UNIVERSITY OF CALIFORNIA, SAN DIEGO

Compressing and Querrying the Human Genome

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Christos A. Kozanitis

Committee in charge:

Professor Vineet Bafna, Chair
Professor George Varghese, Co-Chair
Professor Alin Bernard Deutsch
Professor Rajesh Gupta
Professor Theresa Gaasterland

2013

The Dissertation of Christos A. Kozanitis is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California, San Diego

2013

DEDICATION

To the memory of Christos Nasikas

EPIGRAPH

῞Η τάν ἤ ἐπί τᾶς

Farewell of Spartans mothers to the departing warrior while giving him his shield. The meaning is: Either win and come back with this shield or die and come back on it. But do not save yourself by throwing it and running away.

Δόξα τῳ Θεῴ, τῳ καταξιώσαντί με τοιούτον ἔργον ἐπιτελέσαι.

Said by Byzantine Emperor Ioustinianos (Justinian) I upon the completion of the building of Hagia Sophia in Constantinople in 537AD. The meaning is: Glory to the God who helped me put through this project

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to thank all the people with whom I shared the journey of my graduate studies.

First of all I would like to thank my family for all the support and encouraging during all these years.

In addition, I would like to express my gratitude to my advisor George Varghese for his great mentorship and guidance all these years and his providing me with nothing less than what any graduate student would imagine from his advisor. I was blessed to spend plenty of hours (per week) talking with him both on research and personal problems and grow both as a person and as a professional. After a while he was not my "boss" any more but a powerful partner/friend who was fighting on my side trying to perfect the output of my research, secure my funding each year, help me find summer internships strongly relevant to my research, and finally help me get job offers from the best academic places in the world.

Equally thanks to Vineet Bafna whom I was fortunate to have as co-advisor and chair of my thesis committee. Vineet took great care of me, especially during the last three years that George was away from San Diego. He guided steadily both myself and George in the area of his expertise, genomics, and he vigilantly kept us away of problems of minor importance in his community. He had the energy and patience to be there and listen to a lot of random (and sometimes naive) ideas and he was always kind enough to explain to us the important problems of his field. He also edited our papers to make sure that they are of interest to the people of his community.

I would also like to thank Alin Deutsch for providing his database expertise in our GQL project and giving me confidence during its first stages.

Moreover I would like to thank the remaining two members of my thesis committee for their help and support: Rajesh Gupta and Terry Gaasterland. Rajesh was one of the first people at UCSD who believed in our ideas and his constructive feedback helped me getting a CSRO fellowship from the CalIT^2. Finally, Terry for her constructive comments during my thesis proposal, for her enthusiasm on database-related research in Bioinformatics and for her encouragement to continue working on it.

I would also like to thank my supporters of my projects: Lucila Ohno-Machado, Vish Krishnan, Ramesh Rao, and Larry Smarr. Lucila leads the iDASH project at UCSD and she expressed her support to GQL through funding during the last two years, co-authoring our CACM paper and advertising GQL to the medical world. Vish, Ramesh and Larry provided useful discussions and support in the means of a CSRO fellowship during the pre-mature stages of GQL.

Thanks also to my collaborators from industry Semyon Kruglyak and Chris Saunders from

Illumina and Flavio Bonomi, Sushil Singh and John Hubber from Cisco. I worked with Semyon and Chris in a highly productive summer (June-September 2009) and that collaboration enabled the development of SLIMGENE. In the earlier years of my graduate life when I was still working in networking problems, I enjoyed collaborating with Flavio, Sushil and John at Cisco during an internship (Summer 2007) which gave birth to our Kangaroo packet parser.

I would also like to thank my collaborators Andrew Heiberg and Myoung Lah that went out of their way and found some time in their busy MS schedule to contribute to the GQL and SLIMGENErespectively. Andrew created an amazing GUI that helped us showcasing the power of GQL. He was always there to make sure that the GUI was running online with no problems. Myoung modernized SLIMGENEby adding support for BAM files. He worked hard to make his modifications efficient and understand all the back end details of the samtools API.

I would also like to thank all members of George's and Vineet's labs, especially The Vinh (Terry) Lam, Sangwoo Kim, Nitin Udpa, Anand Patel for useful discussions and support.

Last but not least I would like to thank all my friends that helped me enjoy life to the maximum and with whom I created beautiful memories from Southern California.

Chapter 2, in full, is a reprint of the material as it appears in "Compressing Genomic Sequencing Fragments Using SLIMGENE" in *Journal of Computational Biology*. Kozanitis, Christos; Saunders, Chris; Kruglyak, Semyon; Bafna, Vineet; Varghese, George. Vol 18, Issue 3, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 1 and Chapter 3 in part, are a reprint of the material as it appears in:

1. "Abstractions for Genomics", in Communications of the ACM, Bafna, Vineet; Deutsch, Alin; Heiberg, Andrew; Kozanitis, Christos; Ohno-Machado, Lucila; Varghese, George. Volume 56, Issue 1, 2013.

2. "Using the Genome Query Language (GQL) to uncover genetic variation.", Kozanitis, Christos; Heiberg, Andrew; Varghese, George; Bafna, Vineet. To appear in Bioinformatics

3. "'Making Genomics Interactive: Towards Querying a Million Genomics in a few seconds.", Kozanitis, Christos; Bafna, Vineet, Pandya, Ravi; Varghese, George. Submitted to the Symposium of Operating Systems and Principles (SOSP) 2013.

The dissertation author was the primary investigator of these papers. The dissertation author was also the primary author of papers 2 and 3.

Chapter 4, Chapter 5 and Chapter 6 in part, are a reprint of the material as it appears on papers:

1. “Using the Genome Query Language (GQL) to uncover genetic variation.”, Kozanitis, Christos; Heiberg, Andrew; Varghese, George; Bafna, Vineet. To appear in Bioinformatics

2. “‘Making Genomics Interactive: Towards Querying a Million Genomics in a few seconds.”, Kozanitis, Christos; Bafna, Vineet, Pandya, Ravi; Varghese, George. Submitted to the Symposium of Operating Systems and Principles (SOSP) 2013.

The dissertation author was the primary investigator and author of these papers.

Finally I acknowledge my funding resources for the papers of this thesis. This work was funded in part by the National Institutes of Health-sponsored iDASH project (Grant U54 HL108460), NIH 5R01-HG004962, and a Calit2 Strategic Research Opportunity scholarship for Christos Kozanitis. We also acknowledge support from the NSF (IIS0810905) and and a U54 center grand (5U54HL108460).

VITA

2006	Diploma in Electronic and Computer Engineering, Technical University of Crete, Chania, Crete, Greece
2009	M. S. in Computer Science, University of California, San Diego, California, United States
2006–2013	Graduate Student Researcher, University of California, San Diego, California, United States
2013	Ph. D. in Computer Science (Computer Engineering), University of California, San Diego, California, United States

ABSTRACT OF THE DISSERTATION

Compressing and Querrying the Human Genome

by

Christos A. Kozanitis

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2013

Professor Vineet Bafna, Chair
Professor George Varghese, Co-Chair

With high throughput DNA sequencing costs dropping below $1000 for human genomes, data storage, retrieval and analysis are the major bottlenecks in biological studies. In order to address the large-data challenges on genomics, this thesis advocates: 1) A highly efficient read-level compression of the data which is achieved through reference-based compression by a tool called SLIMGENEand 2) a clean separation between evidence collection and inference in variant calling which is achieved though our Genome Query Language (GQL) that allows for the rapid collection of evidence needed for calling variants.

The first contribution, SLIMGENE, introduces a set of domain specific lossless compression schemes that achieve over 40x compression of the ASCII representation of short reads, outperforming bzip2 by over 6x. Including quality values, we show 5x compression using less running time than bzip2.

Secondly, given the discrepancy between the compression factor obtained with and without quality values, we initiate the study of using lossy transformations of the quality values. Specifically we show that a lossy quality value quantization results in 14x compression but has minimal impact on downstream applications like SNP calling that use quality values.

The second contribution, GQL, introduces a novel framework for querying large genomic datasets. We provide a number of cases to showcase the user of GQL for complex evidence collection, such as the evidence for large structural variations. Specifically, typical GQL queries can be written in 5-10 lines of code and search large datasets (100GB) in only a few minutes on a cheap desktop computer. We show that GQL is faster and more concise than writing equivalent queries in existing frameworks such as GATK. We show that existing callers by an order of magnitude by using GQL to retrieve evidence. We also show how GQL output can be visualized using the UCSC browser.

Chapter 1

Introduction

"Now that we have a 1000 dollar genome, we need to make sure we don't have a 10,000 dollar analysis."
Francis Collins

In this chapter, we introduce the new paradigm of interactive genomics and the motivating applications of personalized medicine and large scale discovery. To help the reader, we next provide an overview of genomics for computer scientists. We then propose a radical refactoring of genomic software into layers akin to the layers used in many computer systems such as the Internet. We make a case for two layers in particular – a compression layer and an evidence layer; this thesis focuses on abstractions and mechanisms for these two layers. The chapter ends with a summary of the thesis contributions and the principal thesis statement: given falling hardware costs and the importance of interactively querying genomes, it is imperative to have new abstractions for efficiently retrieving and querying genome data.

1.1 Towards Personalized Medicine

Humans are a product of nature and nurture, meaning our phenotype (the composite of all outward, measurable, characteristics, including our health parameters) is a function of two things: our genotype (the DNA program in all cells) and the environment (all inputs to a human, like food and medicine). This arrangement is analogous to how the output of a program (such as a search engine) is a function of both the program and the input (keywords typed by a user). Using the same input with a different program (such as Google search vs. Bing) can result in different output. In this analogy, the role of the medical professional is to provide information that is "diagnostic" (such as, "Is there a bug in the program based on observed output?"), "prognostic" (such as, "Can output/outcome be predicted, given specific inputs, like diet?"), or "therapeutic" (such as, "Can a specific input, like a drug, lead to the desired output?"). Also, the electronic

medical record (EMR) of a patient can be viewed as an archive of previously acquired inputs and outputs.

Unlike computers, the human program is largely hidden. Hence, traditional medicine is "depersonalized", with doctors providing treatment by comparing the patient's phenotype (symptoms) against empirical observations of outputs from a large number of individuals. Limited customization is based on coarse classes, like "race." All this changed with the sequencing of the human genome in early 2000 and the subsequent drop in costs from hundreds of millions of dollars to $1,000 on small desktop sequencing machines. The ability to cheaply read the program of each human underlies the great promise of personalized medicine, or treatment based on symptoms and the patient's distinctive DNA program. We frame this point with a classic example: The blood-thinner drug Warfarin is widely prescribed to prevent blood clots. Dosage is critical; too high and the patient can bleed to death, too low and the drug might not prevent life-threatening blood clots. Often, the right dosage is established through multiple visits to the clinic and regular testing. However, recent reports [58] suggest that knowledge of the patient's genetic program can help establish the right dosage.

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[24].[1] *Data Velocity:* 30,000 full sequence genomes were produced in 2011 versus 2700 in 2010. The Beijing Genomics Institute [6] 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 [5] 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 [32]. 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.

[1]The costs refer to a 30$\times$ coverage of a single genome. Our thesis 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.

1.1.1 Today: 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 [50, 45] 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 [7] and the TCGA Cancer Archive [42]. However, download times and the use 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 [38]. Haussler's OSDI keynote [32] sounded similar themes: genomic big data, cancer as a driving application, and the need for new tools.

1.2 The Dream: Interactive Genomics

Consider the scenario (Figure 1.1) 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 1.1). 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 ≈ 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.

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

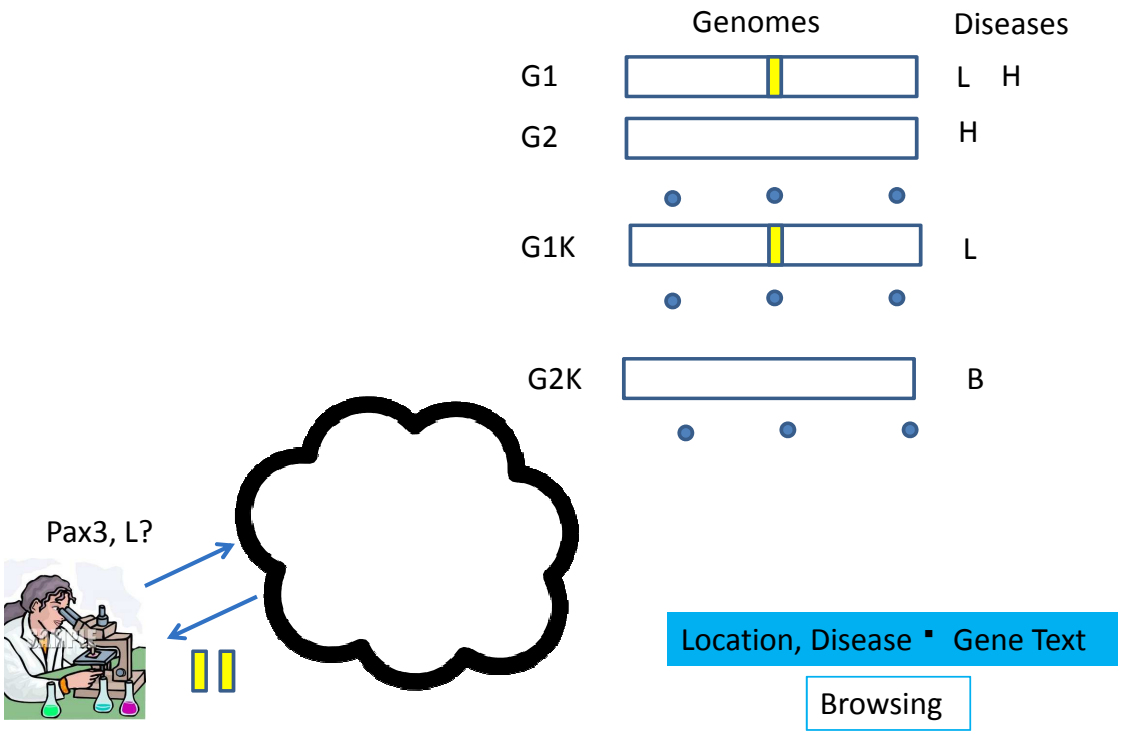


Figure 1.1. Scenario 1. Browsing the genome.

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[1].

Just as interactivity has transformed other fields (e.g., from librarians to search engines, punch cards to timesharing), 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 [48]) 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.

We limit our scope to genomics, ignoring dynamic aspects of genomic analysis (such as transcriptomics, proteomics expression, and networks). Genomic information is traditionally analyzed using two complementary paradigms: First, in comparative genomics, where different species are compared, most regions are dissimilar, and the conserved regions are functionally interesting.[33, 31] The second is population genomics, where genomes from a single population are compared under the baseline hypothesis that the genomes are identical, and it is the variations that define phenotypes and are functionally interesting. We focus on population genomics and its application to personalized medicine. We do not discuss specific sequencing technologies (such as strobe sequencing vs. color space encoding).

1.3 Genetics for Computer Scientists

We start with a brief introduction to genetics for computer scientists; standard references (such as Alberts et al.[9]) provide more details. All living creatures consist of cells, each like a computer running a program, or its DNA. The program uses three billion characters (nucleotides/base pairs, or bp) from a fixed alphabet $\{A,C,G,T\}$.Humans are diploid; that is, two programs control each cell, one inherited from the father and one from the mother. Further, each program is broken up into 23 "modules" called chromosomes, and within each chromosome are sparsely scattered small functional blocks called genes. The module pairs from the father and mother are called homologous chromosomes, with each human having a pair of (homologous) genes from each parent.

The "hardware" of the cell consists of cellular organelles and proteins - the cellular machinery. Proteins perform specific cellular functions (such as catalyzing metabolic reactions and transducing signals). The gene contains the "code" for manufacturing proteins, with each gene executing in one of many "ribosomes" (analogous to a CPU). Information travels from the nucleus (where the packaged DNA resides) to the ribosome via a "messenger" molecule (mRNA) that is essentially a copy of the coding DNA. The ribosome "reads" the code 3 bases (one codon) at a time; each codon is analogous to an OpCode instructing the ribosome to attach a specific amino acid to the protein sequence being constructed. The DNA program thus provides instructions for making the hardware that in turn performs all cellular

functions.

A change (mutation) in the DNA can change the amino acid and, correspondingly, the cellular machinery resulting in a different phenotype (output). In the Mendelian paradigm, each of the two homologous copies of the gene controls one phenotypic trait (such as ability to clot blood). A mutation in one gene might affect the phenotype strongly (dominant), not at all (recessive mutation), or somewhere in between. Most phenotypes are complex, controlled by the paired copies of multiple genes. Nevertheless, DNA controls traits, so even the simplest queries on DNA are useful (such as "Compared to a 'normal' individual, have parts of the patient's DNA program mutated?").

Advances in sequencing have made it possible to cheaply scan an individual's genetic program for mutations, or variations.

Figure 1.2 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.

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 [45].

