Einreichung für: BTW 2017,
Geplant als Veröffentlichung innerhalb der Lecture Notes in Informatics (LNI)

Efficient Storage and Analysis of Genome Data in Databases

Sebastian Dorok[1] Sebastian Breß[2] Jens Teubner[3] Horstfried Läpple[4] Gunter Saake[5] Volker Markl[6]

Abstract: Genome-analysis enables researchers to detect mutations within genomes and deduce their consequences. Researchers need reliable analysis platforms to ensure reproducible and comprehensive analysis results. Database systems provide vital support to implement the required sustainable procedures. Nevertheless, they are not used throughout the complete genome-analysis process, because (1) database systems suffer from high storage overhead for genome data and (2) they introduce overhead during domain-specific analysis. To overcome these limitations, we integrate genome-specific compression into database systems using a specialized database schema. Thus, we can reduce the storage overhead to 30%. Moreover, we can exploit genome-data characteristics during query processing allowing us to analyze real-world data sets up to five times faster than specialized analysis tools and eight times faster than a straightforward database approach.

Keywords: main-memory database systems, genome analysis, variant calling

1 Introduction

Genome sequencing and analysis promises to detect, predict and prevent diseases based on genetic variations more efficiently than traditional medicine can do [Br13]. Due to next-generation sequencing techniques, genome sequencing becomes cheaper and faster [Li12]. For that reason, reading genome sequences via sequencing machines is not the bottleneck anymore, but the management, analysis and assessment of large amounts of genome data, i.e. *detecting genetic variations* and investigating their consequences [Ma10].

To detect genetic variations, researchers use specialized tools. To investigate the consequences of potential variations, they use database systems that allow for convenient integration with other data sources [Ku07; Le06; Sh05; Tö08]. This separation introduces additional and partly manual effort to ensure reproducibility of results [Sa13]. Avoiding this separation enables researchers to analyze genomes completely within the database system. As a consequence, we can declaratively analyze genome data and improve the comprehensibility of analysis results [RB09]. Furthermore, database systems are able to provide comprehensive data-management features, such as provenance tracking [EOA07] or annotation management [Bh04], that would be available throughout the complete genome analysis process.

[1] Bayer Business Services GmbH & University Magdeburg (Work was done in part when employed at Bayer Pharma AG), sebastian.dorok@ovgu.de
[2] DFKI GmbH (Work was done in part when employed at TU Dortmund), sebastian.bress@dfki.de
[3] TU Dortmund, jens.teubner@tu-dortmund.de
[4] Bayer HealthCare AG, lapple@alumni.stanford.edu
[5] University Magdeburg, gunter.saake@ovgu.de
[6] TU Berlin, volker.markl@tu-berlin.de

Our experiments on integrating genome-analysis tasks, such as detecting genetic variation, into a database system (DBS) demonstrate that we can achieve competitive runtime performance compared to specialized analysis tools. However, DBSs lack appropriate light-weight compression techniques for genome data. For that reason, storage consumption increases by more than a factor of two compared to state-of-the-art flat files, because genome data consists mostly of unique strings that are hard to compress using standard light-weight compression schemes. Additionally, unfavorable string conversions during genome data processing increase time to knowledge within a DBS. Common genome-data encodings represent necessary analysis information implicitly within strings. Considering the required string manipulations, DBSs usually cannot keep pace with specialized analysis tools. To overcome both issues, we exploit genome-specific data and query characteristics within DBSs leading to a database-native application design. We make the following contributions:

1. We **identify genome-data related bottlenecks** in DBSs by performing an in-depth analysis of a straightforward database-approach for variant detection. As optimization targets, we identify missing genome-specific compression schemes and on-the-fly string conversion of genome data via user-defined functions (UDFs).
2. We enable **light-weight genome-specific compression** for DBSs. To this end, we use a specialized database schema allowing us to integrate genome-specific compression. At the same time, it allows us to perform string conversions once during data import, which improves analysis runtime by up to a factor of 1.5.
3. We propose a **genome-specific filtering technique** called *base pruning*. *Base pruning* leverages the characteristic of genome data to be very similar to a given reference genome reducing the number of genome positions to be processed and improving runtime by up to a factor of 5.
4. We **combine genome-specific compression and query optimization** to improve overall performance. Therefore, we outline how we can leverage reference-based compressed data to reduce the runtime of *base pruning*. Moreover, we explain why heavy-weight compression limits the overall benefit of *base pruning*.

Compared to state-of-the-art flat file formats, our techniques can reduce the storage overhead of a DBS approach to 30% without using heavy-weight compression. At the same time, we can detect genetic variation within whole genomes up to five times faster than state-of-the-art analysis tools and up to eight times faster than a straightforward DBS approach.

The remainder of the paper is structured as follows. In Section 2, we introduce the basics of variant detection. In Section 3, we assess a straightforward database-approach for variant detection regarding storage consumption and analysis performance to identify optimization targets. In Section 4, we introduce genome-specific compression schemes for database systems. In Section 5, we explain *base pruning*. In Section 6, we evaluate storage consumption and query performance of our approaches using three real world data sets. In Section 7, we provide an overview of related work.

2 Detecting genetic variation

In this section, we outline the basics of variant detection. First, we motivate the need for variant detection. Then, we describe an important variant detection approach: *Single Nucleotide Variant (SNV)* calling. Finally, we explain a common genome data encoding.

2.1 DNA sequencing and read mapping

DNA molecules encode genetic information via sequences of the four (nucleo)bases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). DNA sequencing machines make this genetic information digitally readable. To this end, they “read” the sequence of bases within DNA molecules and generate sequences of the characters A, C, G, and T. Therefore, the generated sequences are called *reads*. DNA sequencing techniques are not capable to process complete DNA molecules, but small parts of them only [Qu12]. Thus, in order to reconstruct the sample’s complete genome, reads must be assembled.

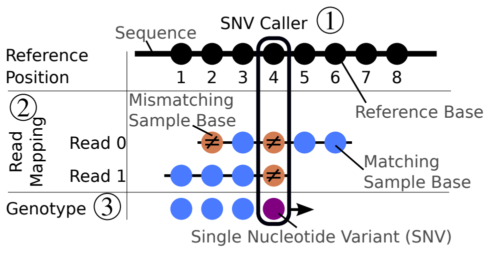


Fig. 1: From mapped reads to SNVs: A SNV caller ① aggregates bases of mapped reads ② per genome position, derives a genotype ③ and calls a SNV if genotype and reference base differ.

