Implementation and Evaluation of Genome Type Processing for Disease-Causal Gene Studies on DBMS

Yoshifumi Ujibashi
Fujitsu Laboratories Ltd.
Kawasaki, Japan
ujibashi@jp.fujitsu.com

Motoyuki Kawaba
Fujitsu Laboratories Ltd.
Kawasaki, Japan
kawaba@jp.fujitsu.com

Lilian Harada
Fujitsu Laboratories Ltd.
Kawasaki, Japan
harada.lilian@jp.fujitsu.com

ABSTRACT

Recent development of next generation sequencer (NGS) and algorithms for genomic analysis are contributing to the understanding of human genetic variation and thus to personalized medicine. In leveraging this genomic data, the difficult task of finding out the genes relevant to dedicated phenotypes, e.g., disease-causal gene analysis, is becoming increasingly important. In a previous work, we have introduced a user defined function called "genome type" into PostgreSQL open source relational database management system (RDBMS) to accelerate genetic analysis with the aid of an efficient data structure in which all the genotypes are packed into one record.

However, there are still some challenges to be addressed in order to efficiently implement the proposed genome type when using real data. One problem is that, although the majority of variants are composed of three types of genotypes, this number is not fixed and can be highly skewed. Another problem is that the amount of genomic data necessary for accurate association analysis is huge and speed-up is necessary to make an iterative analysis feasible.

To solve these problems, we developed a new method which efficiently stores and processes variants of variable sizes. We also applied query parallelization techniques and exploited instruction level parallelization (SIMD) on Intel Xeon processor. Our performance evaluation shows that the processing of very large scale genomic data can be reduced to some few seconds.

1. INTRODUCTION

Due to recent technological innovations on genome sequencing and analysis, the speed and cost of acquiring genome information have drastically reduced and thus huge amounts of genome information has been collected. Human DNA consists of 3 billion DNA sequences and is represented by a sequence of four base AGCTs. Currently, it is said that there are tens of millions of variations such as SNP (Single nucleotide polymorphism), INDEL (insert-deletion), CNV (Copy Number Variation) that derive individual differences in their phenotypes. In recent years, disease-causal gene studies such like cohort studies and case-control studies that are based on statistical association analysis of gene variants and diseases, as well as lifestyle habits and physical characteristic have attracted much attention. In our work we focus on the large-scale statistical association analysis, as seen in Genome Wide Association Study (GWAS) that targets the entire genome-wide scale data, and propose new methods to speed up the statistical analysis of disease-causal gene study on relational databases.

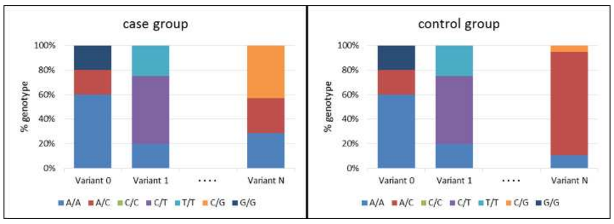


Figure 1: Example of genotype distributions

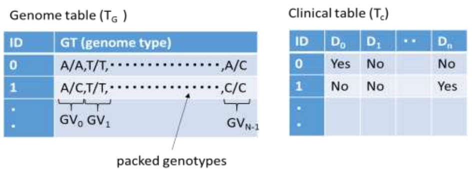
Disease-causal gene study aims at finding genotypes (types of gene variants) that are relevant to a target disease. This is done by dividing patients into a group with disease (case group) and another without disease (control group), and finding genotypes with different distributions between the two groups. For example, in Figure 1, there is no difference in genotype distribution between the case group and the control group for variants 0 and 1. However, since a significant difference can be seen in the distribution for variant N, this variant may be relevant to the target disease. The association between the target disease and the variants is calculated by performing a significance difference test under the null hypothesis (Cochrane-Armitage test, χ^2 test, etc.) for each gene variant, and when the p value falls below the significance level, the variant is considered relevant to the disease.

