Querying the genome

Christos Kozanitis

June 13, 2012

Abstract

1 Introduction

2 Software Layering for Genomics

3 Software layers and interfaces for genomics (from cacm)

Our vision is inspired by analogy with systems and networks, where software layering has solved similar problems. For example, the Internet has successfully dealt with a wide variety of new link technologies (from dialup to wireless) and applications (from email to social networks) via the "hourglass" model using the key abstractions of TCP and IP (Figure 1a).

In the same way, we propose that Genomic Processing software be layered into an instrument layer, a compression layer, an evidence layer, an inference layer, and a variation layer that can insulate genomic applications from sequencing technology. Note that the only way to achieve such modularity is to forgo some possible efficiencies that could be gained by leaking information across layers. For example, biological inferences can be sharpened by considering which sequencing technology is being used (e.g., Illumina versus Life Technologies) but we suggest that modularity is paramount.

Some of the initial interfaces are already in vogue with standard formats for exchange. Many instruments now produce sequence data as 'fastq' format; The output of mapping reads is often represented as SAM/BAM format [5], although other compressed formats are coming into vogue [4]. At the highest level, there exists standard such as VCF [6] to describe variants in a standardized way. The existing formats are shown bolded in Figure 1a)

Here, we propose additional layering between the mapped tools and applications. Specifically, we separate the collection of *evidence* required to support a query (deterministic, large data movement, standardized) from the *inference* (probabilistic, comparatively smaller data movement, little agreement on techniques).

We show that the Evidence Layer (EL) can be implemented on a compute cloud while the more volatile Inference Layer can be implemented on the desktop. We assert that while Inference methods vary considerably, the Evidence for inferences is fairly standard and hence propose a Genome Query Language (GQL) that permits efficient implementation, and presents a flexible mechanism for gathering evidence for Inference.

Note that while we focus on the Evidence-Inference layer in this paper, a careful specification of a variation layer (Figure 1a)) is also important. While the data format of a variation is standardized using, for example, VCF [6], the interface functions are not. An application like personalized medicine or discovery will query the variation layer and join the variation information with phenotype information gleaned from medical records (EMRs).

3.1 The case for an evidence layer

Consider a complex biological query: " identify all deletions that disrupt genes in a certain biological network, and the frequency of those deletions in a natural population". The bioinformatician struggles to transfer this

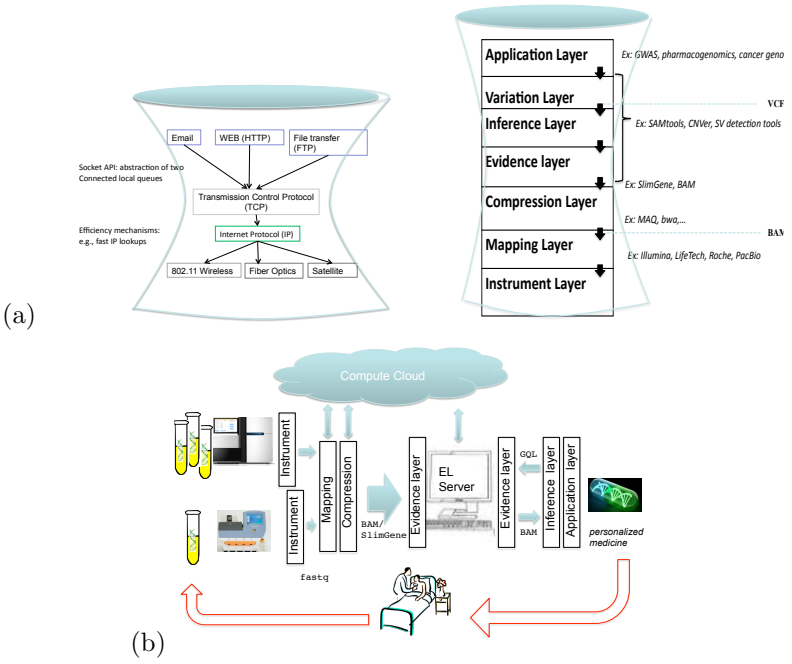

Figure 1: Abstraction for genomics.

