicates the position of the first base of the read. A solution could be to shrink the interval to a single genome position and to count all reads that overlap the single genome position. Obviously, this introduces much processing overhead, because we execute the query 1,001 times instead of once.

Cijvat et al. propose to join the reads with an artificial table representing the genome region of interest [23]. In Listing 3.4, we depict the idea. The UDF *generate_series* creates a sequence of continuous numbers in the range of 1,000 to 2,000 representing the single positions of the genome region of interest (line 6). Then, the reads fulfilling the filter predicates are joined with this sequence of numbers. The join predicate checks whether the current position is covered by the mapped read (line 7) using the same UDF to evaluate the CIGAR as in Listing 3.3. Grouping by *pos* (line 9) and counting the reads that were joined with the same genome position results in the coverage (line 1).

Another approach is to use table-valued UDFs that split the reads of interest into single bases allowing for easy aggregation. For example, Röhm and Blakeley show how to integrate such functionality into SQL Server 2008 [108]. In Listing 3.5, we show the SQL query adapted to fit our general database schema. Commercial database systems such as SQL Server or Oracle provide a *CROSS APPLY* operator that is similar to an *INNER JOIN*, but accepts table-valued UDFs. The *CROSS APPLY* operator can apply the rows of the left side to the table-valued UDF and combines each left row with all rows generated by the table-valued UDF on the right side. The UDF introduced by

```
SELECT pos, COUNT(base) AS coverage
FROM Read JOIN Sample_Genome
        ON R_SG_ID = SG_ID
    JOIN Contig ON R_CID = C_ID
    JOIN Reference_Genome ON C_RG_ID = RG_ID
    /* returns (pos, base, qual) tuples */
    CROSS APPLY PivotAlignment(R_POS,R_SEQ,R_QUAL,R_CIGAR)
WHERE SG_NAME = "human1" AND RG_NAME = "human" AND C_NAME = "chromosome1"
    AND R_POS BETWEEN 1,000 AND 2,000
    OR R_POS + CIGAR_LENGTH(R_CIGAR) BETWEEN 1,000 AND 2,000
    OR (R_POS < 1,000 AND R_POS + CIGAR_LENGTH(R_CIGAR) > 2,000)
GROUP BY pos;
```

Listing 3.5: Computing coverage and calling consensus bases (genotypes) (adapted from [108]).

```
1 SELECT pos, ref_base, COUNT(base) AS coverage,
2     CallBase(base, qual) AS genotype
3 FROM Read JOIN /* join clause (see Listing 3.5) */
4     /* returns (pos, base, qual, ref_base) tuples */
5     CROSS APPLY PivotAlignment(R_POS,R_SEQ,R_QUAL,R_CIGAR,C_SEQ)
6 WHERE /* Region and other filters (see Listing 3.5) */
7 GROUP BY pos, ref_base
8 HAVING genoype <> ref_base
9 ORDER BY pos;
```

Listing 3.6: Calling SNVs with the approach by Röhm and Blakeley [108].

Röhm and Blakeley to use with *CROSS APPLY* is *PivotAlignment*. *PivotAlignment* converts a given read (from the left side) into $(pos, base, qual)$ tuples based on the given read sequence (R_SEQ, R_QUAL), mapping position (R_POS) and CIGAR string (R_CIGAR). This allows us to group reads by *pos* and *COUNT* bases per genome position.

Röhm and Blakeley originally introduced this approach to call a consensus sequence. To this end, they introduced a UDA *CallBase* that takes the single base and quality values per genome position and derives a consensus base per genome position. Concatenating the consensus bases of all positions results in a consensus sequence. Our use case is related. We are also interested in consensus bases per genome position that are genotypes, but for the purpose of finding differences to a given reference sequence. What is missing in the approach by Röhm, is to incorporate reference sequence data. To this end, we could extend the *PivotAlignment* function to also convert the reference sequence. In Listing 3.6, we depict the idea. *PivotAlignment* additionally consumes the reference sequence (C_SEQ) belonging to the read and returns tuples of $(pos, base, qual, ref\_base)$ (line 5). Then, we can use a *HAVING* clause to compare the computed genotype (line 2) with the reference base and filter those positions that

```
 1 SELECT pos, ref_base, CallBase(base, qual) AS genotype
 2 FROM
 3     (SELECT s.value AS pos,
 4         _RefBase(s.value, C_SEQ) AS ref_base,
 5         _Base(s.value, R_SEQ, R_POS, R_CIGAR) AS base,
 6         _Qual(s.value, R_QUAL, R_POS, R_CIGAR) AS qual
 7     FROM Read JOIN /* join clause (see Listing 3.4) */
 8     /* returns a sequence of continuous numbers in the requested range */
 9         JOIN generate_series(1000,2000) AS s
10             ON s.value BETWEEN R_POS AND R_POS + CIGAR_LENGTH(R_CIGAR)
11     WHERE /* Filter predicates (see Listing 3.4) */)
12 GROUP BY pos, ref_base
13 HAVING genoype <> ref_base
14 ORDER BY pos;
```

Listing 3.7: Calling SNVs adapted from the approach by Cijvat et al. [23].

do not differ (line 8). The final *ORDER BY* clause returns the found SNVs sorted by position (line 9), since specialized analysis tools also provide sorted outputs.

We can extend the approach by Cijvat et al. in a similar way. In Listing 3.7, we depict necessary extensions to call SNVs. The inner query extracts all genome-position related information required for SNV calling: the reference base (line 4, UDF *_RefBase*) and the information about each base and its quality per read (line 5 – 6, UDFs *_Base* and *_Qual*). Then, in the outer query *CallBase* aggregates all bases of all reads that overlap the genome region of interest (line 9). Finally, we use the same *HAVING* clause as in Listing 3.6 to filter SNVs (line 13).

3.2.3 Qualitative assessment

Röhm and Blakeley state that their approach leads to large intermediate results due to the *PivotAlignment* function. Instead of storing position information implicitly within the read sequence, i.e., using strings, these are now made explicit and must be materialized. To reduce the overhead, Röhm and Blakeley introduce another UDF called *AssembleConsensus()*, which encapsulates the complete query and makes the assumption that reads are sorted by genome position. Thus, reads overlap during processing and can be split and aggregated interleaved, which reduces the intermediate result size. This approach is similar to state-of-the-art analysis tools such as SAMTOOLS (cf. Section 3.1.2).

Furthermore, the authors state that parallelizing their approach requires additional logic. Since we operate on already sorted data, a suitable parallelization strategy is to process contiguous genome regions in parallel. Thus, we have to make sure that we do not miss overlapping reads during chunking. Apart from this additional logic that must be integrated into the database system, we already hide a lot of analysis functionality within the specialized UDF. Consequently, we end up with a specialized analysis tool integrated into the DBMS. This limits the portability and comprehensibility of the approach and sacrifices the advantages of a declarative query processing engine.

In the last decade, the amount of available main memory within computer systems increased by one order of magnitude [55] allowing to keep much larger amounts of data, e.g., mapped reads (complete databases) or intermediate results, in memory. This removes the need for highly specialized UDFs that rely on sorted reads as suggested by Röhm and Blakeley, but also requires the use of well designed DBMSs to benefit most from increasing amounts of main memory and modern parallel computer architectures [125]. For example, Cijvat et al. use the column-oriented main-memory DBMS MonetDB for analyzing NGS data. Their work shows that data conversion is the most time consuming task besides data import within their mapped read analysis process [23]. Thus, they suggest to cache the conversion result to speedup subsequent analyses, which becomes a valid option due to increasing main-memory capacities.

Overall, we conclude that the approach proposed by Cijvat et al. is more promising to provide a competitive SNV calling solution based on relational database technology. The highly complex UDF introduced by Röhm and Blakeley might lead to competitive analysis runtime, but sacrifices the benefits of a DBMS with regard to data management capabilities such as self-optimizing and flexible declarative query processing. The runtime conversion proposed by Cijvat et al. is an alternative, but introduces processing overhead due to the conversion. Thus, we raise the question how can we store read mapping information explicitly removing the need for data conversion during runtime at all. Then, we do not have to execute conversion UDFs and should be able to query genome data mainly using relational database operators. Such an approach also frees resources and should allow us to keep more mapped read data in memory, because we do not have to cache converted data additionally.

3.3 A pileup approach for relational SNV calling

In the previous section, we extended existing approaches using a read-centric database schema for mapped read analysis to support SNV calling. The qualitative assessment revealed that the need for data conversion introduces inherent overhead and requires caching strategies to reduce it. In this section, we introduce the *base-centric database schema* that can be seen as relational implementation of a base pileup (cf. Section 3.1.2). The aim of our design is to enable users to perform position-specific analysis steps such as SNV calling without the need of UDFs for data conversion. To this end, we store mapping information already explicit within the database allowing for convenient access to single bases using standard database operators. Thus, we can perform SNV calling requiring only a genome-specific user-defined aggregation function (UDA). Finally, we discuss benefits and challenges of our proposed approach for relational SNV calling.

3.3.1 The base-centric database schema

The base-centric database schema centers all information around the single bases of mapped reads. We depict the schema in Figure 3.8. The schema consists of six tables: *Reference_ Genome*, *Contig*, *Reference_ Base*, *Sample_ Genome*, *Read* and *Sample_ Genome*. Conceptually, we extend the read-centric schema (cf. Section 3.2.1) by the two tables *Sample_ Base* and *Reference_ Base*.

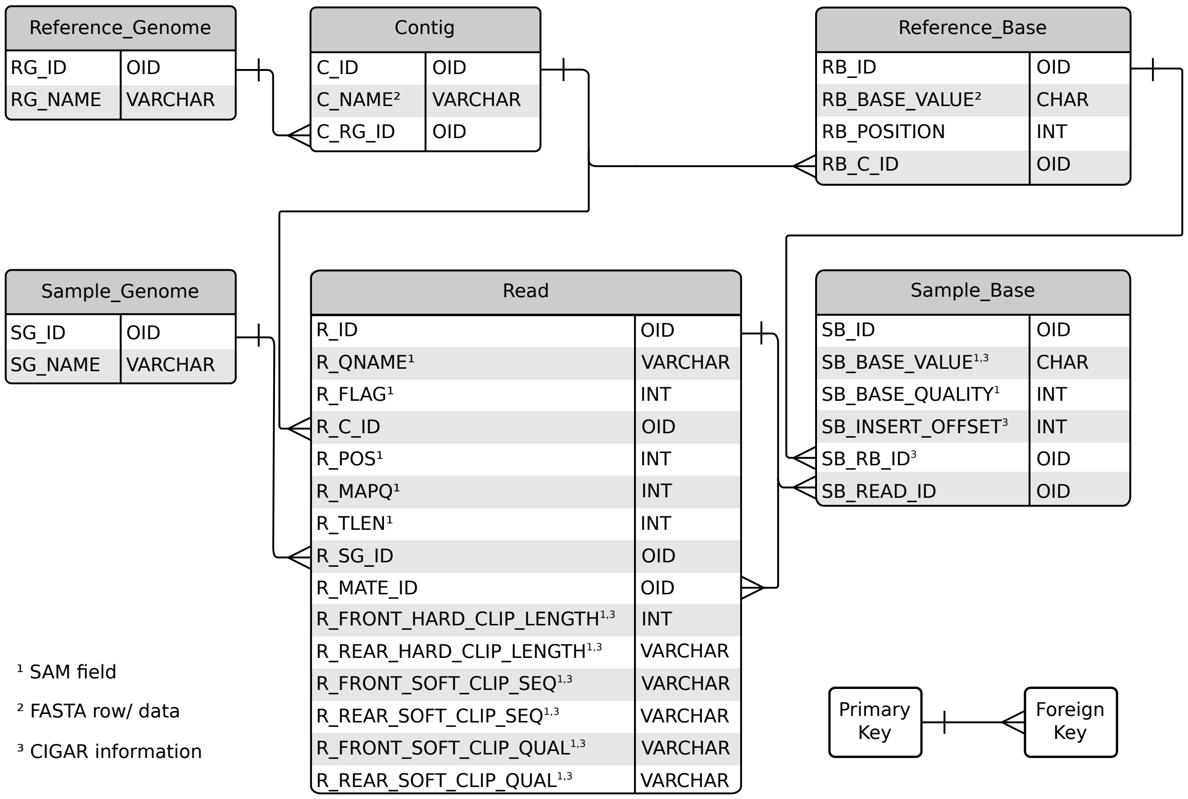
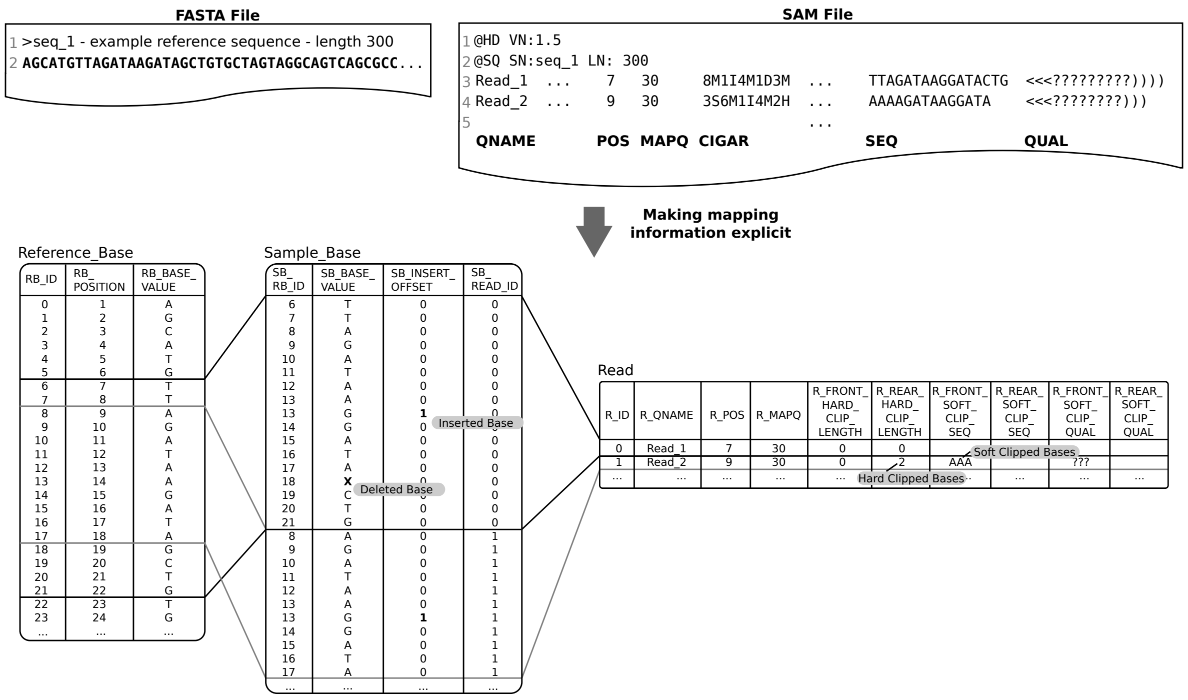


Figure 3.8: The base-centric database schema stores mapping information formerly encoded within CIGAR strings explicit facilitating base- and position-centric analyses.

The table *Reference_ Genome* provides meta data about reference genome data sets. A reference genome consists of several *Contig*uous regions. In contrast to the read-centric database schema, we do not store the reference sequences in the *Contig* table as strings, but store the single bases including their position information in the new table *Reference_ Base*. Moreover, we keep a foreign-key (RB_C_ID) for every reference base referencing the contiguous region it belongs to.

Table *Sample_ Genome* contains meta data about NGS data sets. Such data sets consist of many reads that are stored in table *Read*. In contrast to the read-centric database schema, we avoid to store read sequences as strings, but store every single base in table *Sample_ Base*. Moreover, we establish a direct connection between each mapped sample base and the corresponding reference base using a foreign-key relationship (SB_RB_ID). In contrast to the straightforward conversion of reference sequences into single reference bases, we have to consider the implicit mapping information encoded in the CIGAR string of each read when converting the base sequence into single bases. To this end, we extend table *Read* and *Sample_ Base* with further attributes. We use these additional attributes to make the implicit information encoded in a CIGAR string explicit. In Figure 3.9, we provide an example of the conversion of the first two reads of our running example from Figure 3.1. We only show attributes of tables *Reference_ Base*, *Sample_ Base* and *Read* that are used to store the map-



```
>seq_1 - example reference sequence - length 300
AGCATGTTAGATAAGATAGCTGTGCTAGTAGGCAGTCAGCGCC...
```

```
@HD VN:1.5
@SQ SN:seq_1 LN: 300
Read_1  ...   7   30    8M1I4M1D3M  ...   TTAGATAAGGATACTG  <<<?????????))))
Read_2  ...   9   30    3S6M1I4M2H  ...   AAAAGATAAGGATA    <<<????????)))
                                    ...
QNAME        POS  MAPQ  CIGAR              SEQ               QUAL
```

RB_ID	RB_POSITION	RB_BASE_VALUE
0	1	A
1	2	G
2	3	C
3	4	A
4	5	T
5	6	G
6	7	T
7	8	T
8	9	A
9	10	G
10	11	A
11	12	T
12	13	A
13	14	A
14	15	G
15	16	A
16	17	T
17	18	A
18	19	G
19	20	C
20	21	T
21	22	G
22	23	T
23	24	G
...	...	...

SB_RB_ID	SB_BASE_VALUE	SB_INSERT_OFFSET	SB_READ_ID
6	T	0	0
7	T	0	0
8	A	0	0
9	G	0	0
10	A	0	0
11	T	0	0
12	A	0	0
13	A	0	0
13	G	**1**	0
14	G	0	0
15	A	0	0
16	T	0	0
17	A	0	0
18	**X**	0	0
19	C	0	0
20	T	0	0
21	G	0	0
8	A	0	1
9	G	0	1
10	A	0	1
11	T	0	1
12	A	0	1
13	A	0	1
13	G	**1**	1
14	G	0	1
15	A	0	1
16	T	0	1
17	A	0	1
...	...	...	...

R_ID	R_QNAME	R_POS	R_MAPQ	R_FRONT_HARD_CLIP_LENGTH	R_REAR_HARD_CLIP_LENGTH	R_FRONT_SOFT_CLIP_SEQ	R_REAR_SOFT_CLIP_SEQ	R_FRONT_SOFT_CLIP_QUAL	R_REAR_SOFT_CLIP_QUAL
0	Read_1	7	30	0	0				
1	Read_2	9	30	0	2	AAA		???	
...	...	...	...	...	...	...	...	...	...

Figure 3.9: The mapping information is distributed over several attributes within the base-centric database schema to enable direct access to data at single genome-positions.

ping information and the base sequences. The lines between table *Reference_ Base* and *Sample_ Base* indicate the mapping of the reads to the reference sequence that is encoded by the foreign-key relationship between attributes *RB_ ID* and *SB_ RB_ ID*. The lines between tables *Sample_ Base* and *Read* visualize the decomposition of the base sequence.

In the following, we explain in detail how we establish the foreign-key relationship between reference and sample bases. Moreover, we explain how we convert single CIGAR operations to encode the individual mapping of single sample bases.

Deriving foreign keys to encode base mappings. The challenge for deriving the correct foreign keys between table *Sample_ Base* and *Reference_ Base* is that the necessary data is distributed over several data files. Reference sequences (contigs) are stored in FASTA files and read mappings in SAM files. The mapping between them is expressed by the given start position of the read mapping and the reference sequence (contig) (cf. Section 3.1.1.2). To derive the mapping between every single base of a read and the respective base of a contig, we assume that we imported the respective contig before. During the import, we ensure that the single bases of a contig are imported in order of the given sequence. We assign an artificial primary key value (RB_ID) to each base that is incremented by 1. If table *Reference_ Base* is empty the first reference base gets the primary key 0. In the case that we already imported another contig, the primary key of the first base of the currently imported reference sequence is $\#tuples_{Reference\_Base} - 1$. Thus, we can ensure that the primary key values of the bases of the same reference sequence are continuous. In Figure 3.9, we show an example for the reference sequence of our running example. To import a read that is mapped to the contig, we have to determine the primary key id of the first base of the contig *chromosome1*. We can do this using following aggregation query:

```
SELECT min(RB_ID) AS MIN_C_ID FROM Reference_Base
JOIN CONTIG on RB_C_ID = C_ID WHERE C_NAME = 'chromosome1';
```

Then, we can use the minimum primary key value within the contig (MIN_C_ID), the start mapping position of a read (POS) and the offset of the specific base within a read (BASE_OFFSET(POS, CIGAR)) to compute the SB_RB_ID:

$$SB\_RB\_ID = MIN\_C\_ID + POS + BASE\_OFFSET(POS, CIGAR) - 1$$

Subtracting 1 is only necessary if the position information is one-based and not zero-based, i.e., the first base in a reference sequence has position 1 and not 0. The BASE_OFFSET depends on the position and the CIGAR operation applied to the base. Usually, the BASE_OFFSET is equal to the zero-based position of the base within a read. However, certain CIGAR operations modify the BASE_OFFSET, e.g., to encode the position of inserted bases correctly.

Deriving BASE_OFFSETs and representing CIGAR operations. In the following, we explain how we map the single CIGAR operations to our database schema and derive BASE_OFFSETs:

Match (M) A matching base is inserted into table *Sample_Base* with an *SB_Insert_-Offset* of 0. The quality value is stored as normal phred-scaled quality score and not as ASCII encoded quality score as specified by the SAM format (cf. Section 3.1.1.2). The necessary BASE_OFFSET to compute the SB_RB_ID is equal to the zero-based position of the base within the read minus the number of bases that were *inserted* before the matching base.

Deletion (D) A deleted base is handled like a matching base having the artificial base value *X* that is not part of the official alphabet encoding nucleotide bases [97].

Insertion (I) An inserted base is indicated by an *SB_Insert_Offset* greater 0. The BASE_OFFSET is the same as of the last matching or deleted base, which leads to the same *SB_RB_ID* value of the last matching or deleted base.

Hard clip (H) A hard clipped base was ignored during read mapping and removed from the read sequence. Thus, we have no bases to store in table *Sample_Base*. Since we translate CIGAR operations into database fields, we would loose the information of the hard clip. Thus, we store the number of hard clipped bases as read attribute. According to the SAM format specification [111], a hard clip can only be present as first and/or last operation. Thus, we require two attributes to encode hard clip information: one for a possible front and one for a possible rear hard clip, i.e, R_FRONT_HARD_CLIP_LENGTH and R_REAR_HARD_-CLIP_LENGTH respectively.

Soft clip (S) A soft clip is like a hard clip, except that the ignored bases and their quality scores are still part of the overall base sequence and and quality string. Since, we have no mapping information to store the single base within table *Sample_Base*, we also use read attributes to save this information. According to the SAM format specification, soft clips "may only have hard clips between them and the ends of the CIGAR string" [111]. Consequently, we need four string attributes: two indicating the clipped bases and their quality scores in front of a CIGAR string (R_FRONT_SOFT_CLIP_SEQ, R_FRONT_SOFT_CLIP_QUAL) and two attributes for clipped bases and their quality scores at the end of a CIGAR string (R_REAR_SOFT_CLIP_SEQ, R_REAR_SOFT_CLIP_QUAL).

The SAM format specification lists two further CIGAR operations: *N* and *P*. *N* indicates that a certain number of reference bases is simply skipped. Nevertheless, according to the manual [111], *N* is not defined for NGS read mapping. Thus, we do not define a rule for it. *P* can be used to specify the alignment between reads in case of a differing number of inserted bases at the same genome position. We can encode the correct ordering using additionally inserted a*N*y bases.

These rules only apply for mapped reads (R_FLAG & 4 <> 0). In case of unmapped reads, we can store the base sequence and the corresponding quality scores using the attributes R_FRONT_SOFT_CLIP_SEQ and R_FRONT_SOFT_CLIP_QUAL.

3.3.2 Relational SNV calling

The base-centric database schema resembles a star schema. Table *Sample_Base* is the fact table. All other tables are dimension tables. The base-centric database schema allows us to call SNVs purely relational with the help of genome-specific UDAs. In Listing 3.8, we show the SQL query to call SNVs. We join the fact table *Sample_Base* with all necessary dimension tables (lines 3 – 7). This allows for filtering specific data sets (lines 8 – 9). The storage of every single reference and sample base allows for precise position filtering without additional genome-specific functionality (line 10). Moreover, we can filter low quality sample bases (line 13). Additionally, we have to exclude inserted sample bases (line 12) as these are assigned to a reference base of interest, but only to encode their insertion position. They are not really mapped to a reference base and, thus, modify the analysis result if not filtered.

The join result is aggregated using the genome-specific UDF *CallBase* (line 2). The aggregation result is finally compared with the reference base and those rows are excluded that show no differing genotype or have a coverage below a given threshold (line 18). Note that we additionally return the contig name (C_NAME) (line 1) and, thus, sort by the contig name (line 19) to provide support for multi contig queries (line 9), which is important when querying complete genomes. This extended functionality can also be applied to the read-centric SNV calling approaches (cf. Listing 3.6 and Listing 3.7).

```
SELECT C_NAME, RB_POSITION, RB_BASE_VALUE, COUNT(SB_BASE_VALUE) AS
    coverage, CallBase(SB_BASE_VALUE, SB_BASE_QUALITY) genotype
FROM Read JOIN Sample_Genome ON R_SG_ID = SG_ID
    JOIN Contig ON R_CID = C_ID
    JOIN Reference_Genome ON C_RG_ID = RG_ID
    JOIN Reference_Base SB_RB_ID = RB_ID
    JOIN Sample_Base ON SB_R_ID = R_ID
WHERE SG_NAME = "human1" AND RG_NAME = "human"
    AND (C_NAME = "chromosome1" OR /* multiple contigs */)
    AND RB_POSITION BETWEEN 1,000 AND 2,000
    AND R_MAPQ > 29 AND R_FLAG & 4 <> 0
    AND SB_INSERT_OFFSET = 0
    AND SB_BASE_CALL_QUALITY > 30
    /* if the read is part of a paired-end read (1st bit set in FLAG)
       than both ends should be mapped (2nd bit is set) */
    AND (R_FLAG & 1 = 0 OR (R_FLAG & 1 <> 0 AND R_FLAG & 2 = 0))
GROUP BY C_NAME, RB_POSITION, RB_BASE_VALUE
HAVING RB_BASE_VALUE <> genotype AND coverage > 5
ORDER BY C_NAME, RB_POSITION;
```

Listing 3.8: Calling genotypes using the base-centric database schema.

3.3.3 Relationship to read-centric database approaches

Obviously, base-centric and read-centric database schema share the same data hierarchy. The base-centric schema adds one further level to the read-centric database schema making mapping information explicit. Thus, the base-centric database schema can be seen as a relational encoding of a *base pileup* allowing for convenient access to single bases and genome positions without the need of UDFs.

The base-centric schema can also be interpreted as view on read-centric data. To actually transform the data, a UDF would be required that converts data stored in the read-centric database schema into the base-centric layout similar to converting raw

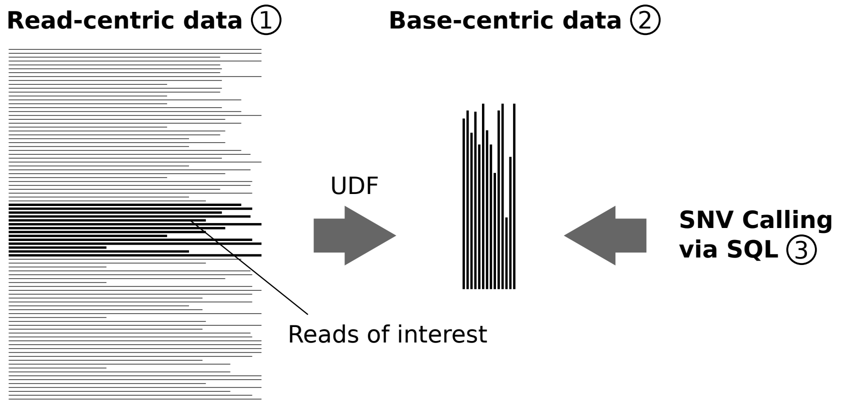


Figure 3.10: The base-centric database schema can be used as intermediate data representation ② to perform SNV calling ③ on a subset of read-centric data ①.

```
 1 SELECT R_ID,
 2     R_FRONT_SOFT_CLIP_SEQ +
 3         _SEQ(R_POS,SB_INSERT_OFFSET,SB_BASE_VALUE) +
 4         R_REAR_SOFT_CLIP_SEQ,
 5     R_FRONT_SOFT_CLIP_QUAL +
 6         _QUAL(R_POS,SB_INSERT_OFFSET,SB_BASE_VALUE,
 7                                 SB_BASE_QUALITY,SB_INSERT_OFFSET) +
 8         R_FRONT_SOFT_CLIP_QUAL,
 9     _CIGAR(R_POS,SB_INSERT_OFFSET,SB_BASE_VALUE,
10                                 R_... /* clipping fields */)
11 FROM Read JOIN Sample_Base ON R_ID = SB_READ_ID
12 WHERE SB_READ_ID IS IN (
13         SELECT DISTINCT R_ID
14         FROM Sample_Genome ON R_SG_ID = SG_ID
15             JOIN Contig ON R_CID = C_ID
16             JOIN Reference_Genome ON C_RG_ID = RG_ID
17             JOIN Reference_Base SB_RB_ID = RB_ID
18             JOIN Sample_Base ON SB_R_ID = R_ID
19         WHERE SG_NAME = "human1" AND RG_NAME = "human"
20             AND C_NAME = "chromosome1"
21             AND RB_POSITION BETWEEN 1,000 AND 2,000
22     )
23 GROUP BY R_ID;
```

Listing 3.9: Converting reads from the base-centric database schema into the SAM format.

input data from files (cf. Figure 3.9). Thus, we can use the base-centric database schema as intermediate data representation to implement a read-centric SNV calling approach. We depict the idea in Figure 3.10. We select the reads of interest ①, e.g., those reads that overlap a specific genome region, and convert them into a base-centric data layout ②. Then, we call SNVs on the base-centric data as described in Section 3.3.2 ③. This approach is highly related to the approaches by Cijvat et al. and Röhm and Blakeley that we conceptually extended to support SNV calling. Thus, we use it to evaluate a read-centric SNV calling approach within this thesis.

3.3.4 A word on SAM formatted data exports

Since the base-centric database schema stores mapping information explicitly, it requires additional effort to reconstruct SAM formatted reads. In Listing 3.9, we show a query to export reads of *human1* that cover the region from base 1000 to 2000 of *chromosome1* in the *human* reference genome. The three UDAs *_SEQ* (line 3), *_QUAL* (line 6) and *_CIGAR* (line 9) are responsible to reconstruct the respective SAM fields. Each UDA requires the actual position and insert offset of the base or quality score to determine the correct position in the resulting string. If we can guarantee that the order of tuples is the same as during the import, then we do not require the position information, because all information comes in the correct sequence. The inner query (line 13 – 21) returns a distinct set of *R_ID*s that fulfill the region criteria.

3.3.5 Qualitative assessment

In this section, we examine the benefits and challenges of our base-centric database schema in comparison to the read-centric database schema.

Clearly, the ability to perform genome-position related analyses such as SNV calling without using a conversion UDF is an advantage of our base-centric database schema, because it removes conversion overhead. Moreover, we can rely on relational database operators during processing and do not have to integrate genome-specific functionality into the processing stack, except UDAs for calling genotypes or to reconstruct SAM fields. This increases the applicability of our approach to different relational DBMSs and facilitates query processing using a relational database engine. To sum it up, we see the following two key benefits of storing read mapping information explicitly as in the base-centric database schema:

- The query processing relies completely on relational database operators. Thus, we can use existing database operators to perform SNV calling within a relational database system, which reduces implementation and maintenance effort and facilitates declarative analyses.
- We avoid conversion overhead during runtime and do not have to use non-relational conversion UDFs.

