Interactive Genomics: Rapidly Querying Genomes in the Cloud

Christos Kozanitis(UCSD), Vineet Bafna(UCSD), Ravi Pandya(MSR), George Varghese(MSR)

Abstract

Genome sequence data is now "Big Data" in both volume and velocity. Joined with medical records, genome data can be mined for insights for treating disease. Genomics today is dominated by *batch processing*: simple analytical questions take days to answer. We propose instead that genomics be made *interactive* so that queries on a large genome database in the cloud are answered across the network in seconds. Towards this vision, we introduce a query language, Genome Query Language (GQL), in which intervals are first class, and joins are based on intersection not equality. GQL can be used to query for large structural variations on the TCGA cancer archive using only 5-10 lines of high level code that takes around 60 seconds to execute in the Azure cloud on an input BAM file of 83 GB. GQL results can be *incrementally deployed* both on the UCSC browser and by refactoring an existing variant caller to provide 6x speedup. Our paper focuses on the system design and five key optimizations — cached parsing, lazy joins, materialized views and chromosomal parallelism — that speed up query processing by 100x. We also reflect on 3 years of experience designing and using GQL.

1 Introduction

We argue that genomics is being commoditized and outline a vision for interactive genomics. Each cell in the body is controlled by "programmed instructions" written into its total DNA (genome). The program text is a string of around 3 billion characters drawn from an alphabet of $\{A, C, G, T\}$ modularized into functional routines called *genes*. Genes contain the instructions for making proteins, which are the functional machinery in the cell. Thus, any *mutation/variation* in the instructions can cause the machinery to be faulty, causing disease. These mutations may be inherited (and therefore, present in all cells), or somatic, acquired during the lifetime of the individual. Pernicious somatic mutations might cause a cell to grow uncontrolled and become a tumor cell. Such somatic mutations can be discovered by comparing the genomes of normal and cancer cells. The term *personalized medicine* is often applied to the context in which the treatment is specifically tailored to individual genomic variation. Personalized medicine and large-scale discovery of variation are likely in a few years because of the following trends:

Falling costs: Sequencing costs fell from \$100M in 2001 to around \$3,000 in 2012, faster than Moore's Law. Costs should fall to below \$1000 in a few years[16].[1] *Data Velocity:* 30,000 full sequence genomes were produced in 2011 versus 2700 in 2010. The Beijing Genomics Institute [11] has 128 sequencers that can produce 40,000 sequences per year in China alone. *Electronic Medical Records: Phenotype* information such as the patient's age, sex, disease codes, diet and and drug treatments are traditionally stored in a patient medical record. The HITECH act [10] mandates that medical records become electronically readable EMRs by 2014. *Cancer Genomics:* Cancer treatments have not improved significantly in 10 years, incentivizing patients and physicians to try new treatments [18]. Drugs such as Herceptin and Gleevec have targeted genomic pathways and have had great success for a few cancers.

In summary, reduced costs will lead to universal sequencing. Millions of genomes and associated medical records will then be mined to provide actionable insights for disease treatment, especially for cancer.

Genomic Batch Processing: The standard workflow today for discovering genomic variations in say cancer patients is:

1. Assemble: The researcher assembles a small (< 100) cohort of patients (cases) as well as normals (controls) over a few months because of the medical protocols involved. *2. Sequence:* Tissue sample from each patient/normal is used by a sequencing machine to produce a sequence in roughly 1 day. *3. Analyze:* An *ad hoc* program is written to test for hypotheses such as "Gene X is deleted in this cancer sub-type compared to normals". Frameworks [27, 25] reduce programming effort, but much hand-crafted code is still required; it may take months and several iterations as the hypothesis is refined.

Further, *sharing* between researchers is rare. Each full sequence genome is around 100 GB and (optimistically) takes 1 day to download. Genomes are often shipped by Federal Express/UPS. Some NIH grants require that genomes be made public in repositories such as the 1000 Genome Project [12] and the TCGA Cancer Archive [23]. However, download times and the use

[1] The costs refer to a $30\times$ coverage of a single genome. Our paper studies full sequencing (reading the entire DNA) as opposed to sampling selected regions by a more limited technology known as microarrays used by say 23andme.

of multiple ad hoc analysis programs by individual researchers makes it hard to reuse (or even validate) findings across studies. One study was able to fully reproduce only 2 of 18 results and partially 6 more [21]. Haussler's OSDI keynote [18] sounded similar themes: genomic big data, cancer as a driving application, and the need for new tools.

1.1 Interactive Genomics

Consider the scenario (Figure 1) where all genomes are on a cloud database. Each genome G_i is annotated with a set of disease codes as well as other clinical metadata (e.g., sex, weight, drug regimen).

1. Genome Browsing: Alice, a biologist, is working on the ApoE gene(Figure 1). Wondering how ApoE gene variations mediate cardiovascular disease (CV), she types in a query with parameters "ApoE, CV"; the database returns the ApoE region of the genome in (say) ten patients with CV. Since the ApoE gene is $\approx$ 0.1 Mbases, the data can return in a few seconds to Alice's tablet if the query is processed in seconds in the cloud. Customized bioinformatics software on her machine mines the data to identify variations.

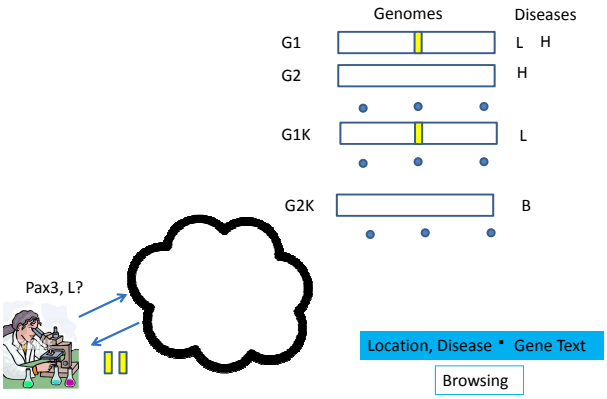

Figure 1: **Scenario 1. Browsing the genome.**

2. Drug Discovery: Bob, a drug designer, knows that deletions in tumor suppressor genes are key to cancer progression. He types in a query that asks for gene deletions common to all acute lymphoblastic leukemia (ALL) patients, and those common to acute myeloid leukemia (AML) patients with the goal of discovering bio-markers, a small but distinct set of deletions that are characteristic of the Leukemia subtype, and can thus be mined and returned in a few seconds.

3. Personalized medicine: Charlie, a physician who works with an elderly patient Sally, finds that Sally has side-effects with all common hypertension (HT) medication. Genome analysis shows that Sally has a so-called SNP (a mutation) in gene X of Chromosome 1, and this SNP is uncommon. Charlie types in a query that asks for all HT treatments for patients with variations in gene X. The search returns a rare HT medication called iloprost[5].

Just as interactivity has transformed other fields (e.g., from librarians to search engines, punch cards to time-sharing), we suggest that interactivity can also transform genetics. Of course, interactivity only allows *rapid hypothesis generation*. Ultimately, all hypothesis have to be tested in wet labs (Scenarios 1 and 2) or in medical clinics (Scenario 3). Despite this, since lab experiments and patient trials are expensive, quickly ruling out unworkable hypotheses is worthwhile. More precisely, interactive genomics can gain in each step of discovery:

Assemble: The cloud database allows selecting a suitable *logical* cohort (e.g., breast cancer patients over 45) in seconds. *Sequence:* Queries made on genomes already in the database do not see the delay of sequencing and shipping. *Analyze and Share:* Analysts write short declarative queries using a language like SQL. If every result states the query used, verification and reuse are facilitated.

This vision is not as distant as it may appear. Church's PGP (Personal Genome Project [26]) already has a database of 2000 genomes annotated with medical history from volunteers who have given up the right to privacy. However, the query facility in PGP is crude and only allows retrieval of whole genomes.

1.2 Related Work

The popular UCSC genome browser [22] can be used to browse regions of the reference genome. However, the UCSC browser does not allow browsing individual genomes, and only allows browsing by location. Variant callers (e.g., [14, 31, 19] take a donor genome and output a much smaller list of variations compared to the reference. Unfortunately, variant calling is an inexact science; results show [18] that the major variant callers differ greatly.

Custom genomic analysis can benefit from existing frameworks such as GATK [27], Samtools [25], and BEDtools [29] that offer routines for commonly used biological functions. However, these tools require substantial hand-crafted procedural scripts and are slow (Section 8.3).

