University of Magdeburg
School of Computer Science

Department of Technical and Business Information Systems

Database Workgroup

Master's Thesis

Towards Genome Analysis on Modern Database Systems

Author:

Sebastian Dorok

October 28, 2013

Advisors:

Prof. Dr. habil. Gunter Saake
M. Sc. Sebastian Breß

University of Magdeburg

Dr. Horstfried Läpple
Dr. Philip Groth

Bayer Healthcare AG

Dorok, Sebastian:
Towards Genome Analysis on Modern Database Systems
Master's Thesis, University of Magdeburg, 2013.

Abstract

Genome analysis is an important method to improve disease detection and treatment. The introduction of next generation sequencing techniques allows to generate genome data for genome analysis in less time and at reasonable cost. In order to provide fast and reliable genome analysis, despite ever increasing amounts of genome data, genome data management and analysis techniques must also improve. In this thesis, we discuss advantages of using modern database management systems for genome analysis and present an approach to perform genome analysis on column-oriented, in-memory database management systems. We prepare comprehensive experiments to evaluate the analysis performance and scalability and the disk storage consumption of our approach on two different modern database management systems, namely SAP HANA and MonetDB. Moreover, we show how the integration of genome analysis tasks into the database management system can improve analysis performance of variant calling.

Acknowledgements

I thank my advisors Gunter Saake, Horstfried Läpple, Sebastian Breß, and Philip Groth for their encouraging supervision of this thesis. The guidance of Gunter Saake finally led me to the topic of this thesis and the contribution of Horstfried Läpple shaped the topic to what it is now. Special thanks to Sebastian Breß for his valueable comments on the content and style of this thesis, but also his continous motivation and inspiration. Additonally, I thank Horstfried Läpple and Philip Groth for their help to improve the content and style of my thesis and their effort to provide evaluation hardware at SAP AG and Bayer Healthcare AG, respectively.

I thank Eduard Hergenreider and Emanuel Ziegler from SAP AG for their great support on SAP HANA and fruitful discussions about my evaluation results and possible future work.

Finally, I thank my family and especially my girlfriend Vicki for their infinite patience and support during the preparation of this thesis. Thank you!

Contents

List of Figures xii

List of Tables xiii

List of Code Listings xv

List of Acronyms xvii

1 Introduction 1
- 1.1 Problem . . . 2
- 1.2 Goal . . . 5
- 1.3 Structure . . . 5
- 1.4 Project Environment . . . 6

2 Background 7
- 2.1 Genome Analysis . . . 7
  - 2.1.1 Basic Terms . . . 7
  - 2.1.2 Steps of Genome Analysis . . . 8
    - 2.1.2.1 DNA Sequencing . . . 8
    - 2.1.2.2 Sequence Alignment . . . 9
    - 2.1.2.3 Variant Calling . . . 10
    - 2.1.2.4 Downstream Analysis . . . 11
  - 2.1.3 Common File and Data Formats . . . 11
    - 2.1.3.1 FASTA Format . . . 11
    - 2.1.3.2 FASTQ Format . . . 12
    - 2.1.3.3 SAM Format . . . 13
- 2.2 Database Systems . . . 16
  - 2.2.1 Basic Concepts . . . 16
    - 2.2.1.1 Database System Approach . . . 16
    - 2.2.1.2 Features of Database Management Systems . . . 17
    - 2.2.1.3 Database Models . . . 19
    - 2.2.1.4 Structured Query Language . . . 21
    - 2.2.1.5 Database Application Types . . . 23
  - 2.2.2 Components of Database Management Systems . . . 25
  - 2.2.3 Implementation Techniques of Database Management Systems . 26

2.2.3.1 In-memory versus Disk-based Data Storage 26
2.2.3.2 Column-oriented versus Row-oriented Data Layout . . 28
2.2.3.3 Bulk versus Iterator-like Data Processing 31
2.2.4 Modern versus Traditional DBMS 31
2.3 Summary 32

3 Why to use Modern Database Systems for Genome Analysis? 33
3.1 Challenges in Genome Analysis 33
3.2 Addressing Challenges in Genome Analysis with Modern Database Systems 36
3.3 Conclusion 38

4 Related Work 39

5 Genome Analysis on Modern Database Systems 43
5.1 Genome Analysis as Database Application 43
5.1.1 Application Characteristics 43
5.1.2 Implementation Approaches 45
5.1.3 Research Focus 48
5.2 Variant Calling on Modern Database Systems 48
5.2.1 Basic Concept and Functional Requirements 48
5.2.2 Database Schema 50
5.2.3 Querying and Filtering 52
5.2.4 Analysis Integration 55
5.3 Summary 57

6 Evaluation 59
6.1 Evaluation Objectives 60
6.1.1 Independent Variables 60
6.1.2 Dependent Variables 61
6.2 Experimental Design 62
6.2.1 Confounding Variables 63
6.2.2 Test Environment 64
6.2.2.1 Software 65
6.2.2.2 Hardware 65
6.2.2.3 Test Data 65
6.2.3 Experiments 67
6.2.3.1 Disk Storage Consumption 67
6.2.3.2 Compression Effectiveness 69
6.2.3.3 Analysis Performance and Scalability 70
6.2.3.4 Integrated vs. External Variant Calling Performance and Scalability 71
6.3 Results and Discussion 72
6.3.1 Disk Storage Consumption 72
6.3.2 Compression Effectiveness 74

6.3.3 Analysis Performance and Scalability 76
6.3.3.1 Analysis Performance and Scalability on SAP HANA . 76
6.3.3.2 Further Experiment on Analysis Performance and Scalability on SAP HANA 81
6.3.3.3 Analysis Performance and Scalability on MonetDB . . 83
6.3.4 Integrated vs. External Variant Calling Performance and Scalability 87
6.4 Threats to Validity 90
6.4.1 Threats to Internal Validity 90
6.4.2 Threats to External Validity 93
6.4.3 Conclusion Regarding Threats to Validity 94
6.5 Summary 95

7 Conclusion **97**

A Appendix **103**

Bibliography **109**

List of Figures

1.1 Exemplary genome analysis pipeline 3

2.1 Steps of genome analysis 9
2.2 Structure of database systems 17
2.3 Three schema architecture 18
2.4 Example ER schema 20
2.5 Example relational schema 21
2.6 Example sql schema 22
2.7 Database management components 25
2.8 Example relation *person* 28
2.9 Different data layouts of example data 28
2.10 Dictionary compression of column *city* 30
2.11 Run-length encoding of column *age* 30

3.1 Sequencing throughput and costs by sequencer 34
3.2 Read length and error rate by sequencer 35

5.1 Analysis support 46
5.2 Analysis integration 47
5.3 Basic concept 49
5.4 ER schema with min/max-notation 51
5.5 Genotype and variant calling at a specific genome site 55

6.1 Disk storage consumption of complete database 72

6.2 Breakdown of disk storage consumption in database systems 73
6.3 Average query runtime on SAP HANA 77
6.4 Average query runtime on SAP HANA for queries to detect potentially deleted sites in ranges with sizes between 10,000 and 200,000 base pairs 78
6.5 Logical query plan and simplified execution plan in SAP HANA 79
6.6 Average query runtime on SAP HANA w/o primary and foreign keys . 81
6.7 Average query runtime on MonetDB 83
6.8 Logical query plan to detect potentially variant sites on MonetDB . . . 84
6.9 Execution times for operations of an exemplary query to detect potentially variant sites in a 100,000 base pair range on MonetDB 86
6.10 Average runtime of integrated and external variant calling 88
6.11 Venn diagram for single nucleotide variant calls of bcftools and the DBMS approach on a sample base pair range of size 100,000 89
6.12 Box plot for runtime of queries to detect potentially deleted sites on SAP HANA (30 queries per base pair range size between 10,000 to 200,000 base pairs in 10,000 base pair steps) 92

A.1 Exemplary execution plan for a query to detect insertions in a 100,000 base pair range on database schema with keys 104
A.2 Exemplary execution plan for a query to detect insertions in a 100,000 base pair range on database schema with no keys 105
A.3 Exemplary execution plan for queries to detect variants in a 100,000 base pair range on database schema with keys 106
A.4 Exemplary execution plan for queries to detect variants in a 100,000 base pair range on database schema with no keys 107

List of Tables

2.1 List of single letter codes for nucleobases 12

2.2 Comparison of OLTP and OLAP requirements 24

2.3 Latencies for different actions on storage devices 27

4.1 Database approaches for genome analysis 41

6.1 Hardware and OS specification of Amazon EC2 and SAP HANA instance 66

6.2 Compression rates of single columns of SAP HANA compared to MonetDB 75

6.3 Number of rows of table HG00096_chromosome1 matching different predicates 80

List of Code Listings

2.1 Sequences in FASTA format 12

2.2 Sequences and according quality values file in FASTQ format 13

2.3 Aligned reads in SAM format 15

2.4 Queries to define tables 23

2.5 SELECT-FROM-WHERE statement 23

5.1 Relational model as SQL statement 52

5.2 Data querying 53

5.3 Additional filtering 53

5.4 Query to detect potentially variant sites 54

5.5 Query to detect potentially inserted sites 55

5.6 Query to detect potentially deleted sites 55

5.7 Query for genotype calling 56

5.8 Query for variant calling 57

6.1 Query to detect potentially variant sites using a *left outer join* 85

List of Acronyms

Q_{Phred}
: Phred Quality

CPU
: central processing unit

CSV
: character separated value

DBMS
: database management system

DBS
: database system

DNA
: deoxyribonucleic acid

FASTA
: Fast-All

FASTQ
: Fast-Quality

FTP
: file transfer protocol

GB
: gigabyte; $1GB = 1000^3 byte$

GiB
: gibibyte; $1GiB = 1024^3 byte$

GPU
: graphics processing unit

InDels
small insertions and deletions

OLAP
online analytical processing

OLTP
online transaction processing

RDP
Remote Desktop Protocol

SAM
Sequence Alignment/ Map

SNV
single nucleotide variant

SQL
structured query language

SSH
Secure Script Shell

1. Introduction

According to an International Data Corporation study from 2012, the amount of available data, the so called digital universe, will double every two years until 2020 [GR12]. A research field that facilitates this trend is genome analysis. Thereby, scientists extract and analyze the genetic information of living organisms, the genome. The genetic information is encoded in the deoxyribonucleic acid (DNA) molecule as sequence of base pairs. These base pairs are made up of two of the four bases **A**denine, **C**ytosine, **G**uanine and **T**hymine. A base pair consists of either **A**denine and **T**hymine or **C**ytosine and **G**uanine. Thus, a human genome, comprising 3.2 billion base pairs, can be unambiguously encoded as sequence of the four characters **A**, **C**, **G** and **T**. The sequence of base pairs in a DNA molecule is read in a process called DNA sequencing. The introduction of next generation sequencing techniques [Met09, LLL+12] allows the sequencing of one human genome in days whereas the Human Genome Project that started in 1990 [VA+01] needed years to read the genetic information of humans.

The time savings and decreasing costs in DNA sequencing due to next generation sequencing techniques enable the analysis of more and more human genomes in less time and at reasonable cost. A current example for this trend is the 1000 Genomes Project [VGB10], where nearly 2000 human genomes are sequenced and analyzed in order to get new insights into human life. The analysis of human genomes is important to improve detection and treatment of diseases like cancer or cardiovascular disorders, which are one of the top most causes of death in Europe [WHO12]. With genome analysis, it is possible to detect variations in the genome that can trigger such diseases [Ros00, Sha02] or influence the efficacy of drugs to treat them [EJ01, GBA+07, WMW11].

Since no sequencing technique is capable to sequence the whole DNA at once, the DNA is duplicated and arbitrarily fragmented into millions or billions of overlapping sequences consisting of several hundreds of base pairs. After these sequences are read, they are aligned with the help of computers to restore the complete, original genome sequence. Assuming an uncompressed storage cost of one byte per base pair, one aligned human

genome has a size of 3.2 gigabyte (GB). However, sequence alignment is not perfect due to errors while reading or aligning the base pair sequences. Thus, the raw data must also be stored in order to allow the validation or the repetition of sequence alignments. Human genome sequencing projects generate at least 100 billion base pairs of raw data [LTPS09]. Thus, storing a human genome inclusive the raw base pair sequences would then require more than 100 GB of disk storage per genome.

Despite these huge amounts of data that become available very quickly, genome analysis must be fast and reliable. To allow fast and reliable genome analysis, data management and analysis techniques for genome data must also improve.

In the following sections, we first describe problems of current genome analysis pipelines and introduce our idea to use modern database systems for genome analysis. Then, we describe the goal of this thesis and the thesis' structure. Finally, we introduce the two stakeholders of this project.

1.1 Problem

Genome analysis consists of various steps, always dependent on the concrete analysis' purpose. For analysis from scratch (i.e., the genome is neither sequenced nor aligned), four main steps can be identified (derived from Mount [Mou04]):

Step 1: DNA Sequencing DNA sequencing determines the precise order of bases within fragments of a sample genome. The fragments are called reads.

Step 2: Sequence Alignment Sequence alignment is the alignment of reads to a known reference genome.

Step 3: Variant Calling Variant calling is the process of detecting variations between sample and reference genome.

Step 4: Downstream Analyses Downstream analyses comprises any kind of further analysis on aligned genome data such as detection of regions that encode genes, annotating gene functions, and statistical analysis between different sample genomes in order to find similarities and differences in populations.

A variety of tools exists to perform these steps. These tools are mainly command line tools using flat files as database. The quasi standard flat file formats are Fast-Quality (FASTQ) [CFG+10] to store sequencing data, Sequence Alignment/ Map (SAM) [LHW+09] to store aligned genome sequences, and Variant Calling Format (VCF) [DA+11] to store the results of variant calling. In Figure 1.1, we show an exemplary genome analysis work flow. The results of each step can be used as input for the next steps. Thus, a tool chain that supports all steps of genome analysis can be called a pipeline as the results are piped from tool to tool. Such flat file approaches are common in the field of computational biology [Nob09]. Using flat files has following advantages:

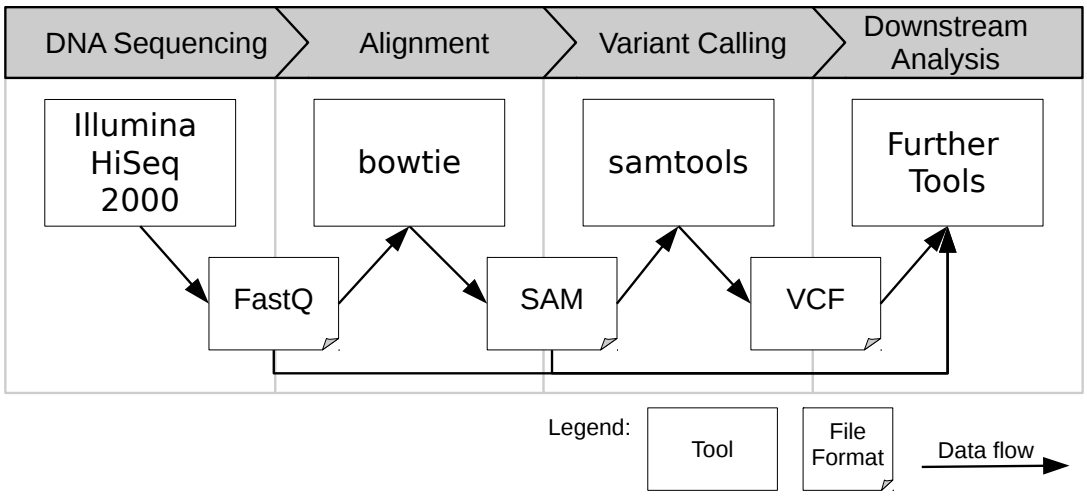


Figure 1.1: Exemplary genome analysis pipeline

Easy Distribution It is easy to distribute genome data via flat files. For example, researchers just have to download the files from servers (e.g., via file transfer protocol (FTP))

No Data Management Software Overhead To store genome data in flat files no additional software is needed, because any file system is sufficient to store flat files (e.g., local or distributed).

Easy Access In case that the genome data is not encoded or compressed, flat files can be easily read with any file reader (e.g., Linux's *less*).

To organize flat files that contain genome data, hundreds of databases exist that provide access to such flat files and try to integrate the contained data with other sources [FSG13], but mainly on a meta data level. Furthermore, to simplify the usage of genome analysis pipelines, web based work flow systems exist [BKC+10] that allow users to choose which tools to run on which data and in which order. Thus, users do not have to care about installing and executing tools on a local machine.

As we show in Figure 1.1, all intermediate results of a genome analysis pipeline are possibly needed for certain downstream analysis tasks and thus all intermediate flat files must be stored. For example, in order to verify a sequence alignment or variant calling result, the raw sequencing data must be available. Thus, storage consumption does not only increase because more sequencing data becomes available, but also because derived intermediate results must be stored. To reduce disk storage consumption of flat files, various compression techniques are proposed with rather good compression ratios [BWB09]. But, these compression techniques are not suited for every kind of genome data (e.g., reads) and do not allow direct processing of compressed data [WRB+12].

Thus, reduced storage consumption is gained at the cost of additional decompression overhead when processing such data.

Another challenge is how fast can a sample genome be processed and analyzed by an analysis pipeline (i.e., analysis performance). Thereby, analysis performance is mainly limited by the most processing intensive tasks sequence alignment and variant calling [WRB+12]. To speed up the processing of sequence alignment and variant calling, tools were developed that use distributed execution environments such as Hadoop[1] [LSL+09]. However, such approaches mainly run remotely on computing clusters and thus the data to process and the processing results must be transferred over the network. With increasing amounts of data this could become a bottleneck [WRB+12].

Furthermore, downstream analysis requires integrated data sources to perform complex analyses [ZBY+11]. But, using flat files prevents or at least limits data integration and each downstream analysis application has to handle the complex integration of different data sources itself. Thus, easy to use data integration features that can be used by analysis applications would improve genome analysis. To address the challenge of limited data integration, academic and commercial approaches were proposed that utilize the data integration strengths of database management systems. Academic approaches are BioDWH [TBAR08], Atlas [SHX+05] and BioWarehouse [LPW+06] that store not only genome sequencing data, but integrate also information about molecular interactions, functional annotations of genes, and biological ontologies to support downstream analysis. Oracle provides an environment that allows alignment and variant calling on distributed platforms [Ora13] with command line tools and stores the result data in an integrated database [Ora12]. On top of these databases, further tools can operate for downstream analyses. But, all approaches store all intermediate results of an analysis pipeline in a database. Thus, with increasing amounts of genome data, storage consumption also increases.

Based on the database approaches for downstream analysis, we suggest to use modern database systems such as in-memory and column-oriented database systems to support all steps of genome analysis (i.e., sequence alignment, variant calling, and downstream analysis). We expect following advantages when using modern database systems for genome analysis:

1. Data management capabilities (e.g., data integration) are available for the complete genome analysis pipeline.
2. Light-weight compression features of modern database systems reduce storage consumption and allow processing of compressed data.
3. Analysis becomes faster due to in-memory processing combined with bulk-processing capabilities of modern database systems.
4. When processing performance is acceptable, intermediate results can be computed on demand instead of storing them. Thus, storage consumption is reduced further.

[1] http://hadoop.apache.org/

1.2 Goal

The goal of this thesis is to show how modern database systems can be used to support genome analysis. Our research questions that we want to answer in this thesis are:

RQ 1 How can modern database systems address challenges in genome analysis?

RQ 2 How to use modern database systems for genome analysis?

RQ 3 How to store genome data in a modern database system?

RQ 4 Is the disk storage consumption of approaches that use modern database systems for genome analysis competitive to flat file approaches?

RQ 5 Is the analysis performance of genome analysis on modern database systems competitive to state-of-the-art tools?

RQ 6 Is the result quality of genome analysis on modern database systems competitive to state-of-the-art tools?

We answer these questions by qualitative discussion and conducting experiments. In order to succeed, we defined following tasks:

- Analyze challenges of genome analysis and show how modern database system can address them.
- Analyze the applicability of modern database systems for genome analysis.
- Create a concept for performing genome analysis on modern database systems.
- Create and conduct experiments to evaluate storage consumption and analysis performance.

First, we analyze what advantages the use of modern database systems for genome analysis has. Next, we analyze whether modern database systems are applicable for genome analysis. Then, we create a concept to support genome analysis on modern database systems. Afterwards, we evaluate this concept regarding storage consumption and analysis performance and present our findings.

1.3 Structure

The thesis is structured as follows:

Chapter 2 In Chapter 2, we introduce important concepts that are necessary for the further understanding of this thesis. Therefore, we will explain topics in the field of genome analysis and database technology.

Chapter 3 In Chapter 3, we discuss how modern database systems address challenges in genome analysis.

Chapter 4 In Chapter 4, we present related work to our research.

Chapter 5 In Chapter 5, we introduce our approach for genome analysis on modern database systems.

Chapter 6 In Chapter 6, we evaluate our approach and conduct experiments to answer our research questions.

Chapter 7 In Chapter 7, we summarize our findings and provide an outlook to future work and new research questions that emerged during preparing this thesis.

1.4 Project Environment

The project has two stakeholders. First stakeholder is the database workgroup of the Department of Technical and Business Information Systems at the University of Magdeburg. The workgroup is interested in database technologies and new applications for databases. Thus, the goals stated in Section 1.2 highly correlate with the workgroup's interests. Second stakeholder is Bayer Healthcare AG. One of Bayer Healthcare AG's research interests is the development of new methodologies to improve disease therapies and drugs. Thus, genome analysis is a hot topic for Bayer Healthcare AG. This thesis supports this interest by evaluating whether modern database systems can improve the genome analysis process.

2. Background

In this chapter, we prepare fundamentals that are necessary to understand further concepts of this thesis. Therefore, we first introduce background information about genome analysis and then explain foundations of modern database technology.

2.1 Genome Analysis

In this section, we introduce basic terms and concepts of genome analysis. Therefore, we first introduce basic terms in genome analysis. Then, we describe the basic steps of genome analysis. Finally, we introduce file and data formats currently used for genome data.

2.1.1 Basic Terms

In this section, we briefly explain important terms that will come up throughout this thesis.

DNA Deoxyribonucleic acid (DNA) is a molecule that encodes the genetic information of living organisms. For example, human DNA consists of two strands of biopolymers that are connected and form a double helix structure. Thereby, each strand consists of nucleotides.

Nucleotide A nucleotide is a small molecule that is made up of a sugar and phosphate group. At each sugar, a nucleobase is attached.

(Nucleo)base There are four nucleobases **a**denine, **c**ytosine, **g**uanine and **t**hymine.

Base Pair DNA consists of two strands of nucleotides that are connected and form a double helix. Each of these connections is made up between two nulceobases, whereby either adenine and thymine or cytosine and guanine form a base pair. Thus, a base pair can be unambiguously encoded by just one base.

Chromosome A chromosome is a physical, single part of DNA containing certain nucleotide sequences such as genes.

Gene A gene comprises a certain nucleotide sequence in a DNA molecule that encode biologically relevant information such as proteins. Thereby, each gene resides at a specific *locus* on a chromosome.

Ploidy Organisms can have multiple chromosomes encoding the same genetic information. These multiple chromosomes make up chromosome sets. The ploidy is the number of chromosomes in such a chromosome set. An organism that has only one chromosome per chromosome set is called *haploid*, whereas an organism with two chromosomes per chromosome set is called *diploid* and so forth. For example, humans have 23 chromosome sets, each containing two chromosomes and are thus diploid organisms.

Allele An allele is a possible variant of DNA sequence at a given locus (e.g., a gene or a position of a DNA sequence). If the two chromosomes in a chromosome set of a diploid organism have different alleles at the same locus, the cells have heterozygous *genotype* with respect to that locus. If both chromosomes have the same allele, the cells have a homozygous *genotype* with respect to that locus.

Genome The genome comprises "[a]ll the genetic material in the chromosomes of a particular organism" [Pev03, p. 898]. Thus, the genome contains all hereditary information of an organism.

2.1.2 Steps of Genome Analysis

Genome analysis is the structural and functional study of genomes. Analyzing genomes of living organisms reveals insights into structural and functional characteristics of life. Genome analysis comprises the process of sequencing and analyzing genomes [Mou04]. In Figure 2.1, we show the four main steps of genome analysis: DNA sequencing (1), sequence alignment (2), variant calling (3), and downstream analysis (4). In the following sections, we explain each step in more detail.

2.1.2.1 DNA Sequencing

DNA sequencing is the process of reading the nucleobases of a DNA sequence (e.g., a chromosome). In Figure 2.1, we show the important steps of DNA sequencing. First, the DNA is extracted from the sample's cells (1.1). In a second step, the DNA is fragmented and duplicated to facilitate the later reading process (1.2). This procedure is also called shotgun sequencing. Afterwards, the fragments are read and the concrete sequence of bases is determined, the so called *read* (1.3). The determination of a nucleobase is also called *calling* as the DNA sequencer has to determine a base value per site in a read. The more reads overlap one genome site, the higher is the sequencing coverage or depth and thus the more base calls are available for the specific genome site. For example, Nielsen et al. state that sequencing data with 20 or more reads that overlap one genome

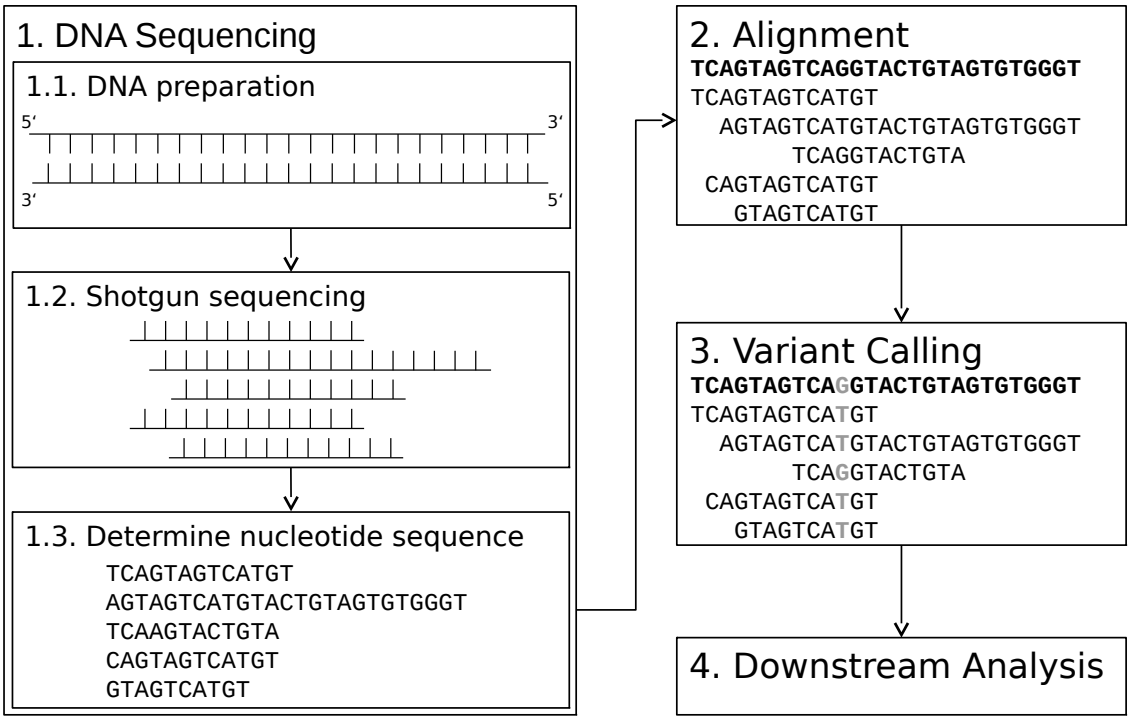


Figure 2.1: Steps of genome analysis

site have a high sequencing coverage [NPAS11]. On the one hand, higher sequencing coverage supports subsequent analysis steps such as sequence alignment and variant calling as more data is available. On the other hand, higher sequencing coverage leads to higher amounts of genome data that must be stored and processed.

2.1.2.2 Sequence Alignment

After DNA sequencing is finished, all small reads must be assembled to the original DNA sequence. Therefore, overlapping reads are determined in order to reconstruct the original DNA sequence. With smaller reads, the assembly task becomes harder, because the overlapping parts of the reads become shorter. Thus, the probability of erroneous assembly increases. Because of this, sequence alignment is used to improve this process. Thereby, all small reads are mapped against a already known reference sequence (e.g., a chromosome or complete genome). The result of such an alignment can be seen in step two in Figure 2.1. The top most, bold sequence of bases is the reference sequence and all reads that were determined in a previous DNA sequencing step are aligned to this reference. A big challenge in sequence alignment is to find the correct mapping site, despite erroneous base calls, real mutations, or variants that exist in the sample sequence and are not present in the reference sequence, and multiple matching sites. In order to map the reads as fast as possible and to find inexact matches, heuristic approaches are used [LH10]. Such approaches speed up the alignment. But, sequence alignment is only possible when a reference sequence is available. Furthermore, the

mapping quality depends on the degree of heuristic the algorithm uses, the quality of the reference sequence, and the base call error rate of the DNA sequencer.

2.1.2.3 Variant Calling

Variant calling algorithms determine variances between a sample genome and a reference genome. Examples for variances are single nucleotide variants (SNVs) and small insertions and deletions (InDels). In the following paragraphs, we explain how SNVs and InDels can be called.

Calling Single Nucleotide Variants

Single nucleotide variants (SNVs) are variances of single bases at a single genome site. In order to detect SNVs, two aligned genome sequences are compared site by site. Thereby, SNV calling algorithms have to distinguish real variants from erroneous base calls or alignments. Furthermore, the task is complicated because reference genomes are mainly haploid, but humans for example are diploid organisms. Thus, a SNV at a first glance can indicate a heterozygous genotype at a second glance. An example for a SNV can be seen in step three in Figure 2.1. The gray base in the reference sequence (top most, bold sequence of bases) is a "G". But four of five reads (sequences below) indicate a "T" at this site. Thus, the variant calling algorithm has to decide whether these four reads are erroneous mapped, or whether the bases are erroneous called or whether it is a real variant.

Nielsen et al. describe different approaches for detecting single nucleotide variants [NPAS11]. They distinguish approaches based on cut-off rules and probabilistic methods.

Cut-off Approaches During sequence alignment, different reads are mapped to a reference sequence (e.g., a chromosome or genome). Thereby, different reads can have different bases that are aligned to the same genome site. Such differences indicate potential variant sites. Approaches based on cut-off rules determine the frequency of different bases at a genome site. Then, they use cut-off rules to determine whether it is a homozygous or heterozygous genotype. For example, a heterozygous genotype is called, if the frequency of one base is between 20 and 80 percent. If not, a homozygous genotype is called. In latter case, this can be either a variant or not. It is a variant, if the non-reference base has a frequency above 80 percent.

Probabilistic Approaches Probabilistic approaches use the base call quality values for variant calling. Based on these quality values, they return a confidence or quality value for a specific variant call. This approaches compute the posterior probability $p(G|B)$ for every possible genotype G (i.e., in case of haploid genotypes $G \in \{A, T, C, G\}$) at a specific genome site given a set of bases B aligned to this site. The genotype with the highest posterior probability is called. It is a variant,

if it differs from the reference. The posterior probability can be used as quality value for this variant call. In order to compute $p(G|B)$, *Bayes' theorem* [Bay63] is used:

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

$p(B|G)$ is the probability to call bases B given a specific genotype G. This probability can be determined by using the single base call quality values. The probability $p(G)$ is the prior probability for genotype G. In case of four possible genotypes and all genotypes have the same probability, the prior probability could be set to 0.25. The probability $p(B)$ is a scaling factor and can be omitted.

Calling Small Insertions and Deletions

Insertions and deletions can be detected in aligned reads using information from the alignment. The alignment tool determines an exact position for each base regarding a reference sequence. If the alignment tool detects an insertion of a base, whereby the read from the sample sequence contains a base that is not present in the reference sequence, but the adjacent bases are, it labels such sites as inserted. Furthermore, if the alignment tool detects a deletion of a base, whereby a reference base is missing in a sample sequence, but the adjacent reference bases are present, it labels these sites as deleted. Detecting deletions and insertions can be done by evaluating such labels.

2.1.2.4 Downstream Analysis

Downstream analysis comprises any further analysis of genome data. Common starting points for downstream analysis are variant sites, because such variances can be triggers for diseases or differences in drug responsiveness. In order to reveal new insights, further data sources must be integrated for downstream analysis. Thus, the genome data is annotated using available information such as gene locations. Furthermore, statistical analysis can be performed to determine allele frequencies within populations. Other applications are visualization of and navigation through genome data.

2.1.3 Common File and Data Formats

In this section, we introduce the quasi-standard file and data formats in genome analysis. We concentrate on the Fast-All (FASTA), Fast-Quality (FASTQ) and Sequence Alignment/ Map (SAM) format as they are most important for our further work.

