

Memory efficient processing of DNA sequences in relational main-memory database systems

Sebastian Dorok
Bayer Pharma AG
Otto-von-Guericke-University Magdeburg
Institute for Technical and Business Information Systems
Magdeburg, Germany
sebastian.dorok@ovgu.de

ABSTRACT

Pipeline breaking operators such as aggregations or joins require database systems to materialize intermediate results. In case that the database system exceeds main memory capacities due to large intermediate results, main-memory database systems experience a massive performance degradation due to paging or even abort queries.

In our current research on efficiently analyzing DNA sequencing data using main-memory database systems, we often face the problem that main memory becomes scarce due to large intermediate results during hash join and sort-based aggregation processing. Therefore, in this paper, we discuss alternative join and aggregation techniques suited for our use case and compare their characteristics regarding memory requirements during processing. Moreover, we evaluate different combinations of these techniques with regard to overall execution runtime and scalability to increasing amounts of data to process. We show that a combination of invisible join and array-based aggregation increases memory efficiency enabling to query genome ranges that are one order of magnitude larger than using a hash join counterpart in combination with sort-based aggregation.

General Terms

Database, Genome Analysis

Keywords

Variant Calling, Invisible Join, Array-based Aggregation

1. INTRODUCTION

Recent research on efficiently querying DNA sequencing data using relational database systems focuses on using main-memory technology [5, 9, 11]. The goal is to benefit from the increased analysis performance and to enable research departments and scientists to perform analyses of DNA sequencing data efficiently and in a scalable way in a relational

database management system. Such an approach would eliminate several data management issues within genome analysis such as result traceability and repeatability as data can be analyzed from end-to-end in a database system [7].

The size of single genome data sets varies between several hundred to thousand megabytes. Besides varying data set size, also the analysis target varies between querying small genome regions such as genes and bulk processing complete genomes. With increasing data sizes to analyze, larger amounts of tuples have to be processed limiting the applicability of main-memory database systems as pipeline-breaking operators such as aggregations and joins require the database system to materialize intermediate results. Such result materialization accompanies materialization costs and increases main memory consumption within main-memory database systems. During analysis of DNA sequencing data, large amounts of tuples have to be processed due to low selectivity. Thus, it is quite common that the intermediate result materialization exceeds available main memory leading to a massive performance degradation of the system.

Current work in the field of genome analysis using main-memory database systems does not address this scalability issue explicitly. Fähnrich et al. show that main-memory technology outperforms highly tuned analysis tools [11]. The authors examine the scalability of their approach with regard to multiple threads but not regarding data size. Civjat et al. use the main-memory database system MonetDB to process DNA sequencing data but do not provide insights into the scalability of their approach regarding data size [5].

In this work, we focus on a base-centric representation of DNA sequences and investigate different join and aggregation techniques with regard to their overall runtime and scalability to increasing genome ranges to query.

The remainder of the paper is structured as follows. In Section 2, we provide basic information about genome analysis, our used database schema and the executed query type. In Section 3, we discuss different join and aggregation processing techniques from the perspective of our use case. In Section 4, we compare the different combinations of the discussed processing techniques and evaluate their overall runtime and scalability with regard to different data sizes to process. In Section 5, we discuss related work.

2. BACKGROUND

In this section, we explain the basic steps of genome analysis. Then, we explain how we can perform variant calling, a typical genome analysis task, using SQL.

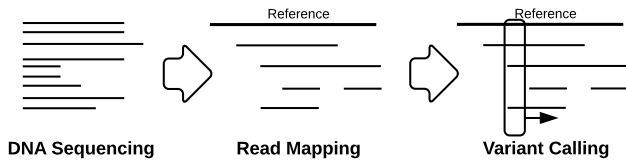


Figure 1: The genome analysis process in brief. DNA sequencers generate reads that are mapped against a reference. Afterwards, scientists analyze DNA sequencing data, e.g. they search for variants.