By contrast, GQL, as described in this paper, allows analysis using concise declarative statements together with automatic optimization. Our earlier paper [**?**] described the GQL vision but no system details or optimizations. In a companion unpublished paper, we describe biological validation of results on 70 Yoruban individuals using GQL.

1.3 Contributions, Paper Outline:

Our key contributions are:

1. Interactive Genomics (Section 1.1, Figure 1): We sug- gest querying noisy genomic data interactively to remove fruitless hypothesis quickly.

2. Layering for Genomics (Figure 5): We propose sepa- rating probabilistic inferencing from deterministic evidence gathering to allow interactivity and better software engineer- ing.

3. Operators for Genomic Sofware Processing (Section 4, Section 5): We propose abstracting genomic processing us- ing 3 operators that embody noise-tolerant interval process- ing.

4. Optimizations (Section 7): We show how to gain inter- active speeds using two novel optimizations: cached parsing and lazy joins.

5. Evaluation (Section 8): We show that GQL queries take 60 seconds to execute in the Azure cloud on an input BAM le of 83 GB. Further, GQL is 20x more concise and yet 8x faster than GATK for a prototypical query.

The paper is organized as follows. Section 2 describes background in variant calling. Section 3 outlines the system architecture and GQL operators. Section 4 describes the features of GQL. Section 5 describes sample biological queries and interactions via GQL. Section 6 details our implementation, and 5 key optimizations; Section 8 provides performance results. Section 9 describes incremental deployment to the UCSC browser and an existing caller. Section 10 describes lessons learned; Section 11 describes ideas for further scaling.

2 Genetics Background

Figure 2 models DNA as a string of characters drawn from the A,C, G, T alphabet. Each human cell contains two such strings of 3 billion characters, one inherited from the father and one from the mother. Current technologies cannot read the genome from left to right like a tape. Instead, after extraction, the DNA is cut randomly into a very large number of fragments of length L that depends on technology. For example, Illumina 454 uses $L = 150$. Each short fragment is then scanned by say optical technology to form a *Read* of length L.

Think of a Read as first picking the paternal or maternal copy with probability 1/2, selecting a random offset with some probability, and then reading the next L characters. The probability of error increases towards the end of a Read, which explains why Read lengths are limited today. To ensure that the entire genome is examined, this process is repeated many times.[2] The ratio of the sum of the lengths of all Reads to the size of the original genome is called the *coverage*. Intuitively, coverage is the average number of Reads that span any genome location. A Read stores meta data and L characters with an 8-bit quality score per character, $\approx$ 100 Gbyte BAM file for 30x coverage of a human.

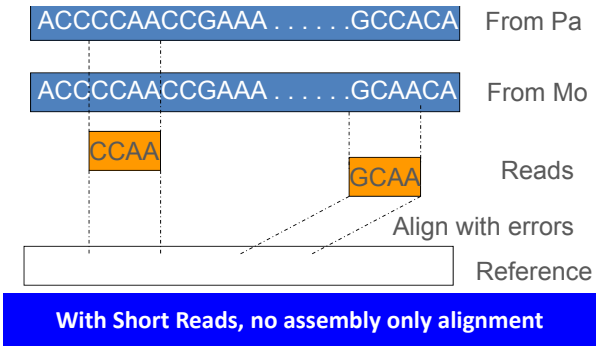

Figure 2: **Sequencing by Short Reads**

While reassembling Reads would be ideal, assembly is complicated by a large number of repetitive sequences, much as in a jigsaw puzzle with much blue sky. Instead, each Read is *aligned* to a reference genome called the Human Genome. In sum, cheap sequencing today results in a set of raw Reads. These Reads are aligned to find the closest string match with the reference, allowing substitution errors and a small number of inserted/deleted characters. The resulting file of Reads and matching location data is stored in a BAM file [25].

2.1 Variant Calling Today

Alignment is still useful without assembly because most humans are 99.5% similar, and so most Reads will align to the reference modulo small edits. Thus a genome can be compactly summarized by differences from the reference [24, 20]. Software *callers* process BAM files to produce a smaller VCF (Variant Call Format) file.

The simplest example of a variant call is a Single Nucleotide Variant (SNV) where a single character differs from the reference. Figure 3 shows a subject genome with an A in location 2000 of say the maternal genome while the reference has a C. This can be detected by first extracting all Reads that overlap location 2000 from the BAM file. Recall that alignment allows substitution errors.

Notice that some of the Reads in the Evidence have a C in location 2000, and some have an A. The final inference process must deal with 3 confounding error sources. The instrument could make an error and erroneously read a C as an A. Second, the mapping algorithm could map the Read to the wrong location. Third, if the maternal genome has an A but the paternal has an A in the same location, we expect half the Reads in the Evidence to have a $C$and half to have an A. Thus, SNV callers "weigh" the evidence and assign a probability for the location being

[2]This process is parallelized in "shotgun" sequencing.

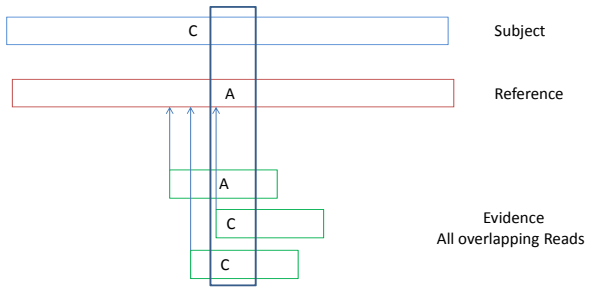

Figure 3: **Inferring SNPs. Evidence for the probabilistic call is the deterministic subset of Reads overlapping the SNP location.**

a SNV.

The probability is computed using an inference algorithm that takes into account the probability of instrument errors, the quality of the mapping and more sophisticated factors. Inference algorithms vary considerably and use a variety of Bayesian and Frequentist methods; however, all SNV callers we know use a deterministic subset of Reads (e.g., the Reads whose alignments overlap location 1000) we refer to as the *Evidence*.

Beyond SNVs, large deletions can be inferred (Figure 4) by modifying the hardware to fragment the genome into *pairs* of Reads that are a constant distance D apart. The donor genome is first fragmented into pieces of $2L + D$, and then L bases are read from the left and right ends. This limits the number of bases read to $2L$ but results in what is called "paired end" Reads that should normally be D apart. But consider what happens when a portion of the donor genome is deleted (Figure 4) and the paired ends are to the left and right of the deletion point. When mapped/aligned to the reference, the mapped locations will be a distance greater than D, in fact $D + X$ where X is the size of the deletion.

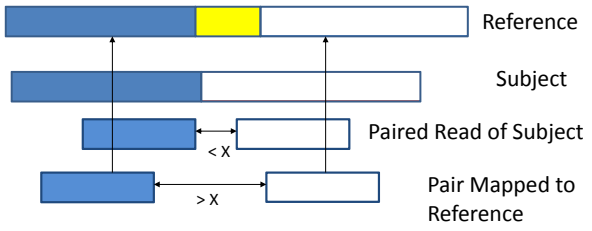

Figure 4: **Inferring Large Scale Deletions. The Evidence is the subset of discrepant Reads that overlap the deletion.**

Read pairs whose mapped locations differ beyond a threshold are called *discrepant* and indicate a deletion. As with SNPs, confounding factors include machine errors, mapping errors, and whether the deletion is in 1 or both copies; in addition, the distance D is often not fixed but has some distribution. Again, while inference algorithms vary, two standard pieces of evidence are discrepant reads and the reduction in the number of Reads mapped to the deleted portion.

Inversions are detected similarly when 1 Read is reversed when compared to the reference. Other analyses include haplotyping (assigning child variations to the mother or father) and copy number (replication of certain genome segments beyond a threshold). Genomic analyses matter because even a simple mutation can result in producing a harmful protein.

3 Proposed Genomics Architecture

Our architecture separates inference (probabilistic) from evidence (deterministic) into separate layers with a query language as the API.

3.1 Separating Evidence and Inference

Imagine a million genomes stored in the cloud as in Figure1. Downloading a single 100 Giga byte genome at current Internet speeds can take a day. For interactive access we must bring the query to the data. We could run current variant callers in the cloud and make relational queries on the smaller VCF files. However, this approach has several problems.