2.1.3.1 FASTA Format

FASTA format is a text-based format for representing nucleotide or protein sequences. It is named after the FASTA software package for biological sequence comparisons [PL88] that used this format to store nucleotide and protein sequences. Over the years, the format changed slightly. In the following, we concentrate our description of the format on nucleotide sequences and the current version from 2013 that can be found in [Pea13].

```
1  >sequence one (e.g., chromosome 1)
2  CCACTACTAAAAACATAATGCTATAAATAGAATGATGTCTTTTGTTTCCAAAGTCAATAT
3  ACTCGAGCAATGCAAAAATAATAATAAAAGTGAGATACTTCATGGCAAAGCTGCCGCAGG
4  ATAAACATTGCAGCCACAAGTGCCCCCAGTATTCTCGGGGCAAACTGGAAAAGGGCTAAC
5  AGGCAACATTTTCATGTTATTCTACTGAGTGCAGTAATTATTTTTAAAAATATACATGAA
6  >sequence two (e.g., chromosome 2)
7  ACTAAAAGAAGCAACAGCAAAACTGTCACGCATTTGGAGCCATGGCCTGGGTTGGGCCGG
8  TGTAAAGCTCTCCGCCCTCTGGAGCAAGTCTGGGCCCCAGCGGCTGGCATGTGGGCACTG
9  CAGGGCCTGGGTTGGGCAGGTGTGCAGCTCTCCGTCATCTGAGCCTAGTCTGAGGCCTGG
10 TGGCTGGCACGTGGGCCCTGCAGGGCCTCTACTTCTCACCCCAGCTCCACTTCCCTCCCT
```

Listing 2.1: Sequences in FASTA format

Symbol	Meaning	Symbol	Meaning
G	G	S	G or C
A	A	W	A or T
T	T	H	A or C or T
C	C	B	G or T or C
R	G or A	V	G or C or A
Y	T or C	D	G or A or T
M	A or C	N	G or A or T or C
K	G or T		

Table 2.1: List of single letter codes for nucleobases [Nom85, Table 1]

Every file in FASTA format consists of at least one line starting with ">" followed by a line with the actual sequence. The first line is used to describe the sequence starting in the next line. The actual sequence is represented as ASCII characters that encode nucleotide bases. Every line starting with ">" signals the beginning of a new sequence. In Listing 2.1, we present an example file in FASTA format with two sequences. The file starts with the description of the first sequence followed by the sequence itself. Then, in line 6, a new sequence is announced that starts in line 7. Every sequence could represent the nucleotide base sequence of chromosomes of a human reference genome.

The FASTA nucleotide base alphabet comprises more than four letters. In Table 2.1, we give an overview of possible letters in the FASTA nucleotide base alphabet. These codes are necessary to express base call errors due to ambiguous signals. Thus, tools that operate on such data have the possibility to react on such errors.

2.1.3.2 FASTQ Format

FASTQ format is a text-based format for representing biological sequences (e.g., nucleotide sequences) and their corresponding quality scores (e.g., base call qualities). It is used by DNA sequencers to store the sequenced but unaligned reads.

The sequences are expressed as ASCII characters as in FASTA format. Thus, the same alphabet for nucleotide bases applies for FASTQ sequences as for FASTA sequences

```
1 @sequence 1 (e.g., read 1)
2 GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
3 +
4 !''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
5 @sequence 2 (e.g., read 2)
6 GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
7 +
8 !''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Listing 2.2: Sequences and according quality values file in FASTQ format

The quality values are important as DNA sequencing is not 100% correct due to base call errors. In order to improve further analysis of sequencing data, DNA sequencers determine an error probability for every base call. This error probability is stored in FASTQ files. Thereby, the error probabilities are converted into Phred Quality (Q_{Phred}) values. The name *Phred* originates from the identically named program *Phred* for base calling [EHWG98]. *Phred* uses log-transformed error probabilities [EG98], in order to support working with error rates that are very close to 0 as these are the most important base calls. The transformed base call quality is defined as

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

where P is the estimated probability that the according base call is wrong. Thus, a error probability of 0.001 is expressed by a Q_{Phred} value of 30. In FASTQ files, the Q_{Phred} value is stored by an ASCII representation. In order to allow human readable signs that start at 32(=Space) and end with 126(=]) an offset is added on the Q_{Phred} value [CFG+10]. In FASTQ files this offset is 33, thus, Q_{Phred} values from 0 to 93 can be encoded. Because this conversion is not standardized, different sequencers use different quality value conversions [CFG+10]. This circumstance must be considered when comparing data from different sequencers.

In Listing 2.2, we show an example file in FASTQ format with two sequences. In FASTQ format every four lines represent one sequence and its quality values. A line starting with "@" signals a new sequence and contains a description of the sequence in the next line. A line starting with "+" signals the end of the sequence and contains a description. The quality values are stored in the next line.

2.1.3.3 SAM Format

The SAM format is a text-based format to store sequence alignments [LHW+09]. A SAM file consists of a header section containing meta data about the reference and read sequences and used programs to generate the SAM file. The header section is followed by the alignment section that contains information about the aligned reads. Lines in the header section start with an "@" followed by certain character combinations to specify the header line type. In the following, we briefly describe the usage and content of each header line type.

Header Line (@HD)
Defines meta information about the document such as format version and whether the aligned reads in the alignment section are sorted.

Reference Sequence Dictionary (@SQ)
Every read is aligned to a certain reference sequence. Information about this reference sequence is stored here.

Read Group (@RG)
A read group is used to describe meta information about reads such as the sequencing center and the generation date.

Program (@PG)
In a program line, information about tools can be stored that were used to generate the SAM file or to preprocess the reads inclusive command line options.

Comment (@CO)
A comment line can be used for comments.

However, all these header information are optional. Multiple lines of the same header line type are possible, except the initial Header Line.

In the alignment section, every line describes the alignment of exactly one read to a reference sequence. Thereby, one line consists of 11 mandatory fields that must be always present. Furthermore, certain optional fields can be appended. These fields are tab delimited. In the following, we briefly describe the mandatory fields.

QNAME is a describing name for the read.

FLAG stores further information about the alignment process encoded as a bit mask.

RNAME contains the name of the reference sequence the read is mapped to. If a reference sequence dictionary line is present in the header section, further information about the reference sequence is available.

POS is the left-most (1-based) position, the read maps to the reference sequence. The first base in in the reference sequence has position 1.

MAPQ is the Phred scaled (cf. Section 2.1.3.2) mapping quality value.

CIGAR contains information about matching, deleted, inserted and not considered bases in the alignment of the read.

RNEXT is the reference sequence name the next read is mapped to.

PNEXT is the position the next read is mapped to.

TLEN is the number of mapped bases of the aligned read.

```
1 @HD VN:1.5
2 @SQ SN:ref LN:45
3 r001 0 ref 9  30 3S6M2I4M * 0 0 AAAAGATAAGGATA ))%%%++)(%%%%)
4 r002 0 ref 9  30 5S6M      * 0 0 GCCTAAGCTAA     ''*((((***+
5 r003 0 ref 16 30 6M14N5M   * 0 0 ATAGCTTCAGC    >>CCCCCCC65
```

Listing 2.3: Aligned reads in SAM format (adapted from [SAM13])

SEQ is the base sequence of the mapped read.

QUAL is the ASCII representation of the base call quality values of the read bases.

These fields are mandatory, but if a certain information is not available, this is expressed with a "*" or "0" depending on the field. For example, a "0" in the field POS means that the read is not aligned. This is only an overview about the information contained in a SAM file. For further information about this file format, the interested reader is referred to [SAM13].

In Listing 2.3, we show an example SAM file. The header section comprises two lines. The first line states that the file format is 1.5 (current version). The second line describes the reference sequence dictionary that comprises only one reference sequence, because only one "@SQ" line is present. In line three to five read alignments are described. For our following explanations, we use the term column to refer to the single fields a line contains in the alignment section. Thereby, column 1 refers to the QNAME field, column 2 to the FLAG field and so forth. The first two reads ("r001" and "r002" (column 1)) are both aligned to position 9 (column 4) of the reference sequence. Both reads are mapped with a Phred mapping quality of 30 (column 5). But, when looking at the base sequence of both reads (column 10), the sequences differ much and should not be aligned at the same position. Thus, is the alignment probably wrong? And if yes which one? The answer is that the alignment is correct (referred to this example without further information). The key to success is the CIGAR string in column 6. The CIGAR string encodes how the bases in column 10 must be arranged in order to get an alignment. A CIGAR string consists of pairs of numbers and characters (e.g., 3S6M1P1I4M). The character defines an operation over the number of bases. Possible operations are:

- M ... matches reference position
- I ... inserted at reference position
- D ... deleted regarding reference
- S ... soft clipping means starting bases does not map but are contained in the SAM file, but the position skips these bases
- H ... clipping bases are completely deleted

- P ... padding - soft deletion means reference got a padding
- N ... skipped region from the reference - hints to an intron when mapping mRNA to genome, otherwise not defined

For example, the CIGAR string (column 6) of read "r002" in Listing 2.3 states that the first 5 bases are soft clipped and the next 6 bases match. Thus, the aligned sequence at position 9 (column 4) is not "GCCTAAGCTAA" but "AGCTAA". Repeating this procedure on read "r001", the aligned sequence is not "AAAAGATAAGGATA" but "AGATAAGGATA". Considering these two sequences, it is more clear why they are mapped at the same position, although they differ at the third site, where "r002" has a "C" and "r001" has a "A". This could be due to a base call error, a real mutation or just due to wrong alignment. This question must be answered by a variant calling tool.

2.2 Database Systems

In this section, we explain database concepts and technologies that are important for further understanding of this thesis. Therefore, first, we introduce basic concepts in database research and then, we explain basic components of a database management system. Next, we describe different implementation techniques of database management systems and compare modern and traditional database management systems.

2.2.1 Basic Concepts

In this section, we explain basic concepts in the field of database systems.

2.2.1.1 Database System Approach

In this section, we explain the basic structure and idea behind database systems.

Database systems (DBSs) were developed and introduced to manage the partly concurrent access of many users on huge amounts of data in a robust and efficient manner (cf. [Ram98, p. 4] and [SKS97, p. 1]). Furthermore, DBSs were developed to overcome data management weaknesses of file systems (cf. [Ram98, p. 4] and [SKS97, pp. 2-4]) such as limited data security and integrity.

In Figure 2.2, we depict the structure of a DBS. A DBS consists of a database management system (DBMS) and a database. A database is a structured collection of related data such as an address book (database) with names, ages, addresses, and telephone numbers of persons (data). A DBMS is a set of software components and tools to manage a database. For example, a DBMS allows to query a database of persons for a specific telephone number and guarantees that the age of persons is never negative. To get access to data inside the database, the DBMS provides well-defined interfaces, which applications and users can use to query the database. Thus, all access to the database is controlled by the DBMS. Furthermore, details about the internal data representation are hidden from applications and users.

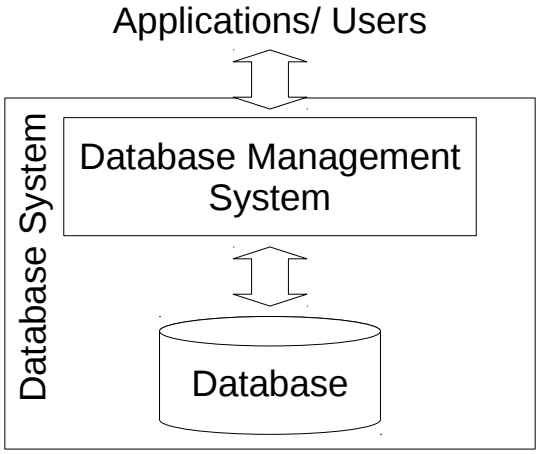


Figure 2.2: Structure of database systems (derived from Elmasri and Navathe [EN10, Figure 1.1])

2.2.1.2 Features of Database Management Systems

In this section, we describe basic features a DBMSs should provide for robust, efficient and effective data management. The following features are derived from Codd [Cod82]:

Integration A DBMS enables integration of data from different sources into one integrated data schema.

Operations A DBMS provides a defined set of operations to query, insert, delete and update data records.

Data Dictionary A data dictionary stores meta data about the stored data such as data schema or value domains of certain data attributes that are used to ensure data integrity.

User Views A view enables to limit the accessible data to an application. Thus, for example, only relevant or unclassified data is made available to an application or user.

Consistency A DBMS is responsible to keep data consistent and guarantee data integrity while data operations are performed.

Access Control A DBMS prevents unauthorized data access.

Transactions A transaction is a sequence of data operations. If all operations are successful, the result is stored permanently in the database. If at least one operation fails, all operations are reverted. Thus, a DBMS guarantees that the database is in consistent state in any case.

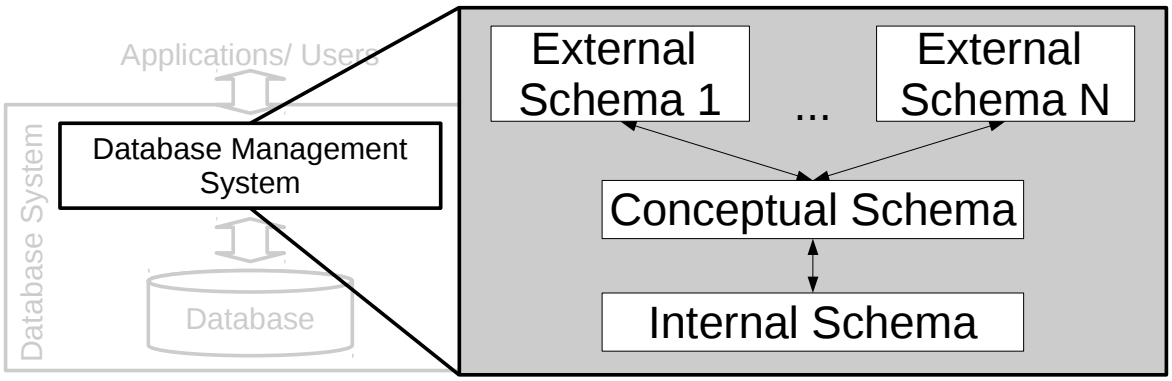


Figure 2.3: Three schema architecture (derived from Elmasri and Navathe [EN10, p. 34])

Synchronization A DBS is used by multiple users. All users can initiate transactions. Such transactions can be performed on the same data items at the same time, which leads to write or read conflicts. Thus, a DBMS cares about avoiding concurrent data access.

Recovery A DBMS provides means to recover data in a database after system failures or other events that lead to data loss.

Beside these features, a DBMS provides data independence by using a three schema architecture. In Figure 2.3, we show the three schema architecture. Every schema describes a level of data abstraction.

External Schemas An external schema describes the view one application has on the data. There can be different external schemas for different applications and users.

Conceptual Schema The conceptual schema describes the logical structure of the whole database. Thereby, available data entities and constraints are defined.

Internal Schema The internal schema describes how the data is physically stored at a storage medium.

Two types of data independence are distinguished:

Logical Data Independence allows to change the conceptual schema without or at least less effect on the external schemas. Thus, the conceptual schema can be extended (e.g., to integrate a new data source), but existing external schemas do not have to be changed or at least slightly in case of incompatible changes.

Physical Data Independence allows to change the internal schema without changing the conceptual and also external schemas. Thus, the physical data store can be optimized or reorganized without attention to the logical structure of the database.

The effect of data independence is a decoupling of applications from the internal data representations. Thus, data independence facilitates application development, because application developers do not have to care about efficient access to data as they just query the database via well-defined interfaces (e.g., SQL) and the DBMS handles efficient query processing and data management tasks.

2.2.1.3 Database Models

In this section, we describe two models that are important when designing a database application. We describe the entity-relationship (ER) model proposed by Chen [Che76] and the relational model proposed by Codd [Cod70]. Database models are important to design and implement a database application as they describe the data model of a database application. Thereby, two different model types are distinguished: conceptual and logical models. A conceptual model is used to describe the database schema at a high level. A logical model describes more implementation details of the database schema.

The entity-relationship (ER) model is a conceptual model. Our following explanations base on Elmasri and Navathe [EN10, Chapter 7]. An ER schema describes the data of a database application as entities and their corresponding relationships. Thereby, entities represent real world "things" in the database application context such as persons or cars. Every entity belongs to an entity type that has a describing name, describes the attributes an entity of this entity type has and is used to express the relationships between entities of different entity types. An example application is a student database, where students can attend courses. The entities in this example database are the students and courses. Appropriate entity types are student and course. The student entity type has the attributes id, name, age and city and the course entity type has the attribute name. Thereby, the id of the student entity type and the name of the course entity type are unique for every entity. Between entities of type student and course exists following relationship: a student can attend courses, vice versa a course is attended by students. This is called a *many-to-many* relationship. Other forms of relationships are *one-to-one*, *one-to-many* and *many-to-one* relationships. In Figure 2.4, we show the ER diagram of the previously described ER schema. Entity types are depicted as rectangles, their attributes as ellipses (unique attributes are underlined) and the relationships between entities are depicted as diamonds. In order to express the type of relationship between entity types, cardinalities are used that describe the minimum and maximum occurrence of an entity in a relationship. Different notations exist to express cardinalities. We use the min/max-notation: [min, max] (e.g., [0, N]). In our example, zero to many entities of type student are related to zero to many entities of

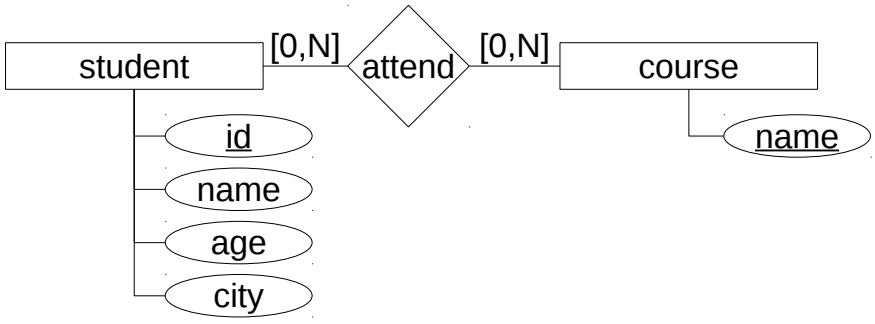


Figure 2.4: Example ER schema

type course. Thus, we explicitly allow empty courses and students that do not attend any course.

The next step after conceptual modeling is to create a logical database schema. Examples of logical database models are hierarchical, object-oriented, and relational model. Below, we explain details of the relational database model as it is the most widely used logical database model to provide implementation details of a database schema. Our following explanations base on Elmasri and Navathe [EN10, Chapter 3]. The relational model comprises relations, tuples and relation schemas. Thereby, a relation schema is a set of attributes that have a name and a data type. A tuple is a set of attribute values that match a relation schema and a relation is a set of tuples that all match the same relation schema. Moreover, the relational model provides the concepts of *primary keys* and *referential integrity*. A *primary key* is an attribute or set of attributes that is unique for each tuple of a relation. The concept of *referential integrity* expresses a dependency between two tuples of different or same relation. Thereby, the tuple that is referred by another tuple must exist. It is used to express relationships between different relations. The relational schema of the example application above is depicted in Figure 2.5. The relational schema consists of three relations. These relations are *student* and *course* with their respective attributes, whereby the unique attributes in the ER schema were transformed into *primary keys*. These *primary keys* are then referenced in the third relation *student-attends-course* to express the relationship between students and courses. Thereby, the referenced *primary keys* are used as *primary key* for this relation to avoid that a student attends a course twice. We derived this relational schema from the previously introduced ER schema. For more information on ER to relational schema mapping, we refer to Elmasri and Navathe [EN10, Chapter 9]. Besides the structural components, the relational model also describes a relational algebra. The relational algebra is a set of operations over relations. Every operation uses one or more relations as input and creates one relation as output. Important operations are *selection*, *projection* and *θ-join*. A *selection* has one relation as input and returns all tuples of these relation that match a specific predicate. For example, to retrieve all students that are older than 26, we have to do a *selection* on relation *student* with the predicate $age > 26$. A *projection* has one relation as input and selects defined

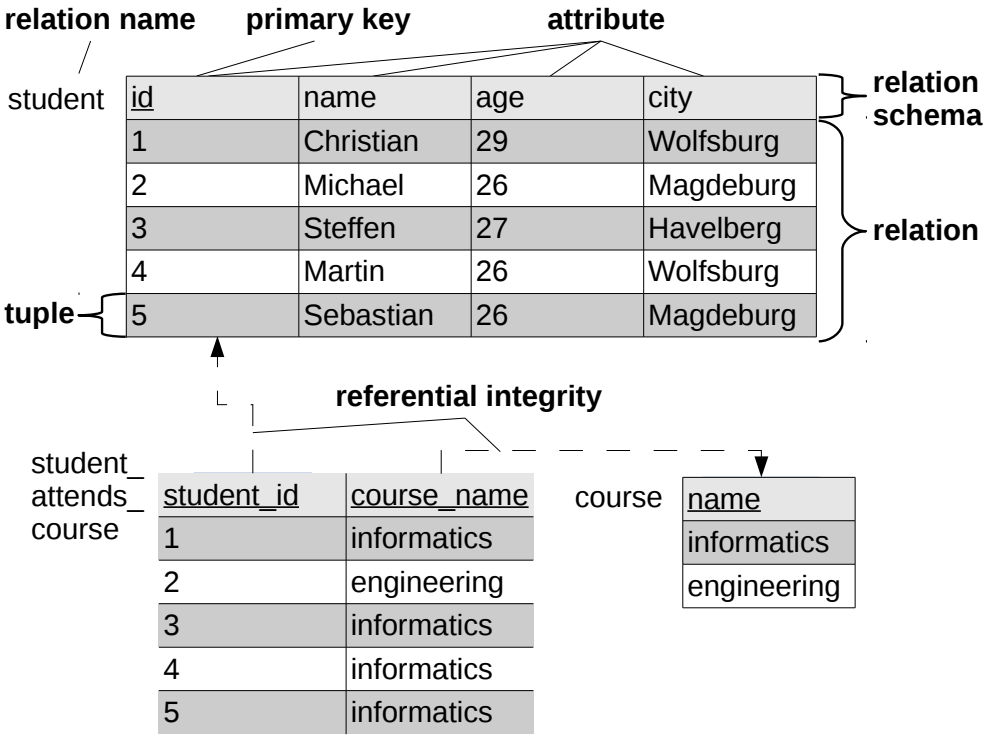


Figure 2.5: Example relational schema

attributes from all tuples of the input relation. For example, to retrieve only students' names, we have to do a *projection* on relation *student* with the argument *name*. A *θ-join* has two input relations and combines them regarding a defined predicate over the attributes of both input relations such as two attributes are equal or unequal. For example, to retrieve which courses students attend, we do a *θ-join* on relation *student* and *student_attends_course* with predicate $id = student_id$.

This is a rough overview about the ER model and the relational model, for more information the interested reader is referred to Elmasri and Navathe [EN10] or Saake et al. [SSH10]

2.2.1.4 Structured Query Language

In this section, we describe basic features of the structured query language (SQL).

SQL is a query language for relational DBMSs that allows *a*) to define relational database schemas; *b*) to manipulate (i.e., insert, delete and update) data in the database; *c*) to manage transactions and access control; and *d*) to query data in the database. First of all, SQL is an implementation of the relational algebra and thus it is used to query the relational database (d), but SQL also supports further syntax to provide the

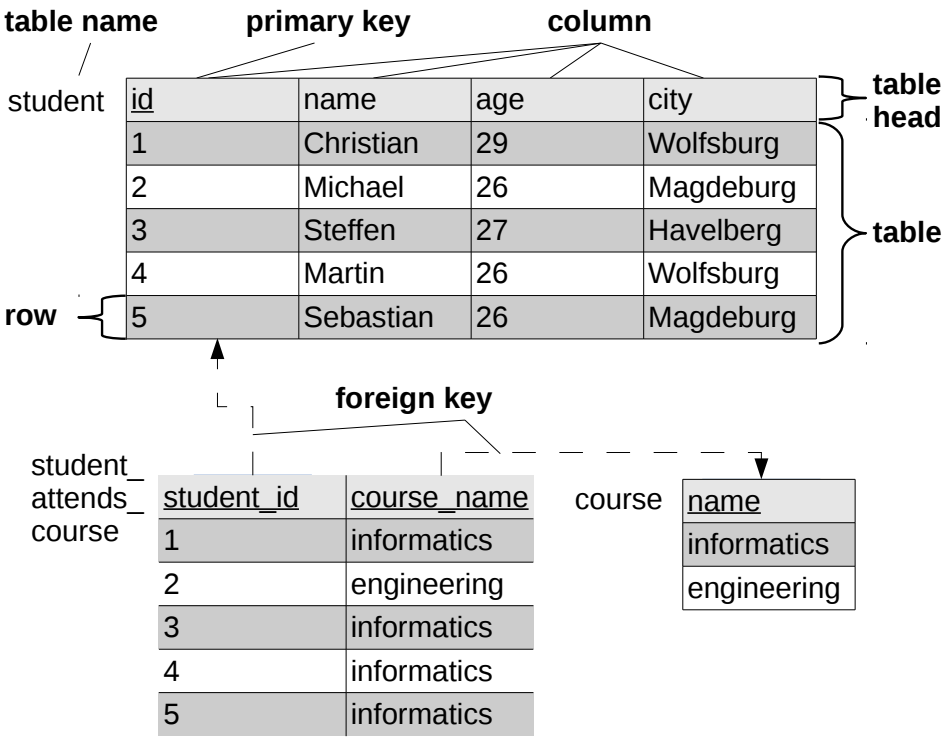


Figure 2.6: Example sql schema

features a) to c). In this section, we concentrate on defining (a) and querying (d) a relational database with SQL as it is important for our contribution.

SQL defines a slightly different data model as the original relational model. In Figure 2.6, we depict the relational schema from Figure 2.5 in terms of SQL. For example, a *tuple* is a called *row* and an *attribute* is called *column*. Besides the different terms, SQL does not support a set semantic such as the relational model, but a list semantic. Thus, the order of columns and rows is fix in SQL. The fixed order allows duplicate rows in the table and duplicate columns in the table head that are not allowed in relational algebra. Furthermore, SQL enables grouping and aggregation functionalities that are not provided by standard relational algebra [EN10, p. 165].

In Listing 2.4, we show the scaffold to create a table. Therefore, we have to use the *CREATE TABLE* command (line 2). A table consists of a table name that is used to reference the table to operate on it. Furthermore, a list of columns and their according data types has to be defined to specify the single values of each row in the table. The queries in line 4 and 6 are used to define integrity constraints that are offered by the relational model (cf. Section 2.2.1.3).

In Listing 2.5, we show the scaffold to query a relational database using SQL. A query to retrieve data always consists of a *SELECT-FROM-WHERE* block (lines 2 - 4). The

```
1 -- create table statement --
2 CREATE TABLE [table_name] ([attribute_list]);
3 -- create primary key
4 ALTER TABLE [table_name] ADD PRIMARY KEY ([attribute]);
5 -- create foreign key to guarantee referential integrity
6 ALTER TABLE [table_name] ADD FOREIGN KEY ([attribute])
7     REFERENCES [other_table_name] ([other_table_attribute]);
```

Listing 2.4: Queries to define tables

```
1 -- create table statement --
2 SELECT [column_names | aggregation_function]
3 FROM [table_expression]
4 WHERE [predicates]
5 -- additional functionality
6 GROUP BY [column_names]
7 ORDER BY [column_names]
8 HAVING [aggregation_function_predicate]
9 ;
```

Listing 2.5: SELECT-FROM-WHERE statement

SELECT defines which column values of rows in a table are returned. The table is referenced by its table name in the *FROM* part, whereby not only single tables can be queried, but also joined tables by using the keyword *JOIN*. The *WHERE* part defines predicates on values in a row that must be fulfilled in order to return the selected column values of that row . Using these basic constructs, all features of relational algebra can be expressed. Furthermore, SQL allows additional features such as grouping of data values and sorting (lines 6 - 8). In order to group or sort the resulting rows of a query, the keywords *GROUP BY* and *ORDER BY* followed by a list of column names are used, respectively. When grouping is used, column values that should be returned, but are not used for grouping, must be aggregated with a special aggregation function. Therefore, SQL defines five standard aggregation functions: *sum*, *max*, *min*, *count* and *avg*. Otherwise, the column values cannot be returned deterministic as the query processor does not know how to handle the column values in a group. Furthermore, it is possible to filter groups by using an aggregation function and a predicate such as *count(column)* > 1 to filter all groups that consist of more than one element.

2.2.1.5 Database Application Types

In this section, we characterize the two main database application types: online transaction processing (OLTP) and online analytical processing (OLAP) applications (cf. [EN10, p. 1068]).

Online transaction processing (OLTP) applications OLTP applications are database applications for operational usage such as inventory management or banking. Such applications operate on current, operational data and thus historic data is

Requirement	OLAP	OLTP
Number of users	few	many
Data records per query	many	few
Operations	read, bulk insert	read, delete, update, insert
Optimization goal	response time	throughput

Table 2.2: Comparison of OLTP and OLAP requirements (derived from Kifer et al. [KBL06, p. 712])

deleted or archived, but is not available for querying. Furthermore, OLTP applications are characterized by accessing and processing only few but complete data records per query. Such queries comprise simple reading, but also updating, inserting or deleting of data records. Because of the operational background of OLTP applications, many users access the database concurrently. Thus, high throughput is needed to serve all users. Furthermore, due to possible concurrent read and write operations, transaction management is important.

Online analytical processing (OLAP) applications OLAP applications are database applications for decision support such as market basket analysis. These applications require access to historic data in order to make time line analysis such as trend analysis. Furthermore, queries in OLAP applications are mainly long running read operations on many data records. Thereby, data is consolidated and aggregated regarding specific analysis needs. Moreover, new data is periodically added to the database. Thereby, not only a few records are added, but many from different data sources. Thus, fast response times are needed for querying the data as well as for importing new data. Because decision support applications are only used by a few users and mainly in read only mode, transactional concepts are less important.

In Table 2.2, we summarize requirements of both application types. One DBS can hardly satisfy the requirements of both application types. For example, depending on the transaction synchronization strategy, a long running OLAP operation locks huge parts of the database, because it processes a lot of data records. Thus, the operational business is strongly influenced, because OLTP queries such as inserts or updates have to wait until OLAP operations are finished. In order to avoid conflicts, the concept of data warehousing was introduced, where data for OLAP applications is integrated into a separate DBMS, the data warehouse. Thus, it is possible to use DBMS implementations that are specialized regarding the different requirements of OLAP and OLTP applications such as query response time or throughput.

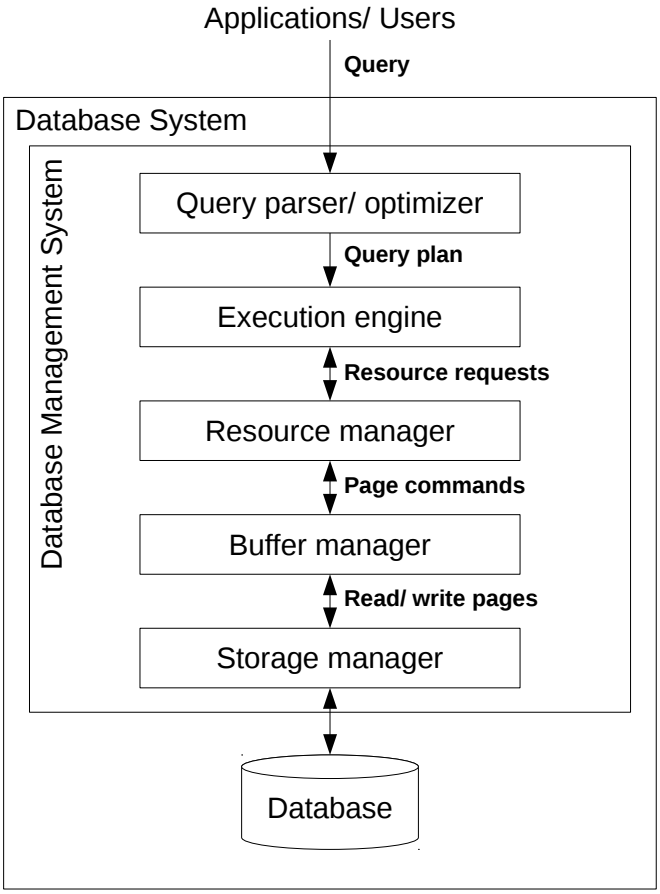


Figure 2.7: Database management components (derived from Garcia-Molina et al. [GMUW00, p. 7])

2.2.2 Components of Database Management Systems

In this section, we describe basic components for implementing a database management system.

Basic components of a DBMS are **query parser and optimizer**, **execution engine**, **resource manager**, **buffer manager** and **storage manager**. In Figure 2.7, we show how these components interact with each other. Below, we describe the single components and their interaction. As long as not explicitly given, our descriptions base on the explanations of Garcia-Molina et al. [GMUW00, p. 8].

In order to retrieve data from a DBS, the application or user has to forward a query to the DBMS. This query is parsed by the **query parser and optimizer** component. Result of this parsing is an optimized **query plan** consisting of several steps to retrieve the requested data. This query plan is passed to the **execution engine** that executes the plan and handles transactions. Thereby, the execution engine initiates a sequence of requests to retrieve the requested data according to the query plan. These requests

hit the **resource manager** that knows about what data records are stored in what data files and in which format. Furthermore, the resource manager knows about index files that are used to speed up data access. With this knowledge, data files that contain requested data can be accessed or data can be directly extracted from indexes. Only data that resides in main memory can be processed. Because main memory size is much smaller than secondary disk storage size, data resides permanently on disk and only the needed data for processing resides in main memory. Therefore, the resource manager requests the needed data from the **buffer manager**. The buffer manager is responsible to buffer data in main memory that is needed for processing. Thereby, the buffer manager does not handle just single records of data but pages that contain more than one data record. If the requested data is found in a buffered page, the buffer manager returns the requested data. Otherwise, the buffer manager initiates the read of missing pages that contain the requested data from disk into main memory via the **storage manager**. Thereby, existing pages can be replaced. Because replaced pages could have changed, while they resided in main memory, the pages must be written back to disk. Thus, the buffer manager also initiates the write back of replaced pages.

This is just a rough overview of DBMS components. For further information about query and transaction processing, indexes and the collaboration of DBMS components, the interested reader is referred to Garcia-Molina et al. [GMUW00] and Ramakrishnan [Ram98].

2.2.3 Implementation Techniques of Database Management Systems

In this section, we compare different implementation techniques for some DBMS components that highly influence DBMS characteristics. First, we consider DBMSs that store data permanently either in main memory or on disks. Then, we compare column- and row-oriented approaches to store data. Finally, we explain differences between bulk data processing and iterator-like data processing. Thereby, we describe advantages and disadvantages of each technique.

2.2.3.1 In-memory versus Disk-based Data Storage

In this section, we compare in-memory DBMSs with disk-based DBMSs. Thereby, we first introduce the disk-based approach and then describe the in-memory approach. Moreover, we name advantages and disadvantages of both approaches.

In a disk-based DBMS, all data is permanently stored on disks. Because disks are cheap and big, disk-based DBMSs can be used to store huge amounts of data at reasonable costs. As pointed out in Section 2.2.2, data must be available in main memory in order to process it. Because disk access is slow, sophisticated buffer managers are needed that minimize disk accesses by caching data in main memory. Furthermore, the storage manager must efficiently transfer data from and to disks. For small amounts of data, this buffer approaches are sufficient, but when huge amounts of data are required disk access occurs still frequently [PZ11, p. 13] and slows down query performance. The

Action	Time
L1 cache reference	1 ns
Main memory reference	100 ns
SSD random read	16,000 ns
Disk seek	4 ms = 4,000,000 ns

Table 2.3: Latencies for different actions on storage devices (derived from [Sco12]

obvious approach to overcome high access latencies of disks during query processing is to store all data permanently in main memory. Thus, in a in-memory DBMS, all data of the database or at least hot data, data that is frequently used, resides always in main memory [GMS92]. In Table 2.3, we show latencies for data access and seek on different storage devices. Accessing data in main memory is 40,000 times faster than seeking it on disk. Such an in-memory DBMS provides faster data access than disk-based DBMSs due to lower access times of main memory, but also comes with certain drawbacks. First drawback is limited memory size. Although costs per GB of disk and main memory storage decrease rapidly, costs per GB of main memory are much higher than costs per GB of disk [PZ11, Figure 1.2]. Thus, good compression techniques are needed in order to store more (compressed) data in-memory. At the same time, these compression techniques must allow fast decompression of data or processing of compressed data. Otherwise, compression and decompression would wreck the performance benefits of in-memory storage. Second disadvantage is persistence. Because main memory is volatile, system shutdowns lead to data loss. Thus, mechanisms are needed to recover data from non-volatile storage mediums such as disks. Thereby, mirroring the data that is hold in-memory is one option; another would be to store a log of data manipulation queries that is rerun after a system shutdown. Furthermore, storing data in-memory reveals a new bottleneck, the main memory itself, because CPUs can process data much faster than main memory can provide it [MBK00], [PZ11, p. 17]. In disk-based DBMS, this bottleneck is hidden due to high disk access times. Solution to this drawback is to efficiently use central processing unit (CPU) caches that are orders of magnitudes faster than main memory, but caches can "reduce memory latency only when requested data is found in the cache" [MBK00]. Furthermore, disk based techniques are not designed to support such fast devices [PFG+13]. Thus, Manegold et al. suggest to store data in the decomposed storage model proposed by Copeland et al. [CK85], because that facilitates cache-conscious data processing that weakens the drawback [MBK00]. We explain the theory behind the decomposed storage model in the next section.

According to our overview of DBMS components in Section 2.2.2, the decision for a in-memory or disk-based DBMS affects the implementation of the buffer and storage manager. In case of in-memory DBMSs, the buffer manager must handle all data in main memory. The storage manager is only needed for persistence purposes.

Person

Name	Age	City
Christian	29	Wolfsburg
Michael	26	Magdeburg
Steffen	27	Havelberg
Martin	26	Wolfsburg
Sebastian	26	Magdeburg

Figure 2.8: Example relation *person*

