Bioinformatics, 2016, 1–8
doi: 10.1093/bioinformatics/btw199
Original Paper

Sequence analysis

GORpipe: a query tool for working with sequence data based on a Genomic Ordered Relational (GOR) architecture

Hákon Guðbjartsson[1,*], Guðmundur Fr. Georgsson[2], Sigurjón A. Guðjónsson[2], Ragnar Þór Valdimarsson[2], Jóhann H. Sigurðsson[1], Sigmar K. Stefánsson[1], Gísli Másson[2], Gísli Magnússon[1], Vilmundur Pálmason[1] and Kári Stefánsson[2]

[1]WuXiNextCode Genomics, Sturlugata 8, 101 Reykjavík, Iceland and [2]deCODE Genetics, Sturlugata 8, 101 Reykjavík, Iceland

*To whom correspondence should be addressed.

Associate Editor: Inanc Birol

Received on October 10, 2015; revised on March 16, 2016; accepted on April 11, 2016

Abstract

Motivation: Our aim was to create a general-purpose relational data format and analysis tools to provide an efficient and coherent framework for working with large volumes of DNA sequence data.

Results: For this purpose we developed the GORpipe software system. It is based on a genomic ordered architecture and uses a declarative query language that combines features from SQL and shell pipe syntax in a novel manner. The system can for instance be used to annotate sequence variants, find genomic spatial overlap between various types of genomic features, filter and aggregate them in various ways.

Availability and Implementation: The GORpipe software is freely available for non-commercial academic usage and can be downloaded from www.nextcode.com/gorpipe.

Contact: hakon@wuxinextcode.com

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Rapid advances in high-throughput sequencing technology have introduced formidable data management challenges for bioinformaticians. Conventional relational database management systems (RDBMS) are ill suited to the task of dealing with high volume of sequence reads and sequence variation data, partly because of their requirements for fine-grained transactions and partly because of their legacy command set and data structures. In the field of bioinformatics, the exodus from the on-hand relational databases has introduced a myriad of specialized tools and file formats, that are often incompatible, causing much computation and IO being spent on data conversions and developer's time to master a large set of disparate tools.

The work presented here originates over 5 years back within deCODE's large scale population based sequencing project in Iceland (Gudbjartsson *et al.*, 2015). The aim was to create general-purpose relational data format and analysis tools in order to be able to annotate sequence variants, find genomic overlap between various types of features, filter and aggregate them in various ways. Essentially, a replacement for the conventional RDBMS; capable of handling sequence and variation data from hundreds of thousand samples and analyze them in the context of other genomic features.

Having worked with very large SNP genotype datasets in the past, our key observation was that in order to support many of the envisioned usage scenarios, such as range queries, data joins and aggregation, we would need a genomic ordered data architecture, where the analysis algorithms largely treat the input data as ordered streams.

Furthermore, for on-going sequence analysis, such as in clinical practice or in very large research projects, where practically unlimited number of variations is captured with modern sequencing technology, sparse data formats are needed to represent the sequence variations, as compared to the finite grid structures used for chip-based SNPs in the GWAS era. Rather than using the convention of accessing the data via an index, based on an ID scheme such as rsIDs, the variation data has to be identified and accessed solely based on its position and the sequence difference from a particular reference genome build.

In order to achieve the above, we developed a system based on a genomic ordered relational architecture; a GOR format and a query system, using a declarative relational query language, which syntax is a hybrid of Unix shell command pipe syntax and SQL. By design, our system works only for the analysis downstream of the sequence read alignment step, and while it does incorporate some functions for variation calling, the primary focus has been to develop query tools for tertiary analysis.

While the GOR data format shares many features with other earlier formats such as BED (Quinlan and Hall, 2010) and BAM (Li *et al.*, 2009) and the more general purpose format Tabix (Li, 2011), it differs by being an *index-free* tabular storage format, where the first two columns representing the genomic position must be consistently ordered to allow for a binary-like seek. Similarly, when it comes to the data layout, we emphasize relational normalization against de-normalization and columnar storage against excessive usage of attribute-value columns. This we do based on the same arguments that have been extensively used in the literature on RDBMS when preaching good relational schema designs for ease of data updates and querying.

As to be expected, our GORpipe program overlaps with many tools that have been developed in recent years such as BEDtools (Quinlan and Hall, 2010), Galaxy (Giardine *et al.*, 2005), VCFtools (Danecek *et al.*, 2011), GATK (McKenna *et al.*, 2010), ANNOVAR (Wang *et al.* 2010) and SNPeff/SNPsift (Cingolani *et al.*, 2012a, b) to name few. For instance, many of these tools include features for intersecting and annotating variation data. Our design is, however, very different in the sense that it is more of a query system or a language based approach. In this regard, it has similarities with the work on the genome query language (GQL) (Kozanitis *et al.*, 2014).