1. Differing Results: Current callers use a variety of inference techniques; results vary considerably [18]. Which one should one trust? *2. Hidden Evidence:* Partly as a result of Problem 1, biologists do not trust black box callers and often wish to examine the raw Reads (evidence) behind a call, which is hard with existing callers. *3. Too many calls:* The number of calls produced by a typical caller is too large for a human to sift. Running Breakdancer[14] on a typical full sequence (NA 18506 from 1000 Genomes) produces over 30,000 calls. *4. Slow Custom analysis:* Callers have a limited repertoire. Assembling a pipeline using a framework like GATK to do custom analysis requires a skilled coder, and takes days to code and much longer to run than the equivalent GQL query(Section 8.3).

Probabilistic processing suggests the need for heavyweight probabilistic databases [9]. However, all callers we know of use a deterministic subset of the Reads to make their calls; so it appears far easier to separate out genome analysis into two layers: one that gathers evidence, and one that assigns probabilities based on evidence on the right in Figure 5. Contrast this picture with the current layering picture shown on the left. Our proposed layering also uses a compression layer (e.g., [24, 20]) to reduce storage.

Our layering picture is similar to standard systems layering pictures. A possibly unusual feature is that the API between layers is a query language called GQL. Intuitively, GQL allows selecting arbitrary subsets of Reads from the Reads encoded in the BAM file.

For SNVs at location L, the subset is "All Reads whose mapped location overlaps L"; for deletions the subset is "All pairs of reads whose mapped distance is greater than D". Rather than hardwire these specific predicates, we

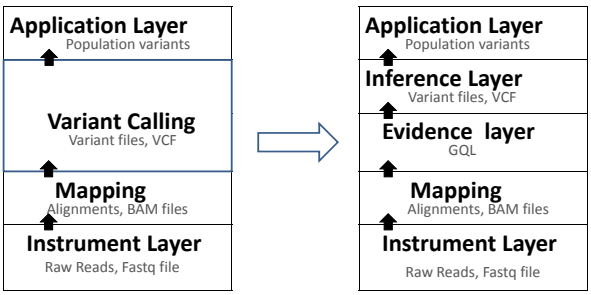

Figure 5: **Layering for genomics today (left) and our proposal (right)**

provide a more general set of operators. This allows us to incorporate new evidence as technology evolves, but also offers solutions to the 4 problems listed above.

First, it solves Problem 2 by making explicit the evidence used by any caller; evidence can now be retrieved and visualized independent of the caller; in fact a biologist can make GQL calls to retrieve subsets of Reads without going through a caller, as Alice does in Scenario 1.[3] Second, for Problem 3, GQL as an interval query language can be used to sift the called variations in VCF Format, itself a set of labelled intervals. Third, for Problem 4 GQL makes it easier to write custom analysis scripts compared to GATK as we will demonstrate in Section5.

The evidence-inference separation also offers help for Problem 1. First, if evidence is fairly standard but inference varies it makes sense to build a scalable evidence providing engine that answers GQL queries in the cloud, and relegate inference to the workstation where users can experiment with different callers. A clean API allows an industry to build a well-defined layer (Evidence) while inference is being standardized.

4 GQL Specification

The biggest conceptual difference between GQL and SQL is that intervals are *first class* in GQL because most inputs (Reads, genes) and outputs (regions of variation, VCF) are intervals. GQL uses Interval Tables which are standard database tables except for two well defined columns, IntervalStart and IntervalEnd, representing offset into the reference.

Conceptually, the BAM file itself is an Interval Table since it consists of a set of mapped Reads, and the mapped location (*Location* and *Location* + *length* form an interval). In practice, we have to map the BAM file into a Reads table because the BAM file is a list of Reads. The other columns in the Read Table include simple metadata such as the *MateLocation* and the strand number (DNA has two strands, this is a key parameter to detect inversions).

[3]Samtools allows retrieving Reads by location but do not allow more general queries such as retrieving discrepant Reads.

In addition to these standard tables, GQL accepts a freely formatted *Text* table which can be any table that a user defines. The user has the option of creating an interval table from any table by marking two of the attributes as begin and end and the third attribute is updated automatically.

The simplest operator is SELECT stolen from SQL, which selects a subset of rows based on arithmetic expressions on the metadata columns. A second and important operator is *IntervalJoin* which is a join based on interval intersection. In Figure 6 for example, imagine a table with rows containing intervals 1, 2, 3, and 4 and a second table containing intervals a, b, c all of whose relative positions on the reference line are shown. If we *IntervalJoin* these two tables, there will be joined entries for $(1, a), (2, a)$ and $(3, c)$ because only these pairs of intervals intersect. Further, if there are additional columns of Table 1 and Table 2, the corresponding values in these tables will be joined together in each output table. For example, if Table 1 had $(grape, 1)$ and Table 2 had $(a, cake)$, the joined tuple would become $(grape, 1, a, cake)$ in the output table.

We find *IntervalJoin* to be essential; we use it, for instance, to select subsets of BAM file by joining with the much smaller set of Genes; we also frequently use it to compare variations *between* human genomes, between parent and child, and between cancer and germline.

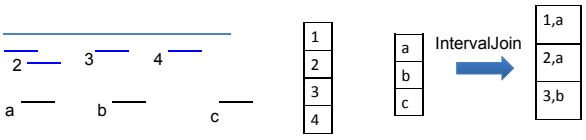

Figure 6: ***IntervalJoin* based on interval intersection. For example, $(1, a)$ is in the output because intervals 1 and a intersect**

While *IntervalJoin* appears in temporal [32] and spatial databases [8], our last operator may be more unique. *MergeIntervals* merges a set of input intervals to form a smaller set of intervals. For example, a deletion may result in a large number of discrepant Reads, all of which overlap. It is more useful to merge such discrepant Reads into wider deleted regions. However, two discrepant Reads may overlap erroneously. For robustness, we add a coverage parameter and require that all points in the merged region are covered by at least k Reads, where k is a parameter.

Note that MergeIntervals cannot easily be simulated in SQL using two variables for the start and end of a region. Consider two intervals $(10, 100)$ and $(50, 150)$. The region that contains 2 overlapping intervals is $(50, 100)$ which takes one value from the row containing $(10, 100)$ and one value from the row containing $(50, 150)$. This appears to be difficult with the relational Project operator. More generally, naive mappings to SQL have poor per-

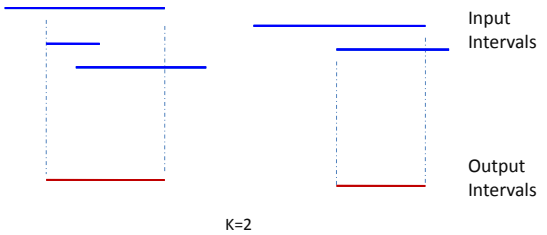

Figure 7: **In the simplest case, Merging Intervals corresponds to find the union of a number of intervals. More generally, it outputs the largest disjoint intervals such that at least k of the input intervals are contained in each output interval**

formance; building GQL allowed us control of the internals which enabled aggressive optimization (Section 7).

4.1 GQL Syntax

All GQL statements have the form SELECT ⟨attributes⟩ FROM ⟨tables⟩ WHERE ⟨condition⟩.

- The FROM statement specifies the input tables to the statement in the scope of this keyword.
- The SELECT statement corresponds to the Project operator and returns a subset of the attributes of the input table.
- The WHERE statement selects the subset of records of the input tables that satisfy the following filter expression.
- The *using intervals()* expression follows a table listed in FROM. It produces an interval for each entry of the corresponding table, allowing the user to specify both the beginning and ending of each interval. If the input table is of type READS the user has the ability to add the keyword *both_mates* as a third argument to the expression *using intervals* to denote that a mate pair is treated as a single interval. This expression does not return any table, and must be used with either *create_intervals* or *IntervalJoin*.
- The *create_intervals* function creates a table of intervals from the input table. When the function is called, the table in the FROMstatement is followed by the statement *using intervals(a,b)* so that the function knows which fields to use as intervals.
- The *IntervalJoin* statement which takes two tables as input and joins any two entries of those tables provided that the corresponding intervals intersect. The user specifies the intervals of interest with the expression *using intervals* next to each input table.
- The *merge_intervals(interval_count op const)* which is a function whose input table needs to be of type *Intervals*. It creates a new table of intervals from the regions that overlap with at least or at most the number of intervals specified inside the parenthesis.