question into a problem of statistical inference with bounds on false-positive and false-negative errors. However, the first part of any such query would be the gathering of the evidence. Here, the evidence would consist of all reads, and their mappings that satisfy certain properties. For example, the reads must overlap regions encoding genes in the given biological network, and must fall in one of three categories: (a) Concordantly mapping reads as these reads suggest heterozygosity of the deletion; (b) discordant reads: all reads where the read and its pair map to locations much further apart than could be explained by natural variation in insert length, indicating a deletion in the donor; (c) reads in which only one of the two ends are mapped, and which are proximal to discordant reads, corresponding to reads that map at the boundary of the deletion breakpoint. The development of an EL to support such queries provides several advantages

- The separation allows Inference Layer designers to start thinking of alternate forms of evidence to improve the confidence of their queries. For example, we might consider looking for "partially mapped" READs. If one pair of READs overlaps the boundary of a deletion, then that READ will match a prefix or suffix of the reference and the other may match perfectly. If the mapping shows that one maps correctly and the second does not, it provides some evidence as to the location of the boundary; further a small amount of effort can be used to decide if the non-mapped pair has a partial match with the reference.
- The EL often poses a 'data bottleneck' as it involves sifting through large sets of genomic reads. By contrast, the inference layer may be compute intensive, but typically works on smaller amounts of data (filtered by the evidence layer). We can implement EL on the cloud while the Inference Layer can be implemented either on the cloud or on client workstations. The evidence can easily be transported across the network interactively (Mbytes versus Gbytes). We have already seen moves by commercial cloud operators like Amazon to host the 1000 genome data sets on their infrastructure. The cloud allows rented computation on demand without the cost of maintaining a permanent cluster by every genomics researcher.
- The standardization of the EL will allow vendors time to creating a fast and scalable EL implementation. It is hard to do with the Inference Layer today as it is a moving target.

In the following, we develop this intuitive idea further by describing a Genome Query Language to support the Evidence Layer.

4 GQA: a relational algebra for genome queries(from cacm)

We show that we can express most genomic queries in a relational algebra (the *Genome Query Algebra-GQA*, as also an SQL-like language (the *Genome Query Language-GQL*). While our syntax is SQL-like, we need some special operators and relations that may not fit with available Relational Database Management Systems (RDBMS). In parallel, we are developing a database system to archive genomes and support genomic queries.

Our database has some key relations. The first is *R (Reads)* that describe the set of read fragments output by the sequencer, and all of their properties. For example, for paired-end sequencing, R could contain identifiers for each read r, and its paired-end. A second relation encodes a set of intervals on the genome, where each is specified by the triple "⟨chromosome, start-position, end-position⟩". A specific, and important, case of the interval relation is the relation G (the genome) where each entry is a point interval corresponding to a specific location (*locus*) on the genome. As G is large ($3 \cdot 10^9$ loci), we do not materialize G explicitly, using it instead as a virtual (conceptual) relation to facilitate the relational expression of queries. A final relation P describes the set of all individuals in a population, and will be discussed in a subsequent section.

As shown below, the key benefit of our proposed relational modeling is to enable manipulation via algebraic operators (the standard relational algebra operators, as well as domain-specific operators defined shortly). This in turn creates the opportunity to leverage proven techniques developed by the data management community for efficiently querying large collections of data. Regardless of the fact that some of these techniques need to be adapted or extended to our specific domain, while others can be applied directly, the overarching contribution here is to unlock the potential of large-scale data processing in the context of a huge data scale challenge.