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

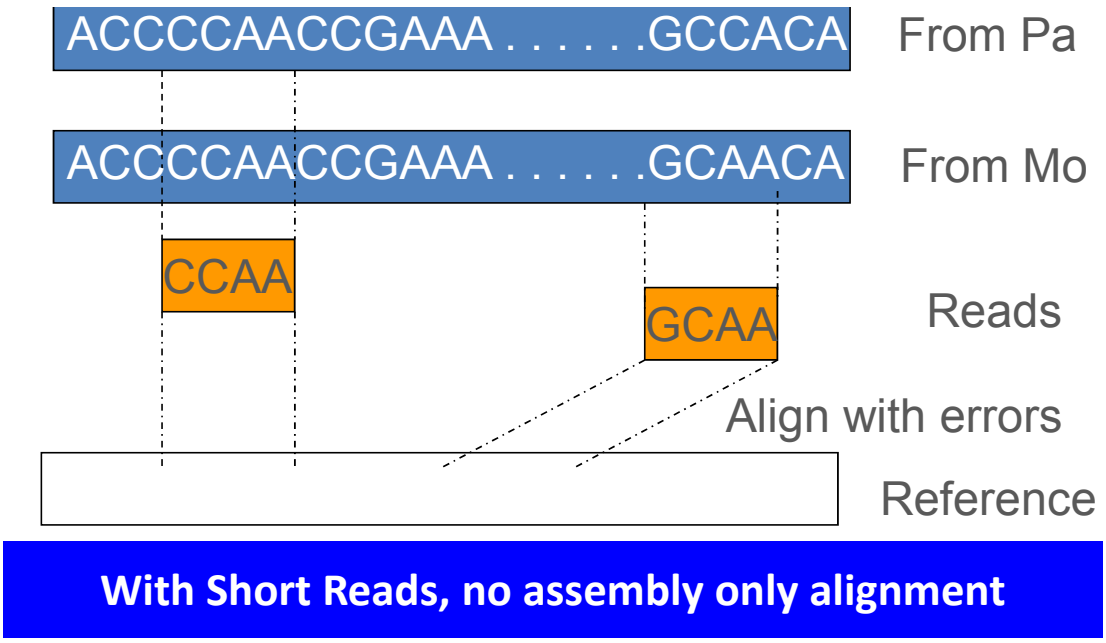


Figure 1.2. Sequencing by Short Reads

1.3.1 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 [43, 35]. 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 1.3 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 a SNV.

The probability is computed using an inference algorithm that takes into account the probability

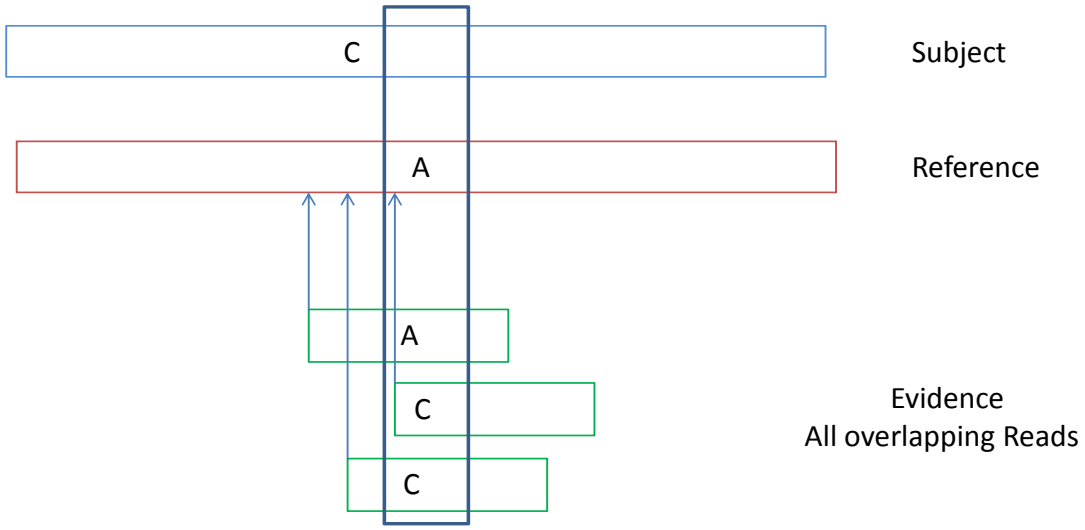


Figure 1.3. Inferring SNPs. Evidence for the probabilistic call is the deterministic subset of Reads overlapping the SNP location.

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 1.4) 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 1.4) 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.

Although there is no concensus between the various variant callers on how to infer deleted regions, it is a fact that read pairs whose mapped locations differ beyond a threshold are called *discrepant* and indicate a deletion. GASV [60] arranges overlapping discordantly mapping pair-end reads on the Cartesian plane and draws the grid of possible breakpoint locations under the assumption that the discordancy is a result of a single SV. Breakdancer [20] finds all areas that contain at least two discordant pair-end reads and it uses a Poisson model to evaluate the probability that those areas contain a SV as a function of the

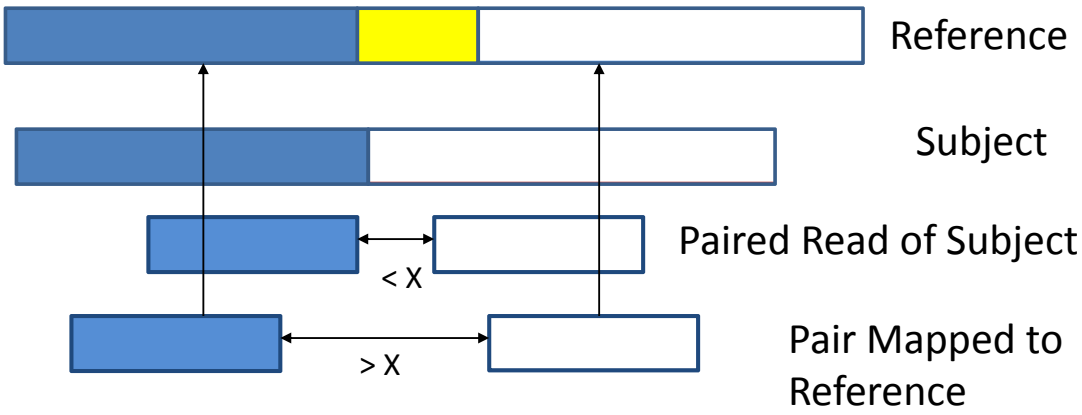


Figure 1.4. Inferring Large Scale Deletions. The Evidence is the subset of discrepant Reads that overlap the deletion.

total number of discordant reads of each of those areas. VariationHunter [34] reports that regions of SV are the ones that minimize the total number of clusters that the pair ends can form. Given the complexity of the data, and the different inference methodologies, all of these methods have significant type 1 (false-positive), and type 2 (false-negative) errors. Further, as the authors of VariationHunter [34] point out, there are a number of confounding factors for discovery. For example, repetitive regions, sequencing errors, could all lead to incorrect mappings. At the same time, incorrect calls cannot be easily detected because tools need to be modified to re-examine the source data.

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.

1.4 Layering for genomics

Our vision is inspired by analogy with systems and networks. For example, the Internet has successfully dealt with a wide variety of new link technologies (from fiber to wireless) and applications (from email to social networks) via the “hourglass” model using the key abstractions of TCP and IP (Figure 1.5a).

Similarly, 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. Such modularity requires computer systems to forgo efficiencies

that can 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 modularity is paramount.

Some initial interfaces are in vogue among geneticists today. Many instruments now produce sequence data in the "fastq" format. The output of mapping reads is often represented as "SAM/BAM" format, though other compressed formats are being proposed.[44] At a higher level, standards (such as the Variant Call Format, or VCF) are used to describe variants (See Figure 1.5a).

We propose additional layering between the mapped tools and applications. Specifically, our architecture separates 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). While Inference methods vary considerably, the Evidence for inferences is fairly standard. To gather it in a flexible and efficient manner, we propose a Genome Query Language (GQL). Though we do not address it here, a careful specification of a variation layer (Figure 1.5a)) is also important. While the data format of a variation is standardized using, say, VCF, the interface functions are not.

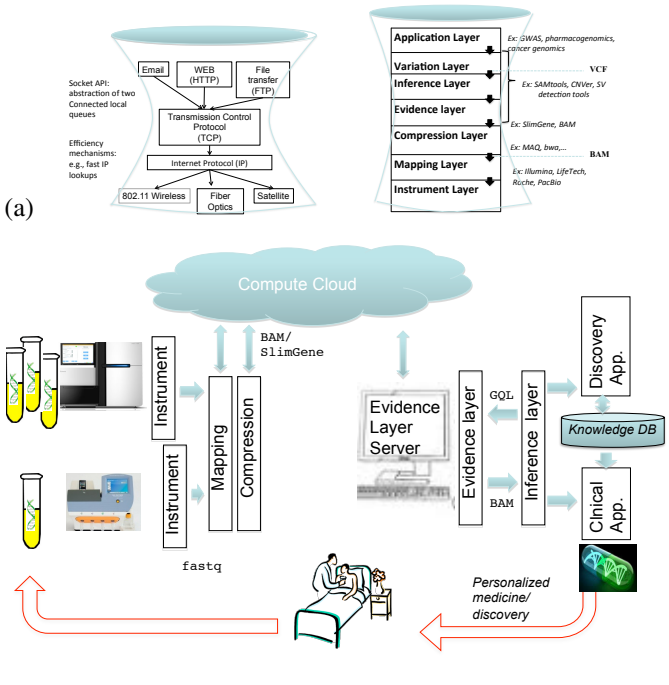


Figure 1.5. Layers of genomic processing software.

1.4.1 The case for a compression layer

Even with the limited amount of genetic information available today, many genome centers already spend millions of dollars on storage [27]. Beyond research laboratories, the fastest growing market for sequencing studies is big pharmaceutical companies [27]. Further, population studies on hundreds of thousands of individuals in the future will be extremely slow if individual disks have to be shipped to an analysis center. The *single* genome data set we use for our experiments takes 285GB in uncompressed form. At a network download rate of 10Mb/s this data set would take 63.3 hours to transfer over the Internet. In summary, reducing storage costs and improving interactivity for genomic analysis makes it imperative to look for ways to compress genomic data.

While agnostic compression schemes like Lempel-Ziv [67] can certainly be used, we ask if we can exploit the specific domain to achieve better compression. As an example, domain-specific compression schemes like MPEG-2 exploit the use of a dictionary or reference specific to the domain. Here, we exploit the fact that human genomes are similar between each other in 99% of the bases and thus the existing human assembly can be used as a reference for encoding. Thus storing 100's of Gigabytes for each human's DNA seems wasteful and will slow down access times. However, it is not as straightforward as it seems because DNA is stored as a set of fragments not a complete string; there are errors caused by the instrument and mapping software which also reduces the effective compression; finally much of the inflation comes from storing so-called quality scores with each base call and these are fundamentally hard to compress. We mostly consider loss-less compression algorithms. Specifically, given a set of genomic data S, we define a compression algorithm by a pair of functions $(\mathscr{C}, \mathscr{D})$ such that $\mathscr{D}(\mathscr{C}(S)) = S$, The compression factor *c.f.*, defined by $|S|/|\mathscr{C}(\mathscr{S})|$ describes the amount of compression achieved.

1.4.2 The case for an evidence layer

Genomes, each several hundred gigabytes long, are being produced at different locations around the world. To realize the vision outlined in Figure 1.6, individual laboratories in other parts of the world must be able to process them to reveal variations and correlate them with medical outcomes/phenotypes at each place a discovery study or personalized medicine assay is undertaken. The obvious alternatives are not workable, as described in the following paragraphs:

Downloading raw data. Transporting ~ 100 for each of ~ 1000 genomes across the network is infeasible today. Compression can mitigate ($5\times$) but not completely avoid the problem. Massive computational infrastructure must be replicated at every study location for analysis.

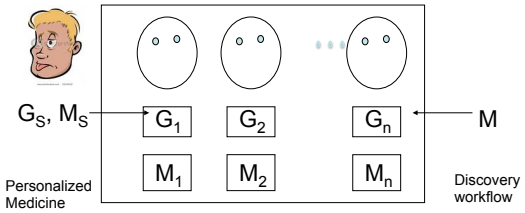


Figure 1.6. Universal sequencing, discovery, and personalized medicine. Assume every individual is sequenced at birth. In discovery, clinical geneticists logically select a subset of individuals with a specific phenotype (such as disease) and another without the phenotype, then identify genetic determinants for the phenotype. By contrast, in personalized medicine medical professionals retrieve the medical records of all patients genetically similar to a sick patient S.

Downloading variation information. Alternatively, the genomic repositories could run standard-variant-calling pipelines [25] and produce much smaller lists of variations in a standard format (such as VCF). Unfortunately, variant calling is an inexact science; researchers often want to use their own callers and almost always want to see "evidence" for specific variants. Discovery applications are thus very likely to need raw genomic evidence. By contrast, personalized genomics applications might query only called variants and a knowledgebase that correlates genotypes and phenotypes. However, even medical personnel might occasionally need to review the raw evidence for critical diagnoses.

Our approach provides a desirable compromise, allowing retrieval of evidence for variations on demand through a query language. The query server itself uses a large compute (cloud) resource and implements a query interface that can return the subset of reads (evidence) supporting specific variations. Some recent approaches have indeed hinted at such an evidence layer, including SRA and Samtools, but in a limited scenario useful mainly for SNV/SNP calling. The Genome Analysis Toolkit (http://www.broadinstitute.org/gatk/) provides a procedural framework for genome analysis with built-in support for parallelism. However, our approach -GQL- goes further, allowing declarative querying for intervals with putative structural variation (such as with discrepant reads supporting a deletion) or copying number changes. GQL also supports multiple types of inference, changing definitions of variation and pooling evidence across instrument types.

Consider this complex biological query: Identify all deletions that disrupt genes in a certain biological network and the frequency of the deletions in a natural population. For any statistical-inference algorithm, the evidence would consist of mapped reads that satisfy certain properties, including: length-discordant reads; reads with reduced depth of coverage; and reads with one end unmapped. The evidence layer supports queries to get those reads and delivers the following benefits:

Alternate forms of evidence. The separation allows inference-layer designers to start thinking of alternate forms of evidence to improve the confidence of their queries (such as split-end reads that map to the deletion breakpoints);

The cloud. The evidence layer can be 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). The evidence layer can be implemented in the cloud, while the inference layer can be implemented either in the cloud or on client workstations; and

Moving target. A standardized evidence layer gives vendors time to create a fast, scalable implementation; by contrast, the inference layer today is a moving target.

1.5 Thesis Contributions

In summary, the contributions of this thesis are the following:

- Implementation of the compression layer which works on top of aligned reads using SLIMGENE. SLIMGENE is the first tool that uses reference based compression to compress short reads and it is also the first advocator of lossy compression of quality scores. The latter influenced Illumina Inc to reduce the resolution of the quality scores in its pipelines.

- Abstraction of the evidence layer using GQL. Given the heterogeneous ways that the inference layer demands data, we propose a complete set of abstractions whose syntax resemples SQL.

- Efficient implementation of GQL. Given the sizes of the data involved we used a number of optimizations, namely meta data extraction, cached parsing, lazy joins, interval trees, stack based interval traversals to enable an efficient execution of GQL both in terms of time and space.

- Demonstration of the power of GQL through a number of experiments of genetic discovery and variation. In those experiments we ran queries either on single or multiple sequences from datasets obtained from the 1000 genomes and The Cancer Genome Atlas (TCGA) archives.

- Demonstration of the scalability of GQL through a large scale experiment with 24 processing nodes on the Windows Azure cloud platform.

1.6 Thesis organization

The rest of the thesis is organized as follows. Chapter 2 describes the SLIMGENE compression tool, Chapter 3 introduces the GQL abstraction, reviews the related work and discusses the challenges of the project, Chapter 4 provides the implementation details of GQL and demonstrates its scalability on the Azure cloud, Chapter 5 demonstrate the power of GQL through experiments of structural variation discovery nature against a number of genomes and Chapter 6 describes the conclusions of the thesis.

1.7 Acknowledgement

Chapter 1 in part, is a reprint of the material that appears in:

1. "Abstractions for Genomics", in Communications of the ACM, Bafna, Vineet; Deutsch, Alin; Heiberg, Andrew; Kozanitis, Christos; Ohno-Machado, Lucila; Varghese, George. Volume 56, Issue 1, 2013.

2. "Using the Genome Query Language (GQL) to uncover genetic variation.", Kozanitis, Christos; Heiberg, Andrew; Varghese, George; Bafna, Vineet. To appear in Bioinformatics

3. "'Making Genomics Interactive: Towards Querying a Million Genomics in a few seconds.", Kozanitis, Christos; Bafna, Vineet, Pandya, Ravi; Varghese, George. Submitted to the Symposium of Operating Systems and Principles (SOSP) 2013.

The dissertation author was the primary investigator of these papers. The dissertation author was also the primary author of papers 2 and 3.

Chapter 2

Compressing Genomes using SLIMGENE.

" Efficiency tends to deal with Things.
Effectiveness tends to deal with People.
We manage things, we lead people."
Peter F. Drucker

As the quote suggests, impressive things can happen if we manage data efficiently. In Chapter 1, we described our vision for interactive genomics and made a case for a compression layer. Clearly, compression is more fundamental than querying as compression can reduce the cost of storing and retrieving genomes, and is a substrate on which querying can be built. The rest of the chapter is organized as follows. Section 2.1 describes the nature of the data and states our contributions. Section 2.2 describes the input datasets to our experiments. Section 2.3 describes the implementation of the reference based read compression scheme. Section 2.4 describes our efforts on lossless quality values compression. Section 2.5 discusses our contribution on lossy quality value compression that influenced Illumina's pipelines. Section 2.6 gives the results of compressing genomes obtained with Illumina technology. Section 2.7 compares the efficiency of SLIMGENEwith a projection of the BAM efficiency. Finally Section 2.8 concludes the chapter.