The complete GQL manual can be found in [3].

5 GQL in Action

To give the reader a feel for using GQL, we present a quick example of data retrieval followed by a series of "explorations" of a breast cancer genome including a "diff" query between cancer and germline suggested by Haussler in his OSDI keynote [18].

Perhaps the simplest query is based on investigating variants in a 'favorite' gene. A researcher who is studying the role of a specific gene for some disease is likely to be interested in all variations in the gene. To investigate all structural variation (for example), she will query for paired-ends of Reads in these regions that either had *1.* one pair end unmapped, an indication of insertion or *2.* Distance between pairs greater than 1000, evidence of deletion in Illumina Sequencing or *3.* an inverted orientation from the expected one, suggesting an inversion.

We wrote a GQL deletion query in a few minutes in response to a request from a collaborator; the query took a minute to run on a cheap desktop (no parallelization), and immediately made all interesting data available to the researchers, also allowing them to change parameters of the query. To save space, we omit the query but it can be found in [4].

The next series of queries is on genomes with ids *a2-a04p-01* and *a2-a04p-10* from the TCGA cancer archive [23] which are from a tumor and normal cell respectively in the same breast cancer patient. We hope these queries convey that answers in genomics are rarely definite; instead, further queries are needed to refine or understand earlier results.

Deletion Query on Breast Cancer Genome: The evolution of a tumor genome is marked by many lesions on the genome. These genomic variations (involving single nucleotides as well as large rearrangements) often increase copies of oncogenes that promote tumor growth, and ablate or delete the effect of 'tumor suppressor genes' which check this growth either by invoking DNA repair, or through programmed cell death. Therefore, we investigated whole genome sequence from a breast tumor and matched normal sample from the TCGA project.

Our first step was to query using Algorithm 1 which looks for deletions by identifying regions that overlap with ≥ 5 length-discrepant paired-ends (mapped locations are between 700 and $100,000$bp; (distances longer than $100,000$ are considered unreliable).

The algorithm ran in 150 seconds and produced $20,862$ candidate deleted regions merely for chr1. This had far too many possibilities to explore manually, so we applied a number of filters using GQL.

Differencing the Cancer and Germline: We first asked for deletions that were not in the germline of the

Algorithm 1 **A query that looks for deleted regions.**

```
Disc_reads=select * from READS
where location>=0 and mate_loc>=0(
    (mate_loc+length-location>700 and mate_loc+
        length-location<100000) or
    (location+length-mate_loc>700 and location+
        length-mate_loc<100000)
)

Disc_intervals=select create_intervals() from
    Disc_reads using intervals(location,
    mate_loc+length, both_mates)

out=select merge_intervals(interval_count >= 5)
    from H2

print out
```

same individual as tumors are often dictated by somatically acquired mutations; such difference queries are standard [18]. Exclusion queries are hard to express in relational algebra; so we manipulated the interval operators of GQL as follows.

We created a table which concatenates the output of the execution of Algorithm 1 (which produces disjoint "deletion intervals") for both the tumor and the normal cell. We then computed the interval join of this table *with itself*. Thus we apply the *MergeIntervals* operator to select the areas that overlap with exactly one interval we eliminate common "deletion regions". Algorithm 2 shows the GQL query that encodes this procedure.

Algorithm 2 **Querying for deletions that are not common to cancer and germline.**

```
self_join=select * from intervaljoin
    combined_deletions using
intervals(begin, end), combined_deletions using
    intervals(begin, end)

self_join_intrvl=select create_intervals() from
    self_join using intervals
(begin1, end1)

diffs=select merge_intervals(interval_count <=
    1) from self_join_intrvl
print diffs
```

The output still contained $20,809$ entries for chr1 itself. It makes sense because many tumor genomes are extensively mutated relative to normal genomes, so much of the variation remained after the filter step.

Zooming in to Disrupted Genes: As a second filter, we investigated deletions that removed a significant fraction of the gene, as the loss of gene function is more apparent with large deletions, and also the filter is expected to be effective because coding regions account for only 2% of the genome. We performed a join with a table of gene intervals, and experimented with many

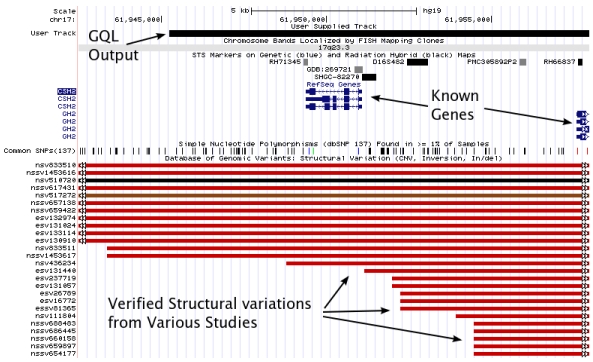

Figure 8: **Sample snapshot of the UCSC genome browser on an output interval from Algorithm 3**

versions of "significant disruption" by varying the minimum amount of overlap required from 1bp to 50% of the gene being deleted. An exemplar GQL query appears in Algorithm 3.

Algorithm 3 **Querying for genes that are significantly disrupted.**

```
large_del=select * from tumor_unique_del where
    (end-begin > 2000)

aff_genes=select * from intervaljoin
    refGenehg19 using intervals (txStart,
txEnd), large_del using intervals(begin, end)

del_50_percent=select * from aff_genes where
(txStart<begin and txEnd>end and (end-begin)>((
    txEnd-txStart)*1/2))
or
(txStart<begin and txEnd>begin and (txEnd-begin
    )>((txEnd-txStart)/2))
or
(begin<txStart and end>txEnd )
or
(begin<txStart and end>txStart and (end-txStart
    )>((txEnd-txStart)/2))

print del_50_percent
```

The deleted regions that overlapped with genes provide a treasure of data for this sample, and can be visualized on the UCSC genome browser (e.g.,Figure 8). Of a list of 24 known tumor suppressor genes (http://themedicalbiochemistrypage.org/tumor-suppressors.php), we identified deletions in 5. These include an 840bp deletions in the well-known familial breast cancer gene BRCA1. Mutations in BRCA1 are used as bio-markers [13]. We also identify multiple deletions in the Cadherin gene cluster on Chr 16. A nearby family member CDH1 is a known tumor suppressor in lobular breast cancer. Other events include: multiple deletions in the gene *deleted-in-colorectal-cancer (DCC1)*, which (as the name suggests) is a tumor suppressor known to be deleted in colon cancer; deletion in PTEN, known tumor suppressor for multiple cancers, including breast; multiple deletions in RB1, the first confirmed tumor suppressor gene.

While additional experimentation needs to be done to

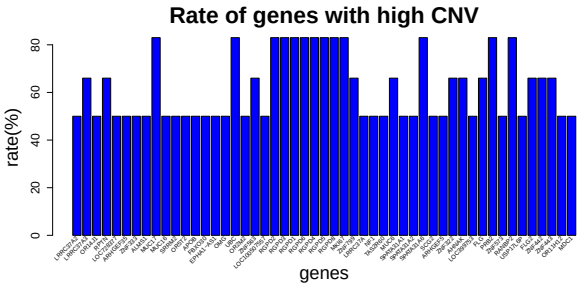

Figure 9: **Genes that overlap with a copy number region in at least 60% of the exomes of our population.**

validate these deletions, the advantage of using GQL to quickly interact with the data and identify interesting cases is apparent. While many of our assumptions (e.g., "significant disruption") are arbitrary, the reader should note that all genetic analysis is rife with such assumptions. GQL allows us to *quickly* modify assumptions (e.g. 50% to 25% disruption) and hypothesis (e.g., deletions to inversions) and understand the results, just as we interactively debug a C program using gdb.

Population Queries: So far we have concentrated on a single patient. Beyond deletions of cancer suppressors, a second hypothesis is that cancers multiply certain genes called *oncogenes* which should have a high copy number. A natural query is to look for genes with high copy numbers across a set of patients with the same cancer. The TCGA database has relatively few full sequence genomes but a large number of cancer genomes that only have exomes (the portion corresponding to genes). Exome data has a number of limitations especially in inferring deletions; however, exome data suffices for detecting high copy number. We ran the copy number query described in [2] on 5 exomes from TCGA, 3 of which are tumor and the rest are normal. The query identifies genes that overlap with a region of at least 1K bases long whose copy number is at least 300. Figure 9 shows the genes that commonly overlap with high copy number regions in at least 60% of the population.