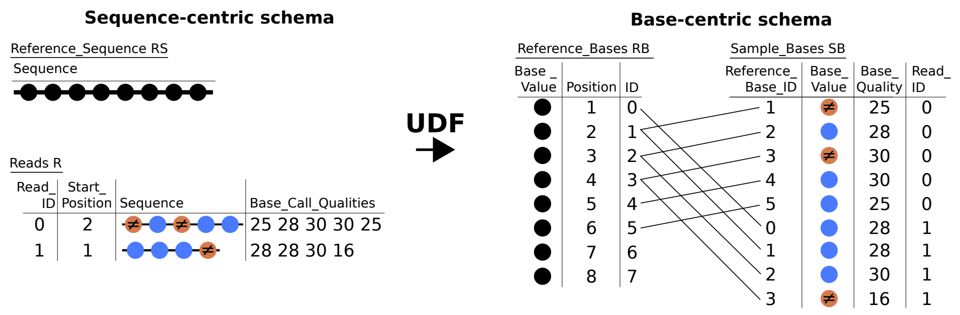
A common technique to assemble reads is read mapping [LH10]. Read mapping tools leverage already known reference sequences to reconstruct the sample’s genome by mapping reads to the best matching position. In Figure 1, we depict the mapping of two reads (chains of colored circles) to a reference sequence (chain of black circles). Read mapping is a challenging task and has to cope with several difficulties such as deletions, insertions and mismatches (cf. circles with $\neq$ symbol). These variances can be real variations but also DNA sequencing errors. Therefore, every base in a read has an associated quality value indicating the probability that the base is wrong. In this work, we refer to the output of read mappers using the term *genome data*.

2.2 Variant calling

Usually, scientists are interested in genome sites that differ from a given reference used during read mapping. Such genome sites are called *variants*. The process of detecting variants is called *variant calling*, i.e. determining whether a variant is present or not based on the mapped reads and associated quality information [Ni11]. A special class of variants are SNVs, i.e. differing genotypes at single genome positions (cf. purple circle in Figure 1). The detection of SNVs plays a vital role in genome analysis, because these are known to trigger diseases such as cancer [Ma13]. According to Nielsen et al., two general approaches for SNV calling exist: (1) frequency approaches with fixed cut-off rules and (2) probabilistic approaches incorporating uncertainty in data due to base call and read mapping errors [Ni11]. Independent of the concrete SNV calling approach, the general idea is to aggregate all bases that are mapped to a specific genome position. We depict this idea in Figure 1. The SNV caller (black box) consumes all bases that are mapped to the same genome position and computes a genotype. Afterwards, the SNV caller compares the genotype with the corresponding reference base. In case of a difference, it calls a SNV.

2.3 Encoding of reads

Existing flat-file formats for genome data are optimized to reduce storage consumption. Thus, they encode much information implicitly. For example, the *Sequence Alignment/*



Reference_Sequence RS

Sequence
●●●●●●●●

Reads R

Read_ ID	Start_ Position	Sequence	Base_Call_Qualities
0	2	≠●≠●●	25 28 30 30 25
1	1	●●●≠	28 28 30 16

Reference_Bases RB

Base_ Value	Position	ID
●	1	0
●	2	1
●	3	2
●	4	3
●	5	4
●	6	5
●	7	6
●	8	7

Sample_Bases SB

Reference_ Base_ID	Base_ Value	Base_ Quality	Read_ ID
1	≠	25	0
2	●	28	0
3	≠	30	0
4	●	30	0
5	●	25	0
0	●	28	1
1	●	28	1
2	●	30	1
3	≠	16	1

Fig. 2: A sequence-centric schema implicitly models read mapping information similar to flat files. In contrast, a base-centric schema makes mapping information explicit allowing for direct processing.

Map (SAM) format encodes the actual mapping of a read to a given reference as triple of a *Start position*, *DNA sequence* and *CIGAR string* [SA15]. The CIGAR string encodes whether a specific base within the DNA sequence is deleted, inserted, mismatched or matched and, thus, has impact on the actual position of the base. Thus, before performing SNV calling, position information of bases must be made explicit.

3 A straightforward database approach for SNV calling

A straightforward approach for SNV calling using database systems is to store reads similar to the SAM format and to process them similar to specialized analysis tools such as SAMTOOLS [Li09]. In the left part of Figure 2, we depict the basic idea. For every read in table `Reads (R)`, we store the sequence of bases (`R.Sequence`) together with the CIGAR string[7], the corresponding base call quality values (`R.Base_Call_Qualities`) and the start position (`R.Start_Position`). In table `Reference_Sequence (RS)`, we store the reference sequence as string (`RS.Sequence`). We call this data representation *sequence-centric database schema*.

To perform SNV calling, reads must be converted to get explicit access to all bases mapped to a specific genome position (cf. Section 2.3). For example, the analysis tool SAMTOOLS [Li09] transforms mapped reads into an intermediate data structure called *pileup*. A *pileup* lists all bases that map to a specific genome position. Then, SAMTOOLS computes genotypes by aggregating the bases in a *pileup*. We emulate this approach in a database-native way relying on standard database operators where possible. First, we convert the data via a UDF into an intermediate base-centric data representation [DBS14] making read mapping information explicitly available as shown in the right part of Figure 2. The reference sequence and the mapped reads are split into single bases by storing them, literally spoken, vertically in tables `Reference_Bases (RB)` and `Sample_Bases (SB)`[8]. Using a foreign-key relationship (cf. `SB.Reference_Base_ID`), we can explicitly encode which sample base belongs to which reference base. Further information such as read containment (`SB.Read_ID`), position within the genome (`RB.Position`) and base call qualities (`SB.Base_Quality`) is stored in adjacent columns. The explicit mapping

[7] For simplicity, we only consider mismatching bases and omit inserted or deleted bases.

[8] Using the base-centric database schema, we already apply CIGAR operations to the base values of reads.

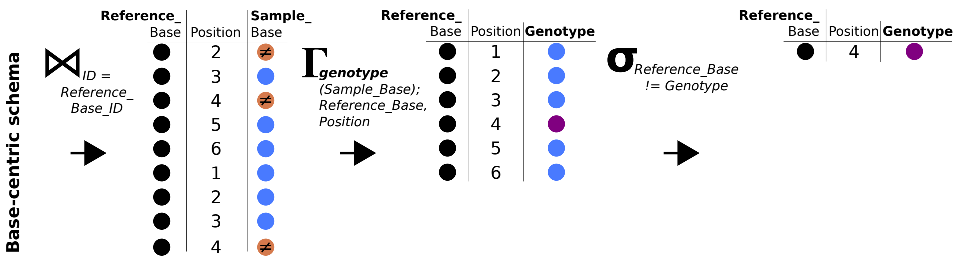


Fig. 3: The base-centric schema allows for direct access and processing of mapped reads via relational database operators.

information allows us to process genome data as shown in Figure 3 using relational database operators. To process a genome region of interest, we join the related bases and aggregate them by genome position using a domain-specific aggregation function called *genotype*. Finally, we filter genotypes that differ from the reference. In the example, we found position four to be a SNV.

In the following, we assess the straightforward approach regarding storage consumption and SNV calling performance to identify advantages and disadvantages. Of course, we have to distinguish between the logical data representation and the physical one. Without loss of generality, we assume that the physical data representation resembles the logical one. As evaluation system, we use CoGaDB [BFT16], a main memory database system, which stores and processes data column oriented similar to MonetDB. Furthermore, we chose CoGaDB, because it provides light-weight compression techniques that we use as baseline for our proposed optimizations. We use a complete human genome provided by the 1000 genomes project that comprises ca. 14 billion mapped sample bases to process. More information on the used data set and the evaluation machine can be found in Section 6.