```
Christian|29|Wolfsburg|Michael|26
|Magdeburg|Steffen|27|Havelberg|M
artin|26|Wolfsburg|Sebastian|26|M
agdeburg|
```

(a) Row-oriented data layout

```
Christian|Michael|Steffen|Martin|
Sebastian|29|26|27|26|26|Wolfsbur
g|Magdeburg|Havelberg|Wolfsburg|M
agdeburg|
```

(b) Column-oriented data layout

Figure 2.9: Different data layouts of example data from Figure 2.8

2.2.3.2 Column-oriented versus Row-oriented Data Layout

In this section, we explain column-oriented data layout and compare it with row-oriented data layout. Therefore, we first explain the different data layouts and then state advantages and disadvantages of either data layout.

For our explanations, we refer to an example relation that we depict in Figure 2.8. We store five tuples of the relation *person* that has a *name*, *age* and *city* attribute. In a row-oriented data layout, all attributes of a tuple (row) are consecutively stored on a disk page and therefore in main memory. In Figure 2.9(a), we show the row-oriented data layout. In a column-oriented data layout proposed by Copeland and Khoshafian as decomposed storage model [CK85], the values of every attribute (column) are stored consecutively on a disk page. In Figure 2.9(b), we show the column-oriented data layout for our example data. In order to be able to restore a tuple, all attributes of one tuple must have a tuple identifier. The tuple identifier can either be stored explicitly or determined implicitly. In former case, the tuple identifier is physically stored together with the tuple's attribute at the cost of additional storage consumption. In contrast, the tuple identifier can also be implicitly determined via the records position [MBK00] such as in an array where every data item has an exact index.

The row-oriented data layout is better suited for operations that access complete tuples than the column-oriented data layout. Because in the row-oriented data layout all attribute values of one tuple are stored consecutively on the same page, all needed attributes for one tuple are available in main memory with one page load. In contrast, tuple reconstruction in a column-oriented data layout needs to load more pages from

disk as the attribute values of one tuple are not stored consecutively but scattered across different pages. In worst case, as many pages as the number of attributes of the relation must be loaded.

Another advantage of a row-oriented data layout is the lower effort for inserting new tuples than in a column-oriented data layout. In a row-oriented data layout, the attributes of the new tuple are simply appended to an existing page whereas in a column-oriented data layout, in worst case, each attribute value has to be stored on a different page. Thus, an insert operation in a row-oriented data layout need at most one page load whereas column-oriented data layouts require more page loads.

But the column-oriented data layout has advantages, too. When aggregating over one attribute of a relation, all needed data can be loaded more efficiently into main memory when it is organized in column-oriented data layout than in a row-oriented data layout. In a row-oriented data layout, all attributes of one tuple are stored consecutively on a page. Thus, when aggregating over just a one attribute, many pages must be loaded into main memory in order to collect all needed data for aggregation. The column-oriented data layout uses storage space more efficient in this case as all attribute values of interest are stored consecutively on a page, only those pages have to be loaded that contain needed attribute values. Furthermore, the characteristic that all values of one attribute of a relation are stored consecutively on a page can be leveraged for query processing. The query processing strategy is called late materialization, whereby the operators operate as long as possible only on needed columns and the materialization of tuples is done as late as possible. Thus, the amounts of data that are hold in main memory and are processed during query execution can be limited if appropriate. Late materialization becomes more attractive when data is bulk-processed, meaning that one operator processes chunks of data instead of just single attributes or tuples (cf. Section 2.2.3.3).

A further advantage of a column-oriented data layout is that it allows better data compression than a row-oriented data layout [AMF06]. Thereby, the characteristic that data of one data type is stored consecutively is important, because it is possible to use compression algorithms that are optimized for a certain data type [PZ11, p. 68]. Furthermore, it is possible to directly process such compressed data. In Figure 2.10, we show dictionary compression of attribute *city* from our example data from Figure 2.8. First, all distinct values of an attribute are collected. These values are then stored in a dictionary that allows to translate each value into an alternative and smaller value. In this example, we use the position of the value in the dictionary as replacement for the original values. Thus, *Wolfsburg* is replaced with *1*, *Magdeburg* with *2* and *Havelberg* with *3*. The replacements are then stored on the disk pages instead of the original values. In order to retrieve the original values, a lookup in the dictionary at the specific position is necessary. This compression technique allows direct processing of compressed values. For example, to retrieve all persons living in *Magdeburg*, the data is searched for tuples that have a *2* at the specific storage location. In Figure 2.11, we show run-length encoding of attribute *age* from our example data from Figure 2.8. The idea behind run-length encoding is to store repetitions of values by a shorter expression.

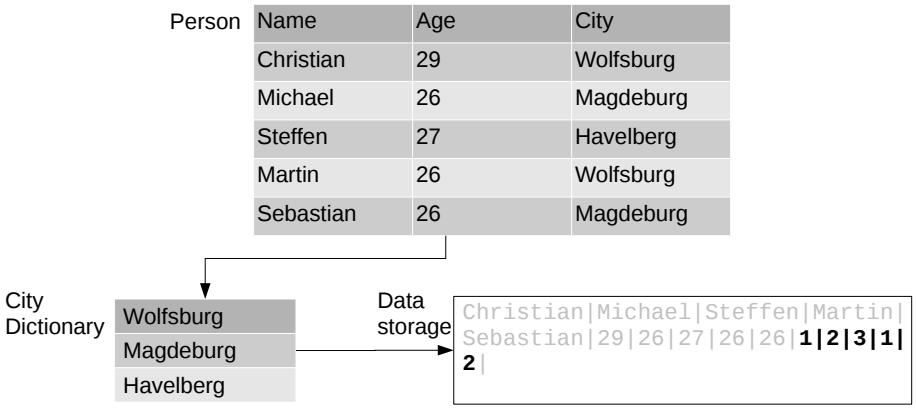


Figure 2.10: Dictionary compression of column *city*

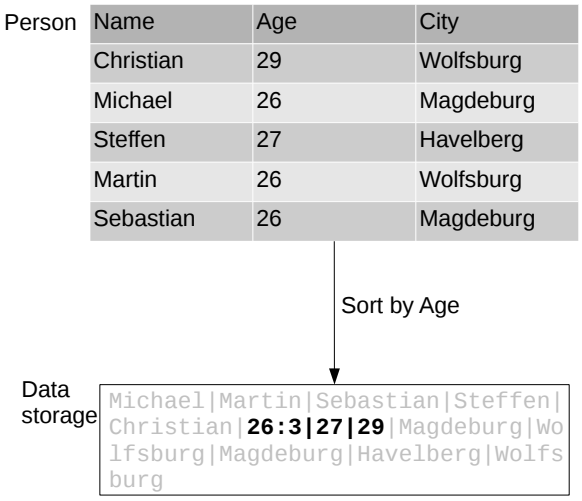


Figure 2.11: Run-length encoding of column *age*

This approach is easy to adapt for column-oriented data layouts, because values of the same data type are stored consecutively. When many consecutively stored values are equal, a replacement can be made. In the example, we compress the attribute column *age*. In order to improve the repetition of values, we sort all tuples by attribute *age*. Thus, we can replace the three values *26* by the shorter description *26:3* (i.e., the current and the next two values are *26*). If the value is not repeated, we just store one value as in case of *27* and *29*. Data that is compressed via run-length encoding can also be processed in the compressed way by considering the number of repetitions at processing time [PZ11, p. 70]. Because data in a column-oriented data layout can be compressed in a light-weight way that allows direct processing of data, it is well suited to be used in in-memory DBMSs (cf. Section 2.2.3.1). The compression capabilities allow to store more data in limited main memory without additional effort for decompression.

Thus, performance benefits due to main memory storage are not affected. More effective compression techniques are also possible, but these need more effort for compression and decompression [PZ11, p. 73].

Another advantage is that a column-oriented data layout improves cache-locality [Aba07], when access to values of single attributes is needed such as for aggregation functions. In a row-oriented data layout, not needed surrounding data values would waste cache space and lead to frequent cache misses that slow down query performance.

According to our overview of DBMS components in Section 2.2.2, the implementation of a row- or column-oriented data layout affects the execution engine and the resource manager. The resource manager implements whether data is stored in a row-oriented or column-oriented manner. The execution engine must consider data layout specific processing such as as the late materialization strategy.

2.2.3.3 Bulk versus Iterator-like Data Processing

In this section, we explain bulk data processing and compare it with iterator-like data processing. Thereby, we first introduce the iterator-like data processing approach and then describe the bulk data processing approach. Moreover, we name advantages and disadvantages of both approaches.

The iterator-like data processing or Volcano-style processing model [Gra94] processes one tuple at a time. Thus, it iterates over the tuples of a relation. Thereby, the processed data types and processing functions can be configured and are evaluated at runtime. This allows a high flexibility and parallelization of processing due to pipelining. But, this procedure is not very CPU efficient [PFG+13], because of indirections of data and functions that must be resolved at runtime. In case of disk-based DBMSs, the CPU inefficiency is hidden by the high latencies of disks. But, in in-memory DBMSs, this inefficiency limits the expected speed up dueto low main memory latencies. To improve CPU efficiency bulk processing was proposed [KPvdB92]. Thereby, one operator processes all data at a time and passes the result to the next operator. For disk-based DBMSs, this approach is inappropriate as huge intermediate results must be materialized in main memory. In in-memory DBMSs, data is already in main memory, thus, overhead is limited. Bulk data processing is facilitated by column-oriented data layouts [AMH08]. Thus, in combination with column-oriented data layout, bulk processing algorithms can achieve a good cache and CPU utilization for OLAP workloads [PFG+13], where data of single columns is processed. The drawback of bulk processing is less flexibility as algorithms must be designed carefully.

According to our overview of DBMS components in Section 2.2.2, the implementation of bulk- or iterator-like data processing is realized in the execution engine.

2.2.4 Modern versus Traditional DBMS

In this section, we compare traditional and modern database management systems. Therefore, we match implementation techniques described above to traditional or modern DBMSs and describe the resulting characteristics.

Traditional DBMSs are disk-based and row-oriented DBMSs that prefer a iterator-like data processing. These DBMSs are well suited for OLTP applications, where access to and manipulation of single tuples is the most common task. These systems are capable to store huge amounts of data on disks and allow efficient access to it. When processing huge amounts of data per query as in OLAP applications, traditional DBMSs reach their limits regarding response time. In order to speed up such analysis, modern DBMSs are used that use main memory technology and column-oriented data layout to speed up OLAP applications. Modern DBMSs are well suited for processing huge amounts of data per query fast. Therefore, modern DBMSs use cache-conscious data layout in combination with CPU efficient algorithms. But column-oriented DBMSs do not support OLTP applications well, because of higher efforts for tuple reconstruction and inserts [Aba07]. But, current research efforts in database systems aim at supporting OLAP as well as OLTP applications by modern DBMSs [Pla09]. Different solutions are proposed to reach the goal of performing OLTP and OLAP applications on the same database. SAP HANA, for example, allows the database administrator to create a table either in column-oriented or row-oriented data layout [SAP13, p. 151]. Kemper et al. proposed HyPer, an approach that uses "hardware-assisted replication mechanisms to maintain consistent snapshots of the transactional data" [KN11] to perform long running OLAP queries without interfering with the OLTP workload. Grund et al. proposed HyRise, a system that dynamically partitions tables vertically and thus, uses a partially decomposed storage model [GKP+10]. Pirk et al. use the partially decomposed storage model combined with just-in-time compiled queries to speed up query processing further [PFG+13]. In order to gain higher performance, all these approaches leverage modern hardware technologies and concepts such as cache- and CPU consciousness, main memory storage and vector-wise processing. Another upcoming technology in the field of databases are co-processors such as graphics processing units (GPUs) [BBR+13]. Besides finding suitable and efficient algorithms for GPUs, new optimizer strategies are needed to decide when to run an operation on CPU and when on GPU [BGS+12].

2.3 Summary

In this chapter, we explained important concepts of genome analysis and modern database systems that are necessary for understanding our further work. Regarding genome analysis, we introduced basic terms and explained the basic steps of genome analysis. Furthermore, we introduced common file and data formats used in genome analysis. We later on use these common flat files to generate our test data (cf. Section 6.2.2.3). Furthermore, we explained database foundations and described technologies used in modern database systems. Moreover, we described differences between traditional and modern database management systems and current research trends.

3. Why to use Modern Database Systems for Genome Analysis?

In this chapter, we answer our first research question: *How can modern database systems address challenges in genome analysis?* (RQ 1) Therefore, we first explain challenges in genome analysis. Then, we describe how modern database systems can help to overcome these challenges by using modern database technology for genome data management and processing. Finally, we summarize our findings.

3.1 Challenges in Genome Analysis

In this section, we present challenges in genome analysis that must be addressed and handled by genome analysis pipelines due to the introduction of next generation sequencing techniques and new use cases such as personalized medicine. Therefore, we discuss following challenges: *a*) increasing amounts of genome sequencing data; *b*) decreasing quality of genome sequencing data; *c*) structural and functional complexity of genome analysis; and *d*) personalized medicine. Moreover, we state requirements to overcome these challenges.

Increasing amounts of genome sequencing data

Due to the introduction of next generation sequencers for DNA sequencing, the amount of genome sequencing data that is generated in one hour is order of magnitudes higher than with classical sequencing techniques. Literally spoken, the next generation sequencers perform DNA sequencing in a highly parallel manner, therefore they are also called *massively parallel* sequencers. In Figure 3.1, we show the development in sequencing technologies regarding throughput and costs. Thereby, *Sanger 3730xl* is a representative of the classical sequencing approaches and the other systems belong to the category of next generation sequencers [LLL+12]. The numbers in brackets below

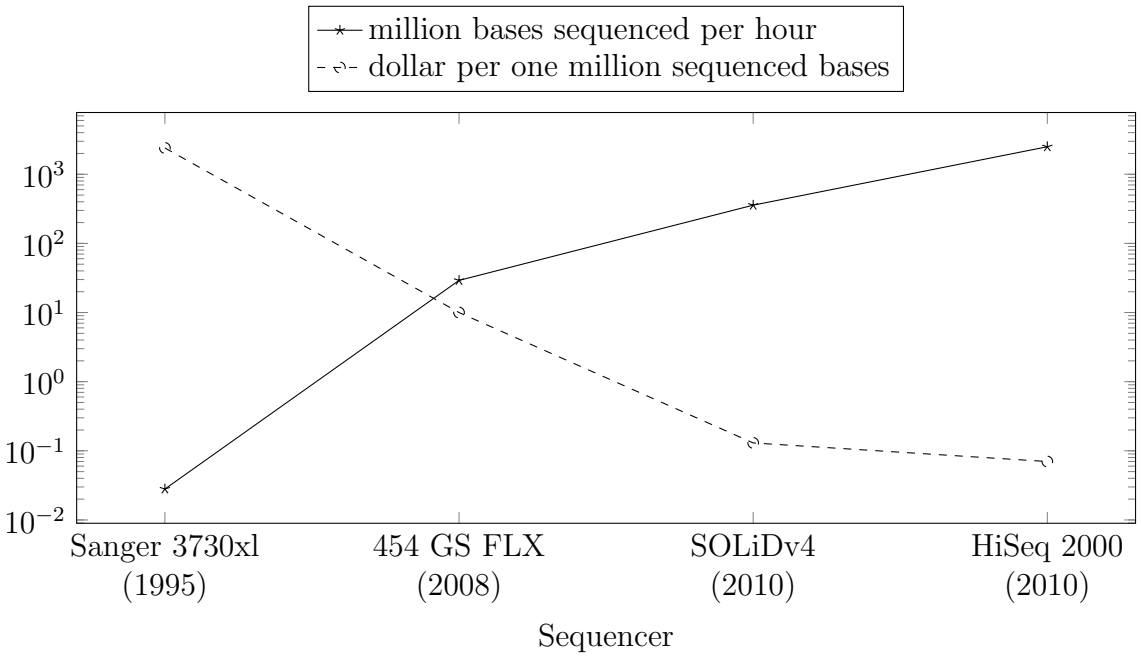


Figure 3.1: Sequencing throughput and costs by sequencer (derived from [LLL+12])

the sequencer names, state the year of introduction of the according sequencer. The solid graph shows the throughput of each sequencer in million bases per hour. The dashed graph shows the costs for sequencing one million bases with the according sequencer. The y axis is log scaled. Both graphs show a clear reverse trend. Thus, with every new sequencer generation, the throughput increases and the sequencing costs decrease.

The ability to sequence huge amounts of DNA at reasonable costs leads to an increase of storage consumption when storing the genome sequencing data on disk. Assuming a theoretical storage size per sequenced base of one byte, the output of the *HiSeq 2000* sequencer, for example, consumes 2.5 GB of storage per hour. Furthermore, these huge amounts of data must be processed in reasonable time by sequence alignment and variant calling tools. Thus, effective means for data compression and processing of compressed data are needed to address the challenge of increasing amounts of genome sequencing data.

Decreasing quality of genome sequencing data

After DNA is sequenced, the single reads are aligned with the help of computers to restore the original genome sequence (cf. Section 2.1.2.2). The time to process a sequence alignment depends on the length of reads, because it is more probable that shorter reads map to different sites at the reference genome than longer ones. Thus, the effort of mapping reads increases. Furthermore, alignment quality is affected by base call quality. A higher number of correctly called bases by the DNA sequencer improve the result quality of alignment as misalignments due to erroneous base calls are avoided.

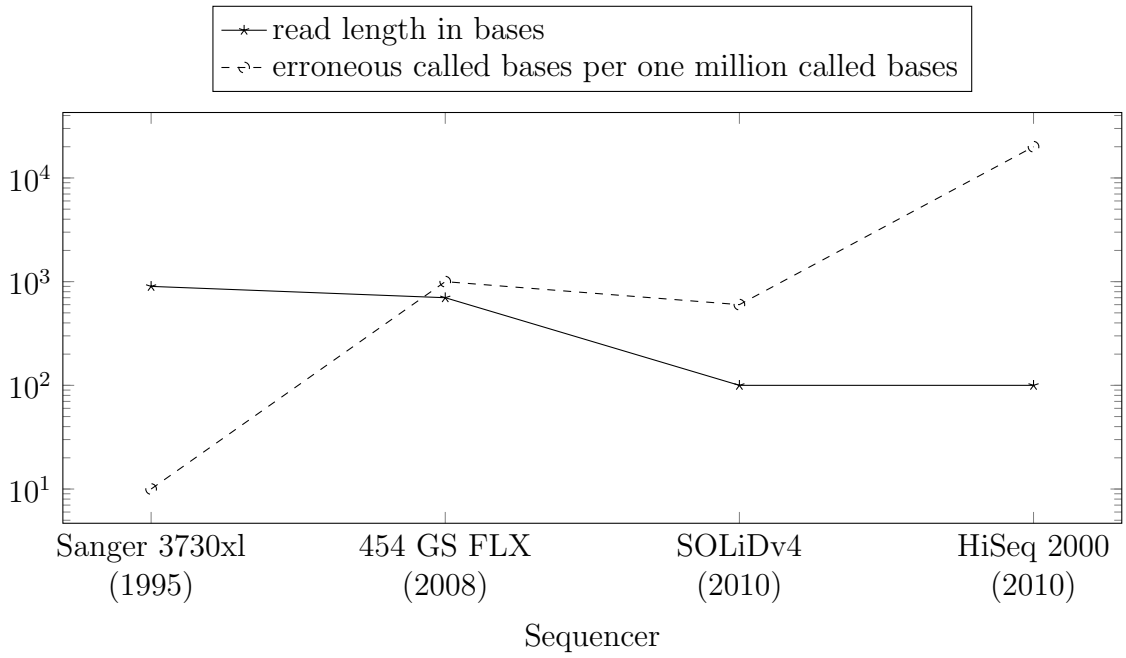


Figure 3.2: Read length and error rate by sequencer (derived from [LLL+12])

But, the high sequencing throughput of next generation sequencers comes at the cost of smaller read length and higher base call error rate. In Figure 3.2, we show the read length and the average number of erroneous called bases for the different sequencers from Figure 3.1. The solid graph shows the read length in bases. The dashed graph represents the number of erroneous called bases per one million called bases. The y-axis is log scaled. Newer sequencers such as *HiSeq2000* create smaller reads (100 base pairs) and produce more base call errors than classical sequencing techniques.

The lower quality of genome sequencing data generated by next generation sequencing techniques (i.e., smaller read length and higher base call error rate) requires algorithms that can handle errors, use heuristics to reduce the search space and work with probabilities (cf. Section 2.1.2.2 and Section 2.1.2.3). Thus, the results are not perfect and it becomes necessary to reprocess source genome sequencing data later on with different algorithms or under different parametrization of an algorithm in order to improve results. Furthermore, in order to validate results of downstream analysis, it also becomes necessary to be able to audit the source genome sequencing data at any time and to reprocess it. Thus, source genome sequencing data must be stored permanently for reprocessing and validating analysis results.

Structural and functional complexity of genome analysis

In order to reveal new insights by genome analysis it is absolutely necessary to integrate genome data with further data sources such as information about coding regions that encode genes and annotation information about gene functions [ZBY+11]. Then, the

integrated database must be accessible by the many people needed especially for downstream analysis tasks [Mar10]. Many databases exist that provide access to genome data and information [FSG13] that store data in different data structures. Moreover, the interpretation of genome data is not always consistent. For example, DNA sequencers encode base call error probabilities differently (cf. Section 2.1.3.2). Furthermore, algorithms to process genome sequencing data must be able to handle errors and uncertainties such as possible base call errors. Such algorithms are not trivial and provide complex functionalities.

In order to address the complexity of genome analysis, comprehensive data integration capabilities are needed that extract, transform, and load data into well-defined data structures that are interpreted consistently. Furthermore, data access interfaces for multiple users are needed to facilitate complex genome analysis work flows.

Personalized medicine

When thinking about personalized medicine, where genome analysis becomes daily business, data privacy must be accounted when patients want to keep control over their personal data, what can even be enforced by law [WRB+12]. Moreover, reliability and compliance of genome analysis become important to ensure correct and valid analysis results.

In order to address the data management challenges of personalized medicine, data security capabilities of genome analysis systems are necessary. Thereby, the systems must guaranteed that only authorized manipulation of data is allowed for authorized persons. Moreover, all interactions must be logged to keep overview about data manipulation and satisfy compliance requirements that arise to ensure that analysis results are reliable and comprehensible.

3.2 Addressing Challenges in Genome Analysis with Modern Database Systems

In this section, we explain how modern database systems can be beneficially used for genome analysis in order to address the challenges described in Section 3.1.

Increasing amounts of genome sequencing data

The increasing amounts of genome sequencing data require effective means for data compression and processing of compressed data. Modern database systems provide capabilities to compress data efficiently and allow processing of compressed data at the same time. For example, column-oriented database systems achieve better **data compression** rates than row-oriented ones due to the application of more efficient compression algorithms. In column-oriented database systems, all values of one column are adjacently stored in physical memory and have the same data type. This allows lightweight compression of column values (e.g., dictionary encoding or run-length encoding).

Furthermore, light-weight compression techniques allow direct processing of compressed data instead of decompressing it beforehand (cf. Section 2.2.3.2). Another idea to save storage space is to compute information instead of storing it. However, this requires that such information can be computed fast enough. For that reason, column-oriented **in-memory** database systems become interesting. In in-memory database systems, all data resides in main memory or at least the data that is needed for processing. This allows faster data access as in traditional database systems. In combination with data compression and the ability of **processing compressed data** directly, more data can be stored in main memory without processing overhead due to decompression. Moreover, column-oriented database systems allow a more cache efficient processing of single columns, because all values of one column are stored adjacently. They can also be loaded adjacently into CPU caches. Thus, queries and functions that compute the result over all values of one column (e.g., aggregation functions) would highly benefit from column-oriented storage. Another advantage of column-oriented storage is the ability for late-materialization where query processing is done as long as possible on a few columns instead of reconstructing tuples right at the beginning. This reduces the amount of data to process. Thus, column-oriented in-memory database systems are promising systems to increase the analysis performance.

Decreasing quality of genome sequencing data

The lower quality of genome sequencing data generated by next generation sequencing techniques (i.e., smaller read length and higher base call error rate) requires to store genome sequencing data permanently to ensure that analysis results can be validated and reprocessed later on. As modern database systems provide efficient **compression techniques**, additional storage consumption due to storing source genome sequencing data can be limited. The lower data quality also requires to process the genome sequencing data with different or improved algorithms for sequence alignment, variant calling, and downstream analysis. Modern database systems provide the concept of **data independence** that decouples the data from its internal representation [EN10, p. 35]. Thus, changes to the internal data format are latent and users can still access the data using the external schema via descriptive query languages (e.g., SQL). In contrast, flat file based approaches have to deal with the internal representation of data. Philippi and Köhler describe that using flat files always requires an appropriate parser in order to read the data in the specific format [PK06]. Furthermore, they notice that often format updates become problematic, because parsers are not adapted to the new format at the same time. Thus, access to data can be limited after format updates. They refer to the use of XML parsers because there are universal XML parser in nearly any programming language. But this would not free the users from implementing and updating the parser on their own. Thus, a standardized data access interface such as SQL could highly improve development of applications for genome analysis.

Structural and functional complexity of genome analysis

To address the complexity of genome analysis, modern database systems provide comprehensive **data integration** capabilities that enable the integration of data from differ-

ent data sources into an integrated database schema as needed for downstream analysis. Thereby, the database schema is defined in the **catalog** of a database management system and mandatory for all database operations. Furthermore, modern database systems provide efficient access to data by **multiple users** via **descriptive query languages** such as SQL. Thus, all people that take part in genome analysis have access to the same database and can access it via a defined set of operations, whereby the modern database systems handle efficient data access.

Personalized medicine

Data privacy capabilities are necessary when dealing with personalized genome information. Additionally, reliability, comprehensibility, and repeatability of analysis and analysis results must be guaranteed. Modern database systems provide **data security** mechanisms that restrict data access by user accounts. In contrast, flat file approaches for genome analysis rely on the ability of file systems to restrict access to data. Furthermore, database management systems allow fine-granular data access control on attribute level, whereas file systems are only capable to restrict access on whole files. Moreover, modern database systems provide **data integrity** features that ensure that data is modified only in a previously specified scope. For example, modern database systems provide integrity constraints such as domain constraints that limit the value domain of attributes. Thus, a database administrator can define and enforce that allowed values for a base attribute are A, C, G, T, or N. Moreover, database management systems provide integrated logging mechanisms that record data access and manipulation and thus **data provenance** can be ensured. Flat files allow optional documentation of data processing via comment fields (cf. Section 2.1.3.3), but do not enforce it.

3.3 Conclusion

In this chapter, we have shown important challenges in genome analysis and how modern database systems address them. Especially, the challenges regarding data integration and data security can be handled well by modern database systems as they provide built-in features to solve such requirements. Another advantage of using modern database systems is the decoupling of data access and application logic that supports faster application development. Thus, implementations of new methodologies using database management systems for genome analysis do not have to care about efficient data access. Moreover, the use of database systems for an application enforces a database schema that is well-defined and accessible by different tools via descriptive query languages such as SQL. Regarding storage consumption and analysis performance, we can only assume positive effects but cannot prove it without experiments. Thus, in the following chapters, we create a database schema to store and analyze genome data and evaluate this approach regarding storage consumption and analysis performance.

4. Related Work

In this section, we provide an overview of existing approaches that use database systems for genome analysis. We use this knowledge to identify starting points for our own research and to isolate our research from existing approaches.

A lot of systems exist that provide access to biological data [FSG13]. Not all systems use a database management system to access and manage their database (cf. Section 2.2.1.1), but manage the database themselves. The systems provide different types of biological data such as genome data, protein data, or ontology information. Furthermore, the possibilities to access the data differ from system to system. Cornell et al. categorize systems that provide biological data by their access capabilities [CPW+01]. They distinguish browsing, visualizing, querying, and analyzing capabilities of systems. In the following, we define these capabilities:

Browsing Browsing is the act of navigating through data sets (e.g., by following links).

Visualizing Visualizing functionality allows the presentation of genome data. A system that allows visualizing genome data is the UCSC Human Genome Browser [KSF+02]. Visualization is often related to browsing.

Querying Querying is the act of retrieving facts on demand from a database.

Analyzing Analyzing capabilities allow users to search for information within data. Therefore, data must be processed and interpreted.

We limit our search for related work on systems that use database management systems to manage their data and store genome data such as sequences and variant calls. Furthermore, as we search for approaches that support genome analysis, we search for approaches that provide at least querying and analyzing capabilities. Thus, we ignore systems that only allow browsing or visualizing genome data but use database systems to store their data.

Below, we list approaches for genome analysis using database systems and briefly describe them:

GIMS The Genome Information Management System (GIMS) uses an object database to integrate genome sequence data with functional data on the transcriptome and on protein-protein interactions [CPW+01]. The system provides graphical interfaces for querying the genome data. Furthermore, the approach provides canned queries via graphical interfaces that can be parameterized to perform analysis steps.

Atlas The Atlas system provides a relational database schema to integrate several different data sources for genome data [SHX+05]. The data can be directly accessed via SQL, which allows flexible access to data. Furthermore, application programming interfaces for C++, Java, and Perl exist that use SQL to access the data from within programs. Moreover, tools are provided that can be used for data import, export, and analysis.

BioWarehouse The BioWarehouse system integrates different databases into an integrated database schema and makes it accessible via SQL [LPW+06]. Furthermore, the system provides data loaders written in C and Java that extract genome data, transform it into the database schema, and load it into the database. Thereby, certain normalization tasks are performed to integrate the data from different databases.

dbBlast This approach integrates the local alignment search into a database management system by using stored procedures [RD06]. Thereby, special indexes are used to speed up the alignment processing.

BioDWH BioDWH is system that uses the Hibernate[1] framework to provide an object-relational mapping between database and programming entities [TBAR08]. The data is persisted in a relational database system and accessed via Java by using Hibernate.

SQL Server-based approach This approach provides an entity-relationship schema to store genome data in relational databases [RB09]. Thereby, the capabilities of SQL Server were used to integrate original flat files as file streams into the database and thus, make the data stored within flat files accessible via SQL. Furthermore, the use of custom aggregation functions is introduced to derive a consensus sequence from the aligned sequences.

CASTOR The Comprehensive Analysis and STORage (CASTOR) methodology provides a multi table database schema to store genotype and phenotype data. Thereby, the data is fragmented horizontally into several tables. According to Bouffard et al., this fragmentation improves data import speed and storage consumption of indexes [BPB+10].

Approach	Alignment	Variant Calling	Downstream Analysis
GIMS [CPW+01]			X
Atlas [SHX+05]			X
BioWarehouse [LPW+06]			X
dbBlast [RD06]	X		
BioDWH [TBAR08]			X
SQL Server-based approach [RB09]		(X)	X
CASTOR [BPB+10]			X

Table 4.1: Database approaches for genome analysis

In Table 4.1, we categorize the above approaches regarding supported steps of genome analysis (cf. Section 2.1.2). The majority of approaches for genome analysis on database systems use the obvious data integration advantage of database systems. Hence, database systems are mostly used for downstream analysis. These approaches use object database systems (e.g., [CPW+01]) or relational database systems (e.g., [SHX+05, LPW+06, TBAR08, RB09]). Furthermore, these systems provide tools to extract, transform, and load data into an integrated database schema. The computational intensive steps of genome analysis, sequence alignment and variant calling, are not addressed by these data warehouse like approaches. Instead, these approaches rely on upstream processing of sequence alignment and variant calling and integrate the result data. Therefore, complete genome sequences are stored inside the database and the information from sequence alignment and variant calling is integrated [SHX+05, LPW+06, RB09]. Such approaches have two disadvantages:

1. Flat files are still used at the beginning of the analysis process. Thus, the accompanying disadvantages of flat files, namely missing data integrity and security, remain in the first processing steps.

2. The additional storage of computable alignment and variant calling information increases the storage consumption.

Only two approaches by Röhm et al. cover at least small parts of sequence alignment and variant calling [RD06, RB09]. Röhm et al. implement a sequence alignment search on relational databases, but the results are not satisfying for read length greater 60 base pairs [RD06]. Furthermore, Röhm et al. show consensus calling via SQL using user defined functions [RB09]. As genome sequences are stored as strings, a user defined function is needed to split genome sequences on the fly in order to create a consensus

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

base from all bases at a specific genome site. The aggregation of the bases at one genome site is performed by a user defined aggregation function. Thereby, Röhm et al. point out that the parallelization of such user defined functions is problematic and thus the performance lacks.

Summary

The literature research shows that the most research effort concentrates on building integrated databases to support downstream analysis. Little has been done to support sequence alignment and variant calling with database systems efficiently. Furthermore, none of the above approaches uses modern database technologies such as in-memory storage or column-oriented data layouts. Thus, the potentials of modern database technology for genome analysis are not examined yet. The CASTOR approach shows that specialized data structures are beneficial to improve analysis performance [BPB+10]. Moreover, the approach by Röhm et al. shows that storing genome sequences as strings is related with additional overhead when processing single sites of a genome [RB09]. We keep this in mind when we design our database schema to support genome analysis with modern database systems.

5. Genome Analysis on Modern Database Systems

In this chapter, we introduce our concept for genome analysis on modern database systems. First, we analyze whether genome analysis in general is an applicable database application. Then, we present two different implementation approaches to support genome analysis with modern database systems, and therefore answer our second research question (RQ 2): *How to use modern database systems for genome analysis?* Based on these considerations, we develop a concept for variant calling on modern database systems and show possible analysis in the field of variant calling. Thereby, we propose a database schema to store genome data and thus give a first answer on our third research question (RQ 3): *How to store genome data in a modern database system?* Finally, we summarize our contribution.

5.1 Genome Analysis as Database Application

In this section, we analyze how genome analysis can be implemented as database application. First, we analyze the applicability of genome analysis as database application. Then, we present two different approaches to use modern database systems for genome analysis. Finally, we decide what steps of genome analysis we implement and evaluate on modern database systems in this thesis.

5.1.1 Application Characteristics

In this section, we evaluate application characteristics of genome analysis as database application. Therefore, we first explain criteria when not not use database systems and apply them to genome analysis. Moreover, we discuss whether genome analysis is more an OLTP or OLAP application.

The use of database management systems is always related with certain overhead costs [EN10, p. 26] such as system maintenance costs or additional processing costs (e.g., for integrity checks). Thus, Elmasri and Navathe define following criteria when not to use a database management system for an application [EN10, p. 26 - 27]:

- If the database application is **simple and well-defined** and not expected to change at all, because optimized applications outperform database management systems due to overhead costs.
- If the application requires **stringent, real-time requirements** that cannot be met because of DBMS related overhead.
- If the application runs on systems that have **limited resources** such as limited storage or processing capacity, where a general-purpose DBMS would consume all resources.
- If **no multiple-user support** is needed, database management systems' capabilities to handle concurrent data access are overhead.

Additionally, we have derived a further criterion regarding **general usability of data**. If data is only needed for one specific task and no other application or user can use it, it is not worth to take the database management overhead and storing it in a database system.

In the following, we evaluate genome analysis regarding the formerly introduced criteria when not to use a database management system.

Simple and well-defined application

In genome analysis quasi-standard file formats exist, especially for DNA sequencing data, sequence alignment data, and variant calling information. But, the tools to perform each of the genome analysis steps are not standardized. There are different tools for alignment, variant calling, and especially downstream analysis that furthermore generate different results. For example, O'Rawe et al. have shown that the results of different variant calling pipelines have only 60% overlap [OJS+13].

Limited resources

Genome analysis is not performed on machines with limited resources such as embedded devices, but on server hardware.

Stringent, real-time requirements

Against the backdrop of increasing amounts of available genome sequencing data that must be analyzed, genome analysis pipelines must be fast. But this requirement does not imply stringent, real-time processing.

No multiple-user support
Many people are involved in genome analysis, at least in the downstream analysis step. Thus, genome analysis needs multiple-user support.

General usability of data
In general, all data that is generated during genome analysis is useful for further analysis as can be seen by current database approaches.

With our analysis, we show that there is no concrete reason not to support genome analysis with database systems. The generality of data generated in each step of genome analysis for further analysis is given. Furthermore, real-time processing would be desirable but is no explicit requirement now. Moreover, multiple-user support is needed especially in downstream analysis. Multiple-user systems would also have the advantage to save storage by saving the genome data only once. All users that have access to the database system use the same database and the database management system handles data access. Finally, genome analysis on limited hardware systems such as embedded systems is no requirement at all, although resources such as disk storage or main memory can be limited.

Thus, using when database systems for genome analysis, the question arises whether genome analysis is more an OLTP or OLAP application. According to our characterization of these two different application types in Table 2.2 in Section 2.2.1.5, we classify genome analysis to be more an OLAP application than an OLTP application. We base our decision primarily on the characteristics of genome analysis regarding: *a*) the number of processed data records per query; *b*) the performed database operations; and *c*) the optimization goal. When considering sequence alignment and variant calling, we expect a high number of data records to process per query. Thereby, mainly read operations are performed and the results are delivered to the client. New data must be inserted via bulk-loading as the output of DNA sequencers comprises gigabytes of sequencing data. Regarding the optimization goal, we are undecided. On the one hand, response time is important to generate analysis results as fast as possible. On the other hand, throughput is also important as the number of users could be high. Thus, the use of modern database technology is appropriate as current research effort aims at supporting OLAP as well as OLTP workloads with the same database system (cf. Section 2.2.4).

To summarize, we have shown that all steps of genome analysis are appropriate applications for databases systems. Moreover, we have motivated the use of modern database systems due to the fact that genome analysis is no classical OLAP as well as OLTP application. Thus, genome analysis can benefit from current research efforts to support both application types with one modern database system.

5.1.2 Implementation Approaches

In this section, we explain two different approaches to support genome analysis with modern database systems. Thereby, we explain advantages, disadvantages, and requirements of each approach.

Analysis Support

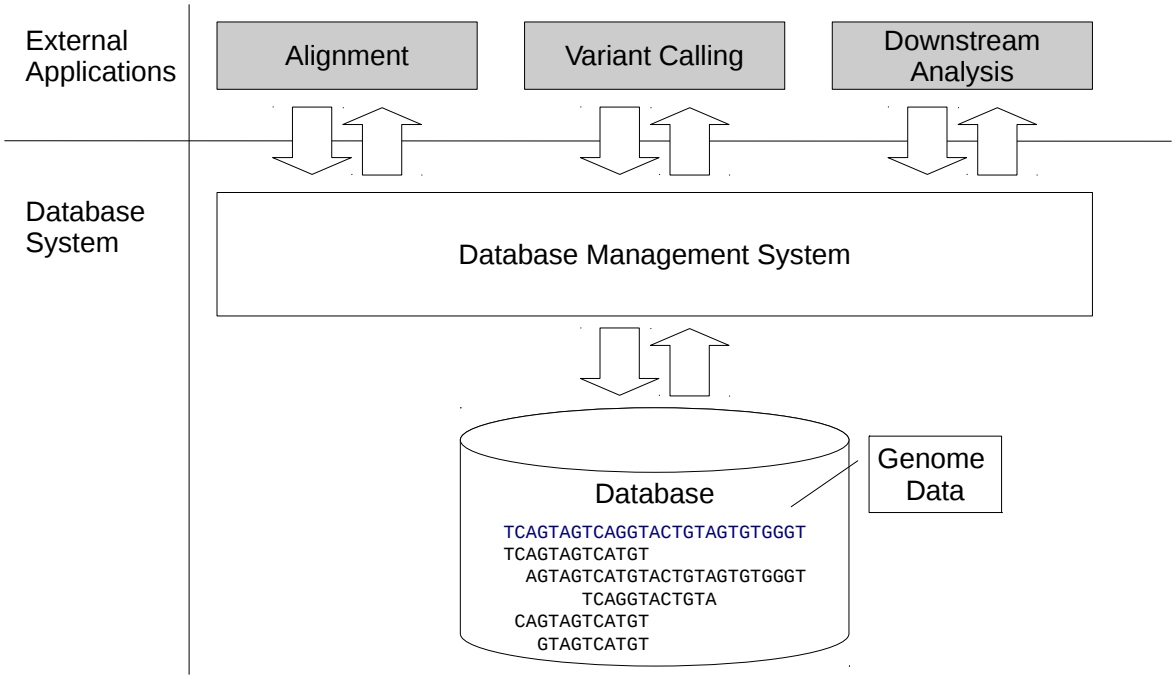


Figure 5.1: Analysis support

First, modern database systems can be used as storage solution and thus just to support genome analysis. In Figure 5.1, we illustrate this approach. All analysis steps of genome analysis are implemented as external applications. All genome data is stored in a database that is managed by a database management system. The external applications access the database system and the database management system is responsible for accessing the database. The external tools do not only query data, but also insert new data. Thus, the alignment and variant calling applications save the results into the database to make it available for downstream analysis.

One advantage of this approach is that the external applications do not have to care about efficient data access as this is handled by the database management system. Furthermore, data management quality increases for the complete genome analysis process, because data integrity and data security is guaranteed by the database management system. Moreover, all data is stored at a central place once. Thus, storage is saved, because all tools that have access to the database system, access the same data. A disadvantage is the need to store all intermediate results. In order to implement this approach, a suitable database schema must be developed. Furthermore, the data schema as well as the database management must allow fast access to the required data.

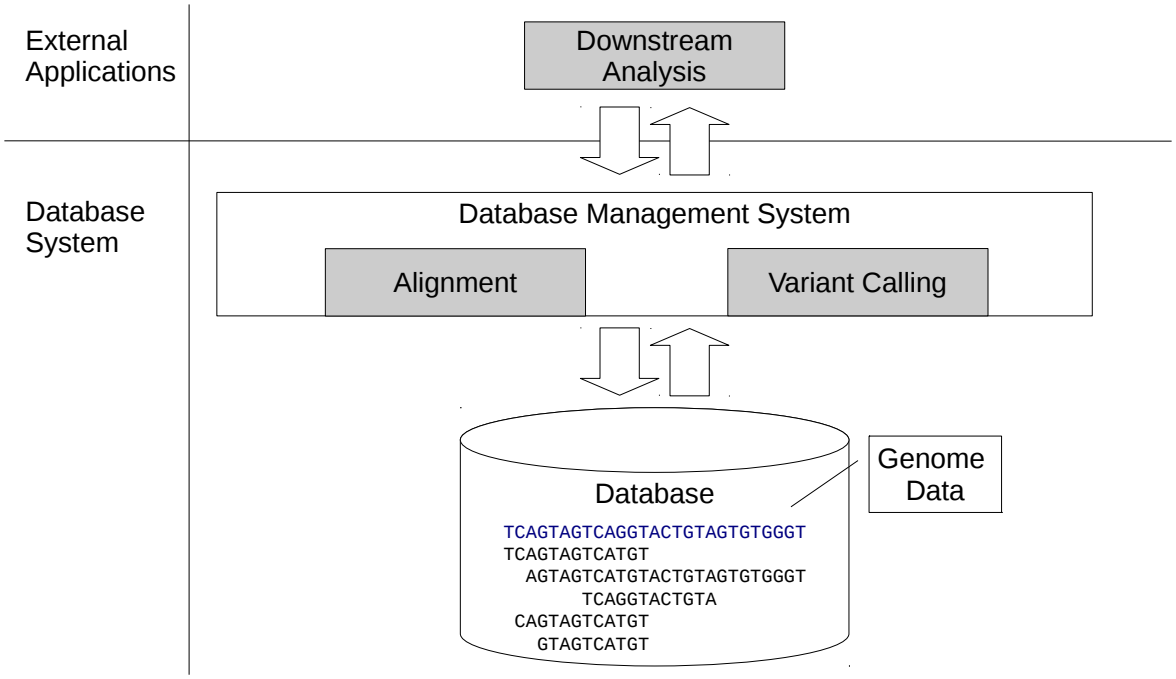


Figure 5.2: Analysis integration

Analysis Integration

In a second approach, we extend the previous approach by integrating genome analysis steps into the database system. In Figure 5.2, we depict the approach. The genome analysis steps sequence alignment and variant calling are integrated into the database management system. Thus, they can be executed from external applications.

Advantage of this approach is that analysis can be computed directly on the database. This direct computation of analysis reduces the amount of data to transfer between external applications and the database system. Furthermore, it allows to compute analysis results instead of storing them and thus saves storage. For example, an external application that needs the result of variant calling just executes the analysis on demand and uses the result for further processing. The ability to integrate analysis computation into a DBMS highly depends on extension capabilities of DBMSs. DBMSs provide common extension interfaces such as stored procedures or the execution of native code inside the DBMS context. In order to implement this approach a suitable database schema is needed. Furthermore, the database systems must have fast processing capabilities and must provide capabilities to be extended by custom functions that can be directly used in database queries.

5.1.3 Research Focus

In this section, we prepare the further research focus of this thesis. In Chapter 3, we identified benefits of using modern database systems to address challenges in genome analysis such as data integration capabilities and utilization of modern hardware architectures to query and process huge amounts of data fast. Furthermore, in Section 5.1.1, we have shown that genome analysis is an applicable database application. Moreover, the use of database systems, especially for downstream analysis, shows that genome analysis can be performed on database systems (cf. Chapter 4). Thus, we prepared the foundation for using modern database for genome analysis. Given the existing approaches introduced in Chapter 4, we conclude that downstream analysis is also possible using modern database systems. As we want to support all steps of genome analysis with database systems, we focus our further research in this thesis on integrating and supporting variant calling with modern database systems. Implementing and supporting sequence alignment with modern database systems is subject of our future research. Variant calling is the link between sequence alignment and downstream analysis (cf. Section 2.1.2). Our intention behind the decision to concentrate on variant calling is to provide this link. Thus, we can ensure gapless analysis capabilities as downstream analysis has already been integrated into database systems (cf. Chapter 4).

5.2 Variant Calling on Modern Database Systems

In this section, we introduce our concept for variant calling on modern database systems. First, we describe the basic concept and functional requirements. Then, we present our database schema and queries to implement these requirements.

5.2.1 Basic Concept and Functional Requirements

The goal of our concept for variant calling on modern database systems is to allow fast access to needed data for variant calling. Furthermore, we want to demonstrate the possibilities to integrate the analysis into the database management system. Advantage of the latter approach is reduced storage consumption, because derivable information does not have to be stored, but is computed. Requirement to achieve such on demand analysis is high processing speed.

Basic idea behind our concept is to import aligned reads from flat files into a database and use a database management system to perform queries and analysis on this data. In Figure 5.3, we depict the basic idea. The aligned reads are stored in a database and the database management system handles efficient data access. As pointed out in Section 5.1.3, we concentrate our research in this thesis to support variant calling. Therefore, we have defined the following functional requirements:

Querying

Q1 The user must be able to retrieve all genome data for a specific genome site in order to call variants externally.

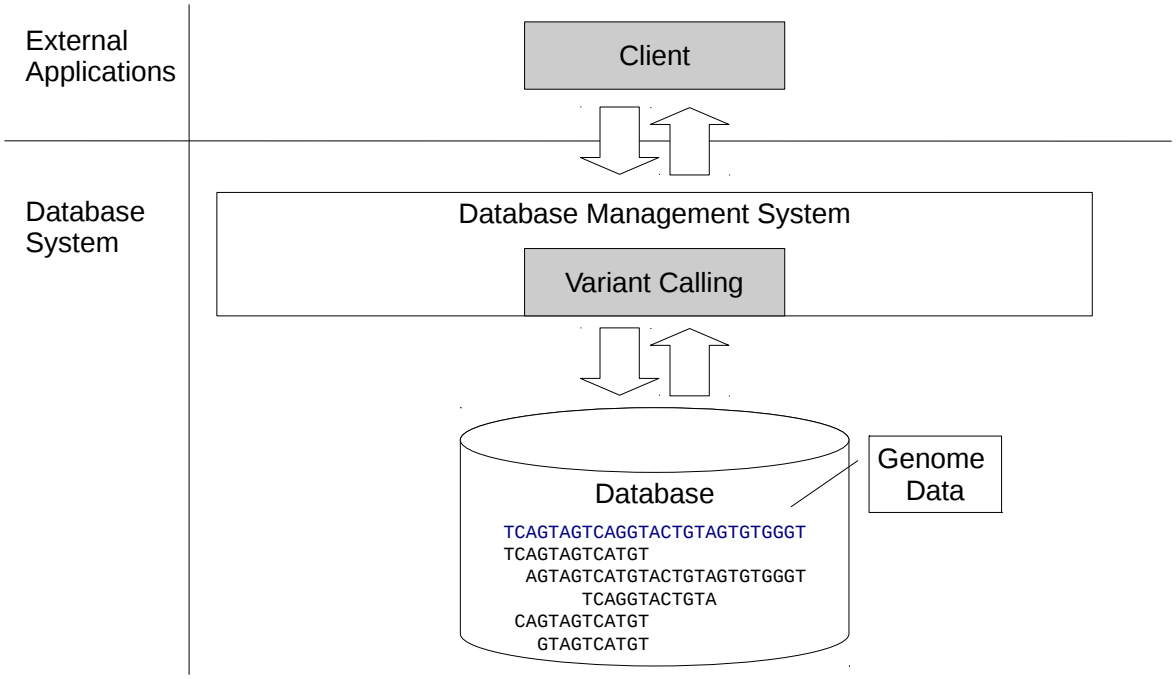


Figure 5.3: Basic concept

Following attribute values must be accessible by an external application to perform variant calling:

- sample bases at a specific site,
- the according base call qualities,
- mapping qualities of containing reads,
- and the reference base at the specific site

Q2 The user must be able to retrieve all potentially variant sites in a specific genome region. Potentially variant sites are all sites were at least one base of the sample sequence differs from the reference sequence.

Q3 The user must be able to retrieve all potentially inserted sites in a specific genome region. Potentially inserted sites are all sites were a base of the sample sequence is marked as inserted.

Q4 The user must be able to retrieve all potentially deleted sites in a specific genome region. Potentially deleted sites are all sites were a base of the sample sequence is marked as deleted.

Filtering

F1 Genome data must be filterable by base call quality.

F2 Genome data must be filterable by mapping quality.

F3 Genome data must be filterable by coverage.

Analysis Computation

A1 User can directly call a genotype via a database query.

A2 User can directly call variants via a database query.

In order to implement these requirements, we develop a suitable database schema and provide queries.

5.2.2 Database Schema

In this section, we introduce our database schema to support variant calling on modern database systems.

In Chapter 2, we have described procedures for calling SNVs and InDels. Basically, all procedures rely on the information, what bases are aligned to what genome site. We will utilize this characteristic while developing a data schema in order to efficiently support variant calling.

Over the last years, many database schemas were proposed to store not only sequencing data, but a lot of further genome information (cf. Chapter 4). These schemas are aiming at downstream analyses. Basis for these database schemas are reads (cf. [SHX+05, LPW+06, RB09]). But variant calling is done on a set of bases per genome site. Thus, read oriented approaches are inappropriate for variant calling, because the reads containing bases at a desired genome site must be split on the fly. Röhm et al. implemented such an approach for consensus base calling that is related with variant calling [RB09]. Doing this for 20 to 50 reads per site as in high coverage data sets, seems to be an enormous overhead. Moreover, Röhm et al. point out that this procedure leads to huge intermediate results that reduce the performance of relational disk-based database systems. To avoid this overhead and better support variant calling algorithms, we propose a base-oriented database schema. In Figure 5.4, we show the entity-relationship schema (cf. Chapter 2). It consists of three entities: *read*, *sample_base* and *reference_site*. The *reference_site* entity describes one specific site in a reference sequence. The *reference_site* entity has three attributes:

- The *id* attribute is a artificial, unique identifier for every reference site.
- The *position* attribute defines the specific position in a reference sequence.
- The *base* attribute contains the base value at the given position in the reference sequence. The possible base value is either A, C, G, T or N.

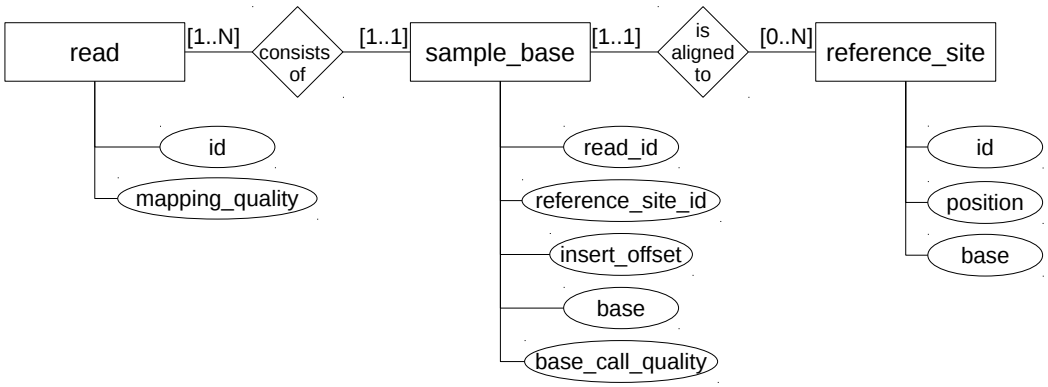


Figure 5.4: ER schema with min/max-notation

The *read* entity describes a read that is aligned against a reference genome and has two attributes:

- The *id* attribute is a artificial, unique identifier for every read.
- The *mapping_quality* attribute contains the quality value of the aligned read.

The *sample_base* entity describes one base in a read of a sample sequence. The *sample_base* entity has five attributes: *read_id*, *reference_site_id*, *insert_offset*, *base* and *base_call_quality*.

- The *read_id* is used to store the reference to the read that contains the current *sample_base*.
- The *reference_site_id* is used to store the reference to the *reference_site* the current *sample_base* is aligned to.
- The *insert_offset* attribute is 0 for bases that match to the reference genome. If a base is inserted, this is indicated with an *insert_offset* greater 0. If more than one base is inserted at a reference site, the *insert_offset* is used to define the correct insert order.
- The *base* attribute contains the nucleotide base at the given site. The base value is one of the defined base values in the FASTA format (cf. Table 2.1). In this format A, C, G and T encode the bases that make up a genome sequence. We use the "X" to explicitly indicate that a base is deleted at the according reference site. This is necessary for variant calling to avoid that erroneous alignments of single bases lead to false positives.