Nevertheless, we trade off these benefits for several challenges. A first drawback of storing mapping information explicitly is that it takes additional effort to export data in an implicit format such as the SAM format (cf. Section 3.3.4). A read-centric database schema has clear advantages here. However, it should be the goal to analyze the data completely inside the database, when it is stored there once. Thus, an export as SAM formatted output should be an exception.

A critical disadvantage of the explicit encoding of read mappings is that we always have to process the complete table *Sample_Base*. The size of table *Sample_Base* directly depends on the data set size. For example, the human reference genome comprises ca. 3.2 billion bases. Thus, an NGS data set with a coverage of four contains up to 12.8 billion mapped bases. Each of the reference and sample bases is represented by a row in table *Reference_Base* or *Sample_Base* respectively. Thus, in order to apply a genome region filter, we first filter the dimension table, table *Reference_Base*, and then join the result with the complete fact table, table *Sample_Base*, containing 12.8 billion mapped bases, even for small genome regions to analyze. In contrast, in a read-centric approach, we filter table *Read* and join it with table *Contig*. Following our example of 12.8 billion bases and assuming an average read length of 100 bases, we *only* have to filter and join 128 million reads, which is 100 times less the effort than using the base-centric approach. Then, we convert the selected sample and reference data using a genome-specific conversion UDF. The UDF can directly produce a ready-to-aggregate output, joining the single bases internally or we can use a base-centric-like processing approach (cf. Section 3.3.2 and Section 3.3.3) joining two smaller tables (depending on the filters on table *Read* and *Contig*). Thus, the processing runtime using a base-centric database schema appears to be not competitive on small genome regions of interest compared to a read-centric analysis approach or state-of-the-art analysis tools.

Another challenge of the base-centric database schema is the increased storage demand, which is a direct consequence of explicitly storing mapping information. For example, we store the mapping position for every mapped base as a foreign key. Considering our example from above, we have to store 12.8 billion foreign keys, one for each sample base. In contrast, the read-centric database schema encodes the concrete mapping position via the CIGAR string and the position of the base within the sequence string. Nevertheless, also a read-centric database approach faces the problem of the large storage consumption of the internal base pileup, but has the possibility to create base pileups on chunks of the mapped reads assuming the reads are sorted by mapping position [108].

In sum, we see the following two key challenges when storing read mapping information explicitly as in the base-centric database schema:

- The explicit encoding of read mappings introduces large overhead during query processing likely leading to non competitive runtime performance.
- Furthermore, the explicit encoding of read mappings increases the data volume drastically making it hard to compete with state-of-the-art storage approaches.

Conclusion. The base-centric database schema allows us to express SNV calling using relational database operators and additionally avoids conversion overhead during processing. Therefore, we require to store read mapping information explicitly. However, processing and storing explicit read mapping information leads to overheads that sacrifice the advantages. Consequently, to enable relational SNV calling using a base-centric database schema, we have to investigate efficient processing and storage techniques, otherwise the practical use of the base-centric database schema is questionable.

3.4 Related work

In this section, we review further related work to our work. First, we consider approaches that use DBMSs for analysis of mapped reads. Then, we discuss approaches that provide declarative access for NGS data sets.

Mapped read analysis using DBMSs

We already analyzed the approaches by Cijvat et al. and Röhm and Blakeley and explained the relationship to our approach. Both approaches store data in a read-centric database schema and need to make mapping information explicit during runtime to perform position-related analyses such as SNV calling. We can use our base-centric database schema to represent the result of the conversion and use our defined processing scheme to call SNVs. In addition, we can use our database schema standalone to encode read mapping information in a database-friendly way allowing for convenient access via SQL and processing using standard relational database operators.

Another approach that directly provides SNV calling functionality within a database systems is proposed by Fähnrich et al. [43]. This approach also works on read-centric data and uses a map-reduce processing framework to convert reads into base pileups. The work shows that the in-memory map-reduce approach scales well with the number of cores that modern computer systems provide. We aim to also benefit from in-memory processing, but want to use mostly relational database operators that are optimized and well integrated into the overall system.

Declarative data access for NGS data

In recent years, several approaches were proposed to facilitate genome analysis, especially NGS analysis, by using declarative query languages. Motivated by the idea of a layered genome analysis, Bafna et al. propose the Genome Query Language (GQL). GQL is designed to provide declarative access to the evidence layer, i.e., the raw data such as mapped reads, from the inference layer, e.g., a SNV caller consuming all mapped reads in specific genome region [6]. Considering the idea of a layered genome analysis, we integrate both layers in a relational database system. We can query the evidence data, i.e., mapped reads, in a specific genome region, and then infer a genotype to call SNVs.

GQL also supports interval processing, e.g., to search for regions that overlap a specific interval, intersect different intervals to find overlaps or merging intervals [67]. To this end, special operators are introduced such as the *mapjoin* operator that joins intervals by searching for intersections. Furthermore, GQL can be used to uncover structural genetic variation [67]. In contrast, our approach focuses on SNV calling in specific intervals.

The GenAp approach integrates the idea of interval joins into SparkSQL [68] to benefit from the high performance computing infrastructure. Thus, it should also be possible to integrate the interval processing into a relational DBMS. Another interesting approach that uses an SQL-like syntax, but in a non relational database system is GORpipe [49]. The purpose of GORpipe is to process position-specific data fast. Therefore, the authors define a data format and propose a query language that combines unix pipes with an SQL-like syntax to filter or join data. The approach allows for fast stream processing of position-specific data. The authors build GORpipe as replacement for their conventional relational DBMS that was not able to scale to increasing amounts of data. Nevertheless, this example shows the potential usefulness of SQL in the context of genome analysis.

Another declarative query language proposed for genome-related data is the Genometric Query Language (GMQL) [90]. It also facilitates region-specific analysis of downstream analysis results such as variant calls. Furthermore, it provides a flexible data format to encode a wide variety of region-specific data. Using the underlying data model, the position-specific data can be enriched with metadata [22]. The syntax is similar to SQL, but the underlying execution environment is build from scratch and not integrated into a relational DBMS.

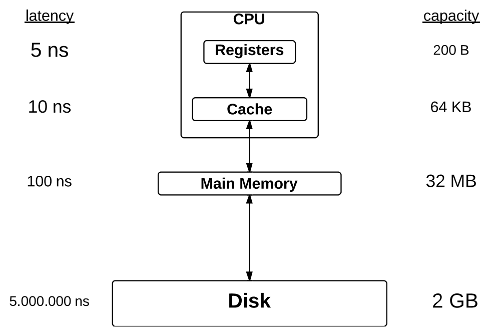
3.5 Wrap up

In this chapter, we answered our second research question: *How can we express variant detection using relational DBMS operators as a basis?* First, we provided conceptual extensions for existing approaches to analyze mapped reads allowing us to support SNV calling. However, these extended approaches rely partly on complex conversion UDFs. Since we aim to perform variant calling mainly using relational database operators to leverage the optimized processing engine of a DBMS, we derived the base-centric database schema. The base-centric database schema is a relational representation of a base pileup. We can use this schema either standalone to store mapped reads avoiding to make mapping information explicit during query processing using complex UDFs. Furthermore, we can use the base-centric database schema as intermediate representation when calling SNVs using the read-centric database schema. Nevertheless, our approach poses two challenges: First, the explicit encoding of mapping information leads to a large increase of storage size. Second, representing every single base in an NGS data set always requires to process the complete fact table independent of the queried genome range. Moreover, the size of the fact table depends on the stored data set size. Thus, it is not clear yet whether a base-centric approach can provide competitive runtime performance compared to state-of-the art analysis approaches. In the following chapters, we will investigate solutions to address these challenges. In Chapter 4, we examine efficient processing strategies for base-centric data. Then, we consider genome-specific compression schemes and processing of compressed genome data in Chapter 5.

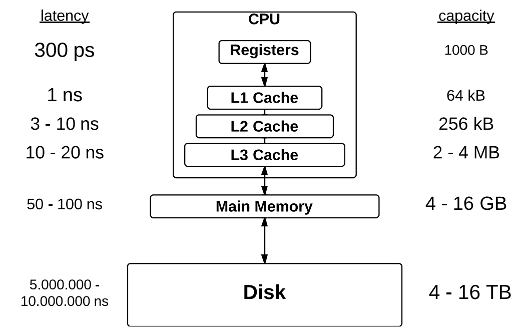
4. Efficient SNV detection using relational database operators

In this chapter, we examine strategies to process mapped read data efficiently using a relational DBMS. To this end, we examine the implementation space of our proposed relational SNV calling approach from Section 3.3.2. For our evaluation, we use a machine with enough main memory to keep intermediate results in main memory. Moreover, we use a main-memory DBMS that is designed to make benefit from increasing amounts of main memory and accompanying parallel computer architectures. With this setup, we do not have to rely on specialized main-memory saving UDFs that limit the benefits of DBMSs (cf. Section 3.2.3). In contrast, we use standard relational database operators to perform the analysis. Our investigation will lead us to an answer to our third research question: *How can we process genome data sets as efficient as specialized analysis tools using relational DBMSs?*

In Section 4.1, we give a brief overview on the design space of main-memory DBMSs and introduce our evaluation database system. Then, we perform an initial runtime and scalability analysis of the SNV detection query based on a hash join and sort-based aggregation implementation. We report and discuss the results in Section 4.2. Our analysis reveals that database approaches suffer from processing overhead due to joins, especially when we analyze large genome regions. In contrast, on small genome regions, we can achieve competitive runtime performance. To provide better scalability to larger genome regions, we discuss alternative join processing strategies in Section 4.3. The most promising approach is the *invisible join technique* [3]. Although restricted in its applicability, we show how we can apply it to our SNV calling use case. Using the invisible join technique, aggregation processing becomes the bottleneck in our processing pipeline. In Section 4.4, we investigate a technique called *array-based aggregation* to further speed up the SNV calling analysis.



(a) 1995 (Data taken from [54, Fig. 1.15])



(b) 2010 (Data taken from [55, Fig. 2.1])

Figure 4.1: Overview on latency and capacity of the single components of the memory hierarchy of a typical server computer from 1995 and 2010. Increasing capacities of main memory allow for using main memory as primary data storage. To unleash the full potential of a main memory storage system, optimizing for efficient memory access is vital to reduce the increased gap between access times of higher hierarchy elements.

4.1 A primer on main-memory DBMSs

We use a main-memory DBMS to evaluate our SNV calling concepts from Chapter 3, since main-memory DBMSs are designed to leverage modern computer architectures and promise significant speedups especially for analytical workloads [44, 60] compared to disk-based DBMSs. In this section, we briefly describe important concepts of main-memory DBMSs to give insights about the inner mechanics of our evaluation DBMS.

4.1.1 Disk-based vs. main-memory DBMSs

The CPU of a computer system can process data only if it is available in the registers of the CPU. In earlier computer systems, registers were small in size and data had to be fetched from the larger but slower main memory. To mitigate the lower access performance, a fast hardware cache was built in between to reduce the number of main memory accesses during data processing. Nevertheless, the main memory capacities of a computer system were also too small to keep all data required by a program such as a DBMS in main memory. Moreover, main memory is volatile. If the system is shutdown, the data is lost. Thus, data was stored primarily on disk and had to be loaded into main memory to process it. Within this memory hierarchy, the gap between access latencies of the different levels increases from top to bottom reaching its peak when accessing data on disk that was more than four orders of magnitude slower than to access data in main memory and up to six orders of magnitude slower than accessing data in the CPU registers. In Figure 4.1a, we show the relationship of latencies and capacities of the single elements of the memory hierarchy for a typical server computer from 1995.

Within the last two decades the gap between access latencies of higher hierarchy elements (registers vs. caches and main memory) increased by up to three orders of magnitude requiring the use of much more sophisticated hardware caches with several layers to fully utilize the potential power of a modern CPU (cf. Figure 4.1b). At the same time, the capacity of the lower hierarchy elements (disk and main memory vs. caches and registers) increased by two to three orders of magnitude allowing to keep the complete database in main memory, which avoids costly disk accesses.

The idea of main memory DBMSs is not an invention of the last decade. For example, Garcia-Molina and Salem already provide an overview on important concepts for main-memory DBMSs in 1992 [45]. However, only the growth of main memory capacities in recent years enabled the practical use of main-memory DBMSs. The obvious advantage of a main-memory DBMS is that it does not have to access the disk anymore to get the required data. Thus, there is no need for a sophisticated buffer manager. Studies have shown that the buffer manager accounts for up to 30% of the executed instructions in a typical database workload [52] indicating the potential performance gain by completely removing the buffer manager. Furthermore, other design decisions of traditional DBMSs should be carefully revised to increase the performance further [125]. For example, hardware prices dropped over the last 20 years allowing users to buy many servers instead of just one. Thus, a second server can be used as hot standby to reduce the risk of data loss due to main memory only storage in case one machines fails [64]. This will also reduce the need for costly backups of data on disks. Nevertheless, to fully exploit the potential of main memory storage in terms of improved runtime performance, the mechanisms and functioning of hardware caches have to be considered more carefully than they were in disk-based database systems [86] due to the focus on hiding disk latencies and avoiding disk accesses. Since cache efficiency is a primary design goal in main-memory DBMSs to provide peak performance compared to disk-based DBMSs, we briefly outline how hardware caches work.

How do hardware caches work? Hardware caches are used to close the large access gap between main memory and CPU (registers). Modern CPUs use several layers of hardware caches (cf. Figure 4.1b). An upper cache layer always contains a subset of the data of the lower cache layer or main memory. Higher cache levels provide faster access, but are much more expensive. Thus, they are kept small. If we access a data item in main memory, first, we check whether the data item already resides in one of the cache layers. In case we find the data item in the first level cache (L1 cache hit), we can directly access it without accessing lower levels leading to optimal access performance. In case of a cache miss, we check the lower cache levels. If the data item does not reside in one of the cache layers, we have to take the highest access penalties and retrieve the data item from main memory. To this end, the data item is loaded into all cache levels, possibly evicting already loaded data, since the cache is full. Finally, the CPU can access the data item. To increase the chance of cache hits, caches make use of the principles of spatial and temporal locality.

Leveraging spatial locality. To reduce the effort of loading data to caches, not only one data item is loaded into the cache, but several data items forming a data block of a specific size. To this end, the complete cache is divided into cache lines that are loaded and evicted as whole. A typical cache line size of a first level cache

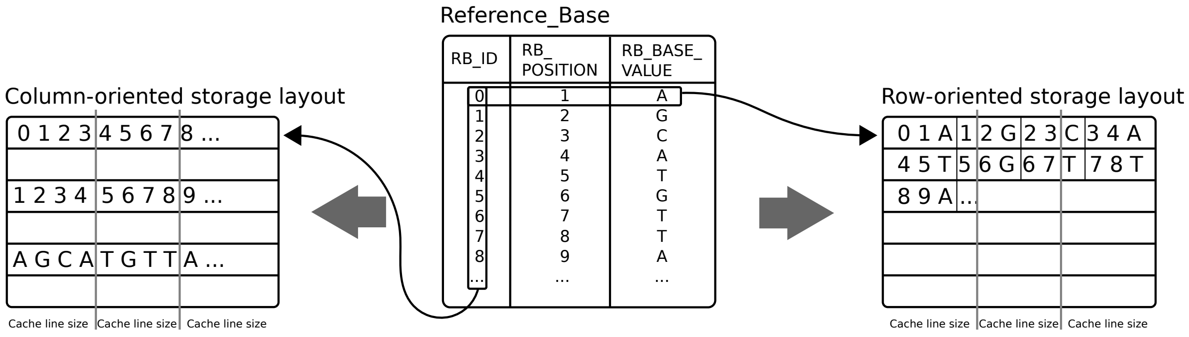


Figure 4.2: Column-oriented vs. row-oriented storage layout: Storing complete tuples consecutively provides highest cache efficiency when we often access all attributes of a tuple. If we only focus on single columns, a column-oriented storage layout allows for cache efficient data access rather than a row-oriented storage layout.

is 64 bytes. Thus, in case that we access data items that reside in consecutive main memory locations, i.e., are spatially close, the chance is high that we already cached the next data item as we have loaded a complete cache line.

Leveraging temporal locality. In case all cache lines are full and we need to load another non-cached data block, we have to decide which already loaded cache line has to be evicted. Typical strategies are *first in, first out* (FIFO) or *least recently used* (LRU). Especially, LRU allows to leverage temporal locality in the case that we access one data item several times. Thus, we can avoid to always evict and load the data item, i.e., the corresponding data block.

Having the general principles of hardware caches in mind, the algorithm definition as well as the choice of the data layout has great impact on whether we can leverage the caches to provide peak performance or not.

4.1.2 Column-oriented vs. row-oriented storage layout

A first decision, we have to make in a relational database systems is whether to use a column-oriented storage layout [26] or a row-oriented storage layout to represent a relational table in main memory. In a row-oriented storage layout, we store the single attribute values of a tuple consecutively in main memory. In contrast, a column-oriented storage layout keeps all attribute values of all tuples in a table (a column) consecutively in main memory. In Figure 4.2, we depict both storage layouts for the table *Reference_ Base* that we use to store reference sequences in the base-centric database schema (cf. Section 3.3.1).

Cache effectiveness. Considering cache effectiveness, it depends on the concrete use case whether a row store or a column store is more favorable. In case that we often access all attributes of a single tuple right after each other, we would prefer a row store, because when we load the first attribute to access it, it is very likely that we already loaded the following attributes. Such a use case is typical in a transaction processing system that manipulates or inserts complete records. In contrast, analytical queries often filter and aggregate single columns. Thus, a column store increases the cache efficiency, because we load multiple attribute values required for processing when we access a single attribute within the column (cf. gray lines in Figure 4.2). This difference in the application domain is not only restricted to main memory DBMSs, but also plays a vital role for disk-resident DBMSs to reduce the amount of data that has to be transferred from disk to main memory [3, 126].

Compression. Another advantage of column stores is the possibility to compress single columns using a compression scheme that fits best to the data. For example, a column that stores sorted numerical values is best compressed using run-length encoding that represents runs of the same value storing the value itself once and an additional run length [2]. Furthermore, using lightweight compression schemes makes it possible to process the data in its compressed form avoiding processing overhead [2]. For example, a scan on a run-length compressed column just has to compare the value of a run once and can induce the resulting tuple identifiers that match instead of scanning all values of all tuples. In contrast, using a row-oriented storage layout limits the applicability of compression schemes, since we store values of different columns (and possibly different value domains) consecutively in memory that are harder to compress, in particular using lightweight compression schemes.

Tuple reconstruction. In a row store, all attribute values of a tuple are stored consecutively in memory. Thus, reconstructing tuples is straightforward. In contrast, in a column stores all values of a tuple are spread over multiple columns located at different memory locations. Thus, it is likely that we cannot benefit from cache-efficient access. Such tuple reconstructions are necessary for nearly all queries, since we either process multiple columns of a table or have to return the query result consisting of multiple attributes per tuple. Thus, the question is when to materialize the tuples. In a column store it is usually a good choice to postpone the tuple reconstruction to the latest point, so called *late materialization* [2]. Thus, we can process multiple columns independently, which might avoid to materialize not needed tuples due to a filter reducing the tuple reconstruction overhead at all. Moreover, aggregation operations likely reduce the result set. Furthermore, we can operate as long as possible on compressed columns and benefit from cache-efficient processing during the complete query processing (assuming a column-oriented workload), since we do not reconstruct tuples in between.

4.1.3 Tuple-at-a-time vs. operator-at-a-time processing

Another decision that we have to make in a database system is whether to use an operator-at-a-processing [87] or a tuple-at-a-time processing model [47]. In Figure 4.3, we depict the general principle of a tuple-at-a-time processing engine. The single operators that make up a query such as a selection, join or aggregation operation are arranged in a sequence. Each subsequent operator can request the *next()* tuple from the previous operator. This processing model allows for pipeline processing, since multiple tuples

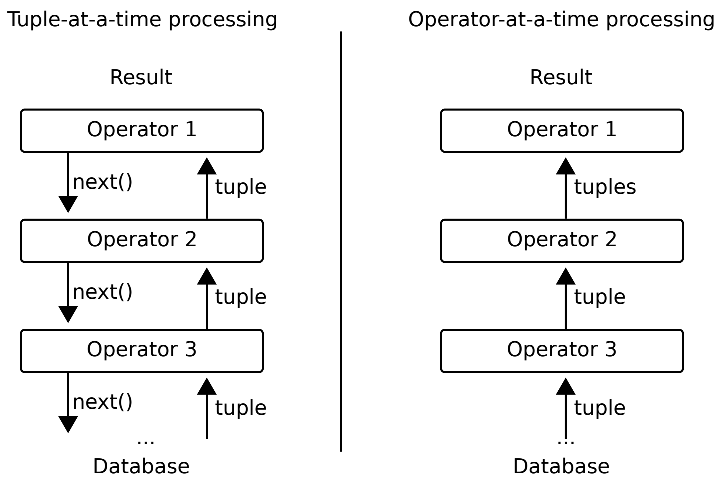


Figure 4.3: Tuple-at-a-time vs. operator-at-a-time processing: An operator-at-a-time processing engine executes each operator that process the complete input exactly once. In the tuple-at-a-time processing model, each operator is called per tuple introducing large function call overhead.

can be processed at the same time at different stages of the pipeline. Hence, it is well suited if data arrives with delay. Moreover, the size of intermediate results is small, because every operator produces a single tuple, except pipeline breaking operators such as sort or aggregation. Nevertheless, such operations are usually performed on small intermediate results at the end of the query plan.

A different approach is operator-at-a-time processing of complete tables or columns. In Figure 4.3, we depict the idea on the right side. Every operator consumes either all tuples of the input table or the complete output of the previous operator. To this end, every operator has to compute and materialize its complete result before the next operator can start. This processing model allows for intra-operator parallelism using loop unrolling or SIMD, since multiple tuples are processed within one operator.

Function call overhead. Furthermore, an operator-at-a-time processing engine requires only a few function calls to process the data. Function calls always introduce overhead due to stack manipulations, e.g., for copying function arguments on the stack and restoring registers when returning. Since a tuple-at-a-time processing engine usually processes only a single tuple within one function call, the function call overhead per processed tuple is large. In contrast, an operator of an operator-at-a-time processing engine is called once to process the complete data set [13].

Memory consumption. On the other hand, an operator-at-a-time processing engine can require large amounts of memory, because all intermediate results have to be materialized. Here, a tuple-at-a-time processing engine has advantages, since no results have to be materialized during processing. Moreover, the pipelining of tuples within a tuple-at-a-time processing engine allows to start processing data although not all data already resides in memory. This, is especially important in disk-based DBMSs that focus on hiding disk access latencies.

4.1.4 Evaluation system

In Chapter 3, we formulated the task of SNV calling as an aggregation task within a relational database, which requires efficient processing of large tables. Since we operate mostly on single columns, we choose a column store to perform our evaluations within this chapter. Furthermore, we decide for an operator-at-a-time processing engine to reduce the function call overhead and provide most efficient execution of our analytical workloads.

Since 2012, the database working group at the University of Magdeburg investigates how co-processors, in particular graphics processing units (GPUs), can be exploited to speedup database systems [15, 17, 18, 93]. These efforts resulted in the development of CoGaDB (the Column-Oriented GPU-Accelerated DBMS) [14] allowing for robust query processing in co-processor accelerated database systems [16]. CoGaDB provides state-of-the-art implementations of relational database operators for CPU and GPU. Since the integration of GPUs into DBMSs provides only significant speedups if data already resides in main memory [53], CoGaDB is a main-memory DBMS by design. In addition, CoGaDB provides extension interfaces for compression techniques. To conduct our research on efficient SNV calling using relational, main-memory DBMS, we choose CoGaDB as evaluation platform.

4.2 An initial runtime evaluation

In this section, we conduct an initial performance evaluation of database-driven SNV detection. Our goal is to investigate the runtime behavior of our proposed SNV calling query presented in Listing 3.8 using CoGaDB and to determine how an operator-at-at-time query execution engine competes with specialized analysis tools. First, we explain the logical query plan behind the SQL query that we use for all our experiments. Then, we perform an initial runtime performance analysis using a hash join and sort-based aggregation strategy. Furthermore, we investigate the impact of the conceptual differences when storing mapped read data using either a read-centric or base-centric database schema (cf. Section 3.3.5). We implement and evaluate both approaches. We call the different storage setups DBS_{seq} and DBS_{base}. DBS_{seq} stores DNA *seq*uences such as reads and contigs in a read-centric database schema (cf. Section 3.2.1) and converts data on-the-fly into a base-centric representation to finally execute the SNV calling query. DBS_{base} stores the data directly in a base-centric database schema (cf. Section 3.3.1).

4.2.1 Logical query plan

In order to perform our runtime performance analysis, we fixed the logical query plan to avoid side effects in our evaluation due to optimizer decisions as these might have impact on the runtime results. Furthermore, automatic query optimization is not in the focus of our work. We are interested in how a relational operator-at-a-time query engine and their operators behave assuming the following query workload.

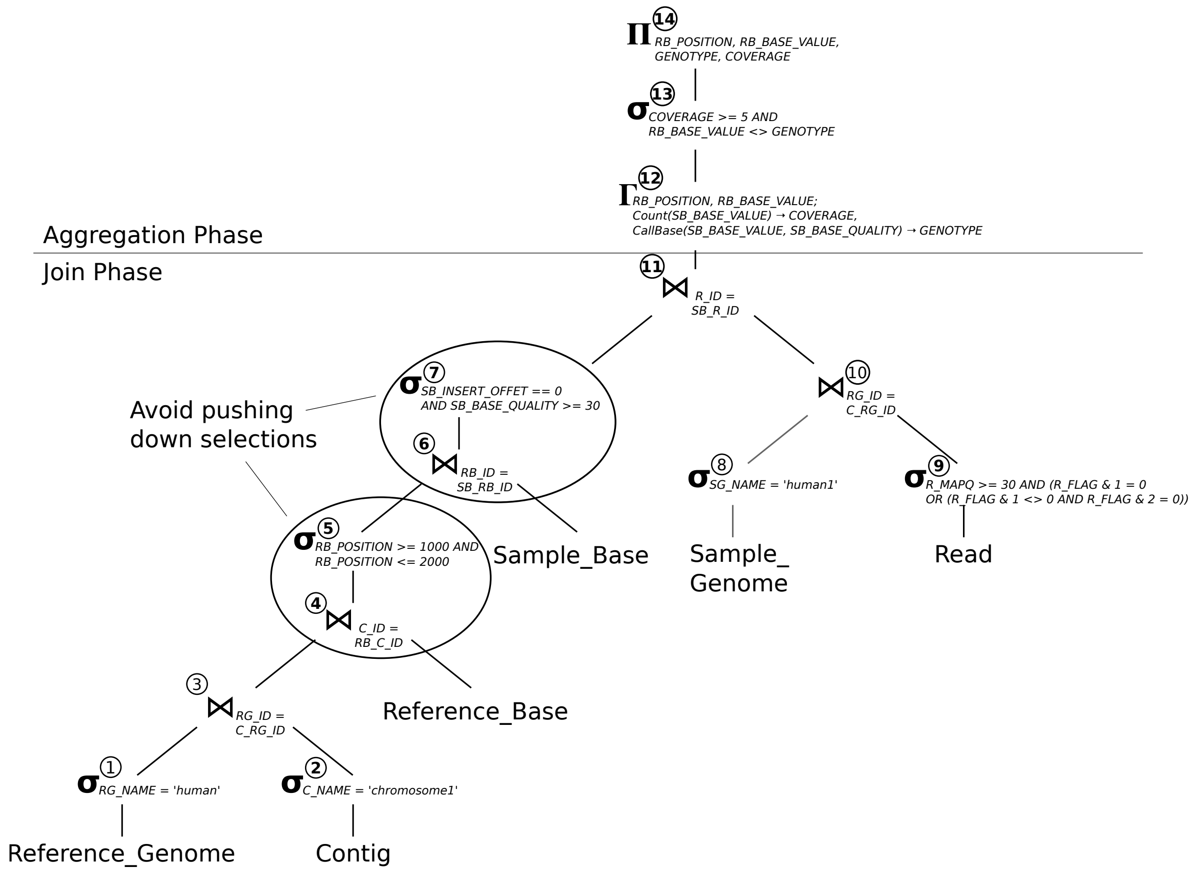

Π 14 RB_POSITION, RB_BASE_VALUE, GENOTYPE, COVERAGE
σ 13 COVERAGE >= 5 AND RB_BASE_VALUE <> GENOTYPE
Γ 12 RB_POSITION, RB_BASE_VALUE; Count(SB_BASE_VALUE) → COVERAGE, CallBase(SB_BASE_VALUE, SB_BASE_QUALITY) → GENOTYPE
Aggregation Phase
Join Phase
⋈ 11 R_ID = SB_R_ID
σ 7 SB_INSERT_OFFET == 0 AND SB_BASE_QUALITY >= 30
Avoid pushing down selections
⋈ 10 RG_ID = C_RG_ID
⋈ 6 RB_ID = SB_RB_ID
σ 8 SG_NAME = 'human1'
σ 9 R_MAPQ >= 30 AND (R_FLAG & 1 = 0 OR (R_FLAG & 1 <> 0 AND R_FLAG & 2 = 0))
σ 5 RB_POSITION >= 1000 AND RB_POSITION <= 2000
Sample_Base
Sample_ Genome
Read
⋈ 4 C_ID = RB_C_ID
⋈ 3 RG_ID = C_RG_ID
Reference_Base
σ 1 RG_NAME = 'human'
σ 2 C_NAME = 'chromosome1'
Reference_Genome
Contig

Figure 4.4: The optimized logical query plan for the SNV calling query from Listing 3.8. We avoid to push down all selections, because filters on dimension tables (Reference_Base) appear to be more selective improving runtime for querying small genome regions. We omit the steps with non-bold numbers during query execution in our evaluation, since we do not consider the respective filters in our query scenario.

Assumptions. For all our experiments, we assume that only one reference genome is loaded into the database and we are interested in querying the complete available reference genome or only parts of it. Moreover, we assume that all available reads in the database have to be processed. Thus, we do not filter reads according to their original sample genome. This behavior resembles the procedure of specialized flat file tools that analyze all data present in the input files. Our proposed query plan is optimized for such scenarios. Moreover, we use an operator-at-a-time processing engine that has highly parallelized operators. Thus, we execute all operators in sequence to not restrict the available processing resources per operator. Furthermore, we assume a single user scenario.

A logical query plan for SNV calling. Currently, CoGaDB applies only simple optimization rules during the evaluation of query plans such as *pushing down selections* and *always hash the smaller input table* during hash join processing. The former rule requires the DBMS to apply filters to single tables before joining the tables, since a filter is usually more selective and less computationally expensive than a join reducing the effort during join computation. The latter rule ensures that the hash table is as small as possible to speedup the probing phase. Nevertheless, in our final logical query plan that we used for our evaluations, we make two exceptions from the rule *pushing down selections*. In Figure 4.4, we show the logical query plan behind our SNV calling query from Listing 3.8. In the following, we explain it in detail and reason our decisions.

In order to perform SNV calling on a specific genome region, first, we filter (σ) the tables *Reference_Genome* ① and *Contig* ② and join ($\bowtie$) the results ③. Note that the filter on table *Reference_Genome* is not considered in our evaluation, since we assume to store only one reference genome in the database. For that reason, we omit step ① and ③ during the execution of the query in our evaluation. In the next step, we join the result with table *Reference_Base* ④ and filter those reference bases that are in the genome region of interest afterwards ⑤. At this point, we make a first exception from the rule *pushing down selections*. This decision reflects the worst case, in which we always query complete *contig*uous sequences, e.g., a chromosome. The overhead is constant, since the reference genome has always the same size. Pushing down the selection will likely speed up the processing performance.