6 System Design

Figure 10 depicts the GQL pipeline. A user enters GQL code which gets parsed with a compiler we wrote using Flex and Bison tools. The query planner, written in Python, assigns the appropriate execution routines from the backend and generates custom C++ files for the cached parsing optimization (see Section 7). The backend is written in C++ and uses a set of optimizations described in Section 7.

We omit details of the (fairly straightforward) Query Planner and proceed to describe the more interesting optimizations.

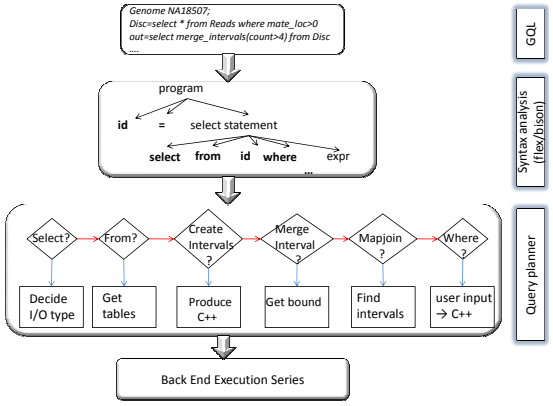

Figure 10: **The GQL pipeline.**

7 Optimizations

This section describes the optimizations that transformed GQL from a toy into a tool.

7.1 Connecting Pair Ends

We use a memory-efficient hashing mechanism to identify pair ends. Two reads are pairs in BAM when their *QNAME* fields are identical; *QNAME* is a string of $15 - 30$ characters. The conventional method which involves re-sorting of the file by the *QNAME* field, is slow. Moreover, keeping Reads sorted by mapping location allows quick range retrievals.

Instead, we hash *QNAME* and reduce memory by observing that a *QNAME* value should appear at most twice, for a Read and its pair. Thus, when processing a Read, we add its *QNAME* to a chaining hash table if it is not already present; if it is present, we identify the mate but also remove the older value from the hash table since it has been "matched". The matched mate information is added to an index table described next.

7.2 Meta Data for Speedup

Variant callers [14, 31, 19] make extensive use of the following four metadata fields of a read and its pair end including the mapped chromosome, the mapped location, the mapping strand, and the Read length. In particular, a typical query sifts through a billion locations searching for the few (say $10,000$) locations where "interesting" events occur such as discrepant Reads. The BAM file contains information (characters, quality scores) often irrelevant in determining interesting regions. Thus, considerable disk bandwidth can be saved by replicating Read metadata in a separate file, and scanning the meta data file instead of the BAM file to determine regions.

Storing the metadata file as a separate index file adds $10 - 30\%$ more storage but reduces query time by over $50\times$ (Section 8.1). When processing a query, the meta datafile of a chromosome is read into memory; query processing for the chromosome is completed before moving to the next chromosome.

A tabular array is used to index Reads where the first table entry keeps the information for the first read and so on. Each entry keeps four values: *Location* is the mapped location in the reference (4 bytes), *Strand* (1 bit denoting whether the Read is mapped to the Forward or Reverse Strand, *Byte Offset* is an 8-byte pointer to the location in the BAM file on disk where the full Read is stored; *Read Length* is the length of the Read; *Mate link* is a 4-byte offset into the array found by the preprocessing step of Section 7.2

Currently, we use around 20 bytes of metadata. We have not yet compressed the index file as in say vertical databases [28]. For example, the 8 byte offset into disk could be reduced to 1 byte with simple bucketing but it would complicate random access. Even without compression, our index per chromosome is small enough to fit into main memory of a cheap computer. For example, the size of the index for the largest chromosome (chr1) on a BAM file of coverage $35\times$ and read length of 100 is $1.5GB$.

Algorithm 4 Querying for Reads that map to the reference. Unmapped Reads have negative location by convention.

```
H1=select * from READS
where location >= 0
```

7.3 Cached Parsing

Interpreted expression evaluation can be expensive because of extra memory accesses. Consider the algebraic part of the *where* statement of Algorithm 4. The straightforward evaluation of this expression requires the backend to find for each read the value of *location* and use a stack-based calculator to evaluate the expression. However given the large number of reads (close to 10^9 per sequence), the overhead of memory-based stack operations becomes noticeable.

The GQL compiler eliminates this overhead by translating algebraic operations into custom C++ functions. In the example of Algorithm 4 the corresponding C++ customized function is shown below; the backend simply calls this function for each read. This is a major optimization as we show in Section 8.1.

```
int Reads::condition(i){
        return vector[i].location >=0;
}
```

7.4 Interval Tree Based Interval Joins

We optimize the implementation of *IntervalJoin* operator using a *centered interval tree* data-structure [6]. The most expensive part of evaluating a Join involves a search for overlaps between two arrays of intervals. Compared to the quadratic brute force algorithm, the interval tree allows an interval to query for intersection against a set of n intervals in time O($\log n$) time. Further, the construction of the interval tree takes time O($n \log n$) and $O(n)$ space.

The *center* of the root is the midpoint of the union of all intervals. The root contains all intervals that contain the center value. For example, for the intervals $(1-5)$, $(7-15)$, $(16-19)$, $(20-25)$ and $(22-28)$, the union of all intervals is $(1-28)$ and center of the root is 13. The tree is built recursively such that the left (respectively right) subtree contains intervals with end points completely to the left (respectively right) of the subtree *center*. When the query interval intersects the *center* of the root the output are the root intervals of the root that intersect with the query together with the output of the recursive querying of both the left and right subtrees.

7.5 Lazy Joins

As the name indicates, Lazy Joins postpone materializing a Join, as the cross-product of GQL tables can be large. Instead, the *IntervalJoin* table of two tables A and B is represented as an array of tuples (x, y), logically denoting the joining of the $x-th$ entry of A with the $y-th$ entry of B. Since one or both of the source tables can also be *IntervalJoin* tables, we need to maintain such "logical joining" recursively using a tree.

Consider for example table E of Figure 11a the result of the *IntervalJoin* of tables C and D, which in turn is the result of the *IntervalJoin* of tables A and B. We assign to any unknown variable of an expression a structure which keeps track of which entry of the parent table contains the desired value; we recursively update the fields of this structure when we open the parent table. In Figure 11a, assume that one needs to calculate the value $E[i] = arga + argb + argc$. Since $arga$, $argb$ and $argc$ are unknown variables, we assign a structure to each of them as in Figure 11b.

The values of all three variables are initialized to unknown. After reading table E, we learn that $arga$ and $argb$ come from the $19-th$ entry of table D while $argc$ comes from the $40-th$ entry of table C. Since the latter is a base table, we update the structure with the actual value of $argc$ as soon as we scan its contents. However, since table D is also a product of *IntervalJoin* we open it and learn from the entry $D[19]$ that $arga$ came from $A[18]$ and $argb$ from $B[21]$. Finally, we proceed as with table C to obtain the values for $A[18]$ and $B[21]$, and end up with the final state shown in Figure 11b.

In evaluating a SELECT operation on a *IntervalJoin* table, we simply evaluate the provided boolean expression on all tuples of the table and outputs those tuples that satisfy the expression.

We have not found lazy joins described in the database literature. In standard query planning [30], if a small portion of the results of several joins is selected at the end of a query, the last Select can be hoisted in front of the Joins to optimize the query. However, for Interval-based Joins such query optimization is unsound, especially when the final Select statement chooses regions covered by k intervals.

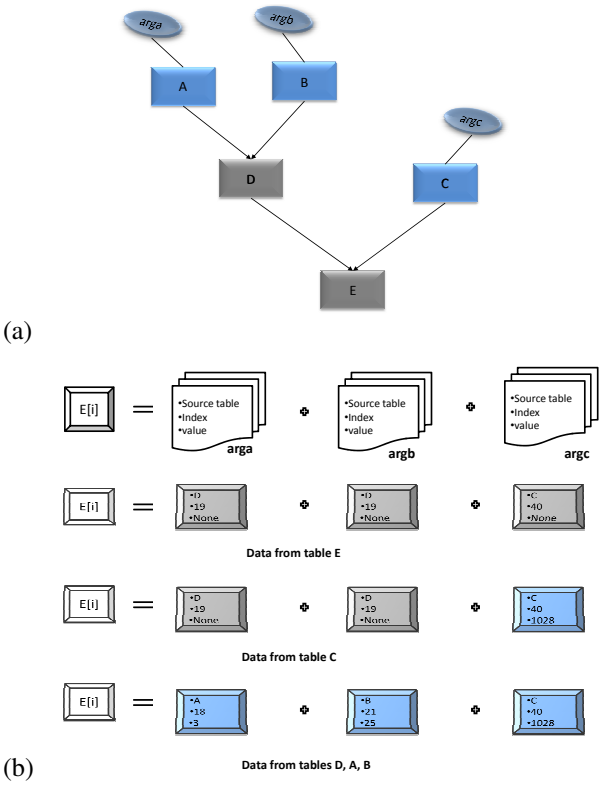

Figure 11: **An example of GQL evaluating a set of nested joins**.