```
1  -- create table for reference sites --
2  CREATE TABLE "reference_genome" ("id" int, "position" int, "base" char(1));
3  ALTER TABLE "reference_genome" ADD PRIMARY KEY ("id");
4
5  -- create table for reads --
6  CREATE TABLE "reads" ("id" int, "mapping_quality" int);
7  ALTER TABLE "reads" ADD PRIMARY KEY ("id");
8
9  -- create table for sample bases --
10 CREATE TABLE "sample_genome" ("read_id" int, "reference_site_id" int,
11     "insert_offset" int, "base" char(1), "base_call_quality" int);
12 ALTER TABLE "sample_genome" ADD PRIMARY KEY ("read_id",
13     "reference_site_id", "insert_offset");
14 ALTER TABLE "sample_genome" ADD FOREIGN KEY ("reference_site_id")
15     REFERENCES "reference_genome" ("id");
16 ALTER TABLE "sample_genome" ADD FOREIGN KEY ("read_id")
17     REFERENCES "read" ("id");
```

Listing 5.1: Relational model as SQL statement

- The *base_call_quality* attribute stores the *Phred* encoded base call quality value for this base (cf. Section 2.1.3.2). In case it is a deleted base, no quality value is available in the input data. Therefore, the quality value is derived from the base call quality values of all bases in the read that contains the deletion. Thereby, the base call quality value is the average of all base call quality values. Thus, the quality of a deletion is dependent on the quality of the base calls of all bases a read consists of.

Every *sample_base* entity is aligned to exactly one *reference_site* entity, but there can be more than one *sample_base* entity that are aligned to the same *reference_site* entity. Furthermore, the *read* entity consists of at least one *sample_base* entity, whereas every *sample_base* entity is related to exactly one *read* entity.

In Listing 5.1, we show the relational database schema derived from the entity-relationship schema as SQL queries. The schema consists of three tables, one for each entity in the entity-relationship schema. Furthermore, primary and foreign key constraints are used to ensure data integrity aspects. We use this database schema for our further explanations.

5.2.3 Querying and Filtering

In this section, we present SQL queries to fulfill the functional requirements regarding querying (Q1 to Q4) and filtering (F1 to F3) presented in Section 5.2.1. We present the SQL queries as scaffolds, meaning there are placeholders. We mark placeholders by embracing them with dollar signs: *$placeholder$*. This notation is not standard SQL.

To query all available information for a specific genome site (Q1), we can use the query depicted in Listing 5.2. Our starting point is the *sample_base* table. In order to access

```
1  -- specify data to retrieve --
2  SELECT $columns$
3  -- join to access reference information --
4  FROM "sample_genome" s
5  JOIN "reference_genome" r
6  ON s."reference_site_id" = r."id"
7  -- join to access read information --
8  JOIN "read" read
9  ON s."read_id" = read."id"
10 -- access only data certain position --
11 WHERE r."position" = $position$;
```

Listing 5.2: Data querying

```
1      SELECT r."position"
2  FROM "sample_genome" s JOIN "reference_genome" r
3  ON s."reference_site_id" = r."id"
4  JOIN "read" read
5  ON s."read_id" = read."id"
6  -- filter for certain region --
7  AND r."position" BETWEEN $start$ AND $end$
8  -- base quality filter --
9  AND r."base_call_quality" > $threshold$
10 -- mapping quality filter --
11 AND read."mapping_quality" > $threshold$
12 -- filter for insertions --
13 AND s."insert_offset" = $offset$
14 -- filter for deletions --
15 AND s."base" = 'X'
16 -- filter for coverage --
17 GROUP BY r."position"
18 HAVING count(s."base") > $threshold$
```

Listing 5.3: Additional filtering

all information available for a certain base in the sample genome, we have to join this table with table *reference_site* and *read* by using the foreign key relationship (line 4 - 9). This join also allows us to access only data at a specific genome site by restricting the *position* attribute of the *reference_genome* table (line 11). Furthermore, we have to specify what data to retrieve by replacing the placeholder in line 2 with the desired column names of the three tables.

Another requirement for our concept is to allow filtering of genome data. In order to fulfill this requirement, we use WHERE predicates in SQL. We already used such a filter in Listing 5.2 in line 11. But, we can define more filters in that way. Listing 5.3 shows such additional filters. In line 7, we show how to specify a range query for reference base positions. Thus, certain genome regions of specific sizes in base pairs can be examined. Quality filters for base call quality (F1) or mapping quality (F2) are shown in line 9 and 11, respectively. The filters in line 13 and 15 can be used to filter deleted or

```
1  SELECT DISTINCT r."position"
2  FROM "sample_genome" s JOIN "reference_genome" r
3  -- join condition: different bases --
4  ON s."reference_site_id" = r."id" AND s."base" <> r."base"
5  -- exclude insertions --
6  WHERE s."insert_offset" = 0
7  -- exclude deletions --
8  AND s."base" <> 'X'
9  -- exclude unknown sites --
10 AND s."base" <> 'N'
11 AND r."base" <> 'N';
```

Listing 5.4: Query to detect potentially variant sites

inserted positions, respectively. The filter for coverage (F3) is realized by a GROUP BY statement and a HAVING clause that counts the sample bases at this position (line 17 - 18). All filters are optional but can influence each other, because the order of application cannot be controlled. Thus, a filter for coverage is always applied as last filter. In Listing 5.3, all filters must be true in order to select data. It is also possible to combine filters with an OR instead of AND, except the coverage filter, because of its definition via HAVING. All in all, SQL provides all means to implement effective data filtering. Moreover, further filters are possible.

After describing how to access and filter genome data, we explain now how to query for potentially variant, inserted, or deleted sites.

In Listing 5.4, we show the SQL query to determine potentially variant sites (Q2). Inserted and completely deleted sites cannot be target for SNV calling, because either no reference base or sample bases exist to compare with. Thus, insertions are excluded with a condition in the WHERE clause that filters out bases with an *insert_offset* greater 0 (line 6). Deleted sites in the sample genome are filtered out with another WHERE predicate that excludes "X" as base value (line 8). These filters guarantee that, if at least one non-reference base was aligned to a single site at the reference genome, this site is returned as a potentially variant site. Furthermore, it is guaranteed that, if all bases at a single site were deleted, this site is not recognized for variant calling. Additionally, sites are filtered where all sample bases are "N" or the reference base is "N" (line 10 and 11). The result is a distinct list of potentially variant sites.

In Listing 5.5, we demonstrate a way to detect potentially inserted sites in the sample genome regarding a reference genome (Q3). Our data schema (cf. Section 5.2.2) explicitly stores the information, whether a base was inserted or matches a site of the reference. That is why a simple filtering on *insert_offset* is enough to detect insertions (line 5). The final result is a distinct list of potentially inserted sites.

In Listing 5.6, we show how to detect potentially deleted sites (Q4). Because we store the information about deleted bases explicitly, we can query deleted sites by filtering for *base* attributes storing an "X" (line 5). The result is a distinct list of sites where at least one base is marked as deleted.