The processing required for the described association analysis is thus executed in two steps: (1) aggregating to generate the genotype distribution and (2) significant difference test. For the huge amount of genome data available nowadays step (1) requires much processing, and the total analysis may require some days if naively executed. Tools such as plink [3] address the statistical processing used in GWAS, and the performance of those analysis has shown improvements recently. In those tools, it is common to use genome information in a flat format file such as VCF, pad, or bad format [3]. However, in association analysis, the selection of the population greatly affects the correctness of the statistical result. Therefore, to search for meaningful results, an iterative approach where the selection of the population and the corresponding statistical processing are repeated a number of times is necessary to obtain correct statistical results. In order to repeatedly select and process data, it would be much easier if all the necessary data such as the genome data as well as the physical characteristic data (sex, race, age, etc.), the medical treatment data (presence/absence of disease, diagnosis results, etc.) related to the analysis were stored and managed in a single RDBMS not in flat format files. Recently, some studies that store and manage the whole necessary data in RDBMS have been proposed [4] [5], but none of the studies focus on high-speed processing of large-scale data on those RDBMS. It motivated us to speed up the processing on the data in RDBMS. In the previous work [1], we proposed a novel genome type and aggregate function as extended user-defined types of PostgreSQL and developed a new genotype data structure for efficient aggregate processing. However, in order to handle actual data, here we extend our previous work to handle variants with variable number of genotypes and to further speed-up the processing of large scale data.

In this paper, we first review our genome type and aggregate function in Section 2. Then, we introduce a new data structure to handle variants with variable size genotypes, and parallelization to handle large scale data in Sections 3 and 4, respectively. We finalize with some concluding remarks in Section 5.

2. Genome type and aggregate function

In our previous work [1], we have introduced a new genome type and aggregate function as extensions of PostgreSQL, and verified that those extensions enabled efficient genome analysis on RDBMS. Figure 2 illustrates genome table T_G where each row stores an individual ID and a field of genome type (GT) that packs the genotypes of all the N genome variants ($GV_0 \ldots GV_{N-1}$). Additional data on each individual such as clinical, demographic, lifestyle information, etc. can also be stored in other tables. Figure 2 illustrates a clinical table T_C that registers the disease of each individual. SQL 1 shows the SQL statement when executing aggregation using the genome type aggregate function fjgeno_count(). Since all the genotypes of the gene variants were packed in a single field, the genome aggregate function could efficiently count up through all genotypes of each individual at once.



ID	GT (genome type)
0	A/A,T/T,•••••••••••••••,A/C
1	A/C,T/T,•••••••••••••••,C/C
.	
.	

ID	D_0	D_1	••	D_n
0	Yes	No		No
1	No	No		Yes
.				
.				

Figure 2: Database schema of genome type

```
SELECT fjgeno_count(TG.GT) FROM TG
WHERE TG.ID = Tc.ID AND Tc.D0 = YES;
```

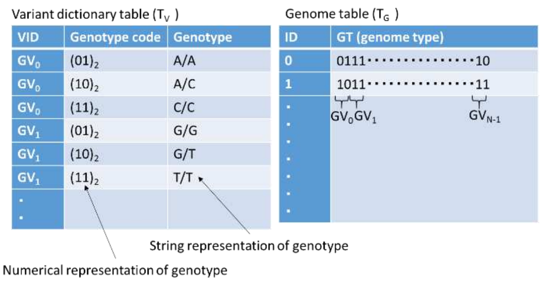
SQL 1: Statement with genome type aggregation function

3. New data structure for genome type

3.1 Dictionary-based encoding

3.1.1 Data structure

As illustrated in Figure 2, the genome type introduced in our previous work stored the genotype information in text format with all the genotypes as strings packed in one line and delimited by comma. In order to further improve its efficiency, here we introduce the utilization of a dictionary to compress data by numerically encoding the genotypes. Note that most gene variants can have a few variations of genotype patterns. For instance, a simple SNP can have three genotype patterns 'A/A', 'A/C' or 'C/C'. When represented in text format, a genotype 'A/A' plus a delimiter character would need four bytes. However, a numeric code requires only two bits and thus has a compression factor of 1/16.