Our approach does indeed follow the Unix design philosophy of combining 'many small sharp tools' with clear and simple interfaces. That being said, we made a conscious decision not to implement the system using Linux shell commands and rely on the operating systems pipe mechanism, although we also recognized many benefits of using that approach. Rather, we implemented our own multi-threaded command pipe framework, which runs in a single JVM instance, allowing us to create lightweight pipes with full control. This enables us to create efficient queries consisting of a very high number of commands (pipe steps) and importantly, to create declarative and nested streams which are *seekable*, unlike regular process substitutions in shells. This allows us also to optimize the query execution plan, although our current implementation does nowhere near exhaust the possibilities on that frontier.

In the rest of this paper we introduce the GOR architecture, briefly describe our tab-delimited storage format and explain how other genomic ordered tabular formats can be used with our system. To do this, we introduce the basic GOR command, which reads the data from one or more files. We also introduce GOR tables (dictionaries), which provide a relational abstraction to equivalent data in multiple files.

We then give an informal description of the GOR query language. A formal definition of the syntax and a detailed implementation description is, however, outside the scope of this work. Rather, we like to demonstrate the unique nature of the GORpipe system and some of its commands through selected examples, something we think is more intuitive and valuable for practicing bio-informaticians. Additionally, we briefly describe a related NOR syntax, for non-ordered relational data, which together with the GOR syntax, provides a powerful framework for working with both clinical and genetic data.

2 The GOR data architecture

2.1 The GOR format

The initial use of the GOR format at deCODE was to provide a general-purpose storage format to represent feature tracks in a genome browser. There is a vast literature on various approaches to provide fast stabbing and range queries for interval data, such as with partition or genome binning (Kent *et al.*, 2002), with tree index structures (Enderle *et al.*, 2004) and bitmap-indexing (Egilsson and Gudbjartsson, 2003). For the purpose of genome browser interval queries, we had observed, that simply by ordering genomic features based on their starting position, it allowed us to retrieve them quickly for all practical purposes by a binary-like search. By knowing the maximum span size of the features at hand, we could easily offset the seek, to ensure that the subsequent scan of the data retrieves all the features which partially overlap the view range in the genome browser. The penalty paid for this approach is that following the seek, it may retrieve features which do not overlap the range of interest. While this is true, for most real case scenarios, when the feature size is large, the number of features that need to be scanned is typically small, if the track contains features that are fairly uniform in size. Another benefit of this simple indexing approach is that it suggests a relatively simple implementation scheme for *seekable* nested streams, which we will introduce later.

Also, for whole genome query analysis, one often needs to be able to perform genome spatial joins on a large number of features quickly. It is well recognized that for such queries, index-look-up does not perform very well (Enderle *et al.*, 2004), and the key is to be able to stream the data with minimal random access. This calls for a genomic ordering of the data; something which was clearly also recognized by the developers of the BAM (Li *et al.*, 2009) and Tabix (Li, 2011) format specification. However, unlike in their approach, to keep things simple and for the reasons mentioned above, we choose not to store additional segment index for GOR files, but rather let the data itself be the index. Hence, the GOR format is strikingly simple and hardly deserves a special description. It is simply an ordered tab-separated table (TSV file) with optional comment lines in the header as show below:

```
##Build=hg19
```

#Chrom	Pos	Col1	Col2	...	ColN
chr1	1	x	y		z
chr1	2	xx	yy		zz
...					
chr10	1	xxx	yyy		zzz
chr10	3	xxxx	yyyy		zzzz
...					
chr11	1	a	b		c
...					
chrY	100	aa	bb		cc

Optionally, the file can begin with comments and a header line analogous to the VCF specification (Danecek *et al.*, 2011). The GOR format standard does however mandate alphabetical ordering of the first column and a numerical ordering of the second column.

This is to make the ordering scheme generic; not base it on a particular ordering such as for the human chromosomes. For them, the GOR standard uses chr1, chr10 and chrX as compared to 1, 10, X, etc. However the GOR ordering scheme is not restricted to the human chromosomes, since it is alphabetical on the sequence name.

The GOR command can be used to read data from a file, like the cat command in Unix/Linux. Unlike the cat command, the GOR command allows a specific range from the genome to be read with the -p option:

Example 1. Reading a slice from `file1.gor` on chromosome 10

```
gorpipe "gor -p chr10:1-3 file1.gor"
```

returns the output shown below:

#Chrom	Pos	Col1	Col2	...	ColN
chr10	1	xxx	yyy		zzz
chr10	3	xxxx	yyyy		zzzz

When the GOR command is used to seek into a file or retrieve a slice of rows, a binary-like seek can be performed because the genomic position is consistently stored in the first two columns of each row. If the rows are evenly distributed, a seek method that uses a linear interpolation to estimate the next file position access, performs considerably fewer random access reads than a regular binary seek.