Standard relational algebra operators and language constructs Standard algebraic operators we use in the examples below include:

- The projection operator Π where $\Pi_X(E)$ returns, from the relation E evaluates to, the restriction of the tuples to the attributes named in X. In GQL, we express this operation using

  SELECT X FROM E

- The selection operator σ, where $\sigma_\theta(E)$ selects the subset of precisely those tuples in the result of expression E that satisfy filter predicate θ. Correspondingly, in GQL:

  SELECT * FROM E WHERE θ

- The Cartesian product operator $\times$, where $E \times F$ refers to all tuples of the form (e, f) where $e \in E, f \in F$. In GQL:

  SELECT * FROM E,F

- The renaming operator $\rho_N(E)$, which sets to N the name of the relation returned by E.
- The group-by operator Γ, where in

  $$_{G_1,G_2,\ldots,G_k}\Gamma_{N_1:F_1(A_1),\ldots,N_l:F_l(A_l)}(E)$$

  , E is an algebraic expression that evaluates to a relation, $G_1, \ldots, G_k$ are group-by attributes, each F_i is an aggregate function and each N_i, A_i are attribute names. The meaning of the group-by operator is the standard one: all tuples in the result of E are partitioned into groups, such that two tuples are in the same group if and only if they agree on the values of all group-by attributes. Thus, the groups can be identified by the value of the group-by attributes. For each group $(g_1, \ldots, g_k)$, the result contains a tuple of attributes $G_1, \ldots, G_k, N_1, \ldots, N_l$ and values

  $$(g_1, \ldots, g_k, n_1, \ldots, n_l)$$

  , respectively. Here, each n_i is computed by applying function F_i to the collection of attribute A_i's values found in the group. In GQL:

  SELECT $G_1, \ldots, G_k, N_1 = F_1(A_1), \ldots, N_l = F_l(A_l)$
  FROM E
  GROUPBY $G_1, \ldots G_k$

By default, this collection is viewed as a multiset, i.e. duplicate values are not removed. To apply F_i after eliminating duplicates, we use the syntax

$$N_i : F_i(\text{distinct } A_i)$$

. The syntax of the count aggregation is an exception, in that no A_i attribute need be specified.

The Map join and the Project Interval operator As reads are sequenced, they are also mapped to the reference genome using various mapping algorithms. Some mapping algorithms map each read to a unique location, or not at all; others may choose to identify all intervals where the underlying genomic string is similar (in terms of edit distance) to the read. For now, assume that each read is mapped to its best location by a mapping algorithm M, and the relation R includes the interval of the best map.

For an arbitrary relation of genomic intervals, A, define a *map-join* operation $R \boxtimes A$ by a pair of tuples $(r, a) \in R \times A$ such that (r.chr, r.start, r.end) and (a.chr, a.start, a.end) 'intersect'. Recall that we defined the relation G as the set of all genomic loci. Therefore, $R \boxtimes G$, denotes the set of all tuples r, g where g denotes a genomic coordinate that r maps to. In GQL:

$$\text{SELECT } * \text{ FROM MAPJOIN } R, G$$

In many practical instances, the relation is quite large. If 33% of the genome was mapped on the average by 10 reads, $R \boxtimes G$ would contain $\simeq \frac{1}{3} 3 \cdot 10^9 \cdot 10 = 10^{10}$ tuples. To reduce output size, we define a special *Project-interval* operator Π^i. $\Pi^i_{A.ID}(R \boxtimes A)$ outputs all genomic intervals after 'fusing' adjacent and overlapping intervals. Thus,

$$\Pi^i_{G.ID}(R \boxtimes G)$$

would result in the set of all disjoint genomic intervals with some read mapping to them. Correspondingly, in GQL:

$$\text{SELECT MERGED G.ID FROM MAPJOIN } R, G$$

We show, mostly by example, that a large corpus of biological queries can be supported by these abstractions.

5 Implementation

This section introduces our indexing scheme and demonstrates its capabilities by describing the implementation of a series of important abstractions. Section 5.3 describes our indexing, Section **??** describes the range retrieval query, Section **??** describes the coverage query, Section **??** describes the query that finds areas that contain multiple discordant clones and Section **??** describes creation of a haplotyping graph from a set of SNP locations.