2.1 Stucture of the data

The genomic data *S* itself can have multiple forms and depends upon the technology used. Therefore, the notion of 'loss-less' must be clarified in context. In the Illumina Genome Analyzer, each cycle produces 4 images, one for each of the nucleotides; consequently, the set *S* consists of the set of all images in all cycles. By contrast, the ABI technology maps adjacent pairs of nucleotides to a 'color-space' in the unprocessed stage. We refer to compression at this raw level as *a. Trace Compression*. The raw,

trace data is processed into base-calls creating a set of fragments (or, reads). This processing may have errors, and a quality value (typically a Phred-like score given by $-\lfloor 10\log(P_{\text{error}})\rfloor$) is used to encode the confidence in the base-call. In *b. Fragment Compression*, we define the genomic data S by the set of reads, along with quality values of each base-call. Note that the set of reads all come from the genomic sequence of an individual. In *c. Sequence Level Compression*, we define the set S simply as the diploid genome of the individual.

There has been some recent work on compressing at the sequence level [19, 21, 22, 47]. Brandon and colleagues introduce the important notion of maintaining differences against a genomic reference, and integer codes for storing offsets [19]. However, such sequence compression relies on having the fragments reconciled into a single (or diploid) sequence. While populations of entire genomes are available for mitochondria, and other microbial strains sequenced using Sanger reads, current technologies provide the data as small genomic fragments. The analysis of this data is evolving, and researchers demand access to the fragments and use proprietary methods to identify variation, not only small mutations, but also large structural variations [37, 29, 59, 53]. Further, there are several applications (e.g., identifying SNPs, structural variation) of fragment data that do not require the intermediate step of constructing a complete sequence.

Clearly, trace data are the lowest level data, and the most difficult to compress. However, it is typically accessed only by a few expert researchers (if at all), focusing on a smaller subset of fragments. Once the base-calls are made (along with quality values), the trace data is usually discarded.

For these reasons, we focus here on fragment level compression. Note that we share the common idea of compressing with respect to a reference sequence. However, our input data are a collection of potentially overlapping fragments (each, say 100 bps long) annotated with quality values. These lead to different compression needs and algorithms from [19, 22] because fragment compression must address the additional redundancy caused by high coverage and quality values. Further, the compression must efficiently encode differences due to normal variation *and* sequencing errors, for the downstream researcher.

Contribution: This chapter introduces two contributions, implemented in a tool, SLIMGENE. First, we introduce a set of domain specific loss-less compression schemes that achieve over $40\times$ compression of fragments, outperforming bzip2 by over $6\times$. Including quality values, we show a $5\times$ compression. Unoptimized versions of SLIMGENE run at comparable speeds to bzip2. Second, given the discrepancy between the compression factor obtained with and without quality values, we initiate the study of using 'lossy' quality values and investigate its effect on downstream applications. Specifically, we show that

using a lossy quality value quantization results in 14× compression but has minimal impact on SNP calls using the CASAVA software. Less than 1% of the calls are discrepant, and we show that the discrepant SNPs are so close to the threshold of detection, that no miscalls can be attributed to lossy compression. While there are dozens of downstream applications and much work needs to be done to ensure that coarsely quantized quality values will be acceptable for users, this chapter suggests this is a promising direction for investigation.

2.2 Data-sets and generic compression techniques

Generic compression techniques: Consider the data as a string over an alphabet Σ. We consider some generic techniques. First, we use a reference string so that we only need to encode the differences from the reference. As each fragment is very small, it is critical to encode the differences carefully. Second, suppose that the letters of $\sigma \in \Sigma$ are distributed according to probability $P(\sigma)$. Then, known compression schemes (Ex: Huffman codes, Arithmetic codes) encode each symbol σ using $\log_2 \frac{1}{P(\sigma)}$ bits, giving an average of $\mathscr{H}(P)$ (entropy of the distribution) bits per symbol, which is optimal for the distribution, and degrades to $\log(|\Sigma|)$ bits in the worst case.

Our goal is to devise an encoding (based on domain specific knowledge) that minimizes the entropy. In the following, we will often use this scheme, describing the suitability of the encoding by providing the entropy values. Also, while it is asymptotically perfect, the exact reduction is achievable only if the probabilities are powers of 2. Therefore, we often resort to techniques that mimic the effect of Huffman codes. Finally, if there are inherent redundancies in data, we can compress by maintaining pointers to the first occurrence of a repeated string. This is efficiently done by tools such as bzip2, and we reuse the tools.

Data formats: Many formats have been proposed for packaging and exporting genomic fragments, including the SAM/BAM format [46], and the Illumina Export format. Here, we work with the Illumina Export format, which provides a standard representation of Illumina data. It is a tab delimited format, in which every row corresponds to a read, and different columns provide qualifiers for the read. These include ReadID, CloneID, fragment sequence, and a collection of quality values. In addition, the format also encodes information obtained from aligning the read to a reference, including the chromosome strand, and position of the match. The key thing to note is that the fragment sequences, the quality values, and the match to the chromosomes represent about 80% of the data, and we will focus on compressing these. In SLIMGENE, each column is compressed independently, and the resulting data is concatenated.

2.2.1 Data Sets: experimental, and simulated

We consider a data-set of human fragments, obtained using the Illumina Genome Analyzer, and mapped to the reference (NCBI 36.1, Mar. 2006). A total of $1.1B$ reads of length 100 were mapped, representing $35\times$ base-coverage of the haploid genome. We refer to this data-set as GAHUM. The fragments differ from the reference either due to sequencing errors or genetic variation, but we refer to all changes as errors. The number of errors per fragment is distributed roughly exponentially, with a heavy tail, and a mean of 2.3 errors per fragment, as shown in Table 2.1. Because of the heavy tail, we did not attempt to fit the experimental data to a standard distribution.

Table 2.1. Distribution of the number of errors per read. The first column shows a number of errors with which a read aligns with the reference. The second column shows the probability with which an alignment occurs with the respective number of errors.

#Errors(k)	0	1	2	3	4	5	6	7	8	9	≥ 10
Pr(k errors)	0.43	0.2	0.09	0.06	0.04	0.03	0.02	0.02	0.01	0.01	0.09

Simulating coverage: While we show all compression results on GAHUM, the results could vary on other data-sets depending upon the quality of the reads, and the coverage. To examine this dependence, we simulated data-sets with different error-rates, and coverage values. We choose fragments of length 100 at random locations from Chr 20, with a read coverage given by parameter c. To simulate errors, we use a single parameter P_0 as the probability of 0 errors in the fragment. For all $k > 0$, the probability of a fragment having exactly k errors is given by $P_k = \lambda \Pr(k \text{ errors})$ from the distribution of Table 2.1. The parameter λ is adjusted to balance the distribution ($\sum_i P_i = 1$). The simulated data-set with parameters c, P_0 is denoted as GASIM(c, P_0).

2.3 Compressing fragment sequences

Consider an experiment with sequence coverage $c(\sim 30\times)$, which describes the expected number of times each nucleotide is sequenced. Each fragment is a string of characters of length $L(\simeq 100)$ over the nucleotide alphabet Σ ($\Sigma = \{A, C, G, T, N\}$). The naive requirement is $8c$ bits per nucleotide, which could be reduced to $c \log(|\Sigma|) \simeq 2.32c$ bits with a more efficient encoding. We describe an encoding based on comparison to a reference that all fragments have been mapped to.

The position vector: Assume a *Position* bit vector POS with one position for every possible location of the human genome. We set POS$[i] = 1$ if at least one fragment maps to position i (POS$[i] = 0$ otherwise).

For illustration, imagine an 8-character reference sequence, ACGTACGC, as depicted in Figure 2.1. We consider two 4bp fragments, CGTA and TACG, aligned to positions 2 and 4, respectively, with no error. Then, POS $= [0,1,0,1,0,0,0,0]$. The bit vector POS would suffice if (a) each fragment matched perfectly (no errors), (b) matches to the forward strand and (c) at most one fragment aligns to a single position (possible if $L > c$). The space needed reduces to 1 bit per nucleotide, (possibly smaller with a compression of POS), independent of coverage c.

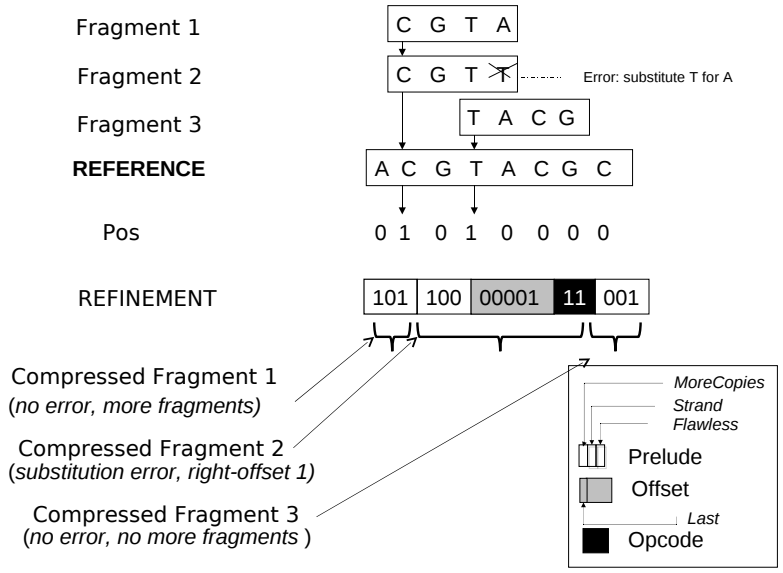


Figure 2.1. A simple proposal for fragment compression starts by mapping fragments to a reference sequence. The fragments are encoded by a Position Vector and a Refinement Vector consisting of variable size records representing each compressed fragment. The compressed fragments are encoded on a "pay as needed" basis in which more bits are used to encode fragments that map with more errors.

In reality, these assumptions are not true. For example, two Fragments 1 and 2 match at position 2, and Fragment 2 matches with a substitution (Figure 2.1). We use a *Refinement* vector that adds count and error information. However, the Refinement vector is designed on a "pay as needed basis" – in other words, fragments that align with fewer errors and fewer repeats need fewer bits to encode.

The refinement vector: The Refinement Vector is a vector of records, one for each fragment, each entry of which consists of a static *Prelude* (3 bits) and an *ErrorInstruction* record, with a variable number of bits for each error in the alignment.

The 3-bit *Prelude* consists of a *MoreCopies*, a *Strand* and a *Flawless* bit. All fragments that align with the same location of the reference genome are placed consecutively in the Refinement Vector

and their *MoreCopies* bits share the same value, while the respective bits of consecutive fragments that align to different locations differ. Thus, in a set of fragments, the *MoreCopies* bit changes value when the chromosome location varies. The *Strand* bit is set to 0 if the fragment aligns with the forward strand and 1 otherwise, while the *Flawless* bit indicates whether the fragment aligns perfectly with the reference, in which case there is no following ErrorInstruction.

When indicated by the *Flawless* bit, the *Prelude* is followed by an *ErrorInstruction*, one for every error in the alignment. The ErrorInstruction consists of an *Offset* code (# bp from the last erroneous location), followed by a variable length *Operation Code or OpCode* field describing the type of error.

Opcode: As sequencing errors are mostly nucleotide substitutions, the latter are encoded by using 2 bits, while the overhead of allocating more space to other types of error is negligible. Opcode 00 is reserved for other errors. To describe all substitutions using only 3 possibilities, we use the circular chain $A \to C \to G \to T \to A$. The opcode specifies the distance in chain between the original and substituted nucleotide. For example, an A to C substitution is encoded as 01. Insertions, deletions, and counts of N are encoded using a Huffman-like code, to get an average of $T = 3$ bits for Opcode.

Offset: Clearly, no more than $O = \log_2 L$ bits are needed to encode the offset. To improve upon the $\log_2(100) \simeq 7$ bits per error, note that the quality of base calling worsens in the later cycles of a run. Therefore, we use a *back-to-front* error ordering to exploit this fact, and a Huffman-like code to decrease O.

The record for Fragment 2 (CGTT, Figure 2.1) provides an example for the error encoding, with a prelude equal to 100 (last fragment that maps to this location and error instructions follow) followed by a single ErrorInstruction. The next 5 bits (00001) indicate the relative offset of the error from the *end* of the fragment. The first bit of the offset is a "Last" bit that indicates that there are no more errors. The offset field is followed by an opcode (11) which describes a substitution of T for A, a circular shift of 3. Further improvement is possible.

Compact Offset encoding: We now describe a method of encoding errors that exploits the locality of errors to create a more efficient encoding than if errors were formally distributed. Let $\mathscr{E}$ denote the expected number of errors per fragment, implying an offset of $\frac{L}{\mathscr{E}}$bp. Instead, we use a single bitmap, ERROR, to mark the positions of all errors from all fragments. Second, we specify the error location for a given fragment as the number of bits we need to skip in ERROR from the start offset of the fragment to reach the error. We expect to see a '1' after $\max\{1, \frac{L}{c\mathscr{E}}\}$ bits in ERROR. Thus, instead of encoding the error offset as

$\frac{L}{\mathscr{E}}$ bp, we encode it as the count using

$$O = \log_2 \frac{L/\mathscr{E}}{\max\{1, \frac{L}{c\mathscr{E}}\}} = \min\{\log_2 \frac{L}{\mathscr{E}}, \log_2 c\}$$

bits. For smaller coverage $c < \frac{L}{\mathscr{E}}$, we can gain a few bits in computing O. Overall, the back-to-front ordering, and compact offset encoding leads to $O \simeq 4$ bits.

Compression analysis: Here, we do a back-of-the-envelope calculation of compression gains, mainly to understand bottlenecks. Compression results on real data, and simulations will be shown in Section 2.3.2. To encode *Refinement*, each fragment contributes a *Prelude* (3 bits), followed by a collection of *Opcodes* (T bits each), and *Offsets* (O bits each). Let $\mathscr{E}$ denote the expected number of errors per fragment, implying a refinement vector length of

$$3 + \mathscr{E} \cdot (T + O)$$

per fragment. Also, encoding POS, and ERROR requires 1 bit each, per nucleotide of the reference. The total number of bits needed per nucleotide of the reference is given by

$$2 + \frac{c}{L} \cdot (3 + \mathscr{E} \cdot (O + T)) \tag{2.1}$$

Substituting $T = 3, O = 4, L = 100$, we have

$$\text{c.f.} = \frac{8c}{2 + \frac{c}{L} \cdot (3 + 7 \cdot \mathscr{E})} \tag{2.2}$$

Equation 2.2 provides a basic calculation of the impact of error-rate and coverage on compressibility using SLIMGENE. For GAHUM, $\mathscr{E} = 2.3$ (Section 2.2.1). For high coverages, the c.f. is $\simeq 8/0.19 \simeq 42$. For lower coverages, the fixed costs are more important, but the POS and ERROR bitmaps are very sparse and can be compressed well, by (say) bzip2.

2.3.1 Encoding Offsets and Opcodes

The reader may wonder how our seemingly ad hoc encoding technique compares to information theoretic bounds. We first did an experiment to evaluate the effectiveness of OpCode assignment. We tabulated the probability of each type of error (all possible substitutions, deletions, and insertions) on our data set and used these probabilities to compute the expected OpCode length using our encoding scheme. We found that the expected OpCode length using our encoding was 2.2 which compares favorably with the

entropy which was 1.97.

Opcodes: Table 2.2 summarizes the probability with which each type of error appears while aligning fragments against GAHUM. Note that we can encode the 12 possible substitutions using 3 symbols by using the reference. Insertions and deletions create an additional 5 symbols. To encode runs of Ns, we use distinct symbols up to 10, and a single symbol for 10 or more Ns. Any true encoding must additionally store the number of N's. Therefore the entropy of distribution implied by Table 2.2 is a lower bound on the number of bits needed for Opcodes.

In our implementation, 2 bits encode the three substitution cases between nucleotide letters A, C, G and T, while 5 bits encode all four cases of insertion, deletions and single substitution between a nucleotide and a N. We use 12 bits to encode runs of N's. Empirically, the number of bits needed for the Opcode is computed to be 2.2, which is close to the entropy of 1.97.

Offsets: The width of the error location O depends on the number of bits that we need to skip in ERROR to reach the error location for a given fragment. According to the histogram of Figure 2.2 which has been obtained from the error distribution of chromosome 20 of GAHUM, the majority of cases involves the skipping of up to 10 bits in ERROR. Indeed, the entropy of the distribution of Figure 2.2 is 3.69. Thus, the allocation of 3 or 4 bits for the offset encoding is compatible with the theoretical limits.

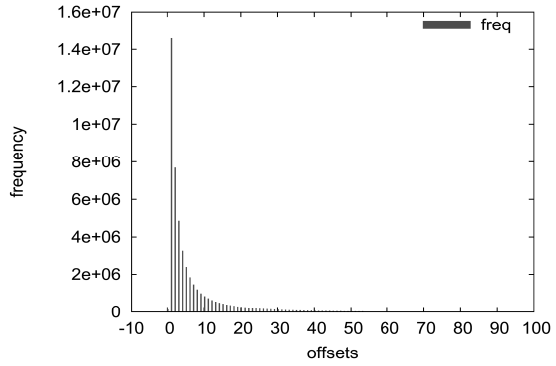


Figure 2.2. Distribution of offsets to the next ERROR bit. The data has been obtained from the error distribution of chromosome 20 of GAHUM and represent the number of bits that we need to skip in ERROR to reach the error location for a given fragment. The entropy of the distribution is 3.69 which agrees with the observation that the majority of cases involves the skipping of up to 10 bits in that vector.