We have also extended GOR into a block compression format, denoted GORZ, which still allows for a binary-like seek without uncompression, because the positional information for the last row in each block is stored outside of the compressed data. Unlike the simple GOR format, which files can be written with pretty much any software tool, a special driver is needed. For instance, we can use GORpipe to create a zipped version of `file1.gor`:

Example 2. Writing file1.gor to a compressed gor file

```
gorpipe "gor file1.gor | write file2.gorz"
```

Another striking difference between the GOR command and the cat command is that if two ordered files are merged the output remains ordered. From now on only writing the quoted part for the GORpipe command:

Example 3. Reading a slice from file1.gor and file2.gorz

```
gor -p chr10:1-3 -s FileName file1.gor file2.gorz
```

returns the output shown below:

#Chrom	Pos	Col1	Col2	...	ColN	FileName
chr10	1	xxx	yyy		zzz	file1.gor
chr10	1	xxx	yyy		zzz	file2.gorz
chr10	3	xxxx	yyyy		zzzz	file1.gor
chr10	3	xxxx	yyyy		zzzz	file2.gorz

Thus, the GOR command performs merge, analogously as done in a merge-sort, however, without of having to sort each file (input stream) since they are assumed to be genomic ordered.

2.2 Other related ordered formats

GORpipe has drivers to read data directly from other commonly used formats such as BAM, VCF and Tabix. Both BAM and VCF store the genomic positions in the first 2 columns like the GOR format; however, these formats do not specify a strict ordering scheme and naming convention. In order to avoid rewriting of these files, to comply with the GOR ordering scheme, the GOR command has a driver which detects if the chromosomes are numbered 1, 2, 3, ... as compared to chr1, chr11, chr12, ... and outputs the rows in an order consistent with the GOR specification. The same is done for Tabix files, but additionally the column ordering is changed for GFF/GTF formats (if the ending is .gff.gz or .gtf.gz), such that the start and stop columns are in the second and third column instead of 4 and 5.

2.3 GOR tables

Although modern RDBMS support a very high number of tables and columns, it is not a custom to organize data from say different patients into separate columns or tables. There are several reasons for this, an important one being that the standard SQL query language does not provide powerful means to formulate queries where the metadata (such as table names and columns) is constrained using expressions. Special query languages such as SchemaSQL (Lakshmanan *et al.*, 1996) have been introduced for this type of queries, however, they are not available in conventional RDBMS.

We wanted to be able to provide a data model abstraction where for instance all the variation data from all available subjects can be considered to reside in a single table, each row representing a single sequence variant from one specified individual. For this we introduce GOR-tables, which are essentially simple metadata files mapping files to tag-values. As an example consider the file `variations.gort` shown below, which associates variation files of individuals with patient IDs:

#File	PID
sampleA.vcf.gorz	A
sampleB.vcf.gorz	B
sampleC.vcf.gorz	C

One can think of the file `variations.gort` as a table partitioned by patient IDs. For instance, if we would like to read all the variations from patients A and C on chr2, we could write as shown in Example 4:

Example 4. Reading variations for patients A and C on chrom 2.

```
gor -p chr2:1000- variations.gort -f A,C
```

The above command might return an output looking something like:

#Chrom	Pos	Ref	Alt	Zygosity	Depth	PID
chr2	1000	C	A	Hom	33	A
chr2	1000	C	A	Het	25	C
chr2	1112	C	CA	Het	30	C
chr2	1150	G	A	Het	29	A
...						

The above example demonstrates how a single table can be used to access variants from any patients, in this case subjects with PIDs A and C. An equivalent SQL expression could look something like

```
select * from variants where chrom = 'chr2' and
pos >= 1000 and PID in ('A','C')
```

Although the GORpipe syntax support a WHERE command, currently we use an option -f to instruct the GOR command and its corresponding file driver which data to access, as compared to the more sophisticated approach of deriving that from the predicates; an approach typically taken in advanced SQL systems. In the next sections on the GORpipe syntax, we will however show how this approach can be used effectively to determine what data to retrieve from GOR tables, such as the variation table described above.

3 The GORpipe syntax

3.1 Pipe streaming

In previous section, we introduced the basic GOR command to read GOR data from one or more sources. As in Unix, the output can then be piped into other commands, e.g.:

Example 5. Pipe syntax in GOR queries

```
gor genes.gor | where gene_symbol in ('BRCA1','BRCA2')
| select chrom,gene_start,gene_stop
```

The WHERE command filters the stream from the GOR command based on the column gene_symbol and then the SELECT command projects the stream, i.e. picks only the three columns representing the span of the genes. Similar things could be achieved in Linux shell using commands such as GREP and AWK. However, the GOR commands are typically much easier to use since they both allow reference to columns based on their number (e.g. #1-#5) and their names.