After we determined the required reference bases involved in our genome region of interest (① - ⑤), we join them with table *Sample_Base* ⑥ and apply filters for non-inserted and high-quality sample bases afterwards ⑦. At this point, we make the second exception from the rule *pushing down selections*. This time we expect that the filtering for a specific region on table *Reference_Base* and, thus, the join between table *Reference_Base* and *Sample_Base* is usually more selective than the filter regarding non-inserted (*SB_INSERT_OFFSET* == 0), high quality sample bases (*SB_BASE_QUALITY* >= *threshold*) leading to improved runtime, in particular on small genome regions. For example, using a standard quality threshold of *13* as done by SAMTOOLS for the data set that we use in our evaluations within this chapter (cf. Section 4.2.2.1), the combined selectivity factor of both predicates is roughly 99%. In contrast, filtering for a specific genome region will likely be more selective.

Then, we join ⑪ the result so far with the join result ⑩ between the filtered tables *Sample_Genome* ⑧ and *Read* ⑨. Note that the filter on table *Sample_Genome* is not considered in our evaluation, since we assume to process all available reads in the database. For that reason, we omit step ⑧ and ⑩ during the execution of the query in our evaluation. In the next step, we perform the aggregation (Γ) ⑫ and filter the aggregated results for SNVs ⑬. Finally, we output (Π) the found SNVs including their position and respective reference base ⑭.

Possible optimizations. In the previous description, we already mentioned that depending on data and query characteristics, the logical query plan might change. Further optimizations are possible that we shortly discuss in the following. For example, if we increase the quality threshold, it could be better to push down the respective selection ⑦, since the selectivity increases. Moreover, if our assumption that we always analyze all available reads in the database does not hold, it could be better to first join tables *Sample_Base* and *Read* ⑪ before joining table *Sample_Base* and table *Reference_Base* ⑥. This decision is advantageous if we apply a filter to table *Read*, i.e., we analyze only reads of a specific sample genome ⑧, which can be more selective than the genome region filter ⑤. Overall, the query plan that we consider in our evaluation reflects a worst case scenario and possible optimizer decisions will improve the runtime in corner cases or under different assumptions.

4.2.2 Runtime evaluation

We can roughly divide the logical query plan presented in Figure 4.4 into two phases: a join and an aggregation phase. DBS_{seq} has an additional conversion phase. In the following, we investigate the runtime performance of the overall query and the single phases using our evaluation system CoGaDB. We evaluate the DBS_{seq} and DBS_{base} approach using a straightforward query execution engine and compare their runtime with the runtime of SAMTOOLS[1].

4.2.2.1 Experimental setup

Our evaluation platform is a machine with two Intel Xeon E5-2609 v2 with four cores @2.5 GHz and 256 GB main memory. We run CoGaDB on Ubuntu 14.04.3 (64 Bit) and compile it using gcc 4.8.4 with optimization level *-O3*. To execute the single phases, we use a hash join and a sort-based aggregation strategy. A hash join is a common join processing technique in many relational DBMS. We decide for a sort-based aggregation, because the final result has to be sorted by an attribute combination that is a subset of the grouping attributes for the aggregation. Our sort-based aggregation implementation guarantees the final global sorting. All processing steps within CoGaDB are parallelized to provide peak performance and utilize the available multi-core system.

Data set. To not interfere with main memory constraints in these experiments, we use the read mapping data of chromosome 1 of the human genome HG00096[2] provided by the 1000 Genomes Project [1]. The reference sequence of chromosome 1 comprises 249,250,621 bases. The mapping data set comprises 11,247,898 mapped reads consisting of 1,079,623,529 bases together. Thus, we have an average coverage of 4. DBS_{seq} requires roughly 3.2 GB of main memory for storing the uncompressed raw data. DBS_{base} blows up the storage size to 30 GB. The original compressed flat files require 1.5 GB. Before starting the experiments, we load the database into main memory.

Baseline. We use the SNV calling runtime of SAMTOOLS 1.3 as baseline for our evaluation. Although specialized analysis tools such as SAMTOOLS operate on disk-resident data, we argue that a comparison with our main-memory DBMS approaches leads to meaningful results. The primary goal of our work is not to show how to

[1]SNV calling using SAMTOOLS involves a second tool called BCFTOOLS. For simplicity, we just use SAMTOOLS to refer to both tools.

[2]data is available at ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00096/

outperform specialized analysis tools, but to investigate strategies allowing DBMSs to be competitive in processing mapped read data. This also includes the choice of data layout and storage medium. To ensure a balanced comparison, we assume that mapped reads are presorted by position and do not take sorting time into account for the runtime assessment. The presorting allows specialized analysis tools to process data in streams hiding the disk latency and works well with compressed data. Our approach does not depend on sorted read data, but we assume that data is already available in main-memory.

We use the same probabilistic error model function within our database approaches as SAMTOOLS. In Section 3.1.3, we explained that SAMTOOLS applies further statistical methods to improve the quality of analysis results. Since we are interested in the basic processing capabilities, we deactivate this functionality within SAMTOOLS. Moreover, we manually parallelized SAMTOOLS, because it does not provide such an option. To this end, we execute as many instances of SAMTOOLS as CPU cores are available on distinct, contiguous and equally sized chunks of the reads. Due to the sorting of reads by mapping position, each SAMTOOLS instance can determine which reads it has to process efficiently. Furthermore, each SAMTOOLS instance accesses compressed read mapping data stored on a ramdisk.

Query set. We perform the SNV detection using varying selectivity with respect to the size of the queried genome range. We report the runtime of chosen selectivity factors ranging from 0.001% (ca. 2500 reference bases) to 100% (complete chromosome 1). Processing the data set in a bulk (selectivity factor 100%) is a common request to get an overview on all genetic variations present. In contrast, lower selectivity factors correspond to the use case to investigate single genes. The average gene size is about 27,000 bases [131]. We report the average runtime of 30 queries.

The actual number of processed sample bases depends on the coverage of the data set. The average coverage of our chosen data set is four. Nevertheless, certain regions of the genome will have a higher actual coverage than others. While this circumstance does not have much impact on the runtime considering selectivity factors of 10% (processing 100 million sample bases) and above, we have to take care of it for lower selectivity factors. For that reason, considering selectivity factors below 10%, we perform the SNV detection on 30 randomly chosen genome regions. For all other selectivity factors, we query the same genome region starting at the first position of the reference sequence.

4.2.2.2 Results

We show the results in Figure 4.5. Since the runtimes between a query with highest selectivity (0.001%) and a query with lowest selectivity (100%) differ more than a magnitude, we report an overview on all runtime results in Figure 4.5a. To provide insights on queries having high selectivity, we zoom in the results in Figure 4.5b.

In both figures, we show for every selectivity factor the runtime results in following order: DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS. For DBS$_{base}$, we breakdown the overall runtime in the two phases for joining and aggregating the data. For DBS_{seq}, we additionally report the time to convert the data.

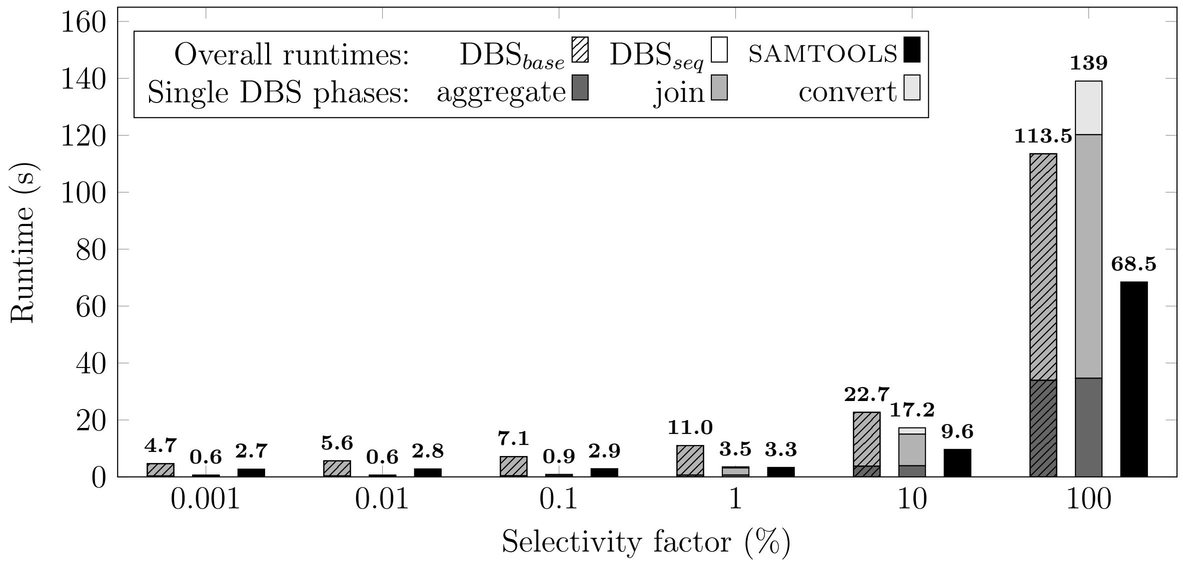
Overall runtime. We observe that SAMTOOLS is up to twice as fast as any of our database approaches on larger genome regions (selectivity factor $>=$ 10%). On smaller genome regions DBS_{seq} outperforms DBS_{base} and SAMTOOLS. Considering the database approaches only, we find that DBS_{base} is faster than DBS_{seq} exactly saving the conversion runtime of DBS_{seq} when querying the complete data set. Considering a selectivity factor of 10% or less, DBS_{base} is slower than DBS_{seq}, although DBS_{seq} converts data during the query processing.

Runtime of single phases. We see that the conversion time in DBS_{seq} is negligible considering the overall runtime of all queries. The runtime for the aggregation phase within DBS_{base} and DBS_{seq} is similar independent of the selectivity factor. Considering the join phases, we can observe this similarity only when we process the complete data set (selectivity factor 100%). In all other cases, we see that the join phase using DBS_{seq} is significantly shorter than the join phase within DBS_{base}. Thus, although DBS_{seq} converts data at runtime, DBS_{seq} is significantly faster than DBS_{base} on smaller genome regions.

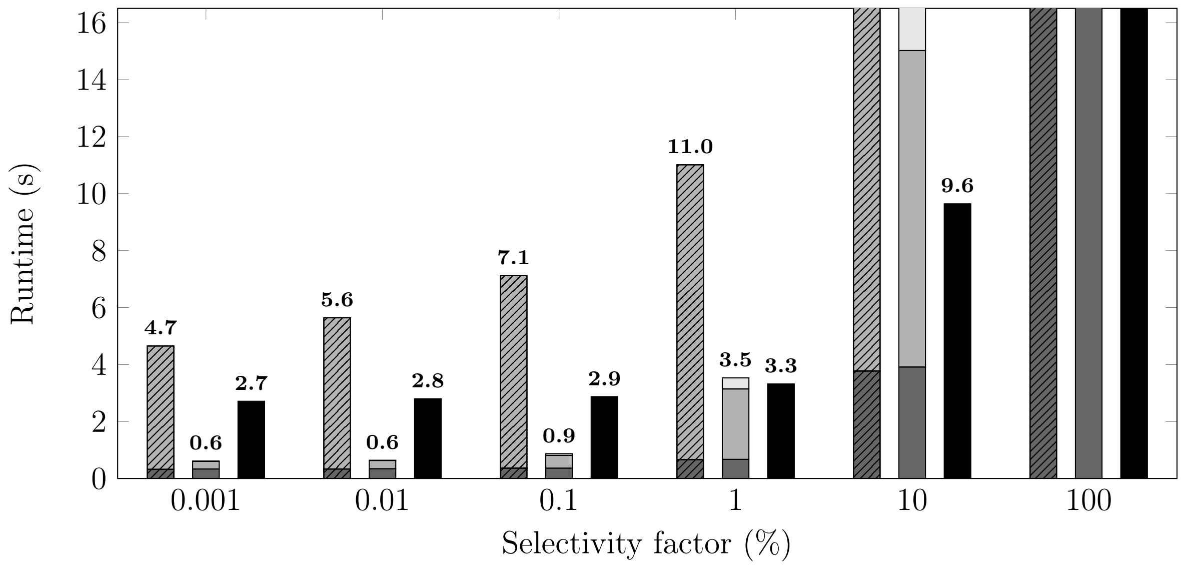
4.2.2.3 Discussion

The results reveal that our chosen relational processing strategy based on a hash join and sort-based aggregation strategy introduces overhead compared to the specialized analysis tool SAMTOOLS when we process the complete data set (selectivity factor 100%). Considering the execution times of the single phases when we process the complete data set, the join processing phase of DBS_{base} and DBS_{seq} dominates the overall execution time. Even DBS_{base} is nearly two times slower than SAMTOOLS, although we avoid the data conversion. Specialized analysis tools rely on reads that are sorted by mapping position. This circumstance reduces the effort to find all sample bases that are mapped to the same genome position required for aggregation (cf. Section 3.1.2). Using our database processing strategies, we have to process all reads and their sample bases to find those sample bases that are mapped to the same genome position. This is a notable disadvantage explaining the runtime differences.

Nevertheless, on smaller genome regions (selectivity factor $<=$ 10%), the overall runtime of DBS_{seq} is competitive to SAMTOOLS and DBS_{seq} even outperforms SAMTOOLS. Based on this observation, we would expect that DBS_{base} also outperforms SAMTOOLS on small genome ranges, because it avoids data conversion at all. Unfortunately, the runtime of the join processing phase of DBS_{base} is much higher than the combined runtime of join processing and data conversion of DBS_{seq} together (selectivity factors $<=$ 10%). The reason for this difference in the runtime of the join processing phase is that within DBS_{base}, we always have to probe all sample bases of the data set during the hash join, which are ca. one billion tuples. Only the hash table size gets smaller if we select a smaller genome region for processing. We show the detailed number of bases that have to be processed for each approach in Table 4.1. Using DBS_{seq}, we only have to



(a) Initial SNV calling runtimes on chosen selectivity factors.



(b) Initial SNV calling runtimes zoomed in.

Figure 4.5: Calling SNVs on chromosome 1 of a low coverage human genome using DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS. The conversion phase of DBS$_{seq}$ is negligible considering the overall runtime. In particular on small genome regions (selectivity factor $<$ 1%), DBS$_{seq}$ offers superior runtime performance. DBS$_{base}$ cannot meet the expectation to be as fast as DBS$_{seq}$ without data conversion for small genome regions (selectivity factor $<=$ 10%), because additional join effort sacrifices the benefit of avoiding the data conversion.

Selectivity factor	# of bases to join		
in %	reference	sample (DBS$_{seq}$)	sample (DBS$_{base}$)
0.001	2,493	13,317*	1,079,623,529
0.01	24,925	110,091*	1,079,623,529
0.1	249,251	1,078,055*	1,079,623,529
1	2,492,506	10,150,788*	1,079,623,529
10	24,925,062	110,454,734	1,079,623,529
100	249,250,621	1,079,623,529	1,079,623,529

Table 4.1: DBS$_{seq}$ and DBS$_{base}$ have to join a different number of sample and reference bases depending on the chosen selectivity factor. Since DBS$_{seq}$ converts read data on the fly, the number of sample and reference bases to join increases with increasing selectivity factor. DBS$_{base}$ also processes less reference bases if the selectivity factor is low, but always has to process all sample bases independent of the selectivity factor.

*Average of 30 different, randomly selected genome regions.

probe those sample bases that belong to the preselected reads saving much effort during join processing (cf. Figure 3.10). This is in accordance to the runtime behavior when processing the complete data set (selectivity factor 100%, cf. Figure 4.5a). Here, the combined aggregation and join phase of DBS$_{base}$ and DBS$_{seq}$ have a similar runtime, because using either DBS$_{base}$ or DBS$_{seq}$, we have to probe all sample bases in the data set. Thus, DBS$_{base}$ seems to be only beneficial for bulk processing complete data sets.

Overall, it is evident that DBS$_{seq}$ provides best runtime performance on small genome regions, i.e., genes, allowing for fast, on demand analysis. In contrast, DBS$_{base}$ is the fastest choice for the use case of processing complete data sets within a database, since it avoids data conversion at runtime. Nevertheless, considering larger genome regions, SAMTOOLS outperforms the database approaches, because the database approaches suffer from large join processing overhead. In the following section, we discuss strategies to speed up the join phase within the database approaches. We expect an overall runtime performance improvement for DBS$_{base}$ over all queried genome ranges. Since DBS$_{seq}$ already provides superior performance on small genome regions, a faster join processing mainly aims at speeding up the analysis of large genome regions.

4.3 Accelerating the join phase

In this section, we discuss strategies to speed up the join processing phase within our database approaches. Note that our proposed optimizations do not change the query (cf. Listing 3.8) or its output. We focus our efforts on the implementation of the join operators, especially on the join between the largest tables within the database, i.e., tables *Reference_Base* and *Sample_Base* (cf. ⑥ in Figure 4.4). While we can reduce the join effort in DBS$_{seq}$ on small genome regions, DBS$_{base}$ always has to process the complete table *Sample_Base*, even on small genome regions. Thus, the challenge is to reduce the effort for joining such large tables. In the following, we discuss optimization options and evaluate the most promising one.

4.3.1 Optimization options

We see two possible optimization options to speed up the join processing between tables *Reference_Base* and *Sample_Base*. First, we can apply different join processing strategies. Second, we can use a tailor-made database schema that avoids the join processing at all. In the following, we discuss advantages and disadvantages of these approaches.

Optimized join processing

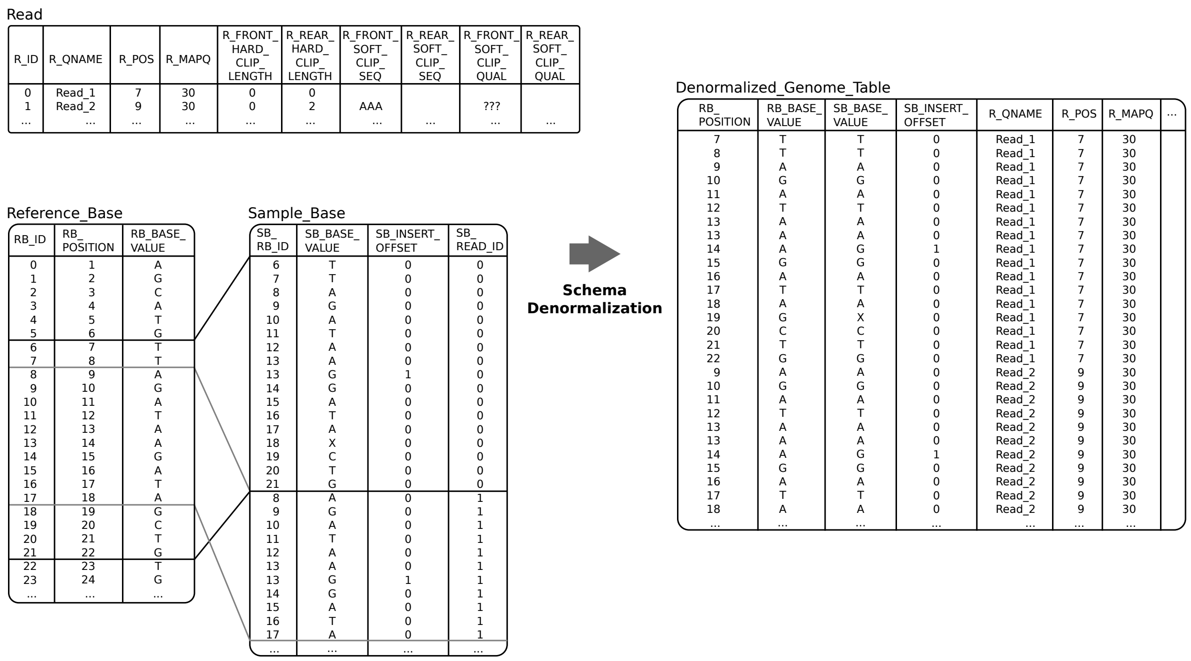
The hash join implementation that we used in the previous experiment probes the larger table, i.e., table *Sample_Base*, in parallel. The hash table itself is build serially avoiding costly synchronization, in particular when the input table, i.e., table *Reference_Base*, has a small size.

Nevertheless, there is some room for optimizations. First, we could build the hash table in parallel using a *shared* hash table that requires synchronization during the build phase [11]. Another idea is to use a partitioning hash join that divides both input tables into partitions of tuples that can be joined independently using hashing [117] or radix partitioning [84]. These approaches can be tuned to improve memory access by reducing the partition size, which allows us to keep the required hash table completely in the caches reducing the number of cache misses during the probe phase. However, our experiments with the radix-join implementation provided by Teubner et al. [127] only led to a small runtime improvement, but revealed another challenge of partitioning hash joins: cache-inefficient subsequent result lookups. The problem is that the resulting tuple identifiers computed during the hash join are not in order leading to inefficient memory-access patterns. Manegold et al. proposed a strategy to improve such subsequent result lookups [85], but obviously not without additional effort.

Another strategy could be sort-merge join that sorts (partitions of) the input tables and joins them during the merge phase. Hardware-conscious approaches are available [4, 65] and there is extensive research whether hash or sort-merge joins are superior regarding runtime [7, 65]. However, we do not reduce the inherent effort of sorting or probing (nearly) all tuples of table *Sample_Base* using any of these techniques, especially for our DBS_{base} approach. With increasing dataset sizes leading to table sizes with several billion rows, it is foreseeable that the join processing within DBS_{base} will not lead to competitive runtime results even if we apply advanced join implementations, because specialized analysis tools benefit from presorted data during processing. Moreover, DBS_{seq} will also not scale to larger genome regions.

Alternative database schema

Another strategy to speed up the join phase is to avoid the join at all. The idea is known as denormalization and the usefulness for analytical workloads despite data redundancies was shown recently [80]. In Figure 4.6, we depict the idea in our context. Instead of using a foreign-key relationship to associate a sample base to its reference base, we



Read

R_ID	R_QNAME	R_POS	R_MAPQ	R_FRONT_ HARD_ CLIP_ LENGTH	R_REAR_ HARD_ CLIP_ LENGTH	R_FRONT_ SOFT_ CLIP_ SEQ	R_REAR_ SOFT_ CLIP_ SEQ	R_FRONT_ SOFT_ CLIP_ QUAL	R_REAR_ SOFT_ CLIP_ QUAL
0	Read_1	7	30	0	0				
1	Read_2	9	30	0	2	AAA		???	
...	...	...	...	...	...	...	...	...	...

Reference_Base

RB_ID	RB_ POSITION	RB_BASE_ VALUE
0	1	A
1	2	G
2	3	C
3	4	A
4	5	T
5	6	G
6	7	T
7	8	T
8	9	A
9	10	G
10	11	A
11	12	T
12	13	A
13	14	A
14	15	G
15	16	A
16	17	T
17	18	A
18	19	G
19	20	C
20	21	T
21	22	G
22	23	T
23	24	G
...	...	...

Sample_Base

SB_ RB_ID	SB_BASE_ VALUE	SB_INSERT_ OFFSET	SB_ READ_ID
6	T	0	0
7	T	0	0
8	A	0	0
9	G	0	0
10	A	0	0
11	T	0	0
12	A	0	0
13	A	0	0
13	G	1	0
14	G	0	0
15	A	0	0
16	T	0	0
17	A	0	0
18	X	0	0
19	C	0	0
20	T	0	0
21	G	0	0
8	A	0	1
9	G	0	1
10	A	0	1
11	T	0	1
12	A	0	1
13	A	0	1
13	G	1	1
14	G	0	1
15	A	0	1
16	T	0	1
17	A	0	1
...	...	...	...

Denormalized_Genome_Table

RB_ POSITION	RB_BASE_ VALUE	SB_BASE_ VALUE	SB_INSERT_ OFFSET	R_QNAME	R_POS	R_MAPQ	...
7	T	T	0	Read_1	7	30	
8	T	T	0	Read_1	7	30	
9	A	A	0	Read_1	7	30	
10	G	G	0	Read_1	7	30	
11	A	A	0	Read_1	7	30	
12	T	T	0	Read_1	7	30	
13	A	A	0	Read_1	7	30	
13	A	A	0	Read_1	7	30	
14	A	G	1	Read_1	7	30	
15	G	G	0	Read_1	7	30	
16	A	A	0	Read_1	7	30	
17	T	T	0	Read_1	7	30	
18	A	A	0	Read_1	7	30	
19	G	X	0	Read_1	7	30	
20	C	C	0	Read_1	7	30	
21	T	T	0	Read_1	7	30	
22	G	G	0	Read_1	7	30	
9	A	A	0	Read_2	9	30	
10	G	G	0	Read_2	9	30	
11	A	A	0	Read_2	9	30	
12	T	T	0	Read_2	9	30	
13	A	A	0	Read_2	9	30	
13	A	A	0	Read_2	9	30	
14	A	G	1	Read_2	9	30	
15	G	G	0	Read_2	9	30	
16	A	A	0	Read_2	9	30	
17	T	T	0	Read_2	9	30	
18	A	A	0	Read_2	9	30	
...	...	...	...	...	...	...	

Figure 4.6: Instead of resolving associated information referenced via foreign keys using a join, we simply store the join result. On the one hand, we gain a performance increase, since we avoid to join the data at all. On the other hand, we introduce data redundancy leading to increased compression and maintenance effort.

just store the reference base (RB_BASE_VALUE), its position (RB_POSITION), the corresponding contig and reference genome (not shown). We can apply the same idea to the read and sample genome tables. Thus, in order to call SNVs using SQL, we simply have to scan the respective columns to filter data of interest. In Listing 4.1, we show the resulting SQL query derived from the query in Listing 3.8. All we have to change is to replace the five join clauses in the FROM clause with the new denormalized table. Obviously, scanning multiple columns, especially in a column store, is less expensive than joining the data.

Although this approach might be appealing from a performance perspective, the introduced redundancy (cf. columns R_QNAME and R_POS) leads to large storage overhead. Lightweight compression schemes such as dictionary compression (cf. columns R_QNAME) and run-length encoding (cf. columns R_MAPQ) can be used to reduce the overhead, especially in column-oriented database systems [2]. Nevertheless, we do not expect that such a database schema is a practical solution for permanent storage or representation of mapped read data due to the large redundancy of dimension information such as read data. For example, if we want to extract data for a specific read, we find the information at various places making it a non straightforward undertaking to query the data. Moreover, it is not straightforward to store (parts of) reference sequences that have no sample bases mapped to it. Consequently, we would apply such

```
SELECT C_NAME, RB_POSITION, RB_BASE_VALUE, COUNT(SB_BASE_VALUE) coverage,
    CallBase(SB_BASE_VALUE, SB_BASE_QUALITY) genotype
FROM Denormalized_Genome_Table
    /* instead of the 5 join clauses of Listing 3.8 */
WHERE SG_NAME = "human1" AND RG_NAME = "human"
    AND (C_NAME = "chromosome1" OR /* multiple contigs */)
    AND RB_POSITION BETWEEN 1,000 AND 2,000
    AND R_MAPQ > 29 AND R_FLAG & 4 <> 0
    /* if the read is part of a paired-end read (1st bit set in FLAG)
       than both ends should be mapped (2nd bit is set) */
    AND (R_FLAG & 1 = 0 OR (R_FLAG & 1 <> 0 AND R_FLAG & 2 = 0))
GROUP BY C_NAME, RB_POSITION, RB_BASE_VALUE
HAVING RB_BASE_VALUE <> genotype
ORDER BY C_NAME, RB_POSITION;
```

Listing 4.1: Calling genotypes using a denormalized database schema.

a denormalized database schema as intermediate data representation for the output of the conversion of DBS$_{seq}$.

Denormalization without joining. If we use the denormalized database schema only as intermediate data representation, the critical question is *How can we compute the join result needed for denormalization without actually joining the data?* If we would simply join it using a hash or sort-merge join, we would not have gained much compared to the current approach using a dedicated join phase. The answer is that we can leverage the foreign key relationship to perform efficient positional lookups of data. Such lookups are cheap in a column store. In Section 3.3.1, we explained in detail how we construct the foreign key relationship from the existing read mapping data. The key to success is the use of artificial, continuous ids used as primary and foreign keys. Since we can guarantee that the primary keys always start at zero and have a continuous range with the maximum value of $\#tuples - 1$ of the respective table, we can use them as indexes for direct lookup in the columns of a table, e.g., to retrieve data such as RB_BASE_VALUE and RB_POSITION of table *Reference_Base*. We can apply this idea to all foreign-key relationships within the base-centric database schema allowing to denormalize the schema without the need of joining the data.

Virtual denormalization using invisible joins. At a first glance, we can apply the idea of denormalization only to DBS$_{seq}$ efficiently, because combining it with DBS$_{base}$ would require to transform the complete database, which would introduce even larger storage overhead. However, the idea of using foreign keys for a positional lookup in a column store was also leveraged by Abadi et al. for their join technique called *invisible join* [3]. Using an invisible join, we determine all rows of the fact table, i.e., table *Sample_Base*, and then use the foreign keys to lookup the related dimension data per row. This is similar to the approach that we described above to transform mapped read data into an intermediate denormalized database schema, but without the need for materializing the denormalization.

Additionally, we can extend the approach to apply the positional lookup only to those rows of the fact table that fulfill certain predicates reducing the invisible join effort further. The challenge is that predicates, e.g., a region filter such as `RB_POSITION BETWEEN 1,000 AND 2,000`, are often defined on dimension attributes and cannot be applied directly to the rows of the fact table, e.g., table *Sample_ Base*. To rewrite such predicates, we have to determine the primary keys of the dimension table, e.g., table *Reference_ Base*, that fulfill the predicate, store them in a hash table and iterate through all rows of the fact table to determine the selected rows for performing the positional lookup. Obviously, rewriting the predicate using a hash table leads to similar effort than using a hash join. Fortunately, our most selective dimension predicate, the genome region filter, is usually a *between predicate*. Abadi et al. found out that such *between predicates* can often be rewritten as *between predicates* on the foreign key column of the fact table [3]. In our use case, we are able to rewrite a predicate such as `RB_POSITION BETWEEN 1,000 AND 2,000` into `SB_RB_ID BETWEEN 999 AND 1,999` that can be directly applied to the fact table *Sample_ Base*. Thus, the task for probing every row of table *Sample_ Base* can be transformed into a *between predicate* filter which should lead to a significant reduction of runtime when joining large tables. This should be especially important for DBS$_{base}$ that has to join the complete table *Sample_ Base* even when we process small genome ranges.

Overall, the idea of using the invisible join technique in combination with *between predicate* rewriting to implement a virtual denormalization is very promising for our use case. In the following section, we evaluate the impact of this technique on our database approaches.