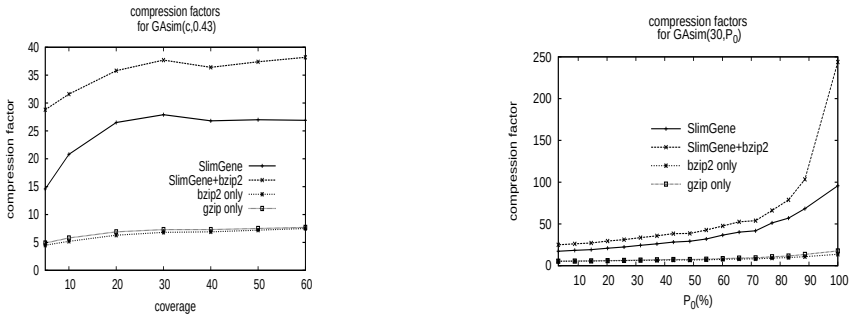
Table 2.2. The probabilities of alignment errors. The probabilities are empirically estimated by aligning Chromosome 1 fragments from GAHUM against the human reference (hg18). We encode the 12 possible substitutions using 3 symbols by using the reference. Insertions and deletions create an additional 5 symbols. To encode runs of Ns, we use distinct symbols up to 10. Here, use a single symbol for 10 or more Ns to get a lower bound on the entropy of the distribution.

Operation		Probability
Substitute	$A \to C$ $C \to G$ $G \to T$ $T \to A$	0.31
Substitute	$A \to T$ $C \to T$ $G \to A$ $T \to C$	0.31
Substitute	$A \to G$ $C \to A$ $G \to C$ $T \to G$	0.31
Insert	1 N	0.056
Insert	2 N's	0.0041
Insert	3 N's	0.0016
Insert	4 N's	0.00069
Insert	5 N's	0.00038
Insert	6 N's	0.00041
Insert	7 N's	0.00030
Insert	8 N's	0.00027
Insert	9 N's	0.00056
Insert	$\geq$ 10 N's	0.0019
Insert	A	0.001225
Insert	C	0.001225
Insert	G	0.001225
Insert	T	0.001225
Delete		0.0049

2.3.2 Experimental results on GASIM(c, P_0)

We tested SLIMGENE on GASIM(c, P_0) to investigate the effect of coverage and errors on compressibility. Recall that for GAHUM, $P_0 = 0.43$, $c = 30$, $\mathscr{E} = 2.3$. As P_0 is varied, $\mathscr{E}$ is approximately $\approx \frac{2.3}{1-0.43} \cdot (1-P_0) \simeq 4(1-P_0)$.

In Figure 2.3a, we fix $P_0 = 0.43$, and test compressibility of GASIM$(c, 0.43)$. As suggested by Eq. 2.2, the compressibility of SLIMGENE stabilizes once the coverage is sufficient. Also, using SLIMGENE+bzip2, the compressibility for lower coverage is enhanced due to better compression of POS



(a) (b)

Figure 2.3. Compressibility of GASIM(c,P_0). (a) The compression factors achieved with change in coverage. c.f. is lower for lower coverage due to the fixed costs of POS, and ERROR, and stabilizes subsequently. (b) Compressibility as a function of errors. With high values of P_0 (low error), up to 2 orders of magnitude compression is possible. Note that the values of P_0 are multiplied by 100.

and ERROR. Figure 2.3b explores the dependency on the error rates using GASIM$(30,P_0)$. Again, the experimental results follow the calculations in Eq. 2.2, which can be rewritten as

$$\frac{8\cdot 30}{2+0.1\cdot 30\cdot 4(1-P_0)} \simeq \frac{20}{1-P_0}$$

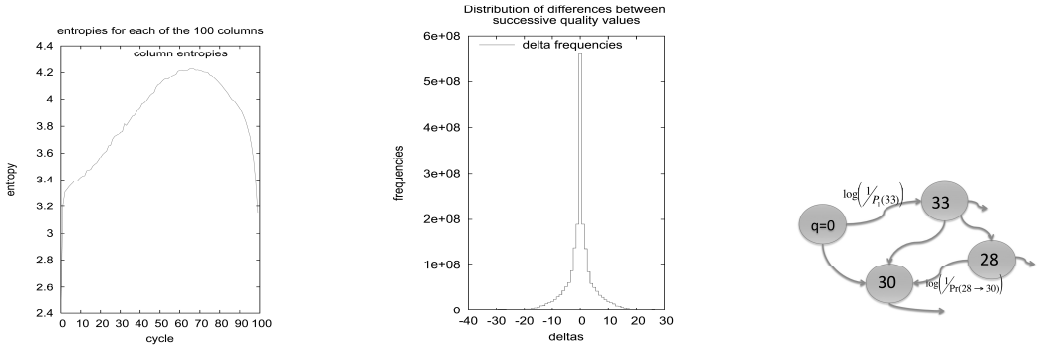
At high values of P_0, SLIMGENE produces 2 orders of magnitude compression. However, it outperforms bzip2 and gzip even for lower values of P_0.

2.4 Compressing Quality Values

For the Genome analyzer, as well as other technologies, the Quality values are often described as $\approx -\log(P_{\text{err}})$. Specifically, the Phred score is given to be $\lfloor -10\cdot\log(P_{\text{err}})\rfloor$. The default encoding for GAHUM require 8 bits to encode each Q-value. We started by testing empirically if there was a non-uniform distribution on these values (see Section 2.2). The entropy of the Q-values is 4.01. A bzip2 compression of the data-set resulted in 3.8 bits per Q-value. For further compression, we need to use some characteristics of common sequencers.

Position dependent quality: Base calling is highly accurate in the early cycles, while it gradually loses its accuracy in subsequent cycles. Thus, earlier cycles are populated by higher quality values and later cycles by lower values. To exploit this, consider a matrix in which each row corresponds to the Q-values of a single read in order. Each column therefore corresponds (approximately) to the Q-values of all reads in a

single cycle. In Figure 2.4a, we plot the entropy of Q-value distribution at each columns. Not surprisingly, the entropy is low at the beginning (all values are high), and at the end (all values are low), but increases in the middle, with an average entropy of 3.85.



(a) (b) (c)

Figure 2.4. Distribution of quality, and Δ values, and Markov encoding. (a) A distribution of Q-values at each position. The entropy is low at the beginning (all values are high), and at the end (all values are low), but increases in the middle. (b) A histogram of Δ-values. (c) Markov encoding: Each string of Q-values is described by a unique path in the automaton starting from $q = 0$, and is encoded by concatenating the code for each transition. Huffman encoding bounds the number of required bits by the entropy of the distribution. Edge labels describe the required number of bits for each transition.

Encoding Δ values: The gradual degradation of the Q-values leads to the observation that Q-values that belong to neighboring positions differ slightly. Thus, if instead of encoding the quality values, one encodes their differences between adjacent values (Δ), it is expected that such a representation would be populated by smaller differences. For instance, Figure 2.4b shows a histogram of the distribution of Δ-values. However, the entropy of the distribution is 4.26 bits per Δ-value.

Markov encoding: We can combine the two ideas above by noting that the Δ-values also have a Markovian property. As a simple example, assume that all Q-values from 2 to 41 are equally abundant in the empirical data. Then, a straightforward encoding would need $\lceil \log_2(41-2+1) \rceil = 6$ bits. However, suppose when we are at quality value (say) 34 (Figure 2.4c), the next quality value is always one of $33, 32, 31, 30$. Therefore, instead of encoding Q' using 6 bits, we can encode it using 2 bits, conditioning on the previous Q-value of 34.

We formalize this using a Markov model. Consider an automaton M in which there is a distinct node q for each quality value, and an additional start state $q = 0$. To start with, there is a transition from 0 to q with probability $P_1(q)$. In each subsequent step, M transitions from q to q' with probability $\Pr(q \to q')$.

Using an empirical data-set D of quality values, we can estimate the transition probabilities as

$$\Pr(q \to q') = \begin{cases} 0 & (* \text{ if } q' = 0 *) \\ \text{fraction of reads with initial quality } q' & (* \text{ if } q = 0 *) \\ \frac{\#\text{pairs } (q,q') \text{ in } D}{\#\text{occurrences of } q \text{ in } D} & (* \text{ otherwise } *) \end{cases} \tag{2.3}$$

Assuming a fixed length L for fragments, the Entropy of the Markov distribution is given by

$$\mathscr{H}(M) = \frac{1}{L}\mathscr{H}(P_1) + \frac{L-1}{L}\sum_{q,q' \neq 0} \Pr(q \to q') \log\left(\frac{1}{\Pr(q \to q')}\right) \tag{2.4}$$

Empirical calculations show the entropy to be 3.3 bits. To match this, we use a custom encoding scheme (denoted as *Markov-encoding*) in which every transition $q \to q'$ is encoded using a Huffman code of $-\log(\Pr(q \to q'))$ bits. Table 2.3 summarizes the results of Q-value compression. The Markov encoding scheme provides a 2.32× compression, requiring 3.45 bits per character. Further compression using bzip2 does not improve on this.

Table 2.3. Quality value compression results. In the uncompressed form a quality value required 8 bits for its representation. Using bzip2 and delta encoding it needs 3.8 and 4.25 bits respectively. Finally, Markov encoding assigns 3.45 bits per quality value which results in a 2.32× compression.

	Raw File	bzip2	Δ (Huffman)	Markov (Huffman)
Bits per character	8	3.8	4.25	3.45
c.f.	1	2.11	1.88	2.32

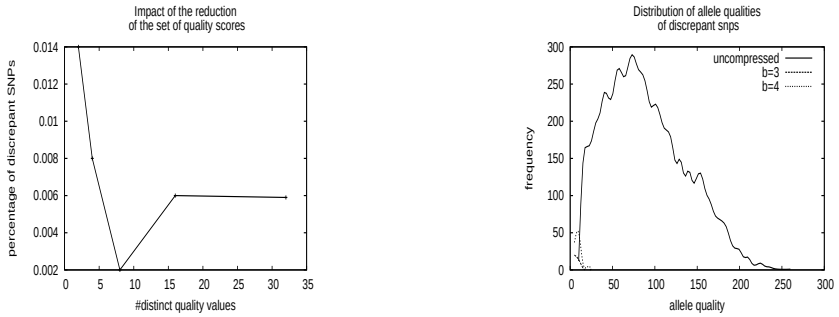
2.5 Lossy compression of Quality Values

Certainly, further compression of Quality values remains an intriguing research question. However, even with increasing sophistication, it is likely that Q-value compression will be the bottleneck in fragment-level compression. So we ask the sacrilegious question: *can the quality values be discarded*? Possibly in the future, base-calling will improve to the point that Q-values become largely irrelevant. Unfortunately, the answer today seems to be 'no'. Many downstream applications such as variant calling, and many others consider Q-values as a critical part of inference, and indeed, would not accept fragment data without Q-values. Here, we ask a different question: *is the downstream application robust to small changes in Q-values*? If so, a 'lossy encoding' could be immaterial to the downstream application.

Denote the number of distinct quality values produced as $|Q| = Q_{\max} - Q_{\min}$, which is encoded using $\log_2(|Q|)$ bits. Note that a Q-score computation such as $\lfloor -10 \cdot \log_2(P_{\text{err}}) \rfloor$ already involves a loss of precision. The error here can be reduced by rounding.

We test the impact of the lossy scheme on Illumina's downstream application called CASAVA (see http://www.illumina.com/pages.ilmn?ID=314) that calls alleles based on fragments and associated Q-scores. CASAVA was run over a 50M wide portion of the Chr 2 of GAHUM using the original Q-values, and it returned a set S of $|S| = 17,021$ variants that matched an internal threshold (the allele quality must exceed 10; in heterozygous cases the respective threshold for the weak allele is 6). For each choice of parameter $b \in \{1,\ldots,5\}$, we reran CASAVA after replacing the original score Q with $Q_{\min} + |Q| \cdot \text{LQ-score}_b(Q)$. Denote each of the resulting variant sets as S_b. A variant $s \in S\hat{\text{S}}_b$ is concordant. It is considered *discrepant* if $s \in (S \setminus S_b) \cup (S_b \setminus S)$.

The results in Figure 2.5 are surprising. Even with $b = 1$ (using 1 bit to encode Q values), 98.6% of the variant calls in S are concordant. This improves to 99.4% using $b = 3$. Moreover, we observe (in Fig. 2.5b) that the discrepant SNPs are close to the threshold. Specifically, 85% of the discrepant SNPs have allele qualities ≤ 10.



(a) (b)

Figure 2.5. Impact of Lossy Compression on CASAVA. CASAVA was run on a 50M wide region of Chr 2 of GAHUM using lossless and lossy compression schemes. The y-axis plots the fraction of discrepant SNPs as a function of lossy compression. The x-axis shows the number of bits used to encode Q-scores. (b) The allele quality distribution of all lossless SNPs and the discrepant SNPs for 3 and 4-bit quantization. The plot indicates that the discrepant variants are close to the threshold of detection (allele quality of 6 for weak alleles in the heterozygous case, 10 for the homozygous case).

2.5.1 Is the loss-less scheme always better?

We consider the 38 positions in Chr 2 where the lossy (3-bits) compression is discrepant from the loss-less case. On the face of it, this implies a 0.2% error in SNP calling, clearly unacceptable when scaled to the size of the human genome. However, this assumes that the loss-less call is always correct. We show that this is clearly not true by comparing the SNP calls based on lossy and loss-less data in these 38 positions with the corresponding entries, if any, in dbSNP (version 29). We show that most discrepancies come from marginal decisions between homozygote and heterozygote calls.

For simplicity, we only describe the $26/38$ SNPs with single nucleotide substitution in dbSNP. In all discrepant cases, the coverage is no more than 5 reads (despite the fact that the mean coverage is $30\times$). Further, in all but 2 cases, both lossy, and lossless agree with dbSNP, and the main discrepancy is in calling heterozygote versus homozygotes. Specifically, lossy calls 14/10 homozygotes and heterozygotes, against lossless (12/12). With coverage ≤ 5, the distinction between homozygote and heterozygotes is hard to make. Close to 50% of the differences were due to consideration of extra alleles due to lossy compression, while in the remaining, alleles are discarded. Given those numbers, it is totally unclear that our lossy compression scheme yields *worse* results than the lossless set, not to mention that in some cases it can lead to better results.

We next consider the two positions where the discrepant SNPs produced by the lossy scheme completely disagree with the dbSNP call. Table 2.4 shows that at position 43150343 dbSNP reports C/T. The loss-less Q-values and allele calls were respectively $39G, 28G, 20G, 30G$; CASAVA did not make a call. On the other hand, the lossy reconstruction led to values $41G, 27G, 22G, 32G$, which pushed the overall allele quality marginally over the threshold, and led to the CASAVA call of 'G'. In this case, the lossy reconstruction is quite reasonable, and there is no way to conclude that an error was made. The second discrepant case tells an identical story.

Given the inherent errors in SNP calling (lossy *or* lossless), we suggest that the applications of these SNP calls are inherently robust to errors. The downstream applications are usually one of two types. In the first case, the genotype of the individual is important in determining correlations with a phenotype. In such cases, small coverage of an important SNP must always be validated by targeted sequencing. In the second case, the SNP calls are used to determine allelic frequencies and SNP discovery in a population. In such cases, marginally increasing the population size will correct errors in individual SNP calls (especially ones due to low coverage). Our results suggest that we can tremendously reduce storage while not impacting downstream applications by coarsely quantizing quality values.

Table 2.4. Analysis of discrepant alleles. In both cases the lossy quality values result in a score which marginally exceeds the threshold of 10 used to call the allele. Thus CASAVA would make a SNP call only in the lossy scheme of both cases.

position	dbSNP entry	scheme	Qvalues				allele quality	Decision
43150343	C/T	lossy-8	41G	27G	22G	32G	10.2	G
		lossless	39G	28G	20G	30G	9.9	-
46014280	A/G	lossy-8	27C	37C	37C		10.1	C
		lossless	27C	36C	36C		9.9	-

2.6 Putting it all together: compression results

We used SLIMGENE to compress the GAHUM data-set with $1.1B$ reads, which are sorted according to the alignment location and their total size is 285GB. We focus on the columns containing the reads, their chromosome locations and match descriptors (124.7GB), and the column containing Q-values (103.4GB), for a total size of 228.1GB. The results are presented in Table 2.5 and show a $40\times$ compression of fragments. Using a lossy 1-bit encoding of Q-values results in a net compression of $14\times$ ($8\times$ with a 3-bit encoding).

Table 2.5. Compression of GAHUM using SLIMGENE. Using a loss-less Q-value compression, we reduce the size by $5\times$. A lossy Q-value quantization results in a further $3\times$ compression, with minimal effect on downstream applications.

	fragments+ alignment(GB)	Q-values (GB)	total (GB)	execution time (hr)
Uncompressed	124.7	103.4	228.1	N/A
gzip (in isolation)	15.83	49.92	65.75	N/A
bzip2 (in isolation)	17.9	46.49	64.39	10.79
SLIMGENE	3.2	42.23	45.43	7.38
SLIMGENE+bzip2	3.04	42.34	45.38	7.38
SLIMGENE+lossy Q-values($b=3$)	3.2	26	29.8	7.38
SLIMGENE+lossy Q-values($b=1$)	3.2	13.5	16.7	7.38

2.7 Comparison with SAMtools

Table 2.6 compares the performance of the SLIMGENE with SAMtools. The input of SLIMGENE is the set of export files that comprise the GAHUM while the SAMtools compress the GAHUM files after their conversion into the SAM format. Note that not only does the input dataset include the columns of Table 2.5, but also read names, alignment scores, mate pointers and unmapped reads which are compressed by SLIMGENEusing gzip. As we can see, SLIMGENE provides higher compression rates in a lower execution time.