5.1 Arrangement of the Input Data

The input to our system consists of two main categories of data. Files of Reads and tabular text files which represent intervals.

Read files are typically large binary files that follow the BAM specification [5] and for each read they contain all properties of the relation R that we describe earlier. They are sorted according to their alignment location and we require that all alignment location are *chromosome-isolated* that is that each file contains reads that map to a single chromosome.

All other files are ASCII tabular files 4 fields of which contain an *entryID*, a chromosome, a starting position and an ending position.

5.2 Connection between Mate Pairs

Two reads belong to the same pair when their *QNAME* fields are identical. In the SAM/BAM specification *QNAME* is a string of characters whose usual length is 15 to 20 characters and its semantics carry details about the experiment and the topology of the read during the sequencing process.

The conventional way of locating all mate pairs of a read file, which is the re-sorting of the file *QNAME* field, is both inefficient and non functional. The sorting of a read file takes prohibitive amount of time because of the size of the data. Remember that a typical BAM file that is produced by Illumina Sequence Analyzer and contains all reads that map with chromosome 1 contains $90M$ reads and its size is 5-6GB, while the size of the entire dataset is around 80GB. Moreover, end applications prefer reads to be sorted by mapping location because they use indexes for quick range retrievals. Thus, data need to be re-sorted again for those applications.

Our approach bypasses the sorting requirement with the hashing of the *QNAME* fields. For each read, the *QNAME* field is added to a hash table if it is absent from it. When a search of a *QNAME* field matches exactly with another record, we keep as mate pair metadata the pointers of the reads that are involved in the match.

5.3 Indexing reads by their ranks

Figure 2 shows the array that we use to index a file of reads. The ranking of the reads addresses the entries of the array. For example, the first entry of the table keeps the information of the first read and so on. Each entry contains all the meta-data that enable quick in-memory operations and also pointers for random accessing to the read file. Thus, the index includes the location the strand and the length of a read, its byte offset within the file and a link to its mate entry which can be found with the pre-processing step of Section 5.2

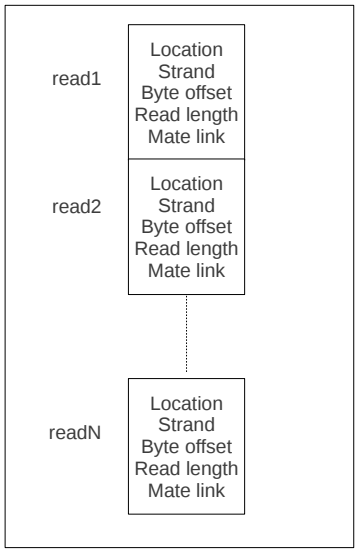

Figure 2: Our indexing scheme

We claim that our index is small enough to fit in the main memory of any computer. Since we use 8 bytes for the byte offset, 4 bytes for the mapping location and the mate link, 10 bits for the read length and 1 bit for the strand information, the size of the index of the set of alignments of the largest chromosome (chr1) of coverage $35\times$ and read length of 100 is $1.5GB$.

The building of the index occurrs in a small time frame. In our experiments, a dataset of approximately $90M$ reads from NA18507 that map with chr1 requires 6 minutes to create the index, while the memory footprint of the execution does not exceed $2GB$

5.4 Read Selection

The size of the index makes the implementation of read selection straightforward. Since the index of an entire chromosome fits into the main memory, the selection operation iterates over the index and keeps the pointers of

the reads that satisfy a condition.

5.5 Read Projection

TODO: Do we have anything to say here??? we usually type select * as we always create bam files in the output.

5.6 Selection on Interval

TODO: Need to make clear that we are talking about coverage query here

A conventional way of implementing the module that looks for the maximum interval under a read coverage constraint is an array of counters for each chromosome coordinate which keep the number of intervals that include the respective location. The example of Figure 3 shows the contents of the counters in an imaginary scenario of a chromosome of length 10 and reads of length 2. The value of each counter is the number of reads that overlap with its location.