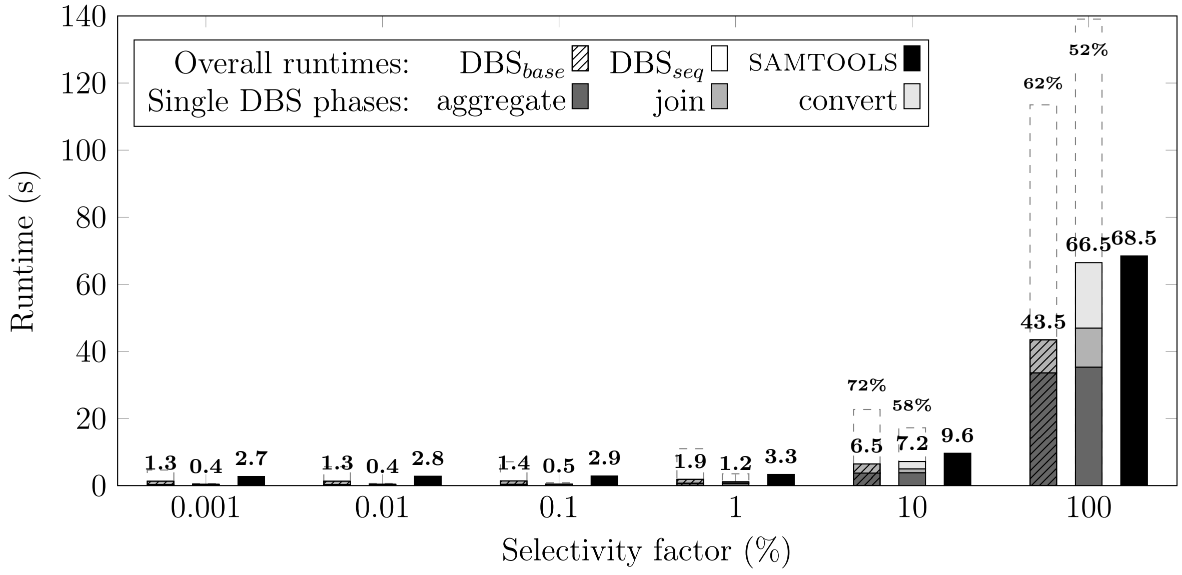
4.3.2 Runtime evaluation

In this section, we evaluate the runtime performance of our database approaches DBS$_{base}$ and DBS$_{seq}$ using the invisible join technique and compare it with SAMTOOLS. We use the same experimental setup as in Section 4.2.2. Our goal is to identify to which extent the database approaches benefit from the use of the invisible join technique compared to a hash join processing strategy. Moreover, we want to compare the overall runtime performance.

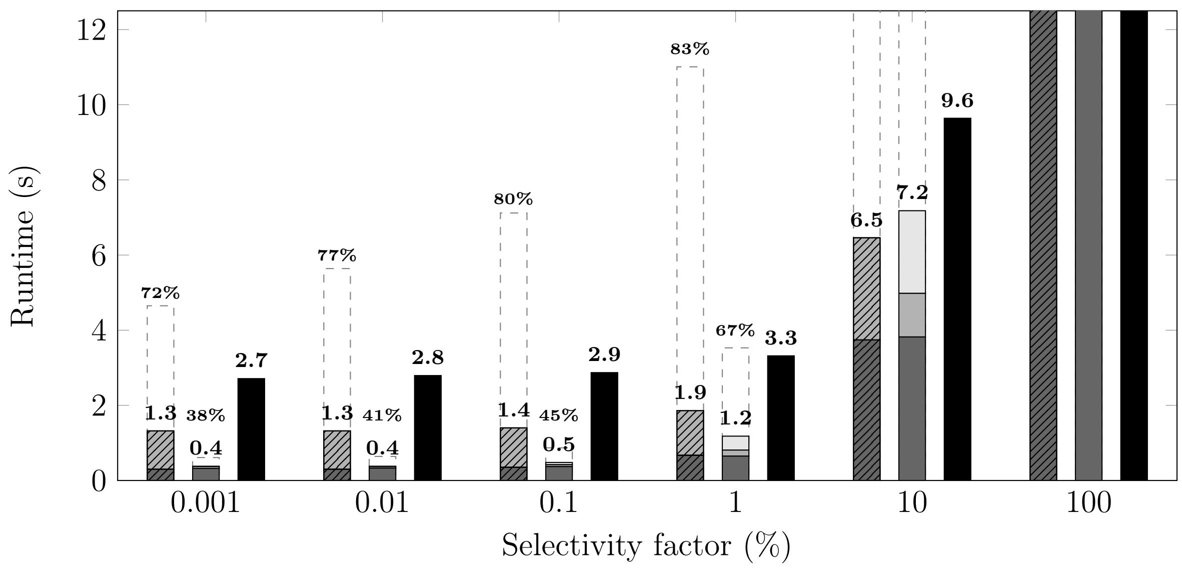
4.3.2.1 Results

We show the results in Figure 4.7. Since the runtimes between a query with highest selectivity (0.001%) and a query with lowest selectivity (100%) differ more than a magnitude, we report an overview on all runtime results in Figure 4.5a. To provide insights on queries having high selectivity, we zoom in the results in Figure 4.5b.

In both figures, we show for every selectivity factor the runtime results in following order: DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS. For DBS$_{base}$, we breakdown the overall runtime in the two phases for joining and aggregating the data. For DBS$_{seq}$, we additionally report the time to convert the data. The dashed bars indicate the overall runtimes of the database approaches from the previous experiment in Section 4.2.2. If the dashed



(a) SNV calling runtimes on chosen selectivity factors using invisible joins. The dashed bars indicate the runtime results from Figure 4.5. The numbers above the dashed bars show the overall runtime reduction compared to the results from Figure 4.5 in percent.



(b) SNV calling runtimes using invisible joins zoomed in. The dashed bars the dashed bars indicate the runtime results from Figure 4.5. The numbers above show the overall runtime reduction compared to the results from Figure 4.5 in percent.

Figure 4.7: Calling SNVs using invisible joins on chromosome 1 of a low coverage human genome using DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS. The invisible join technique in combination with between-predicate rewriting results in a significant performance improvement allowing to provide competitive analysis runtime performance. In particular on large genome regions (selectivity factor $> 10\%$), the aggregation phase within DBS$_{base}$ and DBS$_{seq}$ dominates the overall runtime.

bar is visible, we were able to reduce the runtime using the invisible join approach. If the dashed bar is not visible, the current bar hides it completely indicating that the runtime is equal or increased due to the invisible join.

Overall runtime. Using the invisible join, our database approaches are faster than SAMTOOLS independent of the chosen selectivity factor. On larger genome regions (selectivity factor $> 1\%$), DBS$_{base}$ is the fastest approach. On smaller genome regions, DBS$_{seq}$ is the fastest approach.

Impact on single database approaches. The invisible join strategy improves the runtime of both database approaches. Considering all genome ranges, DBS$_{base}$ benefits most from invisible join processing, which is indicated by the size of the dashed bars representing the absolute runtime savings.

4.3.2.2 Discussion

The conceptual change to replace the classical join processing by efficient scanning and lookup operations pays out. The fact that DBS$_{base}$ benefits most from this optimization is due to the fact that DBS$_{base}$ has to process all tuples in table *Sample_Base* independent of the requested genome range. Thus, an improvement of this processing step as we achieve it by replacing the probing in a hash table with a simpler between-predicate scan pays out most. However, using DBS$_{base}$ we can still not reach the performance of DBS$_{seq}$ on small genome regions (selectivity factor $< 10\%$). The reason for this difference is the additional effort of DBS$_{base}$ to filter the complete SB_RB_ID column with one billion rows. In contrast, DBS$_{seq}$ preselects the reads to be processed resulting in less sample bases to scan (cf. Table 4.1). Nevertheless, the effort for filtering and positional lookup is much lower than performing a hash join (cf. dashed bars and percent numbers in Figure 4.7).

The results show that the use of the invisible join is required to provide competitive analysis runtime on mapped read data sets when calling SNVs. Considering the runtime of all phases, the aggregation phase is now the longest phase within the database approaches on all genome regions. To further improve the overall runtime, we focus on speeding up the aggregation phase in next section.

4.4 Accelerating the aggregation phase

In this section, we discuss possibilities to speed up the aggregation phase within our database approaches. Note that our proposed optimizations do not change the query (cf. Listing 3.8) or its output. We focus our efforts on the implementation of the GROUP BY operator within the aggregation phase (cf. ⑫ in Figure 4.4), since the final filters ⑬ are usually less expensive as they operate on a reduced data set. In our previous experiments, we used a sort-based aggregation approach, because this reduces the effort to generate sorted results as required by our SNV calling query from Listing 3.8. Using a different aggregation strategy, we have to keep the possible sorting overhead of the result in mind.

Selectivity factor in %	# of bases to aggregate	
	reference	sample
0.001	2,493	10,217*
0.01	24,925	101,840*
0.1	249,251	1,025,273*
1	2,492,506	9,471,402*
10	24,925,062	101,940,994
100	249,250,621	1,022,483,513

Table 4.2: The number of sample bases to aggregate using DBS$_{seq}$ or DBS$_{base}$ is equal and depends on the chosen selectivity factor. Nevertheless, depending on the chosen aggregation strategy, i.e., sort-based or hash-based aggregation, we either have to sort more than one billion tuples or only 250 million tuples respectively to provide a sorted analysis result.

*Average of 30 different, randomly selected genome regions.

4.4.1 Optimization options

In our previous experiments, we already use an optimized sort-based aggregation strategy. According to our SNV calling query, we have to group by three different intermediate result columns, i.e., `GROUP BY C_NAME, RB_POSITION, RB_BASE_VALUE` (cf. Listing 3.8), increasing the effort for grouping, because we have to sort three columns to get the final sort order. To reduce the effort, we can create packed codes from the row values of the involved columns [63, 106] that reflect the final overall sort order. Thus, we can compute the correct sort order for the grouping in a single sorting step. Another strategy is to simply rewrite the GROUP BY clause to just use one attribute. In our use case, we can rewrite the GROUP BY clause into `GROUP BY RB_ID`, because RB_ID functionally determines C_NAME via RB_C_ID, RB_POSITION and RB_BASE_VALUE. Thus, we can restrict the sorting effort to one column without the effort to create packed codes.

However, especially on large genome regions, our sorting-based aggregation strategy introduces overhead, because we have to sort a large number of intermediate results before aggregation can take place (cf. Figure 4.7). On small genome ranges, the effort for sorting is moderate. For example, if we process the complete data set (selectivity factor 100%), the intermediate results comprise over one billion tuples that must be sorted. In Table 4.2, we show the number of sample bases that have to be aggregated and, thus, have to be sorted when we query differently sized genome ranges. Additionally, with increasing coverage of the data set, i.e., the number of sample bases that cover a single genome position, the sorting effort will increase.

An alternative aggregation strategy that does not require sorting of intermediate results is hash-based aggregation. The idea is to compute the group that a tuple belongs to using a hash function over the grouping attributes. Of course, this introduces effort for computing the hash value and looking up the aggregation bucket, but avoids sorting.

In order to provide sorted analysis results, we would only have to sort the resulting groups. The number of resulting groups only depends on the selectivity factor of the query. Thus, when querying the complete data set, we would only have to sort ca. 250 million resulting tuples instead of one billion.

Array-based aggregation. We can also apply the rewriting of the GROUP BY clause as done for the sort-based aggregation to the hash-based aggregation approach. Thus, we only have to hash the value of RB_ID to determine the hash bucket for aggregation. RB_ID is a primary key in our database schema and has a value range starting at zero. The maximum value is always $\#reference\_base\_tuples - 1$. In the evaluation data set within this chapter, the maximum value of SB_ID is 249,250,620. Since we often query ranges, we have to process reference bases that have a continuous range of RB_ID values. We already leverage this inherent characteristic for *between-predicate* rewriting (cf. Section 4.3.1) and can also leverage it for improving the hashing during hash-based aggregation. We can use the value of column RB_ID as index for an aggregation array. The size of the array is determined by the number of distinct values in the given RB_ID range. Using an array, we also avoid the need for maintaining and accessing a hash table. This idea is related to the approach by Krikellas et al. to use mapping directories for grouping attributes to determine an offset within an aggregation array [69]. Fortunately, we even have no need for a mapping directory avoiding a further indirection. In the best case, we can simply use the RB_ID value as index to the array. In the following, we discuss two special cases:

Index range not starting at zero. In the case that we process a genome region leading to a range of RB_IDs that does not start at zero, we subtract the minimum value of the range from the RB_ID to compute the actual array index.

Querying multiple contigs. Within a single contig, e.g., chromosome 1, we can guarantee that the range of RB_IDs is continuous. If we query multiple contigs, we can not directly ensure that the RB_ID range is continuous. For example, we import the contigs of a reference genome in order of appearance in the FASTA file (cf. Section 3.3.1). Thus, consecutive contigs, e.g., chromosome 1 and 2, will lead to a continuous value range of RB_ID values. Querying chromosome 1 and 3, we introduce a gap in the range of the size of chromosome 2. Thus, we have two options, either we process the two contigs sequentially or we create an array knowing that we do not access a large number of indexes in between.

The proposed array-based aggregation has a second advantage besides avoiding to sort a large number of intermediate results: We also avoid to sort the final result. The reason for this is that the RB_ID values directly reflect the required sort order. This is a direct consequence of our rewriting of the GROUP BY clause. Thus, the aggregation array inherently stores the aggregates in the final sort order. Nevertheless, the approach has two disadvantages:

Large memory consumption. We have to allocate an array of the size of the range of the RB_ID values. Querying larger genome regions leads to an increase of the array size. A strategy to avoid exceeding the available main memory is to split the range.

Need for synchronization. Since we now process the tuples in the order provided by the join phase, we cannot guarantee that different threads process distinct chunks of data as it was ensured by a sort-based aggregation strategy. Therefore, we have to synchronize threads that concurrently access the same array index. However, on large genome ranges, we expect the chance of concurrent access to be low.

Overall, we expect that array-based aggregation will improve the analysis runtime especially on large genome regions, since the overhead of sorting is moderate on smaller genome regions.

4.4.2 Runtime evaluation

In this section, we evaluate the runtime performance of our database approaches DBS_{base} and DBS_{seq} using the array-based aggregation technique and compare it with SAMTOOLS. We use the same experimental setup as in Section 4.2.2, except that we use the invisible join and array-based aggregation technique. Our goal is to identify to which extent the database approaches benefit from the use of the array-based aggregation technique compared to a sort-based aggregation strategy. Moreover, we want to compare the overall runtime performance.

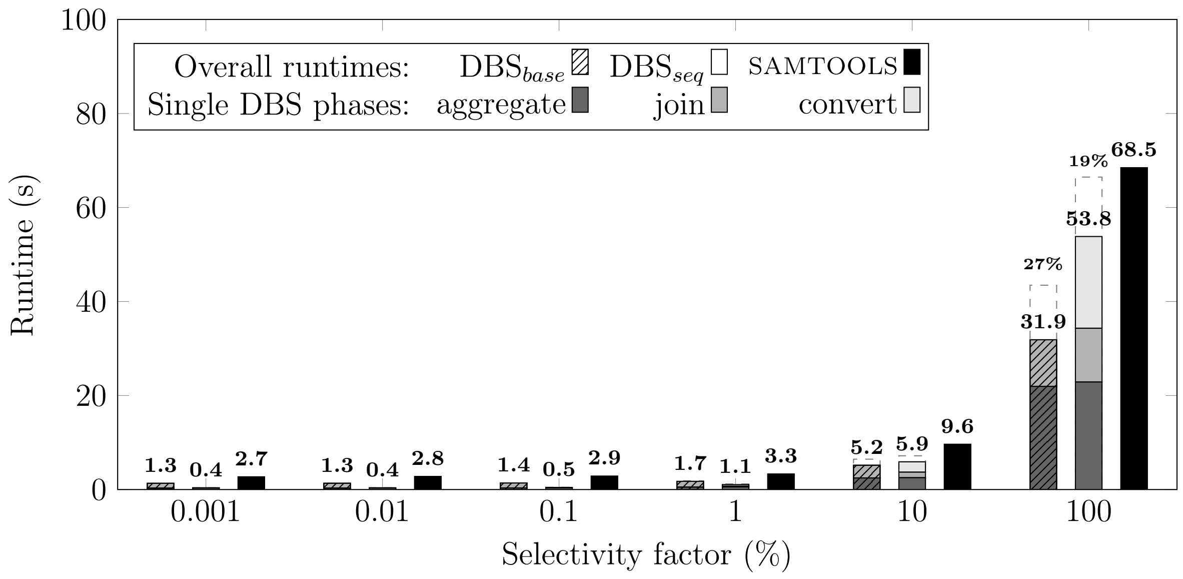
4.4.2.1 Results

We show the results in Figure 4.8. Again we report an overview on all runtime results in Figure 4.5a, because the runtimes between a query with highest selectivity (0.001%) and a query with lowest selectivity (100%) differ more than a magnitude. To provide insights on queries having high selectivity, we zoom in the results in Figure 4.5b.

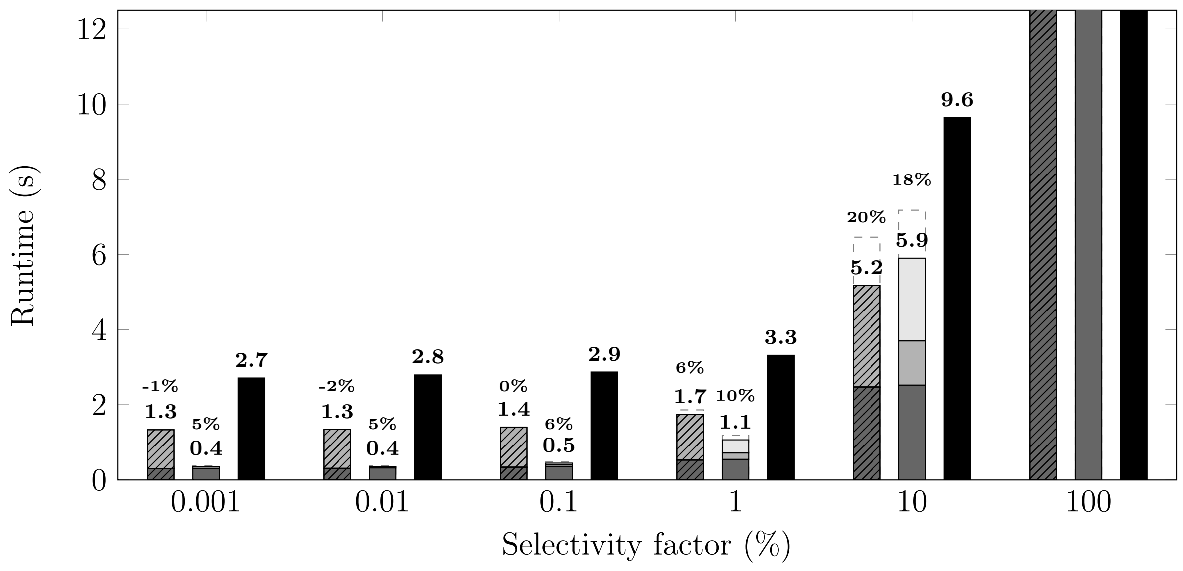
In both figures, we show for every selectivity factor the runtime results in following order: DBS_{base}, DBS_{seq} and SAMTOOLS. For DBS_{base}, we breakdown the overall runtime in the two phases for joining and aggregating the data. For DBS_{seq}, we additionally report the time to convert the data. The dashed bars indicate the overall runtimes of the database approaches from the previous experiment in Section 4.3.2. If the bar is visible, we were able to reduce the runtime using the array-based aggregation approach.

Overall runtime. Using the array-based aggregation, our database approaches become only faster on large genome regions (selectivity factor $> 1\%$). On smaller genome regions, we cannot detect a significant runtime improvement compared to a sort-based aggregation strategy.

Impact on single database approaches. There is no significant difference in runtime performance improvements between both database approaches, since the aggregation



(a) SNV calling runtimes on chosen selectivity factors using invisible joins and array-based aggregation. The dashed bars indicate the runtime results from Figure 4.7. The numbers above the dashed bars show the overall runtime reduction compared to the results from Figure 4.7 in percent.



(b) SNV calling runtimes using invisible joins and array-based aggregation zoomed in. The dashed bars indicate the runtime results from Figure 4.7. The numbers above the dashed bars show the overall runtime reduction compared to the results from Figure 4.7 in percent.

Figure 4.8: Calling SNVs using invisible joins and array-based aggregation on chromosome 1 of a low coverage human genome using DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS. The array-based aggregation can effectively reduce the runtime on large genome regions (selectivity factor >= 10%). On small genome regions (selectivity factor < 10%), the runtime savings are negligible.

Selectivity	Sort-based aggregation phase*		Array-based aggregation phase	
factor (%)	Runtime (s)	Share sorting (%)	Runtime (s)	Runtime savings (%)
0.001	0.305	1.3	0.305	0.0
0.01	0.302	2.0	0.310	-3.3
0.1	0.352	7.1	0.341	2.9
1	0.669	24.0	0.531	20.9
10	3.737	41.8	2.466	34.0
100	33.582	48.4	21.949	34.6

Table 4.3: Comparison of the runtime of a sort-based and array-based aggregation phase per selectivity factor. Array-based aggregation avoids sorting of intermediate results. Thus, array-based aggregation saves more runtime on larger genome regions, since the sorting runtime increases due to the growing size of intermediate results. On small genome regions, array-based aggregation sacrifices the potential runtime savings due to locking overhead.

*Based on DBS$_{base}$ results from the experiment in Section 4.3.2.

effort of both approaches is identical. In particular, on large genome regions (selectivity factor $> 1\%$), both approaches show a similar absolute speed up (cf. height of the dashed bars). Nevertheless, on small genome regions (selectivity factor $< 1\%$), we observe that especially DBS$_{base}$ appears to slow down due to the use of array-based aggregation. However, we observed slight deviations of runtimes during the invisible join phase compared to our experiments from Section 4.3.2. Since the dashed bars report the overall runtime savings and the runtime share of the aggregation phase on small genome regions is much smaller than the runtime share of the join phase using DBS$_{base}$, these deviations lead to the observed slow down.

4.4.2.2 Discussion

As expected, our array-based aggregation strategy can improve the overall runtime of our database approaches, because it avoids the sorting of intermediate results and final results. Especially, on large genome regions, we see the largest performance improvements compared to a sort-based aggregation strategy. On small genome regions, the runtime of array-based aggregation is comparable to a sort-based aggregation strategy.

In Table 4.3, we analyze the runtime differences between the aggregation phase of DBS$_{base}$ using sort-based or array-based aggregation in detail per selectivity factor. The data about sort-based aggregation is taken from the experiment in Section 4.3.2. The second and fourth columns show the runtime of the sort-based and array-based aggregation phase using DBS$_{base}$ respectively. The third column indicates the runtime share of the actual sorting within the sort-based aggregation phase. We can use this runtime share as rough estimation of the potential runtime savings using an array-based aggregation strategy, because we avoid the sorting at all. We can clearly see that the potential performance improvement that we can achieve grows when we query larger genome regions. On small genome regions, we have to sort small intermediate results, which is very efficient. In contrast, the runtime share of the sorting grows up to 50% querying the complete data set. This rough estimate corresponds well with the actually achieved runtime savings using the array-based aggregation strategy (cf. last column). However, we never achieve the maximum (estimated) runtime savings, because array-based aggregation introduces locking overhead, which completely sacrifices the potential runtime savings on small genome regions.

4.5 Applicability to disk-based DBMSs

For our evaluation, we use a main-memory DBMS. In this section, we discuss the applicability of our results to a disk-based DBMS. For this, we assume that our primary database resides on disk in a column-oriented data layout and a buffer manager manages the disk access (cf. Section 4.1.1). In the following, we describe how we can integrate the two different storage and processing strategies into a disk-based DBMS:

Applying DBS$_{seq}$. Applying the idea of DBS$_{seq}$, i.e., first convert the necessary data and then analyze it, to a disk-based DBMS requires following three steps:

1. Retrieve relevant read and reference sequence data from disk
2. Convert and buffer read and reference sequence data
3. Apply the execution strategies described in this chapter

This strategy resembles the idea by Röhm et al. [108] that we described in Section 3.2.2. Röhm et al. decided to encapsulate the complete functionality into a specialized UDF to reduce main memory consumption. We assume to have enough main memory for keeping the intermediate data representation in main memory. If we cannot guarantee to have enough main memory, we would process the data in chunks.

Applying DBS$_{base}$. Since DBS$_{base}$ already stores the data in a converted format, we can directly execute the proposed execution plan without the need to buffer a converted data representation. Since we process the data using the invisible join and array-based aggregation techniques mainly in sequential order, we benefit from the block-based access pattern of hard drives. The access pattern and the column-oriented storage layout guarantee that the data read within a block is actually needed for further processing.

Overall, we expect that a disk-based DBMS will provide worse runtime results, in particular on large data sets, since we face additional effort due to the indirections of the buffer manager. Due to this indirection we have to check whether data is already in the main memory or not. Techniques like pointer swizzling [48] could reduce this effort. Another idea, related to DBS$_{seq}$, is to mark an intermediate table to be in-memory and to offer a special in-memory execution engine [98].

4.6 Wrap up

In this chapter, we answered our third research question: *How can we process genome data sets as efficient as specialized analysis tools using relational DBMSs?* To this end, we examined the implementation space of the logical query plan behind our SNV detection query (cf. Figure 4.4). We learned that standard database operator implementations such as hash join and sort-based aggregation introduce overhead and bottlenecks that prevent a DBMS to be competitive with specialized analysis tools such as SAMTOOLS, in particular when we process large genome regions.

To overcome these limitations, we can leverage query and data characteristics that are inherent to the SNV detection problem and to our base-centric database schema. Thus, we can apply advanced processing schemes such as *invisible join* and a special kind of hash-based aggregation that we call *array-based aggregation*. Both techniques allow us to decrease analysis runtime. Compared to specialized analysis tools, we achieve at least competitive analysis runtime performance. Thus, using a main-memory DBMS and combining it with advanced relational processing strategies, we can mitigate the processing overhead of database systems for this specific workload allowing researchers to rethink their decision whether to use a DBMS for SNV analysis or specialized analysis tools.

Overall our results show that the choice of the database schema depends on the size of the analyzed genome region. If the standard use case is to analyze small genome regions, e.g., single genes, we would suggest to use the read-centric database schema. If bulk analysis of genomes is of interest, a base-centric database schema pays out due to the avoidance of data conversion. For mixed use cases, our results indicate that a base-centric database schema is the best choice to provide high average analysis speed.

Nevertheless, so far, we only considered a small data set within our experiments of this chapter due to the increased data volume of the base-centric database schema. In the following chapter, we will discuss compression of genome data stored in relational DBMS to improve the storage consumption and increase the scalability of our approaches to bigger data sets. Of course, compression introduces additional effort for data processing. Thus, we will also investigate how we can process compressed genome data efficiently within the database system.

5. Genome-specific storage and query optimizations for relational DBMSs

In Chapter 3, we devised a concept to express SNV detection as relational database query. Then, in Chapter 4, we investigated processing strategies for relational DBMSs to execute the SNV detection query efficiently on a reduced data set to not interfere with main memory limitations. Considering the growing amounts of genome data, it is critical to store and process mapped read data efficiently. For that reason, in this chapter, we examine compression schemes to reduce the storage consumption of mapped read data within a relational database. In essence, we put special focus on lightweight compression schemes that allow for compressing data while enabling the database system to process data values in their compressed form reducing decompression overhead. Furthermore, we will investigate how we can leverage data characteristics during query processing to decrease the analysis runtime. Our investigations help us to answer our fourth research question: *How can we store genome data sets using a relational DBMS as efficient as state-of-the-art flat-file approaches without sacrificing analysis speed?*

In Section 5.1, we start with a brief primer on data compression with special focus on lightweight compression schemes for database systems. Then, in Section 5.2, we perform an initial storage consumption analysis using standard lightweight database compression schemes. These standard compression schemes do not allow us to compete with compressed genome-specific flat-file formats that additionally use heavyweight compression schemes. To improve the storage consumption of a relational DBMS, we examine how we can integrate genome-specific compression schemes in Section 5.3. After devising lightweight genome-specific compression schemes, we introduce a technique that we call *base pruning* that leverages data characteristics during the processing of the SNV detection query. Finally, in Section 5.5, we examine the impact of our proposed techniques on storing and querying three real-world data sets.

5.1 A primer on data compression

In this section, we provide a brief overview on data compression within main-memory DBMSs. Using compression within a main-memory DBMS is a two-edged sword. On the one hand, we can reduce the amount of memory needed to store data allowing us to store more data. Furthermore, we can utilize the available memory bandwidth better, because we require less bits to transfer the same amount of information. This is especially helpful for memory-bound tasks, i.e., a scan that has less CPU effort and, thus, can be sped up if we can transfer more data values for processing per memory access. On the other hand, we have to keep decompression overhead low to not introduce processing overhead and sacrifice the potential performance improvements due to avoiding costly disk accesses. For that reason, we have to choose compression schemes in a main-memory DBMS carefully.

5.1.1 Heavyweight vs. lightweight compression

The general idea behind data compression is to replace patterns[1] within the original data by smaller codes. Decompression means to retrieve the original data for a given code. We can distinguish heavyweight and lightweight compression schemes that differ in the computational effort required for compressing and decompressing data.

Heavyweight compression schemes are designed for universal use and typically achieve high compression rates. Nevertheless, they are also accompanied with high computational effort. A typical heavyweight compression scheme is gzip [31] combining Lempel-Ziv [135] and Huffman [59] encoding. Huffman encoding assigns codes of variable length to patterns within the data. Patterns that occur more often are replaced by a smaller code, while those patterns that occur less often get a longer code assigned. Lempel-Ziv encoding creates a dictionary of patterns within the original data that get replaced by a fixed-length code. In case that a known pattern occurs, it is replaced by the code. To save more storage, information for decompression is not saved as long as we can compute it during decompression, e.g., the dictionary used for Lempel-Ziv encoding. Thus, we have to restore this information by decompressing the complete data set to access the original data values. If we want to access a specific data value[2], a heavyweight compression approach introduces significant overhead, since we have to decompress the complete data set. To reduce the overhead of decompression, blocking can be used as done by the BGZF format [111]. The idea is to chunk the input data and to compress each chunk, which requires less decompression overhead to look up a single data value if the chunk containing the value can be determined in advance.

In contrast to heavyweight compression schemes, lightweight compression schemes usually trade off faster data access for overall storage reduction of the complete data set. To this end, they are designed to leverage specific data characteristics for compression, e.g., a specific value domain or inherent relationships between data values. Due

[1]Assuming that the input data consists of characters, a pattern can be a sequence of characters in the input data, e.g., a single character or substring, or a repetition of characters within the input data.

[2]In the context of relational DBMSs, a data value is a specific attribute value of a tuple.

to their reduced processing overhead, we often use lightweight compression schemes in main-memory DBMSs, since fast data access is critical. Of course, lightweight compression schemes will only achieve competitive compression ratios compared to heavyweight compression schemes, if the data shows the required data characteristics. Using a column-oriented storage layout, we can improve the effectiveness of lightweight compression schemes (cf. Section 4.1.2), since data values of the same domain are stored consecutively in memory [2]. Moreover, a column-oriented storage layout facilitates the processing of compressed data values [2] avoiding the need for decompression of data values at all. We will discuss some lightweight compression schemes used within this thesis in the following section.

5.1.2 Lightweight data compression schemes

In this section, we give a brief introduction to standard lightweight compression schemes that are available within our evaluation DBMS CoGaDB and used throughout our storage evaluation.