Table 2.6. Compression of GAHUM using SLIMGENE and SAMtools. The size of the entire uncompressed dataset is 294 GB. SLIMGENE compresses GAHUM in 7.39 hours for a compression factor of 5.15×. On the other hand SAMtools provide a compression factor of 3.28× in 10.79 hours. In the absence of quality values (all bases have an identical quality), the advantage of SLIMGENE is magnified.

	Quality Values		*No Quality Values*		
	Size (GB)	c.f	Size (GB)	c.f	Time(hr)
Uncompressed	294	1	294	1	
SLIMGENE	57.0	5.16	16.2	18.15	7.39
SAMtools	89.5	3.28	32.8	8.96	10.79

In the absence of the quality scores, SLIMGENE would have been achieved higher compression rates as compared to SAMtools. Although SAMtools cannot compress SAM files without quality scores, we substituted all quality scores of GAHUM with the same string. In this way we make sure that any general purpose tool can recognize the repetition and minimize the impact of the size of the quality scores on the compressed files. As Table 2.6 shows, we get an 18× compression with SLIMGENE versus 9× for SAMtools.

2.8 Discussion

The SLIMGENE toolkit described here is available on request from the authors. While we have obtained compression factors of 35 or more for fragment compression, we believe we could do somewhat better and get closer to information theoretic limits. Currently, error-encoding is the bottleneck, and we do not distinguish between sequencing errors and genetic variation. By storing multiple (even synthetic) references, common genetic variation can be captured by exact matches instead of error-encoding. To do this, we only have to increase the POS vector while greatly reducing the number of ErrorInstructions. This trade-off between extra storage at the compressor/decompressor versus reduced transmission can be explored further.

While this chapter has focused on fragment compression as opposed to sequence compression (Brandon et al. [19]), we believe both forms of compression are important, and in fact, complementary. In the *future*, if individuals have complete diploid genome sequences available as part of their personal health records, the focus will shift to sequence-level compression. It seems likely that fragment level compression will continue to be important to advance knowledge of human genetic variation, and is the pressing problem faced by researchers *today*. We note that Brandon et al. [19] also mention fragment compression briefly, but describe no techniques.

While we have shown 2-3$\times$ compression of quality values, we believe it is unlikely this can be improved further. It is barely conceivable that unsuspected relations exist, which allow us to predict Q-values at some positions using Q-values from other positions; this can then be exploited for additional compression. However, there is nothing in the physics of the sequencing process that suggests such complicated correlations exist. Further, it would be computationally hard to search for such relations.

If compressing quality values beyond 3$\times$ is indeed infeasible, then lossy compression is the only alternative for order of magnitude reductions. Our results suggest that the loss is not significant for interpretation. However, we have only scratched the surface. Using *companding* (from Pulse Code Modulation [16]), we plan to deviate from uniform quantization, and focus on wider quantization spacings for the middle quality values and smaller spacing for very high and very low quantization values. Further, we need to investigate the effect of quantization on other analysis programs for say *de novo* assembly, structural variation, and CNV detection. The number of quantization values in SLIMGENE is parameterized, and so different application programs can choose the level of quantization for their needs. A more intriguing idea is to use multi-level encoding as has been suggested for video [64]; initially, coarsely quantized quality values are transmitted, and the analysis program only requests finely quantized values if needed.

As sequencing of individuals becomes commoditized, its production will shift from large sequencing centers to small, distributed laboratories. Further, analysis is also likely be distributed among specialists who focus on specific aspects of human biology. Our thesis initiates a study of fragment compression, both loss-less and lossy, which should reduce the effort of distributing and synthesizing this vast genomic resource.

2.9 Acknowledgement

Chapter 2, in full, is a reprint of the material as it appears in "Compressing Genomic Sequencing Fragments Using SLIMGENE" in *Journal of Computational Biology*. Kozanitis, Christos; Saunders, Chris; Kruglyak, Semyon; Bafna, Vineet; Varghese, George. Vol 18, Issue 3, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 3

The Genome Query Language

"An abstraction is one thing that
represents several real things equally well."
Edsger W. Dijkstra

Chapter 2 described the implementation of the compression layer of the genomic stack. This chapter introduces the Genome Query Language (GQL) which is our proposed set of abstractions for the evidence layer. Section 3.1 provides the GQL specification. Section 3.2 provides the challenges that an implementation of the evidence layer faces. Section 3.3 reviews the related work and Section 3.4 concludes the chapter.

3.1 GQL Specification

We wanted to develop a query language that is complete (capable of handling all evidence level queries), efficient, and easy to express, and that uses standard input/output.Ideally, the language would allow selecting from a set of reads we call READS and output a subset of reads in a standard format, like BAM. GQL uses a standard SQL like syntax that is easy to use and familiar to most programmers. However, a standard relational database does not work well.

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).

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 SELECTstolen 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 3.1 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.

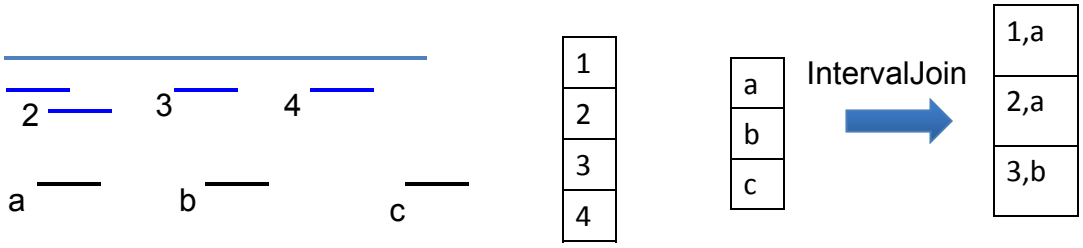


Figure 3.1. *IntervalJoin* based on interval intersection. For example, $(1,a)$ is in the output because intervals 1 and a intersect

While *IntervalJoin* appears in temporal [62] and spatial databases [4], 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.

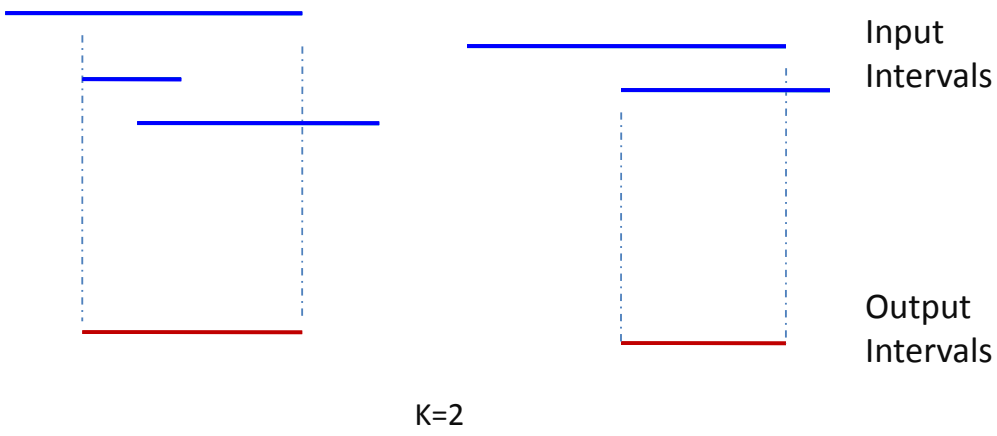


Figure 3.2. In the simplest case, Merging Intervals corresponds to find the union of a number of intervals. More generally, it outputs the largest disjoint intervals such that at least k of the input intervals are contained in each output interval

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 performance; building GQL allowed us control of the internals which enabled aggressive optimization (Section 4.2).

3.1.1 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 Section A.3.

3.1.2 Sample Queries

Here, we discuss several examples to demonstrate the expressive power of that algebra. In Chapter 5, we show GQL's expressive power captures in a series of real scenarios of biological discovery and/or validation.

What is the genotype at a specific position (such as SNV)?

Query: Define an interval by the triple ⟨'chr','beg','end'⟩ signifying the beginning and ending coordinates on a specific chromosome. Let the SNV of interest be located at a point interval A (⟨'chr','beg','end'⟩). The evidence for the genotype is provided by alignments of reads that map to the location; we can either query for the mapped reads, or for the alignments themselves, which are often stored as a mapped read attribute (such as R.ALIGNSTR). Thus

GQL: SELECT R.ID, R.ALIGNSTR

FROM *IntervalJoin* R,A

What are the diploid haplotypes across a set of linked loci in a dataset?

Query: This is more challenging than the first. Assembling haplotypes requiers a collection of reads each (perhaps along with their paired-end reads) connecting at least two polymorphic sites. Let attribute R.CloneId denote the clone identifier so that the paired-end reads r_1, r_2 derived from the same clone satisfy r_1.CloneId $= r_2$.CloneId. Also, let relation S denote the collection of point intervals, one for each variant locus.

(a) Find a subset of READS mapping to the loci and the count of sites the reads or their paired-ends map to (call this count c):

GQL: $RC = \text{SELECT } R.CloneId, c = count(*)$

$\text{FROM } IntervalJoinR, S$

$\text{GROUPBY } R.CloneID$

(b) Return IDs of reads with count ≥ 2:

GQL: $\text{SELECT } R.ID$

$\text{FROM } R, RC$

$\text{WHERE } R.CloneID = RC.CloneID$

$\text{AND } (RC.c \geq 2)$

What genomic loci are affected by Copy Number Variations (CNVs)?

Query: If the number of donor reads mapping to a region exceeds some threshold T then the inference might be that the region has been duplicated in the donor genome. Such CNVs have been implicated as an important variation for many disease phenotypes. To gather evidence, a geneticist would like to identify all intervals where the number of mapped reads exceeds, say, threshold t. Let $G.loc$ denote a specific chromosome and location.

(a) Compute for each location, the number of reads that map to the location:

GQL: $V = \text{SELECT } G.loc, c = \text{COUNT}(*)$

$\text{FROM } IntervalJoin\ R, G$

$\text{GROUPBY } G.loc$

(b) Return all "merged regions" where the read count exceeds threshold t.

GQL: *MergeIntervals* RS.loc

$\text{FROM } V$

$\text{WHERE } V.c > t$

Identify all regions in the donor genome with large deletions.

Query: As discussed earlier, the evidence for deletion comes from a variety of sources. Suppose a user prefers discrepant paired-end mapping. Paired-end reads from clones of, say, length 500 should map $\simeq$ 500bp apart on the reference genome. If, instead, the ends happen to map *discrepantly far* (such as, ℓ apart for some $\ell >> 500$, like $\ell \simeq 10000$), they support the case for a deletion in the donor genome. The goal is thus to identify all regions with at least t discrepant paired-end reads:

(a) Use a join in which each record contains the mapping locations of the read as well as its paired-end.

GQL: READS already contains this join.

(b) Select records containing discrepant reads.

GQL: $H_2 =$ SELECT * FROM READS

WHERE $abs(loc - mateloc) > 10000$

(c) Select intervals containing at least t discrepant reads.

GQL: *MergeIntervals* G.loc FROM H_2

GROUPBY G.loc, c=count(*)

WHERE $c > t$

3.1.3 Population based queries

The true power of querying genomes comes from the ability to query populations. Indeed, existing tools (such as Samtools) support the ability to extract reads from multiple individuals at specific locations corresponding to polymorphic sites. GQL extends the full power of genomic queries to interrogate populations. In the Warfarin example, the goal was to query for Warfarin dosage and genetic variation in candidate genes (identified through a discovery work flow) among individuals on the Warfarin regimen. This suggests the following query: "Report Warfarin dosage and genomic intervals and reads in individuals such that the copy number of mapped reads is at least twice the expected coverage in the interval."

The query for this would be similar to that of a single individual, but repeated via a "join" with a population P, using

GQL: SELECT * FROM P,*IntervalJoin*R,E

WHERE P.WD = TRUE

Using earlier arguments, GQL can be used to count the read depth, and report high-CNV regions. A similar idea applies to a *personalized workflow*, where a geneticist might be interested in patients with copy numbers similar to the specific query individual.

Group Inference without Accurate Individual Inference: The ability to query populations has an important benefit: Individual genomes may have little evidence for SNV calls at low-coverage sequencing. However, if a large number of affected individuals in a population (such as 800 out of 1,000) all show the same SNV while controls do not, an inference tool can reliably predict an association, even with unreliable calls for individuals. While more work is required to demonstrate the benefits of group inference, the point is that GQL provides query support for group inference.

3.2 Challenges

We have made the case for a set of genomic layers, including an evidence layer where evidence is retrieved through GQL. Successful implementation of this vision depends on new ideas from computer science:

3.2.1 Query Power (Database Theory)

Is GQL sufficiently powerful to address all evidence-layer queries needed in practice? The goal is to have the evidence layer handle as much data-intensive computation as possible while preserving performance; without the performance goal, any query can be trivially satisfied by passing the entire genome to the inference layer. Note that GQLs expressive power coincides with that of first-order logic over the schema of the three relations R, G, P, a signature of aggregation functions, and a group-by operator. However, user feedback may require GQL developers to add extensions. In implementing extensions, care must be taken to balance expressive power with efficient evaluation.

3.2.2 Query Speed (Database Systems)

We designed a corresponding algebra GQA as an internal representation for optimization and evaluating query plans; for example, assume queries on populations are automatically decomposed into queries on individuals. Consider queries of the general form SELECT a FROM $IntervalJoin\, R, G$ WHERE b. The two steps are needed to construct such a query:

1. Select for relations that satisfy constraints b; and
2. Project (while removing duplicates) on to attributes a.

GQL uses a location based index LTOR, where $\text{LTOR}(\ell)$ is a pointer to first read that maps to the point-interval l. For each individual, GQL keeps a compressed index of the mapped reads in memory. The index can be used for select operations based on specific locations (such as reads that map to specific genes).

However many queries involve scanning the entire genome for maximal intervals; for example, find all maximal regions where there is a disproportionate increase in the number of mapped reads (High copy number). For efficient implementation of these queries, GQL constructs special indices that allow filtering for READS according to a user-defined constraint. Define a *strength vector* S_θ for a constraint θ as a vector of length G (the entire genome). For any location $\ell \in G$, $S_\theta[\ell]$ gives the strength of the evidence

at that location, and can be precomputed for common constraints θ. To reduce memory, GQL also chooses a minimum strength cut-off, and maintains $C_{\theta,t}$ as a sorted sequence of intervals $i_1, i_2, \ldots$ such that each i_j is a maximal interval satisfying $S_\theta[\ell] \geq t$ for all $\ell \in i_j$. The compressed vectors reduce memory and computation requirements and can be recomputed on demand.

3.2.3 EMRs (information retrieval)

The phenotypes associated with each sequenced individual are already in patient medical records. Initial results from the eMERGE network indicate that, for a limited set of diseases, EMRs can be used for phenotype characterization in genome-wide association studies within a reasonable margin of error.[40, 54] We anticipate that most health-care institutions will be using EMRs by 2014, given incentives provided by the Health Information Technology for Economic and Clinical Health Act of 2009.[63] Increasing adherence to interoperability standards [26] and advances in biomedical natural language processing[51] make efficient querying possible. However, there is no integration of genotype and phenotype data today. GQL should be useful for both interrogating a single genome and interrogating several genomes across groups of individuals but will need to integrate with existing EMR systems so phenotype data can be queried together with genomes.

3.2.4 Privacy (Computer Security)

The genome is the ultimate unique identifier. All privacy is lost once the public has access to the genome of an individual, but current regulation, based on the Health Information Portability and Accountability Act, is silent about it.[10, 17, 49] Though the Genetic Information Nondiscrimination Act addresses accountability for the use of genetic information[36] privacy laws must change to ensure sensitive information is available only to the appropriate agents. Checking that a given study satisfies a specific privacy definition requires formal reasoning about the data manipulations that generated the disclosed dataimpossible without a declarative specification (such as GQL) of such manipulations

3.2.5 Provenance (Software Engineering)

GQL is an ideal way to record the provenance of genomic study conclusions. Current scripts (such as GATK) often consist of code that is too ad hoc for human readability and span various programming languages too low level for automatic analysis. By contrast, publishing the set of declarative GQL queries along with their results would significantly enhance the clarity and reproducibility of a study's claims.

Provenance queries also enable scientists to reuse the data of previously published computing-intensive studies. Rather than run their expensive queries directly on the original input databases, these scientists would prefer to launch an automatic search for previously published studies in which provenance queries correspond to (parts of) the computation needed by their own queries. The results of provenance queries can be directly imported and used as partial results of a new study's queries, skipping re-computation. This scenario corresponds in relational database practice to rewriting queries using views.

3.2.6 Scaling (probabilistic inference)

Learning the correlation between diseases and variations can be tackled differently if there are a large number of genomes. It may be less critical to accurately evaluate individual variations for such a discovery problem, as erroneous variations are unlikely to occur over a large group of randomly selected individuals. More generally, do other inference techniques leverage the presence of data at scale? As an example, Google leverages the big-data collections it has to find common misspellings. Note that accurately screening individual variations is still needed for personalized medicine.

3.2.7 Crowd sourcing (data mining)

Crowdsourcing might be able to address difficult challenges like cancer,[55] but to do so, the query system must have mechanisms to allow a group to work coherently on a problem. Imagine that a group of talented high-school science students are looking for genetic associations from cases and controls for a disease. A potentially useful GQL mechanism would be selecting a random subset of cases and controls that are nevertheless genetically matched (arising from a single mixing population). Researchers could then query for a random subset of 100 individuals with a fraction of network bandwidth while still providing similar statistical power for detecting associations.

3.2.8 Reducing Costs (computer systems)

Personalized medicine must be commoditized to be successful so requires computer systems research; for example, since most genomes are read-only, could solid-state disks be leveraged? Efficient decomposition between the cloud and a workstation is key to reducing data traffic in and out of the cloud. While genomics has been dominated by expensive parallel computers, it makes economic sense to adapt genomic software to harness the parallelism in todays cheap multicore CPUs.