Variant dictionary table (T_V)
Genome table (T_G)
GV$_0$GV$_1$
GV$_{N-1}$
String representation of genotype
Numerical representation of genotype

VID	Genotype code	Genotype
GV_0	$(01)_2$	A/A
GV_0	$(10)_2$	A/C
GV_0	$(11)_2$	C/C
GV_1	$(01)_2$	G/G
GV_1	$(10)_2$	G/T
GV_1	$(11)_2$	T/T
.		
.		

ID	GT (genome type)
0	0111•••••••••••••••10
1	1011•••••••••••••••11
.	
.	

Figure 3: Two-bit encoded genome type and variant dictionary table

Figure 3 illustrates a variant dictionary table (T_V) that, for each of the N variants ($GV_0 \ldots GV_{N-1}$), maps the genotype to its numeric code. In this example, each GV_i can have three types of genotypes ('A/A', 'A/C', 'C/C' for GV_0, 'G/G', 'G/T', 'T/T' for GV_1, …) and thus each GV_i can be encoded using two bits ($(01)_2$, $(10)_2$, $(11)_2$). Therefore the genome table T_G is represented as an array of 2N bits to store the genotypes of the N genome variants.

3.1.2 Performance of the new data structure

The proposed dictionary-based data structure also contributes to the acceleration of the aggregate processing by avoiding heavy text parsing processing. We measured the execution time of the query in SQL 1 using the genome type based on dictionary-encoding and on text developed in our previous work. We executed the query on a PRIMERGY RX2540 M1 machine with Xeon E5-2660 3 @2.60GHz, 576GB memory, and shared buffers = 128GB for the PostgreSQL configuration parameter, using data of 100,000 individuals with 100,000 variants. As shown in Table 1, while the execution time of aggregate processing takes about 46 seconds when parsing text, it takes only about 8 seconds using the proposed dictionary method, which results in more than 5x faster execution.

Table 1: Execution time of aggregate processing

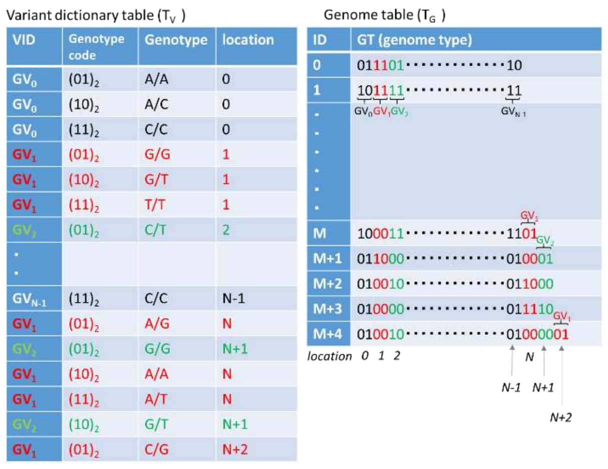
Genome Type	Execution time (sec)
Text parsing (Previous work)	45.743
Dictionary based (This work)	8.331

3.2 Support for variable genotypes

3.2.1 Dynamic data structure

As shown in 3.1, a dictionary encoding the genotypes to a fixed length bitwise code can compress data and improve the aggregate processing. As illustrated in Section 3.1, a large number of variants presents a small number of genotype patterns, namely three. However, even some simple SNPs and INDELs can present more than three patterns, and especially STRs (short tandem repeat) have many repeated patterns and so can present more than a hundred patterns. We can easily come up with two naïve methods to handle the variable number of genotype patterns. One way is to use a fixed length code that is long enough to represent the maximum number of possible genotype patterns among all variants. Another way is to use different length codes for each of the variants so that each variant code is long enough to represent the maximum number of possible patterns for that variant. However, note that both approaches are static and require a prior knowledge of the maximum number of patterns for the variants. In case a new individual's genome type is inserted with a new pattern that requires a longer variant code, both approaches require the reconstruction of the genome table as well as the dictionary. We believe that the preknowledge of the maximum number of genotype patterns is not practical in real genome analysis where new data emerges constantly. Since the reconstruction of dictionary and genome table are too heavy, we introduce a new dynamic method that efficiently handles new genotype patterns without prior knowledge.