As another example, consider a query that calculates the number of variations, per megabase, in the table variations.gort:

Example 6. Counting variation rows per megabase

```
gor variations.gort | group 1000000 -count
```

The GROUP command derives its name from a similar aggregate group by clause in SQL. In GOR, grouping is always based on a sliding genomic bin size (can be overlapping). Rather than specifying an integer to denote a range in base pairs, it is also possible to group per chromosome or genome. As an example, to count the number of variations per genome we can write:

Example 7. Counting variation rows over all the genome

```
gor variations.gort | group genome -count
```

The GROUP command always returns it aggregate results as annotations on genomic segments and it uses 'chrA' to denote a segment spanning the entire genome, because each command must output GOR data.

If we want to calculate the number of variants per patient, we can specify additional grouping using the -gc option with the PID column:

Example 8. Counting all variations per subject

```
gor variations.gort | group genome -gc PID -count
```

The GORpipe syntax has a CALC command to add a new calculated column. For instance, if we want to divide the count between transition and transversion SNPs and calculate their ratio, we can write:

Example 9. Calculating transition-transversion ratio

```
gor variations.gort | where len(ref)=1 and len(alt)=1
| calc transition=if(ref+'>'+alt
      in ('A>G','G>A','C>T','T>C'),1,0)
| calc transversion=1 - transition
| group genome -gc PID -sum -ic transition,transversion
| calc TiTv_ratio float(sum_transition)/sum_transversion
```

We see how the WHERE command is used to filter out non-SNPs based on the length of the ref and alt columns. The first CALC command creates the column transition; which takes the value 1 or 0 based on the nature of the SNPs. Likewise, the column transversion is calculated as its complement. The GROUP command then sums the two integer columns, transition and transversion over the genome per PID value and the final CALC command calculates the floating point ratio of the two sums.

The system infers the data type of columns (int/float/string based on the first 10k rows), however, it is also possible to enforce the type, such as with the float function. The other thing to notice is the automatic naming convention, e.g. the -sum operator prefixes the columns with sum_ and likewise the -count option always returns a column named allCount (reference to columns is however case-insensitive). If the naming of a column collides with existing column, this is resolved by adding an 'x' to the column name, repeated until the added column has a unique name.

3.2 Joins

The examples in previous sections have only covered commands using a single relation GOR stream. One of the key design principles in relational databases is to split data based on its nature into multiple relations and to have effective means of joining them together. The following example finds SNPs in GWAS dataset that are within 20 bp distance from exons by joining together two tables:

Example 10. Joining exons with SNPs from GWAS results

```
gor exons.gor | join -segsnp -f 20 snp_gwas_results.gorz
```

Notice that the JOIN command has an option to specify the join type. In the above case, it is segments against single nucleotide position (SNPs). In the GOR format, it is mandatory to have the first two columns chromosome and position. For segments the position represents a zero-based start position (using the UCSC convention) while for SNP data (and VCF data) it is by convention a one-based position. The end position for segments can in principle be in any other column, however, it is strongly recommended that the third column is used and by default all commands assume that.

In the output of the above query, the first two columns represent the start position of the exons, which are guaranteed to be in correct GOR order, while the position of the SNPs follows the last column in the left-source (exons.gor). If we change the join-order, the SNP position would occupy the first two columns and the 'begin' and 'end' positions of the exon segments would follow the last column in the GWAS table. Now it depends on the number and density of features in the left- and right-source, which join-order performs better. Unlike in RDBMS and SQL, the GORpipe system always lets the left-source 'control' the join. We can modify the query and reorder the columns using the SELECT command:

Example 11. Possibly spoiling genome order by moving columns

```
gor exons.gor
| join -segsnp -f 20 -rprefix gwas snp_gwas_results.gorz
| select chrom,gwas_pos,gwas_rsid,gwas_pval,
                    gwas_phenotype,gene_symbol
| verifyorder
```

Notice that in the join, we use the -rprefix option to prefix the columns from the right-source with gwas_. In practice, exon segments from different genes may overlap, therefore the position column of the GWAS SNPs is no-longer guaranteed to be in genomic order and indeed the VERIFYORDER command would most likely raise exception (will depend on the right-source as well). We can however use the SEGSPAN command to modify the left-source to a non-overlapping segment stream, which has the same span as the exon input stream:

Example 12. Projecting the left-source with SEGSPAN ensures genomic order

```
gor exons.gor | segspan
| join -segsnp -f 20 -rprefix gwas snp_gwas_results.gorz
| select chrom,gwas_*
```

Now the ordering of the SNP position column is guaranteed to be correct. Notice however that since we used the SEGSPAN command, we can no-longer use the gene_symbol column from the exon source. Alternatively, instead of using the SEGSPAN command we can 'correct' the ordering after the selection:

Example 13. Using a sliding window sort to guarantee genomic order

```
gor exons.gor
| join -segsnp -f 20 -rprefix gwas snp_gwas_results.gorz
| select chrom,gwas_*,gene_symbol
| sort 100000
```

A key thing to notice here is that the SORT command takes a parameter denoting a window in base pairs, which instructs it 'how incorrect' the ordering is. Thus, the sort is like a sliding window operation, which corrects the ordering over the window size. Although it would also have been possible to write 'sort chrom' or 'sort genome', it would be much less efficient since the SORT command would have to block the stream until it has all the data for a whole chromosome or the whole genome, respectively. Because the largest exon is around 90 kb, we know the maximum deviation from correct genomic order in the SNP position column of the right-source is less than 100 kb, i.e. deviation from GOR due to the overlap of exon segments and the -segsnp type join. Thus, we can use a highly efficient *stream aware* sort. Importantly, if we would not correct the ordering using sort, subsequent commands such as JOIN, GROUP, PIVOT or RANK would not work, because their implementation relies on consistent GOR ordering, to allow for maximum efficiency and minimal memory footprint.

When the query in Example 13 is executed, it will depend on the density of the data in the left- and right-source whether the query engine will stream the data or use seek involving random access. If we constrain the exons to come only from genes, which symbol starts with BRC, then most likely the system will use seek:

Example 14. A sparse left-source causing seek in right-source

```
gor exons.gor | where gene_symbol ~ 'BRC*'
| join -segsnp -f 20 -rprefix gwas snp_gwas_results.gorz
| select chrom,gwas_*,gene_symbol
| sort 100000
```

Even if the file snp_gwas_results.gorz has tens of millions of rows, this query will return in sub-second time since only exons from three genes satisfy the gene symbol predicate. This is indeed an example where we have very high selectivity on the left-source.

Now lets turn this around and take an example where there is very high selectivity on the GWAS data, e.g. filtering based on a strong phenotype association. In this scenario, we are better off writing the query in this manner:

Example 15. A sparse left-source causing seek in right-source

```
gor snp_gwas_results.gorz | where pval <=1.0e-10
| join -snpseg -f 20 -maxseg 100000 exons.gor
|select chrom,pos,rsid,phenotype,pval,gene_symbol
| prefix pos-pval gwas
```

The last PREFIX command is simply to keep the naming of the columns consistent with the former queries, e.g. prefix column range from pos to pval with gwas_. Notice also that now the join type is -snpseg. Again, it will depend on the density of the sources weather the query engine will simply stream the data or perform seek on the right-source. In the latter event, and when the right-source is segments as in the above case, the simple 'indexing approach' used in GOR, i.e. to order the data by the starting position of the features, requires that the system seeks in a manner such that all segments which cover the SNPs are retrieved, taking into consideration their size and the fuzz-factor specified with the -f option.

Hence, for join types such as -snpseg and -segseg, the system must know the maximum segment size in the right-source. While this information can be stored in a metadata file (e.g. spatial index structure) and specified per region in the genome, for simplicity we have chosen not to rely on such metadata file in GORpipe, although our initial use of the GOR format in our genome browser did so. Rather, we allow the user to specify a -maxseg option to define the maximum span of segments in the right-source. If this option is not provided and the right-source is a file, the query engine will scan the the file to derive this number. When the right-source is a nested query (process substitution), which will be described in Section 3.3, with the default configuration, GORpipe will raise an exception if -maxseg is not provided. Since we know that the maximum span of exons in the human genome is smaller than 100 kb, we can safely use that number to obtain better search efficiency. For these query examples, we will hardly notice the difference in performance, however, for non-streaming queries which involve for instance join into dense reads in BAM files, it will be much more efficient to set -maxseg to a number like 1000 bp.

3.3 Nested queries

One of the many powerful features of the GORpipe syntax is the support for nested query expressions. While the syntax notation reflects process substitution in Unix commands, functionally they have more in common with nested sub-queries in SQL, because nested GORpipe query expressions are *seekable*. Nested queries greatly increase the expressivity of the syntax and can often prevent the creation of many unnecessary temporary files, thereby help to minimize overall IO cost.