3.3 Related Work

The popular UCSC genome browser [39] 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., [20, 60, 34] 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 [32] that the major variant callers differ greatly.

Custom genomic analysis can benefit from existing frameworks such as GATK [50], Samtools [45], and BEDtools [56] that offer routines for commonly used biological functions. However, these tools require substantial hand-crafted procedural scripts and are slow (Section 5.1.1).

3.4 Conclusion

Genomics is moving from an era of scarcity (a few genomes with imperfect coverage) to abundance (universal sequencing with high coverage and cheap re-sequencing when needed). This shift requires geneticists and computer scientists alike to rethink genomic processing from ad hoc tools that support a few scientists to commodity software that supports a world of medicine. The history of computer systems teaches us that as systems move from scarcity to abundance, modularity is paramount; ad hoc software must be replaced by a set of layers with well-defined interfaces. That these trends have been recognized by industry is seen in the shift from machine-specific formats (such as Illumina) to standards (such as BAM) and from vendor-specific variant formats to VCF. The 1000 Genomes Project (http://www.1000genomes.org/) has gained momentum, with a large number of sequences accessible today. However, much progress has involved defining data formats without powerful interface functionality; using an Internet analogy, it is as if TCP packet formats were defined without the socket interface.

We propose going beyond the layering implicit in current industry standards to enable personalized medicine and discovery. We advocate the separation of evidence from inference and individual variation from variation across groups (see Figure 1.5a). We propose a specific interface between the evidence layer and the inference layer via the GQL. While GQL is based on a relational model using a virtual interval relation, further effort is required beyond standard relational optimization to allow GQL to scale to large genomes and large populations.

We described several benefits from separating evidence from inference; for example, a genome repository accessible by GQL offers the ability to reuse genomic data across studies, logically assemble

case-control cohorts, and quickly change queries without ad hoc programming. GQL also offers the ability to reduce the quality of inference on an individual basis when applying group inference over large populations. We also described simple ideas for scaling GQL to populations using compressed-strength indices and for doing evidence-layer processing in the cloud.

We emphasized that GQL and the evidence layer are only our initial attempt at capturing abstractions for genomics. We hope this article prompts a wider conversation between computer scientists and biologists to tease out the right interfaces and layers for medical applications of genomics. Beyond abstractions much work remains to complete the vision, including better large-scale inference, system optimizations to increase efficiency, information retrieval to make medical records computer readable, and security mechanisms. While the work is a challenge, making genetics interactive is potentially transformative as in the shift from batch processing to time sharing. Moreover, computer scientists only occasionally get a chance to work on large systems (such as the Internet or Unix) that can change the world.

3.5 Acknowledgement

Chapter 3 in part, is a reprint of the material that appears in:

1. “Abstractions for Genomics”, in Communications of the ACM, Bafna, Vineet; Deutsch, Alin; Heiberg, Andrew; Kozanitis, Christos; Ohno-Machado, Lucila; Varghese, George. Volume 56, Issue 1, 2013.

2. “Using the Genome Query Language (GQL) to uncover genetic variation.”, Kozanitis, Christos; Heiberg, Andrew; Varghese, George; Bafna, Vineet. To appear in Bioinformatics

3. “‘Making Genomics Interactive: Towards Querying a Million Genomics in a few seconds.”, Kozanitis, Christos; Bafna, Vineet, Pandya, Ravi; Varghese, George. Submitted to the Symposium of Operating Systems and Principles (SOSP) 2013.

The dissertation author was the primary investigator of these papers. The dissertation author was also the primary author of papers 2 and 3.

Chapter 4

GQL implementation

"A good idea is about ten percent
and implementation and hard work, and luck is 90 percent."
Guy Kawasaki

This chapter provides the implemenation of the abstraction that Chapter 3 introduced and provides a number of experiments that demonstrate the efficiency of this implementation both in a single desktop execution and in the cloud. Section 4.1 shows the pipeline of GQL. Section 4.2 describes the set of optimizations that make GQL efficient and capable of supporting large amounts of data. Section 4.3 demonstrates this efficiency through a number of micro and macro experiments. Section 4.4 introduces our online graphical interface. We conclude the chapter with Section 4.5 which describes the next steps that are required to scale GQL to support queries acrosss millions of genomes.

4.1 System Design

Figure 4.1 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 4.2). The backend is written in C++ and uses a set of optimizations described in Section 4.2.

We omit details of the (fairly straightforward) compiler (see Section A.1 and Section A.2 for details of the grammar that the compiler implements) Query Planner (see Figure A.1 for a detailed decision diagram) and proceed to describe the more interesting optimizations.

4.2 Optimizations

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

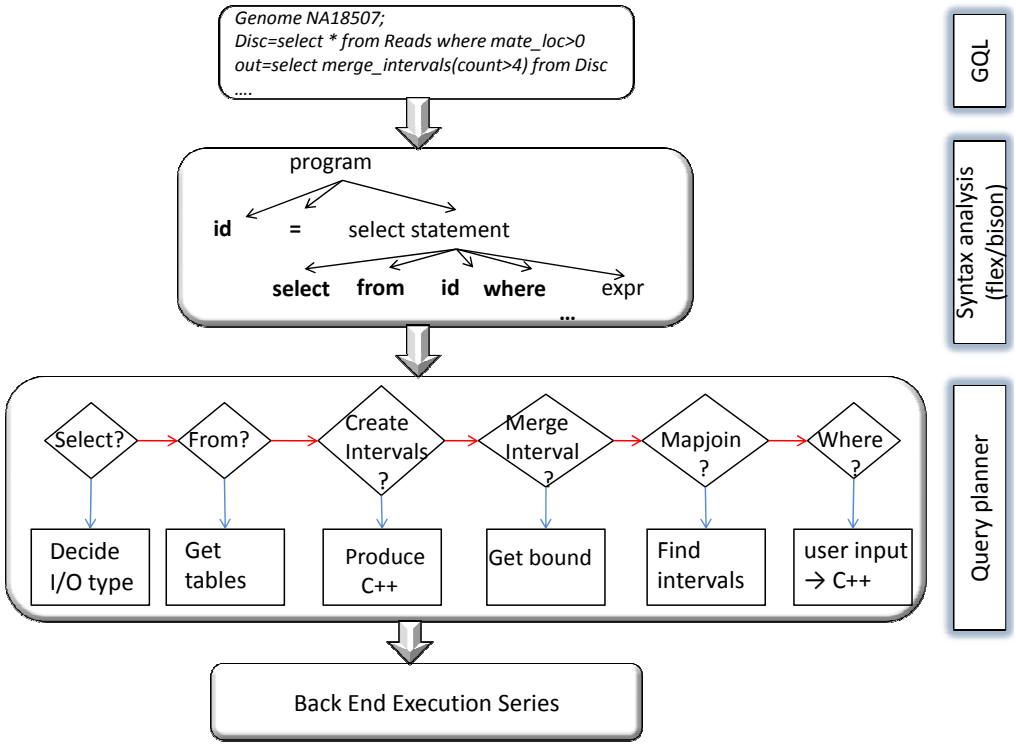


Figure 4.1. The GQL pipeline.

4.2.1 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.

4.2.2 Meta Data for Speedup

Variant callers [20, 60, 34] 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 4.3.1). 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 4.2.2

Currently, we use around 20 bytes of metadata. We have not yet compressed the index file as in say vertical databases [52]. 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 1. Querying for Reads that map to the reference. Unmapped Reads have negative location by convention.

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

4.2.3 Cached Parsing

Interpreted expression evaluation can be expensive because of extra memory accesses. Consider the algebraic part of the *where* statement of Algorithm 1. 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 1 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 4.3.1.

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

4.2.4 Interval Tree Based Interval Joins

We optimize the implementation of *IntervalJoin* operator using a *centered interval tree* data-structure [2]. 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.

4.2.5 Lazy Joins

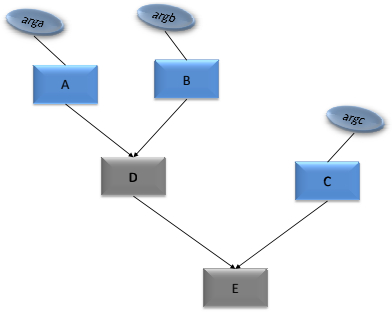
As the name indicates, Lazy Joins postpone materializing a Join, as the cross-product of GQL tables can be large. Instead, the *IntervalJoin* table of two tables A and B is represented as an array of tuples (x,y), logically denoting the joining of the $x-th$ entry of A with the $y-th$ entry of B. Since 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 4.2a 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

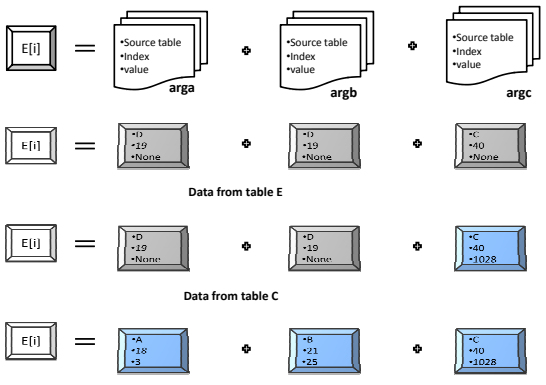
an expression a structure which keeps track of which entry of the parent table contains the desired value; we recursively update the fields of this structure when we open the parent table. In Figure 4.2a, assume that one needs to calculate the value $E[i] = arga + argb + argc$. Since *arga*, *argb* and *argc* are unknown variables, we assign a structure to each of them as in Figure 4.2b.

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 4.2b.

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.



(a)



(b)

Figure 4.2. An example of GQL evaluating a set of nested joins

.

We have not found lazy joins described in the database literature. In standard query planning [57], 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.

4.2.6 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.

4.3 Evaluation

We describe microbenchmarks in Section 4.3.1, macro benchmarks in Section 4.3.2, and a comparison with writing queries in the existing GATK framework in Section 5.1.1.

4.3.1 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 4.3 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*

Speedups due to Metadata indexing Figure 4.4 compares taken to compute the query of Algorithm 1 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.

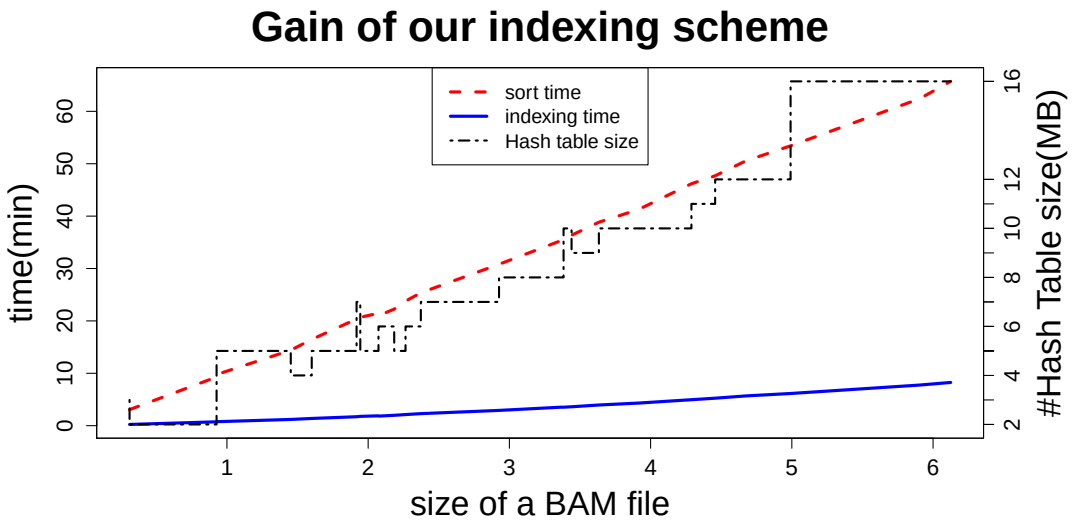


Figure 4.3. Comparison of our hash table based indexing method with sorting. Sorting per *QNAME* gives a lower bound of the time it takes to discover the links between the pair ends in a BAM file. The figure also contains the respective maximum required capacity of our hash table.

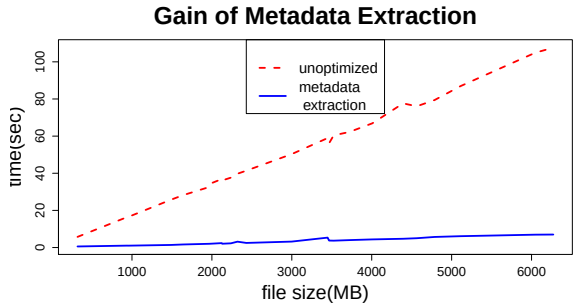


Figure 4.4. Comparison of execution times when query Algorithm 1 uses the BAM file instead of the metadata file. The file sizes are for chromosomes in NA18506 ranging from 320MB (chrY) to 6.5GB (chr1).

Cached Parsing Figure 4.5 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.

Interval Tree Gains Figure 4.6 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

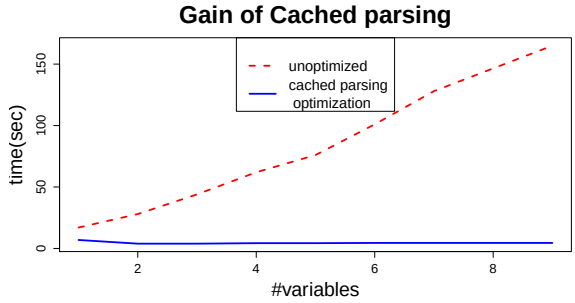


Figure 4.5. Impact of cached parsing. In the absence of the optimization the execution time depends on the interpreter delay and grows linearly with the number of variables contained in an expression.

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.

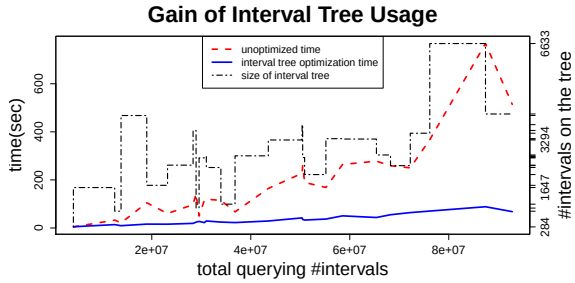


Figure 4.6. Impact of using Interval trees

Lazy Joins Figure 4.7 and Figure 4.8 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 Theorem 4.1 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

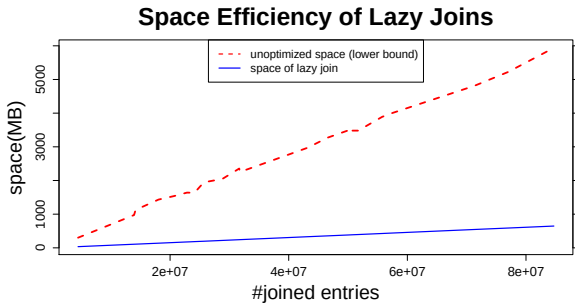


Figure 4.7. Space savings for lazy joins.

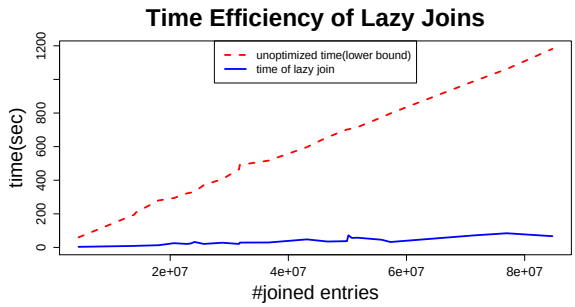


Figure 4.8. Time savings for lazy joins.

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

Table 4.1. Ease of access of expression evaluations on lazily joined tables

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

4.3.2 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 4.9 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.

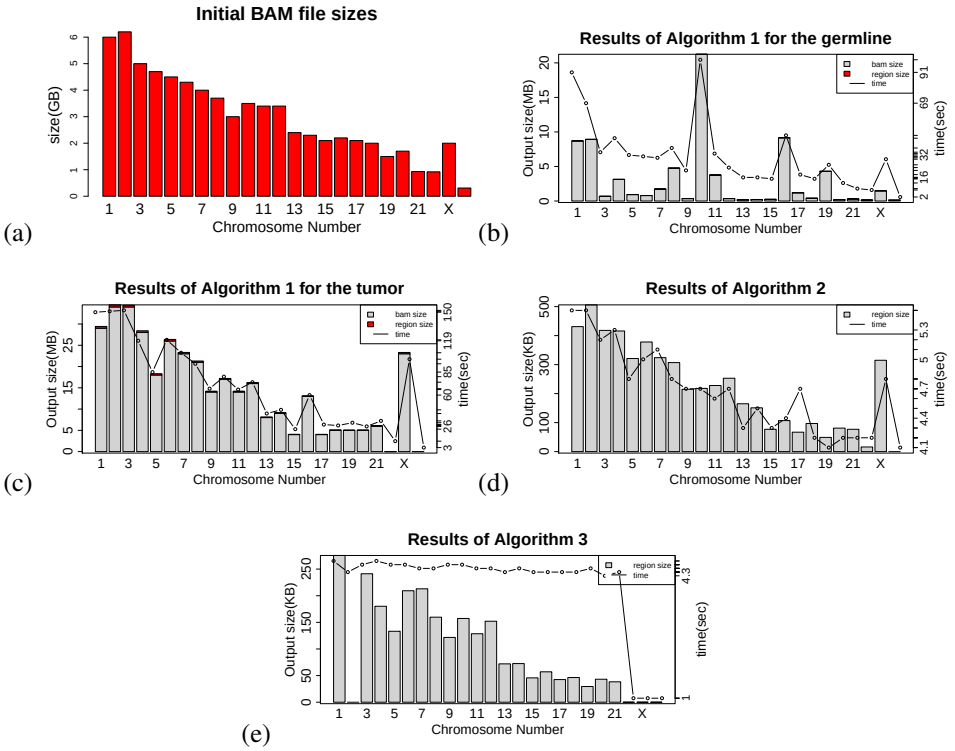


Figure 4.9. Macro benchmark experiments on Azure Virtual Machines. The execution times shown at (b)-(e) are per node execution times.