Our dynamic approach handles new patterns that exceed the capability of initial bitwise codes by appending new bitwise codes at the end of our data structure. As illustrated in Figure 4, let's suppose that for the first M individuals, all the N genome variants contain only three patterns that are represented by two-bit codes $(01)_2$, $(10)_2$ and $(11)_2$. Therefore, for individuals 0 to M-1, GT is an array with two-bit spaces allocated to each of the N variants. Now, let's suppose that an individual M with a genome pattern 'A/G' for GV_1, that is different to the previous patterns 'G/G', 'G/T' and 'T/T' that appeared so far in GV_1, is inserted. In that case, the genome array for individual M is extended with a new two-bit space, and 'A/G' is encoded as $(01)_2$ for this new two-bit space for GV_1, while the original two-bit space for GV_1 is inserted with code $(00)_2$ indicating that the necessary code is stored



VID	Genotype code	Genotype	location
GV_0	$(01)_2$	A/A	0
GV_0	$(10)_2$	A/C	0
GV_0	$(11)_2$	C/C	0
GV_1	$(01)_2$	G/G	1
GV_1	$(10)_2$	G/T	1
GV_1	$(11)_2$	T/T	1
GV_2	$(01)_2$	C/T	2
.			
.			
GV_{N-1}	$(11)_2$	C/C	N-1
GV_1	$(01)_2$	A/G	N
GV_2	$(01)_2$	G/G	N+1
GV_1	$(10)_2$	A/A	N
GV_1	$(11)_2$	A/T	N
GV_2	$(10)_2$	G/T	N+1
GV_1	$(01)_2$	C/G	N+2

ID	GT (genome type)
0	011101··············10
1	101111··············11
.	
.	
.	
.	
.	
M	100011··············1101
M+1	011000··············010001
M+2	010010··············011000
M+3	010000··············011110
M+4	010010··············01000001

Figure 4: Adding genotype patterns

elsewhere in a new two-bit space. Note that the dictionary T_v is updated and a new entry gives that 'A/G' is encoded as $(01)_2$ for GV_1 and the code is located at the Nth two-bit space in GT array.

Analogously, the procedure continues as follows:

- ✓ When a fourth genotype pattern 'G/G' newly appears for GV_2 by inserting individual M+1, another space is added for GV_2 following the second space for GV_1 with code $(01)_2$, and the first space for GV_2 is filled with $(00)_2$;
- ✓ When a fifth pattern 'A/A' newly appears for GV_1 by inserting individual M+2, the code $(10)_2$ is stored in the existing second space for GV_1, and the first space for GV_1 is filled with $(00)_2$;
- ✓ When a sixth pattern 'A/T' for GV_1 and a fifth pattern 'G/T' for GV_2 appear by inserting individual M+3, they are stored in the existing second spaces for GV_1 and GV_2 as codes $(11)_2$ and $(10)_2$, and the other spaces for GV_1 and GV_2 are filled with $(00)_2$, respectively;
- ✓ When a seventh pattern 'C/G' newly appears for GV_1 by inserting individual M+4, a third space is needed for GV_1. In this case, the third space for GV_1 containing $(01)_2$ is added following the second space for GV_2, and $(00)_2$ is inserted into the first and second spaces for GV_1;

Note that all these information are registered on the variant dictionary table T_v that is extended with a location field that gives the two-bit space in which a pattern for each of the variants is found. The physical address of such location is decided when loading the genome data, and then when new patterns are inserted and new space is needed to store them.

In our method a fast array access is feasible because of its fixed length bitwise codes, and although the information for one variant is distributed among several spaces, each summation is efficiently processed over two-bit fixed elements. After the counts for each two-bit fixed element are processed, they are aggregated to generate the final counts for each variant by using the information on the variant dictionary table.

3.2.2 *Efficiency of the dynamic data structure*

To evaluate the efficiency of this method, we run the query on SQL 1 using the data distribution shown in Table 2. It emulates actual data containing 90,000 normal SNPs with three genotype patterns, 9900 irregular SNPs with six genotype patterns, and 100 STRs with 55 genotype patterns.