3.1 Storage consumption

Efficiently storing genome data is hard to achieve as it mainly consists of unique strings. For example, reads are mostly unique to prevent a possible bias within analysis results [De11]. Thus, light-weight string compression schemes such as dictionary encoding increase storage size rather than compressing data. Therefore, we do not compress `RS.Sequence` and `R.Sequence` in a straightforward DBS approach. In Figure 4, we show the results of the DBS approach and the compressed SAM formats CRAM [CR15] and BAM [SA15].

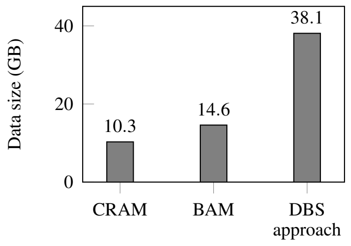


Fig. 4: A DBS approach requires three to four times more storage, which is due to missing compression capabilities.

Using the DBS approach, we require 3.7 times more storage space compared to CRAM and 2.6 times more storage compared to BAM. This is mainly due to the limited compression capabilities for unique strings. Both flat-file formats use heavy-weight compression such as BGZF encoding. CRAM additionally applies reference-based compression [Hs11].

3.2 SNV calling runtime

Now, we compare the SNV calling runtime of the DBS approach with the analysis tool SAMTOOLS [Li09]. We show the runtime results in Figure 5. The DBS approach uses a preloaded database. SAMTOOLS accesses flat files stored in a ramdisk. To make the comparison fair, we compare only the runtime for detecting SNVs and do not consider post-processing validations that can be applied to both approaches. Moreover, we parallelized SAMTOOLS to use all available threads, because SAMTOOLS does not provide such an option natively.

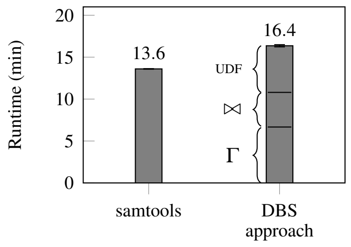


Fig. 5: SNV calling runtime using a DBS approach is comparable to the highly optimized SAMTOOLS.

Considering the overall SNV calling runtime, the DBS approach can be competitive to SAMTOOLS. The read conversion and aggregation phase dominate the runtime. We can speedup the aggregation phase using heuristics to reduce the number of groups to be processed. To speedup the conversion phase, we can integrate domain-specific processing mechanisms similar to specialized analysis tools such as SAMTOOLS. These tools rely on stream processing of genome data and require reads to be sorted by starting position to guarantee low response time. The sorting allows them to interleave data loading with conversion and aggregation, because compressed data-blocks read from disk contain reads of the same genome range. However, we should avoid such black-box behavior within a DBS, because it limits the transparency and portability and is hard to parallelize [RB09].

3.3 Wanted: Genome-specific extensions

Considering the results of the storage and SNV calling runtime experiments, a straightforward DBS approach suffers from missing genome-specific storage and processing capabilities. To reduce storage consumption, we focus on integrating reference-based compression, which achieves good compression ratios [Hs11]. Our goal is to integrate reference-based compression in a light-weight manner to avoid decompression overhead. To improve analysis performance, we want to provide a data layout that avoids string conversions, because these are non-relational operations that are hard to optimize by the DBS. In the following section, we explain how we achieve both goals.

4 Integrating genome-specific compression

State-of-the-art flat-file formats such as CRAM [CR15] use heavy-weight and genome-specific compression schemes to achieve good compression ratios of genome data. Disk-based database systems can hide the decompression overhead of heavy-weight compression when loading data from disk. In contrast, if we use heavy-weight compression in a main-memory database system, we would sacrifice the performance potentials gained from main-memory storage of data. Therefore, we aim to integrate genome-specific compression schemes in a light-weight manner into a DBS reducing decompression overhead by allowing us to process compressed genome data and to decompress single data items fast.

4.1 Light-weight reference-based compression

In the following, we explain how we integrate light-weight reference-based compression into a column-oriented database system. We provide an example in Figure 6.

Reference-based compression [Hs11] is a genome-specific compression scheme. It exploits read mapping information. Usually, the mapped reads and the reference sequence match to a high degree. Thus, the idea is to encode the mapped reads according to the reference sequence. Therefore, we need information about which sample base of a read maps to which reference base of the reference sequence. In a sequence-centric database schema where we store reads as strings, the required information is only given implicitly. This leads to additional overhead when (de-)compressing the data, because we have to extract necessary information before we can use it. What we essentially need is a mapping between sample and reference bases. If we introduce this mapping, we logically end up with a base-centric data representation. Consequently, our idea is to use the base-centric database schema, that encodes the mapping via a foreign-key relationship (cf. Figure 2), as primary data representation to integrate light-weight reference-based compression. At the same time, we remove the overhead of data conversion for SNV calling, if we can store genome data directly using the base-centric database schema.

Concept. The base-centric schema stores the mapping between sample base and reference bases explicitly via the `SB.Reference_Base_ID` column. We can use the column values as index to the `RB.Base_Value` column. Thus, we can look up the reference base to which a sample base is mapped and check whether it is different or not. Instead of storing each base value of a sample, we only store those bases in an *exception* list that are different from their respective reference base. Furthermore, we use a bitmap to mark the unequal bases. In case of good mapping quality, we do not have to store all sample bases.

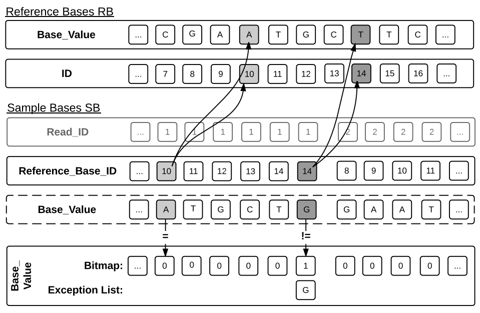


Fig. 6: Reference-based compression uses the existing foreign key relationship to compress sample bases. Differing sample bases are marked and stored in an exception list.

To make this technique efficient, we assume that the reference genome fits into main memory. Thus, we can retrieve sample bases by their row id as follows:

1. Given a row id, check whether the value is different according to the reference by scanning the bitmap.

2.1 In case of difference, use the prefix sum over the bitmap as index to look up the exception value.

2.2 In case of a match, look up the base in column `RB.Base_Value`. As look up index we can use the respective `SB.Reference_Base_ID`.

Improving storage consumption. Nevertheless, this approach requires to store one bit for every sample base. Consequently, if we store all sample bases of the data set used in the previous section, we require ca. 14 billion bits that is ca. 1.75 GB. To further reduce the data size, we use a compressing word-aligned hybrid (WAH) bitmap instead of a plain

Table	Column	GB	%
Sample_Bases SB	Reference_Base_ID	111.3	72.8
Reference_Bases RB	Position	12.5	8.2
Sample_Bases SB	Base_Value	5.2	3.4
Other columns		23.9	15.6

Tab. 1: Storage breakdown of a human genome using a base-centric database schema. Explicit position information stored in `SB.Reference_Base_ID` and `RB.Position` lead to a large storage blow up.

bitmap [WOS06]. In a WAHBitmap, zeroes and ones are organized in words of a specific size, e.g. 32 bit. If a word contains only zeroes or ones, it is converted into a fill word encoding the number of words that contain only zeroes or ones. Thus, long runs of zeroes or ones can be compressed effectively. Assuming a reasonable read mapping quality, our bitmap contains many zeroes.