4.4 A Web Interface

We have developed a prototype implementation (see Figure 4.10) on the web. The uploaded genome (in BAM format) can be queried through a simple text interface that allows the user to write a GQL query. The query is compiled and executed, and the output returned as a (smaller) BAM file that can be visualized through appropriate browsers (such as jbrowse, http://jbrowse.org) or downloaded to the client for further analysis.

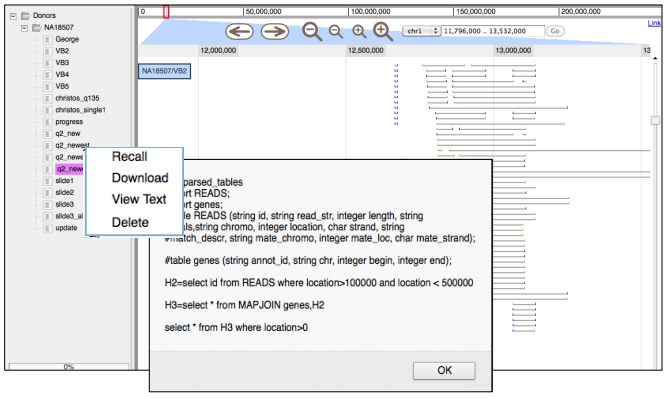


Figure 4.10. Prototype implementation of GQL, with visualization using the open-source tool jbrowse; included are discordant paired-end reads supporting deletions.

4.5 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 [3] 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 [52].

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 chapter. Further parallelism would divide chromosomes into blocks with a small amount of overlap. SSDs could be leveraged to store genome metadata.

4.6 Acknowledgement

Chapter 4 is in part a reprint of the material as it appears on papers:

1. “Using the Genome Query Language (GQL) to uncover genetic variation.”, Kozanitis, Christos; Heiberg, Andrew; Varghese, George; Bafna, Vineet. To appear in Bioinformatics

2. “‘Making Genomics Interactive: Towards Querying a Million Genomics in a few seconds.”, Kozanitis, Christos; Bafna, Vineet, Pandya, Ravi; Varghese, George. Submitted to the Symposium of Operating Systems and Principles (SOSP) 2013.

The dissertation author was the primary investigator and author of these papers.

Chapter 5

GQL in Action

"Having rounded up my horses, I now put them through their paces."
Arnold Toynbee

In Section 5.1 we query a number of sequences from the TCGA archive, that were obtained using both whole genome and exome sequences runs. In the same section (Section 5.1.1) we also stress the effort it takes to write similar queries in GATK which is the state of the art of evidence collection today. Section 5.2 contains experiments with evidence retrieval queries from sequences of the 1000 genomes project. Section 5.3 provides ideas of deployment of GQL and Section 5.4 discusses the lessons that we learnt from the project.

5.1 Results on Datasets from the TCGA archive

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 [32].

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 [13].

The next series of queries is on genomes with ids *a2-a04p-01* and *a2-a04p-10* from the TCGA cancer archive [42] 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 2 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).

Algorithm 2. 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
```

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 same individual as tumors are often dictated by somatically acquired mutations; such difference queries are standard [32]. 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 2 (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 3 shows the GQL query that encodes this procedure.

Algorithm 3. 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 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 4.

Algorithm 4. 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
```

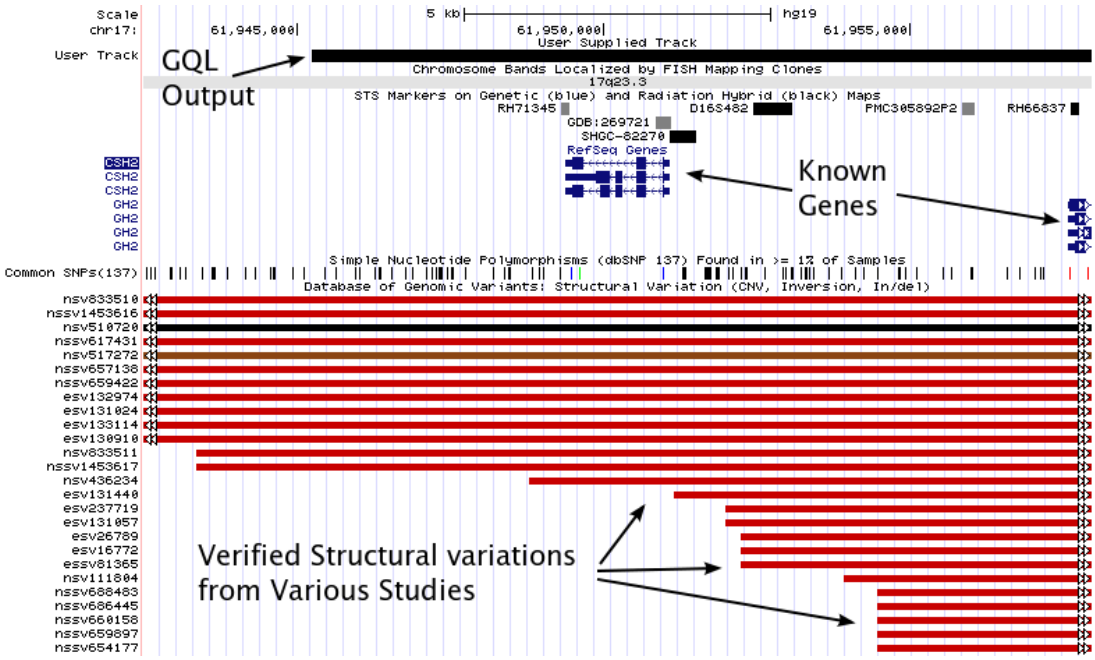


Figure 5.1. Sample snapshot of the UCSC genome browser on an output interval from Algorithm 4

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 5.1). 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 [14]. 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 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.

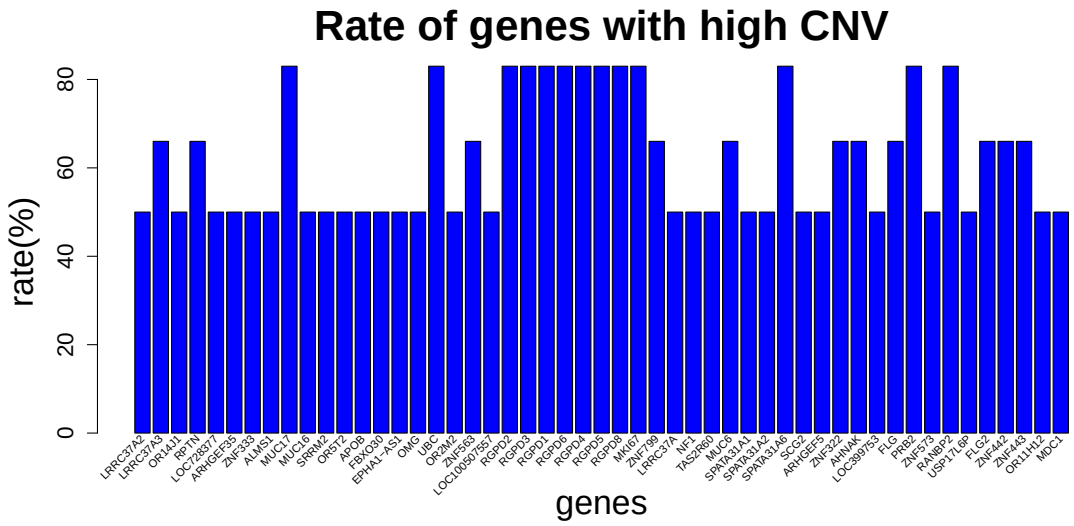


Figure 5.2. Genes that overlap with a copy number region in at least 60% of the exomes of our population.

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 [12] 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 5.2 shows the genes that commonly overlap with high copy number regions in at least 60% of the population.

5.1.1 Comparison with GATK

We also implemented the deletion query of Algorithm 2 using the commonly used GATK (Genome Analysis Toolkit) [50]. 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 [11]. Finally, on a single VM when run on chr1 of the breast cancer genome *a2-04p-01.bam* used in Section 4.3, 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 [57] 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.

5.2 Results on Datasets from 1000genomes

We demonstrate the flexibility and efficiency of GQL by walking through some use cases that involve identifying donor regions with variations, and supplying the read evidence supporting the variation. We use the Yoruban trio (both parents and a child) from the Hapmap project [8] that were sequenced as part of the 1000-genome project. The genomes are labeled NA18507, NA18508, and NA18506 for the father, mother, and the child respectively. Each genome was sequenced to about $40\times$ coverage ($\sim 1B$ mapped reads) using 2×100bp paired-end reads from 300bp inserts. We use the large deletion detection problem for a case study. The corresponding input BAM file sizes are all in the range 73-100 GB.

Large deletions on NA18506. Paired-end mapping provides an important source of evidence for identifying large structural variations [41, 15]. Length discordant clones (pairs of reads with uncharacteristically long alignment distance between the two mapped ends) are indicative of a deletion event. We start by using GQL to identify all genomic regions (in reference coordinates) that have an 'abundance' of length-discordant clones.

1. Select inserts where both ends are mapped, and the insert size is at least 350bp(i.e. the insert size exceeds the mean insert size by more than $5\times$ the standard deviation), and at most 1Mbp.

```
Discordant=select * from READS
where location>=0
and mate_loc>=0
and abs(mate_loc+length-location)>350
and abs(mate_loc+length-location)<1000000)
```

2. Create an interval table, with an interval for each discordant read (by specifying the begin and end coordinates). This changes the output data type from reads to intervals on the reference.

```
Disc2Intrvl=select create_intervals() from Discordant
using intervals(location, mate_loc, both_mates)
```

3. We then merge overlapping intervals, and identify maximal intervals that are overlapped by at least 5 clones. This set of intervals points to all regions of the reference with evidence of a deletion.

```
Predicted_deletions=select merge_intervals(interval_count > 4)
from Disc2Intrvl
```

4. To output the intervals, and the supporting discordant reads, we do a final MAPJOIN and print the result. This changes the output data type back to reads stored in a BAM file that can be used downstream by other software.

```
out=select * from MAPJOIN Predicted_deletions, Discordant
using intervals(location, mate_loc, both_mates)
print out
```

Note that entire query is a few lines of GQL. Further, the query took 10 minutes to execute on the entire genome. All the results we report used a single i7 Intel CPU with 18 GB of RAM and 2 TB of disk. Much of the execution time was spent on printing the large output (12K regions, with 44MB of supporting evidence).

These observations suggest that complex GQL queries could efficiently be performed in the cloud. Given that organizations can easily obtain 100 Mbps speeds today (Amazon's Direct Connect even allows 1 Gbps speeds), the output of 44 MB can be retrieved interactively to a user desktop in less than 4 seconds. By contrast, downloading the original BAM files would take more than 2 hours at 100 Mbps. Second, while we measured the query to take 10 minutes on a single processor, simple parallelism by chromosome

(easily and cheaply feasible in the cloud) should provide a factor of 20 speedup, leading to a query time of 30 seconds.

Further, we wrote a program to convert the intervals output by a GQL query to the *BED* format, which can then be uploaded to the UCSC genome browser for further investigation, including comparisons with other reference annotations. See Figure 5.3a,b for examples showing overlap between output regions, and known CNVs, and a previously reported Indel in NA18507 [18], the father of NA18506.

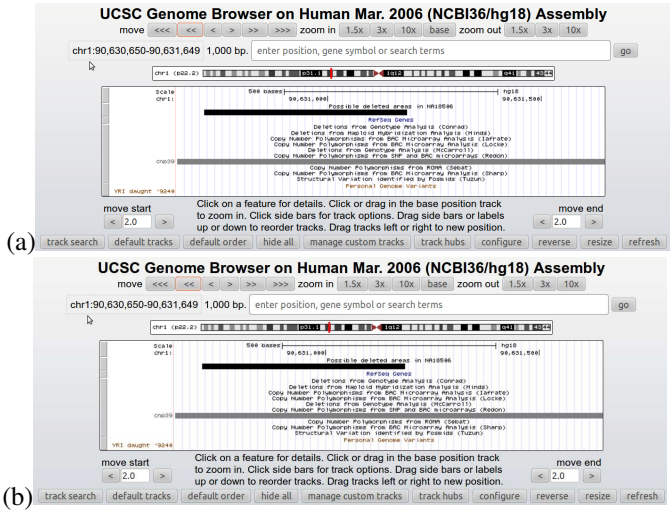


Figure 5.3. UCSC browser snapshots from areas of NA18506 which are candidate deletions
(a) An area which overlaps with a known CNV site. (b) An area which overlaps with a known deletion of the father NA18507.

Conrad and colleagues [23] used array hybridization to identify deletions in a number of genomes, including NA18506. We wrote a GQL query to compare the two sets of predictions as follows: we created a table with all of Conrad's predictions, and performed a *IntervalJoin* with our predicted intervals. We then output the common regions that overlap, and the corresponding discordant read evidence. This query ran in 7 minutes and the total size of the produced bam files was 164KB. Our results overlap with 15 out of the 23 findings of [23]. To look at support for regions identified by Conrad, but not by us, we used *IntervalJoin* to identify all discordant and concordant reads that overlap with Conrad-only predictions using the following GQL query.

```
Discordant=select * from READS
where location>=0 and mate_loc>=0
and abs(mate_loc+length-location)>350
and abs(mate_loc+length-location)<1000000)
```

```
out=select * from mapjoin conr_only_intrvls, Discordant
using intervals(location, mate_loc, both_mates)
```

Interestingly, none of the 8 Conrad-only predictions were supported by discordant-reads. Further, 6 of the 8 Conrad-only regions had concordant read coverage exceeding 30; the other two had coverage exceeding 10, suggesting heterozygous deletion, at best. The GQL query to retrieve the entire evidence took 12 minutes, and a few lines of code.

To validate regions with discordant read evidence output by us, but not predicted by Conrad, we ran the same deletion-query in the parents of NA18506 to see if they are reproduced in the parents. 3 of the predicted deletions overlapped with discordant reads in both parents, and 9 in one parent. Only 3 of our predicted deletions do not appear in any of the parents. In each case, further statistical analysis can be used on the greatly reduced data-set to help validate the predicted deletions.

Inversions in the donor genome. To detect putative inversions, we locate regions that are covered by at least 5 different pairs of orientation discordant reads.

```
Discordant=select * from READS using intervals (location,mate_loc, both_mates
    )
where location>=0 and mate_loc>=0
and strand==mate_strand
and abs(mate_loc+length-location)>270
and abs(mate_loc+length-location)<1000000)
```

The query needs 8 minutes to run and identifies 366 regions of possible inversion events and returns 47324 alignments which are stored in BAM files of size 3 MB.

High CNV. The average coverage of reads in the data-set is 40. To identify high copy number regions (possible duplications), we locate regions with ≥ 200 coverage.

```
H1=select create_intervals() from READS
where location>=0
out=select merge_intervals(interval_coverage>200) from H1
```

The query needs 30 minutes to run and the evidence data of the output consists of 9.7M reads stored in BAM files of size 664 MB. We contrast this with the original BAM file of size of 72 GB that a

caller would have parsed had GQL not been used.

Reads mapping to specific intervals. Here, we output all reads that map to known genic regions. The query uses an external table which consists of all known genes based on the RefSeq database. The execution time is 288 minutes, limited by the huge output size (495M reads).

```
mapped_reads=select * from reads
where location>0
out=select *
from mapjoin Refseq_genes using intervals(txStart, txEnd),
mapped_reads using intervals (location, location+length, both_mates)
```

Efficacy of Mapjoin implementation. In most databases, Joins are typically the most time-expensive operation. Typical GQL queries use intermediate MapJoin operations extensively. We implement a special Lazy Join procedure to greatly improve the run-time, explained here with an example as "output all reads that overlap with genes whose transcriptional start is in a CpG island". Tables describing CpG islands are available (e.g., [30]) and can be downloaded from the UCSC genome browser. A non-expert user might write the following sub-optimal GQL code which applies the position restriction on the (large) output of the *IntervalJoin* between all reads and genes.

```
mapped_reads=select * from reads
where location>0
reads_genes=select * from
mapjoin Refseq_genes using intervals(txStart, txEnd),
mapped_reads using intervals (location, location+length)
out=select * from mapjoin cpgIsland_hg18 using intervals(chromStart,
chromEnd), reads_genes using intervals(location, location+length)
```

In the absence of lazy evaluation, the execution time of this snippet would be bounded by the extremely large execution time (288 minutes) of the data intensive query of the previous paragraph. Lazy evaluation, which allows us to join using virtual joins and bypasses intermediate data generation, provides the same result within 42 minutes for the entire genome.

Common deletions in the YRI population. Here, we extend queries from single donor to multiple donors. Kidd et al [41] validated deletions in 8 geographically diverse individuals using a fosmid sub-

cloning strategy, including NA18507. Of these deletions, the ones that overlap with genes suggest a phenotypically interesting deletion. Therefore, we ask how many of such Chr 1 deletions are prevalent in the entire HapMap YRI sub-population (77 individuals).