Here we will give an example of how we can find GWAS results, which overlap either with exons or with transcription factor binding sites with sufficient number of cell counts. We can express this is several ways, the first one being a merge of two join queries:

Example 16. Merging a stream from a nested-query

```
gor snp_gwas_results.gorz
| join -i -snpseg -maxseg 100000 exons.gor
| merge < (gor snp_gwas_results.gorz
        |join -i -snpseg -maxseg 30000 encode_tfbs.gorz
        | where cell_count > 5)
| distinct
```

First, note the -i option in the JOIN command, used to denote intersect join, i.e. each row in the left-source which overlaps the right-source is returned, maximally once and without the columns from the right-source. Second, the MERGE command takes the output of the nested query and merges it with the left-stream in a GOR fashion. Finally, since a SNP can potentially overlap both exon and transcription factor, we apply the DISTINCT command. Unlike 'select distinct *' in SQL, it is *stream aware* and very efficient because it takes advantage of the genomic order of the output from the MERGE command.

Alternatively, rather than merging two joins, we can join into a nested query containing a merge:

Example 17. Joining with a nested merge query

```
gor snp_gwas_results.gorz | join -i -snpseg -maxseg 100000
< (gor exons.gor | merge < (gor encode_tfbs.gorz
  | where cell_count > 5))
```

While the last two queries will most likely only involve streaming, this query formulation will also work well with highly selective filtering on the left-stream, e.g.:

Example 18. Joining a sparse stream with a nested merge query

```
gor snp_gwas_results.gorz | where pval <=1.0e-50
| join -i -snpseg -maxseg 100000
    <(gor exons.gor | merge <(gor encode_tfbs.gorz
        | where cell_count >5))
```

In the above example, the query will most likely only pass very few rows from the left-source and cause seek to be applied on both the exon and transcription factor files.

For yet another example, consider the case where we want to list all compound heterozygous variants of an individual in a particular gene. This requires locating all variants overlapping with the exons of the gene and to formulate a Cartesian join between all of them and filter the combinations based on allele and phase (e.g. imputed paternal/maternal or phase from sequence reads). We could write it in the following manner:

Example 19. Finding compound heterozygous genotypes in BRCA2

```
gor genes.gor | where gene_symbol ='BRCA2'
| rename gene_symbol gene1
|join -segvar -rprefix var1 <(gor variations.gort -f A
|join -varseg -maxseg 100000 -rprefix exon exons.gor)
| where gene1 =var1_exon_gene_symbol
| join -snpsnp <(gor genes.gor | rename gene_symbol gene2
  | join -segvar -rprefix var2 <(gor variations.gort -f A
  | join -varseg -maxseg 100000 -rprefix exon exons.gor)
  | where gene2 =var2_exon_gene_symbol)
| where gene1 =gene2 and
(var1_phase!=var2_phase or
var1_Zygosity ='Hom' and var1_pos =var2_pos
and var1_ref =var2_ref and var1_alt =var2_alt)
| select chrom,var1_pos,var1_Ref,var1_Alt,
         var1_Zygosity,gene1,var2_pos-var2_Zygosity
| sort 3000000
```

Note that we are essentially joining two queries on the start positions of genes. Each of these two queries involve joins between genes and variants; variants which are joined with exons to eliminate intronic variants. Another thing worth pointing out for those who are familiar with SQL is that the GORpipe syntax does not support correlated sub queries. Therefore, the predicate `gene1=var1_exon_gene_symbol` has to be placed after the variation-exon join.

Although the above query expression may at first look somewhat complicated, on a regular laptop computer it runs in less than a second and if the gene symbol predicate is changed to a wildcard, `gene_symbol 'BR*'`, it still only takes around a second to evaluate for the three genes BRCA1, BRCA2 and BRCC3.

3.4 Materialized queries, views and alias definitions

The GORpipe system has a `DEF` command to define simple C-like preprocessing macros and aliases and a `CREATE` command that allows query expressions to be materialized into intermediate tables. These can be useful to build queries or *make queries* in stages and can reduce computation time if they are often referred to, especially when computation to disk-space ratio is high.

Consider for instance the case where we are only interested in GWAS results of high significance and for a subset of phenotypes, e.g. related to cancer. We can inspect if SNPs associated with cancer are closely spaced in the genome like the following:

Example 20. Using `CREATE` to define materialized views

```
create ##GWASsubset## =gor snp_gwas_results.gorz
| where pval <1.0e-4 and contains(phenotype,'cancer');

gor [##gwassubset##] | where pval <1.0e-10 | prefix #2- A
| join -snpsnps -f 100000 -rprefix B [##gwassubset##]
| where not (A_phenotype =B_phenotype and A_pos =B_pos)
| rank 1 B_pval -o asc
| where rank_B_pval <=10
```

In other words, we form a join between all strongly significant cancer associations and other weakly significant associations within a 100kb distance. For each locus from the left-source, we rank the associations from the right-source and return only the top-ten most interesting associations. If we change for instance the allowed overlap distance, the virtual relation `[##GWASsubset##]` will not have to be re-executed, unless the source file `snp_gwas_results.gorz` is modified or the create definition changed.