Fast data access. The WAHBitmap applies run-length encoding on bitmaps. Consequently, random access performance is an issue as count values have to be summed up in order to determine the value of a given row. Thus, we extend the WAHBitmap to store the first row id that is not within a word similar as suggested by Abadi et al. for run-length compressed columns [AMF06]. Then, we can use binary search to access random row ids faster.

Since scientists are interested in consecutive genome ranges such as genes or chromosomes, it is quite common that our SNV calling query processes complete reads that are mapped to the specific genome region. Therefore, the chances to access consecutive values are high. Thus, we integrate a mechanism to speed up sequential row accesses by caching the last accessed index and visited word avoiding binary searches.

4.2 Delta+RLE encoding

On the one hand, the base-centric database schema enables us to integrate genome-specific compression in a light-weight way. On the other hand, the base-centric database schema has the drawback to increase the data volume due to explicit encoding of information. In Table 1, we breakdown the storage requirements for single columns of the base-centric database schema when storing the complete human genome data set from Section 3. The breakdown reveals that explicit position information stored in `SB.Reference_Base_ID` and `RB.Position` lead to a massive storage blow up. This information is usually implicitly stored within strings. Consequently, if we want to use the base-centric database schema as primary data representation, we have to cope with the additional storage overhead.

Both problematic columns contain runs of consecutive values that are incremented by one. This circumstance is inherent to the data as we store the base values of every read consecutively. The `SB.Reference_Base_ID` values are foreign keys to the `Reference_Bases` table. Within a mapped read, it is a common case that consecutive bases are mapped to consecutive reference bases. As we can guarantee that all reference bases are sorted according to their `RB.Position`, `SB.Reference_Base_ID` values are usually incremented by one within the same read. Thus, the delta between two values in one run is mostly one. In the following, we describe an encoding and compression scheme that combines delta encoding and run-length encoding (RLE) to compress such data: *Delta+RLE* encoding.

Concept. In Figure 7, we depict the idea of Delta+RLE compression and apply it to the base-centric database schema. We encode values of column `SB.Reference_Base_ID` using Delta+RLE encoding. Instead of encoding the delta values using run-length encoding, we generalize the concept of run-length encoding to support values that have a fix delta, i.e., one. This way we can compute single values within a run by adding the offset of the value to the actual run value. For example, the first run in Figure 7 has 10 as run value. If we want to decompress the third value within this run that has an index offset of 2, we sum 2 and 10 which gives us the decompressed value 12. Delta+RLE encoding can also be applied to `RB.Position`.

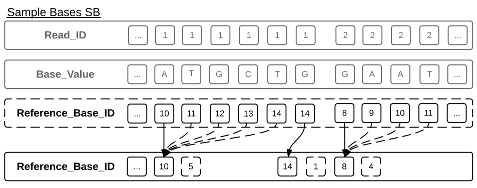


Fig. 7: Delta+RLE encoding represents runs of consecutive values as run value and length value similar to RLE encoding. This leads to an implicit string encoding for DNA sequences.

Fast data access. Delta+RLE has bad random access performance, because count values have to be summed up, in order to determine the value of a given row. Thus, we switch from using count values to prefix sums representing row ids [AMF06]. Given a row id, we can use binary search to determine the containing run. Using a caching mechanism as described in Section 4.1, subsequent sequential accesses do not require a binary search.

4.3 Base-centric schema as primary data layout

Now that we explained how to integrate genome-specific compression into a column-oriented database system, we conclude that the base-centric database schema is the more favorable data representation for genome data stored in a column-oriented database system than the straightforward sequence-centric schema. First, the base-centric database schema has similar storage requirements than the sequence-centric database schema for two reasons:

1. Column `SB.Base_Value` stores the single characters of reads consecutively, thus, the sequence of characters in memory resembles the original read sequence.
2. Using Delta+RLE encoding, we can store explicit position information highly efficient. As bases within reads are mapped to successive positions, we usually have to store one run value and one count value per read. That is similar to the storage requirements for keeping a pointer to a string per read in a sequence-centric database.

Second, the base-centric data representation encodes genome data in a database-native way that allows for direct data processing such as SNV calling (cf. Section 3). Moreover, we can leverage the base-centric schema to integrate genome-specific compression schemes. In the following, we call the database approach using the base-centric schema as primary data layout DBS$_{base}$ to distinguish it from the straightforward approach that we now call DBS$_{seq}$. Note, all optimizations discussed so far can also be applied to DBS$_{seq}$ after genome data has been converted effectively reducing the memory footprint of DBS$_{seq}$.

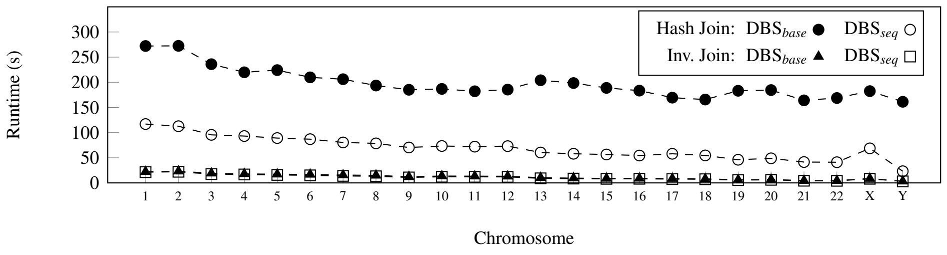


Fig. 8: Hash and invisible join runtimes on single chromosomes of a human genome using DBS$_{base}$ and DBS$_{seq}$. The invisible join is superior to the hash join and required to overcome the processing overhead introduced by the base-centric database schema due to large table sizes.

Advanced join processing required. The remaining challenge when using DBS$_{base}$, i.e., using the base-centric database schema as primary data layout, is the large size of table `Sample_Bases`. For example, if we store the complete genome used in Section 3, table `Sample_Bases` contains more than 14 billion rows, even if we just analyze a single chromosome. In contrast, using DBS$_{seq}$, we only convert the data required for analysis. For example, if we analyze chromosome 22, table `Sample_Bases` contains only 160 million rows. Considering the join between table `Reference_Bases` and `Sample_Bases` during SNV calling (cf. Figure 3), this difference in size becomes critical. Using a hash join, we hash table `Reference_Bases` and probe table `Sample_Bases`. Consequently, DBS$_{base}$ is slower than DBS$_{seq}$, because we always have to probe more rows. In Figure 8, we show the hash join runtimes calling single chromosomes on a human genome (cf. circled plots).