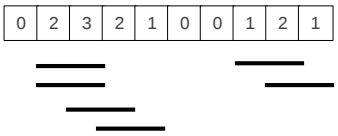

Figure 3: The auxiliary vector of counters of a selection on intervals.

However the linear update of the array of the counters is a computationally expensive task, because each interval updates as many counters as its range. Thus, for a chromosome of length L, a set of n intervals each of which has a worst case range L, the number of times that counters need to be updated is $O(n \times L^2)$, where L in the human genome can be up to $250M$

A more efficient implementation of this module is to model it after the balanced parenthesis problem. A left parenthesis is assigned for every opening of an interval and a closing parenthesis is assigned for every closing of an interval. Thus the area that is covered by at least k intervals is the area where k or more parenthesis are open concurrently. The solution requires a counter which increases each time a parenthesis is open and decreases when a parenthesis closes. The regions of interest are the areas for which the values of the counter are at least k. Now the number of the counter updates that are required by this algorithm is equal to the number of of the input set of intervals n.

As an example consider the intervals of Figure 4. Figure 5 shows the modeling of the same intervals with the use of parenthesis. Note that a parenthesis opens at every position which is the starting point of an interval and it closes at those positions that are the closing ends of intervals. The values of the counter appear in the bottom of this figure and for $k = 3$ the area of interest includes all positions whose value is greater than 2.

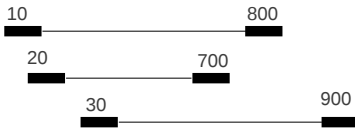

Figure 4: A set of intervals.

5.7 Mapjoin

Following the lead of [2], the implementation of the *Mapjoin* between two tables requires the use of interval trees. We create an interval tree on the table with the smallest set of intervals and we query it with each interval of the

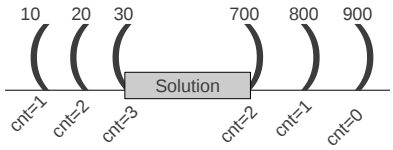

Figure 5: The intervals are modeled as parenthesis. The shaded area is the region which is covered by at least 3 intervals.

largest one. From the matching intervals we generate tuples of pointers, one from each table, that represent the join of the fields that intersect.

6 Visualization

7 Results

The database of our experiments consists of genome NA18507 [3] which is the output of the sequencing of a male person from Yoruba in Ibadan, Nigeria that has been conducted in the context of the 1000-Genomes project [1] using Illumina's latest mate pair based sequencing technology. The data consist of 1176090451 reads of length 100 which is a yield of a genome coverage of 40× and the size of the BAM file that stores the reads is 73GB. The alignment of the reads has been done against version hg18 of the human reference [**?**] using Eland, Illumina's propietary alignment tool. 1104952383 of the reads have been mapped. 1033814315 of the mapped reads have their mate also mapped and the rest 71138068 are singletons. 5832322 of the mapped reads have their mate mapped to a different chromosome.

7.1 A few challenging queries

Get the evidence of all isolated deleted regions The evidence of the deleted regions comes from the observation that the distance between the alignments of the reads of a pair is unjustifiably large. To eliminate outlying information we require that an area to be deleted needs to be supported by at least 3 different clones. Figure 1 shows the gql code of the query which needs 369seconds to run and returns 226011 reads.

Algorithm 1 The gql code of the deletion query.

```
import READS;

H1=select create_intervals()
from READS using intervals (location, mate_loc, both_mates)
where location>=0 and
mate_loc>=0 and
(
(mate_loc-location>1000 and mate_loc-location<100000) or (location-mate_loc>1000
    and location-mate_loc<100000)
)

select merge_intervals(interval_coverage>3) from H1
```

Get the evidence of all inverted regions The evidence of the inverted regions comes from the observation that the orientation of the involved clones is inverted. In Illumina sequencing, normal clones are expected to such that the "leftmost" end of a pair maps with the forward strand while the "rightmost" end maps with the reverse

strand. Thus the query that is shown in Figure 2 looks for regions that are covered by at least 5 different pairs of reads where the leftmost end of each pair maps with the reverse strand and the rightmost end maps with the forward strand. The query needs 476 seconds to run and returns 202632 reads.