2.1 Genome analysis

A basic genome analysis pipeline consists of two main steps: (1) DNA sequencing & read mapping and (2) analysis of DNA sequencing data. In Figure 1, we depict the two main steps. In the following, we briefly introduce both analysis steps.

2.1.1 DNA sequencing & read mapping

DNA sequencing makes genetic information stored in DNA molecules digitally readable. To this end, DNA sequencers convert DNA molecules into sequences of the characters A, C, G, and T. Every single character encodes a nucleobase making up DNA molecules. As DNA sequencing is error-prone, every base is associated with an error probability that indicates the probability that the base is wrong [10]. The sequences of characters are called *reads* each associated with a so called *base call quality* string encoding the error probabilities of all bases in ASCII. DNA sequencing techniques are only able to generate small reads from a given DNA molecule [17]. Thus, these small reads must be assembled to reconstruct the original DNA. Common tools to assemble reads are read mappers [14]. Read mapping tools utilize known reference sequences of organisms and try to find the best matching position for every read. Read mappers have to deal with several difficulties such as deletions, insertions and mismatches within reads. Therefore, read mappings are also associated with an error probability.

2.1.2 Variant calling

Usually, scientists are not interested in mapped reads but in variations within certain ranges of the genome such as genes or chromosomes. Genome positions that differ from a given reference are called *variants*. Variants of single bases are a special class of variants and called *Single nucleotide polymorphisms (SNPs)*. Detecting SNPs is of high interest as these are known to trigger diseases such as cancer [15]. The task of *variant calling* is to first decide on a genotype that is determined by all bases covering a genome site. Then, the genotype is compared against the reference base. Two general approaches exist to compute the genotype [16]. One idea is to count all appearing bases at a genome site and decide based on the frequency which genotype is present. More sophisticated approaches incorporate the available quality information to compute the probabilities of possible genotypes. Overall, the computation of genotypes is an aggregation of bases that are mapped to the same genome site.

2.2 Variant calling in RDBMS

In this section, we briefly introduce the base-centric storage database schema to store DNA sequencing data in the

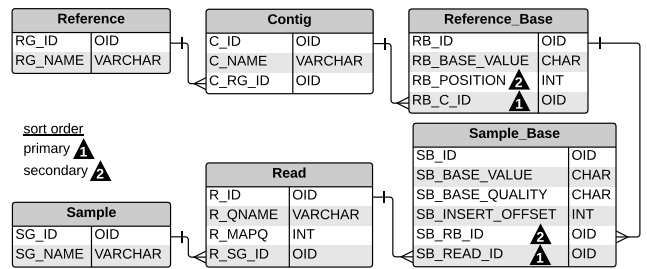


Figure 2: The base-centric database schema explicitly encodes every single bases of a read and genome region (contig) allowing for direct access via SQL.

form of mapped reads. Moreover, we explain how we can integrate variant calling, in particular SNP calling, into a relational database system via SQL to reveal the critical parts of the query.

2.2.1 Database schema

To represent mapped reads, we proposed the base-centric database schema [9]. It explicitly encodes the connection between single bases. Thus, it allows for direct access to all data via SQL that is required for base-centric analyses such as SNP calling. The schema consists of 6 tables. We depict it in Figure 2. Every **Reference** consists of a set of regions called **Contigs**. A contig can be a chromosome that consists of single **Reference_Bases**. For example, chromosome 1 of a human genome consists of nearly 250 million bases. A **Sample** consists of several mapped **Reads** which consist of single **Sample_Bases**. For example, the low coverage genome of the sample HG00096 provided by the 1000 genomes project consists of nearly 14 billion bases¹. Every **Sample_Base** is mapped to one **Reference_Base** which is encoded using a foreign-key relationship.