An alternative join technique is the *invisible join* proposed by Abadi et al. [AMH08]. The key idea is to apply predicates on dimension tables directly to the fact table (predicate rewriting) and to reconstruct join tuples later via positional lookups using foreign keys as indexes. A hash-based semi join between dimensions and fact table is a general strategy for predicate rewriting, but still requires to probe billions of rows of table `Sample_Bases`. In order to make the predicate rewriting more efficient, Abadi et al. introduce the so called *between-predicate rewriting*. They observed that predicates on dimension tables can often be rewritten as between-predicate on the respective foreign-key column of the fact table. In our case, we can rewrite the predicate on table `Reference_Bases` to filter for single chromosomes into a between predicate on column `SB.Reference_Base_ID` in table `Sample_Bases`, because reference bases are stored consecutively leading to consecutive primary keys. This reduces the memory footprint of our database approaches as we do not have to create intermediate hash tables [Do16]. Moreover, we only have to scan the foreign key column avoiding the effort of hash probing. Usually, the scan also has to touch every row in table `Sample_Bases`. In combination with Delta+RLE encoding, we are able to skip all rows represented by a run if the run disqualifies for the between predicate. Using the invisible join technique in combination with Delta+RLE encoding leads to large performance improvements of the join phase in DBS$_{base}$ and DBS$_{seq}$ (cf. Figure 8).

5 Base pruning

We also analyzed the general functionality of SNV callers. We came up with the conclusion that only those genome positions show a differing genotype than the reference base if at least one sample base differs from the reference base. Thus, if we know that a genome position has only matching sample bases mapped to it, we can exclude it from further processing during SNV calling, i.e., applying a domain-specific filter on the data. We show the idea in Figure 9. The black boxes indicate which genome positions have to be processed. The other genome positions have no mismatching bases (circles with $\neq$ symbol) mapped to it. Thus, we can reduce the processing effort during query processing. We only have to join those sample bases that may lead to a SNV call. Finally, the join result only contains those sample-base/reference-base tuples that really need to be aggregated. A traditional optimizer is not able to apply this optimization as it has no knowledge about the semantics of the genotype UDF.

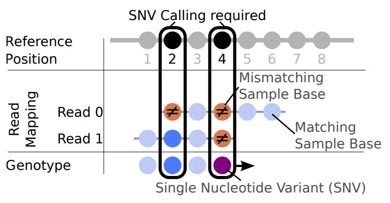


Fig. 9: Base pruning filters out genome positions where no mismatching base has been mapped which reduces the number of genome positions to process during SNV calling.

5.1 Approaches

In order to compute which genome positions have no differing sample base, we have to know in advance, which sample bases are mapped to which reference base. Using the base-centric database schema for storing genome data, we have this knowledge already encoded as foreign-key relationship between tables `Sample_Bases` and `Reference_Bases`.

Straightforward approach. In a straightforward approach, we scan the `Sample_Bases` table and compare each sample base with the corresponding reference base. Thus, we collect all genome positions that have at least one differing sample base. In a second step, we scan the table `Sample_Bases` again in order to determine all sample bases that are mapped to a genome position that has at least one differing sample base. Consequently, the straightforward approach requires two table scans over table `Sample_Bases`. These can be implemented very efficient. Nevertheless, during the first scan, we look up reference bases from the table `Reference_Bases`. Within one read, these lookups are cache-efficient as a read usually maps to a consecutive region within the reference genome. Thus, we cache further accesses to subsequent reference bases. Nevertheless, this approach introduces overhead due to comparisons. Moreover, for every new read, we make a random lookup into table `Reference_Bases` as different reads do not have to map to consecutive regions leading to cache misses.

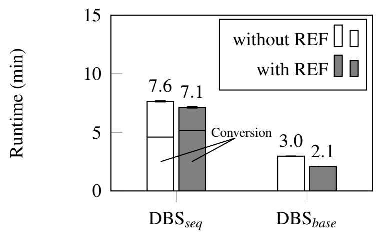


Fig. 10: Overall SNV calling runtime on a whole human genome using *base pruning*. with and without reference-based compression. Reference-based compression always improves runtime.

Indexed approach. Using reference-based compression, we can improve the straightforward *base pruning* computation. The reference-based compressed column `SB.Base_Value` already encodes which sample base is different according to the reference base. Thus, we can use it as index and extract all row ids of all differing sample bases from the bitmap of the compressed `SB.Base_Value` column. Therefore, we only have to scan the bitmap and return all row ids that are marked with one. Hence, we avoid to access table `Reference_Bases` at all. Furthermore, we do not have to make the comparison between every sample and reference base as the result is already encoded in the bitmap. In case of high mapping quality, an optimized bitmap, e.g., a WAHBitmap, contains many zeroes, which allows for skipping all rows represented by a zero fill word. In Figure 10, we show the impact of light-weight reference-based compression on the runtime of DBS_{seq} and DBS_{base} using *base pruning*. Using DBS_{base}, we can reduce the runtime by 30%. Using DBS_{seq}, the runtime reduction is roughly 7%, because data conversion takes most of the runtime that cannot be reduced using *base pruning*. Moreover, using reference-based compression increases the conversion runtime in DBS_{seq}. Nevertheless, the overall runtime savings due to *base pruning* pay off.

5.2 Applicability to specialized analysis tools

So far, we considered *base pruning* in the context of our database approaches. We can also apply it to specialized analysis tools such as SAMTOOLS [Li09]. Nevertheless, heavy-weight compression used by state-of-the-art flat file formats limits the effectiveness:

Applicability to aggregation phase only. We can apply *base pruning* only to improve the aggregation phase of the analysis tool. The reason for this is that SAMTOOLS operates on heavy-weight compressed data. To guarantee reasonable performance, also for random lookups within genomes, samtools requires mapped reads to be sorted by their genome position before being compressed. This is essentially the grouping attribute of the aggregation. Hence, SAMTOOLS can interleave decompression and conversion process and generate a ready-to-aggregate output, a so called pileup, because consecutive reads belong to the same genome region. Consequently, before we know which bases belong to which reference base in order to apply *base pruning*, data is already ready for aggregation. Compared to DBS_{seq}, we have already computed the join result. What remains is to check which pileups contain no differing base and do not have to be aggregated saving analysis runtime.

Reference-based compression cannot be exploited. We cannot exploit reference-based compressed data as index for improving the *base pruning* computation. This is a direct consequence from the first limitation. We perform the *base pruning* computation after the data is decompressed. Thus, we already lost the advantage of exploiting reference-based compression to reduce the computational effort.

6 Evaluation

In this section, we evaluate the database-driven approaches DBS_{seq} and DBS_{base} for SNV calling with regard to runtime performance and storage consumption. First, we investigate the storage consumption and compare it with the state-of-the-art flat-file formats CRAM and BAM. We want to find out whether we can cope with the storage blowup of the base-centric database schema. Moreover, we want to investigate what data characteristics impact our compression schemes at most. Then, we examine the SNV calling runtime on three real world data sets and compare it with the state-of-the-art analysis tool SAMTOOLS 1.3 [Li09]. We are interested in the overall analysis performance and how data characteristics influence it. Moreover, we want to find out to what extent the *base pruning* technique improves analysis runtime.

Organism	*Homo sapiens*		*Hordeum vulgare*
DataSet	*1*	*2*	*3*
# Mapped Bases	13.9B	11.8B	3.9B
# Reference Bases	3.1B	249M	1.9B
∅ Coverage	4	47	2
∅ Read Length	100	250	100
Mismatch Rate %	0.3	0.8	1.4

Tab. 2: Genome data sets differ in their key characteristics, which have impact on storage consumption and processing performance.