```
SELECT DISTINCT r."position"
FROM "sample_genome" s JOIN "reference_genome" r
ON s."reference_site_id" = r."id"
-- inserted bases have a insert_offset greater than 0 --
WHERE s."insert_offset" > 0 ;
```

Listing 5.5: Query to detect potentially inserted sites

```
SELECT DISTINCT r."position"
FROM "sample_genome" s JOIN "reference_genome" r
ON s."reference_site_id" = r."id"
-- deleted sites have a 'X' as base value --
WHERE s."base" = 'X';
```

Listing 5.6: Query to detect potentially deleted sites

Using the above queries, an external application is able to query potentially variant, deleted, and inserted sites in a sample genome. Furthermore, an application can retrieve all available information for such sites in order to further analyze them.

5.2.4 Analysis Integration

In this section, we present queries to perform genotype and variant calling (requirements A1 and A2) directly on the database. Thus, an external application gets the result of genotype or variant calling from the database and does not have to compute it externally.

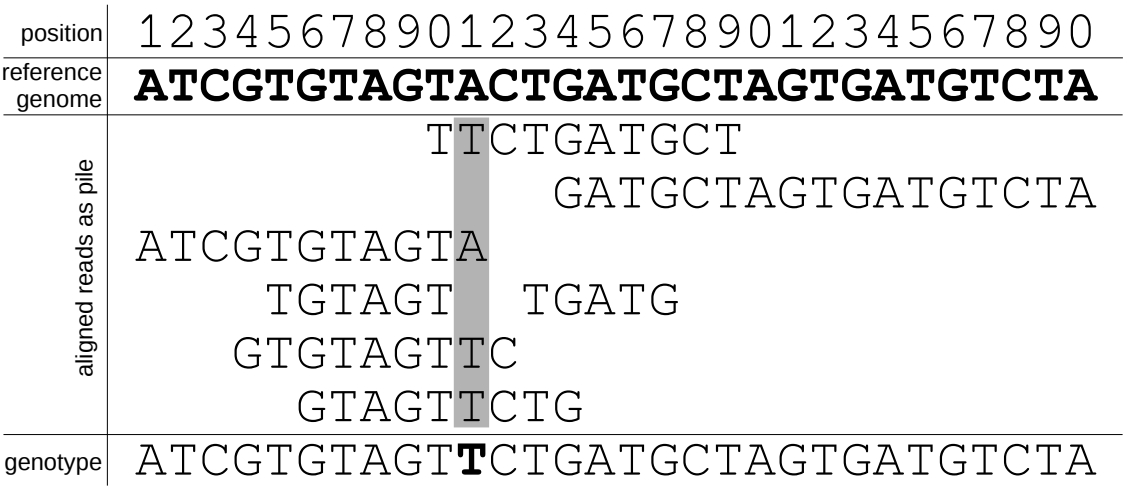


Figure 5.5: Genotype and variant calling at a specific genome site

The result of sequence alignment can be visualized as a pile of reads according to a reference sequence (see Figure 5.5). Calling a genotype or variant at a specific genome

site requires at least the information about all sample bases that are aligned to this site. In Figure 5.5, the gray bar indicates all sample bases that were aligned to position 11 of the reference genome. A genotype call at this position could result in "T", because most reads aligned to the reference genome in this region contain base "T" at position 11. Thus, a later variant caller could decide that at position 11 a variant exists, because the reference base is an "A". In Section 5.2.2, we proposed a base-oriented data schema, that allows direct access to aligned bases. We use this fact and use built-in database functionality to perform genotype calling at specific genome sites. In Listing 5.7, we show the scaffold for genotype calling (A1). We utilize the GROUP BY feature of SQL to aggregate data at genome sites (line 14). The grouping requires that all non grouping attributes must be aggregated to one value. Thus, we have to and want to provide an aggregation function over the sample bases that aggregates all bases by calling a genotype for the specific site (line 2). Hereby, we can also use the previously introduced filtering operations (cf. Listing 5.3). Thus, we can exclude bases with low base call quality or bases of reads with low mapping quality.

```
1  -- aggregation function derives genotype --
2  SELECT r."position", s."insert_offset", genotype(s."base")
3  -- joins to allow access to all related information --
4  FROM "reference_genome" r JOIN "sample_genome" s
5  ON s."reference_site_id" = r."id"
6  [JOIN "reads" reads
7  ON s."read_id" = reads."id"]
8  WHERE s."insert_offset" = 0
9  -- certain filters to improve the genotype calling --
10 [AND reads."mapping_quality" > $threshold$]
11 [AND s."base_call_quality" > $threshold$]
12 [AND r."position" BETWEEN $start$ AND $end$]
13 -- creates the pile of bases --
14 GROUP BY r."position", s."insert_offset";
```

Listing 5.7: Query for genotype calling

In Listing 5.8, we show how to use the genotype aggregation function for genotype calling to call variants (A2). Therefore, we use the previously introduced query for genotype calling (cf. Listing 5.7) as in-line view (line 2 - 15). The in-line view is processed before our regular query (line 17 - 19). Thus, we can use the in-line view like a normal table in our regular query. Our simple variant caller decides to call a variant, if the called genotype differs from the reference base (line 19).

In all analysis queries above, data access is handled directly via built-in database operators. These operators are highly optimized and thus data access should be as fast as possible. In case of genotype calling, we have to develop a custom aggregation function, which is not trivial in all database systems. Using this genotype calling function, we could easily implement a variant caller.

```
1  -- in-line view for genotype calling --
2  WITH aux ("position", "ref_base", "genotype")
3  AS (
4  SELECT r."position", r."base", genotype(s."base") as "genotype"
5  FROM "reference_genome" r JOIN "sample_genome" s
6  ON s."reference_site_id" = r."id"
7  -- optional join with read table to filter for mapping quality --
8  [JOIN "reads" reads
9  ON s."read_id" = reads."id"]
10 WHERE s."insert_offset" = 0
11 [AND reads."mapping_quality" > $threshold$]
12 [AND s."base_call_quality" > $threshold$]
13 [AND r."position" BETWEEN $start$ AND $end$]
14 GROUP BY r."position", r."base"
15 )
16 -- variant calling step --
17 SELECT aux."position", aux."ref_base", aux."genotype"
18 FROM aux
19 WHERE aux."ref_base" <> aux."genotype";
```

Listing 5.8: Query for variant calling

5.3 Summary

In this section, we briefly summarize our contribution of this chapter.

In Section 5.1, we showed that genome analysis is a suitable database application by evaluating genome analysis against criteria that exclude or at least limit the applicability of database systems. Furthermore, we developed two implementation approaches to support genome analysis with modern database systems. First approach is to use database systems as a data storage and for querying genome data. The second approach is to extend the database management system with genome analysis functionality such as sequence alignment or variant calling and thus freeing external tools from performing these analysis steps themselves.

In Section 5.2, we developed a concept to perform variant calling with modern database systems. Therefore, we created a suitable database schema and presented queries to query genome data. Moreover, we presented a solution to integrate variant calling functionality into a database management system via a custom aggregation function.

6. Evaluation

In this chapter, we evaluate our concept for variant calling on modern database systems from Chapter 5. Until now, we have answered the first three of our six research questions from Section 1.2 by qualitative discussion in Chapter 3 and Chapter 5. The remaining three research questions are:

RQ 4 Is the disk storage consumption of approaches that use modern database systems for genome analysis competitive to flat file approaches?

RQ 5 Is the analysis performance of genome analysis on modern database systems competitive to state-of-the-art tools?

RQ 6 Is the result quality of genome analysis on modern database systems competitive to state-of-the-art tools?

During preparation of our thesis, we identified four more research questions:

RQ 7 How much disk storage is effectively used for genome data?

RQ 8 How scalable is genome analysis on modern database systems regarding analysis performance?

RQ 9 Does integrated variant calling outperform external variant calling?

RQ 10 How effective is the compression of genome data that is stored in our database schema?

In order to answer the remaining research questions, we have to conduct experiments. Therefore, we first define the objectives of our evaluation in order to prepare appropriate experiments to answer our questions. Then, we describe our experimental design to conduct our experiments. Afterwards, we present the experiment results and interpret them. Finally, we state threats to validity and summarize our findings.

6.1 Evaluation Objectives

In this section, we specify the variables of our experiments, and thereby our evaluation objectives. Experiment variables can be categorized into dependent and independent variables [Fei09, p. 25]. Independent variables are those that are intentionally varied during experiments. Dependent variables describe a certain outcome of an experiment that is interpreted regarding the chosen independent variables to answer a research question. For example, if we want to evaluate the scalability of a range query regarding response time, we measure the response time (dependent variable) for that range query and vary the range size (independent variable). By concreting our evaluation objectives, we are able to prepare appropriate experiments. Furthermore, we are able to construct hypotheses about the outcome of the experiments. In the example above, we could hypothesize that the response time increases with increasing range size. Then, we can interpret the experimentally determined response times and evaluate the scalability of the range query.

6.1.1 Independent Variables

The independent variables of our evaluation are: (i) database management system; (ii) range size; (iii) location of analysis computation; (iv) hardware; and (v) genome data.

In the following, we describe the independent variables of our evaluation:

Database management system The selection of a database management system highly influences the outcome of experiments as different DBMSs use different approaches to provide DBMS functionalities. For example, every DBMS has its own query optimizer that affects query performance. As database management systems, we use MonetDB 5 and SAP HANA. MonetDB is an open-source DBMS developed at the Centrum Wiskunde & Informatica (CWI[1]) in the Netherlands. It is a research prototype and mainly used to evaluate self-optimizing techniques. Thus, users have no abilities to create indexes themselves as MonetDB decides when to create an index for optimizing queries. SAP HANA is a commercial in-memory DBMS that can store data either row or column oriented [SAP13, p. 151]. We use the column oriented storage feature. Furthermore, SAP HANA makes use of different compression techniques such as run-length and dictionary encoding, whereas MonetDB only supports dictionary encoding on string values. We have chosen these two database management systems as they are column-oriented, in-memory DBMSs, whereby SAP HANA is an commercial product and MonetDB is open-source software.

Range size Varying the query range of our queries, gives us hints about scalability. Using different query ranges, we can simulate the analysis of specific genome regions (e.g., genes) and identify for what use case our concept can be used well.

[1] http://www.cwi.nl/

In our further explanations, we state range sizes of a query range in number of base pairs that are covered on the reference sequence by that range.

Location of analysis computation Computing the result of an analysis directly inside the database management system or externally influences the analysis performance. In the external case all needed data for analysis computation must queried beforehand and then passed to the external application. In contrast, when computing the analysis result internally, only the result must be passed to an external application. To answer our question whether integrated analysis computation outperforms external computation (RQ 9), we implement two versions of a variant calling tool. First version calls the variants externally. Second version uses an analysis query that calls the variants internally.

Hardware Varying the hardware specification (e.g., increasing main memory or number of CPUs) gives hints about scalability of a system. For example, when we increase the size of main memory, an in-memory DBMS can keep more data in main memory, and thus query performance can increase as less data must be loaded from disk in case the complete database does not fit into the main memory. In our experiments, we use different but fixed hardware setups for our different DBMSs, but assure that all data fits into main memory. Despite the different hardware setups, the validity of our results is not decreased as we do not compare the performance of the two DBMSs directly.

Genome data Performing queries on different sets of genome data can influence the query performance. For example, the complete genome of the Escherichia coli (E. coli) K-12 bacterium comprises 4,639,221 base pairs [BIB+97], whereas a complete human genome comprises 3.2 billion base pairs. Thus, the runtime of range queries over complete genomes is dependent on the genome size. Moreover, genome sequencing data has a certain coverage, meaning that one genome site is read multiple times (cf. Section 2.1.2.1). The higher the coverage, the more data must be stored and processed. Thus, sequencing coverage can influence disk storage consumption and query runtime. For our experiments, we use genome reference and alignment data of the first human chromosome. The sequencing coverage of our alignment data is low.

6.1.2 Dependent Variables

We derive the dependent variables from our research questions. The dependent variables are: (i) disk storage consumption; (ii) compression effectiveness; (iii) analysis performance; and (iv) analysis scalability. Since it is not in our scope to develop a competitive product to existing tools, we decide not to evaluate result quality (e.g., correctness or completeness) of our queries and analysis functions. Furthermore, it is very likely that different existing variant calling tools call different variants on the same genome data [OJS+13]. Thus, it is hard to determine the one correct variant calling

result necessary for an evaluation and we leave the evaluation of result quality open for future work.

In the following, we describe how we measure the dependent variables of our evaluation:

Disk storage consumption Disk storage consumption describes the amount of storage data consumes on disk. For our implementation, we consider database systems that persist their data on disk. To get knowledge about disk storage consumption, we use internal database system statistics and the Linux tool *du*. In order to determine the disk storage that is effectively used for genome data, we exclude disk storage consumed by auxiliary data such as indexes that is created by database systems when primary and foreign keys are applied.

Compression effectiveness We define the ratio between compressed and uncompressed data size in main memory as compression effectiveness. In order to determine the compression ratio of a modern database system, we have to measure the compressed and uncompressed size of data. Therefore, we use internal database statistics and the Linux tool *du*.

Analysis performance With analysis performance, we describe how fast the result of an analysis can be computed. In Section 5.2, we distinguish two different analysis use cases: (a) *querying for potentially variant, inserted, and deleted sites* and (b) *analysis integration of genotype and variant calling*. To evaluate the analysis performance of queries (use case a), we measure the query runtime. To successfully process a query, the database system has to perform different tasks such as query parsing, logical and physical optimization, data access, and result delivery (cf. Section 2.2.2). Every step needs processing time. Our interest relies in the overall performance of queries. Thus, we measure the execution time of queries on client site from starting the query until the database has computed the query result. We exclude time to deliver and display the result. Thus, we avoid influences due to network communication, when clients are remote or due to the usage of different clients. To evaluate whether integrated variant calling (use case b) outperforms external variant calling (RQ 9), we also have to measure the analysis performance. This time, we measure the time from starting the analysis computation until the result is computed and displayed in order to make the results comparable.

Analysis scalability The queries that we defined in Section 5.2 are range queries. In order to determine the scalability of these queries over different range sizes, we vary the range size and measure the query runtime.

6.2 Experimental Design

In this section, we describe our experimental design regarding our evaluation objectives from Section 6.1. In order to collect useful data for evaluating disk storage consumption, compression effectiveness, analysis performance, and analysis scalability, we have

to design our experiments carefully, because there are confounding variables that could influence the evaluation results. Thus, we first determine confounding variables for measuring analysis performance and disk storage consumption. Thereby, we explain how to reduce the impact of these confounding variables. Second, we develop appropriate experiments to collect data for each evaluation aspect considering these confounding variables.

6.2.1 Confounding Variables

Confounding variables can influence evaluation results by distorting measurements. We identified following confounding variables for measuring query runtime: (1) cold or warm memory; (2) unexpected CPU load; and (3) network performance. In the following, we describe each confounding variable and present appropriate countermeasures to avoid or at least limit their influence:

1. Cold or warm memory
   In-memory database systems reach fast query performance when all data needed for query processing is available in main memory.

   Problem For our evaluation, we use modern database systems that persist data on disk. Thus, they are able to store more data than fits into main memory. Because of this, not all data must reside in main memory at query processing time. Thus, data must be loaded into main memory query processing. This increases processing time.

   Solution Before we start query performance measurements, we run warm-up queries. Because modern database systems try to cache data for subsequent query execution, running warm-up queries should cache the data in main memory to avoid further data loading for further query processing.

2. Unexpected CPU Load
   Many different processes are running on a computer system. These processes are initiated either by users or the operating system. Some processes are background tasks (e.g., to clean up the file system).

   Problem We cannot guarantee that the database processes are the only active ones when we run our performance measurement.

   Solution We run ten measurement queries instead of just one measurement query. Then we compute the average execution time. Thereby, we exclude outliers by ignoring the two lowest and two highest measurement values. Furthermore, we repeat query sets twice that have an average execution time lower than 10 seconds and use the average of the three average execution times for evaluation. Thus, the effect of unexpected CPU load should be minimized.

3. Network Performance
   Database systems run mainly on dedicated server hardware. Such servers are mainly accessed by remote clients via Secure Script Shell (SSH) or Remote Desktop Protocol (RDP).

   Problem Limited network performance can influence our performance measurement.

   Solution We just measure the query execution time from starting the query until the server has finished to prepare the result. We exclude the time for delivering and displaying the result. Another solution is to avoid network traffic by running measurements directly on the server.

Regarding the measurement of disk storage consumption, we identified following confounding variable that is a feature of file systems: block allocation. File systems allocate disk space for files in blocks of several bytes (e.g., 1,024 or 4,096 bytes) [BC02, p. 575]. A typical block size is 4,096 bytes. Thus, a file of 6 bytes consumes 4,096 bytes on disk; a file of 4,097 bytes consumes 8,192 bytes. The additional space is called *slack space*. The file sizes, we are dealing with in our experiments, have a size of several gigabytes, thus, such small amounts of slack space do not distort the measurements much. Moreover, current file systems allow to use the slack space to store other payload data [BC02, p. 576]. Nevertheless, we have to define whether we measure the disk storage consumption in block sizes or only the disk space used for payload in order to compare consumed disk storage between database systems and flat files properly. Thus, the question for our example above is: *Has the file a size of 4,096 bytes or 6 bytes?*

To measure the disk storage consumption of the database systems, we mainly rely on internal database statistics. However, MonetDB allows us to determine the data files that store the database. Thereby, every column of a table is stored in a separate file. Thus, we are able to crosscheck the file sizes reported by the internal database statistics with the file sizes determined with an external tool such as Linux's *du*. We figured out that MonetDB does not consider the block size when reporting the file or column size, but reports a theoretical size by multiplying the number of records in a relation with the data type size of the respective column. Thus, the statistic reports the payload disk storage without slack space. In SAP HANA, we do not have the information, which file stores which data, and thus we rely on the database statistics. As we assume a similar behavior of SAP HANA as MonetDB shows, we determine the size of files and folders by measuring the disk space used to store the payload. Thus, in our example above, we would say that the file consumes 6 bytes of disk space instead of 4,096 bytes.

6.2.2 Test Environment

In this section, we describe what hardware and software we use for our experiments. Moreover, we describe our evaluation data set.

6.2.2.1 Software

We use different DBMS systems to evaluate our approach for variant calling on database systems. We use MonetDB 5 Feb2013-SP2 release and SAP HANA version 1.00.60.379234.

We compile MonetDB5 from source, because we have implemented a custom aggregation function (cf. Section 5.2.4). We used following configuration options for the build:

```
--enable-debug=no --enable-assert=no --enable-optimize=yes.
```

The SAP HANA software is already tuned and installed on our evaluation hardware by SAP.

6.2.2.2 Hardware

We run our evaluation measurements on two different hardware setups. We install MonetDB on a Amazon Elastic Computing Cloud (EC2) instance[2] with 67 GB main memory and 8 CPU cores. The Ubuntu 13.04 64bit operating system runs on para-virtualized hardware. Para-virtualization allows the guest operating system to interact more directly with the host hardware, and thus the guest can utilize the host hardware more efficient than in a fully virtualized environment [BDF+03]. Furthermore, the disk storage is connected via LAN as so called Elastic Block Storage (EBS[3]) device. Thus, disk access could be slower than access to a directly attached disk. SAP AG provides us a testing server for their SAP HANA software with 24 CPU cores and 141.8 GB main memory. During preparation of our evaluation and against our expectation and assertions from SAP, we detected that the SAP HANA instance runs inside a virtual machine. In Table 6.1, we show a more detailed comparison of used hardware and operating system properties. We extracted the information from Linux's system information files */proc/cpuinfo*, */proc/meminfo*, and */proc/version*. Moreover, we consulted Intel's website[4] for more information about the processors. In summary, it can be stated that both servers run on virtualized hardware. Thereby, the SAP HANA instance has more cores and main memory than the Amazon instance we use to run MonetDB.

6.2.2.3 Test Data

We use genome data provided by the 1000 Genomes Project [VGB10]. The 1000 Genomes Project provides raw sequencing data and aligned sequencing data of humans. Moreover, the 1000 Genomes Project provides the reference genomes used for sequence alignment (cf. Section 2.1.2). We use the aligned sequences of one human sample called HG00096.[5] The alignment data is provided in a binary encoded SAM file (cf. Section 2.1.3.3). With the tool samtools [LHW+09], we converted the file into human readable ASCII encoding. The reads in our source SAM files were mapped against the human chromosome reference sequences (cf. RNAME in Section 2.1.3.3).

[2] http://aws.amazon.com/ec2/instance-types/
[3] http://aws.amazon.com/ebs/
[4] http://ark.intel.com
[5] http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/HG00096/alignment/

Property	Amazon m2.4xlarge instance	SAP HANA instance
CPU Name	Intel(R) Xeon(R) CPU E5-2665	Intel(R) Xeon(R) CPU X5670
# of Cores per CPU	8	6
# of CPUs	1	4
Clock Speed	2.4 GHz	2.93 GHz
L3 Cache Size	20 MB	12 MB
Memory Size	67 GB	141.8 GB
Max Memory Bandwidth	51.2 GB/s	32 GB/s
OS Name	Ubuntu 13.04 64bit	SUSE Linux 64bit
Linux Version	3.8.0	2.6.32.59

Table 6.1: Hardware and OS specification of Amazon EC2 and SAP HANA instance

In order to ensure that all genome data fits into main memory when processing it, we reduce the alignment data. Therefore, we used samtools to extract all reads that were mapped against the reference sequence of chromosome 1. The reference sequences used for sequence alignment are provided in FASTA format (cf. Section 2.1.3.1). The provided FASTA file contains all chromosomes of the human genome as single sequences. We split the FASTA file into the according chromosome sequences using following command csplit $<$fastafile$>$ \$/^>.*/ {*}\$. Thus, the first generated file contains no information but the second file contains the sequence of the first chromosome, the third file the sequence of the second chromosome and so forth. In summary, our test data for evaluation comprises the human reference chromosome 1 and the according alignment data of human sample HG00096. Chromosome 1 is the biggest human chromosome and comprises 249,250,621 reference bases. The alignment data comprises 11,277,558 reads with 1,080,535,160 bases.

We developed a tool to convert the reference FASTA file for chromosome 1 and the sample data for chromosome 1 into files with character separated values (CSVs) that we use for bulk loading data into the databases. The tool is written in Java. The CSV files map the relational data schema as defined in Listing 5.1. FASTA files are converted into CSV files where each line represents a *reference_genome* data record with following information:

$$id|position|base$$

SAM files are converted into CSV files where each lines represents a *sample_base* data record with following information:

$$readid|referencesiteid|insert_offset|base|base_call_quality$$

The information about *insert_offset* and deleted bases is extracted from the CIGAR string (cf. Section 2.1.3.3).

The SAM file also contains the mapping_*quality* of a read. In order to import this information into our database, we generate a third CSV file. Each line in this CSV file represents a *read* data record with following information:

$$id|mapping_quality$$

All data is extracted directly from the source files, except the read ids, which are artificial. To import further genome data (e.g., chromosome 2 data), we either could create new tables for the respective reference and sample genome data or have to manage to set correct *reference_site_id* attribute values externally.

6.2.3 Experiments

In this section, we describe our experiments that we conduct in order to answer our research questions and to evaluate our variant calling concept on modern database systems regarding disk storage consumption, compression effectiveness, analysis performance, and analysis scalability. Therefore, we describe each experiment in detail in the following sections.

6.2.3.1 Disk Storage Consumption

With this experiment, we want to determine the disk storage consumption of our approach for variant calling on modern database systems. Thereby, we compare the disk storage consumption of our database systems with the disk storage consumption of flat files (RQ 4). Furthermore, we want to determine how much disk storage is effectively used for genome data (RQ 7). Therefore, we measure the disk storage consumed by flat files and database systems. Moreover, we measure how much disk storage is effectively used for genome data in database systems.

The experiment proceeds as follows:

1. We create input CSV files for bulk loading as described in Section 6.2.2.3.
2. We create the database schema using the SQL queries from Section 5.2.2. In order to create column-oriented tables in SAP HANA, we have to slightly adjust the stated SQL queries. Instead of using *CREATE TABLE* statements, we use *CREATE COLUMN TABLE* statements to explicitly instruct SAP HANA to store data in column-oriented layout as SAP HANA supports row-oriented storage as well as column-oriented storage [SAP13, p. 151].
3. We import the CSV files using database management system specific import statements. For MonetDB, the import statements look like:

```
COPY INTO "genomics"."table"
FROM '/path/to/csvfile';
```

For SAP HANA, following import statement is used:

```
IMPORT FROM CSV FILE '/path/to/csvfile'
INTO "genomics"."table"
WITH
RECORD DELIMITED BY '\n'
FIELD DELIMITED BY '|';
```

4. We measure the disk storage consumption using the database management system's internal statistic features. MonetDB uses one memory-mapped file per table column to persist data. All files that belong to the same database are stored under the same folder. Thus, we can directly determine the disk storage consumption of the complete database by summing up the file sizes of the related database files. Therefore, we use the Linux command *du -b* to determine the disk storage size of the payload data in the database folder in bytes (cf. Section 6.2.1). In order to get the storage consumption only for database files that store genome data, we use following SQL query:

   ```
   SELECT * FROM storage();
   ```

   This query lists all columns and the respective files that stores the columns' data. With the information about which files store which column data, we can use the Linux command *du -b <filename>* to determine the disk storage size of the columns of interest and sum these up to compute the single table sizes. The listed columns are not only data columns for genome data, but also columns that hold index data. Thus, we can simply determine the disk storage that is effectively used for genome data by summing up the file sizes of the data columns.

   In SAP HANA, we cannot simply measure file sizes, but have to use SAP HANA's internal statistics. In order to determine the storage consumption of the complete database, we use following query:

   ```
   SELECT SUM("DISK_SIZE") FROM "M_TABLE_PERSISTENCE_STATISTICS";
   ```

   The table *M_TABLE_PERSISTENCE_STATISTICS* stores information about the disk size of tables inclusive auxiliary data. Thus, we sum up the *DISK_SIZE* of all tables and compute the overall disk storage consumption. As the table *M_TABLE_PERSISTENCE_STATISTICS* provides information about all tables in the database, we can also extract the information about disk size of the tables that belong to our database schema. In contrast to MonetDB, it is not possible to determine the disk storage consumption of single columns. Furthermore, we found no way to determine the additional disk storage consumed by indexes that were created when primary and foreign keys are applied. For this reason, we repeated the steps two to four on SAP HANA, but we removed the primary and foreign key declaration from the database schema. Thus, we were able to measure the table sizes without auxiliary data.

5. We determine the size of source FASTA and SAM files using the Linux command *du -b <filename>* that lists the amount of space the respective file consumes on disk.

We assume that the disk storage consumption of the source flat file is smaller than the disk storage consumption of the database systems, because the database systems store additional auxiliary information such as indexes. Furthermore, in our approach, we store information about genomes such as positions of reference bases or inserted and deleted bases explicitly. On the one hand, this allows us to query genome data fast and directly with standard SQL queries (cf. Section 5.2.3). On the other hand, flat files use optimized data formats such as the CIGAR string (cf. Section 2.1.3.3) to store information about deleted and inserted sample bases in a very compact way that saves disk storage.

6.2.3.2 Compression Effectiveness

With this experiment, we want to determine the compression effectiveness of modern database systems regarding columns that contain genome data (RQ 10).

MonetDB only provides dictionary compression on strings. As our database schema only uses *char(1)* and *int* (32 bit integers), MonetDB's compression capabilities are limited, because it does not compress *int* values. Furthermore, the storage width in MonetDB ranges from 1 byte to 8 bytes, thus, compressing *char(1)* values of 1 byte length would only produce overhead due to an additional dictionary. But, we use the column sizes in MonetDB as reference for our further investigation of SAP HANA's compression effectiveness. SAP HANA applies different compression techniques such as run-length encoding on all data types, not only strings. We determine the compression effectiveness of SAP HANA as ratio between the (compressed) column size in SAP HANA and the (uncompressed) column size in MonetDB:

$$compression_ratio_{SAPHANA} = \frac{column_size_{SAPHANA}}{column_size_{MonetDB}}$$

A *compression_ratio* value greater than 1 indicates that the compressed column in SAP HANA consumes more memory than the uncompressed in MonetDB. Thus, the compression is not effective as it leads to increased memory usage. A value of 1 indicates that compression has no advantage. A value lower than 1 indicates that memory usage is reduced. The lower the value, the more effective is the compression.

To determine the in-memory column sizes in SAP HANA, we use following query:

```
SELECT * FROM "M_CS_COLUMNS" WHERE "SCHEMA_NAME" = 'genomics';
```

This statements lists all columns of tables in schema *genomics* and provides information about columns such as the used compression technique and in-memory size. To determine the column sizes in MonetDB, we measure the sizes of the corresponding

memory-mapped files that store the column's data, because the memory-mapped files are directly loaded into main memory.

We expect that SAP HANA needs less main memory to store the columns containing genome data than MonetDB as SAP HANA can apply compression techniques on all columns and data types.

6.2.3.3 Analysis Performance and Scalability

With this experiment, we want to evaluate the query performance (RQ 5) of our analysis queries to detect potentially deleted, inserted or variant sites from Section 5.2.3. Furthermore, we want to determine the scalability of these queries (RQ 8). Therefore, we vary the range size of the queries. To define suitable range sizes for our evaluation, we refer to human gene sizes. As the human gene size varies greatly from some 1000 bases pairs towards 2.5 million base pairs [SR99, Figure 7.7], we define following range sizes: 100,000, 2,500,000, 10,000,000, and 100,000,000 base pairs. We choose these ranges randomly, because we currently lack concrete annotation information about gene sites in our genome data. Furthermore, regions coding genes are scattered over the genome and do not have to border. We also apply the queries on the complete chromosome 1.

In order to evaluate the query performance, we need a benchmark runtime of a state-of-the-art variant calling tool with which we compare the runtime of our queries. Therefore, we use the runtime of bcftools that ships with samtools and is used to call variants. We use the runtime of bcftools to get an idea of an acceptable runtime for variant calling. We use Linux's *time* command to measure the runtime. We run bcftools on the same hardware as MonetDB and use the same range sizes as we use for our queries. Because bcftools does more than just querying for potentially inserted, deleted, or variant sites as we do with our queries, we use the runtime of bcftools as upper bound for the query runtimes. When our queries need more time to determine sites that are interesting for further investigation than bcftools needs for calling variants, the query runtime is not acceptable. Reason for this is that subsequent analysis steps such as variant calling at these sites would increase the analysis time further. Certainly, if our database queries run faster, they do not have to necessarily finish in acceptable runtime, because of unknown runtime of subsequent analysis. Thus, we introduce an artificial benchmark runtime that indicates acceptable runtime. We define this benchmark runtime to be 20% of the runtime of bcftools as we currently lack information on how long subsequent analysis needs. Thus, 80% of the bcftools runtime is left for subsequent analysis. If our queries run faster than this benchmark, they have acceptable runtime. Otherwise, the query runtime is not acceptable. As it is not our goal to compare the query runtime between MonetDB and SAP HANA, we can use different hardware setups for our evaluation.

We expect increasing query execution times with increasing range sizes as the intermediate result sizes increase, and thus increasing amounts of data must be processed during query processing.

6.2.3.4 Integrated vs. External Variant Calling Performance and Scalability

With this experiment, we want to examine whether integrated analysis computation outperforms external analysis computation (RQ 9) and how scalable analysis computation based on database systems is (RQ 8). Therefore, we implement two approaches for variant calling with database systems. First approach is to perform the variant calling directly inside the database system and just retrieve the analysis result. The second approach is to query the database for relevant data and compute the analysis externally.

To call variants, we determine the genotype at genome sites within range using cut-off rules as described in Section 2.1.2.3. Thereby, we count the occurrences of all base values A, C, G, and T at a genome site and determine the frequency of occurrence. We extend this approach for variant calling to handle X values that mark deleted bases as we store these explicitly in the database for each genome site. If the occurrence frequency of one base value is more than 80% then we return this base value as genotype or indicate a deleted site when an X is returned. When two or more bases occur with a frequency of 20% and more, we concatenate these base values as result. Then, the genotype is compared to the base value of the reference genome at this site. If the genotype and the reference base value differ the position, reference base and the genotype is reported. The result of variant calling is mostly improved by considering only high quality base calls and sequence alignments. Thus, we consider only bases that have a base call quality value of 30 or more. Moreover, we exclude bases that are contained in reads with a mapping quality less than 30. A quality value of 30 denotes an error probability for erroneous alignment or base calling of 0.001% (cf. Section 2.1.3.2).

We developed a command line tool that supports two analysis computation modes. In the *integrated* mode, the tool queries the database using the query for variant calling given in Listing 5.8 in Section 5.2.4. Therefore, we implemented the algorithm for calling genotypes from above as aggregation function in MonetDB. Thus, the tool just outputs the result of the database query. In *external* mode, the tool queries the database for all bases in a given reference base pair range fulfilling the filter predicate regarding base call quality and mapping quality. Then, the tool computes the analysis result externally according to our algorithm above and outputs the result.

We measure the execution time of our tool on base pair range sizes of 1,000, 10,000, 100,000, 2,500,000, 10,000,000, and 100,000,000 base pairs. Furthermore, we run the analysis on the complete chromosome. We use Linux's *time* command to measure the execution time. The *time* command gives us the information about how long the program effectively computes and waits for other processes. As benchmark runtime, we use the runtime of bcftools over the same range sizes to decide whether the runtime is acceptable. We run our tool on the Amazon EC2 instance where MonetDB is running, and thus avoid network communication.

In *external* mode, our tool queries all data in the given range that fulfills our filter predicate. With increasing range sizes, more data must be transferred. In contrast,

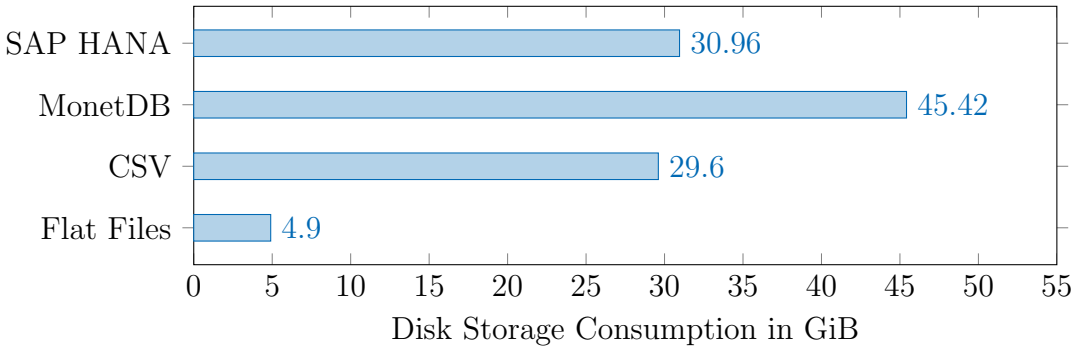


Figure 6.1: Disk storage consumption of complete database

in *integrated* mode, the tool just queries the analysis result. Thus, the amount of data to transfer is smaller than in *external* mode. For that reason, we expect that the performance benefit of the *integrated* approach increases with increasing base pair range size as more data must be transferred in *external* mode than in *integrated* mode when the range size increases.

6.3 Results and Discussion

In this section, we present and discuss the results of our experiments. First, we analyze the disk storage consumption and compression effectiveness of our approach for variant calling on modern database systems. Then, we evaluate the analysis performance and scalability of the different queries for retrieving potentially deleted, inserted, and variant sites in the sample genome. Finally, we assess whether integrated variant calling outperforms external variant calling using modern database management systems.

6.3.1 Disk Storage Consumption

In this section, we describe and discuss the results of our disk storage consumption experiment. First, we present the results regarding disk storage consumption (RQ 4) and disk storage effectively used for genome data (RQ 7). Then, we discuss the results. Finally, we draw conclusions from the experiment results about disk storage consumption in modern database systems.

Results

In Figure 6.1, we depict the disk storage consumption of different databases for genome data in a horizontal bar chart. Thereby, we consider the source flat files (i.e., FASTA and SAM files), the CSV files for bulk loading and the database sizes of the two database management systems MonetDB and SAP HANA. The file size of the source flat files (bottom bar) is nearly six times smaller than the file size of our generated CSV files (second bar from bottom). The database systems consume the most disk storage. It is noticeable that SAP HANA uses only two thirds of the disk storage, which MonetDB uses.

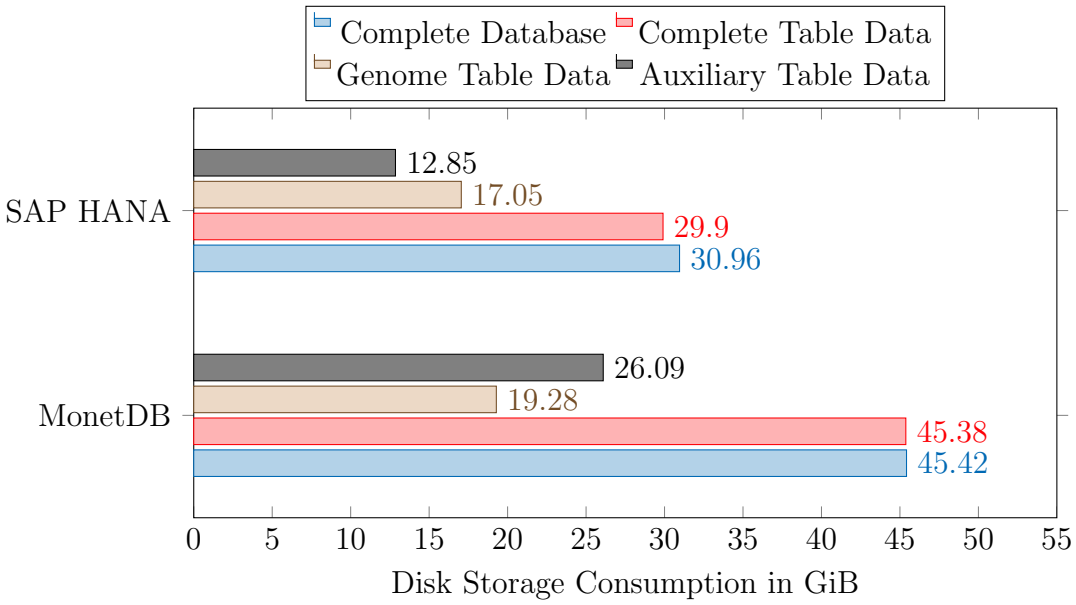


Figure 6.2: Breakdown of disk storage consumption in database systems

In Figure 6.2, we compare the disk storage consumption of the database systems and break it down into disk storage used only for tables defined in our database schema, disk storage used for genome data, and disk storage used for auxiliary data that is created when primary and foreign keys are applied. SAP HANA uses less disk storage to store the data in our database schema than MonetDB. Moreover, SAP HANA uses less disk storage for auxiliary data than for genome data. In contrast, MonetDB uses more disk storage for auxiliary data than for genome data.

Discussion

In Figure 6.1, we presented the disk storage consumption of genome flat files, CSV files and database systems. Although genome flat files and CSV files are ASCII encoded, and therefore human readable, our CSV files consume nearly six times more disk storage than the genome flat files. The reason for this increased storage consumption is that flat files for genome data in FASTA and SAM format use optimized data formats to save storage (e.g., CIGAR strings). Comparing the database systems' disk storage consumption to the CSV files, both database systems need more disk storage. This is unexpected as we assumed that the better data representation inside the database systems (e.g., numbers are stored as integers and not as ASCII encoded strings) saves more storage. But it seems that the auxiliary data that the database systems store (e.g., indexes) outweighs this advantage. Nevertheless, SAP HANA uses only two thirds of the disk storage that MonetDB uses. We relate this result to the circumstance that SAP HANA supports more compression techniques than MonetDB does, and thus stores the same amount of data in more compressed form on disk. The breakdown of storage consumption inside the database systems in Figure 6.2 confirms our assumption from above that the auxiliary data stored by database systems outweighs the more effective

data encoding applied in database systems. MonetDB uses 19.28 gibibyte (GiB) and SAP HANA 17.05 GiB to store only genome data (cf. Genome Table Data in Figure 6.2). Thus, MonetDB uses 10.32 GiB and SAP HANA 12.55 GiB less disk storage than the CSV files that contain the same data. But, the disk storage of both database systems for genome data only is in the same magnitude. Thus, our assumption that SAP HANA applies more compression techniques and therefore save disk storage cannot be confirmed. We relate this result to additional storage overhead that is introduced by compression (cf. Section 6.3.2).

As expected, the main part of the disk storage consumed by the database systems is used for the tables of our database schema (cf. Complete Table Data in Figure 6.2). In case of MonetDB, the ratio between the data amount that is not related to our database schema and the complete database size is less than 0.01% and for SAP HANA it is less than 3.5%. Thus, the additional disk storage consumed by database systems has only little effect on the overall disk storage consumption. But the disk storage used by database systems is not only used to store genome data, but also auxiliary data. To evaluate how much disk storage is effectively used for genome data, we excluded auxiliary data that is created by the database systems when primary and foreign keys are applied. SAP HANA uses 12.85 GiB to store such auxiliary data, whereas MonetDB uses 26.09 GiB. These are 75% and 135% of the disk storage used for genome data only by the respective database system. Thus, SAP HANA uses the consumed disk storage more effectively than MonetDB. We relate this result to different indexing strategies and implementations.

Conclusion

It is not surprising that our database system approach for genome analysis needs more disk storage than flat files as our base-oriented database schema requires to store more information explicitly. Nevertheless, we have shown that disk storage overhead due to DBMS use is small (cf. Figure 6.2) as the main part of the disk storage is consumed by data related to our database schema. But, the effectiveness of using the disk storage is dependent on the chosen database system, and therefore on the implementation (e.g., indexes). To summarize, further improvements are needed to reduce the disk storage consumption (e.g., improvements of the database schema, special compression techniques for genome data, or reduction of auxiliary data).

6.3.2 Compression Effectiveness

In this section, we evaluate the compression effectiveness of modern database management systems (RQ 10). Therefore, we first present the results of our experiment. Then, we discuss the results and summarize our findings.

Results

As already stated in Section 6.2.3.2, MonetDB provides only dictionary encoding on strings, which is not applicable in our database schema as we only store strings of one

#	Table	Column	In-memory size in MiB MonetDB	SAP HANA	Compression Ratio in %
1	HG00096_	read_id	4,122.000	2,662.777	64.599
2	chromosome1	reference_site_id	4,122.000	4,495.983	109.073
3		insert_offset	4,122.000	0.076	0.002
4		base	1,030.635	0.771	0.075
5		base_call_quality	4,122.000	1.758	0.043
6	grch37_	id	950.875	2,614.746	274.983
7	chromosome1	position	950.875	1,782.782	187.489
8		base	237.875	0.003	0.001
9	HG00096_	id	43.125	107.552	249.396
10	chromosome1_reads	mapping_quality	43.125	0.010	0.024
11	overall		19,744.510	11,666.458	59.087

Table 6.2: Compression rates of single columns of SAP HANA compared to MonetDB

byte length. MonetDB uses memory-mapped files to manage data and thus the amount of needed storage for one column in main memory is the same as on disk. Thus, we use MonetDB's column sizes as reference values for SAP HANA's compression capabilities. In Table 6.2, we show the in-memory sizes of all columns that we defined in our database schema and list the according main memory sizes in MonetDB and SAP HANA. In the last column, we show the compression ratio of SAP HANA in percent as ratio between the main memory size in SAP HANA and the (uncompressed) main memory size in MonetDB (cf. Section 6.2.3.2). A compression ratio less than 100% indicates that compression in SAP HANA saves main memory and is therefore effective. A compression ratio greater than 100% indicates that compression leads to overhead, and thus the compression effectiveness of SAP HANA is low. The compression ratio ranges from huge main memory savings (row 3, 0.002%) over smaller effects (row 1, 64.599%) to more than doubled main memory demand (row 6, 274.983%).

Discussion

The huge memory savings (e.g., row 4) are related to the fact of small attribute value domains. For example, there are only 6 possible base values to be stored in our test data: A, C, T, G, N, and X. Thus, dictionary encoding is very effective and in combination with run-length encoding huge savings are generated (cf. Section 2.2.3.2). But dictionary encoding cannot achieve compression and may even increase the data volume in case of very large attribute value domains (e.g., unique integer values or names). Then, the size of the dictionary blows up. This effect can be seen in Table 6.2 in row 6 and 9. For example, table *grch37_chromosome1* comprises 249,250,621 rows as 249,250,621 reference bases must be stored (cf. Section 6.2.2.3). Thus, 249,250,621 distinct *id* values have to be stored. In order to encode 249,250,621 values via dictionary

encoding 28 bit (2^{28} = 268435456) are needed or 3.5 bytes. In contrast to an integer length of 4 bytes, only little main memory can be saved per row, but the dictionary has to be stored, too. The dictionary comprises the original 249,250,621 integer values and their substitutes. Thus, per dictionary entry 7.5 byte are needed (4 byte for the original integer value plus 3.5 byte for the respective substitute), resulting in an overall main memory consumption of 2.741.756.842 byte ($= 11byte * 249,250,621rows$) that is nearly 2.5 times more than the uncompressed in-memory size. Despite these drawbacks of compression, SAP HANA saves overall more than 40% main memory in comparison with MonetDB (row 11). However, when we compare the in-memory size of all columns in SAP HANA that store genome related data with the disk storage consumption of the respective tables from our previous disk storage consumption experiment (cf. Section 6.3.1), we detect that SAP HANA needs more space for the data on disk (17.05 GiB) than in main memory (11.39 GiB). The reason for this is either that SAP HANA compresses data loaded from disk further or that SAP HANA stores additional data on disk that we are not aware now.

Conclusion

We have shown with our experiment that genome data in a base-oriented database schema can be compressed effectively for in-memory usage. We reach memory savings of more than 40% over all genome data. Furthermore, compression ratios of under 1% are impressive. But, compressing every column in spite of attribute domain can lead to higher main memory consumption due to compression overhead. Thus, from our results, a mixture of compressed and uncompressed columns would improve memory consumption further. Nevertheless, better compression algorithms are needed to compress columns with distinct value domains such as identifier columns more efficiently.

6.3.3 Analysis Performance and Scalability

In this section, we evaluate the analysis performance (RQ 5) and scalability (RQ 8) of our database schema implementation on SAP HANA and MonetDB. Therefore, we run the queries to detect potentially deleted, inserted and variant sites presented in Section 5.2.3 over different random ranges (cf. Section 6.2.3.3).

6.3.3.1 Analysis Performance and Scalability on SAP HANA

In this section, we present and discuss the results of our analysis performance and scalability experiment on SAP HANA.

Results

In Figure 6.3, we show the average query runtime for the three different queries on SAP HANA compared to our benchmark runtime derived from bcftools (dashed graph) that is 20% of the runtime of bcftools (solid graph with circle marks) (cf. Section 6.2.3.3). Both axes are log scaled. The five marked points on each graph mark our five chosen

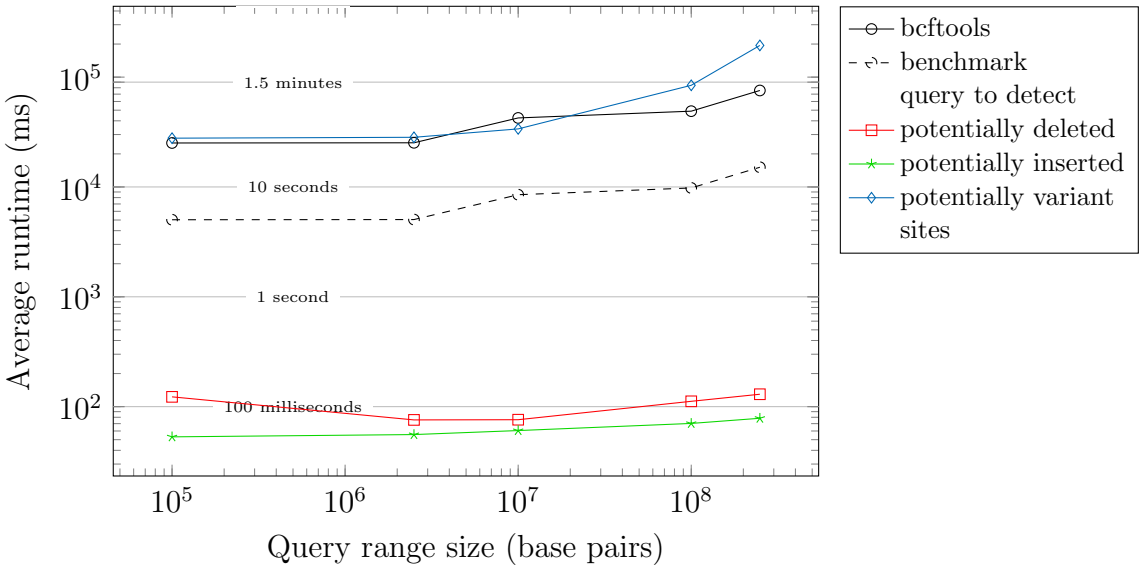


Figure 6.3: Average query runtime on SAP HANA

base pair range sizes: 100,000, 2,500,000, 10,000,000, 100,000,000, and 249,250,621 (complete chromosome 1). The lower graphs depict the average runtime of queries to detect potentially inserted (graph with star marks) and deleted (graph with square marks) sites. The average runtime of detecting potentially inserted sites increases with increasing base pair range size from 51ms for a range size of 100,000 base pairs to 78ms for the complete chromosome. The same increasing trend can be seen for detecting potentially deleted sites, where the average runtime increases from 81ms for a range size of 2,500,000 base pairs to 131ms for the complete chromosome. An unexpected result is the average runtime for detecting potentially deleted sites over range sizes of 100,000 base pairs. The average runtime is 122ms that does not fit into the increasing trend. However, the average query runtime for both queries over all range sizes is always far below our benchmark runtime that indicates our idea of an acceptable query runtime. Detecting potentially variant sites is not possible in acceptable runtime as it nearly lasts at least as long as complete variant calling with bcftools (graph with circle marks). Nevertheless, the query runtime also follows an increasing trend over increasing range sizes.

Discussion

The reason for the increasing trend of query runtime is the increasing size of intermediate results. The greater the range size is, the more data must be processed, and thus the runtime increases. All average query runtimes fit into this increasing trend, except the runtimes for queries to detect potentially deleted sites over ranges with a size of 100,000 base pairs. First, we assumed caching effects that speed up subsequent queries, because we always executed the 100,000 base pair queries before all other queries with greater range sizes. But changing the order did not prevent the effect. Then, we examined the

query execution plans and detected that they are identical for all query range sizes. To further investigate this anomaly, we prepared an experiment, where we increased the range size in 10,000 base pair steps starting from a range size of 10,000 base pairs and ending at 200,000. We depict the average runtime for these queries in Figure 6.4. The

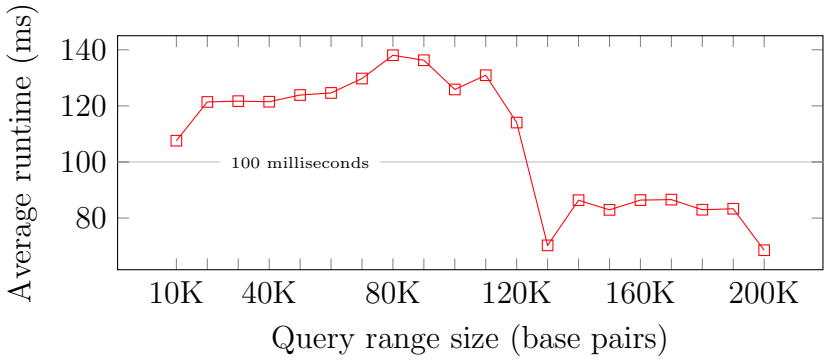
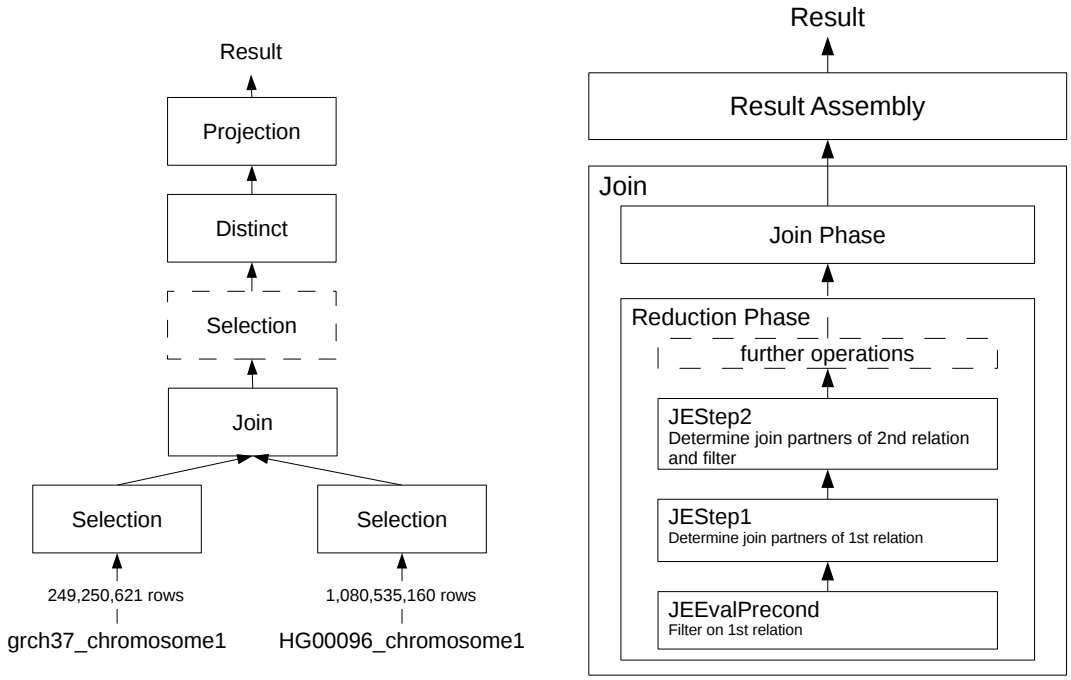


Figure 6.4: Average query runtime on SAP HANA for queries to detect potentially deleted sites in ranges with sizes between 10,000 and 200,000 base pairs

graph shows a remarkable drop of the average query runtime at a range size of 130,000 base pairs. The average query runtime over ranges with sizes smaller than 130,000 base pairs are always less. We discussed this anomaly with our contact at SAP and revealed that the SAP HANA DBMS runs on a virtualized machine. At the end of our thesis, we got access to a SAP HANA instance on real hardware. When we repeated this experiment, we were not able to reproduce the anomaly. Thus, we relate this anomaly to the circumstance that we run our evaluation on virtualized hardware.

In order to explain why the average runtime of queries to detect potentially variant sites differs that much from the other queries to detect potentially inserted or deleted sites, we analyzed the logical and physical query plans for all queries. In the following, we refer to the physical query plan as execution plan. We depict the logical query plan in Figure 6.5(a). The logical query plan is nearly the same for all queries. First, selections are performed on the tables to reduce the number of tuples that must be joined. For example, the range predicate is applied on table *grch37_chromosome1*. Then, the join is computed between all rows that matched the selection predicates. In case of detecting variants an additional selection is made after the join in order to detect potentially variant sites. Finally, the distinct results are determined and the required projection is performed. Such a logical query plan is translated by a DBMS into an execution plan (cf. Section 2.2.2). In Figure 6.5(b), we depict a simplified execution plan that is derived from the original execution plans generated by SAP HANA . The execution plans for the different queries are also similar. Exemplary execution plans generated by SAP HANA can be found in the appendix in Chapter A. The query execution plan consists of a *join step* and a *result assembly step*. Each step can have multiple substeps. In the *result assembly step* the result is assembled based on the intermediate results from the *join step*. At least the final projection is done in this step. The *join step*



(a) Logical query plan (b) Simplified execution plan

Figure 6.5: Logical query plan and simplified execution plan in SAP HANA

consists of a *reduction phase* and a *join phase*. In the *reduction phase*, all rows are determined that are needed in the *join phase*. Therefore, the selections from the logical query plan are performed. Then, in the *join phase*, the corresponding join partners are joined together. Our first assumption was that the more complex join condition in our query to detect potentially variant sites leads to higher query runtime due to processing overhead in the *join phase* of the *join step*. But the analysis of the execution plans shows that the *reduction phase* inside the *join step* is most expensive. For an exemplary query to detect insertions in a 100,000 base pair range, the *reduction phase* takes 70% of the complete execution time of 71ms (cf. Figure A.1 in Chapter A). For an exemplary query to detect potentially variant sites over a 100,000 base pair range, the *reduction phase* takes nearly 97% of the complete execution time of 27 seconds (cf. Figure A.3 in Chapter A). Thus, we examined the *reduction phase* in more detail. SAP HANA starts the *reduction phase* with a so called *JEEvalPrecond* step. In this step, all rows of the first table are filtered and only those rows are selected for further processing that match the given predicate. In the *JEStep1* step, SAP HANA determines the join partners of all rows from the previous *JEEvalPrecond* step. Then, in step *JEStep2*, SAP HANA reuses the intermediate results to compute the needed join partners in the second table. Thereby, the rows are also filtered by the given predicate on the second table. Depending on the concrete query, SAP HANA performs further processing steps.

One factor that influences the runtime of a query is the number of rows that must be processed in each execution step, the intermediate results. If the number of rows is low,

Query to detect potentially	Predicate	# Result Rows	Selectivity in %
deleted sites	s."base" = 'X'	224,906	0.021
inserted sites	s."insert_offset" > 0	164,024	0.015
variant sites	s."insert_offset = 0 AND s."base" <> 'X' AND s.base <> 'N'	1,080,130,144	99.963

Table 6.3: Number of rows of table HG00096_chromosome1 matching different predicates

the single steps execute fast. But when the number of rows is high, processing becomes more expensive. The number of rows to process can be derived from the selectivity of an operation such as a selection. The selectivity is the ratio between the number of rows that pass an operation (i.e., match a given predicate in case of selections) and the number of all rows in a table. If the quotient is low, only few rows have to be processed further, and thus the selectivity is high. But if the quotient is close to 1, nearly all rows of a table have to be processed further, and thus the selectivity is low.

In our experiments, SAP HANA always starts the *reduction phase* with table *HG00096_chromosome1* that contains 1,080,535,160 rows. In step *JEEvalPrecond* the rows are determined that match the given predicate and must be processed further in step *JEStep1*. Thus, the selectivity of the predicates influences the execution time of step *JEStep1*. If the selectivity is high, only a few rows must be processed. If the selectivity is low, many rows must be processed. In Table 6.3, we show the number of rows of table *HG00096_chromosome1* that match the corresponding predicates of our different queries. In the last column, we show the according selectivity in percent. The queries to detect potentially inserted and deleted sites have predicates with a high selectivity, and thus the number of rows that are piped to *JEStep1* is small and the join partners are determined fast. In case of the query to detect potentially variant sites, the predicate has a low selectivity, and thus nearly all rows have to be processed further. That is why a query to detect potentially variant sites has a high average runtime even on a small range size. When the range sizes increase, more rows from table *grch37_chromosome1* must be processed in step *JEStep2*. Thus, the runtime increases even more as more data must be processed in downstream steps such as *JEStep2*. A solution to this problem would be to start the execution with table *grch37_chromosome1* as the intermediate result size to process in *JEStep1* depends on the range size of the query. Thus, at least on small range sizes, the query runtime should improve.

Conclusion

Our experiment revealed that the query runtimes scale over the complete chromosome 1 and acceptable query runtimes are possible, except for our queries to detect potentially variant sites. The reason for this is unfavorable query execution that leads to huge intermediate results when selection predicates have a low selectivity. Furthermore, our

detected query runtime anomaly for queries to detect potentially deleted sites shows that virtualized hardware can influence query execution in unexpected ways.

6.3.3.2 Further Experiment on Analysis Performance and Scalability on SAP HANA

As the query runtimes for queries to detect potentially variant sites from the previous experiment are not satisfying, we conducted a further experiment. This time, we used a database schema without any primary or foreign keys which we already used in our disk storage consumption experiment (cf. Section 6.2.3.1). As query optimizers use a lot of database statistics and schema information to optimize query plans, we assume that the missing primary and foreign keys lead to other execution plans. Our assumption is that the SAP HANA optimizer now generates execution plans depending on the size of intermediate results as no information about potential join tables is available. That should lead to a solution, we stated above, that SAP HANA starts execution with table *grch37_chromosome1*.

Results

In Figure 6.6, we show the results when running the three different queries on the database schema without primary and foreign keys. The average runtime of queries

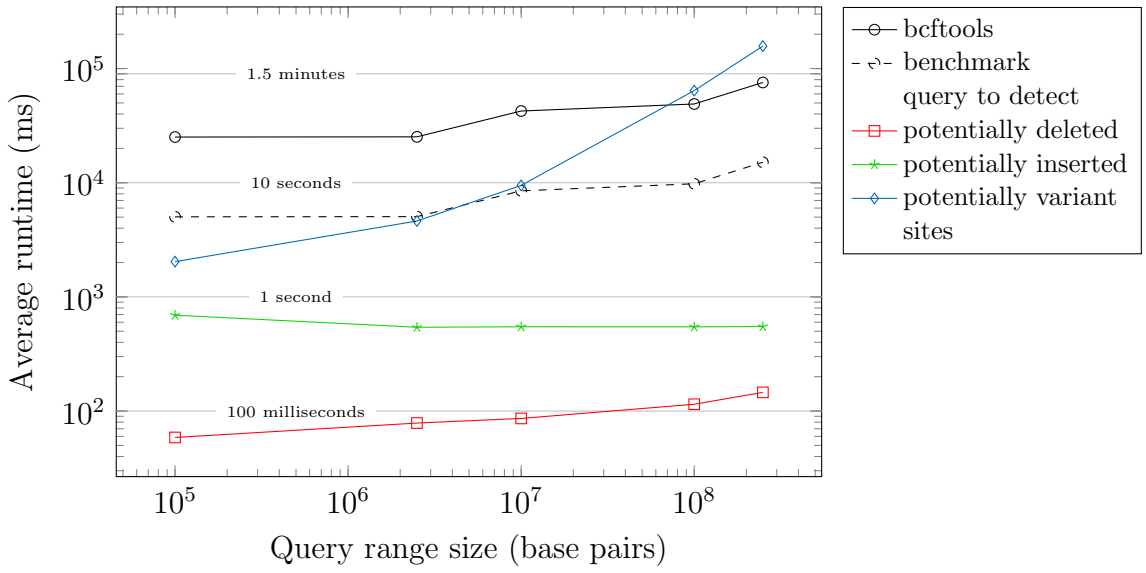


Figure 6.6: Average query runtime on SAP HANA w/o primary and foreign keys

to detect potentially deleted sites is similar to the average runtime in our previous experiment. But the average runtime of queries to detect potentially inserted sites is slower, but still in an acceptable magnitude. A remarkable result is that the average runtime of queries to detect potentially variant sites increases with increasing base pair range size from 2 seconds for a range size of 100,000 base pairs to 2 minutes and

37 seconds for the complete chromosome. Thus, the average query runtime is now acceptable for queries with range sizes lower than 10,000,000 base pairs regarding our definition of acceptable query runtimes (cf. Section 6.2.3.3).

Discussion

The reason for the improvement of average runtime of queries to detect potentially variant sites is a difference in the execution plans. As we have assumed, SAP HANA decides to start the *reduction phase* with table *grch37_chromosome1* (cf. Figure A.4 in Chapter A). Thus, the result size of the *JEEvalPrecond* step is at most as high as the range size and the number of rows to process in *JEStep1* are much less than 1,080,130,144 in the experiment before. Main bottleneck becomes now step *JEStep2* that computes the join partners in table *HG00096_chromosome1*. With increasing base pair range size the execution runtime also increases as the size of intermediate results increases (cf. Figure 6.6). Another effect of using a database schema without primary and foreign keys is that the anomaly that we detected in the experiment before disappears. A big drawback of this approach is the loss of referential integrity as no primary and foreign keys exist. Furthermore, the use of a database schema without any keys does not affect all queries positively. The average runtime of queries to detect potentially inserted sites is higher than before. The analysis of execution plans for queries on different range sizes confirms our assumption that the query optimizer selects a table for starting the *reduction phase* by expected intermediate result size, because when the range size is lower than or equal 160,000 base pairs the optimizer decides to start with table *grch37_chromosome1* as only 160,000 rows have to be processed in the next step. But for range sizes greater than 160,000 base pairs the optimizer decides to start with table *HG00096_chromosome1* as only 164,024 rows have to be processed in the next steps (cf. Table 6.3). The main bottlenecks are in either case the access to table *HG00096_chromosome1*. We assume that due to missing primary keys also indexes are missing that could otherwise speed up the access as in the previous experiment. For queries to detect potentially deleted sites, the range size border to decide with which table to start the *reduction phase* is around 220,000 base pairs. In contrast to queries to detect potentially inserted sites, queries to detect potentially deleted sites do not suffer from missing keys, because access to all tables is nearly as fast as in the previous experiment. Because we have no access to the source code, we cannot examine the reasons for the different access times further.

Conclusion

This experiment revealed that the detection of potentially variant sites is also possible in acceptable runtime for range sizes lower than 10,000,000 base pairs. The big drawback of this solution is the loss of referential integrity. Moreover, the query runtimes of queries to detect potentially deleted and inserted sites are still in an acceptable magnitude, although slightly slower in case of detecting potentially inserted sites.

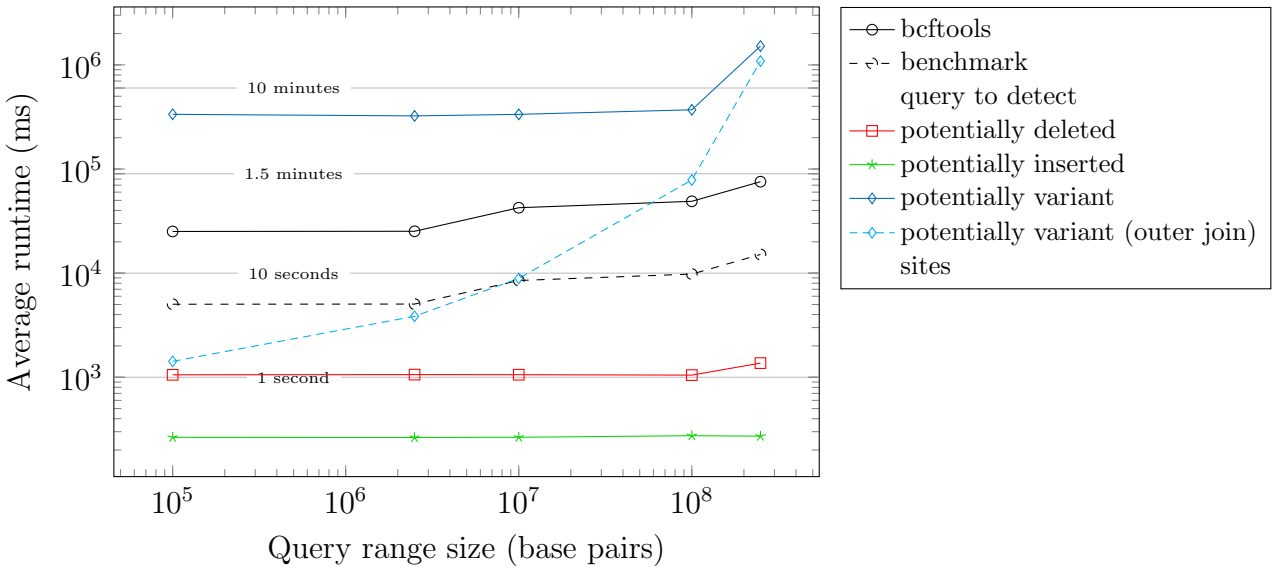


Figure 6.7: Average query runtime on MonetDB

Supplement

At the end of our thesis, we got access to a newer version of SAP HANA (1.00.63.381310) that runs on real hardware. We tested our queries to detect potentially variant sites on that version of SAP HANA with both database schemas. Thereby, we found out that the optimizer has been improved as it now generates the same execution plans no matter what database schema is used (i.e., with or without primary and foreign keys). For that reason, using a database schema without referential integrity is not needed to run the queries to detect potentially variant sites in acceptable runtime.

6.3.3.3 Analysis Performance and Scalability on MonetDB

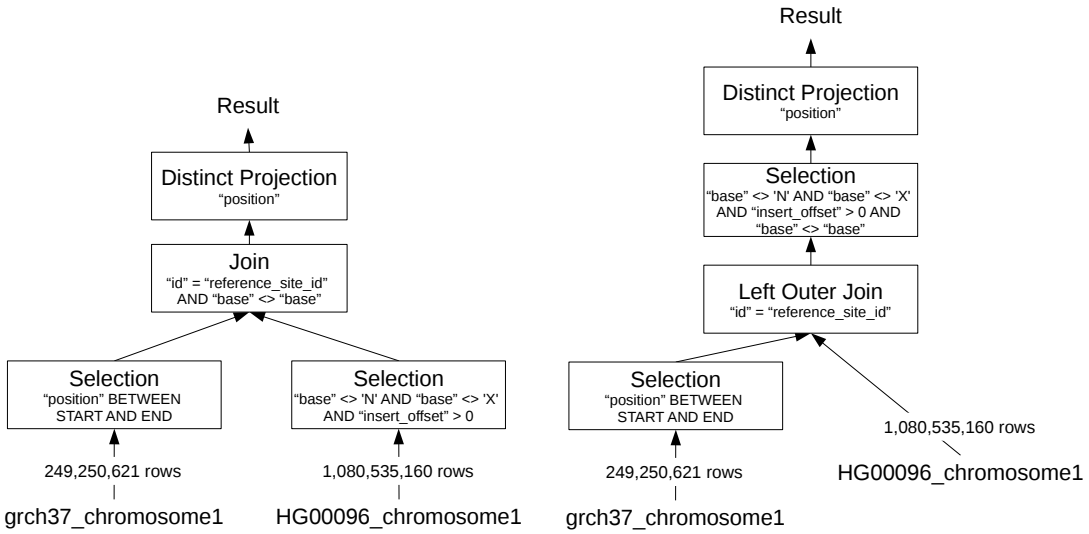
In this section, we present and discuss the results of our analysis performance and scalability experiment on MonetDB.

Results

In Figure 6.7, we show the experiment results when using MonetDB to implement our database approach for variant calling. The average runtime for detecting potentially inserted sites (graph with star marks) is 265ms for ranges with a size of 100,000 base pairs and increases to 272ms for the complete chromosome 1. The average runtimes for detecting potentially deleted sites (graph with square marks) are higher (1 second for 100,000 base range size and 1.3 seconds for the complete chromosome 1), but still in an acceptable magnitude. Detecting potentially variant sites (solid graph with diamond marks) is not possible in acceptable query runtime as it takes longer to determine potentially variant sites than to call variants with bcftools (graph with circle marks).

Discussion

The increasing trend of the average query runtime for queries to detect potentially deleted and inserted sites is related to increasing intermediate results. In order to explain the huge query runtime differences between queries to detect potentially variant sites and the other two queries, we analyzed the logical query plans. In Figure 6.8(a), we depict an exemplary logical query plan for a query to detect potentially variant sites. As can be seen, the selections are pushed down and performed before the join



(a) Logical query plan for inner join query (b) Logical query plan for outer join query

Figure 6.8: Logical query plan to detect potentially variant sites on MonetDB

operation. The logical query plan looks similar for the queries to detect potentially inserted and deleted sites. The join condition of the queries to detect potentially inserted and deleted sites does not contain the condition on unequal bases. Furthermore, the selection predicates on table *HG00096_chromosome1* are different. These different predicates are the main cause for the huge differences in query runtime as they have different selectivity (cf. Table 6.3. The selectivity of the selection predicate is high in queries to detect potentially inserted and deleted sites, and thus only a few rows of table *HG00096_chromosome1* must be processed further. In queries to detect potentially variant sites, the selectivity is very low, and thus nearly all rows of table *HG00096_chromosome1* must be processed. With increasing range sizes the number of rows of table *grch37_chromosome1* to process further also increases. Thus, the average runtime of all queries increases with increasing range size.

In order to perform the queries to detect potentially variant sites on MonetDB faster, the query execution must be improved (e.g., by reducing the size of intermediate results). Thus, we repeated our experiment on a database schema without primary and foreign keys, because we assumed a similar effect as in SAP HANA that lead to faster query