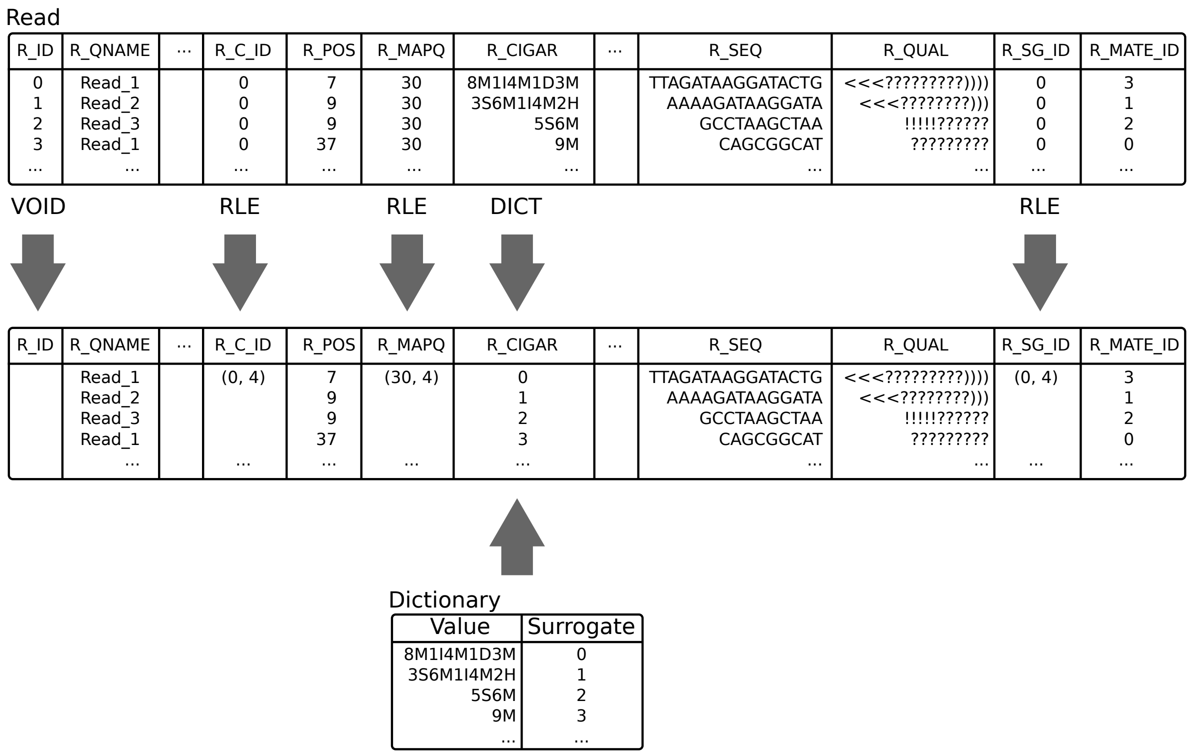
5.1.2.1 Run-length encoding - RLE

Run-length encoding (RLE) replaces runs of the same value with a $(value, length)$ tuple indicating the run value and the length of the run. It is especially well suited to compress sorted data, since the chance of runs of the same value increases. In Figure 5.1, we give an example based on our read-centric database schema (cf. Section 3.3.1). Column R_SG_ID encodes which read belongs to which sample genome. Since we import all reads of the sample genome data set in bulk, we automatically generate runs of the same R_SG_ID value that can be compressed well. In order to look up a single value by tuple id, we have to sum up all *length* values until the sum is greater or equal to the searched tuple id. Then, we return the *value* of the respective run as tuple value. A scan can be done directly on the run values. For each run that satisfies the predicate, we return the tuple ids compressed by the respective run.

5.1.2.2 (Bitpacked) dictionary encoding - (BIT)DICT

The idea of dictionary encoding (DICT) is to replace variable length inputs such as strings by fixed length codes. To this end, data values are looked up in a dictionary and replaced by the respective surrogate code. In Figure 5.1, we show an example based on the read-centric database column R_CIGAR (cf. Section 3.2.1). Each CIGAR string is replaced by a surrogate value and stored together with the respective surrogate value in a dictionary. In case that a value appears more than once, DICT leads to compression of data if the used code is smaller than the replaced value. In our example, we do not achieve any compression, since every CIGAR string appears once and has to be stored in the dictionary.

In real world data sets, specific CIGAR strings appear quite often, e.g., $100M$ that indicates that every of the 100 bases of a read was mapped to a reference base. If we use 32-bit integers as word length for the surrogate keys, we achieve a slight compression,



R_ID	R_QNAME	...	R_C_ID	R_POS	R_MAPQ	R_CIGAR	...	R_SEQ	R_QUAL	R_SG_ID	R_MATE_ID
0	Read_1		0	7	30	8M1I4M1D3M		TTAGATAAGGATACTG	<<<?????????))))	0	3
1	Read_2		0	9	30	3S6M1I4M2H		AAAAGATAAGGATA	<<<????????)))	0	1
2	Read_3		0	9	30	5S6M		GCCTAAGCTAA	!!!!!??????	0	2
3	Read_1		0	37	30	9M		CAGCGGCAT	?????????	0	0
...	...		...	...	...	...		...	...	...	...

R_ID	R_QNAME	...	R_C_ID	R_POS	R_MAPQ	R_CIGAR	...	R_SEQ	R_QUAL	R_SG_ID	R_MATE_ID
	Read_1		(0, 4)	7	(30, 4)	0		TTAGATAAGGATACTG	<<<?????????))))	(0, 4)	3
	Read_2			9		1		AAAAGATAAGGATA	<<<????????)))		1
	Read_3			9		2		GCCTAAGCTAA	!!!!!??????		2
	Read_1			37		3		CAGCGGCAT	?????????		0
	...		...	...	...	...		...	...	...	...

Value	Surrogate
8M1I4M1D3M	0
3S6M1I4M2H	1
5S6M	2
9M	3
...	...

Figure 5.1: Compression of chosen read-centric database columns. We can compress runs of the same value effectively using RLE (e.g., R_C_ID, R_MAPQ and R_SG_ID). Primary key columns such as R_ID can be effectively compressed using VOID compression. String columns such as R_CIGAR can be encoded using DICT to provide fixed length codes facilitating processing. If we can replace strings with smaller surrogate values and the string is stored more than once, we achieve compression.

since the string consists of four characters and a \0 terminator each encoded with one byte. Thus, we save one byte per $100M$ CIGAR. We can increase the effectiveness of DICT for domains with a smaller amount of distinct values than the chosen word length of the surrogate values offers. For example, if we can estimate that we only require 256 different CIGAR values, we could pack 4 surrogates into a 32-bit word reducing the required amount of storage further. This is especially interesting if we have value domains with less than 4, 8 or 16 values. Then, we can use a single byte to represent multiple values. We call such a dictionary compression scheme bit-packed dictionary encoding (BITDICT). Note that BITDICT can become problematic if the number of distinct values increases beyond the number of possible values that can be encoded. Then, we require a complete recoding, which is time consuming or we have to chunk the data using a bigger dictionary for the new chunk of data.

To look up the original value, we have to search the dictionary for the surrogate key. To improve the look up performance, we could also use a lookup array (if the surrogates can be used as indexes). Of course this decreases the potential compression potential.

Furthermore, BITDICT has the drawback of increasing the access time, since we have to additionally extract the surrogate value from the word. It is also possible to process compressed (BIT)DICT data. To this end, we have to rewrite the predicate into the surrogate value. An equality predicate for string $9M$ gets rewritten into a equality predicate for surrogate value 3. *Greater* or *lesser than* operations are only possible if the surrogate values reflect the sort order of the original values.

5.1.2.3 VOID compression

VOID compression is inspired by MonetDB's VOID-headed BATs [60]. MonetDB uses internal data structures called binary association table to encode columns. Each value in a column consists of two values: the *oid* that is the tuple id and the actual *value*. Thus, it is not necessary that the values in the column follow the tuple id order. Nevertheless, in case they do, i.e., the first value belongs to tuple id 0, the second to 1 and so forth, storing the *oid* explicitly is overhead, since the position in the column is the *oid*. Thus, MonetDB converts the *oid* array into a *void* array, storing no *oid* at all as it can be derived from the values position within the *value* array. In our experiments, we found that primary key columns containing numeric, auto-incremented values have the same data characteristic as *void* arrays in BATs. Thus, we implemented a compression scheme called VOID compression to apply on our primary key columns. If we access a specific tuple id in a VOID compressed column, we simply return the tuple id. And a scan returns the tuple ids that satisfy the predicate. In Figure 5.1, we give an example of the VOID compressed primary key column R_ID within our read-centric database schema. The result of VOID compression is that we store no data at all to represent column R_ID, since we can derive the values from the inherently given tuple id.

5.2 Initial storage consumption analysis

In this thesis, we identified two general approaches to store mapped reads (cf. Chapter 3). We can either use the *read-* or the *base-centric* database schema. In the read-centric database schema, we store mapped reads as strings. In the base-centric database schema, we store the single bases of the reads individually. Thus, the base-centric database schema allows us to perform SNV calling without the need of UDFs, since we already store the required mapping information explicitly that is implicit stored using strings within the read-centric database schema. In the qualitative assessment of the base-centric database schema (cf. Section 3.3.5), we already identified that the explicit storage of position and mapping information will increase the data volume. For that reason, we performed our SNV calling experiments in Chapter 4 on a reduced data set to not interfere with main memory constraints.

In this section, we assess the storage consumption of both database schemes in detail using our database approaches DBS$_{seq}$ and DBS$_{base}$ respectively. We use CoGaDB (cf. Section 4.2.2.1 for details on the system setup) to evaluate the storage consumption of DBS$_{base}$ and DBS$_{seq}$ storing a complete human genome data set. The sample genome data set[3] comprises 144,534,109 mapped reads consisting of 13,917,235,121 bases. The reference genome comprises 86 reference sequences (contigs) consisting of 3,137,454,505 bases. We apply standard lightweight compression schemes that are available within CoGaDB, i.e., run-length encoding (RLE), void compression (VOID), bit-packed (BIT-DICT) and standard dictionary compression (DICT), to compress the base- and read-centric database.

In the following section, we assess the applicability of standard compression schemes available in relational database systems (cf. Section 5.1.2) to both database schemata and report the used compression scheme for every column. Then, in Section 5.2.2, we report and discuss the results of our storage consumption experiment and derive a concept to further decrease the storage consumption.

5.2.1 Applicability of standard compression schemes

In Table 5.1, we list per compression scheme which column of the base- or read-centric database schema is compressed using the respective compression scheme. Since base-centric and read-centric database schema share the same data hierarchy (cf. Section 3.3.3), we apply the same compression schemes to the shared columns of both schemata indicated by a cell spanning column two and three. In the last row, we list all uncompressed columns.

Within our database approaches, we have two different kinds of data to compress. We distinguish primary and foreign key data that is used to ensure referential integrity and is not present in flat files. The other columns contain the actual payload (genome) data, i.e., mapped read data and reference sequences. These columns are printed in bold in Table 5.1.

Compression of primary and foreign key columns

Based on the data characteristics, we can apply standard compression schemes such as VOID and RLE to nearly every key column (ending with *ID*) indicating that we can reduce the overhead due to primary and foreign keys effectively. The reason for this is that all primary key columns contain a continuous sequence of numbers starting at zero, which is well suited for VOID compression. The foreign key columns often contain runs of the same values. For example, column SB_READ_ID encodes which base belongs to which read. Since we store all bases of the same read consecutively, we have a sequence of the same SB_READ_ID value.

The only exceptions are columns MATE_ID and SB_RB_ID, since they do not show a characteristic that we can leverage for compression. Column MATE_ID is a foreign key to another read leading to random values. Column SB_RB_ID of the base-centric database schema does not contain runs of the same value. Thus, RLE will not work.

[3]data is available at ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00096/

Compression scheme	Compressed database schema columns	
	read-centric	base-centric
VOID	RG_ID, C_ID, SG_ID, R_ID	
	-	RB_ID, SB_ID
RLE	C_RG_ID, R_SG_ID, R_C_ID, **R_MAPQ**	
	-	RB_C_ID, **SB_INSERT_OFFSET**, SB_READ_ID, ***_HARD_CLIP_***
BITDICT	**R_FLAG**	
	-	**RB_BASE_VALUE**, **SB_BASE_VALUE**
DICT	**R_CIGAR**	-
none	R_MATE_ID, **R_QNAME**, **R_POS**, **R_TLEN**, **C_NAME**, **SG_NAME**, **RG_NAME**	
	R_SEQ, **R_QUAL**, **C_SEQ**	SB_RB_ID, **SB_BASE_CALL_QUAL.**, **RB_POSITION**, ***_SOFT_CLIP_***

Table 5.1: Overview on applied compression schemes to columns of the base- and/or read-centric database schema. Bold printed columns indicate actual payload data, i.e., mapped reads to store. The explicit encoding of mapping information using the base-centric database schema allows us to apply bit-packed dictionary compression to base value columns, which is not possible without further modifications using a read-centric database schema.

Compression of genome data

Considering the actual genome data, we compress columns R_MAPQ, R_FLAG and R_CIGAR. To compress column R_MAPQ, we leverage that the values of subsequent reads might be equal allowing us to apply RLE. To compress R_FLAG, we use BIT-DICT compression to encode the limited set of possible values more efficiently. To compress R_CIGAR, we use DICT to replace common CIGAR strings such as *100M* by a smaller surrogate value.

All other column characteristics prevent the use of compression schemes. R_POS indicates the start mapping position of a read that can be random. The same randomness is present within R_TLEN values indicating the distance between two paired-end reads. Thus, we do not apply any compression. Another characteristic of genome data is that it mostly consists of unique strings, e.g., reads are mostly unique to prevent a possible bias within analysis results [30] leading to unique strings in columns R_SEQ and R_QUAL. Also C_SEQ contains large unique strings. Furthermore, names (RG_NAME, SG_NAME, C_NAME, R_QNAME) are also mostly unique. Thus, using a dictionary encoding scheme likely increases the data volume, because we have to assign a unique surrogate value to every unique value (cf. Section 5.1). Consequently, we do not apply any compression scheme to these columns.

Considering the storage of DNA sequences and the small number of possible base value characters, bit-packing of single base characters of a DNA sequence is an intuitive compression. Assuming four different possible base values[4], i.e., A, C, T and G, we would require 2 bit to encode them instead of 8 bit per character. Nevertheless, reference base sequences often contain Ns requiring a further bit, i.e., 3 bits per base value, to encode the possible base values. In the read-centric database schema, DNA sequences are stored as strings, which requires the introduction of a special string compression scheme to enable BITDICT of base values within DNA strings. In contrast, the base-centric schema encodes mapping information per base explicitly. Thus, we do not store unique DNA sequence strings within a base-centric database, but the single characters, which allow us to apply BITDICT without any further modifications assuming a column-oriented storage layout and processing engine. Consequently, we can apply standard BITDICT to columns SB_BASE_VALUE and RB_BASE_VALUE. Furthermore, we can compress parts of the CIGAR information stored in columns SB_INSERT_OFFSET and hard clip related columns (*_HARD_CLIP_*) within the base-centric database schema (cf. Section 3.3.1) using RLE. Column SB_INSERT_OFFSET contains many zeros, since insertions are rare compared to the matching bases. Also hard clips are rare and, thus, the hard clip lengths are often zero.

Conclusion

Our overview on applied compression schemes for the single columns of the base-centric and read-centric database schema reveals a trade-off between the ability to compress base values using standard BITDICT to save storage compared to the read-centric database schema and the need for additional foreign key columns to store mapping information in the base-centric database schema explicitly. On the one hand, the explicit encoding of mapping information facilitates the compression of genome data using standard lightweight compression schemes. Thus, using the base-centric database schema, we can compress base values using BITDICT, while we are not able to compress DNA sequences in the read-centric database schema without modifications of existing compression schemes. On the other hand, the base-centric database schema introduces additional foreign key columns such as SB_RB_ID and SB_READ_ID to enable the explicit storage of mapping information. These columns will inherently introduce storage overhead. While we can use RLE to compress SB_READ_ID, we have no applicable compression scheme for SB_RB_ID. It is not clear yet, whether we can mitigate the additional storage consumption due to the foreign key columns to benefit from the potential storage savings. To this end, we perform an initial storage consumption analysis in the next section.

5.2.2 Storage evaluation

With our initial storage evaluation, we pursue two goals. First, we want to compare the storage consumption of our database approaches with genome-specific flat file formats

[4]The complete alphabet comprises 16 characters [97], but base values different than A, C, T or G will likely have low quality limiting their use for further analyses such as SNV calling.

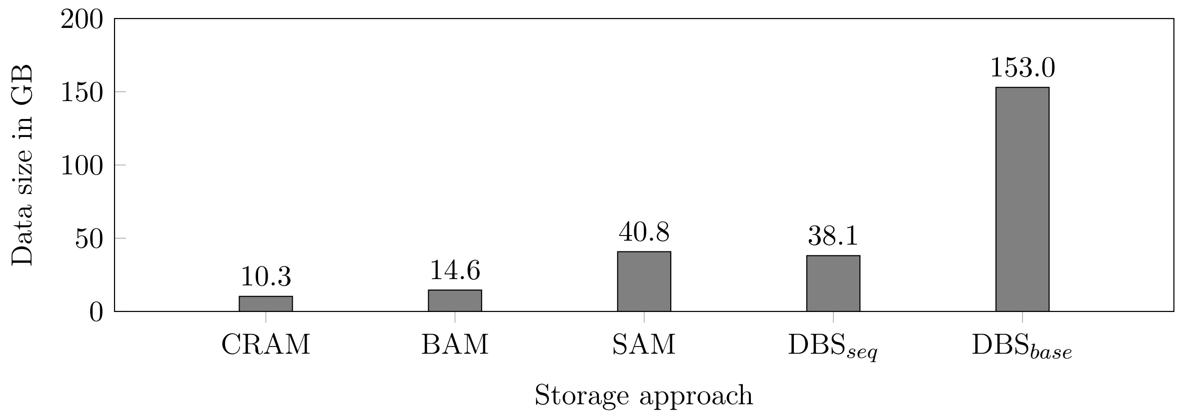


Figure 5.2: A DBS$_{seq}$ approach requires three to four times more storage than the compressed flat file formats CRAM and BAM, which is due to missing compression capabilities for DNA sequencing data. DBS$_{base}$ allows to apply BITDICT to base values, but increases the data volume by more than a magnitude due to explicitly storing position and mapping information.

that rely on heavyweight compression to quantify the storage overhead accompanied with our database approaches. We expect that heavyweight compressed flat files achieve a higher storage reduction compared to our database approaches, since we can apply lightweight compression schemes only to a limited number of database columns (cf. Section 5.2.1), while the heavyweight compression schemes can compress the complete data set due to their universal applicability. Second, we want to examine the differences in storage consumption between the base-centric and the read-centric database schema in detail to quantify the trade-off identified in Section 5.2.1 between the ability to compress base values and the additionally required foreign keys.

To this end, we import the complete human genome data set (cf. Section 5.2) using DBS$_{seq}$ and DBS$_{base}$ applying the compression configuration shown in Table 5.1. We compare the storage size of the database approaches with the storage requirements of the flat file formats SAM, BAM [111] and CRAM [28] for mapped read data. SAM is the uncompressed flat file format serving as baseline. BAM is the binary SAM version that applies bit-packing to single base values. In addition, BAM applies heavyweight compression using the blocked gzip file format (BGZF), which is a blocked compression scheme on top of the original gzip format [31]. CRAM additionally applies a genome-specific compression scheme called reference-based compression to compress DNA sequences of mapped read [56]. To compress the reference sequence data, we use the gzip file format and include the size in the reported storage requirements. In case of uncompressed SAM, we report the uncompressed storage size of the reference FASTA file. We report the results in Figure 5.2.

5.2.2.1 Results

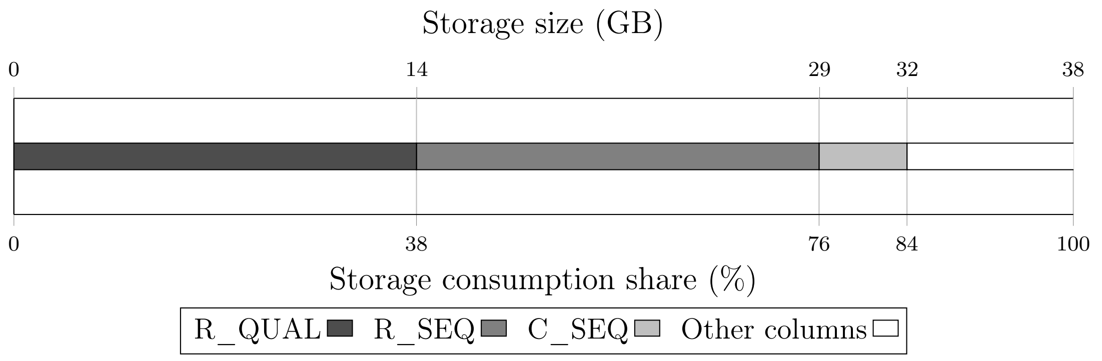
Using the storage size of the uncompressed SAM format as baseline, CRAM and BAM reduce the storage size by 75% and 65% respectively. The difference of 10% or 4 GB is due to the use of reference-based compression within CRAM assuming that this is the only storage optimization compared to BAM. Since BAM internally applies bit packing to sample base values, which leads to a theoretical storage reduction of 20% or 8.35 GB[5], the reference-based compression reduces the storage size of the SAM formatted data by 30%, which is 40% of the overall storage savings of CRAM. The other 60% of the storage savings are due to the use of heavyweight compression.

Considering our database approaches, DBS_{seq} saves only 2 GB of storage compared to SAM, while DBS_{base} increases the storage consumption by a factor of 3.75. Compared to CRAM, DBS_{seq} requires four times more storage space and DBS_{base} requires 15 times more storage space. Consequently, we cannot use one of our database approaches as competitive alternative for compressing genome data. Moreover, we would dramatically increase the storage size using DBS_{base}.

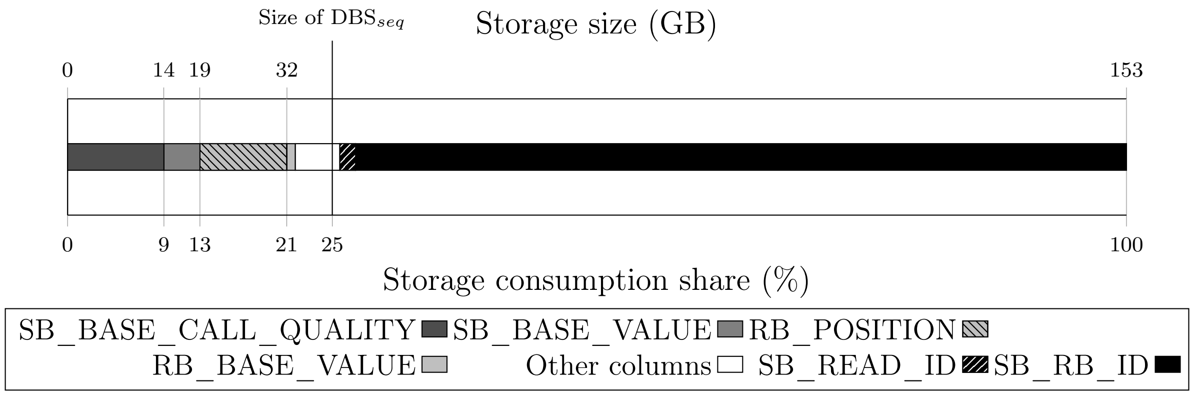
In Figure 5.3, we show in detail which columns require the most storage using DBS_{seq} or DBS_{base}. We assign all columns that have too small single shares to category *other columns*. The breakdown for DBS_{seq} in Figure 5.3a confirms that columns R_SEQ and R_QUAL, the actual mapped read data, are responsible for more than 75% of the overall storage. This explains the bad compression ratio compared to the uncompressed SAM format. In contrast, using DBS_{base}, columns SB_RB_ID and SB_READ_ID, both foreign key columns, require more than 75% of the overall storage. Column SB_RB_ID encodes the mapping between single sample and reference bases. Column SB_READ_ID encodes the read containment of every single base. While we can compress column SB_READ_ID using RLE, the data characteristics of column SB_RB_ID prevent us from applying a compression scheme (cf. Section 5.2.1). Thus, the explicit storage of mapping information leads to the large storage consumption increase if we use DBS_{base}. The storage savings due to compressing columns RB_BASE_VALUE and SB_BASE_VALUE do not outweigh the large storage increase due to column SB_RB_ID. Consequently, the base-centric database schema appears not to be a good choice as primary database schema with regard to storage consumption.

If we do not take the storage size of columns SB_READ_ID and SB_RB_ID into account within our storage comparison between DBS_{base} and DBS_{seq}, we find that the storage consumption of DBS_{base} is similar to DBS_{seq}. However, we already apply BITDICT to compress base values within DBS_{base}. Thus, we would expect that we save storage. Indeed, the impact of BITDICT on the storage size for read sequences can be seen by comparing the absolute storage requirements for R_SEQ (cf. Figure 5.3a) and SB_BASE_VALUE (cf. Figure 5.3b). The difference is roughly 10 GB. Nevertheless, these storage savings are exceeded by the storage requirements of the additional

[5]Assuming 3 bit per base and 32-bit words, we can store 10 bases per 4 byte, which is roughly 8.35 GB for 13,917,235,121 bases.



(a) Breakdown of storage consumption of DBS_{seq} by column.



(b) Breakdown of storage consumption of DBS_{base} by column.

Figure 5.3: Breakdown of storage consumption per database column. Using DBS_{seq}, columns R_SEQ and R_QUAL require more than 75% of the overall storage. Using DBS_{base}, column SB_RB_ID and SB_READ_ID, foreign key columns, consume more than 75% of the overall storage. Not considering the foreign columns, the storage consumption of DBS_{seq} and DBS_{base} is comparable.

column RB_POSITION. This column encodes the position within the reference sequence for every single reference base and requires additional 13 GB. Since column RB_POSITION shows the same data characteristics as SB_RB_ID, we cannot apply one of our standard compression schemes.

5.2.2.2 Discussion

The results show that storing genome data efficiently using a relational DBMS is hard to achieve using our available lightweight compression schemes. Since we cannot apply lightweight compression schemes to columns storing mapped reads within DBS_{seq} (cf. Section 5.2.1), the storage savings of DBS_{seq} are only marginal compared to the storage savings of BAM or CRAM. Using DBS_{base}, we are at least able to apply BITDICT to compress base values. However, the additional storage requirements due to explicitly storing position and mapping information outweigh the storage savings. Nevertheless, if we consider the data characteristics of columns SB_RB_ID and RB_POSITION, we are confident to compress these columns, since the data values are not random, but contain runs of continuous sequences of numbers that are incremented by one (cf. Figure 3.9 for an example). A modified and, thus, genome-specific RLE could be applicable to leverage this data characteristic. If we can successfully reduce the storage overhead of columns SB_RB_ID and RB_POSITION, a base-centric database schema could reduce the overall storage consumption of a relational DBMS using standard lightweight compression schemes. Thus, we do not have to fall back to a read-centric database schema and can avoid to use UDFs for SNV calling (cf. Section 3.3) on large data sets. In any case, if we reduce the storage consumption of data stored using a base-centric database schema, we will directly reduce the memory footprint of our DBS$_{seq}$ approach that uses the base-centric database schema as intermediate data representation.

A further strategy to reduce the storage size of genome data stored via a base-centric database schema is to integrate reference-based compression that allows for better compression than bit packing. Reference-based compression exploits mapping information [56]. The base-centric database schema makes mapping information explicitly available within a relational DBMS already allowing us to perform SNV calling via SQL (cf. Chapter 3 and Chapter 4). Thus, a base-centric database schema should also facilitate the lightweight integration of reference-based compression into a relational DBMS, since the required mapping information is already explicitly encoded and accessible. In contrast, a read-centric database schema stores DNA sequences and related mapping information implicitly via strings, which requires specialized string compression schemes to integrate reference-based compression. Such a string compression approach would internally leverage the mapping information, but we would still require to convert the genome data to enable analyses such as SNV calling via SQL (cf. Chapter 3). Of course, we could use highly specialized UDFs that directly operate on the compressed data, but this would only increase the complexity of the used UDFs and still prevents a DBMS to operate efficiently on the compressed data. Consequently, we focus on reducing the storage consumption of the base-centric database schema.

5.3 Lightweight genome-specific database compression schemes

In this section, we explain how we integrate reference-based compression, a genome-specific compression scheme, in a lightweight way into a relational DBMS based on our base-centric database schema. Moreover, we introduce Delta+RLE encoding to efficiently reduce the overhead due to explicitly storing position and mapping information.

5.3.1 Reference-based compression for column stores

Reference-based compression leverages the similarity between base sequences [56]. In case of read mapping data, we map small reads against a larger reference sequence with

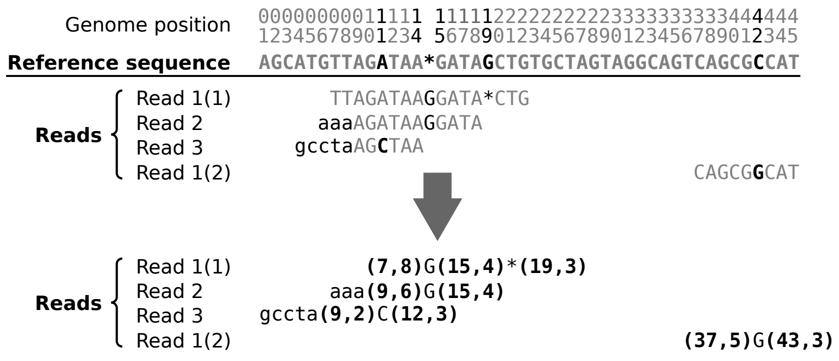
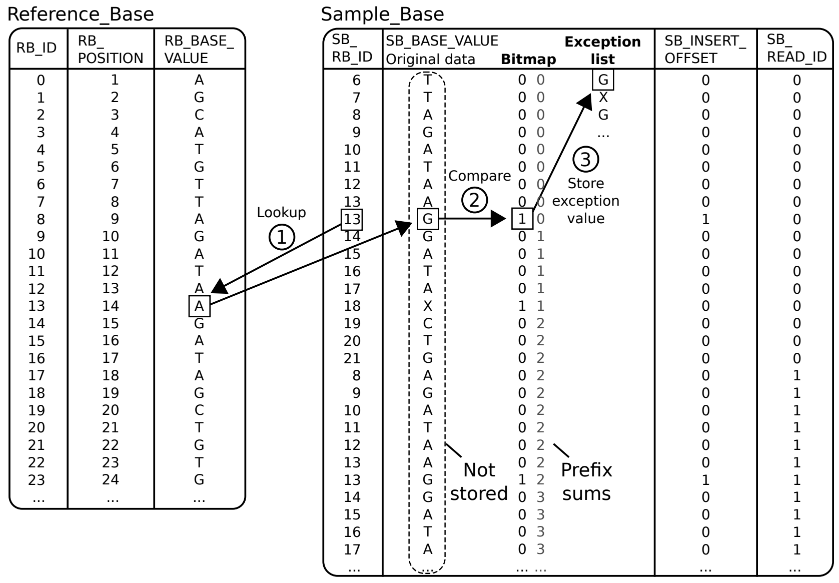


Figure 5.4: The matching parts of reads (gray bases in the upper part) are encoded as (*position*, *length*) tuples with respect to the reference sequence (bold printed in the lower part). Differing bases are kept to restore the original read sequence.

the goal to achieve a high concordance. Thus, the read and the reference sequence have a large part of their base sequence in common. The idea is now to encode (parts of) the reads using a start position and a length value referencing the respective base sequence in the reference sequence. We can directly reuse the read mapping information for this purpose. In Figure 5.4, we show an example of the general idea behind reference-based compression. We replace sequences of matching bases by a (*position*, *length*) tuple. Differing bases are stored separately to enable the reconstruction of the original read.

5.3.1.1 Compression concept

To integrate reference-based compression into a DBMS, we have to leverage the mapping information within mapped read data. In particular, we have to know which sample base is mapped to which reference base. Within the base-centric database schema, we encode this relationship explicitly via column SB_RB_ID. In Section 4.3.1, we already showed that we can use the SB_RB_ID value as index to look up reference bases. We use this characteristic to speed up positional lookups during the invisible join. We can also leverage this inherent characteristic to integrate reference-based compression into a DBMS in a lightweight way. Thus, we can reduce the size of column SB_BASE_VALUE while ensuring efficient data access. In Figure 5.5, we show the basic principle behind reference-based compression in a column store using the base-centric database schema. Instead of storing each sample base value in column SB_BASE_VALUE as single character requiring one byte of storage, we first check whether the base value is different from the reference base value or not. We can do the required lookup of the reference base value efficiently using the foreign key value of column SB_RB_ID as lookup index for column RB_BASE_VALUE ①. In a *bitmap* ②, we mark bases that are different with a 1 and bases that are equal to the reference bases with a 0. Furthermore, we append a differing base to an *exception_list* ③. Thus, we can use the prefix sum, i.e., the sum of ones and zeroes encoded in the bitmap before a specific index, as index to look up an exception value in the *exception_list*.