6.1 Experimental setup

As evaluation platform, we use a machine with two Intel Xeon E5-2609 v2 with four cores @2.5 GHz and 256 GB main memory. On the software side, we use Ubuntu 14.04.3 (64 Bit) as operating system and CoGaDB as database system (cf. Section 3). To compile CoGaDB, we use gcc 4.8.4 with optimization level *-O3*. Before starting the experiments, we pre-load the database into main memory. Similar to our initial experiment in Section 3.2, we use a manually parallelized and functionally reduced version of SAMTOOLS that accesses flat files stored on a ramdisk to make the comparison fair. We report average runtimes of 30 runs. Moreover, for runtime results, we report the 95% confidence intervals.

Data Sets. For our experiments, we use three real world data sets. *DataSet 1* and *2* contain human genome data and *DataSet 3* contains barley genome data. We obtained the human genome data from the 1000 genomes project[9], which provides representative real world data sets [Th15]. *DataSet 2* contains only the mapped reads of human chromosome 1. The plant research institute IPK Gatersleben provided us with barley data.

The data sets differ in their *number of reference bases*, *coverage*, *read length* and *mismatch rate*. In Table 2, we summarize the characteristics. The *number of reference bases* indicates the upper bound for genome positions that have to be analyzed. The *coverage* indicates how many sample bases are mapped on average to a certain reference genome position. Thus, in case of SNV calling, higher coverage leads to more sample bases to aggregate per reference genome position. *Coverage* and *number of reference bases* together determine the *number of mapped bases* in a data set. The *read length* has direct impact on storage consumption. If reads are longer, less read data per sample base must be stored (cf. *DataSet 1* and *2*). The *mismatch rate* indicates how many sample bases within the data set are different from their corresponding reference base. The barley data has a higher mismatch rate than the human data sets, which can have impact on the number of reported SNVs and may impact the effectiveness of reference-based compression and base pruning.

6.2 Storage consumption

In the first experiment, we examine the storage consumption of the database approaches DBS$_{seq}$ and DBS$_{base}$ and the state-of-the-art flat-file formats BAM and CRAM. We have two

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

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

Tab. 3: Storage consumption of DBS$_{seq}$, DBS$_{base}$, BAM and CRAM. DBS$_{base}$ outperforms DBS$_{seq}$ on all data sets. BAM and CRAM are superior as they apply heavy-weight compression. Zipping DBS$_{base}$ leads to a similar similar storage consumption as CRAM.

main objectives: 1) We want to investigate whether we can cope with the storage blowup of the base-centric database schema and 2) We want to find out how data characteristics impact the compression ratio. We report the absolute storage requirements for storing sample genome and respective reference genome data in Table 3 including storage required for indexes to improve data access speed. In brackets, we show the relative storage requirements compared to BAM, which serves as our baseline.

Effective reference-based compression in column-stores. The results show that DBS$_{base}$ always requires less storage than DBS$_{seq}$ independent of the stored data set. Due to Delta+RLE compression that effectively reduces the overhead due to explicit positional information and reference-based compression, we can decrease the storage size in a database system by 26 to 35%. Compared to BAM, DBS$_{base}$ needs 2 to 2.5 times more storage than BAM, since we do not use heavy-weight compression. Still, reference-based compression is an essential mean to reduce the storage size of genome data. The CRAM results show that reference-based compression in combination with heavy-weight compression further reduces the storage size compared to BAM. Compressing the disk-resident data files of DBS$_{base}$ using GZIP leads to a similar result (cf. last row of Table 3).

Data-dependent storage requirements. Table 3 reveals that the storage savings of DBS$_{base}$ compared to DBS$_{seq}$ depend on the data set. The reason for the differences between the three data sets is two-fold: *read length* and *mismatch rate*. In Table 4, we show the impact of these characteristics on the three columns `SB.Reference_Base_ID`, `SB.Read_ID` and `SB.Base_Value`. All other columns' sizes are independent of data set characteristics.

Impact of read length. We use RLE to compress `SB.Read_ID` and our Delta+RLE encoding to compress `SB.Reference_Base_ID`. Both encodings are sensitive to the length of runs within the data. The longer the runs, the better the compression ratio. Since we store bases of the same read consecutively (`SB.Read_ID`) and these bases usually map to successive genome positions (`SB.Reference_Base_ID`), longer reads lead to increased run length. For that reason, in *DataSet 2* with 250 bases per read on average, each column requires 0.6 bit per column per row. The other two data sets require 1.4 bit per column per row, because reads ahve an average length 100 bases.

Impact of mismatch rate. The different mismatch rates of the data sets directly impact the storage requirements of the reference-based compressed column `SB.Base_Value`. Fewer mismatches lead to fewer values to be stored in the exception list. Moreover, the bitmap contains more zeroes that can be leveraged by a WAHBitmap. Therefore, *DataSet 1* requires

	DataSet	*1*	*2*	*3*
	# Mapped Bases	13.9B	11.8B	3.9B
	∅ Read Length	100	250	100
	Mismatch Rate %	0.3	0.8	1.4
Bits per row	`SB.Reference_Base_ID`	1.4	0.6	1.4
	`SB.Read_ID`	1.4	0.6	1.4
	`SB.Base_Value`	0.6	0.8	1.3
	Sum of bits	3.4	2.0	4.1

Tab. 4: Influence of data characteristics on storage consumption using DBS$_{base}$. The longer the reads, the smaller is the overhead of foreign key columns `SB.Reference_Base_ID` and `SB.Read_ID`.

only 0.6 bits on average per mapped base. The other data sets require more bits per mapped base in concordance with their mismatch rate. Among the three columns, the overall storage consumption is dominated by the read length. The overall required number of bits per row in the three columns corresponds to the observed storage savings of DBS$_{base}$ comapred to DBS$_{seq}$ (cf. Table 3).

6.3 SNV calling runtime

In the second experiment, we examine the SNV calling runtime of DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS 1.3 with and without base pruning. Note, we use the same probabilistic error-model in our database approaches as SAMTOOLS. We do not consider post-processing validations that can be applied to the results of all approaches. We show the runtime results on the three different data sets from the experiment before in Figure 11. The hatched bars indicate runtimes with *base pruning*. First, we consider the runtime of the single approaches *without base pruning* and investigate the impact of data characteristics. Then, we examine the impact of *base pruning*.

A base-centric database schema pays off. As expected, DBS$_{base}$ always outperforms DBS$_{seq}$ if we do not apply base pruning, because we do not have to convert data on-the-fly. Although DBS$_{base}$ has to process all sample bases, in particular during the join phase, we can reduce the overhead effectively using the invisible join technique (cf. Section 4.3). Moreover, the experiment reveals that DBS$_{base}$ is competitive in terms of runtime compared to SAMTOOLS due to the use of advanced processing techniques. Thus, the required effort to compress base-centric genome data pays off.

Impact of data characteristics. Considering the results *without base pruning*, we find that the runtime depends on the number of genome positions and mapped bases to process.