Often scientists are more interested in SNPs that are in linkage disequilibrium rather than defined by mere base pair distance. Thus, they would like to perform a 'LD-join' rather than the standard GOR genomic self-join as shown in previous example. For instance at deCODE, we calculated the LD between every marker in a 10Mbase window and stored this in one very large relation (approx. 50 billion rows) of the form (chrom, pos1, maker1, pos2, marker2, rsquare). If we assume that this relation is represented with the alias `#LD#`, an LD version of previous example can be written as:

Example 21. Using a DEF command to define a macro for LD-join

```
def LDjoin(1,2) =prefix #2- A | join -snpsnp #LD#
| where A_mrkName =marker1
| where abs(pos2-A_pos) <$2/2 | select #1,pos2,2-
| sort $2
| join -snpsnp -rprefix B $1 | where B_mrkName =marker2
| calc bpDist B_pos-A_pos | hide pos2x-marker2
| select #1,A_*,B_*
| sort $2;

create ##GWASsubset## =gor snp_gwas_results.gorz
| where pval <1.0e-4 and contains(phenotype,'cancer');

gor [##gwassubset##] | where pval <1.0e-10
| LDjoin([##gwassubset##],100000) |
| where not (A_phenotype =B_phenotype and A_pos =B_pos)
| rank 1 B_pval -o asc
| where rank_B_pval <=10
```

The above `LDjoin` definition, which takes two parameters, can be considered as a parameterized SQL view definition. Notice how the `LDjoin` uses the `SELECT` command to move `pos2` column into the 'GOR position column', because the `JOIN` command only performs spatial joins on the first columns. Since this operation cannot guarantee genomic order, where there are multiple rows in the left-stream, we have to apply the `SORT` command. The sort window takes into account the maximum deviation from genomic order. This deviation is governed by the `#LD#` relation and the filtering '`abs(pos2-A_pos)`'. Again, the fact that the `SORT` command has a very efficient sliding window implementation makes it possible to perform the sort without saving data into temporary files and with minimum memory usage.

So far, we have provided many examples which highlight the basic concept of the GOR architecture. Due to space limitations, we defer

further examples to the supplementary material provided with this paper. There we cover commands which can be applied to sequence reads and variant analysis and show examples of queries which can be used as a step towards creating a very fast variant effect predictor. We also explain a related NOR syntax for non-genomic ordered relations and show how a hybrid syntax of GOR and NOR can be used to setup case-control analysis on sparse variation datasets that take into consideration missing data, due to low sequence read coverage.

4 Discussion

In this paper, we have presented a software system developed around genomic ordered relations and a novel declarative query language. The syntax combines in a novel manner features from both Unix shell commands and SQL. As with most new programming and query languages, it is not so much a question of enabling something that wasn't possible before, but rather, to bring forth an elegant language which allows a good data abstraction, promotes and facilitates efficient query patterns. We do this by combining relational data abstraction patterns from the RDBMS world with pipe shell syntax to implement relational operators. As we show, this flavour of query syntax is not limited to ordered genomic data and is also very convenient for time ordered data or generic relational data. Indeed, since each command is isolated syntactically, we find this type of language structure in many ways easier to extend and work with than languages such as SQL.

Here, we have demonstrated the versatile nature of our syntax to express complex queries. The GORpipe system is also easily integrated into Bash pipe commands or Python scripts. Thus, if the expressive power of our declarative GORpipe syntax is exhausted, one can mix it with imperative programming languages. While the relational concepts are not new, our implementation is very different from existing RDBMS. It is based on a genomic ordered streaming architecture which is better suited to deal with the high volume of sequence data which many sites are just starting to grapple with. Just like SQL, our GORpipe command set goes beyond operations that can be mapped to conventional relational algebra and many of the GORpipe commands have similarity with the more recently introduced SQL windowing functions. Unlike in SQL, for simplicity and performance reasons, GORpipe only supports a single genomic ordering scheme.

We do indeed find that our system performs quite well, although we have yet not undertaken a formal comparison study. For an example, in the variant effect prediction example, which we provide in Section 1.3 in the supplementary material, we see performance which is ten times faster than the comparison. Also, using version 2.10.1 of BEDTOOLS, an intersection of a `dbsnp.bed` table against itself with the '-sorted' option, returns over 118 million rows in 3:36 while the same type of GORpipe query performing a `-snpsnp` join on equivalent GORZ file returns the same number of rows in 3:25.