2.2.2 SNP calling via SQL

The base-centric database schema enables direct access to every single base within a genome. We can use the SQL query in Listing 1 to compute the genotype using a user-defined-aggregation function and finally call variants by comparing the result of the aggregation with the corresponding reference base. The query processing can be split into two separate phases:

- 1. Filter & Join.** In the first phase, we select bases of interest that are not inserted, have high base call quality and belong to reads with high mapping quality (cf. Lines 10–12) by joining the required tables (cf. Lines 6–8).
- 2. Aggregate & filter.** Finally, the genotype aggregation is performed using a user-defined aggregation function. Afterwards, the aggregation result is compared to the reference base to check whether a SNP is present or not (cf. Line 16).

3. MEMORY EFFICIENT SNP CALLING

In Section 2, we explained that variant calling within a relational database system consists of two main phases: *filter & join* and *aggregate & filter*. In this section, we discuss different implementations to execute these phases of the query.

¹<ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00096/>

```

1 select
2     C_NAME, RB_POSITION, RB_BASE_VALUE,
3     genotype(SB_BASE_VALUE) as genotype,
4     count(SB_BASE_VALUE) as coverage
5 from
6     Contig join Reference_Base on C_ID = RB_C_ID
7     join Sample_Base on RB_ID = SB_RB_ID
8     join Read on R_ID = SB_READ_ID
9 where
10    C_NAME = '?' and SB_INSERT_OFFSET = 0
11    and SB_BASE_CALL_QUALITY > ?
12    and R_MAPQ > ?
13 group by
14     C_NAME, RB_POSITION, RB_BASE_VALUE
15 having
16     RB_BASE_VALUE <> genotype and
17     coverage > ? and
18     RB_BASE_VALUE <> 'N' and genotype <> 'X'
19 order by
20     C_NAME, RB_POSITION;

```

Listing 1: SNP calling via SQL. A custom aggregation function is used to compute the domain-specific aggregate *genotype*.

In particular, we concentrate on the implementation of the join and aggregation. Moreover, we assess the alternative strategies according to their memory consumption that we want to reduce to improve the memory efficiency and inherently the scalability of queries to larger genome ranges.

3.1 Filter & join

Within relational database systems, joins are one of the most expensive operations. In this work, we use a hash join as baseline and compare it with the invisible join technique.

Hash join. The hash join approach consists of a build and probe phase. The idea is to index the tuples of one relation using a hash table. Afterwards, the join pairs are computed by probing the tuples of the second relation. One source of high memory consumption are the hash tables created during join processing, especially if the joined tables are large. To reduce the table size, the query optimizer pushes down selections before joining tables. Then, the filtered tables will be joined. Thus, before computing the join, all join columns are materialized. Consequently, predicates with low selectivity lead to high memory consumption.

Invisible join. An alternative join technique is the invisible join [1]. The basic idea is to use foreign-key relationships and positional lookups to compute matching join pairs. The technique was proposed in the context of processing data warehouse schemata such as the star- or snowflake schema. The base-centric database schema for storing DNA sequencing data is a star schema. Every primary key column starts at 0 and following primary keys are incremented. Hence, this setup enables using foreign keys on such primary key columns as row index to any column of the primary key table which can be efficiently implemented in column stores. Knowing the required tuples of the fact table, i.e. *Sample_Base*, we can simply gather all required data for downstream query processing using the foreign key columns *SB_READ_ID* and *SB_RB_ID*.

In order to make this join processing strategy efficient, we have to be able to apply *between-predicate rewriting* to express predicates on dimension tables as between predi-

cates on the fact table. We can apply *between-predicate-rewriting* to rewrite the predicate on *C_NAME* as between predicate on *SB_RB_ID* as we can guarantee that *Reference_Bases* are sorted by *RB_C_ID* and *RB_POSITION*. After applying the rewritten between predicate, we reduce the number of selected *Sample_Base* tuples further in a semi-join fashion. Finally, we compute the join starting at the fact table *Sample_Base* gathering all corresponding tuples from tables *Read* and *Reference_Base*. Thus, the invisible join keeps the memory footprint low as only the tuple identifiers of table *Sample_Base* are materialized and pruned.