7.6 Stack Based Interval Traversal

We implement the *MergeIntervals* operator using a stack-based genome traversal inspired by how a parser evaluates expressions with balanced parentheses. The characters in the expression correspond to locations in the reference; left parenthesis corresponds to interval start, right parenthesis to interval end. When an interval starts, we add it to the stack; when an interval ends we pop from the stack. The number of elements in the stack at any point is the number of open intervals at that location.

8 Evaluation

We describe microbenchmarks in Section 8.1, macro benchmarks in Section 8.2, and a comparison with writing queries in the existing GATK framework in Section 8.3.

8.1 Microbenchmarks

Our microbenchmarks are designed to evaluate the effectiveness of our optimizations. Our experiments were performed on the full Illumina sequencing of genome NA18506 in the 1000 Genomes project, which is the son of a family from the Yoruba region in Nigeria. The dataset is a BAM file of 75 GB which we split into chromosomes, the largest being 6.5GB.

Indexing performance Index building is much faster than sorting. A subset of approximately $97M$ reads from genome NA18506 that map to chr1 required 6 minutes to create the index using a memory footprint that did not exceed $2GB$. By contrast, sorting the same entries using the *QNAME* field took 63 minutes. Figure 12 shows the total time that our indexing takes and the memory footprint that is used for each chromosome. We also compare this time with the time it takes to sort chromosomes per *QNAME*

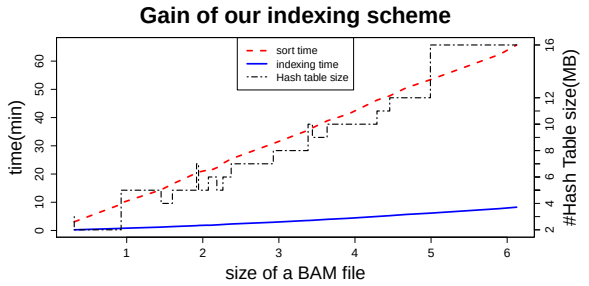

Figure 12: **Comparison of our hash table based indexing method with sorting. Sorting per *QNAME* gives a lower bound of the time it takes to discover the links between the pair ends in a BAM file. The figure also contains the respective maximum required capacity of our hash table.**

Speedups due to Metadata indexing Figure 13 compares taken to compute the query of Algorithm 4 data using the metadata index file versus directly from the BA file. We find speedups of nearly 100x. Since the metadata file is only 3x smaller than the BAM file, most of the gains arise because we load the metadata using sequential disk reads while the BAM file processing uses random access disk I/O. One could theoretically load a large number of Reads from the BAM file to reduce this discrepancy, but BAM APIs currently do not make this easy to do.

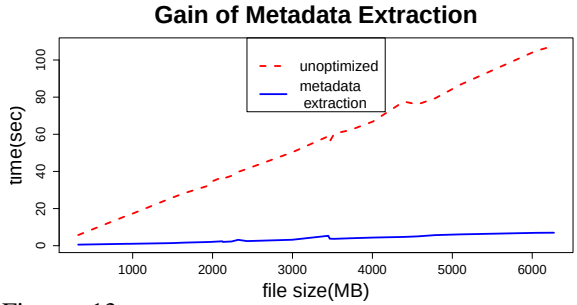

Figure 13: **Comparison of execution times when query Algorithm 4 uses the BAM file instead of the metadata file. The file sizes are for chromosomes in NA18506 ranging from 320MB (chrY) to 6.5GB (chr1).**

Cached Parsing Figure 14 shows that cached parsing allows user expressions to be written without regard to their complexity. The results use queries with a single select statement whose *where* statement contains between 1 and 9 variables. The input dataset consisted of all 97 million reads of NA18506 that map to chr1. Note that even for 3 variables — which we commonly used in our queries for a Select (mate, location, strand) — the speedup is around 25x.

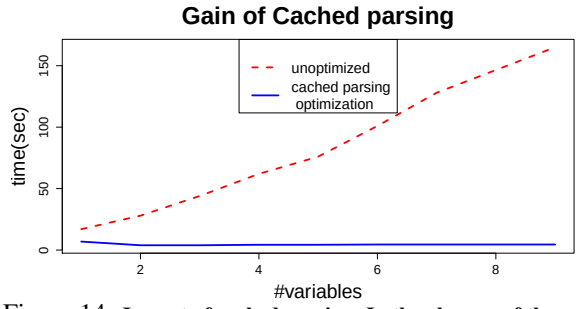

Figure 14: **Impact of cached parsing. In the absence of the optimization the execution time depends on the interpreter delay and grows linearly with the number of variables contained in an expression.**

Interval Tree Gains Figure 15 depicts performance gains that accrue from using interval trees to implement *IntervalJoin* compared to linear traversals. We join all reads of NA18506 that align with each chromosomes and the set of gene intervals provided by the UCSC genome browser. The x-axis shows the number of intervals that query the interval tree in each run; the leftmost y-axis show time and the right y-axis contains the number of intervals that are stored in the respective trees. It is easy to see linear scaling for naive processing with the number of intervals, and logarithmic scaling when using interval trees with speedups of around 70x for large sets of intervals.

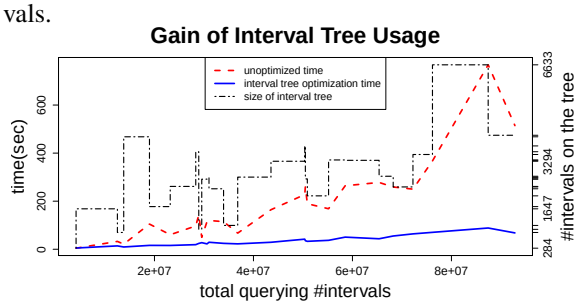

Figure 15: **Impact of using Interval trees**

Lazy Joins Figure 16 and Figure 17 show the power of lazy joins in a realistic use case. The experiment involves the interval join between the set of genes and the set of reads that overlap with each chromosome of NA18506. Note that the timing and space results of the opposing approach (not using lazy joins) are liberal lower bounds since they represent the *minimum* number of entries that have to be present in a join. In reality, the space and time requirements of not using lazy joins will be several times higher because Reads have to be replicated multiple times depending on how much they overlap.

Finally Table 1 shows that the reconstruction of physical entries of lazy join is efficient even in the face of multiple levels of join. The experiment measures the time for interval creation from entries of tables that have been emerged from a sequence of *IntervalJoin* operations. The first entry is from a table that merges the set of genes with a set of CpG rich islands, which is a text table found from the UCSC genome browser. The second entry joins the

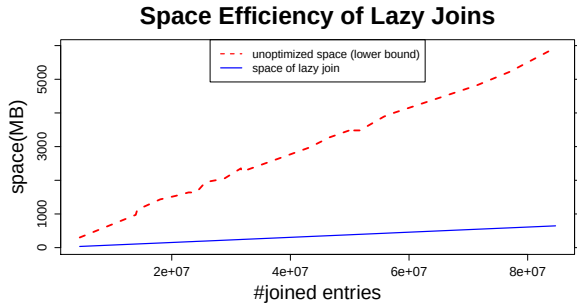

Figure 16: **Space savings for lazy joins.**

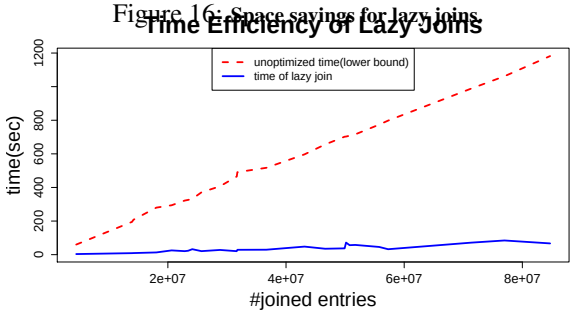

Figure 17: **Time savings for lazy joins.**

output of this table with Reads, the third entry joins the second entry with genes and so on.