Reference_Base

RB_ID	RB_POSITION	RB_BASE_VALUE
0	1	A
1	2	G
2	3	C
3	4	A
4	5	T
5	6	G
6	7	T
7	8	T
8	9	A
9	10	G
10	11	A
11	12	T
12	13	A
13	14	$\boxed{A}$
14	15	G
15	16	A
16	17	T
17	18	A
18	19	G
19	20	C
20	21	T
21	22	G
22	23	T
23	24	G
...	...	...

Sample_Base

SB_RB_ID	SB_BASE_VALUE				SB_INSERT_OFFSET	SB_READ_ID
	Original data	Bitmap		Exception list		
6	T	0	0	$\boxed{G}$	0	0
7	T	0	0	X	0	0
8	A	0	0	G	0	0
9	G	0	0	...	0	0
10	A	0	0		0	0
11	T	0	0		0	0
12	A	0	0		0	0
13	A	0	0		0	0
$\boxed{13}$	$\boxed{G}$	$\boxed{1}$	0		1	0
14	G	0	1		0	0
15	A	0	1		0	0
16	T	0	1		0	0
17	A	0	1		0	0
18	X	1	1		0	0
19	C	0	2		0	0
20	T	0	2		0	0
21	G	0	2		0	0
8	A	0	2		0	1
9	G	0	2		0	1
10	A	0	2		0	1
11	T	0	2		0	1
12	A	0	2		0	1
13	A	0	2		0	1
13	G	1	2		1	1
14	G	0	3		0	1
15	A	0	3		0	1
16	T	0	3		0	1
17	A	0	3		0	1
...	...	...	...		...	...

Figure 5.5: Reference-based compression uses the existing foreign key relationship to compress sample bases ①. Differing sample bases are marked ② and stored in an exception list ③.

Data retrieval. We show the lookup of single base values in a reference-based compressed column in Algorithm 5.1. For simplicity, we assume that we can access specific indexes of the *bitmap*, *exception_list* and columns SB_RB_ID and RB_BASE_-VALUE via the array access operator [ ]. To retrieve a value of a given tuple id *tid*, we check whether the bitmap is set at the given index *tid* (line 2). If the bit is set, i.e., 1, we compute the PREFIXSUM (line 3) and use it as index to look up the base value in the exception list (line 4). In case that the bitmap value is 0 at the given *tid*, we use the foreign key from column SB_RB_ID (line 6) to look up the base value from column RB_BASE_VALUE (line 7).

Storage consumption. Using a simple bitmap, we have to store one bit for every base within column SB_BASE_VALUE. Considering our human genome data set used in our initial evaluation (cf. Section 5.2), we have to store 13,917,235,121 bits that are ca. 1.74 GB. Furthermore, we have to store the mismatching bases within the exception list. Our evaluation data set contains 53,149,605 mismatching bases. Using an internal BITDICT column with 3 bit per base value and a word size of 32 bit, we need ca. 22 MB to store the mismatching base values. Summing up the size of the bitmap and the exception list, we already achieve a reduction of the storage size by a factor of 3 compared to BITDICT using 3 bits and 32-bit words for all bases, which requires 5.56 GB.

5.3.1.2 Reducing the storage consumption

To further reduce the storage consumption, we can use a compressing word-aligned hybrid (WAH) bitmap [134] instead of a plain bitmap. In a WAHBitmap, we organize

Algorithm 5.1 Lookup of values stored in a reference-based compressed column

```
Input: The tid to look up in the column.
Output: The value of the tid encoded by the column.
1: function getValue(tid)
2:     if $bitmap[tid] = 1$ then
3:         $exception\_idx \leftarrow$ prefixSum($bitmap, tid$)
4:         return $exception\_list[exception\_idx]$
5:     else
6:         $rb\_tid \leftarrow SB\_RB\_ID[tid]$
7:         return $RB\_BASE\_VALUE[rb\_tid]$
8:     end if
9: end function
```

zeroes and ones in words of a specific *wordsize*, e.g., 32 bit. During insert, the incoming bits are collected in a *buffer*. If the buffer is full, i.e., we inserted as many bits as our word size, the buffer is converted into a *literal word* if it contains zeroes and ones or into a *fill word* if it contains only zeroes or only ones. A literal word stores the actual bits. In contrast, a fill word stores the number of consecutive words that contain only zeroes or ones. Thus, we can compress long runs of zeroes or ones effectively. In case of high base call quality and, thus, having only a small number of mismatching bases, our bitmap contains many zeroes that can be effectively compressed. In order to distinguish literal and fill words, we use the highest-order bit. To distinguish fill words that contain zeroes or ones, we use the second-highest-order bit. Thus, a literal word can store one bit less than the actual word size, e.g., 31 bits for a word size of 32 bit. Moreover, the longest run of zeroes or ones that a fill word can represent is 2^{word_size-2}, e.g., $2^{30} = 1,073,741,824$ for a 32-bit word representing 33,285,996,544 bases.

The worst case for a WAHBitmap regarding storage consumption is that all words are literal words. This means that we have to store a 1 every 31 values assuming 32-bit words. Then, we would waste one bit per word, since we use the highest-order-bit to signal whether the word is a literal or fill word, and require more storage than a plain bitmap. Fortunately, we can assume that the number of mismatching bases within mapped reads that determine most of the ones within the bitmap occur less frequently (cf. Section 3.1.1.2), e.g, 1 in 1,000 bases is wrong. The human data set that we used in our initial evaluation (cf. Section 5.2), contains 53,149,605 mismatching bases, which is roughly 0.4% of all bases or 4 in 1,000 bases. Thus, we assume that the worst case for a WAHBitmap considering our use case is that every mismatching base is stored in a single literal word and literal and fill words alternate, i.e., we cannot store the maximum run length in a fill word. Then, the worst case size of a WAHBitmap in bits depends on the number of mismatching bases (#*mb*) as follows:

$$\#wahbitmap\_size = \underbrace{\#mb * word\_size}_{literal\ word\ size} + \underbrace{(\#mb + 1) * word\_size}_{fill\ word\ size} + \underbrace{word\_size}_{buffer\ size}$$

Algorithm 5.2 Lookup of values stored in a WAHBitmap

Input: The *tid* to look up in the WAHBitmap.
Output: The value of the *tid* encoded by the bitmap: 0 or 1.

```
1: function getValue(tid)
2:     $word\_idx \leftarrow 0$
3:     $word \leftarrow words[word\_idx]$
4:     $max\_tid \leftarrow$ getNumberOfBases($word$)
5:     while $max\_tid \leq tid$ do                    ▷ Is $tid$ encoded in this word?
6:         $word\_idx \leftarrow word\_idx + 1$
7:         $word \leftarrow words[word\_idx]$
8:         $max\_tid \leftarrow max\_tid +$ getNumberOfBases($word$)
9:     end while
10:    return extractValue($word, tid$)
11: end function
```

In our computation, we assume that the overall number of bases divided by the number of fill words required in the worst case does not exceed the maximum run length that a fill word can store. In the worst case, assuming a word size of 32 bit, we require 426 MB to store the bitmap for our evaluation data set. Every fill word would have to encode 231 zeroes, which is far less than the possible 33,285,996,544 zeroes. Thus, using a WAHBitmap is a storage saving alternative for our implementation of a reference-based compression within a column store.

5.3.1.3 Speed up the data access

The two critical bitmap operations in our reference-based compression are the array access and the PREFIX_SUM() computation (cf. line 2 and 3 in Algorithm 5.1 respectively). While the array access using a plain bitmap can be done via a single modulo operation to determine the word containing the value of a given index, computing the prefix sum requires to sum up all ones in the bitmap up to the given index. This introduces notable effort for looking up exception values. A WAHBitmap reduces the effort for computing the prefix sum, since we can skip multiple tuples at once due to the fill words. In case that we find a fill word representing the value 0, we skip it without further action. In the case that we find a fill word representing the value 1, we can add the number of encoded tuples within the fill word to our prefix sum[6].

Fast random data access. Nevertheless, using a WAHBitmap, we inherently apply a run-length encoding to the bitmap. The random access performance of run-length encoded data structures is bad, since we always have to sum up the length values to find the corresponding word for a given index. Assuming that we store literal and fill words in an array *words* and that we can use a function GETNUMBEROFBASES() to return the encoded number of bases within a given literal or fill word, we can describe the look up of the value of a specific tuple id *tid* as shown in Algorithm 5.2. We sum up

[6]We have to subtract a possible offset if the index of interest is encoded within the fill word.

Algorithm 5.3 Lookup of values stored in a WAHBitmap via binary search

Input: The *tid* to look up in the WAHBitmap.
Output: The value of the *tid* encoded by the bitmap: 0 or 1.

```
1: function GETVALUE(tid)
2:     $word\_idx \leftarrow$ BINARYSEARCH($tids, tid$)        ▷ Binary search optimization
3:     $word \leftarrow words[word\_idx]$
4:     return EXTRACTVALUE($word, tid$)
5: end function
```

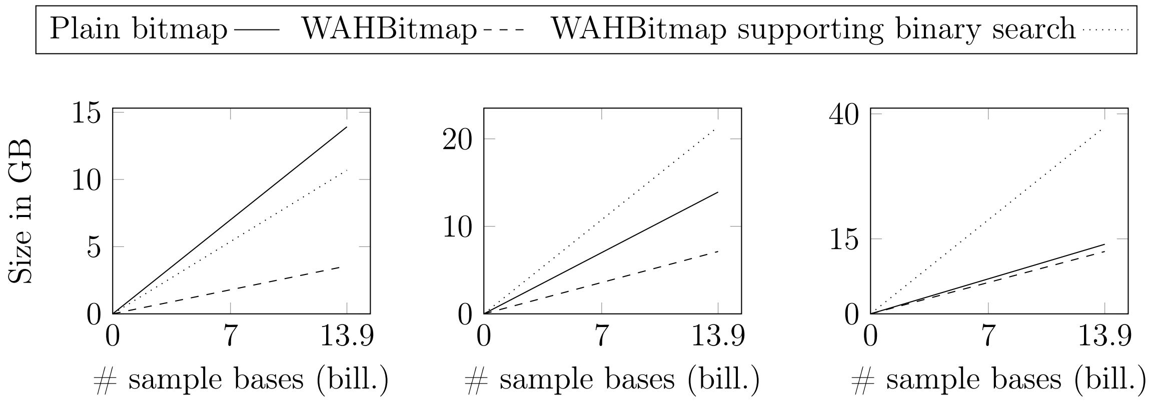
the number of bases encoded by the words (lines 2 - 9), until the sum is greater than *tid* (line 5). Then, we know that the word encodes the value of the given *tid* (line 10), since *max_id* always corresponds to the first tuple id that is not encoded within the word, because we use zero-based tuple ids, i.e., the first tuple has id 0.

Considering the algorithm, we observe that the while loop (line 5) takes longer with increasing tuple id *tid*. Thus, the access runtime highly depends on the chosen index to look up, which makes access times unpredictable. In contrast, using a plain bitmap, we can use a single modulo computation to locate the correct word instead of the while loop. Using our SNV calling query (cf. Listing 3.8), users can analyze random genome regions. For that reason, the random access performance of the WAHBitmap is an issue and might limit the applicability, since it can deteriorate the runtime performance. To improve the random access performance, we extend the WAHBitmap to store the first tuple id *max_tid* that is not encoded within a word in a separate array *tids*. This idea was also suggested by Abadi et al. for run-length compressed columns [2]. Then, we can perform a binary search on the *tids* array to determine the index of the word that contains the value of the *tid*. The binary search method returns the index of the first value that is greater than the given value *tid*. We show the binary search modification in Algorithm 5.3. Instead of the while loop, we perform a binary search on array *tids*.

Of course, the use of an additional array *tids* increases the storage requirements. We have to store one additional tuple id per literal and fill word. Moreover, we have to recognize the data type size of the tuple ids used within the system. The overall WAHBitmap size in bits for our worst case scenario is computed as follows:

$$\#wahbitmap\_size = \underbrace{\#mb * word\_size}_{literal\ word\ size} + \underbrace{(\#mb + 1) * word\_size}_{fill\ word\ size} + \underbrace{word\_size}_{buffer\ size}$$
$$+ \underbrace{(2 * \#mb + 1) * tuple\_id\_size}_{ids\ array\ size}$$

Considering our evaluation data set, we have to use 64-bit tuple ids, since we have to store 13,917,235,121 tuples in table Sample_Base, which exceeds the possible maximum tuple id of 4,294,967,295 using 32-bit tuple ids. Thus, we have to store additional 851 MB to keep the *tids* array, which results in an overall storage consumption for the WAHBitmap of 1.3 GB. In sum, we increase the storage size of the reference-base



(a) 4 mismatches ... (b) 8 mismatches ... (c) 14 mismatches ...

Figure 5.6: in 1000 bases. With increasing number of mismatching bases, the WAHBitmap has to store more words in worst case reducing the compression effect. The binary search optimization within WAHBitmaps sacrifices most of the storage savings.

compressed column by a factor of 3, but we still require less storage than the plain bitmap. However, the storage size for the WAHBitmap in our worst case scenario highly depends on the number of mismatching bases. In Figure 5.6, we show the theoretical storage sizes of the WAHBitmap assuming different mismatch rates. With increasing number of mismatching bases, the compression effect of the WAHBitmap is reduced, since we have to store more words. In combination with the binary search optimization, we even increase the storage requirements compared to a plain bitmap, since we have to store a 64-bit tuple id per word. Thus, we should use this feature with care and only when the use case really requires it. We investigate the impact of the binary search optimization on our SNV calling query setup from Chapter 4 at the end of this section (cf. Figure 5.7) to determine whether the storage overhead pays out.

Fast sequential data access. The binary search solution solves the single random access performance bottleneck, but within our analysis use case, we also often access successive tuples. Considering our SNV calling query, we have to access multiple sample bases that are mapped within a specific genome region. Since the sample bases of a read are imported in sequence (cf. Section 3.3.1) and map to consecutive reference bases, it is likely that we have to access successive tuples within column SB_BASE_VALUE. If we always require a binary search for every access, the runtime increases. Fortunately, our processing pipeline using the invisible join ensures that we perform sequential accesses on column SB_BASE_VALUE, because we create the join results via positional lookups leveraging the foreign keys of table Sample_Base. Thus, we can assume that subsequent accesses to our compressed column SB_BASE_VALUE are likely in sequential order. To speed up sequential access patterns, we integrate a caching mechanism that stores the index of the last visited word and allows us to start the search for the correct word from that index on. We show the idea in Algorithm 5.4. Assuming that *last_word_idx* (line 2) stores the index of the last word that we visited and *last_tid* (line 3) contains the tuple id that we looked up at last, we can look up the value for a

Algorithm 5.4 Cached lookup of values stored in a WAHBitmap

```
Input: The tid to look up in the WAHBitmap.
Output: The value of the tid encoded by the bitmap: 0 or 1.
 1: function getValue(tid)
 2:     $last\_word\_idx \leftarrow (last\ word\_idx\ or\ 0)$                ▷ Caching optimization
 3:     $last\_tid \leftarrow (last\ tid\ or\ 0)$                          ▷ Caching optimization
 4:     $word\_idx \leftarrow last\_word\_idx$                              ▷ Caching optimization
 5:     if $tid < last\_tid$ then        ▷ Caching optimization: Non-sequential access?
 6:         $word\_idx \leftarrow$ binarySearch($tids, tid$)         ▷ Binary search optimization
 7:     end if
 8:     $max\_tid \leftarrow tids[word\_idx]$           ▷ Binary search optimization: tids array
 9:     while $max\_tid \leq tid$ do    ▷ Caching optimization: Sequential scan on words
10:         $word\_idx \leftarrow word\_idx + 1$
11:         $max\_tid \leftarrow tids[word\_idx]$
12:     end while
13:     $last\_word\_idx \leftarrow word\_idx$                              ▷ Caching optimization
14:     $last\_tid \leftarrow tid$                                          ▷ Caching optimization
15:     $word \leftarrow words[word\_idx]$
16:     return extractValue($word, tid$)
17: end function
```

specific tuple id *tid* by reusing the previously computed word index (line 4) or moving it forward until we found the containing word (line 9 - 12). Note, that we combine the caching optimization with the binary search optimization, which saves us to cache the last *max_tid* value (line 8). Moreover, we can perform a binary search (line 6) if the next looked up *tid* is smaller than the previous one (non-sequential access, line 5).

Our caching optimization requires only to store two additional cache values, but allows us to perform fast sequential access on reference-based-compressed data, since we do not have to start always at index 0 to determine the word of a requested tuple id *tid*. Note, that we can also use the idea of our caching technique to speed up the computation of prefix sums to enable lookups in the exception list (cf. Algorithm 5.1 line 3). The idea is to cache the last computed prefix sum and to reuse it if the access is sequential.

Runtime evaluation

In the following, we investigate the impact of the caching and binary search optimization for tuple id access on the SNV calling runtime using DBS_{base}. We use the same experimental setup as in Section 4.2.2. We show the results in Figure 5.7. We execute the query using the invisible join and array-based aggregation techniques. In contrast to the initial setup, we now enable all compression schemes. Moreover, we compare the results on compressed data with the results on uncompressed data taken from Section 4.4.2.

We observe that without any optimization, the runtime increases drastically making SNV calling on compressed data no option. Using either the caching or binary search

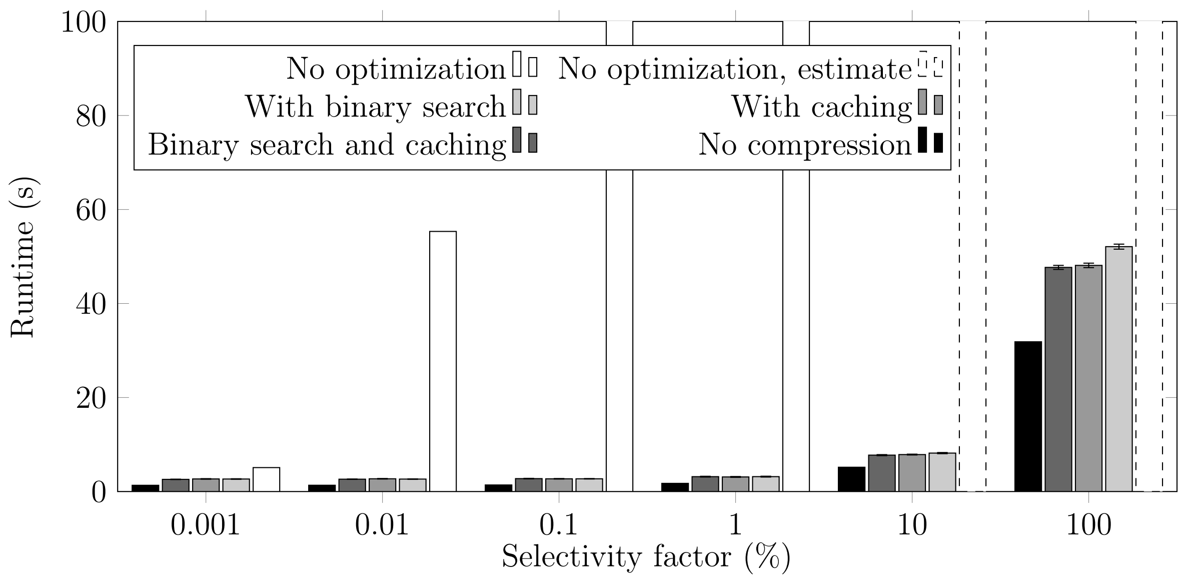
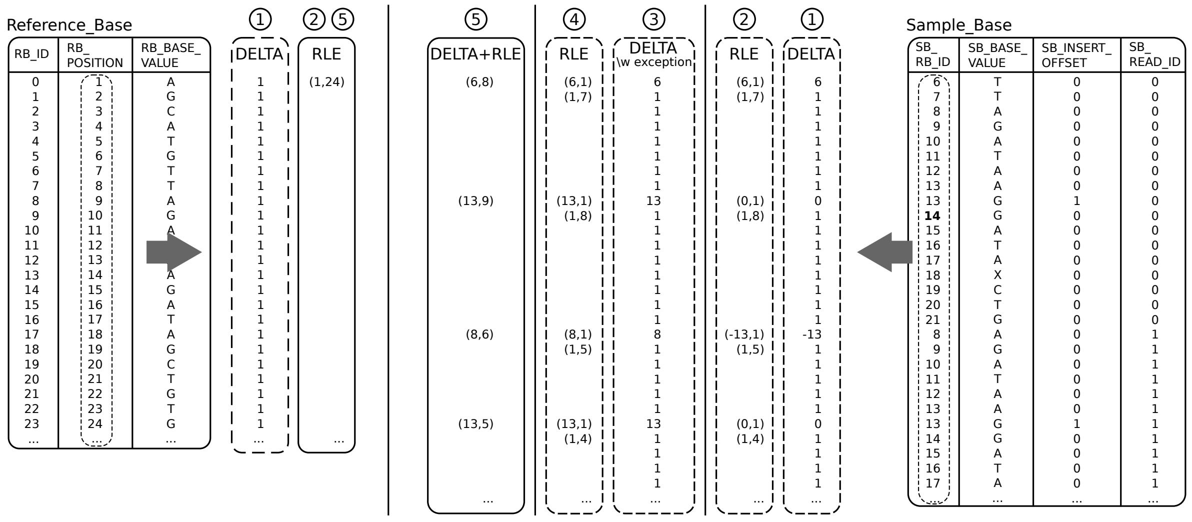


Figure 5.7: SNV calling runtimes on a human chromosome 1 using genome-specific compression schemes with different access optimization strategies. Without any optimization strategy, compression increases the overall runtime drastically. Using either binary search or caching removes the access bottleneck. Nevertheless, compression introduces overhead especially on larger genome regions.

optimization, we are able to reduce the analysis runtime on compressed data significantly. We can reduce the runtime overhead to a factor of 1.5 compared to analyzing uncompressed data. Moreover, we find that caching alone or in combination with binary search provides the shortest runtime considering all selectivity factors. While on small genome regions, the effect is negligible, it becomes significant on larger genome regions. The reason for this is that we always have to perform a binary search that has an average complexity of $\mathcal{O}(log(n))$, while sequential accesses have an access complexity of $\mathcal{O}(1)$. Since our query often requires sequential accesses, we can benefit from the reduced complexity. Consequently, on large genome regions, the difference has to become significant. If we have a random access scenario such as selecting single sample bases or analyzing genome regions randomly, the binary search optimization may become more beneficial to locate the (random) base or start of the region fast. Thus, throughout this thesis, we will report storage consumptions always with enabled caching and binary search optimization.

5.3.2 Delta+RLE encoding

In the previous section, we showed how to integrate reference-based compression into a column store based on the base-centric database schema. However, this storage optimization will not outweigh the introduced overhead due to the explicit storage of position (RB_POSITION) and mapping information (SB_RB_ID) that are responsible for 80% of the storage consumption within a base-centric database (cf. Figure 5.3b).



Reference_Base

RB_ID	RB_POSITION	RB_BASE_VALUE
0	1	A
1	2	G
2	3	C
3	4	A
4	5	T
5	6	G
6	7	T
7	8	T
8	9	A
9	10	G
10	11	A
11	12	
12	13	
13	14	A
14	15	G
15	16	A
16	17	T
17	18	A
18	19	G
19	20	C
20	21	T
21	22	G
22	23	T
23	24	G
...	...	...

Sample_Base

SB_RB_ID	SB_BASE_VALUE	SB_INSERT_OFFSET	SB_READ_ID
6	T	0	0
7	T	0	0
8	A	0	0
9	G	0	0
10	A	0	0
11	T	0	0
12	A	0	0
13	A	0	0
13	G	1	0
14	G	0	0
15	A	0	0
16	T	0	0
17	A	0	0
18	X	0	0
19	C	0	0
20	T	0	0
21	G	0	0
8	A	0	1
9	G	0	1
10	A	0	1
11	T	0	1
12	A	0	1
13	A	0	1
13	G	1	1
14	G	0	1
15	A	0	1
16	T	0	1
17	A	0	1
...	...	...	...

Figure 5.8: Delta+RLE encoding represents runs of continuous values that are incremented by one as run value and length value similar to RLE encoding ⑤. This saves storage and effectively compresses explicit position and mapping information.

Column RB_POSITION stores the position of every reference base within the reference sequence, e.g., a chromosome. Since we import the single bases of a reference in order of appearance within the sequence string (cf. Section 3.3.1), the column contains continuous sequences of numbers incremented by one starting with one[7] per reference sequence (CONTIG). On the left side of Figure 5.8, we show an excerpt of table Reference_Base storing the first 24 reference bases within of a reference sequence.

The data within column SB_RB_ID looks similar than in column RB_POSITION. Column SB_RB_ID contains a foreign key indicating the reference base that every single base of a read is mapped to (cf. Section 3.3.1). Since reference bases are imported in order of appearance in the reference sequence string and we assign primary keys (RB_ID) that are equivalent to tuple ids, the values in column RB_ID have the same characteristic as the RB_POSITION values. Since the bases of a read are usually mapped to successive reference bases (except inserted bases), column SB_RB_ID also contains continuous sequences of numbers incremented by one starting at a specific offset. On the right side of Figure 5.8, we show an excerpt of table Sample_Base storing two different reads mapped to the same reference sequence.

5.3.2.1 Compression concept

To compress such values, we could think of storing the deltas between successive values (DELTA encoding). The assumption is that the deltas have a smaller value domain requiring less bits to store each value. Applying this idea to the values of the columns RB_POSITION and SB_RB_ID, we end up with long sequences of ones, since the

[7]We assume one-based position information. It is also possible to use zero-based positions.

values are usually incremented by one (cf. ① in Figure 5.8). This allows us to additionally apply RLE to compress the sequences (runs) of ones ②. Unfortunately, if we want to access the value of a specific tuple id, we have to decompress and sum up all delta values before and including the respective tuple id leading to increased access times due to decompression overhead.

To improve the runtime, we could avoid DELTA encoding for values that have a delta that is different than one. Such values are stored as plain numbers ③ and, in case of column SB_RB_ID, they appear regularly. Then, we apply RLE ④. In order to access the value of a specific tuple id, we would have to decompress the run for the specific tuple id, extract its offset and add this to the first plain value before the run. Additionally, we can merge the plain value and a run of ones changing the semantics of RLE from encoding runs of similar values to runs of values incremented by a specific delta, e.g., one. We call this approach Delta+RLE encoding ⑤. In case of column RB_POSITION, a DELTA and RLE encoding directly leads to the desired Delta+RLE encoding, since the first value within a run in RB_POSITION is always 1.

Applying predicates. We can directly operate on the Delta+RLE compressed data similar to a RLE compressed column. An important operation that we should support directly on compressed data is to evaluate between predicates, since column SB_RB_ID is involved in the between-predicate rewriting to speed up the invisible join technique (cf. Section 4.3.1). We can leverage the Delta+RLE compressed column to evaluate a between predicate such as `SB_RB_ID BETWEEN min AND max` by excluding runs that will not contain values in the requested range. To this end, we check whether the value and length of a run having index *idx* fulfills following condition:

$$values[idx]\ <=\ max\ AND\ (values[idx]\ +\ length[idx]\ +\ 1)\ >=\ min$$

If this condition does not hold, either the run value is greater than *max* indicating that the run lies right to the range or the end value of the run is smaller than *min* indicating that the run lies left to the range. Consequently, we can exclude the run. Otherwise, we know that at least some tuple ids encoded in the run fulfill the predicate and we have to retrieve them.

Data retrieval. To retrieve a data value by tuple id *tid*, we use the algorithm given in Algorithm 5.5. We have to locate the run that encodes the given tuple id. To this end, we have to sum up all length values stored in array *length* until the sum exceeds the index that we want to look up (line 5 - 8). If we found the run, we have to retrieve the *value* by adding the *position_in_run* of the respective *tid* as offset. To determine the *position_in_run*, we have to subtract the sum of the length values of the previous run from the *tid* (line 12). For example, if we want to retrieve the value of the 10th tuple of table Sample_Base having index 9 (cf. bold printed SB_RB_ID value of table Sample_Base in Figure 5.8), we find the value of this tuple in the second run (13, 9), because the run length of the first run is 8 that is less or equal to the requested index 9. To compute the position within the run, we have to subtract the sum of run lengths of

Algorithm 5.5 Look up of single tuples in a Delta+RLE compressed column

Input: The *tid* to look up in the Delta+RLE column.

Output: The value of the *tid*.

```
 1: function getValue(tid)
 2:     $run\_idx \leftarrow 0$
 3:     $position\_in\_run \leftarrow 0$
 4:     $run\_length \leftarrow length[run\_idx]$
 5:     while $run\_length \leq tid$ do    ▷ Determine the run containing the value of tid
 6:         $run\_idx \leftarrow run\_idx + 1$
 7:         $run\_length \leftarrow run\_length + length[run\_idx]$
 8:     end while
 9:     if $(run\_idx = 0)$ then    ▷ Determine position of tid in the run
10:         $position\_in\_run \leftarrow tid$
11:     else
12:         position_in_run $\leftarrow tid - (run\_length - length[run\_idx])$
13:     end if
14:     return $values[run\_idx]$ + position_in_run
15: end function
```

previous runs, which is 8, since we have only one run before. The result is 1 indicating that the requested tuple is the second value in the run. Since we know that values in a run are continuous and incremented by one, we can use the position in the run as offset to compute the actual value, which is 13 + 1 = 14.

Storage consumption. Our example of Delta+RLE in Figure 5.8 reveals that the worst case for Delta+RLE encoding to compress mapped reads depends on the number of reads (*#reads*) to store and the number of inserted bases (*#ins*). In worst case, we create one run for every read. Furthermore, every inserted base within a read disturbs the continuity of the SB_RB_ID values and requires to create a new run. We can estimate the worst case size in bits of a Delta+RLE column as follows:

$$\#deltarle\_size = \underbrace{(\#reads + \#ins)}_{\#\ of\ runs} * \underbrace{(tuple\_id\_size + length\_value\_size)}_{run\ size\ in\ bits}$$

The *tuple_id_size* indicates the number of bits used for storing tuple ids in the database system, e.g., 32-bit or 64-bit tuple ids. The *length_value_size* indicates the number of bits used to represent run lengths, e.g., a 32-bit integer allows to encode a maximum run length of 4,294,967,295. The less inserted bases, the better is the compression ratio. Moreover, longer reads improve the storage requirements per sample base, since this increases the run length.

5.3.2.2 Improving access times

Since Delta+RLE encoding is based on run-length encoding, it also provides bad random and sequential access performance for the same reasons as the WAHBitmap (cf.

Section 5.3.1.3). We can apply the same optimizations, i.e., binary search to accelerate random access and caching to speed up sequential data access, as discussed for the WAHBitmap. Since the access pattern during our SNV calling query does not differ from column SB_BASE_VALUE, we assume similar effects of both techniques on the runtime. Nevertheless, the additional storage requirements for the binary search optimization differ. We would reuse the array to store *length* values in a Delta+RLE compressed column for storing tuple ids. To this end, we might have to increase the word size of length values. For example, if we use 32-bit length values, but require 64-bit tuple ids, we have to switch the word size of length values to 64-bit. Overall, this is an increase by a factor of 1.3 and, thus, much less than an increase of a factor of 3 within the WAHBitmap. Thus, the decision whether to use the binary search optimization is less critical than for the WAHBitmap. Throughout this thesis, we will report storage consumptions always with enabled caching and binary search optimization.