Based on this qualitative discussion, we expect that the invisible join has less memory overhead than using a hash join. Moreover, the invisible join avoids building and probing hash tables, which should lead to further performance improvements.

3.2 Aggregate & filter

Two general approaches for computing aggregates are sort-based and hash-based techniques. In this section, we compare a sort-based approach with a special kind of hash-based aggregation called *array-based aggregation*.

Sort-based aggregation. Considering the query in Listing 1, the grouping clause is a superset of the order by clause. Thus, it seems to be beneficial to perform a sort-based aggregation as it directly leads to the required sorting of the result. Analyzing functional dependencies, we can reduce the set of group-by columns to the single column *RB_ID*. Thus, we can replace the group-by clause *C_NAME*, *RB_POSITION* and *RB_BASE_VALUE* in the SNP calling query with *RB_ID*. For that reason, we only have to sort a single column. Nevertheless, we have to reorder all columns that are aggregated. This leads to materialization overhead and additional memory consumption as the sorted and unsorted columns must be kept in memory.

Array-based aggregation. Knowing that the grouping is done via column *RB_ID*, we can apply array-based aggregation based on the idea by Krikellas et al. using mapping directories for grouping attributes to compute offsets within a grouping array [12]. As *RB_ID* is a primary key, we can also use it as array index for our aggregation, i.e. having a perfect hash function. Thus, we do not need mapping directories at all. Furthermore, we can guarantee that the *RB_ID* column reflects the ordering given in the order-by clause, thus, we inherently sort the data as required while we aggregate it. In case, the selected *RB_ID*s do not start at 0, we can determine the smallest *RB_ID* and use it as offset to fill the array starting at the first index. Thus, the additional memory consumption of this approach is determined by the size of the array for keeping the intermediate aggregates.

Based on this qualitative discussion, we expect that the array-based aggregation has less memory overhead than using a sort-based aggregation. Moreover, we expect that avoiding to sort saves additional runtime.

3.3 Summary

For each of the two phases, we presented two different techniques and discussed their properties regarding main-memory consumption. We can combine these four approaches leading to four different implementation stacks to execute the same query. Based on our discussion, we expect that the invisible join in combination with array-based aggregation will require less memory than all other combinations.

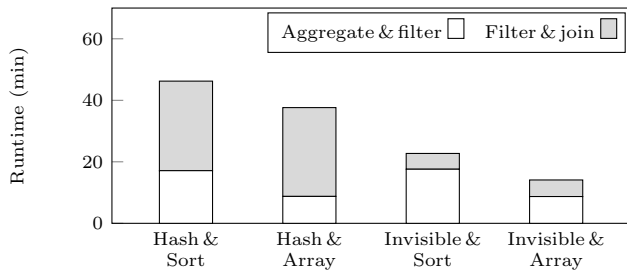


Figure 3: Breakdown of runtime performance of SNP calling over complete low coverage human genome data set. Invisible join in combination with array-based aggregation outperforms the baseline by factor 3.

4. EVALUATION

In this section, we evaluate the resulting four different combinations of join and aggregation strategies regarding their main-memory efficiency which we want to improve by reducing the memory consumption. We expect that reducing the memory consumption and, therefore, the materialization overhead, the runtime performance will increase. To this end, we measure the runtime when calling SNPs on a complete genome.

4.1 Experimental setup

As evaluation platform, we use a machine with two Intel Xeon CPU 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. Before starting the experiments, we pre-load the database into main memory. As evaluation database system, we use CoGaDB [4], a column-oriented main-memory database system².

For our experiments, we use human genome data sets of sample HG00096 from the 1000 genome project [18].

4.2 SNP calling runtime

In our first experiment, we call SNPs on a complete human genome that consists of ca. 14 billion sample bases. We perform the SNP calling on single chromosomes in order to not run out of memory as we are interested in the overall runtime of the approaches. We report the runtimes in Figure 3.