#involved_tables	time(sec)
2	9.1
3	58
4	66
5	85

Table 1: **Ease of access of expression evaluations on lazily joined tables**

8.2 Macrobenchmarks

The parallelization experiments were performed on Windows Azure using 24 virtual machines. Each virtual machine was an Extra Large instance with 8 1.6GHz core, 14 GB RAM, and 2 TB local disk, costing $0.96 per hour. The BAM files for the experiment were stored in a virtual hard disk in Windows Azure blob storage, at cost of $0.07 per GB per month (non-georeplicated). Each VM worked on a separate chromosome in parallel.

Figure 18 shows the execution of algorithms 1-3 on Azure. Consider the most intensive query (Algorithm 1 on the tumor). Working on a single machine would take around 30 minutes (sum of per chromosome times) but working on 24 machines takes 2.5 minutes. The single machine solution costs around 45c to run the query but the parallel solution costs around 1$, a cost increase of 2 (due to idle machines) but a speedup of 12. Finer grain parallelism can easily increase query times to seconds at low costs.

8.3 Comparison with GATK

We also implemented the deletion query of Algorithm 1 using the commonly used GATK (Genome Analysis

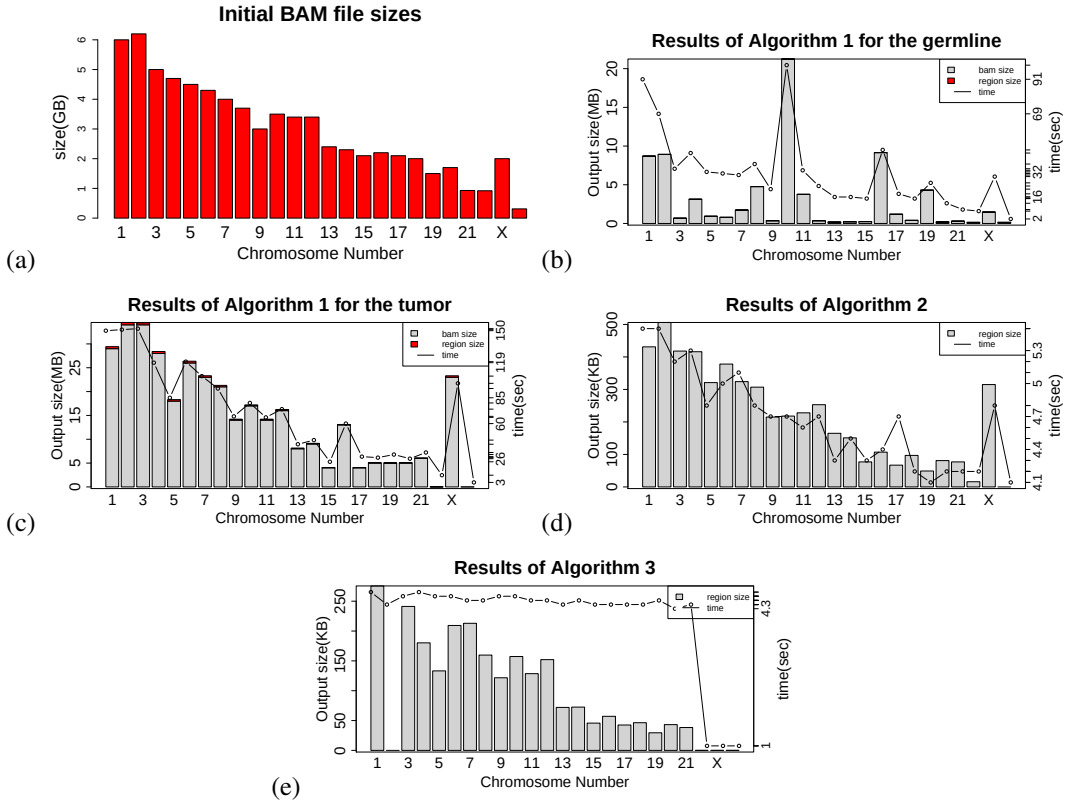

Figure 18: **Macro benchmark experiments on Azure Virtual Machines. The execution times shown at (b)-(e) are per node execution times.**

Toolkit) [27]. We implemented the query as a ReadWalker which processes raw reads through a series of filter, map, and reduce stages to find regions where there are 5 or more read pairs spanning a potential deletion. In the filter stage, we wrote GATK code that essentially simulated the internal GQL implementation of merge interval to find discrepant Reads that whose mapped distance lies between 500 and 1,000,000 bases, using a priority queue (instead of a stack). During the reduce phase, it unions all the intervals to product a set of disjoint intervals that contain potential deletions.

The algorithm comprises about 150 lines of Java code, and took an experienced software engineer about a day to write and debug, starting with moderate familiarity with Java, and no background in writing GATK extensions. The corresponding code took around 8 lines of GQL. A comparison can be found in [1]. Finally, on a single VM when run on chr1 of the breast cancer genome *a2-04p-01.bam* used in Section 8, the GQL code took 2 minutes to run and the GATK code took 13 minutes.

The gains in simplicity and performance are likely to be larger for more complex queries. Most of the gain in performance on this query likely arose because we indexed the metadata. While users can theoretically do this as well, it adds an additional burden of code compared to automatic optimization in GQL, akin to using ISAM [30] before SQL. Optimizing is even harder for users with complex queries with nested Joins. While we have not done so yet, declarative queries allow optimizations across *sequences* of statements by reordering and fusing operators, which seems impossible with frameworks like GATK and BEDtools.

9 Incremental Deployment

GQL can enhance two standard tools used every day by biologists: callers and browsers.

9.1 Speeding up existing callers

In this experiment we demonstrate the speedup that GQL can provide to an *existing* structural variation caller called Breakdancer [14]. A normal run of Breakdancer_max with input being the chr2 alignments of the breast cancer genome *a2-a04p-01* of Section 8 (a BAM file of size 6.7GB) takes 25 minutes, and produces a collection of variations including deletion, inversion, insertion and translocations events.

We used GQL to filter the original BAM file to retain only reads (103MB) that are needed by Breakdancer [14], as determined by reading their algorithm description. Next, we ran Breakdancer_max again using the filtered input. This time the tool needed only 4 mins., a 6× improvement in speed that can be attributed to the reduced data in the input file.

To measure the compatibility of the results between the two runnings we assume an identified variant from the initial experiment as *consistent* with the second run

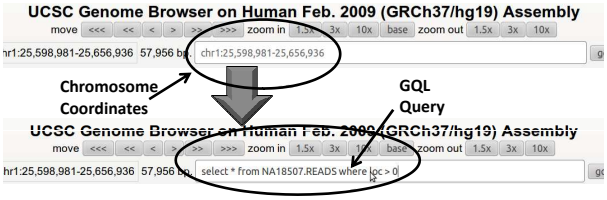

Figure 19: **Semantic versus location browsing.**

if it overlaps by at least 50% of its length with the latter. With this definition, we found that 383 out of 393 of the deletions, 251 out of 256 intra-chromosomal translocations, 6766 of 7152 inversions, and 77 of 4110 of insertions are consistent in the Breakdancer run on the BAM file and the run filtered by GQL, showing that the performance improvement did not come at the expense of call accuracy. Note that these results are based on our surmise of the evidence used by Breakdancer from reading their paper. More accurate results could be obtained if the writers of the caller write the GQL query to gather the evidence they *know* they need.

9.2 Semantic Browsing

While the UCSC browser allows reference genome browsing, it has two limitations today: *1.* Only the reference genome can be browsed. New genomes cannot be loaded easily. *2.* Only browsing by location is supported, which we refer to as *syntactic* genome browsing.

By contrast, GQL allows browsing for all regions containing reads that satisfy a specified property(e.g., discrepant reads). For regions, it was trivial to convert GQL interval tables to the BED format used by the UCSC browser and view the results on the UCSC browser. We refer to this as *semantic* genome browsing and gave an example in Figure 19. Ideally, the UCSC browser should be modified to directly support a field to enter GQL queries as mocked up in Figure 19. Our performance results indicate that semantic browsing can be done in minutes if the answer to the query is small.