Number of genome positions. For example, *DataSet 1* and *DataSet 2* contain roughly the same amount of mapped bases to process, but *DataSet 1* contains data for the complete genome, i.e. 3.1 billion genome positions. In *DataSet 2* all mapped bases only belong to the 249 million genome positions of chromosome 1. We expected a correlation with the overall analysis runtime, because computing more genome positions requires managing more

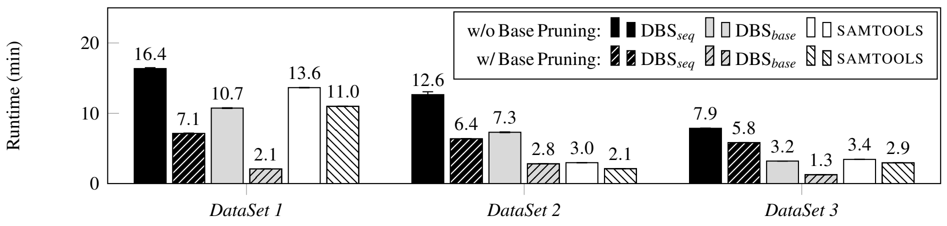


Fig. 11: SNV calling on three different real-world data sets using DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS with and without *base pruning*. DBS$_{seq}$ is always slower due to conversion overhead. DBS$_{base}$ benefits most from base pruning.

aggregation groups. Nevertheless, the large runtime differences of SAMTOOLS between *DataSet 1* and *2* are unexpected. The database approaches roughly require 30% less runtime on *DataSet 2* compared to *DataSet 1*, because less groups have to be initialized and computed. In contrast, SAMTOOLS saves 80%. An indepth analysis of SAMTOOLS revealed that SAMTOOLS has large overhead for writing analysis results per genome position via strings. Thus, analyzing many genome positions (cf. Dataset 1) increases its runtime.

Number of mapped bases. The runtimes on all data sets reveal that the database approaches are more affected by the number of mapped bases to process. Considering *DataSet 1*, the runtime of DBS$_{base}$ and SAMTOOLS are nearly equal, but for *DataSet 2*, using DBS$_{base}$ increases the runtime by factor 2.5 compared to SAMTOOLS. The reason for this difference is that the database approaches suffer from sorting or synchronization overhead during aggregation processing. SAMTOOLS requires presorted data allowing for filterings by genome positions and manual parallelization. The presorting also ensures that different threads operate on distinct genome regions. We can emulate this behavior by using a sort-based aggragtion processing, which obviously introduces overhead during runtime. Another strategy is a hash-based aggregation. Certainly, this strategy requires locking mechanisms, because the work per genome region is distributed between different threads. Consequently, the database aproaches suffer from high coverage data.

The tradeoff of both data characteristics can be seen when processing *DataSet 3*. Again the runtimes of DBS$_{base}$ and SAMTOOLS are competitive. *DataSet 3* contains less genome positions favoring SAMTOOLS, but also less mapped bases to process favoring DBS$_{base}$.

Impact of base pruning.

Using *base pruning*, we aim to restrict the costly processing to those genome positions that might show a variant (cf. Section 5). Using one of the database approaches, we can make benefit of this reduction during the join and aggregation phase. Using SAMTOOLS, we still have to decompress, convert and inspect all genome positions before taking advantage of the *base pruning* technique (cf. Section 5.2). Consequently, the database approaches benefit most from using *base pruning*. In Figure 11, we show the runtime results of all three approaches *with base pruning* indicated with hatched bars.

DBS$_{base}$ outperforms SAMTOOLS on *DataSet 1* and *3*. Even DBS$_{seq}$ is faster than SAMTOOLS on *DataSet 1*. On *DataSet 2*, SAMTOOLS is still faster due to less genome positions to process even without base pruning. Overall DBS$_{base}$ benefits most from *base pruning*, because we reduce the number of genome positions to process during all processing steps.

6.4 Discussion

Our evaluation shows that a base-centric data representation outperforms a straightforward database approach regarding storage consumption. The concrete storage savings depend on the read length and number of mismatching bases. Nevertheless, heavy-weight compression impacts storage size more, even on a sequence-centric data representation. Thus, BAM, which applies only heavy-weight compression, requires less storage than DBS_{base}. CRAM that additionally applies reference-based compression can compress data even better. Our experiments show that a simple compression of database files using GZIP leads to similar results than CRAM. Thus, integrating heavy-weight compression into our database approach would be beneficial especially to keep cold data without overhead on secondary storage.

The second advantage of a base-centric data representation for genome data is the improved analysis performance compared to a straightforward approach. We are able to detect SNVs as efficiently as SAMTOOLS. To achieve this performance, we rely on advanced processing techniques such as invisible join. In combination with *base pruning*, we can further improve analysis runtime. Overall, our proposed techniques and approaches benefit from longer reads and smaller mismatch rate. Thus, database systems using our techniques will benefit from improvements in DNA sequencing that will generate longer and more accurate reads [Li12] leading to less mismatches due to mapping errors.

7 Related work

In the following, we categorize and discuss approaches that use database systems and technology to efficiently store, manage and analyze genome data.

Data warehouse approaches. One of the first approaches to manage and integrate genome data in a database system is AceDB [ST99] using an object database. Several scientists proposed more advanced data warehouse solutions for managing and analyzing genome data and related data from other data sources [Le06; Sh05; Tö08]. The main focus of these solutions is the integration of different heterogeneous data sources to allow for integrated analyses. These approaches do not consider storage size, analysis efficiency or incorporate genome analysis tasks such as SNV calling. Instead they integrate such data. Our proposed approaches complement these data warehouse solutions by integrating SNV calling into a DBS. Moreover, we propose compression schemes that enable a data warehouse to store raw data and compute analysis results on-the-fly.

Integrated data analysis. Besides classical data warehouse solutions, approaches exist that integrate genome analysis functionality into a data management solution. For example, Ceri et al. present a data-management approach that allows for storing and querying genome-position specific data using a simple data model called Genomic Data Model [Ce16]. Furthermore, they propose the GenoMetric Query Language that provides algebraic operations similar to SQL and domain-specific analysis functionality. In contrast, our approach aims at using existing database technology to support genome analysis tasks. *bdbms* proposed by Eltabakh et al. also extends an existing database system with biological functionality [EOA07] such as annotations and provenance tracking. Moreover, it provides pattern matching functionality for compressed sequence data. Our work complements *bdbms* by providing genome-specific compression and analysis functionality.

Our work is mainly related to the work by Röhm and Blakeley [RB09]. They propose an approach to integrate genotyping, the pre-computation step of SNV calling, into a database system. They use a disk-based database system and enable users to operate on the original flat-files. Nevertheless, they report unsatisfactory analysis performance due to the use of multiple user-defined functions that are hard to parallelize. Moreover, within their approach for genotyping, they follow a straightforward flat-file approach introducing additional conversion overhead (cf. Section 3). Our DBS_{base} approach avoids this overhead using a special encoding of genome data requiring only one user-defined function for analysis.

Moreover, several other approaches exist that explicitly use main-memory database systems to integrate genome analysis tasks. The approach by Fähnrich et al. uses a two-phase map-reduce approach to convert reads and perform SNV calling [FSP15]. The approach by Cijvat et al. uses a special user-defined function of MonetDB to convert DNA sequences [Ci15]. As this operation is expensive, they cache the result for further analysis. Thus, our proposed technique to efficiently encode converted genome data complement both approaches.