We observe that the array-based aggregation approach always reduces the runtime independent of the join approach. Using array-based aggregation, we avoid expensive sorting and reordering of columns as we can compute the required order on-the-fly (cf. Section 3.2). In combination with the invisible join, we can reduce the overall runtime of the join and aggregation phases by factor three compared to the slowest implementation using a hash join and sort-based aggregation. The runtime reduction can be explained by less overhead when computing the join as well as less materialization of intermediate results. When using the hash join implementation, all join columns are materialized before the join. Additionally, the sort-based aggregation requires sorting of the grouping column and reordering of aggregation columns.

²<http://cogadb.cs.tu-dortmund.de/wordpress/>

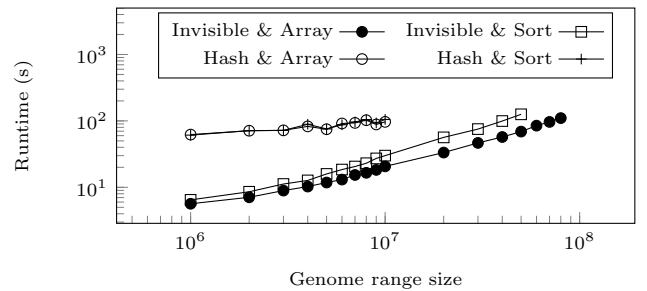


Figure 4: Scalability of different implementation combinations when calling SNPs on chromosome 1 of a high coverage human genome data set. Using array-based aggregation and invisible join, we can query larger genome ranges.

4.3 Memory efficiency

In our second experiment, we investigate the memory efficiency of all four implementation combinations by calling SNPs on a high coverage chromosome 1. This dataset consists only of chromosome 1 and contains ca. 11 billion sample bases. Thus, per genome site ca. 50 bases have to be aggregated. Using this data set, we can scale our query range from 1,000,000 genome sites up to 100,000,000. In case memory becomes scarce, the system interrupts the query execution. Thus, we can indirectly assess the memory efficiency of every implementation combination. We report the results of this experiment in Figure 4.

In accordance to the first experiment, the invisible join implementations are faster than the hash join implementations. On smaller genome ranges the runtime difference is up to one order of magnitude. The invisible join benefits more from higher selectivity given when querying smaller genome regions. The *between-predicate-rewriting* allows to prune the fact table `Sample_Base` drastically leading to less effort for gathering the join pairs.

Considering the scalability of the four approaches, we observe that using the invisible join, we can query genome ranges that are one order magnitude larger than using a hash join implementation at nearly the same speed. This confirms our expectations from Section 3 regarding memory efficiency of the invisible join. Additionally, the array-based aggregation allows us to query slightly larger genome ranges than using a sort-based aggregation until we run out of memory. Overall, the impact of the used aggregation technique on memory efficiency is smaller than those of the used join technique. The hash join implementation always dominates memory efficiency.

4.4 Summary

Our evaluation shows that for our use case, the invisible join technique in combination with array-based aggregation provides best memory efficiency and runtime performance. Moreover, we observe a coherence between scalability to larger genome ranges and processing runtime.

5. RELATED WORK

Within our work, we focus on using main-memory database systems for genome analysis use cases. In previous work, we mainly concentrated on expressing analysis tasks [9] and

showing the potential of such an approach [7, 8]. Within this work, we analyze the problem of memory scarcity during processing limiting the scalability of our approach.

We are aware of two other approaches that explicitly use main-memory database techniques to speed up genome analysis tasks. Fährnich et al. present an approach to integrate SNP calling using a map-reduce like processing scheme [11]. In contrast, our approach only uses relational database operators. Moreover, within their evaluation, Fährnich et al. use two computing nodes each having 1 TB of main memory for processing a data set with ca. 17 billion sample bases. Thus, the problem of memory scarcity during processing is unlikely to appear. Cijvat et al. use MonetDB to analyze Ebola data sets. Within their evaluation, they evaluate the runtime of different analysis queries [5]. Our work complements these approaches, in particular the MonetDB approach.