10 Lessons learned

1. Genomics requires exploration: After using GQL to find deletion in a Yoruban male NA18506 we found conflicting results from Conrad et al [15] on NA18506. Sifting with further GQL queries revealed that the purported deletions found only in [15] had low coverage. Such quick explorations in response to unexpected results are well supported by GQL.

2. General constructs helped: We started by building of modules (as in BEDtools, GATK) used in genome analysis such as "Find discrepant intervals" but the general constructs have proved their use in unanticipated ways.

3. MergeInterval was hard to get right: We tried to do it only with Joins and Selects. To identify output intervals that overlapped k input intervals, we attempted to Join with a virtual table which contained all possible output intervals. We are happier with the simplicity of MergeInterval.

4. Additional operators will be needed. A GroupBy and Count operator would benefit several biological queries including Haplotyping. Other operators/refinements may be needed to simulate the variety of Bioinformatics algorithms.

6. Restructure for population queries: Some design decisions such as running each GQL query in a separate directory on a single genome make population queries (e.g., Section 5) currently hard. Population queries in GQL today are also iterative and hence slow. Scaling to queries on a million genomes in a few seconds will require new indices and restructuring.

11 Scaling GQL

Scaling of GQL to a million genomes could arise from:

1. Materialized Views and Query Planning: Our metadata index table is a materialized view [7] of selected columns in the BAM file. A second potential view would be a file containing the text of a Read without quality scores. Views on "rows" are also compelling; for example, one could store an exomic view corresponding only to genes. A query planner could compute the minimal set of views to read to answer a given query, minimizing disk and memory bandwidth. Views should be compressed as in [28].

2. Strength Vectors: Consider the query "Find variations common to 80% of blastoma patents that are not not in 20% of normal patients". As callers are in flux, an alternative is to use GQL to find a coarse set of intervals for each genome that contains *any* potential variation for the genome that could be 1000X smaller than the BAM file. We read the coarse intervals from disk to populate a strength vector which contains an integer for each genome location, that counts the number of blastoma patients that have a variation in that location. A similar strength vector can be maintained for the normals; then a last scan of both strength vectors can solve the query.

3. Parallelism and SSDs: We have parallelized GQL by chromosome in this paper. Further parallelism would divide chromosomes into blocks with a small amount of overlap. SSDs could be leveraged to store genome metadata.

12 Conclusions

We described a vision for interactive genomics in which hypothesis can be generated and tested in minutes as opposed to days. Incorrect hypotheses can be discarded quickly without time-consuming, expensive wet lab/clinical trials. Genetics, like much science, produces conflicting and unexpected results; the ability to quickly pose queries to clarify or refine earlier results (as in Section 5) seems valuable.

Existing frameworks such as GATK [27] and BEDtools [29] raise the level of abstraction but still require significant effort to write/modify queries (hours, see Section 8.3), and are often slow to run because of lack of indexing and other optimizations. Users are free to write such optimizations, but the analyst wishing to quickly explore genomes is in a quandary: either she carefully optimizes her code which takes time, or her code takes long to run.

Separating evidence and inference allowed GQL to progress without recourse to probabilistic databases. The combination of *IntervalJoin* and MergeInterval operators worked well. The optimizations were key, improving performance by 100x; cached parsing and lazy joins are potentially novel. While aspects of GQL appear in many databases, we found it crucial to get the whole package right.

GQL is still evolving and must retool to scale to a million genomes. Solutions to privacy must be found via contractual arrangements or ideas such as differential privacy [17]. Inference algorithms must be standardized. But the siren song of indexing the world's genomes is alluring; the systems community is needed to make this dream real.

References

[1] A delete finder on gatk. `http://cseweb.ucsd.edu/~ckozanit/gql/supp/CountOverlapsWalker.java`.

[2] Gql code that looks for cnv events on exome sequencing. `http://cseweb.ucsd.edu/~ckozanit/gql/exomehighcnv_gql.txt`.

[3] The gql manual. `http://cseweb.ucsd.edu/~ckozanit/gql/manual.pdf`.

[4] A hello world example. `http://cseweb.ucsd.edu/~ckozanit/gql/supp/helloworld_gql.txt`.

[5] Iloprost. `http://en.wikipedia.org/wiki/Iloprost`.

[6] Interval tree. `http://en.wikipedia.org/wiki/Interval_tree`.

[7] Materialized view. `http://en.wikipedia.org/wiki/Materialized_view`.

[8] Open source gis history. `http://wiki.osgeo.org/wiki/Open_Source_GIS_History`.

[9] Probabilistic database. `http://en.wikipedia.org/wiki/Probabilistic_database`.

[10] Health Information Technology for Economic and Clinical Health (HITECH) Act. *Title XIII of Division A and Title IV of Division B of the American Recovery and Reinvestment Act of 2009 (ARRA)*, L(111-5), Feb 2009.

[11] The dragon's DNA. *Economist*, Jun 2010.

[12] 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.

[13] A. Antoniou, P. D. Pharoah, S. Narod, H. A. Risch, J. E. Eyfjord, J. L. Hopper, N. Loman, H. Olsson, O. Johannsson, A. Borg, B. Pasini, P. Radice, S. Manoukian, D. M. Eccles, N. Tang, E. Olah, H. Anton-Culver, E. Warner, J. Lubinski, J. Gronwald, B. Gorski, H. Tulinius, S. Thorlacius, H. Eerola, H. Nevanlinna, K. Syrjakoski, O. P. Kallioniemi, D. Thompson, C. Evans, J. Peto,

F. Lalloo, D. G. Evans, and D. F. Easton. Average risks of breast and ovarian cancer associated with BRCA1 or BRCA2 mutations detected in case Series unselected for family history: a combined analysis of 22 studies. *Am. J. Hum. Genet.*, 72(5):1117–1130, May 2003.

[14] K. Chen et al. BreakDancer: an algorithm for high-resolution mapping of genomic structural variation. *Nat. Methods*, 6(9):677–681, Sep 2009.

[15] D. Conrad, T. Andrews, N. Carter, M. Hurles, and J. Pritchard. A high-resolution survey of deletion polymorphism in the human genome. *Nat Genet*, 38(1):75–81, Jan 2006.

[16] L. DeFrancesco. Life Technologies promises $1,000 genome. *Nat. Biotechnol.*, 30(2):126, Feb 2012.

[17] C. Dwork. Differential privacy. In *in ICALP*, pages 1–12. Springer, 2006.

[18] D. Haussler. Keynote address: The ucsc cancer genomics hub. `https://www.usenix.org/conference/osdi12/keynote-address`, 2012. Keynote on USENIX OSDI.

[19] F. Hormozdiari, C. Alkan, E. E. Eichler, and S. C. Sahinalp. Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes. *Genome Res.*, 19(7):1270–1278, Jul 2009.

[20] M. Hsi-Yang Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res.*, 21(5):734–740, May 2011.

[21] J. P. Ioannidis, D. B. Allison, C. A. Ball, I. Coulibaly, X. Cui, A. C. Culhane, M. Falchi, C. Furlanello, L. Game, G. Jurman, J. Mangion, T. Mehta, M. Nitzberg, G. P. Page, E. Petretto, and V. van Noort. Repeatability of published microarray gene expression analyses. *Nat. Genet.*, 41(2):149–155, Feb 2009.

[22] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome Res.*, 12(6):996–1006, Jun 2002.

[23] D. C. Koboldt et al. Comprehensive molecular portraits of human breast tumours. *Nature*, 490(7418):61–70, Oct 2012.

[24] 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.

[25] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, Aug 2009.

[26] J. E. Lunshof, J. Bobe, J. Aach, M. Angrist, J. V. Thakuria, D. B. Vorhaus, M. R. Hoehe, and G. M. Church. Personal genomes in progress: from the human genome project to the personal genome project. *Dialogues Clin Neurosci*, 12(1):47–60, 2010.

[27] A. McKenna et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res.*, 20(9):1297–1303, Sep 2010.

[28] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9:680–710, 1984.

[29] A. R. Quinlan and I. M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, Mar 2010.

[30] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[31] S. Sindi, E. Helman, A. Bashir, and B. J. Raphael. A geometric approach for classification and comparison of structural variants. *Bioinformatics*, 25(12):i222–230, Jun 2009.

[32] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.