Algorithm 2 The gql code of the inversion query.

```
import READS;

H1=select create_intervals()
from READS using intervals (location,mate_loc, both_mates)
where location>=0 and
mate_loc>=0 and
(
( mate_loc>location and strand=='-' and mate_strand=='+') or
( location>mate_loc and strand=='+' and mate_strand=='-')
)


select merge_intervals(interval_coverage>5) from H1
```

Find all regions with high copy number Regions that are covered by unrealistically large numbers of reads are called high copy number regions. Normally, the number of reads that are expected to be found in a region is equal to the coverage of the experiment, which in our datasets is 40. Thus the query that is shown in Algorithm 3 looks for regions that are covered by at least 200 reads. The query needs 2141 seconds to run and the evidence data of the output consist of 10979437 reads.

Algorithm 3 The gql code of the high copy number query.

```
import READS;

H1=select create_intervals()
from READS
where location>=0

select merge_intervals(interval_coverage>200) from H1
```

Get all reads and their mates that overlap with some gene The query uses an external table which consists of all known genes according to the RefSeq database. The output of this query is the mapjoin of all mapped reads with the coordinates of the UCSC database. Figure 4 shows the gql syntax of the query which needs 17311seconds to run and returns 495454218 reads. Note that the large execution time of the query is bounded by the IO. To verify that, we utilized samtools, which provide an extremely efficient indexing scheme for range retrieval, to fetch the same ranges of reads. It took samtools 19875seconds to fetch the same ranges of data which is even slower.

Algorithm 4 The gql code that retrieves all reads and their mates that overlap with some gene.

```
import READS;
import genes;

H2=select * from READS where location>0

H3=select *
from MAPJOIN genes using intervals(begin, end),H2 using intervals (location,
    location+length, both_mates)

select * from H3
where location>0
```

7.2 Integration with known software

8 Acknowledgments

References

[1] 1000 Genomes Project Consortium, R. Durbin, G. Abecasis, D. Altshuler, A. Auton, L. Brooks, R. Durbin, R. Gibbs, M. Hurles, and G. McVean. A map of human genome variation from population-scale sequencing. *Nature*, 467:1061–1073, Oct 2010.

[2] A. V. Alekseyenko and C. J. Lee. Nested containment list (nclist): a new algorithm for accelerating interval query of genome alignment and interval databases. *Bioinformatics*, 23(11):1386–1393, 2007.

[3] D. Altshuler, L. Brooks, A. Chakravarti, F. Collins, M. Daly, P. Donnelly, and I. H. Consortium. A haplotype map of the human genome. *Nature*, 437(7063):1299–1320, Oct 2005.

[4] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese. Compressing genomic sequence fragments using SlimGene. *J. Comput. Biol.*, 18:401–413, Mar 2011.

[5] The SAM/BAM format. http://samtools.sourceforge.net/SAM1.pdf.

[6] Variant call format. http://vcftools.sourceforge.net/specs.html.

9 Appendix

1. The input is a collection of intervals. Find all reads that map to these intervals.
2. Find intervals where (a) coverage is at least k, (b) coverage is at most k. The goal is to find regions with high or low copy numbers.
3. Find intervals where the coverage of $(-,+)$ reads is at least k. The motivation is to find regions of tandem duplication.
4. Given an interval I, find (a) all discordant reads that intersect with I (b) all discordant reads whose mate-pair intersects with I.
5. Given a collection of reads R, find all intervals containing at least k of the reads in R.
6. Given a collection of (point) intervals, find all reads that intersect with at least two of these intervals.
7.