Table 2: Example of genotype size distribution

Counts	# of genotype pattern	Supposed variant type
90000	3	2-allelic SNPs
9900	6	3-allelic SNPs
100	55	STRs

Table 3 shows the processing times when all variants have only three patterns (Fixed), and for the case of Table 2 (Variable). We found that even for the case of variable variant patterns, the processing time has only a 20% increase over the "ideal" case of fixed three patterns, which is proportional to the increase in table size. Therefore, we found that the proposed dynamic approach is very efficient in handling genome data without any prior knowledge on the number of variant patterns.

Table 3: Times for fixed and variable patterns

	Fixed	Variable
Time (sec)	8.331	9.958
Size (GB)	2.70	3.32

4. Acceleration by parallelization

4.1 Query parallelization

With the aid of the increasing number of processors and cores in one machine, parallelization is a very effective way for processing speed acceleration. Here we show how the processing of our proposed genome data structures on PostgreSQL could take advantage of parallel processing. Since parallel query capabilities were introduced in the latest release 9.6 of PostgreSQL [2], we run the query SQL 1 on PostgreSQL 9.6. However, as a result of our trial, we found that the introduced parallelization is not effective for performance acceleration of that query. Figure 5 shows which processing are parallelized in the query execution plan created when running the query of SQL 1. We can see that PostgreSQL 9.6 parallelizes the join between

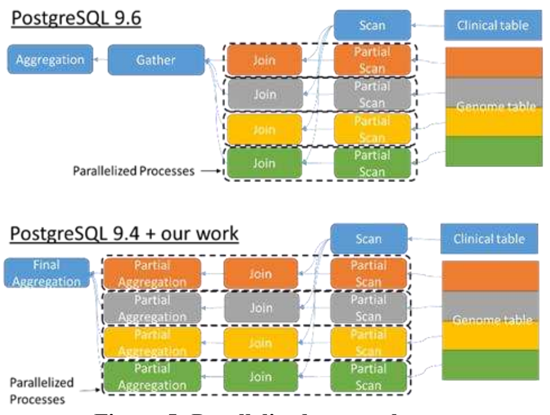


Figure 5: Parallelized query plan

the genome table and the clinical table, however the subsequent aggregate of the large join result is not parallelized and thus can become the bottleneck of the total query processing.

As we show later in Section 4.3, a more efficient parallel execution was achieved when using our original parallel function developed in a previous work [6] based on PostgreSQL 9.4. As shown in Figure 5, this function parallelizes the query execution plan from the topmost plan and thus both join and aggregation are parallelized, resulting in the acceleration of the total performance of the query.

4.2 SIMD processing

In addition, SIMD instructions of the processor could be used to parallelize the processing in the CPU instruction. Currently we can run 256-bit wide vector processing utilizing AVX2 which has been equipped from Sandy Bridge generation of Intel Xeon processors. The aggregate processing is mainly composed of two steps: taking the genotype patterns stored in each variant at the genome type structure, and then, counting the corresponding variant elements in a summation array. In order to efficiently realize the vector processing, we introduce a lookup array which stores vectors corresponding to the pattern values of the genome type and that should be added to the summation array. Figure 6 illustrates how the count-up processing uses the lookup array. First, the value $(00010110)_2$ composed of four variant's pattern codes is taken from the genome type. The decimal value of $(00010110)_2$ is 22 and thus, the 22th element of the lookup array gives the vector with the bits where places corresponding to the four variant's pattern values (GV_n's $(00)_2$, GV_{n+1}'s $(01)_2$, GV_{n+2}'s $(01)_2$, GV_{n+3}'s $(10)_2$) are set to 1. Then, one counting cycle is done by adding the vector to the summation array with SIMD instruction. Note that all the 256 vector values that corresponds to the composition of the four variant's pattern codes are prepared on the lookup array in advance.