5.3.3 Lossy compression

Usually, we do not want to loose any kind of information that we store in database systems. Nevertheless, in certain domains lossy compression is applicable, if the loss of information does not lead to further errors, the errors are small enough to neglect them or the data is error-prone anyway. Within genome data, base call quality values are a good target for lossy compression [56]. Base call quality values are log-transformed [42] and encoded in the SAM format as ASCII characters (cf. Section 3.1.1.2). A base having a quality value of 30 has a probability of 1 in 1000 to be incorrect. Encoding base call quality values as char with 8 bits allows for encoding 128 different values. Nevertheless, base call qualities of 40 and above are reasonable good [61]. To reduce storage size, bit-packing can be applied. Certainly, encoding 40 values still requires 6 bits. To reduce the storage size further, lossy compression schemes where proposed. One idea is to bin quality values. A widely used binning scheme is proposed by Illumina, a sequencing machine vendor, using 8 bins requiring 3 bit to encode [61]. Then, the bins resemble a kind of quality classes instead of real values. We can integrate this approach as column compression scheme on top of the base-centric database schema.

5.3.4 Storage evaluation

In this section, we investigate the concrete impact of our proposed compression schemes on the storage consumption of DBS_{base} for storing the complete human genome data set used in our initial storage evaluation (cf. Section 5.2). In Figure 5.9, we show the storage savings per compression scheme in comparison to DBS_{seq} and DBS_{base} applying only standard compression schemes.

5.3.4.1 Results

Using only standard lightweight compression, DBS_{base} is no option to store larger data sets, since it increases the data volume dramatically due to explicitly storing position and mapping information. Using Delta+RLE, we can mitigate this disadvantage effectively allowing us to consider the base-centric database schema as alternative primary

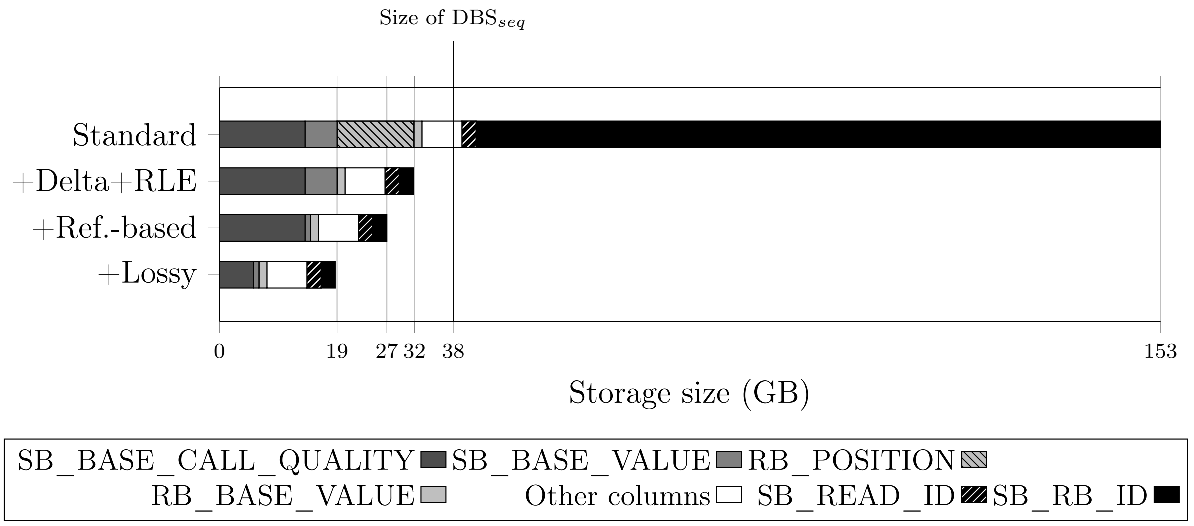


Figure 5.9: Impact of genome-specific compression on the storage consumption of single columns in DBS$_{base}$. Delta+RLE is needed to benefit from storage savings due to reference-based and lossy compression.

database layout for mapped read data. Applying reference-based and lossy compression, allows us to reduce the storage consumption further requiring only half the storage than DBS$_{seq}$.

5.3.4.2 Discussion

The single compression schemes, discussed in this chapter, contribute differently to the storage reduction. Overall, Delta+RLE compression is an important strategy to reduce the overhead of the base-centric database schema and to provide reasonable storage consumption. In combination with BITDICT compression of base values, we already have an easy-to-integrate compression approach for genome data that requires less storage than DBS$_{seq}$. The reference-based compression and lossy compression scheme effectively reduce the actual payload data further. Since we integrated both compression schemes as column compression schemes, they integrate well with our processing engine. Thus, we can reduce the storage consumption effectively, while keeping the negative impact on analysis runtime due to the need for processing compressed data reasonable low (cf. Figure 5.7).

Since our compression schemes are designed for the base-centric database schema, we cannot reduce the storage consumption of DBS$_{seq}$. However, the results show the possible storage reduction of DBS$_{seq}$ using specialized string compression schemes. Nevertheless, the use of string-based compression schemes would require to use specialized analysis UDFs that are able to process the compressed strings. Otherwise, we would have to decompress the strings before we can apply our base-centric processing approach.

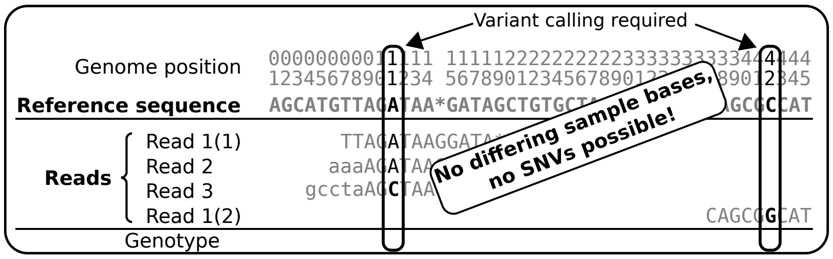


Figure 5.10: Base pruning filters out genome positions where no mismatching base has been mapped, which reduces the number of genome positions that have to be processed during SNV calling.

5.4 Base pruning: Leveraging genome characteristics for query processing

In this section, we introduce a technique called base pruning to effectively reduce the processing effort during SNV calling. Considering the general functionality of SNV callers, we find that only those genome positions will show a differing genotype where at least one mismatching base has been mapped to a reference base. Otherwise, the result of a SNV caller has to be equal to the reference base resulting in a matching genotype. Consequently, if we can detect which genome positions have no mismatching bases in advance, we can exclude these genome positions from further processing reducing the effort for SNV calling, because we do not have to join and aggregate the data. We depict the idea in Figure 5.10. Only genome position 11 and 42 show at least one differing mapped sample base (black boxes). All other positions are either insertions, deletions or show only matching sample bases. Since we encapsulate the domain-specific knowledge about computing genotypes in a UDA (cf. Listing 3.8), a traditional database optimizer cannot apply this optimization without modification. Thus, we have to make this optimization strategy explicitly available.

5.4.1 Approaches

To apply the idea of base pruning, we integrate a *base pruning* filtering step that returns those sample bases that map to a genome position that has at least one differing sample base mapped to it. To integrate such a filtering step, we have to be able to compare sample bases and reference bases. Using the base-centric database schema, we explicitly encode the mapping between sample and reference bases. We can leverage this relationship to perform the comparison efficiently. In the following section, we describe two possible strategies to determine the genome positions and sample bases that must be analyzed.

Straightforward approach. A straightforward approach uses two scans on table Sample_Base to determine those sample bases that must be analyzed. In a first scan, we compare each sample base (in a given range) with the corresponding reference base.

We collect all genome positions that have at least one differing sample base. Then, in a second scan, we check each sample base whether it is mapped to one of the genome positions that has at least one differing sample base mapped to it. Consequently, we require two sequential table scans, which can be implemented efficiently. Nevertheless, during the first scan, we permanently look up reference base values from column RB_BASE_VALUE of table Reference_Base. Within a single read, sample bases are usually mapped to consecutive positions allowing for cache-efficient access to column RB_BASE_VALUE. Unfortunately, every new read that we process, might result in a random lookup within table RB_BASE_VALUE, since consecutive reads do not have to map to consecutive genome regions. Sorting reads by mapping position before importing or converting them will improve the access, but is not a requirement.

Indexed approach. However, we can speed up the *base pruning* filtering step by leveraging the reference-based compression of column SB_BASE_VALUE. The compression already encodes which sample base is different from its reference base. Thus, we can use this information stored in a bitmap to efficiently determine those sample bases that mismatch. From these sample bases, we can retrieve the genome positions that have at least one differing base mapped to it. Using a WAHBitmap, we can avoid a complete column scan, since we can effectively skip those sample bases that are encoded in a fill word representing a 0. Moreover, we avoid the comparison with reference base values from column RB_BASE_VALUE, since the comparison is already done during compression and encoded in the bitmap.

Runtime evaluation

In this section, we evaluate the impact of base pruning on the SNV calling runtime using DBS$_{base}$ and DBS$_{seq}$ considering differently sized genome regions using the same experimental setup as in Section 4.2.2. We perform the query using the invisible join and array-based aggregation techniques. In addition to the initial setup, we enable all compression schemes. DBS$_{base}$ operates directly on compressed data and DBS$_{seq}$ compresses data at runtime. To evaluate the impact of reference-based compression during base pruning on the overall runtime, we also execute the workload without reference-based compression. We show the results in Figure 5.11.

The results show that base pruning has most impact on large genome regions (selectivity factor $>= 10\%$). On smaller genome regions, we cannot detect a significant difference in the runtime. As expected DBS$_{seq}$ outperforms DBS$_{base}$ on small genome regions. This is expected and in concordance to our observations from Chapter 4. Furthermore, leveraging the reference-based compressed data as index during the *base pruning* computation is also most beneficial on large genome regions. In these cases, we would have to perform two complete table scans on the same column. Leveraging reference-based compression, we can avoid one complete column scan. Overall the impact of base pruning on DBS$_{seq}$ is less than its impact on DBS$_{base}$, because we cannot benefit from base pruning during the conversion of data, while we benefit during all processing steps using DBS$_{base}$. Moreover, DBS$_{seq}$ has increased effort during conversion to apply the

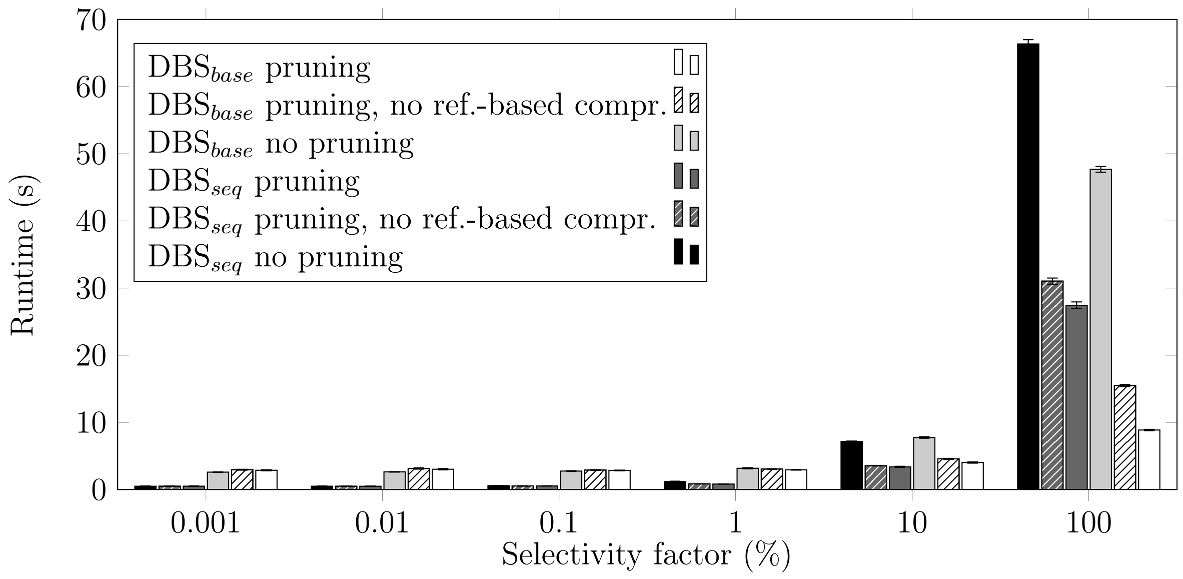


Figure 5.11: SNV calling runtime using DBS$_{base}$ and DBS$_{seq}$ with and without base pruning. Base pruning reduces the analysis runtime especially on large genome regions. On small genome regions, the additional effort for the base pruning computation sacrifices the potential runtime decrease. Additionally leveraging reference-based compression for the base pruning computation speeds up the analysis of large genome regions further, since we avoid one complete table scan and the comparison with the single reference bases for equality.

reference-based compression. We conclude that base pruning is a necessary optimization to provide fast execution times on large genome regions. Although, base pruning requires additional effort, the impact on small genome regions is negligible.

5.4.2 Applicability to specialized analysis tools

We also considered *base pruning* in the context of specialized analysis tools and integrated it into SAMTOOLS [77]. Our final evaluation will show that base pruning also improves the runtime of specialized analysis tools (cf. Section 5.5), but is limited due to conceptual differences in the storage and processing of genome data. In the following, we explain these limitations in detail.

Applicability to aggregation phase only

Specialized analysis tools operate on heavyweight compressed genome data residing on disk. To guarantee reasonable performance, the tools require mapped reads to be sorted by mapping position (cf. Section 3.1.2). Thus, the tools can stream compressed data in blocks from disk and are sure that the data that they decompress belongs to the current genome region to be processed. The tools are highly specialized and directly convert the uncompressed data into a ready-to-aggregate output called base pileup. Comparing the process with DBS$_{seq}$, we converted and already computed the join result. Thus, it becomes clear that we cannot reduce the effort during the *join*, i.e., the decompression and conversion, within the specialized analysis tool. We can only check per genome position which of the base pileups contains at least one differing base and must be processed further.

Reference-based compression cannot be exploited

A direct consequence of operating on heavyweight compressed data is that we cannot leverage the reference-based compression to speed up the base pruning computation. As soon as we start to decompress the data, we already lost the advantage to exploit and process compressed data.

5.5 Putting it all together

So far, we concentrated our runtime analyses on a smaller data set comprising only human chromosome 1 data to reduce side effects due to limited main memory. Moreover, we wanted to examine the impact of specific database operators (Chapter 4) and processing optimizations such as base pruning in detail. Now that we devised a powerful processing pipeline and effective compression schemes, we consider three real world data sets having different data characteristics. We use the same system setup as described in Section 4.2.2.1. Our goal is two-fold: First, we want to confirm our results regarding storage consumption and runtime performance from the small evaluation data set. To this end, we apply our database approaches to larger data sets. Moreover, we want to investigate the impact of different genome data characteristics on analysis runtime and storage consumption. To this end, we will start with a description of the characteristics of the data sets that we use throughout this final evaluation. Then, we discuss the results of our experiments.

5.5.1 Data set characteristics

We use three real world data sets for our experiments. *DataSet 1* and *2* contain human genome data that we obtained from the 1000 genomes project[8], which provides representative real world data sets [1]. The third data set contains barley genome data that we obtained from the plant research institute IPK Gatersleben.

All three data sets differ in following four characteristics: *number of reference bases*, *coverage*, *read length* and *mismatch rate*. In Table 5.2, we summarize the different data characteristics.

Number of reference bases. All three data sets contain a different number of reference bases that indicates the size of the reference genome. Thus, the number of reference bases is an upper bound for the possible genome positions that have to be

[8]data is available at ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00096/

Organism	*Homo sapiens* human		*Hordeum vulgare* barley
DataSet	*1*	*2*	*3*
# Mapped Bases	13.9B	11.8B	3.9B
# Reference Bases	3.1B	249M	1.9B
∅ Coverage	4	47	2
∅ Read Length	100	250	100
Mismatch Rate %	0.38	0.8	1.4

Table 5.2: Genome data sets have different key characteristics that impact storage consumption and processing performance of the respective data set.

analyzed. *DataSet 2* contains a very small number of reference bases compared to the other data sets.

Coverage. Every genome is read multiple times during DNA sequencing to cope with reading errors (cf. Section 2.2.1.1). The coverage indicates how many times a single genome position was read during DNA sequencing. Since the sequencing process cannot be controlled in that detail, the coverage is usually an average over all genome positions. Thus, the product of coverage and the number of reference bases corresponds with the overall number of sample bases within a data set. The higher the coverage is, the more bases must be processed per genome position during SNV calling. To evaluate the impact of coverage and differing number of reference bases, while fixing the number of sample bases, *DataSet 2* contains only the mapped reads of human chromosome 1.

Read length. Different DNA sequencing techniques generate reads of different size. Storing longer reads should be beneficial for RLE and our Delta+RLE compression that are used to compress columns SB_READ_ID and SB_RB_ID. The length of the runs within both columns directly corresponds to the read length. Thus, we expect to achieve better compression when storing longer reads. Moreover, we have to store less read data per sample base for the same amount of sample bases.

Mismatch rate. The mismatch rate describes how many sample bases are different from their corresponding reference base. Reference-based compression and base pruning are directly influenced by the mismatch rate. The more mismatches occur, the more exception values have to be stored. Moreover, the chance increases that a specific genome position must be analyzed.

5.5.2 Storage assessment

In the first experiment, we compare the storage consumption of our database approaches DBS$_{seq}$ and DBS$_{base}$ with the state-of-the-art flat-file formats BAM and CRAM. In this experiment, we include necessary storage overhead due to the binary search optimization to speed up random accesses on compressed data (cf. Section 5.3.1.3). Moreover, we do not use lossy compression.

Approach	Storage size in GB (Relative to DBS$_{seq}$)			Main compression type
	DataSet 1	*DataSet 2*	*DataSet 3*	
DBS$_{seq}$ (baseline)	38.0 (100%)	27.2 (100%)	12.9 (100%)	lightweight
DBS$_{base}$	27.2 (72%)	17.8 (65%)	9.5 (74%)	
BAM	14.6 (38%)	6.9 (25%)	3.9 (30%)	heavyweight
CRAM	10.3 (27%)	4.9 (18%)	3.2 (25%)	
Zipped DBS$_{base}$	11.7 (31%)	6.1 (22%)	3.3 (26%)	

Table 5.3: Storage consumption of three real-world data sets using DBS$_{seq}$, DBS$_{base}$, BAM and CRAM. Due to our genome-specific compression schemes available in DBS$_{base}$, we can outperform DBS$_{seq}$ on all data sets. BAM and CRAM additionally apply heavyweight compression making them superior to our database approaches if we avoid to use heavyweight compression.

With this experiment, we want to investigate whether we can cope with the large storage increase of the base-centric database on all different data sets. Moreover, we are interested how the data characteristics such as read length and mismatch rate influence the concrete storage savings. In Table 5.3, we report the absolute storage sizes for every approach and in brackets the relative storage size compared to DBS$_{seq}$.

5.5.2.1 Overall storage consumption

Independent of the data set, DBS$_{base}$ always requires less storage than DBS$_{seq}$, since our Delat+RLE compression scheme successfully mitigates the overhead due to the explicit storage of position and mapping information within DBS$_{base}$. Additionally applying the genome-specific compression scheme reference-based compression finally allows us to decrease the storage size by 26% to 35% compared to DBS$_{seq}$.

Nevertheless, since we avoid to use heavyweight compression schemes in DBS$_{base}$, DBS$_{base}$ still requires 2 to 2.5 times more storage than BAM. In contrast, BAM applies heavyweight compression that pays out according to our measurements. To show the efficiency of our lightweight compression schemes and used data structures within DBS$_{base}$, we compressed the disk-resident database files of DBS$_{base}$ used to persist the complete database using gzip. The storage size of these compressed files is competitive compared to BAM and CRAM (cf. last row of Table 5.3). Thus, we also conclude that our reference-based compression implementation including all its access optimizations is competitive to the CRAM approach.

5.5.2.2 Impact of data characteristics

The concrete storage savings of DBS$_{base}$ compared to DBS$_{seq}$ vary between the three data sets. Therefore, we investigate the influence of data set characteristics in more detail. The differences depend on the *read length* and *mismatch rate* of the respective data set. In Table 5.4, we show the storage requirements per row for the three columns SB_RB_ID, SB_Read_ID and SB_BASE_Value considering the three data sets. The storage size of the other columns is independent from data set characteristics.

	DataSet	*1*	*2*	*3*
	# Mapped Bases	13.9B	11.8B	3.9B
	∅ Read Length	100	250	100
	Mismatch Rate %	0.3	0.8	1.4
Bits per row	SB_RB_ID	1.4	0.6	1.4
	SB_READ_ID	1.4	0.6	1.4
	SB_BASE_VALUE	0.6	0.8	1.3
	Sum of bits	3.4	2.0	4.1

Table 5.4: The influence of data characteristics on the storage consumption using DBS$_{base}$. With increasing read length the overhead of the foreign key columns SB_RB_ID and SB_READ_ID decreases per row. Less base mismatches reduce the number exception values to store, which increases the compression ratio of the WAH-Bitmap used to implement reference-based compression.

Impact of read length

We compress column SB_READ_ID with RLE and column SB_RB_ID with our Delta+RLE compression scheme. RLE leverages the redundancy of data values that are stored in runs and Delta+RLE leverages the redundancy of deltas between consecutive data values. Thus, their effectiveness depends on the length of the runs. In both columns, we require 1.4 bits on average per stored sample base considering *DataSet 1* and *3*. In contrast, we only need 0.6 bits on average per stored sample base considering *DataSet 2*. The key difference is the read length between the data sets. Since *DataSet 2* has an average read length of 250 bases, the runs are longer in both columns. *DataSet 1* and *3* have an average read length of 100 bases, which limits the length of the runs significantly. The results of all three data sets also show that the impact of inserted bases on the effectiveness of Delta+RLE encoding (cf. Section 5.3.2.1) is negligible, because the number of bits required on average per stored sample base is equal between columns SB_RB_ID and SB_READ_ID, although column SB_RB_ID requires to store an additional run per inserted base (cf. Section 5.3.2.1). Thus, the number of reads to store dominates the number of runs within our three data sets.

Impact of mismatch rate

We already discussed that the mismatch rate has direct impact on reference-based compression (cf. Figure 5.6). Considering the three data sets, we observe the predicted storage overhead of a WAHBitmap due to the increasing mismatch rate. In case of *DataSet 3*, we measure 1.3 bits on average per stored sample base. Since the size of the exception list is negligible (cf. Section 5.3.1.1), we see that a plain bitmap would perform better, since it only requires 1 bit per stored sample base. *DataSet 1* and *2* have smaller mismatch rates improving the storage requirements compared to a plain bitmap.

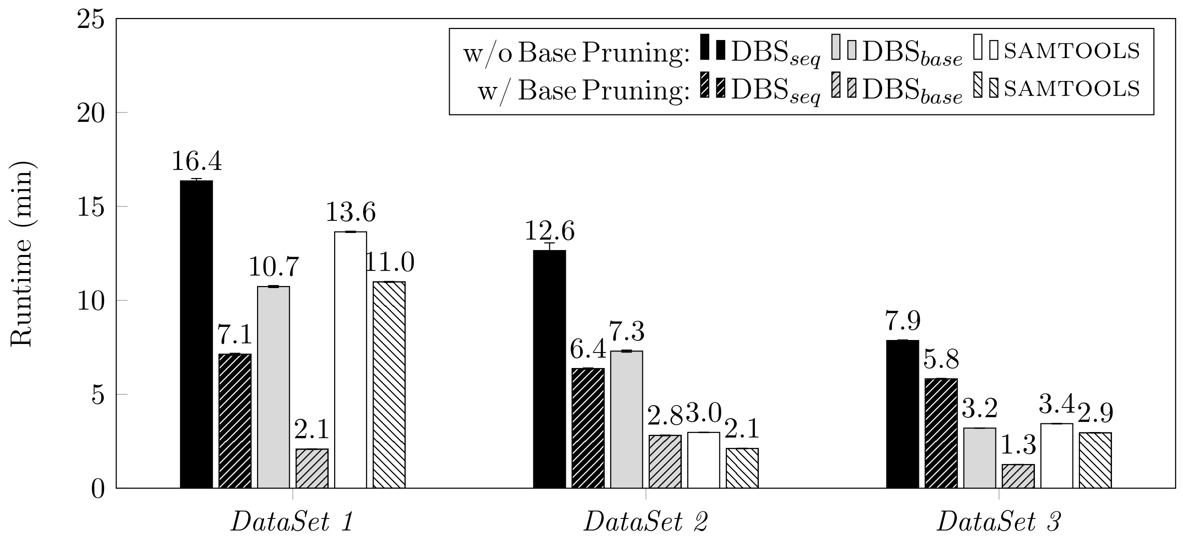


Figure 5.12: SNV calling on three real-world data sets using DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS with and without *base pruning.* Since DBS$_{seq}$ requires to convert data during the analysis, DBS$_{base}$ is always faster. Using base pruning, DBS$_{base}$ can even outperform SAMTOOLS, since SAMTOOLS has to decompress and convert data before making use of base pruning.

Summing up the number of bits over all three columns, gives the number of bits required to store a single base using DBS$_{base}$ ignoring base call quality values[9]. Assuming that we need 8 bit per base value in DBS$_{seq}$, we save most storage on *DataSet 2*, followed by *DataSet 1* and *3*, which corresponds to the overall storage savings reported in Table 5.3.

5.5.3 SNV calling runtime assessment

In the second experiment, we evaluate the SNV calling runtime of DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS 1.3 [77]. As discussed in Section 5.4.2, we also integrate base pruning into SAMTOOLS. In Figure 5.12, we show the runtime results on the three different data sets. We indicate the runtime of approaches using *base pruning* with hatched bars. Within this evaluation, we are interested in the impact of the data characteristic on the analysis runtime. Moreover, we examine the speed up achieved by *base pruning.* In this experiment, we report the runtime to process the complete genome data sets.

5.5.3.1 Overall SNV calling runtime

Considering the bulk analysis runtime *without* base pruning, DBS$_{base}$ always outperforms DBS$_{seq}$. As expected the benefit of DBS$_{base}$ to avoid data conversions at all pays out. Since we process the complete data set, DBS$_{seq}$ cannot benefit from its capability to reduce the join processing effort on small genome regions (cf. Section 4.3.2.2.). The

[9]The number of bits per sample base in column SB_INSERT_OFFSET is less than 0.1 bits and, thus, negligible considering the overall storage consumption

results also show that our database approaches compete with specialized analysis tools on different data sets with regard to analysis runtime. However, SAMTOOLS and our database approaches appear to be sensitive to different data set characteristics.

5.5.3.2 Impact of data characteristics

In this section, we investigate the impact of data characteristics on the runtime of the single approaches in detail. Here, we focus on the runtimes without *base pruning* (non hatched bars).

Impact of reference genome size and coverage

DBS_{base} is significantly faster than SAMTOOLS on *DataSet 1*, but slower on *DataSet 2*, although the number of sample bases that have to be processed does not change that much to explain the large speed up of 80% of SAMTOOLS. SNV calling using SAMTOOLS involves a second tool called BCFTOOLS that performs the actual SNV calling, i.e., assessing the computed probabilities and deciding whether a SNV is present or not (cf. Section 3.1.2). A detailed analysis of the interaction between SAMTOOLS and BCFTOOLS revealed that SAMTOOLS streams intermediate results per genome position in the form of strings to BCFTOOLS, which leads to the observed overhead. Since the reference genome size and, thus, the number of genome positions that have to be processed differs by 90% between *DataSet 1* and *2*, the overhead due to this mode of operation is the explanation for the large speed up of SAMTOOLS on *DataSet 2*. DBS_{base} does not benefit that much from the reduced number of genome positions that have to be analyzed in *DataSet 2*. Of course, we reduce some overhead for managing the aggregation array, but the number of sample bases that have to be processed is the dominating factor for the processing runtime of DBS_{base} as well as DBS_{seq}. We observe the trade-off between both data characteristics, i.e., the number of genome positions that have to be analyzed and the number of sample bases that have to be processed (coverage), considering the runtime of DBS_{base} and SAMTOOLS on *DataSet 3*. Both approaches are equally fast (without base pruning). DBS_{base} benefits from less sample bases that have to be analyzed due to the low average coverage of two, while SAMTOOLS suffers from the the large number of genome positions that have to be analyzed.

Impact of mismatch rate and the effectiveness of base pruning

The fact that our database approaches are dominated by the number of sample bases that have to be processed is confirmed by considering the analysis runtime of DBS_{base} with *base pruning*. *Base pruning* allows us to limit the analysis to those genome positions that have at least one mismatching base mapped to it. Consequently, we reduce the amount of sample bases that have to be processed, which results in large speed ups making DBS_{base} as fast as SAMTOOLS (cf. *DataSet 2*) and even faster (cf. *DataSet 1* and *3*). The performance improvements of SAMTOOLS due to *base pruning* are less than those of DBS_{base}. Since SAMTOOLS operates on heavyweight compressed data, we first have to decompress and convert the mapped reads before we can make use of the information about which genome positions does not have to be analyzed (cf. Section 5.4.2). Moreover, we did not found a possibility to stop SAMTOOLS from streaming intermediate results that are not required due to the use of base pruning. Thus, SAMTOOLS only benefits from base pruning by avoiding the computation of not needed error probabilities during aggregation.

5.5.4 Overall assessment

Our experiments show that the results that we collected on our small evaluation data set to devise and improve our approaches can be confirmed on large real-world data sets. Overall, we are able to compete with specialized analysis tools regarding SNV calling runtime. Although our database approaches suffer from processing large numbers of sample bases, we effectively mitigate this disadvantage using our base pruning technique. In combination with DBS_{base}, we can even outperform SAMTOOLS by up to a factor of five. Moreover, we can reduce the storage overhead of the base-centric database schema effectively. Nevertheless, we still require more storage than specialized flat file formats such as BAM or CRAM that additionally apply heavyweight compression schemes. Thus, incorporating such compression schemes for single columns or to archive cold data might be of interest in future work. Our results already show that simply zipping the database files of DBS_{base} leads to comparable storage sizes than specialized flat file formats.

Overall, we observe that our proposed compression schemes for relational DBMSs, i.e., Delta+RLE and reference-based compression, and our *base pruning* technique benefit from increasing read lengths and more accurate genome data having smaller mismatch rates. DNA sequencing techniques will generate longer and more accurate reads with every new generation [82]. Thus, our techniques will benefit from these improvements and provide a robust foundation for genome analysis based on relational database systems.

5.6 Related work

In this chapter, we leverage the similarity of sample and reference genome to provide lightweight database compression schemes and to speed up the SNV calling using base pruning. In this section, we briefly review further reference-based analysis techniques that are related to our approaches.

CAGe is a variant calling approach that leverages the similarity between reference genome and sample genome to reduce the analysis runtime of variant detection [12]. To this end, CAGe uses the information about sequence similarity to classify genome regions regarding their analysis complexity. If a region has less mismatches it is of low complexity and less sophisticated, but faster variant calling approaches are used to analyze them. In contrast, if a region has many mismatches more sophisticated approaches are applied. We could use our base-pruning approach to further reduce the runtime to analyze low complexity regions, which can improve the overall analysis runtime.

RCSI proposed by Wandelt et al. [132] is another approach that leverages the similarity between reference and sample genome to provide similarity search on referentially compressed genomes. To this end, in a first step, the reference sequence is searched to find matching segments allowing for errors. In a refinement step, the compressed sample genomes are searched based on the findings from the first search. The results are combined to generate the final result. Our base-centric database schema in combination with reference-based compression can be the basis to integrate this technique into a relational database system.