State-of-the-art approaches to perform SNP calling use specialized analysis tools that operate on flat files such as GATK [6] and samtools [13]. These tools are designed to efficiently operate in disk-based systems using machines with small main memory. Therefore, both tools rely on input data that is sorted by genomic region which is the grouping predicate for the aggregation. Thus, when reading data from disk, the tools can decide early when all data for a genomic site is read to compute the aggregate and write the result back to disk. Such an approach is not applicable in a main-memory only approach as data and results reside in memory. Moreover, in our approach, we have to perform the join computation due to schema normalization before we can aggregate the data.

6. CONCLUSION & FUTURE WORK

In this paper, we discuss different techniques for join and aggregation processing in the context of analyzing DNA sequencing data using main-memory database systems. Choosing the optimal implementation combination increases main-memory efficiency significantly and hence, allows for querying larger genome ranges or data sets.

Future work. We are aware of improvements regarding memory efficiency of in-memory hash joins [2, 3]. Therefore, in future work, it might be interesting to use such advanced in-memory hash join techniques. Moreover, we have not investigated the behaviour of sort-merge joins within this work. A second direction in order to increase the scalability of genome analysis tasks in main-memory database systems is to reduce the size of the primary database using lightweight compression, which frees main memory for processing. Finally, partitioning strategies have to be investigated.

7. REFERENCES

- [1] D. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.
- [2] R. Barber, G. Lohman, I. Pandis, et al. Memory-efficient hash joins. *Proc. VLDB Endow.*, 8(4):353–364, 2014.
- [3] S. K. Begley, Z. He, and Y.-P. P. Chen. Mcjoin: A memory-constrained join for column-store main-memory databases. In *SIGMOD*, pages 121–132, New York, NY, USA, 2012. ACM.
- [4] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, 2016.
- [5] R. Cijvat, S. Manegold, M. Kersten, et al. Genome sequence analysis with monetdb. *Datenbank-Spektrum*, 6(17), 2015.
- [6] M. A. DePristo, E. Banks, R. Poplin, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat Genet*, 43(5):491–498, May 2011.
- [7] S. Dorok, S. Breß, H. Läßle, and G. Saake. Toward efficient and reliable genome analysis using main-memory database systems. In *SSDBM*, pages 34:1–34:4. ACM, 2014.
- [8] S. Dorok, S. Breß, and G. Saake. Toward efficient variant calling inside main-memory database systems. In *BIOKDD-DEXA*. IEEE, 2014.
- [9] S. Dorok, S. Breß, J. Teubner, and G. Saake. Flexible analysis of plant genomes in a database management system. In *EDBT*, pages 509–512, 2015.
- [10] B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Research*, 8(3):186–194, 1998.
- [11] C. Fährnich, M. Schapranow, and H. Plattner. Facing the genome data deluge: efficiently identifying genetic variants with in-memory database technology. In *SAC*, pages 18–25, 2015.
- [12] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624. IEEE, 2010.
- [13] H. Li, B. Handsaker, A. Wysoker, et al. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, Aug. 2009.
- [14] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.
- [15] N. Mavaddat, S. Peock, D. Frost, et al. Cancer risks for BRCA1 and BRCA2 mutation carriers: Results from prospective analysis of EMBRACE. *Journal of the National Cancer Institute*, pages djt095+, Apr. 2013.
- [16] R. Nielsen, J. S. Paul, A. Albrechtsen, and Y. S. Song. Genotype and SNP calling from next-generation sequencing data. *Nat. Rev. Genet.*, 12(6):443–51, 2011.
- [17] M. Quail, M. Smith, P. Coupland, et al. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC Genomics*, 13(1):341+, 2012.
- [18] The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, Sept. 2015.