```
SELECT DISTINCT r."position"
FROM "reference_genome" r LEFT OUTER JOIN "sample_genome" s
ON s."reference_site_id" = r."id"
WHERE s."insert_offset" = 0 AND s."base" <> 'X'
AND s."base" <> 'N' AND r."base" <> 'N'
AND s."base" <> r."base";
```

Listing 6.1: Query to detect potentially variant sites using a *left outer join*

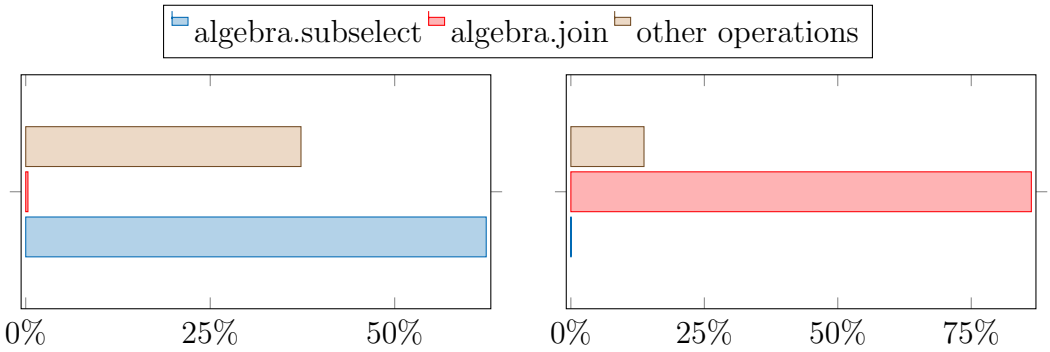
execution. In contrast to SAP HANA, this had no effect on query runtime in MonetDB as the average query runtimes of all queries were nearly the same. Then, we modified our query to detect potentially variant sites and exchanged the *inner join* with a *left outer join*. This modification bases on our observation that MonetDB generates different logical query plans for queries using a *left outer join* and for queries using an *inner join*. Beside the different join operators, the differences concern mainly the execution order of selections. Thus, we wanted to leverage this behavior to optimize our query to detect potentially variant sites by reducing the size of intermediate results. The exchange of the *inner join* with the *left outer join* in our queries to detect potentially variant sites was not straightforward by simply exchanging the respective SQL expressions. We also had to modify the JOIN and WHERE predicate. We illustrate the query to detect potentially variant sites using a *left outer join* in 6.1. The result of a *left outer join* always contains all records of the left table, even if the join predicate does not find any matching record from the right table. Thus, the join predicate is treated differently in a *left outer join* and an *inner join*. This difference could yield wrong results if we use the same join predicate as for an *inner join*. The join predicate for our query to detect potentially variant sites using an *inner join* looks as follows: s." reference_site_id " = r."id" AND s."base"<> r."base" (cf. Section 5.2.3). If we use this join predicate for a *left outer join*, we would retrieve wrong results. For instance, if all bases at a specific genome site in our sample genome are equal to the respective base in the reference genome, the *left outer join* would retrieve this site as potentially variant site. The reason for this is that the *left outer join* operator finds no record in the right table that matches the complete join condition inclusive the unequal predicate of bases, and thus returns the respective site. In order to prevent such erroneous results, we modified the join predicate. We removed the unequal predicate of bases from the join predicate and put it into the WHERE clause. Thus, we retrieve the same results as with an *inner join* as the unequal predicate of bases is not evaluated during the *left outer join* computation.

In Figure 6.8, we show the query plans generated by MonetDB for the query to detect potentially variant sites using an *inner join* (Figure 6.8(a)) and the query using an *left outer join* (Figure 6.8(b)). As depicted, the selection on table *HG00096_chromosome1* is either performed before the join computation or after it. In case of using a *left outer join*, the selection on table *HG00096_chromosome1* is performed after the join, and thus only rows that passed the join operator must be filtered. When the range size is low, the number of rows that pass the join operator is limited by the number of rows

that were selected from the left table, in our case *grch37_chromosome1*. Thus, less rows must be filtered in the subsequent selection step on table *HG00096_chromosome1*. The result of the query modification is an improved average query runtime. For example, with the modified query the detection of potentially variant sites in a range with a size of 100,000 base pairs lasts 1,4 seconds instead of more than 5 minutes. Thereby, the average runtime also follows an increasing trend as for the queries to detect potentially deleted and inserted site (cf. dashed graph with diamond marks in Figure 6.7). At least for range sizes less than 10,000,000 base pairs, the average runtime of our modified query to detect potentially variant sites is in our acceptable query runtime region of 20% of the average runtime of bcftools (dashed graph with circle marks in Figure 6.7).

Operation	Inner Join	Left Outer Join
algebra.join	12,675.537	10,805.976
algebra.subselect	2,273,472.154	5.008
other	1,357,806.958	1,716.302
Overall	3,643,954.649	12,527.286

(a) Processing times of low level operators regarding inner join and left outer join query in ms



(b) Proportion of overall processing time for single low level operators in inner join query in %

(c) Proportion of overall processing time for single low level operators in left outer join query in %

Figure 6.9: Execution times for operations of an exemplary query to detect potentially variant sites in a 100,000 base pair range on MonetDB

In order to confirm our assumption that the query runtime improved due to reduced size of intermediate results, we analyzed the processing time of low level operators. In Figure 6.9(a), we list the aggregated CPU processing times of low level operators in MonetDB regarding the two different query approaches over a range size of 100,000 base pairs. Thereby, the *algebra.join* operator computes a join and the *algebra.subselect* operator performs selections. Comparing the overall processing times, the query using a *left outer join* outperforms the query using the *inner join*. Thereby, the processing times for the join operations are in the same magnitude of ca. 11 seconds, but the processing

time for selections differs orders of magnitudes. In Figure 6.9(b) and Figure 6.9(c), we compare the proportion of the overall processing time in percent of single low level operators for the *inner join* and *left outer join* query, respectively. The *algebra.subselect* operator needs more than 60% of the complete processing time of the *inner join* query (ca. 38 minutes). In contrast, the *algebra.subselect* operator needs less than 1% of the complete processing time of the *left outer join* query (5 seconds). This huge difference can only be explained by the reduced amount of rows to process during selections when using a *left outer join*. But, with increasing range size, the intermediate result size grows and thus the processing time for the selection on table *HG00096_chromosome1* also increases. This trend can be seen in Figure 6.7 (dashed graph with diamond marks), where the average query runtime of the query using a *left outer join* converges to the average query runtime of the query using an *inner join* with increasing range size.

Conclusion

Our experiment shows that detection of potentially deleted, inserted and variant sites is possible in acceptable runtime on MonetDB. In case of queries to detect potentially deleted and inserted sites the query runtimes scale over the complete chromosome 1. Regarding the detection of potentially variant sites, we had to modify our query in order to achieve a more favorable query execution that leads at last for range sizes lower than 10,000,000 base pairs to an acceptable query runtime.

Over all, the average runtime of the different queries have a similar progression for increasing range sizes as on SAP HANA(cf. Section 6.3.3.1), but are most often higher. We infer the higher average runtime from different hardware setups, whereby MonetDB runs on less cores and with less memory than SAP HANA (cf. Section 6.2.2.2). The similar progression is related to the same effects as in SAP HANA: increasing intermediate result size with increasing query range size. Moreover, as on our SAP HANA installation, it is not possible to retrieve potentially variant sites on MonetDB in acceptable runtime, because of the low selectivity of the selection predicates combined with unfavorable query plans. Thus, currently, modifications of the database schema (SAP HANA) or of the query (MonetDB) are needed.

6.3.4 Integrated vs. External Variant Calling Performance and Scalability

In this section, we describe and discuss the results of our experiment to evaluate the analysis performance and scalability of integrated and external variant calling (RQ 9). Therefore, we developed a command line tool that determines variants in a genome in two different modes (cf. Section 6.2.3.4). In external mode, the tool uses MonetDB to provide necessary genome data and calls variants externally. In the integrated mode, our tools uses a custom aggregation function to call variants inside MonetDB. We test our tool on different ranges sizes (RQ 8) and compare the average runtimes with the average runtimes of bcftools (RQ 5). As a result of our previous analysis performance and scalability experiment on MonetDB, we replaced the *inner join* with a *left outer*

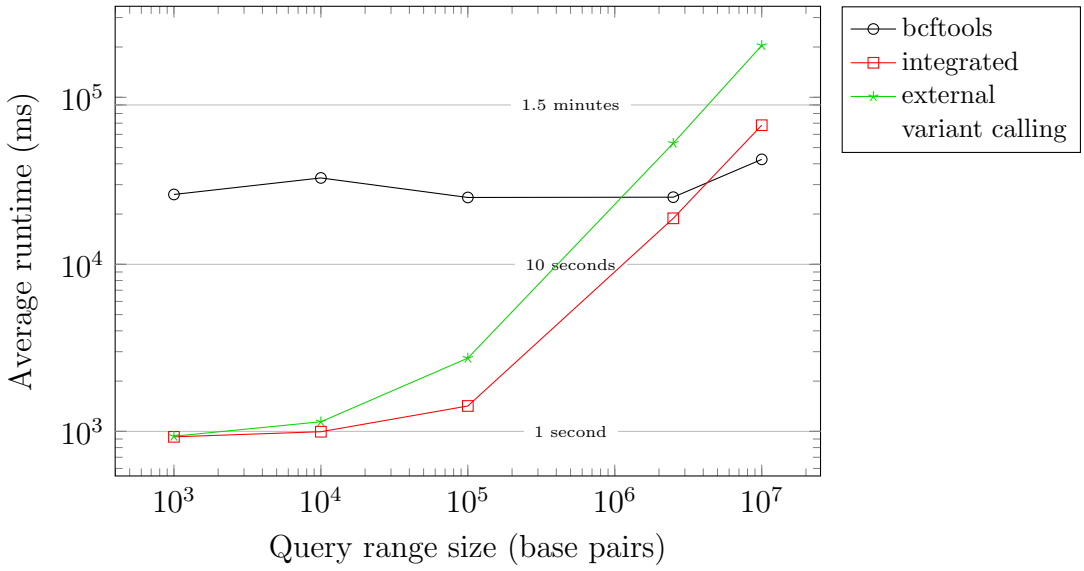


Figure 6.10: Average runtime of integrated and external variant calling

join when appropriate. Otherwise the analysis performance suffers from processing unnecessary huge intermediate results as described in Section 6.3.3.3.

Results

In Figure 6.10, we show the average runtime of our analysis tool in *integrated* (graph with square marks) and *external* mode (graph with star marks) over different range sizes. Both axes are log scaled. In contrast to our experiment description in Section 6.2.3.4, we could not run our analysis tool in *integrated* mode on range sizes above 10,000,000 base pairs, because the execution randomly aborts at more than 20 minutes. We were not able to narrow down the problem, yet, but we assume that the problem is related to our aggregation function and its parallelization. Another explanation for this behavior could be the Linux kernel functionality *Out Of Memory Killer* that abruptly kills processes with excessive main memory consumption in case of main memory shortage.[6] Despite this problem, we were able to collect runtime data of our analysis tool in *integrated* mode for base pair ranges with a size smaller or equal 10,000,000 base pairs. Thus, we depict only the average runtime results for ranges with a size up to 10,000,000 base pairs.

The average runtime of our tool in *integrated* mode is always below the average runtime of our tool in *external* mode. At a base pair range size of 10,000,000, both modes of our tool need more time to call variants than bcftools. In between, at a range size of 2,500,000 base pairs, our tool is only in *integrated* mode faster than bcftools.

[6] http://www.monetdb.org/Documentation/OOMkiller

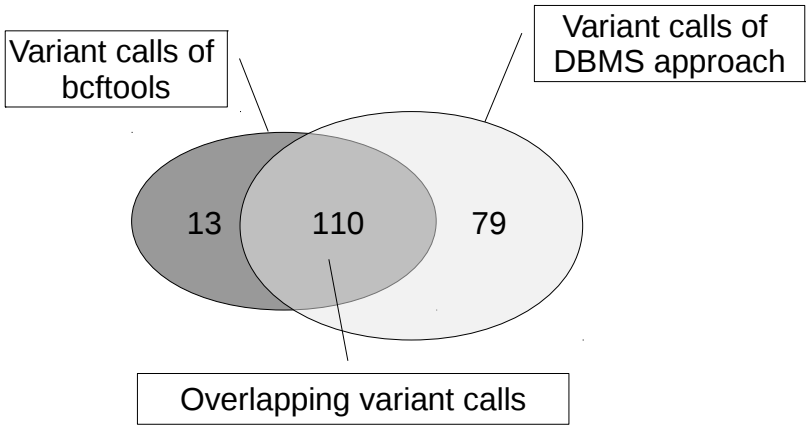


Figure 6.11: Venn diagram for single nucleotide variant calls of bcftools and the DBMS approach on a sample base pair range of size 100,000

Discussion

The reason that our tool in *external* mode is always slower than in *integrated* mode is that in *external* mode the amount of data that must be transferred to our tool is much higher than in *integrated* mode. In *integrated* mode, the database management systems aggregates the data, computes the result of variant calling, and delivers it. But in *external* mode, the database management system transfers all data needed for variant calling in a specific range to the tool. In fact, the time the tool waits in *external* mode to finish the data transfer is higher than in *integrated* mode. Moreover, the tool in *external* mode has to compute the result, whereas in *integrated* mode, the tool just receives the result. Thus, the transfer of requested data and processing it takes much more time than to perform the variant calling directly inside the database system and transferring the result. This also explains, why the average runtime difference of the two different modes is much less at smaller base pair ranges, because the amount of data to transfer is smaller than on greater range sizes and thus can be transferred faster. But, the subsequent computation in *external* mode outweighs this advantage.

Compared to the average runtime of bcftools, the *integrated* approach allows to analyze ranges with a size up to 2,500,000 base pairs faster than bcftools. Furthermore, the performance of our approach scales with increasing range size as the performance decreases. In contrast, bcftools does not show such a trend and the runtime is also high on small range sizes. But, when we interpret the analysis performance, we have to keep the result quality in mind. Although, it is out of our thesis's scope to evaluate the result quality of variant calling on modern database systems, we involve it into this result discussion. In Figure 6.11, we show a Venn diagram for one exemplary genome range with a size of 100,000 base pairs. The Venn diagram shows how many variant calls of bcftools and of our DBMS approach for variant calling on this genome region overlap and how many are unique to each tool. Our DBMS approach that uses simple cut-off rules to determine genotypes and variant calls returns 110 variant calls of the

123 of bcftools. Only 13 are not called. If we assume that bcftools calls all real variants, then our simple approach for variant calling returns 89.4% of the real variants. But the precision of our approach is low as it detects 79 variants that are not recognized by bcftools. Thus, under the assumption that bcftools always delivers all real variants, the proportion of correctly called variants regarding all called variants of our approach is only 58.2% and therefore four of ten variant calls of our approach are wrong. In order to improve the precision, the implementation of more sophisticated variant calling algorithms is needed. But such algorithms could increase analysis runtime.

Conclusion

With the experiment, we have shown that it is possible to perform variant calling on modern database systems faster than with our benchmark tool bcftools, but only on base pair ranges below 2,500,000 base pairs and when we ignore the result quality. Furthermore, our results indicate that integrated variant calling computation is faster than external variant calling. The performance benefit increases with increasing range size. On small base pair range sizes, both approaches have similar performance, but with increasing range sizes the integrated approach outperforms the external one. The difference would even increase, if the tool is connected via network to the database system, because of higher transfer times.

6.4 Threats to Validity

In this section, we discuss threats to validity of our evaluation results and our approach for variant calling on modern database systems. Thereby, we distinguish threats to internal and external validity. Internal validity describes the extent to which our gathered results are valid. External validity reflects the extent to which our findings and our approach for variant calling on modern database systems are generalizable and applicable.

6.4.1 Threats to Internal Validity

In this section, we state internal threats to validity that affect the storage and performance results of our evaluation. Thereby, we discuss the impacts of virtualized hardware that we used to evaluate our approach on MonetDB. Furthermore, we describe how we addressed confounding factors of our storage and performance measurements. Moreover, we describe how we ensured comparable results for our analysis performance measurement.

Hardware Setups

We use different hardware to run the two database management systems (cf. Section 6.2.2.2). Because it is not our goal to compare the performance of the two database management systems directly, we can use different hardware setups in order to evaluate how fast and scalable analysis on modern database systems is. Regarding our disk storage consumption experiment, we expect no influence of different hardware setups.

Virtualization of Hardware

Virtualization of hardware always leads to a performance degradation as an additional layer of abstraction is introduced that increases processing efforts. Thus, the results collected from our experiment with MonetDB could be worse than they would be on non-virtualized hardware.

DBMS Configuration

DBMSs can have a lot of configuration options to adjust them to special application environments. We used predefined standard configuration options for MonetDB as well as SAP HANA. We compiled MonetDB with configuration options proposed by the developers that should be use for performance measurements (cf. Section 6.2.2.1). In case of SAP HANA, we used an instance that was preconfigured by SAP AG. We assume that the system is optimally configured for performance testing as we have no direct control over the configuration.

Self-Optimizing Systems

Both DBMSs apply self-optimization techniques to tune the system performance. MonetDB, for example, decides at runtime whether to create additional indexes to speed up certain queries or not. The user has not direct control over indexes or their usage. Thus, MonetDB could create additional indexes at runtime as it decides that this would be beneficial. Such index creation influences query performance as well as storage consumption. Query performance is influenced as queries run faster. Storage consumption is influenced as MonetDB writes additional data for indexes to disk as it uses memory-mapped files. We observed that MonetDB's storage consumption increases during runtime, but drops after restart. Thus, we measured storage consumption after a restart. But this decreases the validity of our storage consumption experiment to a freshly started MonetDB system. Moreover, SAP HANA decides during runtime and depending of the data distribution what compression technique to use. Thus, query performance and memory consumption is influenced. Query performance can lack due to additional decompression overhead depending on the compression technique. Memory consumption is influenced as different compression techniques have different effectiveness.

Confounding Variables

Many factors can influence performance measurements. Such factors are cold or warm memory, unexpected CPU load, or network communication. In Section 6.2.1, we described measures to address such confounding factors. For example, we ran warm up queries to avoid queries to run on cold memory. Furthermore, we ran several measurement queries instead of just one to minimize the effect of unexpected CPU load. Moreover, we only considered the processing time on the DBMS server when evaluating query performance and excluded time for result delivery and rendering.

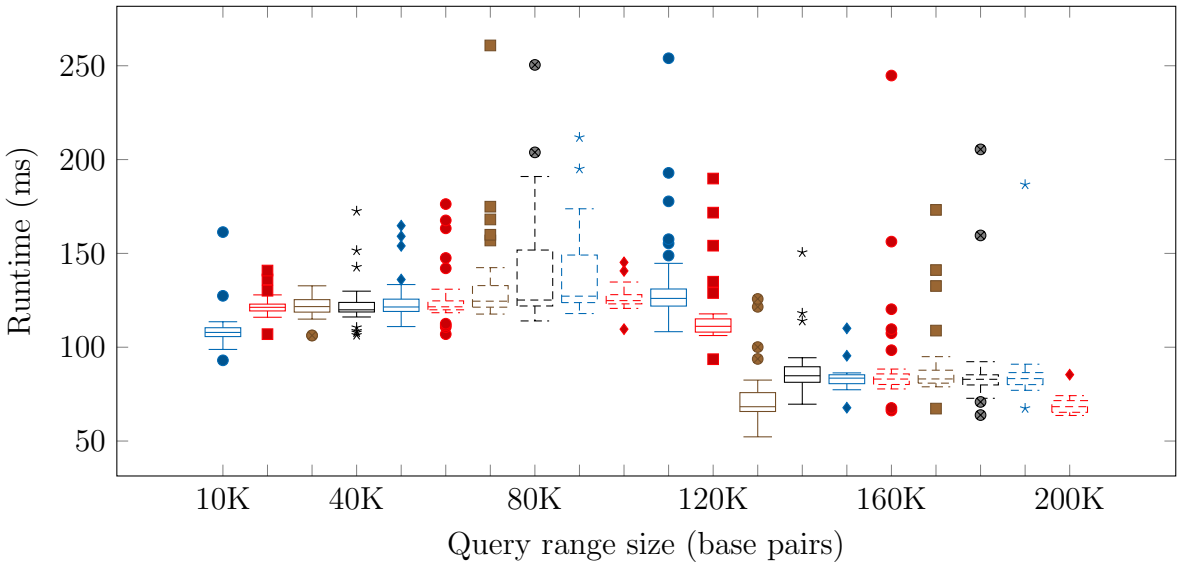


Figure 6.12: Box plot for runtime of queries to detect potentially deleted sites on SAP HANA (30 queries per base pair range size between 10,000 to 200,000 base pairs in 10,000 base pair steps)

In case of unexpected measurements, we used statistical means to determine whether our results really contain anomalies. In order to check whether the unexpected query runtime differences on SAP HANA when detecting potentially deleted sites (cf. Figure 6.4) have a statistical foundation, we used box plots. In Figure 6.12, we show the box plot of query runtime for queries to detect deletions over ranges between 10,000 and 200,000 base pairs. The definite impact of range sizes is confirmed by this plot as the lower whiskers of queries on base pair ranges smaller than 130,000 are always above the upper whiskers of queries on base pair ranges greater or equal 130,000. The range between the whiskers include all values except outliers. For most box plots, the ranges between the whiskers are small and thus we relate the outliers to unexpected CPU load or other confounding variables.

Implementation of Algorithms

The way an algorithm is implemented heavily influence performance. For example, an algorithm implementation that excessively uses function calls can lead to weak performance as the CPU is busy to handle the call stack. An optimization performed by compilers is in-lining of such function calls. Thus, we used equally optimized implementations of our aggregation function and the external variant calling algorithm by reusing certain code fragments and following an imperative implementation schema. Furthermore, we compiled our tool with gcc 4.7.3[7] using the optimization options *-O3* and *-fomit-frame-pointer*[8] that we also used to compile MonetDB.

[7]http://gcc.gnu.org/

[8]For more information about these options we refer the interested reader to http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

6.4.2 Threats to External Validity

In this section, we explain threats to external validity that limit the generizability of our results. Therefore, we cover the test data used for our evaluation and the implemented algorithm for variant calling.

Test Data

For our experiments, we used human genome data with low coverage. Coverage means how many times each sample genome site is sequenced. The more reads cover a certain genome site, the more base values are available for further analysis. On the one hand, this can improve analysis results as more data is available, for example, to determine a genotype. On the other hand, more data has to be stored and processed. The use of low coverage genome data is common in genome sequencing projects such as the 1000 Genomes Project [McV12]. Thus, the use of low coverage data for our evaluation is practically relevant, although the impact of high-coverage data cannot be derived from our experiments. Furthermore, we only used sequencing data for chromosome 1 of one human sample genome. One way to analyze genome data is chromosome by chromosome. Moreover, chromosome 1 is the biggest human chromosome comprising nearly 250 million base pairs. Complete genomes of other organisms are much smaller (e.g., E. coli (K-12) genome comprises 4,639,221 base pairs [BIB+97]). Thus, our results are also applicable to other use cases with other genome data. The use of data of only one sample genome is suitable for our use case of variant calling as we analyze only one sample genome at a time. Further evaluation with genome sequencing data of more samples is needed to examine scalability over data size and to cover further use cases.

Furthermore, we implemented the conversion of sequencing data from the source files into importable CSVs (cf. Section 6.2.2.3) to the best of our knowledge. Thereby, we currently ignore the information stored in the *FLAG* field that gives further information about the sequence alignment (cf. Section 2.1.3.3). Interpretation of information in the *FLAG* field improves data quality of imported data. Hence, we concentrated our evaluation on query performance and disk storage consumption, data quality is not necessary. For future work, we plan to integrate the *FLAG* information.

Limited Genome Analysis Context

Our evaluation was focused our approach for variant calling variant calling on modern database systems. Thus, we cannot simply generalize our analysis performance and scalability results to other analysis contexts such as sequence alignment or downstream analysis. But, we can derive assumptions and methodologies for future work from our evaluation of variant calling on modern database systems.

Variant Calling Algorithm

The variant calling algorithm that we implemented to evaluate integrated and external variant calling performance and scalability is a simple cut-off approach. As far as we

know, the current interfaces to extend MonetDB with custom aggregation functions are limited to functions that aggregate values over one column. Thus, we used the cut-off approach for variant calling that requires at least an aggregation over the base value column (cf. Section 2.1.2.3), although we know that this approach does not perform well regarding result quality on low coverage sequencing data [NPAS11]. In Section 6.3.4, we compared the variant calls in one exemplary genome range of 100,000 base pairs of our approach with bcftools. The comparison reveals that the precision of our approach is low. In order to improve the precision, we have to implement more sophisticated variant calling algorithms. But, the implementation of more sophisticated variant calling algorithms based on probabilistic approaches, for example, would require the aggregation over at least two columns of our database schema as the base values and base call quality values are of interest. Thus, analysis runtime will be affected and therefore, the results of our performance evaluation are limited to our implementation of variant calling. Other alternatives to using custom aggregation functions and SQL queries for variant calling are stored procedures or general user defined functions that perform custom operations on the data. Thus, we also plan the evaluation of stored procedures and user defined functions for variant calling in future work.

Artificial Performance Benchmark

As far as we know, there is no performance benchmark we could use and thus we created one for evaluation. To decide whether the performance of our analysis queries is in an acceptable magnitude, we used a artificial benchmark that we derived from the runtime of bcftools. We determined that 20% of the average runtime that bcftools needs to call variants in a fixed range size is an acceptable runtime for our queries over the same range size. We only assume that the remaining 80% of the average runtime of bcftools are enough to perform further analysis in acceptable runtime. Moreover, other command line tools could process the data faster. Nevertheless, the use of our artificial benchmark does not prevent to get a *first impression* whether using modern database systems for genome analysis leads to acceptable analysis performance or not.

6.4.3 Conclusion Regarding Threats to Validity

We showed threats to validity of our results and appropriate countermeasures to limit or avoid their influence on our results. We conclude that we assured great internal validity as we reduced the impact of confounding variables by repeating the measurements of our performance experiments. Furthermore, we used statistical means to confirm unexpected results such as the average runtime anomaly on SAP HANA for queries to detect potentially deleted sites. Regarding external validity, we showed that our chosen test data is applicable for our use case, although we only use chromosome 1 sequencing data of one human sample genome. Furthermore, we explained that the cut-off approach for variant calling is sufficient for our analysis performance analysis, but we have to use more sophisticated approaches in order to compare result quality that was no objective of our current evaluation. Moreover, we stated that our artificial benchmark limits the external validity of our results.

6.5 Summary

In this section, we summarize the findings of our evaluation of our approach for variant calling on modern database systems. In our evaluation, we considered the disk storage consumption of our database approach for variant calling. Furthermore, we analyzed how effective genome data in our database schema can be compressed. Moreover, we evaluated the performance and scalability of analysis and assessed whether integration of analysis into the database management system outperforms external analysis computation.

With our disk storage consumption experiment (cf. Section 6.3.1), we showed that modern database systems consume more disk storage than flat files commonly used in genome analysis to store genome data. The reason for this disk storage overhead is that our database schema stores data in a more explicit way than the flat files do. Moreover, auxiliary data created by the database management system consumes a large amount of disk storage that is not directly related to genome data. Regarding compression of genome data, we have shown that genome data is a good candidate for compression, mainly due to the limited value domain of attributes (Section 6.3.2). Thus, dictionary encoding or run-length encoding apply well. However, we also revealed that compression should be used carefully when considering attributes that have a distinct value domain such as unique identifiers.