5.7 Wrap up

In this chapter, we answered our fourth research question: *How can we store genome data sets using a relational DBMS as efficient as state-of-the-art flat-file approaches without sacrificing analysis performance?*. To this end, we started with an initial storage consumption evaluation. The evaluation revealed that relational DBMSs suffer from missing genome-specific compression schemes. Mapped read data consists mostly of unique strings that are hard to compress using lightweight compression schemes, since these are designed to leverage specific data characteristics (cf. Section 5.1.1) that are not present in unique DNA sequence strings. Our base-centric database schema mitigates this limitation, since it stores the single bases of DNA sequences explicitly allowing for standard BITDICT compression of genome data. However, the storage overhead due to explicitly storing position and mapping information exceeds the storage savings by far. To overcome this limitation, we integrated reference-based compression based on the base-centric database schema into a relational DBMS, which allows for more efficient compression of DNA sequences than using BITDICT. Furthermore, we proposed Delta+RLE encoding that effectively reduces the large storage increase of the base-centric database schema due to explicitly storing position and mapping information. Using our compression techniques, we can consider the base-centric database schema as alternative primary storage layout that allows for lightweight integration of genome-specific compression and, thus, for efficient storage of genome data sets within relational DBMSs. Using a traditional read-centric database schema, we would have to apply string compression schemes that likely prohibit the efficient and direct processing of mapped read data in a relational DBMS.

Besides investigating genome-specific compression to improve the storage consumption, we also investigated efficient access techniques for compressed data. Moreover, we proposed a technique called *base pruning* that allows us to effectively reduce the number of genome positions that have to be processed by leveraging the explicit mapping information in the base-centric database schema. Our technique is enabled by the holistic view on mapped read data that is not limited to compressed blocks as used by specialized flat file formats. We can even leverage the lightweight reference-based compression scheme to speed up the base pruning computation. Applying the base pruning technique to SAMTOOLS, a specialized analysis tools, revealed the different design principles behind a DBMS and specialized analysis tools. The specialized analysis tools are designed for their specific purpose relying on sophisticated, tightly coupled processing mechanisms making it hard to extend them. Thus, they can hardly benefit from our base pruning optimization.

The final evaluation on three large real-world data sets confirmed that our proposed techniques lead to competitive analysis runtime compared to SAMTOOLS. In best case, we can outperform SAMTOOLS by up to a factor of five, because SAMTOOLS is designed to always process all genome positions and cannot restrict the processing effort to those genome positions that really must be analyzed as our database approaches can do. What remains are further investigations to incorporate heavyweight compression into our database approaches, which would allow us to use the database as archive for genome data, too. Currently, we are able to reduce the storage requirements compared to standard lightweight compression schemes available in relational DBMSs by up to 50% still requiring 2 to 2.5 times more storage than BAM.

6. Conclusion

In this thesis, we investigated approaches to analyze genome data sets from next-generation sequencing (NGS) experiments directly within a relational DBMS. To this end, we mapped the task of variant detection to a relational processing engine, investigated efficient processing strategies and developed genome-specific compression and query processing techniques for relational database systems. Overall, we are able to compete and even outperform specialized analysis tools regarding runtime performance. Our work enables scientists to use relational DBMSs not only to manage results from genome analysis experiments, but to actually perform the analysis directly within the database system. This is a first step to make advanced data management capabilities of relational DBMSs available within the complete genome analysis process. In the following section, we summarize the results of our thesis. Finally, we give an overview on future work.

6.1 Summary

The motivation for our work is the current separation of the analysis process of NGS data into two main analysis steps: detecting genetic variations and investigating their consequences. While the detection of genetic variations is done via specialized analysis tools, the investigation of their consequences is often done using relational DBMSs [70, 72, 115, 128] that provide excellent data integration capabilities for this purpose. The separation complicates reliable data management and analysis, since all involved analysis tools, i.e, specialized analysis tools and DBMSs, have to exchange related information, e.g., about the provenance of data sets or intermediate results, with partly manual effort [112]. An integration of genome analysis tasks into a relational DBMS is one way to remedy the situation requiring a relational DBMS to provide genome-specific analysis functionality as well as efficient processing and storage capabilities for genome data that do not exist yet. To achieve this goal, we raised four research questions. In the following, we summarize our answers and highlight our contributions.

1. Which steps of variant detection should we integrate into a DBMS?

Premise. Read mapping and variant calling are the essential steps of variant detection based on NGS data (cf. Section 2.2.1). Integrating an analysis step into a DBMS should allow us to compute the results on demand in reasonable time within the database system instead of computing them externally and only storing them.

Answer. We should store mapped read data in a DBMS and integrate variant calling as genome-specific analysis task. Read mapping always requires to process the complete data set to generate correct results (cf. Section 2.2.1.2). Thus, it is not realistic to compute read mappings on demand, since the runtime depends on the size of the data set. In contrast, variant calling generates results on a subset of the mapped reads, i.e., reads that map to a specific genome region (cf. Section 2.2.1.3). Thus, we are able to compute analysis results on demand in reasonable time.

Contribution. In Section 2.3, we characterized the essential analysis steps of variant detection and devised a **concept for variant detection using DBMSs**.

2. How can we express variant detection using relational DBMS operators as a basis?

Premise. In this thesis, we focus on the detection of SNVs that are variations at single genome positions, because their impact and their detection is well researched and the process is established (cf. Section 2.2.1.3).

Answer. Since SNV calling requires to process single bases at specific genome positions, we have to use a base-centric database schema that stores mapping information explicitly (cf. Section 3.3.1) and allows for direct access to the required information. Thus, we can perform SNV detection using standard relational operators such as joins and selections (cf. Listing 3.8). In contrast, traditional read-centric approaches store mapped read data similar to established flat-file formats that represent reads as strings and encode read mapping information implicitly. Such approaches require to make read mapping information explicit at runtime, which requires specialized UDFs and introduces conversion overhead. Thus, storing mapped reads in a base-centric database schema should allow for faster analyses. In addition, we can use the base-centric database schema as intermediate data representation of mapped reads stored in a read-centric database schema, which allows us to process read-centric data only using relational database operators after they got converted.

Contribution. In Chapter 3, we conceptually extended existing approaches for analyzing mapped reads to support variant calling. From these theoretical considerations, we derived the base-centric database schema to store read mapping information explicitly in the database allowing for **SNV detection as aggregation task** and, thus, for purely relational SNV calling.

3. How can we process genome data sets as efficient as specialized analysis tools using relational DBMSs?

Premise. We focus our research on main-memory DBMSs that have shown to provide tremendous performance improvements especially for analytical workloads [44, 60].

Answer. We have to use advanced join and aggregation processing such as invisible join [3] and array-based aggregation to enable efficient SNV calling using relational DBMSs. Otherwise, we cannot compete with specialized analysis tools, since traditional relational processing strategies such as hash join and sort-based aggregation introduce inherent processing overhead due to the pruning of hash tables (cf. Section 4.3.1) or sorting of intermediate results (cf. Section 4.4). Specialized analysis tools rely on a presorting of mapped reads by mapping position and leverage the sorting during processing (cf. Section 3.1.2). Integrating such an approach into a DBMS using sophisticated genome-specific UDFs could lead to competitive results, but circumvents the relational processing engine completely, hence, hardly justifying a technology shift and limiting the benefits of using a DBMS.

Using invisible join and array-based aggregation, we leverage inherent data characteristics of the mapped reads stored in a base-centric database schema to speed up the query processing (cf. Section 4.3.1 and Section 4.4.1). Using the proposed processing optimizations, we can reduce the overall processing effort by up to 70% compared to a standard implementation using a hash join and sort-based aggregation. Thus, we are able to compete with specialized analysis tools with regard to analysis runtime.

Contribution. In Chapter 4, we investigated the implementation space of our relational SNV calling query and analyzed possibilities to leverage inherent data characteristics to apply advanced join and aggregation processing approaches providing the foundation for **efficient declarative SNV detection**.

4. How can we store genome data sets using a relational DBMS as efficient as state-of-the-art flat-file approaches without sacrificing analysis performance?

Premise. In particular, in main-memory database system, we should avoid to use heavyweight compression schemes that introduce decompression overhead (cf. Section 5.1.1) and would sacrifice possible performance gains. Therefore, we focus our research on the use of lightweight compression schemes only.

Answer. We have to use a base-centric database schema that stores mapping information explicitly to store *and* process genome data, i.e., mapped reads, as efficient as possible using a relational DBMS. The explicit access to position and mapping information within the base-centric database schema allows us to integrate reference-based compression, a genome-specific compression scheme, in a lightweight fashion into a DBMS (cf. Section 5.3.1). Nevertheless, the base-centric database schema requires to store position and mapping information of every single base explicitly, which obviously leads to a large storage increase compared to a read-centric database schema. However, we can mitigate this overhead using our lightweight Delta+RLE encoding. Delta+RLE encoding compresses runs of consecutive numbers that are incremented by a specific offset, which is an inherent characteristic of explicit position and mapping information (cf. Section 5.3.2). Overall, we are able to reduce the storage requirements of a DBMS using a base-centric database schema compared to a read-centric database schema by up to 25%. Applying lossy compression, the storage reduction is up to 50%. Moreover, we can leverage reference-based compressed data to speed up the calling of SNVs. The compression scheme indirectly indicates which genome positions will not show a SNV (cf. Section 5.4). Thus, we can leave out such genome positions during analysis leading to a speed up of up to a factor of five compared to specialized analysis tools.

However, using only lightweight compression schemes prevents us from storing genome data sets as efficient as state-of-the-art flat-file formats. These flat-file formats rely on heavyweight compression schemes that provide better compression ratios, but also introduce computational overhead due to decompression that we aim to minimize.

Contribution. In Chapter 5, we investigated **genome-specific compression in column stores** and explained how we can integrate reference-based compression into a relational DBMS leveraging the explicit mapping information provided by the base-centric database schema. Moreover, we proposed a compression scheme called Delta+RLE that is designed to effectively reduce the inherent storage overhead of the base-centric database schema due to the explicit storage of mapping information. At the same time, we extended our lightweight compression schemes with mechanisms to speed up random and sequential accesses and showed the impact of these extensions on runtime performance and storage consumption. Moreover, we considered **genome-specific query optimization** and proposed a technique called base pruning that effectively reduces the number of genome positions that we have to analyze. This optimization can leverage reference-based compressed data as index to speed up the processing further.

6.2 Discussion

The current attempts to improve reproducibility in life sciences demand enhancements of existing research methodologies [9, 112]. Introducing advanced data management capabilities plays a vital role within these enhancements enabling researchers to reliably and efficiently manage scientific data as well as to comprehensively track analysis processes. To implement the required data management capabilities, we can either extend existing analysis pipelines with missing data management functionality (option A) or integrate analysis functionality into existing DBMSs that are already designed to offer advanced data management capabilities (option B).

In this thesis, we researched *option B* in the context of variant detection based on genome data from NGS experiments. Our solutions allow us to integrate SNV calling, a part of variant detection, into a relational DBMS. Thus, we can avoid to use specialized analysis tools to operate on the raw data externally. Moreover, we can express analyses in a declarative way using SQL, which inherently provides a description of the performed analyses.

To provide SNV calling via SQL, we took care to use relational database operators as basis to perform the analysis. Thus, we can apply our approach to detect SNVs to any relational database system as long as it can be extended with UDAs. However, to provide competitive analysis speed and storage consumption compared to state-of-the-art solutions that are not based on a DBMS, it is crucial to leverage domain-specific data and query characteristics and to extend the DBMS with advanced operator implementations and lightweight genome-specific compression schemes.

Although we conducted our research using a main-memory DBMS, we can apply our solutions also to disk-based DBMSs. We expect that the disk access increases analysis runtime. However, the sequential data access patterns that are enforced by our processing approach, in particular on sample genome data, should also payout in a disk-based DBMS, since the buffer manager also benefits from sequential data accesses to disk. In combination with a column-oriented data layout and our lightweight compression schemes the available disk bandwidth should be leveraged effectively.

Our relational SNV calling approach enables the DBMS to freely decide about the best execution strategy of an analysis query at runtime. In our experimental setup, we fixed the query execution plan, but we also discussed options for optimizations due to data and query characteristics (cf. Section 4.2.1). Another optimization degree of DBMSs is the decision where to execute (parts of) a query. For example, a GPU-accelerated DBMS can perform (parts of) the query on a GPU to speed up the analysis workload, which allows for co-processor acceleration of genome analysis tasks without manual adjustments or adaptations of specialized analysis tools.

6.3 Future work

In this section, we introduce future research directions based on our results presented in this thesis.

Integrating further genome-specific functionality

In this thesis, we focused on the integration of the core functionality behind SNV calling. In future work, it would be interesting to complement this core functionality with additional genome-specific analysis functionality:

Enhanced SNV calling. In this thesis, we left out several optimizations to improve the result quality of SNV calling such as base call quality recalibration [75] or local realignments of single reads [30] in the presence of small insertion and deletions. Integrating such concepts could lead to improved analysis results. Moreover, our approach offers the freedom to use different genotype aggregation functions during the query processing. Thus, it could be beneficial to combine the results of different approaches to further improve the result quality.

Advanced variant calling. Another research direction is to integrate approaches to detect other kinds of variations such as structural variations or small insertions and deletions (cf. Section 2.2.1.3) that we have not considered in this thesis.

Integrate read mapping functionality. In this thesis, we did not integrate read mapping functionality into a DBMS, since the task is too complex and time consuming to be finished in reasonable time on demand. Read mapping is a necessary preprocessing step of unmapped raw reads that has to be done in advance. Nevertheless, it could be possible to compute alternative read mappings on demand using an existing read mapping as index. Such extensions are related to the idea of realignments of reads before or during variant calling.

Storage and runtime optimizations

In this thesis, we focused on the use of lightweight compression schemes and assumed that the logical data representation described by our database schemata resembles the physical one. In future work, following research directions are possible:

Combine base-centric and read-centric database approaches. We have shown that a read-centric database approach outperforms a base-centric approach on small genome regions (cf. Section 4.2.2), since we can easily reduce the number of reads to analyze according to the genome region of interest. A base-centric database approach requires to process all sample bases stored in the database independent of the given genome region, but does not require to convert data on-the-fly, which leads to superior runtime on large genome regions. Combining both characteristics is a logical idea, which can be achieved by storing reads of a read-centric database schema in a base-centric storage layout. Thus, we can reduce the effort of conversion and use the logical read-centric representation for efficient filtering. At the same time, we reduce the storage footprint of a read-centric database, since we can apply our proposed lightweight compression schemes.

Integrate heavyweight compression and disk-based storage. Keeping all data permanently in main memory will not be a solution in the mid term, since hard disks are still a cheap and robust storage backend. Moreover, our results show that lightweight compression cannot compete with heavyweight compression schemes (cf. Section 5.5.2). Thus, it appears to be necessary to integrate heavyweight compression and disk-based storage to provide a scalable genome analysis platform based on a relational DBMS. Genome-specific indexing should allow us to spot the required data on disk and a column-oriented storage layout in combination with heavyweight compression reduces the effort to transfer data from disk to main memory. However, to ensure a seamless integration with disk-based DBMSs further research is required. For example, it is not clear yet, how we can leverage genome data characteristics to speed up the access to and decompression of heavyweight compressed data residing on disk.

Bibliography

[1] 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012.

[2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[3] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[5] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene. W. Myers, and David J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, October 1990.

[6] Vineet Bafna, Alin Deutsch, Andrew Heiberg, et al. Abstractions for genomics. *Commun. ACM*, 56(1):83–93, January 2013.

[7] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[8] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Philos. Trans. Roy. Soc. London*, 53:370–418, 1763.

[9] C. Glenn Begley and John P. A. Ioannidis. Reproducibility in science: improving the standard for basic and preclinical research. *Circ. Res.*, 116(1):116–126, 2015.

[10] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[12] Adam Bloniarz, Ameet Talwalkar, Jonathan Terhorst, et al. Changepoint analysis for efficient variant calling. In *RECOMB*, pages 20–34, 2014.

[13] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[14] Sebastian Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[15] Sebastian Breß, Felix Beier, Hannes Rauhe, et al. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084–1096, 2013.

[16] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, pages 1891–1906, 2016.

[17] Sebastian Breß, Ingolf Geist, Eike Schallehn, Maik Mory, and Gunter Saake. A framework for cost based optimization of hybrid CPU/GPU query plans in database systems. *Control and Cybernetics*, 41(4):715–742, 2012.

[18] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-accelerated database systems: Survey and open challenges. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 15:1–35, 2014.

[19] Y. Bromberg. Building a genome analysis pipeline to predict disease risk and prevent disease. *J. Mol. Biol.*, 425(21):3993–4005, 2013.

[20] Christian Burks, Michael J.Cinkosky, William M.Fischer, et al. GenBank. *Nucleic Acids Res.*, 20:2065–2069, 1991.

[21] Michele Cargill, David Altshuler, James Ireland, et al. Characterization of single-nucleotide polymorphisms in coding regions of human genes. *Nat. Genet.*, 22(3):231–238, 1999.

[22] Stefano Ceri, Abdulrahman Kaitoua, Marco Masseroli, Pietro Pinoli, and Francesco Venco. Data management for next generation genomic computing. In *EDBT*, pages 485–490, 2016.

[23] Robin Cijvat, Stefan Manegold, Martin Kersten, et al. Genome sequence analysis with MonetDB. *Datenbank-Spektrum*, 15(3):185–191, 2015.

[24] Karen Clark, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and Eric W. Sayers. GenBank. *Nucleic Acids Res.*, 44(D1):D67–D72, 2016.

[25] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[26] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[27] Mike Cornell, Norman W. Paton, Shengli Wu, et al. GIMS – A data warehouse for storage and analysis of genome sequence and functional data. In *BIBE*, pages 15–22, 2001.

[28] CRAM Format Specification Working Group. *CRAM Format Specification*, 2015.

[29] Pietro D'Addabbo, Luca Lenzi, Federica Facchin, et al. GeneRecords: A relational database for GenBank flat file parsing and data manipulation in personal computers. *Bioinformatics*, 20(16):2883–2885, 2004.

[30] Mark DePristo, Eric Banks, Ryan Poplin, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat. Genet.*, 43(5):491–498, 2011.

[31] P. Deutsch. *GZIP File Format Specification Version 4.3.* United States, 1996.

[32] Yanlei Diao, Abhishek Roy, and Toby Bloom. Building highly-optimized, low-latency pipelines for genomic data analysis. In *CIDR*, 2015.

[33] Sebastian Dorok. The relational way to dam the flood of genome data. In *SIGMOD/PODS Ph.D. Symposium*, pages 9–13, 2015.

[34] Sebastian Dorok. Memory efficient processing of DNA sequences in relational main-memory database systems. In *GvDB*, pages 39–43, 2016.

[35] Sebastian Dorok, Sebastian Breß, Horstfried Läpple, and Gunter Saake. Toward efficient and reliable genome analysis using main-memory database systems. In *SSDBM*, pages 34:1–34:4, 2014.

[36] Sebastian Dorok, Sebastian Breß, and Gunter Saake. Toward efficient variant calling inside main-memory database systems. In *BIOKDD-DEXA*, pages 41–45, 2014.

[37] Sebastian Dorok, Sebastian Breß, Jens Teubner, et al. Efficient storage and analysis of genome data in databases. In *BTW*, pages 423–442, 2017.

[38] Sebastian Dorok, Sebastian Breß, Jens Teubner, et al. Efficiently storing and analyzing genome data in database systems. In *Datenbank-Spektrum*, 2017. doi:10.1007/s13222-017-0254-9.

[39] Sebastian Dorok, Sebastian Breß, Jens Teubner, and Gunter Saake. Flexible analysis of plant genomes in a database management system. In *EDBT*, pages 509–512, 2015.

[40] Mohamed Y. Eltabakh, Mourad Ouzzani, and Walid G. Aref. bdbms - A database management system for biological data. In *CIDR*, pages 196–206, 2007.

[41] Adam C. English, William J. Salerno, Oliver A. Hampton, et al. Assessing structural variation in a personal genome - Towards a human reference diploid genome. *BMC Genom.*, 16(1):286, 2015.

[42] Brent Ewing and Phil Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Res.*, 8(3):186–194, 1998.

[43] Cindy Fähnrich, Matthieu-P. Schapranow, and Hasso Plattner. Facing the genome data deluge: Efficiently identifying genetic variants with in-memory database technology. In *SAC*, pages 18–25, 2015.

[44] Franz Färber, Sang K. Cha, Jürgen Primsch, et al. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, 2012.

[45] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

[46] Kathleen M. Giacomini, Claire M. Brett, Russ B. Altman, et al. The pharmacogenetics research network: From SNP discovery to clinical drug response. *Clin. Pharmacol. Ther.*, 81(3):328–345, 2007.

[47] Goetz Graefe. Volcano - An extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[48] Goetz Graefe, Haris Volos, Hideaki Kimura, et al. In-memory performance for big data. *PVLDB*, 8(1):37–48, 2014.

[49] Hákon Guðbjartsson, Guðmundur Fr. Georgsson, Sigurjón A. Guðjónsson, et al. Gorpipe: A query tool for working with sequence data based on a genomic ordered relational (gor) architecture. *Bioinformatics*, 32(20):3081–3088, April 2016.

[50] Margaret A. Hamburg and Francis S. Collins. The path to personalized medicine. *N. Engl. J. Med.*, 363(4):301–304, 2010.

[51] Rajini R. Haraksingh and Michael P. Snyder. Impacts of variation in the human genome on gene regulation. *J. Mol. Biol.*, 425(21):3970 – 3977, 2013.

[52] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[53] Bingsheng He, Mian Lu, Ke Yang, et al. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, 2009.

[54] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2 edition, 1996.

[55] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 5 edition, 2012.

[56] Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res.*, 21(5):734–740, 2011.

[57] Ruey-Lung Hsiao, D. Stott Parker, and Hung-chih Yang. Support for bioindexing in BLASTgres. In *DILS*, pages 284–287, 2005.

[58] Tim Hubbard, Daniel Barker, Ewan Birney, et al. The Ensembl genome database project. *Nucleic Acids Res.*, 30(1):38–41, 2002.

[59] David A. Huffman. A method for construction of minimum-redundancy codes. *Proceedings IRE*, 40(9):1098–1101, 1952.

[60] Stratos Idreos, Fabian Groffen, Niels Nes, et al. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[61] Illumina. Reducing whole-genome data storage footprint. Technical report, Illumina Inc., 2012.

[62] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[63] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[64] Robert Kallman, Hideaki Kimura, Jonathan Natkins, et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[65] Changkyu Kim, Tim Kaldewey, Victor W. Lee, et al. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[66] Yuichi Kodama, Martin Shumway, and Rasko Leinonen. The sequence read archive: Explosive growth of sequencing data. *Nucleic Acids Res.*, 40(D1):D54–D56, 2012.

[67] Christos Kozanitis, Andrew Heiberg, George Varghese, and Vineet Bafna. Using genome query language to uncover genetic variation. *Bioinformatics*, 30(1):1–8, 2014.

[68] Christos Kozanitis and David A. Patterson. GenAp: A distributed SQL interface for genomic data. *BMC Bioinform.*, 17:63, 2016.

[69] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624. IEEE, 2010.

[70] Christian Kuenne, Ivo Grosse, Inge Matthies, et al. Using data warehouse technology in crop plant bioinformatics. *J. Integr. Bioinform.*, 4(1), 2007.

[71] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, 10(3):R25, 2009.

[72] Thomas J. Lee, Yannick Pouliot, Valerie Wagner, et al. BioWarehouse: A bioinformatics database warehouse toolkit. *BMC Bioinform.*, 7(1):170, 2006.

[73] Ulf Leser, Hugues Roest Crollius, Hans Lehrach, and Ralf Sudbrak. IXDB, an X chromosome integrated database. *Nucleic Acids Res.*, 27(1):123–127, 1999.

[74] Heng Li. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics*, 27(21):2987–2993, 2011.

[75] Heng Li. Improving SNP discovery by base alignment quality. *Bioinformatics*, 27(8):1157–1158, 2011.

[76] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[77] Heng Li, Bob Handsaker, Alec Wysoker, et al. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[78] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Brief. Bioinform.*, 11(5):473–483, 2010.

[79] Ruiqiang Li, Yingrui Li, Xiaodong Fang, et al. SNP detection for massively parallel whole-genome resequencing. *Genome Res.*, 19(6):1124–1132, 2009.

[80] Yinan Li and Jignesh M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[81] Yinan Li, Allison Terrell, and Jignesh M. Patel. WHAM: A high-throughput sequence alignment method. In *SIGMOD*, pages 445–456, 2011.

[82] Lin Liu, Yinhu Li, Siliang Li, et al. Comparison of next-generation sequencing systems. *J. Biomed. Biotechnol.*, 2012:1–11, 2012.

[83] Aaron J. Mackey and William R. Pearson. Using relational databases for improved sequence similarity searching and large-scale genomic analyses. In *Current Protocols in Bioinformatics*, chapter 9, pages 9.4.1 – 9.4.25. John Wiley & Sons, Inc., 2004.

[84] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[85] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. Cache-conscious radix-decluster projections. In *VLDB*, pages 684–695, 2004.

[86] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[87] Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database Architecture Evolution: Mammals flourished long before Dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[88] Elaine Mardis. The $1,000 genome, the $100,000 analysis? *Genome Med.*, 2(11):84, 2010.

[89] Victor M. Markowitz, Frank Korzeniewski, Krishna Palaniappan, et al. The integrated microbial genomes (IMG) system: A case study in biological data management. In *VLDB*, VLDB '05, pages 1067–1078. VLDB Endowment, 2005.

[90] Marco Masseroli, Pietro Pinoli, Francesco Venco, et al. Genometric query language: A novel approach to large-scale genomic data management. *Bioinformatics*, 31(12):1881–1888, 2015.

[91] Nasim Mavaddat, Susan Peock, Debra Frost, et al. Cancer risks for BRCA1 and BRCA2 mutation carriers: Results from prospective analysis of EMBRACE. *Journal of the National Cancer Institute*, 105(11):812–822, 2013.

[92] Aaron McKenna, Matthew Hanna, Eric Banks, et al. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res.*, 20(9):1297–1303, 2010.

[93] Andreas Meister, Sebastian Breß, and Gunter Saake. Toward GPU-accelerated database optimization. *Datenbank-Spektrum*, 15(2):131–140, 2015.

[94] Michael L. Metzker. Emerging technologies in DNA sequencing. *Genome Res.*, 15(12):1767–1776, 2005.

[95] Michael L. Metzker. Sequencing technologies - the next generation. *Nat. Rev. Genet.*, 11(1):31–46, 2009.

[96] Rasmus Nielsen, Joshua S. Paul, Anders Albrechtsen, and Yun S. Song. Genotype and SNP calling from next-generation sequencing data. *Nat. Rev. Genet.*, 12(6):443–51, 2011.

[97] Nomenclature Committee of the International Union of Biochemistry (NC-IUB). Nomenclature for incompletely specified bases in nucleic acid sequences. *Biochem. J.*, 229(2):281–286, 1985.

[98] Oracle. *Oracle Database In-Memory with Oracle Database 12c Release 2*, 2016.

[99] Jason O'Rawe, Tao Jiang, Guangqing Sun, et al. Low concordance of multiple variant-calling pipelines: practical implications for exome and genome sequencing. *Genome Med.*, 5(3):28, 2013.

[100] Stephan Pabinger, Andreas Dander, Maria Fischer, et al. A survey of tools for variant analysis of next-generation genome sequencing data. *Brief. Bioinform.*, 15(2):256–278, 2013.

[101] Norman W. Paton, Shakeel A. Khan, Andrew Hayes, et al. Conceptual modelling of genomic information. *Bioinformatics*, 16(6):548–557, 2000.

[102] Peter L. Pearson. The genome data base (GDB) - a human gene mapping repository. *Nucleic Acids Research*, 19(suppl):2237–2239, 1991.

[103] William R. Pearson. *The FASTA program package*, 2015. Manual.

[104] William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85(8):2444–2448, 1988.

[105] Jonathan Pevsner. *Bioinformatics and functional genomics.* Wiley, 3 edition, 2015.

[106] Vijayshankar Raman, Garret Swart, Lin Qiao, et al. Constant-time query processing. In *ICDE*, pages 60–69, 2008.

[107] Daniel J. Rigden, Xosé M. Fernández-Suárez, and Michael Y. Galperin. The 2016 database issue of nucleic acids research and an updated molecular biology database collection. *Nucleic Acids Res.*, 44(D1):D1–D6, 2016.

[108] Uwe Röhm and José A. Blakeley. Data Management for High-Throughput Genomics. In *CIDR*, 2009.

[109] Uwe Röhm and Thanh-Mai Diep. How to BLAST your database - A study of stored procedures for BLAST searches. In *DASFAA*, pages 807–816, 2006.

[110] Wolfgang Sadée and Zunyan Dai. Pharmacogenetics/ -genomics and personalized medicine. *Hum. Mol. Genet.*, 14(suppl 2):R207–R214, 2005.

[111] SAM/BAM Format Specification Working Group. *Sequence Alignment/Map Format Specification*, 2016. Manual.

[112] Geir K. Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLoS Comput. Biol.*, 9(10), 2013.

[113] Frederick Sanger, S Nicklen, and A R Coulson. DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA*, 74(12):5463–5467, 1977.

[114] Matthieu-P. Schapranow and Hasso Plattner. HIG - an in-memory database platform enabling real-time ana- lyses of genome data. In *BigData*, pages 691–696, 2013.

[115] Sohrab P. Shah, Yong Huang, Tao Xu, et al. Atlas - a data warehouse for integrative bioinformatics. *BMC Bioinform.*, 6:34, 2005.

[116] Barkur S. Shastry. SNP alleles in human disease and evolution. *J. Hum. Genet.*, 47(11):561–566, 2002.

[117] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[118] Stephen T. Sherry, Minghong Ward, M Kholodov, et al. dbSNP: The NCBI database of genetic variation. *Nucleic Acids Res.*, 29(1):308–311, 2001.

[119] David Sims, Ian Sudbery, Nicholas E. Ilott, Andreas Heger, and Chris P. Ponting. Sequencing depth and coverage: Key considerations in genomic analyses. *Nat. Rev. Genet.*, 15(2):121–132, 2014.

[120] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, 1981.

[121] Arne Stabenau, Graham McVicker, Craig Melsopp, et al. The Ensembl core software libraries. *Genome Res.*, 14(5):929–933, 2004.

[122] Lincoln D. Stein and Jean Thierry-Mieg. AceDB: A genome database management system. *Computational Methods In Genome Research*, pages 45–55, 1994.

[123] Susie M. Stephens, Jake Y. Chen, Marcel G. Davidson, Shiby Thomas, and Barry M. Trute. Oracle database 10g: A platform for BLAST search and regular expression pattern matching in life sciences. *Nucleic Acids Res.*, 33(suppl 1):D675–D679, 2005.

[124] Guenter Stoesser, Peter Sterk, Mary Ann Tuli, Peter J. Stoehr, and Graham N. Cameron. The embl nucleotide sequence database. *Nucleic Acids Res.*, 25(1):7–13, 1997.

[125] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al. The end of an architectural era (It's time for a complete rewrite). In *PVLDB*, pages 1150–1160, 2007.

[126] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, et al. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[127] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[128] Thoralf Töpel, Benjamin Kormeier, Andreas Klassen, and Ralf Hofestädt. BioDWH: A data warehouse kit for life science data integration. *J. Integr. Bioinform.*, 5(2), 2008.

[129] Cole Trapnell and Steven L. Salzberg. How to map billions of short reads onto genomes. *Nature biotechnology*, 27(5):455–457, 2009.

[130] Erwin L. van Dijk, Hélène Auger, Yan Jaszczyszyn, and Claude Thermes. Ten years of next-generation sequencing technology. *Trends Genet.*, 30(9):418–426, 2014.

[131] J. Craig Venter, Mark D. Adams, Eugene W. Myers, et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[132] Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. RCSI: Scalable Similarity Search in Thousand(s) of Genomes. *PVLDB*, 6(13):1534–1545, 2013.

[133] Zhen Wang and John Moult. SNPs, protein structure, and disease. *Hum. Mutat.*, 17(4):263–270, 2001.

[134] Kesheng Wu, Ekow Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.

[135] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, 1977.

E h r e n e r k l ä r u n g

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 27.04.2017

Sebastian Dorok