Reference-based analysis techniques. Currently, we are aware of one variant calling approach called CAGe that leverages the similarity between reference genome and sample genome to reduce the analysis runtime of variant detection [Bl14]. The approach classifies genome regions regarding their analysis complexity incorporating information about similarity. Regions with high similarities have a low complexity and are analyzed using fast variant calling approaches. On the other hand, regions with many differences are complex and more sophisticated approaches are applied. Our base-pruning approach can improve the overall analysis runtime as it reduces the runtime to analyze low complexity regions.

Another approach that leverages the similarity between reference and sample genome is RCSI proposed by Wandelt et al. [Wa13]. This approach aims at similarity search on referentially compressed genomes. The idea is to first search on the reference sequence finding matching segments that may contain errors. In a second step, the compressed sample genomes are searched at the corresponding segments to generate the final result. Our approach to integrate reference-based compression provides a basis to integrate this technique into a relational database system. Moreover, we can use the optimized database processing engine to look up sample genome segments of interest fast.

8 Conclusion

In this paper, we showed that a base-centric data representation is required to integrate genome-specific compression schemes such as reference-based compression into a database system. Based on this, we proposed a filtering technique called *base pruning* leveraging reference-based compression as index. Using our database-native approach improves the overall runtime up to a factor of five compared to specialized analysis tools. The concrete performance gains depend mainly on coverage and mismatch rate of the analyzed data set.

Overall, our techniques enable scientists and researchers to perform SNV calling within a database on-the-fly, instead of precomputing results. In our experiments, we used a probabilistic calling approach based on the error-model routine of SAMTOOLS. By simply choosing a different aggregation function, we can apply an alternative variant calling approach if necessary [Ni11]. Especially on small genome regions, an interactive and declarative analysis becomes possible. The raw data and analysis results are delivered by a single database system improving traceability of results. Additionally, we will benefit from future improvements of database systems due to our database-native application design. The code used in this paper is available at http://cogadb.dfki.de/download/.

Acknowledgements

The work has received funding from the German Research Foundation (DFG), Collaborative Research Center SFB 876, project C5, from the European Union's Horizon2020 Research & Innovation Program under grant agreement 671500 (project SAGE), and by the German Ministry for Education and Research as Berlin Big Data Center BBDC (01IS14013A).

References

[AMF06] Abadi, D. et al.: Integrating compression and execution in column-oriented database systems. In: SIGMOD. pp. 671–682, 2006.

[AMH08] Abadi, D. et al.: Column-Stores vs. Row-Stores: How different are they really? In: SIGMOD. pp. 967–980, 2008.

[BFT16] Breß, S. et al.: Robust query processing in co-processor-accelerated databases. In: SIGMOD. pp. 1891–1906, 2016.

[Bh04] Bhagwat, D. et al.: An annotation management system for relational databases. In: VLDB. pp. 900–911, 2004.

[Bl14] Bloniarz, A. et al.: Changepoint analysis for efficient variant calling. In: RECOMB. pp. 20–34, 2014.

[Br13] Bromberg, Y.: Building a genome analysis pipeline to predict disease risk and prevent disease. J. Mol. Biol. 425/21, pp. 3993–4005, 2013.

[Ce16] Ceri, S. et al.: Data management for next generation genomic computing. In: EDBT. pp. 485–490, 2016.

[Ci15] Cijvat, R. et al.: Genome sequence analysis with MonetDB - A case study on Ebola virus diversity. Datenbank-Spektrum 6/17, pp. 185–191, 2015.

[CR15] CRAM Format Spec. Working Group: CRAM Format Specification, 2015.

[DBS14] Dorok, S. et al.: Toward efficient variant calling inside main-memory database systems. In: BIOKDD-DEXA. pp. 41–45, 2014.

[De11] DePristo, M. et al.: A framework for variation discovery and genotyping using next-generation DNA sequencing data. Nat. Genet. 43/5, pp. 491–498, 2011.

[Do16] Dorok, S.: Memory efficient processing of DNA sequences in relational main-memory database systems. In: GvDB. pp. 39–43, 2016.

[EOA07] Eltabakh, M. et al.: bdbms - A database management system for biological data. In: CIDR. pp. 196–206, 2007.

[FSP15] Fähnrich, C. et al.: Facing the genome data deluge: Efficiently identifying genetic variants with in-memory database technology. In: SAC. pp. 18–25, 2015.

[Hs11] Hsi-Yang Fritz, M. et al.: Efficient storage of high throughput DNA sequencing data using reference-based compression. Genome Res. 21/5, pp. 734–740, 2011.

[Ku07] Kuenne, C. et al.: Using Data Warehouse Technology in Crop Plant Bioinformatics. J. Integr. Bioinform. 4/1, 2007.

[Le06] Lee, T. J. et al.: BioWarehouse: A bioinformatics database warehouse toolkit. BMC Bioinformatics 7/1, pp. 170+, 2006.

[LH10] Li, H. et al.: A survey of sequence alignment algorithms for next-generation sequencing. Brief. Bioinform. 11/5, pp. 473–483, 2010.

[Li09] Li, H. et al.: The Sequence Alignment/Map format and SAMtools. Bioinformatics 25/16, pp. 2078–2079, 2009.

[Li12] Liu, L. et al.: Comparison of next-generation sequencing systems. J. Biomed. Biotechnol. 2012/, pp. 1–11, 2012.

[Ma10] Mardis, E. R.: The $1,000 genome, the $100,000 analysis? Genome Med. 2/11, pp. 1–3, 2010.

[Ma13] Mavaddat, N. et al.: Cancer risks for BRCA1 and BRCA2 mutation carriers: Results from prospective analysis of EMBRACE. J. Natl. Cancer Inst. 105/11, pp. 812–22, 2013.

[Ni11] Nielsen, R. et al.: Genotype and SNP calling from next-generation sequencing data. Nat. Rev. Genet. 12/6, pp. 443–51, 2011.

[Qu12] Quail, M. et al.: A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. BMC Genomics 13/1, pp. 341+, 2012.

[RB09] Röhm, U. et al.: Data management for high-throughput genomics. In: CIDR. 2009.

[Sa13] Sandve, G. K. et al.: Ten simple rules for reproducible computational research. PLoS Comput. Biol. 9/10, 2013.

[SA15] SAM/BAM Format Spec. Working Group: SAM Format Specification, 2015.

[Sh05] Shah, S. P. et al.: Atlas - A data warehouse for integrative bioinformatics. BMC Bioinformatics 6/1, pp. 34+, 2005.

[ST99] Stein, L. D. et al.: AceDB: A genome database management system. Comput. Sci. Eng. 1/3, pp. 44–52, 1999.

[Th15] The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature 526/7571, pp. 68–74, 2015.

[Tö08] Töpel, T. et al.: BioDWH: A data warehouse kit for life science data integration. J. Integr. Bioinform. 5/2, 2008.

[Wa13] Wandelt, S. et al.: RCSI: Scalable similarity search in thousand(s) of genomes. PVLDB 6/13, pp. 1534–1545, 2013.

[WOS06] Wu, K. et al.: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. 31/1, pp. 1–38, Mar. 2006.