An informal performance comparison between the file formats GORZ and Tabix files, in Section 1.5 in the supplementary material, shows that GORZ provides almost two times better streaming efficiency for GORpipe than the latest Java implementation of Tabix and equivalent seek performance. Although our simple approach of using a binary-like search, as compared to using something like an R-tree structure, may not be as well suited for data files where there are features with very different segment sizes, we argue that features of unlike nature, e.g. due to very different size categories, are indeed better split up into separate tables where appropriate -maxseg options can be applied, since this allows for better streaming utilization in range queries. This approach can indeed also be applied on 'the physical file level', regardless of whether the data is stored in a single file or a collection of files.

As to be expected, some of the language features in GORpipe have been introduced elsewhere in similar form, although not in as coherent manner. Recently, it has been brought to our attention, by one of the reviewers, that support for named columns has for instance been introduced in bioawk, an extension of AWK. Motivated by the GROUP BY clause in SQL, we introduced the `-gc` option in our aggregate commands as well as some other commands where we think grouping is relevant. It turns out that similar grouping option can be found in the command line utility `datamash`. The fact that GORpipe supports a 'fork write' to simultaneously output data to multiple files (partitions) and many commands which support grouping (`-gc`) and equi-join options (`-xl`, `-xr`), makes it easy to express queries that process multiple samples and share the overhead of accessing non-sample reference data for annotations or filtering.

The declarative nature of our syntax means that there is room for many additional optimizations behind the scenes. This may involve changing the order of commands execution and predicate push-down, combining command steps, use of adaptive buffer sizes, fine tuning of the memory model and rely on more powerful iterator abstraction in our implementation. Also, the use of more metadata and more sophisticated query optimization may eliminate the need for the `-maxseg` option in the genome spatial joins.

On the near term roadmap are features to allow users to create new commands and functions, leveraging dynamic class loading, based on any JVM language such as Scala, Jython and Renjin R. We also foresee an integration with the Hadoop stack, including the use of HDFS, Parquet columnar storage and drivers relying on SparkSQL (Massie *et al.*, 2013). Finally, we are working on an extension of the GORpipe system with a full blown elastic cloud based database engine with support for data partition and distributed computing.

Conflict of Interest: none declared.

References

Cingolani,P. *et al.* (2012a) Using drosophila melanogaster as a model for genotoxic chemical mutational studies with a new program, SNPsift. *Front. Genet.*, **3**, 35, doi: 10.3389/fgene.2012.00035.

Cingolani,P. *et al.* (2012b) A program for annotating and predicting the effects of single nucleotide polymorphisms, SNPeff: SNPS in the genome of *Drosophila melanogaster* strain w1118; iso-2; iso-3. *Fly*, **6**, 80–92.

Danecek,P. *et al.* (2011) The variant call format and vcftools. *Bioinformatics*, **27**, 2156–2158.

Egilsson,Á.S. and Gudbjartsson,H. (2003) Indexing of tables referencing complex structures. *CoRR*, cs.DB/0309011.

Enderle,J. *et al.* (2004). Joining interval data in relational databases. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04*. ACM, New York, NY, USA, pp. 683–694.

Giardine,B. *et al.* (2005) Galaxy: a platform for interactive large-scale genome analysis. *Genome Res.*, **15**, 1451–1455.

Gudbjartsson,D.F. *et al.* (2015) Large-scale whole-genome sequencing of the Icelandic population. *Nat. Genet.*, **47**, 435–444.

Kent,W.J. *et al.* (2002) The human genome browser at UCSC. *Genome Res.*, **12**, 996–1006.

Kozanitis,C. *et al.* (2014) Using genome query language to uncover genetic variation. *Bioinformatics*, **30**, 1–8.

Lakshmanan,L.V.S. *et al.* (1996) Schemasql – a language for interoperability in relational multi-database systems. In: *Proceedings of the 22th*

International Conference on Very Large Data Bases, VLDB '96. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 239–250.

Li,H. (2011) Tabix: fast retrieval of sequence features from generic tab-delimited files. *Bioinformatics*, **27**, 718–719.

Li,H. *et al*. (2009) The sequence alignment/map format and samtools. *Bioinformatics*, **25**, 2078–2079.

Massie,M. *et al*. (2013) *Adam: Genomics Formats and Processing Patterns for Cloud Scale Computing*. Technical Report UCB/EECS-2013-207, EECS Department, University of California, Berkeley.

McKenna,A. *et al*. (2010) The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome Res*., **20**, 1297–1303.

Quinlan,A.R. and Hall,I.M. (2010) Bedtools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, **26**, 841–842.

Wang,K. *et al*. (2010) Annovar: functional annotation of genetic variants from high-throughput sequencing data. *Nucleic Acids Res*., **38**, e164.