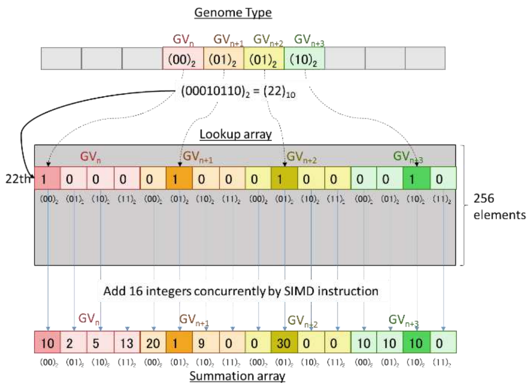


Figure 6: SIMD processing using lookup array

Because the element size of the summation array has to fit one SIMD vector, its size is restricted to 16-bit width. Thus when any element in the summation array achieves the maximum value of 65535 in integer, all the element values are added to a "final summation array" whose elements are wide enough. The summation array elements are then cleared to continue the aggregate processing.

4.3 Performance evaluation

In this section, we evaluate how the parallel processing of the newly proposed genome type accelerates the aggregate processing. We used the same experimental environment in Section 3.1.2. The results are shown in Figure 7 when using PostgreSQL 9.6 and 9.4 extended with our parallel function.

We can see that for one core, the newly released PostgreSQL 9.6 was improved from the older 9.4. However, as we explained in 4.1, since PostgreSQL 9.6 cannot parallelize the aggregate that represents the heaviest operation in the query, its total time does not scale when increasing the number of cores. On the other hand, for PostgreSQL 9.4 with our parallel extension, the execution time of 8.3 seconds on single core improves to 2.1 seconds on eight cores, i.e. about four times faster. In addition, using SIMD instructions, the performance improves by 20% compared with simple aggregate processing using dictionary encoded genome type.

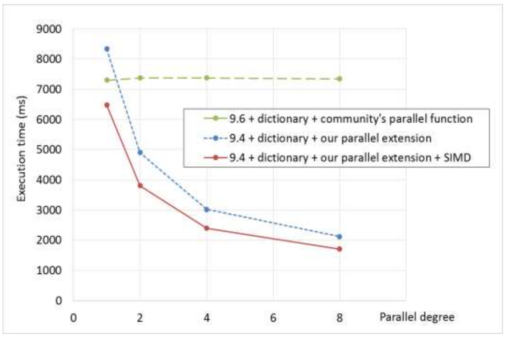


Figure 7: Execution time of aggregate processing

5. Conclusion

In this paper, we present our new efforts on accelerating genomic analysis to deal with actual large-scale data. The dictionary-based data structure described in 3.1.2 enables dealing with genotype patterns of variable maximum number and resulted in 5 times faster aggregate processing. And as shown in the results in 4.3, query parallelization with 8-cores and SIMD processing resulted in 5 times faster aggregate processing, and thus resulting in a total acceleration factor of 25x. This results in an execution time of less than two seconds for the aggregation of genome variants for genome information of 100 thousands individuals with 100 thousands variants. In recent studies, it is reported that more than 10 million variants have been included in the human genome. For such huge data, our aggregate processing would require some hundreds seconds. We believe that meaningful association studies requires a try and error approach where a variety of conditions are run iteratively. We believe our method could contribute to the feasibility of such an iterative approach for genome analysis.

6. REFERENCES

[1] Y. Ujibashi, M. Kawaba, L. Harada (2015) Proposal of Database Type and Aggregation Function for Accelerating Medical Genomics Study on DBMS EDBT 2016: **672-673**

[2] The PostgreSQL Global Development Group. (1996-2016) *PostgreSQL* http://www.postgresql.org/

[3] C. C Chang, C. C Chow, L. CAM Tellier, S. Vattikuti, S. Purcell, J. J Lee (2015) Second-generation PLINK: rising to the challenge of larger and richer datasets. GigaScience, **4**.

[4] A. Ameur, I. Bunkikis, S. Enroth, et al. (2014) CanvasDB: a local database infrastructure for analysis of targeted- and while genome resequencing projects. *Database*, Vol. 2014, Article ID bau098

[5] U. Paila, B. A. Chapman, R. Kirchner (2013) GEMINI: integrative exploration of genetic variation and genome annotations. *PLOS Comput. Biol.*,**9**,e1003153

[6] Y. Ujibashi, M. Nakamura, T. Tabaru., T. Hashida, M. Kawaba, L. Harada.: Design of a Shared Memory mechanism for efficient parallel processing in PostgreSQL. IISA 2015: 1-6