Regarding analysis performance and scalability (cf. Section 6.3.3), we showed that fast querying is possible in acceptable runtime for our queries to detect potentially inserted and deleted sites. At least for range sizes less than 10,000,000 base pairs, the queries to detect potentially variant sites also have an acceptable runtime. However, the database systems do not always create optimal execution plans as the experiments with queries to detect potentially variant sites revealed. Thus, we had to use different join operators (MonetDB) or modify our database schema (SAP HANA) to improve query execution. But, these optimizations do not prevent the processing of one billion values in one column, and thus detecting potentially variant sites on complete chromosomes is a time consuming task.

With our experiment on integrated and external analysis performance and scalability (cf. Figure 6.10), we showed that analysis supported by a modern database system can be faster than using command line tools, except when analyzing huge base pair range sizes, and thus performing analysis on much data as our variant calling tool does. Furthermore, we showed that analysis integration via a custom aggregation function becomes beneficial when the amount of data to transfer for external processing increases. For small amounts of data such as analyzing genome ranges up to 10,000 base pairs in size, analysis integration does not provide performance advantages.

In Section 6.4, we presented threats to internal and external validity of our results. We stated a high internal validity of our experiments due to different countermeasures such as repetition of measurements to reduce the impact of unexpected CPU load. Moreover, we implemented our integrated and external algorithms for variant calling

equally optimized. But, our chosen data set and the implemented algorithm for variant calling limit the external validity of our results, because we only considered low coverage sequencing data and variant calling using cut-off rules in our evaluation. Moreover, our statements regarding acceptable runtimes refer only to an artificial benchmark based on the command line tool bcftools. Nevertheless, our results provide a first insight into the advantages and disadvantages of using modern database systems for genome analysis.

7. Conclusion

In this section, we summarize the outcome of our thesis. First, we recapitulate the motivation and the goal of our thesis. Then, we summarize our contribution. Finally, we conclude and present future work.

Genome analysis is important to detect variances in organisms' genomes that can indicate whether diseases will be expressed or how patients react on drugs. Thus, genome analysis is an important method to improve disease treatment. With increasing amounts of available genome sequencing data due to next generation sequencing techniques, genome analysis tools must be able to handle such huge amounts of data efficiently. In this thesis, we showed whether and how modern database systems can be used for genome analysis as an alternative to state-of-the-art genome analysis pipelines based on flat files. Thereby, we concentrated our research on following research questions:

RQ 1 How can modern database systems address challenges in genome analysis?

RQ 2 How to use modern database systems for genome analysis?

RQ 3 How to store genome data in a modern database system?

RQ 4 Is the disk storage consumption of approaches that use modern database systems for genome analysis competitive to flat file approaches?

RQ 5 Is the analysis performance of genome analysis on modern database systems competitive to state-of-the-art tools?

RQ 6 Is the result quality of genome analysis on modern database systems competitive to state-of-the-art tools?

RQ 7 How much disk storage is effectively used for genome data?

RQ 8 How scalable is genome analysis on modern database systems regarding analysis performance?

RQ 9 Does integrated analysis computation outperform external analysis computation?

RQ 10 How effective is the compression of genome data that is stored in our database schema?

Contribution

First, we identified advantages that modern database management systems provide for genome analysis (RQ 1). The main advantages of using database management systems for genome analysis are the comprehensive data management capabilities of DBMSs. DBMSs provide data integration features to integrate data from different data sources and make it accessible via a well-defined database schema. Furthermore, DBMSs ensure data integrity that is necessary for reliable genome analysis. Further advantages of using DBMSs are data security features to provide fine-grained data access rules. Another advantage is the decoupling of data and applications, the so called logical data independence. Thus, external applications do not have to handle efficient data access themselves but retrieve data from a database system and can perform their computations. Especially modern DBMSs provide means to perform data access and processing even on huge amounts of data fast due to in-memory storage and bulk processing of data. Thus, against the backdrop of ever increasing amounts of genome sequencing data, modern DBMSs are an effective mean to improve genome analysis.

Then, we defined two use cases to use modern database systems for genome analysis (RQ 2). First use case is analysis support, whereby the modern database system is used to manage and query data but analysis is performed externally. Thus, the data management capabilities and efficient data access of database management systems are used. Second use case is analysis integration by extending the database management system with genome analysis capabilities. Thus, analysis can directly be performed on data inside the database and just the result must be delivered to a client. Based on these reflections about how to use modern database systems for genome analysis, we developed an approach to perform variant calling on modern database systems. Therefore, we created a database schema that facilitates compression of genome data by storing genome data in a base-oriented schema instead of storing reads as complete strings as in flat files (RQ 3). Furthermore, we provided query scaffolds to query and analyze the genome data in the database system.

Next, we implemented our approach for variant calling on two different modern database management systems and compared the disk storage consumption (RQ 4). Thereby, our database approach consumes more disk storage than traditional flat file approaches for genome analysis. One reason for this increased disk storage consumption is auxiliary data that is created and stored by the database systems (RQ 7). An analysis of compression ratio of single columns revealed that compression can be highly effective

and at the same time may consume more storage due to compression overhead of compression techniques such as dictionary encoding when compressing columns with a huge distinct value domain (RQ 10). Furthermore, we evaluated the performance of queries on the two different database management systems, namely SAP HANA and MonetDB (RQ 5). The experiments revealed that the query performance on genome data highly depends on the selectivity of predicates and on the amount of data to process. Thus, only our queries to detect potentially deleted and inserted sites run in acceptable runtime over the complete chromosome (RQ 8). The queries to detect potentially variant sites are only in an acceptable runtime magnitude for range sizes less than 10,000,000 base pairs. Moreover, we compared external and integrated variant calling on modern database management systems (RQ 9). Thereby, we showed that analysis integration becomes advantageous when the amount of data to transfer to an external client increases, because time for querying, transferring, and externally processing the data is higher than directly processing the data and transferring the result. But our integrated variant calling implementation that uses a custom aggregation function to determine genotypes must be improved as we currently cannot use it for base pair range sizes greater than 10,000,000 base pairs (RQ 8), because the execution abruptly aborts.

In this thesis, we did not evaluate the result quality of our approach (RQ 6) due to limitations in extending MonetDB (cf. Section 6.4.2).

In the following, we explicitly list the advantages and disadvantages of our approach for variant calling on modern database systems compared to bcftools that uses flat files:

Advantages

- Data management capabilities such as data integration, data integrity and data security are available for the complete genome analysis pipeline
- Data independence allows faster application development as (efficient) data access is provided by DBMSs
- Descriptive query languages such as SQL enable simple and standardized access to genome data
- Scalable analysis computation: time to analyze a genome range increases with increasing range size

Disadvantages

- Increased disk storage consumption
- High effort to integrate analysis computation into DBMSs
- Unacceptable query and analysis runtimes when huge amounts of genome data must be processed (e.g., detect potential variant sites on ranges with a size of 100,000,000 base pairs)

Future Work

The contribution of our thesis is a small step towards genome analysis on modern database systems. With our approach for variant calling, we showed potentials of modern database systems for genome analysis. We will use our approach for future work and improve it.

A big future task is the implementation of more sophisticated variant calling algorithms and the evaluation of result quality. Thereby, the integration of more complex variant calling algorithms is challenging, because data from different columns (i.e., base values and base call quality values) is needed to compute a genotype using a probabilistic approach. Stored procedures seem to be a promising approach to implement more complex variant calling algorithms. Moreover, extending the DBMS with aggregation functions that aggregate over two columns are further interesting approach to solve the problem. Thereby, the further exploitation of modern processing technologies (e.g., cache-consciousness algorithms, parallelization, and SIMD (single instruction, multiple data) operators) and devices (e.g., GPUs) plays an important role in order to achieve similar result quality in genome analysis as state-of-the-art tools while maximizing the analysis performance.

Another future research task will be to further reduce disk storage consumption of our database approach for genome analysis as it is currently rather high compared to flat file approaches. The reasons for this disk storage overhead are limited compression on huge distinct value domains as well as additional auxiliary data that is stored by the database systems. Approaches to reduce the disk storage consumption are more efficient compression strategies and techniques, but also alternatives to our database schema that store data more implicit as flat files do.

Moreover, our approach is currently limited on variant calling. To provide an competitive alternative to existing genome analysis pipelines, we have to examine how to extend our approach to support sequence alignment and downstream analysis, and furthermore how to integrate our approach into existing work flows. Approaches to integrate sequence alignment could be specialized import procedures that compute the alignment externally and store the result in the database or hybrid approaches where the database system directly works on flat files that store the alignment data as presented by Röhm et al. [RB09]. Another approach could be to integrate sequence alignment like variant calling and compute it on demand. To implement such an integrated approach for sequence alignment, fast data processing and appropriate alignment algorithms are needed. Regarding downstream analysis, extensions to our database schema are required in order to store more genome related data such as molecular interactions, functional annotations of genes, and biological ontologies. The integration of gene annotation data could improve query processing by allowing to specify to call variants on a region of a specific gene. Currently, our database schema lacks such information, and thus we select random ranges for evaluation. Another important topic is work flow integration. Thereby, the question is how to integrate existing tools, especially for downstream analysis, into a work flow that uses a modern database system for genome data management. One

solution could be that the database system provides legacy interfaces such as flat file exports. Another approach would be to migrate the existing tools to use standard query languages of databases such as SQL to access the data (cf. [LPW+06]) or to provide specialized application programming interfaces (cf. [SHX+05]).

Furthermore, we have to evaluate the multiple-user performance of our approach and the respective DBMSs used for implementation. DBMSs are designed for multiple-user access and thus should outperform state-of-the-art tools. A further research questions that arises is how well does an approach based on modern database systems scales on increased amounts of genome data. Instead of storing one human sample chromosome, we want to store hundreds of complete genomes and evaluate the analysis performance and scalability. Thereby, caching strategies will become important in order to provide fast access to different data that is needed by many users if main memory is limited. One strategy could be to analyze workloads and derive data that is most promising to be accessed by users and therefore should be cached (e.g., genome data that encodes genes). Moreover, advanced compression techniques will become important to use limited main memory efficiently.

Answering these future research questions will enable us to better understand and justify the usage scenarios of modern database systems for genome analysis.

A. Appendix

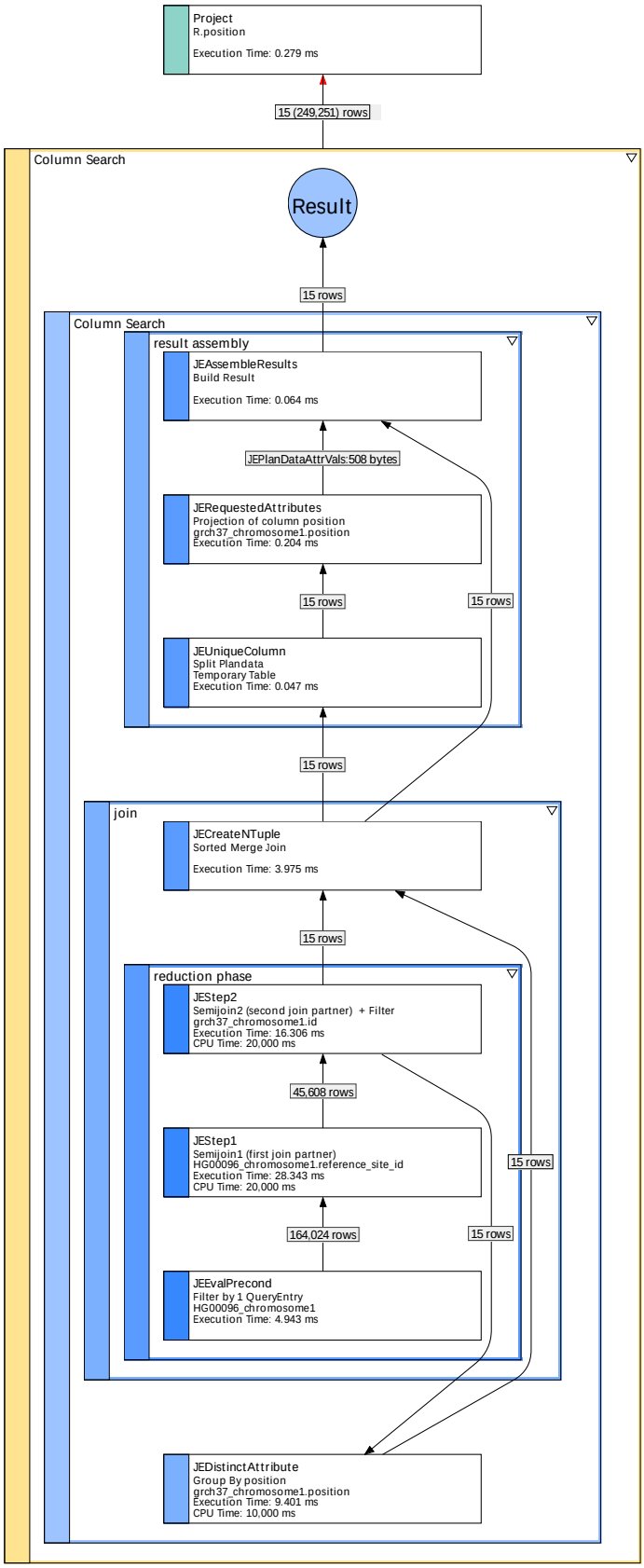


Figure A.1: Exemplary execution plan for a query to detect insertions in a 100,000 base pair range on database schema with keys

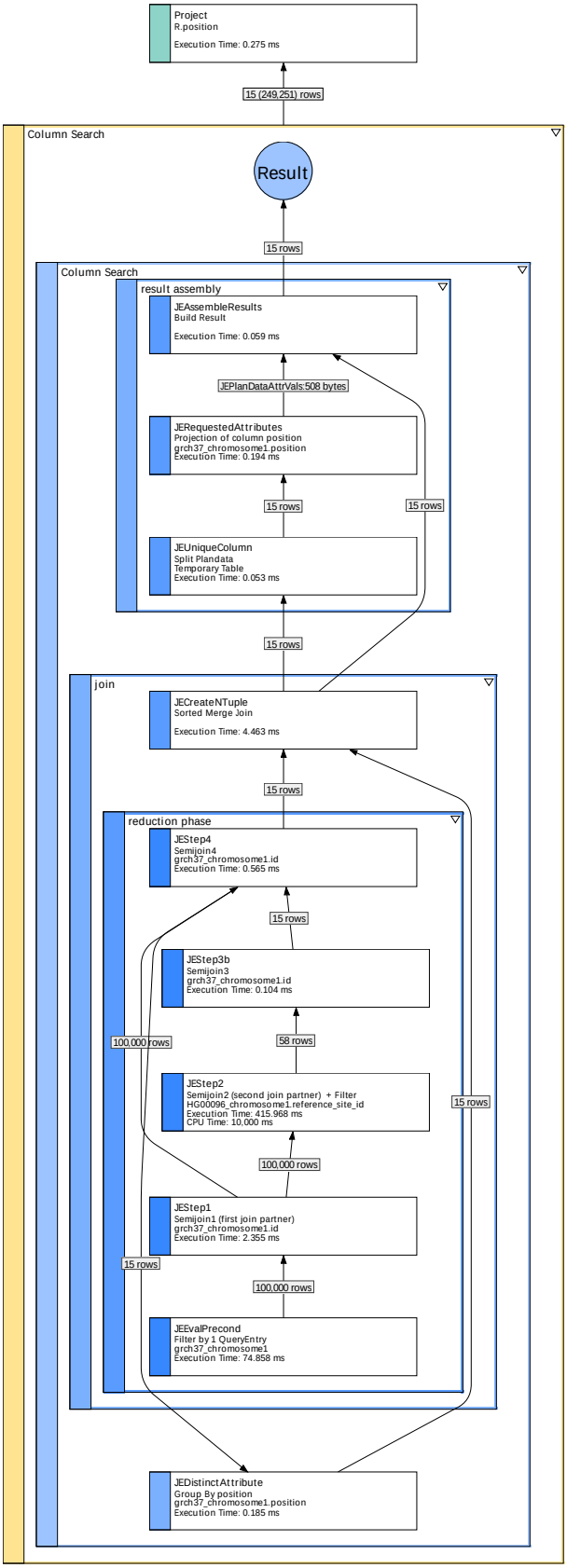


Figure A.2: Exemplary execution plan for a query to detect insertions in a 100,000 base pair range on database schema with no keys

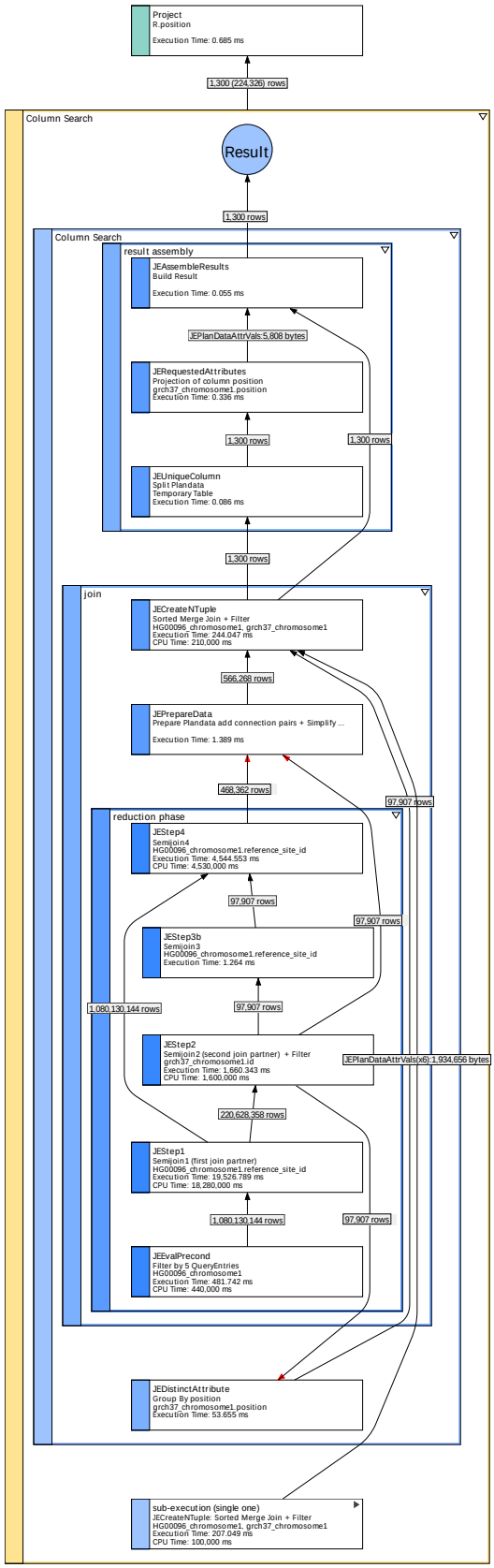


Figure A.3: Exemplary execution plan for queries to detect variants in a 100,000 base pair range on database schema with keys

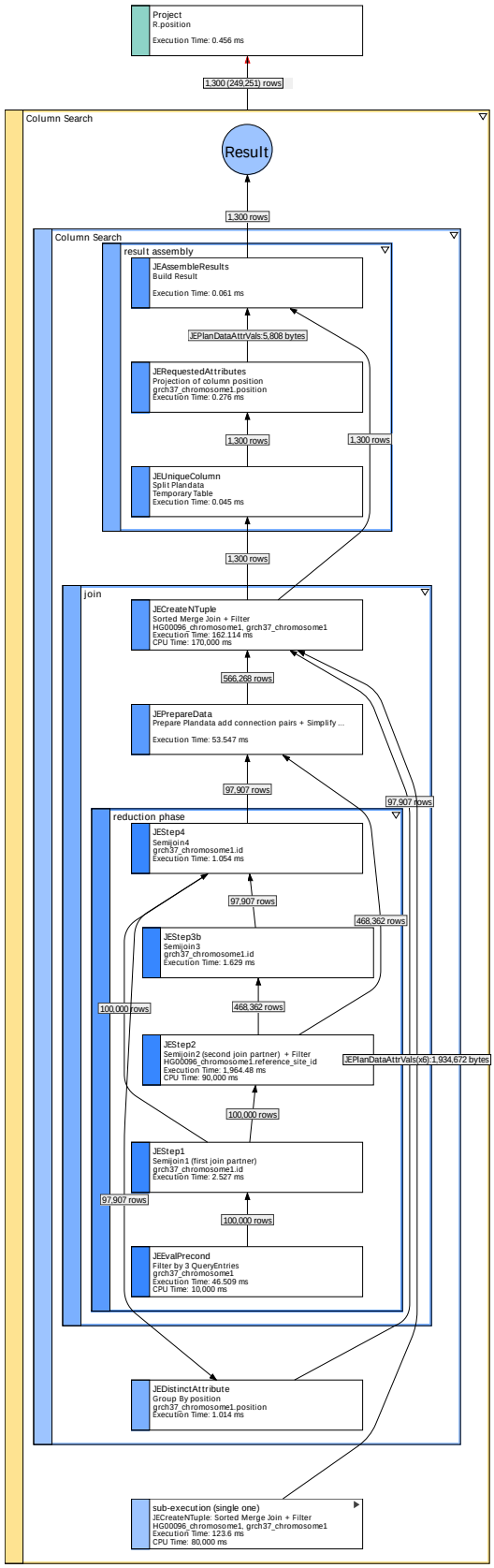


Figure A.4: Exemplary execution plan for queries to detect variants in a 100,000 base pair range on database schema with no keys

Bibliography

[Aba07] Daniel J. Abadi. Column stores for wide and sparse data. In *Proceedings of the Conference on Innovative Data Systems Research*, pages 292–297, 2007. (cited on Page 31 and 32)

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006. (cited on Page 29)

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 967–980, 2008. (cited on Page 31)

[Bay63] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53:370–418, 1763. (cited on Page 11)

[BBR+13] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084–1096, 2013. (cited on Page 32)

[BC02] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 2nd edition, 2002. (cited on Page 64)

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, 2003. (cited on Page 65)

[BGS+12] 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. (cited on Page 32)

[BIB+97] Frederick R. Blattner, Guy Plunkett III, Craig A. Bloch, Nicole T. Perna, Valerie Burland, Monica Riley, Julio Collado-Vides, Jeremy D. Glasner, Christopher K. Rode, George F. Mayhew, Jason Gregor, Nelson Wayne Davis, Heather A. Kirkpatrick, Michael A. Goeden, Debra J. Rose, Bob Mau, and Ying Shao. The complete genome sequence of escherichia coli k-12. *Science*, 277:1453–1462, 1997. (cited on Page 61 and 93)

[BKC+10] Daniel Blankenberg, Gregory Von Kuster, Nathaniel Coraor, Guruprasad Ananda, Ross Lazarus, Mary Mangan, Anton Nekrutenko, and James Taylor. Galaxy: A web-based genome analysis tool for experimentalists. *Current Protocols in Molecular Biology*, 89:19.10.1–19.10.21, 2010. (cited on Page 3)

[BPB+10] Marc Bouffard, Michael S Phillips, Andrew M K Brown, Sharon Marsh, Jean-Claude Tardif, and Tibor van Rooij. Damming the genomic data flood using a comprehensive analysis and storage data structure. *Database (Oxford)*, 2010, 2010. (cited on Page 40, 41, and 42)

[BWB09] Marty C. Brandon, Douglas C. Wallace, and Pierre Baldi. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 25(14):1731–1738, 2009. (cited on Page 3)

[CFG+10] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. The Sanger FASTQ file format for sequences with quality scores and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, 2010. (cited on Page 2 and 13)

[Che76] Peter Pin-Shan Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976. (cited on Page 19)

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279, 1985. (cited on Page 27 and 28)

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. (cited on Page 19)

[Cod82] Edgar F. Codd. Relational database: A practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982. (cited on Page 17)

[CPW+01] Mike Cornell, Norman W. Paton, Shengli Wu, Carole A. Goble, Crispin J. Miller, Paul Kirby, Karen Eilbeck, Andy Brass, Andrew Hayes, and Stephen G. Oliver. GIMS – A data warehouse for storage and analysis of genome sequence and functional data. In *Proceedings of the IEEE International Symposium on Bioinformatics and Bioengineering*, pages 15–22. IEEE Press, 2001. (cited on Page 39, 40, and 41)

[DA+11] Petr Danecek, , Adam Auton, , Goncalo Abecasis, Cornelis A. Albers, Eric Banks, Mark A. DePristo, Robert E. Handsaker, Gerton Lunter, Gabor T. Marth, Stephen T. Sherry, Gilean McVean, , Richard Durbin, , and 1000 Genomes Project Analysis Group. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011. (cited on Page 2)

[EG98] Brent Ewing and Phil Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Research*, 8(3):186–194, 1998. (cited on Page 13)

[EHWG98] Brent Ewing, Ladeana Hillier, Michael C. Wendl, and Phil Green. Base-calling of automated sequencer traces using phred. I. Accuracy assessment. *Genome Research*, 8(3):175–185, 1998. (cited on Page 13)

[EJ01] William E. Evans and Julie A. Johnson. Pharmacogenomics: The inherited basis for interindividual differences in drug response. *Annual Review of Genomics and Human Genetics*, 2(1):9–39, 2001. (cited on Page 1)

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, 6th edition, 2010. (cited on Page 17, 18, 19, 20, 21, 22, 23, 37, and 44)

[Fei09] Janet Feigenspan. Empirical comparison of fosd approaches regarding program comprehension – a feasibility study. Master's thesis, University of Magdeburg, 2009. (cited on Page 60)

[FSG13] Xosé M. Fernández-Suárez and Michael Y. Galperin. The 2013 nucleic acids research database issue and the online molecular biology database collection. *Nucleic Acids Research*, 41(Database-Issue):1–7, 2013. (cited on Page 3, 36, and 39)

[GBA+07] K. M. Giacomini, C. M. Brett, R. B. Altman, N. L. Benowitz, M. E. Dolan, D. A. Flockhart, J. A. Johnson, D. F. Hayes, T. Klein, R. M. Krauss, D. L. Kroetz, H. L. McLeod, A. T. Nguyen, M. J. Ratain, M. . V. Relling, V. Reus, D. M. Roden, C. A. Schaefer, A. R. Shuldiner, T. Skaar, K. Tantisira, R. F. Tyndale, L. Wang, R. M. Weinshilboum, S. T. Weiss, and I. Zineh. The pharmacogenetics research network: From SNP discovery to clinical drug response. *Clinical Pharmacology and Therapeutics*, 81(3):328–345, 2007. (cited on Page 1)

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. Hyrise - a main memory hybrid storage engine. *The Proceedings of the VLDB Endowment*, 4(2):105–116, 2010. (cited on Page 32)

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992. (cited on Page 27)

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice-Hall, 2000. (cited on Page 25 and 26)

[GR12] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf, 2012. Online resource. (cited on Page 1)

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994. (cited on Page 31)

[KBL06] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database systems an application-oriented approach*. Pearson Education, Inc., 2006. (cited on Page 24)

[KN11] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the International Conference on Data Engineering*, pages 195–206, 2011. (cited on Page 32)

[KPvdB92] Martin L. Kersten, Sander Plomp, and Carel A. van den Berg. Object storage management in goblin. In *Proceedings of the International Workshop on Distributed Object Management*, pages 100–116, 1992. (cited on Page 31)

[KSF+02] W. James Kent, Charles W. Sugnet, Terrence S. Furey, Krishna M. Roskin, Tom H. Pringle, Alan M. Zahler, and David Haussler. The human genome browser at ucsc. *Genome Research*, 12(6):996–1006, 2002. (cited on Page 39)

[LH10] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010. (cited on Page 9)

[LHW+09] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009. (cited on Page 2, 13, and 65)

[LLL+12] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. *Journal of Biomedicine & Biotechnology*, 2012:1–11, 2012. (cited on Page 1, 33, 34, and 35)

[LPW+06] Thomas J. Lee, Yannick Pouliot, Valerie Wagner, Priyanka Gupta, David W. J. Stringer-Calvert, Jessica D. Tenenbaum, and Peter D. Karp. Biowarehouse: a bioinformatics database warehouse toolkit. *BMC Bioinformatics*, 7(1):170, 2006. (cited on Page 4, 40, 41, 50, and 101)

[LSL+09] Ben Langmead, Michael C. Schatz, Jimmy Lin, Mihai Pop, and Steven L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):R134, 2009. (cited on Page 4)

[LTPS09] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009. (cited on Page 2)

[Mar10] Elaine Mardis. The $1,000 genome, the $100,000 analysis? *Genome Medicine*, 2(11):84, 2010. (cited on Page 36)

[MBK00] 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. (cited on Page 27 and 28)

[McV12] Gil A. McVean. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012. (cited on Page 93)

[Met09] Michael L. Metzker. Sequencing technologies - the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2009. (cited on Page 1)

[Mou04] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2nd edition, 2004. (cited on Page 2 and 8)

[Nob09] William Stafford Noble. A quick guide to organizing computational biology projects. *PLoS Computational Biology*, 5(7):e1000424, 2009. (cited on Page 2)

[Nom85] Nomenclature Committee of the International Union of Biochemistry (NC-IUB). Nomenclature for incompletely specified bases in nucleic acid sequences. recommendations 1984. *Biochemical Journal*, 229(2):281–286, 1985. (cited on Page 12)

[NPAS11] Rasmus Nielsen, Joshua S. Paul, Anders Albrechtsen, and Yun S. Song. Genotype and snp calling from next-generation sequencing data. *Nature Reviews Genetics*, 12(6):443–51, 2011. (cited on Page 9, 10, and 94)

[OJS+13] Jason O'Rawe, Tao Jiang, Guangqing Sun, Yiyang Wu, Wei Wang, Jingchu Hu, Paul Bodily, Lifeng Tian, Hakon Hakonarson, W. Evan Johnson, Zhi Wei, Kai Wang, and Gholson J. Lyon. Low concordance of multiple variant-calling pipelines: practical implications for exome and genome sequencing. *Genome Medicine*, 5(3):28, 2013. (cited on Page 44 and 61)

[Ora12] Oracle Health Sciences. Oracle health sciences translational research center: A translational medicine platform to address the big data challenge. Technical report, Oracle, 2012. (cited on Page 4)

[Ora13] Oracle. Oracle: Big data for the enterprise. Technical report, Oracle, 2013. (cited on Page 4)

[Pea13] William R. Pearson. *The FASTA program package*, 2013. Manual. (cited on Page 11)

[Pev03] Jonathan Pevsner. *Bioinformatics and functional genomics*. Wiley, 2003. (cited on Page 8)

[PFG+13] Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, and Martin L. Kersten. Cpu and cache efficient management of memory-resident databases. In *Proceedings of the International Conference on Data Engineering*, pages 14–25, 2013. (cited on Page 27, 31, and 32)

[PK06] Stephan Philippi and Jacob Köhler. Addressing the problems with life-science databases for traditional uses and systems biology. *Nature Reviews Genetics*, 7(6):482–488, 2006. (cited on Page 37)

[PL88] William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988. (cited on Page 11)

[Pla09] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–2, 2009. (cited on Page 32)

[PZ11] Hasso Plattner and Alexander Zeier. *In-Memory Data Management - An Inflection Point for Enterprise Applications*. Springer Berlin Heidelberg, 2011. (cited on Page 26, 27, 29, 30, and 31)

[Ram98] Raghu Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1998. (cited on Page 16 and 26)

[RB09] Uwe Röhm and José A. Blakeley. Data management for high-throughput genomics. In *Proceedings of the Conference on Innovative Data Systems Research*, 2009. (cited on Page 40, 41, 42, 50, and 100)

[RD06] Uwe Röhm and Thanh-Mai Diep. How to blast your database - a study of stored procedures for blast searches. In *Proceedings of the International Conference on Database Systems for Advanced Applications*, pages 807–816, 2006. (cited on Page 40 and 41)

[Ros00] Allen D. Roses. Pharmacogenetics and the practice of medicine. *Nature*, 405(6788):857–865, 2000. (cited on Page 1)

[SAM13] SAM/BAM Format Specification Working Group. *Sequence Alignment/Map Format Specification*, 2013. Manual. (cited on Page 15)

[SAP13] SAP AG. *SAP HANA Administration Guide*, 2013. Manual. (cited on Page 32, 60, and 67)

[Sco12] Colin Scott. Interactive latency. http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html, 2012. Online resource. (cited on Page 27)

[Sha02] B. S. Shastry. SNP alleles in human disease and evolution. *Journal of Human Genetics*, 47(11):561–566, 2002. (cited on Page 1)

[SHX+05] Sohrab P. Shah, Yong Huang, Tao Xu, Macaire M S Yuen, John Ling, and B F Francis Ouellette. Atlas - a data warehouse for integrative bioinformatics. *BMC Bioinformatics*, 6:34, 2005. (cited on Page 4, 40, 41, 50, and 101)

[SKS97] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 3rd edition, 1997. (cited on Page 16)

[SR99] Tom Strachan and Andrew P. Read. *Human Molecular Genetics*. New York: Wiley-Liss, 2nd edition, 1999. (cited on Page 70)

[SSH10] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken - Konzepte und Sprachen*. MITP, 4th edition, 2010. (cited on Page 21)

[TBAR08] Thoralf Töpel, Benjamin Kormeier, Andreas Klassen, and Ralf Hofestädt. Biodwh: A data warehouse kit for life science data integration. *Journal of Integrative Bioinformatics*, 5(2), 2008. (cited on Page 4, 40, and 41)

[VA+01] J. Craig Venter, , Mark D. Adams, Eugene W. Myers, Peter W. Li, Richard J. Mural, Granger G. Sutton, Hamilton O. Smith, Mark Yandell, Cheryl A. Evans, and Robert A. Holt. The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001. (cited on Page 1)

[VGB10] Marc Via, Christopher Gignoux, and Esteban González G. Burchard. The 1000 Genomes Project: New opportunities for research and social challenges. *Genome Medicine*, 2(1):3, 2010. (cited on Page 1 and 65)

[WHO12] WHO Regional Office for Europe. Leading causes of death in europe: fact sheet. http://www.euro.who.int/__data/assets/pdf_file/0004/185215/Leading-causes-of-death-in-Europe-Fact-Sheet.pdf, 2012. Online resource. (cited on Page 1)

[WMW11] Liewei Wang, Howard L. McLeod, and Richard M. Weinshilboum. Genomics and drug response. *New England Journal of Medicine*, 364(12):1144–1153, 2011. (cited on Page 1)

[WRB+12] Sebastian Wandelt, Astrid Rheinländer, Marc Bux, Lisa Thalheim, Berit Haldemann, and Ulf Leser. Data management challenges in next generation sequencing. *Datenbank-Spektrum*, 12(3):161–171, 2012. (cited on Page 3, 4, and 36)

[ZBY+11] Zhang Zhang, Vladimir B. Bajic, Jun Yu, Kei-Hoi Cheung, and Jeffrey P. Townsend. Data integration in bioinformatics: Current efforts and challenges. In *Bioinformatics - Trends and Methodologies*, chapter 2, pages 41–56. InTech, 2011. (cited on Page 4 and 35)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 28.10.2013