1. Get intervals with at least 4 length-discordant reads.

```
Disc=select * from Reads
where 350 <= abs(location+mate_loc-length)
and abs(location+mate_loc-length) <= 1000000

De=select merge_intervals(count>4) from Disc
```

2. MapJoin with validated intervals.

```
Del_Overlapping=select * from MAPJOIN Del, Kidd_results
using intervals (begin, end)
```

3. Map Join with known genes.

```
Gene_overlapping=select * from MAPJOIN Del, RefSeq_genes
using intervals (txStart, txEnd)
```

The query takes 5 minutes to find the common regions of chr1 across the entire population and 30 minutes to print the accompanying reads that support the query. Figure 5.4 shows the rate according to which each validated deletion appears to other Yoruban individuals and the affected genes. 8 of the deletions are common in at least 30% of the individuals, 2 are common in at least 50% and 1 deletion is common in 80% of the YRI population. The information provides a quick first look at deletion polymorphisms in the Yoruban population. For example, 81% of the individuals have a deletion in 1q21.1 containing the Neuroblastoma Breakpoint gene family (NBPF), so called because of prevalent translocation event in Neuroblastoma [65]. Also, 42% of the individuals have a deletion in 1p36.11, where the deletion removes a large part of the RHD gene, responsible for Rh group D antigen. Such deletions have previously been reported in other populations [66]. We also find a common deletion (22%) involving Complement factor H-related protein, which has been associated with age related macular degeneration (AMD) [61]. Other common deletions involving multiple genes are shown in Figure 5.4.

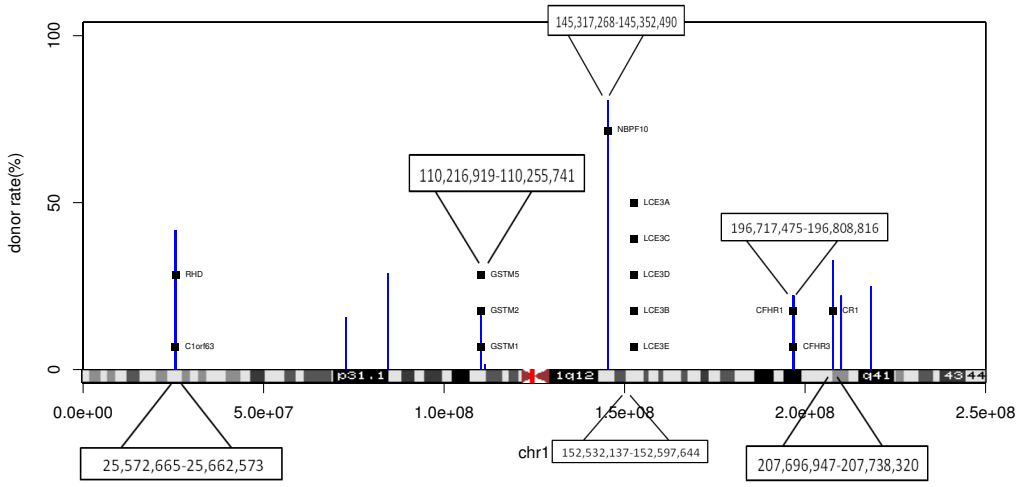


Figure 5.4. Frequencies of common deletions across the YRI population that match the results of [41] for the chromosome 1 of NA18507. For each genome we find candidate deleted areas and we apply a *IntervalJoin*operation with the former deletions. The figure shows how many times each deletion of [41] overlaps with some deletion of other genomes and it also shows the genes that are affected by said deletions.

5.3 Incremental Deployment

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

5.3.1 Speeding up existing callers

In this experiment we demonstrate the speedup that GQL can provide to an *existing* structural variation caller called Breakdancer [20].

A normal run of Breakdancer_max with input being the chr2 alignments of the breast cancer genome *a2-a04p-01* of Section 4.3 (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 [20], 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$\times$ 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 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

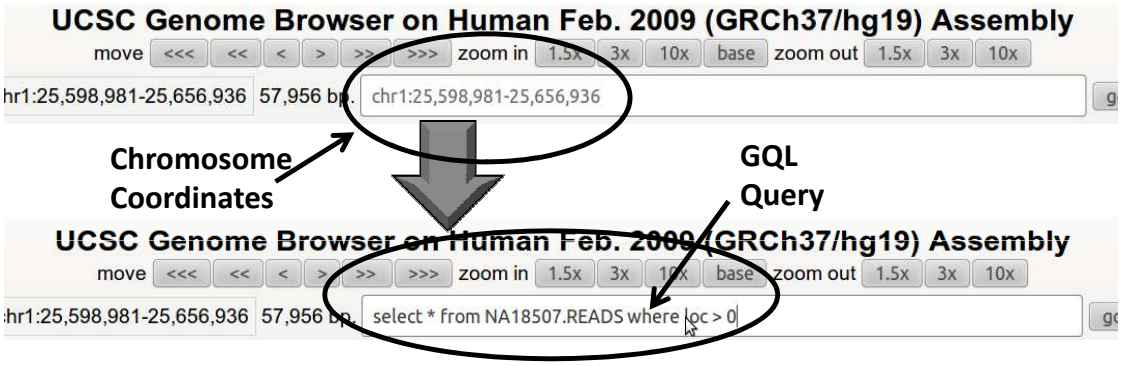


Figure 5.5. Semantic versus location browsing.

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.

5.3.2 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 5.5. Ideally, the UCSC browser should be modified to directly support a field to enter GQL queries as mocked up in Figure 5.5. Our performance results indicate that semantic browsing can be done in minutes if the answer to the query is small.

5.4 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 [23] on NA18506. Sifting with further GQL queries revealed that the purported deletions found only in [23] 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 **??**) 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.

5.5 Acknowledgement

Chapter 5 is in part a reprint of the material as it appears on papers:

1. "Using the Genome Query Language (GQL) to uncover genetic variation.", Kozanitis, Christos; Heiberg, Andrew; Varghese, George; Bafna, Vineet. To appear in Bioinformatics

2. "'Making Genomics Interactive: Towards Querying a Million Genomics in a few seconds.", Kozanitis, Christos; Bafna, Vineet, Pandya, Ravi; Varghese, George. Submitted to the Symposium of Operating Systems and Principles (SOSP) 2013.

The dissertation author was the primary investigator and author of these papers.

5.6 Funding

The authors acknowledge support from the NIH sponsored iDASH project (Grant U54 HL108460), NIH 5R01-HG004962, and a CSRO scholarship to CK.

Chapter 6

Conclusions

"To hackers, DNA is just another operating system waiting to be hacked."
Marc Goodman

Hacking of an operating system requires familiarity with it; sadly the current state of knowledge about human DNA is currently too limited for this thesis to achieve a goal of this scale; rather, this thesis can be viewed as creating an efficient debugger that will help other hackers better understand the "operating system" we call possess.

To do so, 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 5.1) seems valuable.

The first bottleneck involved in this vision, that this thesis addresses, is the sheer size of the data. Given that discovery studies have to consider sequences from large populations and the dataset of a single person's genomes can be several Gigabytes, storage and sharing of the data are of great concern.

SLIMGENEleverages the similarity between DNA of different human beings and the redundancy of the output of the High Throughput Sequencing systems to compress data. Given that different human beings differ between each other in 1% of their genomes, SLIMGENEdiscards all DNA fragments and reconstructs them from their differences from a reference genome. In addition, without significant loss of information, it reduces the quantization levels of quality scores to increase the compression rate. Illumina Inc was motivated by the results of this thesis on lossy quality value compression and created pipelines that produce quality scores from fewer quantization levels.

Although SLIMGENEreduces the size of the data, the resulting files are still too large for individual

researchers to maintain and download and thus the second part of this thesis focuses on browsing of genome data "on demand".

Thus the second contribution of this thesis is the claim that a declarative query language such as GQL can allow interactive browsing of genome data that is hard to do with existing tools. Existing frameworks such as GATK [50] and BEDtools [56] raise the level of abstraction but still require significant effort to write/modify queries (hours, see Section 5.1.1), and are often slow to run because of lack of indexing and other optimizations. In addition, the choice of a declarative syntax allows for richer syntax, including multiple join operations, and operations on population data. Moreover, it separates the implementation from the query, and allows for optimizations in the implementation that are transparent to the naive user.

Separating evidence and inference allowed GQL to progress without recourse to probabilistic databases. The combination of *IntervalJoin* and MergeIntervals 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.

The results suggest that a cloud implementation of GQL is efficient and fast. In particular, for most selective queries, the resulting output is small (Mbytes) and can be retrieved in a few seconds across the network. Further, the query times we report are in the order of minutes using a cheap single processor for genome wide scans. Simple distributed computing style of parallel implementation reduces this to seconds.

Finally, GQL is compatible with existing genomic software. Existing callers can be refactored to retrieve evidence from cloud repositories using GQL, thereby relegating large data handling to GQL with consequent speedups as we demonstrated for Breakdancer. Further, the results of GQL queries can be viewed using the UCSC browser. In principle, we can also support 'semantic browsing' by properties of genomic regions in addition to browsing by position (syntactic browsing).

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 [28]. Inference algorithms must be standardized. But the siren song of indexing the world's genomes is alluring; the computer systems community must work with the genomics community to make this dream real not just by applying existing techniques, but also by introducing new computer science constructs tailormade for genetics.

6.1 Acknowledgement

Chapter 6 is in part a reprint of the material as it appears on papers:

1. “Using the Genome Query Language (GQL) to uncover genetic variation.”, Kozanitis, Christos; Heiberg, Andrew; Varghese, George; Bafna, Vineet. To appear in Bioinformatics

2. “‘Making Genomics Interactive: Towards Querying a Million Genomics in a few seconds.”, Kozanitis, Christos; Bafna, Vineet, Pandya, Ravi; Varghese, George. Submitted to the Symposium of Operating Systems and Principles (SOSP) 2013.

The dissertation author was the primary investigator and author of these papers.

Appendix A

Supplementary Material

A.1 Keywords of GQL

Here is the list of keywords of GQL. Note that the keywords are also recognized when a user enters them with upper case characters.

- both_mates
- include
- print
- genome
- select
- from
- mapjoin
- where
- table
- and
- or
- not
- integer

- float
- char
- string
- using
- interval_creation
- create_intervals
- merge_intervals
- intervals

A.2 The GQL grammar

⟨program⟩→ ⟨table_prototypes⟩ ⟨genome_names⟩ ⟨assigned_selects⟩ ⟨print_statements⟩

⟨table_prototypes⟩→ ⟨table_prototype⟩ ⟨table_prototypes⟩|ε

⟨table_prototype⟩→ ⟨table_keyword⟩ **(** ⟨table_args⟩ **) ;**

⟨table_keyword⟩→ **table** ⟨names⟩

⟨names⟩→ **ID**

⟨numbers⟩→ ⟨digit⟩ ⟨numbers⟩| ⟨digit⟩

⟨digit⟩→ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

⟨table_args⟩→ ⟨table_args⟩ **,** ⟨table_arg⟩
| ⟨table_arg⟩

⟨table_arg⟩→ **integer** ⟨names⟩
| **float** ⟨names⟩
| **char** ⟨names⟩
| **string** ⟨names⟩

⟨genome_names⟩→ ⟨genome_names⟩ ⟨genome_names⟩
| **genome** ⟨names⟩ **;**
| **genome** ⟨names⟩ *** ;**

| **genome * where** ⟨where_args⟩ **;**

⟨assigned_selects⟩→ ⟨assigned_select⟩ ⟨assigned_selects⟩ | ε

⟨assigned_select⟩→ ⟨lvalue⟩ **=** ⟨select_statement⟩

⟨lvalue⟩→ ⟨names⟩

⟨select_statement⟩→ **select** ⟨select_args⟩ **from** ⟨compound_from_arg⟩ **where** ⟨where_args⟩
| **select** ⟨select_intrvl_args⟩ **from** ⟨compound_from_arg⟩
| **select** ⟨select_args⟩ **from mapjoin** ⟨from_args⟩

⟨select_intrvl_args⟩→ **create_intervals ( )**
| **merge_intervals (** ⟨names⟩ > ⟨numbers⟩ **)**
| **merge_intervals (** ⟨names⟩ < ⟨numbers⟩ **)**

⟨select_args⟩→ ***|** ⟨select_arg_series⟩

⟨select_arg_series⟩→ ⟨names⟩ **,** ⟨select_arg_series⟩
| ⟨names⟩
| ⟨obj_names⟩

⟨from_args⟩→ ⟨compound_from_arg⟩ **,** ⟨from_args⟩
| ⟨compound_from_arg⟩

⟨compound_from_arg⟩→ ⟨from_arg⟩
| ⟨from_arg⟩ **using intervals (** ⟨arith_expr⟩ **,** ⟨arith_expr⟩ **)**
| ⟨from_arg⟩ **using intervals (** ⟨arith_expr⟩ **,** ⟨arith_expr⟩ **, both_mates)**

⟨from_arg⟩→ ⟨names⟩

⟨where_args⟩→ ⟨where_args⟩ **,** ⟨where_args⟩
| ⟨where_args⟩ **and** ⟨where_args⟩
| ⟨where_args⟩ **or** ⟨where_args⟩
| **not** ⟨where_args⟩
| **(** ⟨where_args⟩ **)**
| ⟨lowest_expr⟩

⟨lowest_expr⟩→ ⟨arith_expr⟩ ⟨comparison_op⟩ ⟨arith_expr⟩

⟨arith_expr⟩→ ⟨arith_expr⟩ ⟨arith_op⟩ ⟨arith_expr⟩
| ⟨arith_op⟩ ⟨arith_expr⟩
| ⟨names⟩ **(** ⟨arith_expr⟩ **)**
| ⟨names⟩
| ⟨obj_names⟩
| ⟨numbers⟩
| **const_string**

| **const_char**

⟨obj_names⟩→ ⟨names⟩ **.** ⟨names⟩

⟨comparison_op⟩→ == | >= | <= | <

⟨arith_op⟩→ **+** | **-** | ***** | **/** | **%**

⟨print_statements⟩→ ⟨print_statement⟩ ⟨print_statements⟩| ε

⟨print_statement⟩→ **print** ⟨names⟩
| **print** ⟨names⟩ **both_mates**

A.3 THE GQL user guide

A.3.1 The environment

The installation of GQL requires that the following variables are set.

1. SAMLOC should point to the installation directory of samtools
2. DONOR_DIR should point to the parent directory of donor genomes.
3. BIOSQL_HOME should point to the installation directory of GQL
4. PATH should contain SAMLOC.

A.3.2 GQL projects

GQL organizes experiments into projects. All donors that participate in a project are assumed to have been aligned against the same reference and their chromosome naming follows the same conventions.

When a user creates a project with name P, src_tables_dir S and destination directory d, they need to move all donors of the project under under $DONOR_DIR/P, all text tables under S/P and all products of the queries that use this project will be found under d/P.

A.3.3 Text tables

Text tables are tab delimited text files that should be placed under S/P for project P with src_tables S. A table with name T should have a file name T.txt. The first line of a Text table file should always contain the following content: *#table T (type var1, type var2...);* where T is the table name, type should be any of {integer, string, char, float} and var should be any string of characters. Note that at least a field is

expected to contain chromosome information. **ATTENTION** for proper functionality, the table name of the first line should match the file name without the .txt suffix. The rest lines of a text table file can be either comment lines that begin with '#' or data.

A.3.4 Reads Table

The *Reads* table is embedded for every genome and in the current prototype a user can access the following fields.

- integer location –the mapping coordinate of the read
- integer length – the read length
- char strand – the mapping strand
- integer mate_loc – the mapping location of the other end
- mate_strand – the mapping strand of the other end.

Note that the user does not have access to the chromosome field because a query runs over all chromosomes iteratively.

A.3.5 program structure.

Each GQL program consists of 4 distinct parts:

1. The *include* line
2. The donor selection lines
3. A set of select statement assignments
4. A set of print statements.

The include line. Currently the first line of each program should be "include <tables.txt>" which loads the table definitions.

Donor selection. The user selects on which genomes the query should run on. The choice can either be explicit, e.g *genome NA18507, NA18506;* or it can be an expression that contains a wild-card, such as: *genome NA185*;* or an SQL like statement such as *genome * where sex=="male" and population=="YRI"*. Note that

for the latter to be functional a user needs to upload a text table with name *genome_info* that should contain an entry for each donor genome with fields that depend on the information that is available for each donor. The only restriction is that the field that corresponds to a donor name should be labeled as *name*.

Select statement assignments. Each statement should be of the form *variable = select statement*. After the execution of a select statement that left variable is a table of the same type as the return of the select statement and it can be used as input in subsequent statements. There are four types of possible select statements.

- A selection statement of the form:

  *select * from Input-table where where_expr*

  where Input-table can be any table within the scope of the query and where_expr can be any boolean function of the attributes of Input-table. The statement returns a table of the same type and the same arguments as the Input-table.

- An interval creation of the form:

  select create_intervals() from Input-table using intervals (expr1, expr2,[both_mates]).

  This returns a table of Intervals with attributes *begin, end, chromosome* that is created on Input-table. expr1 and expr2 can be integer expressions on any field of Input-table and they specify the begin and end fields respectively. If Input-table is of type *Reads* a user can add the *both_mates* flag which specifies that both ends of a pair end participate in each interval.

- An interval merging of the form:

  select merge_intervals(interval count $\leqslant$ *constant) from Input-table*.

  The input has to be of type *Intervals* and the output is also of type *Intervals*.

- A *IntervalJoin*operation of the form:

  *select * from mapjoin Input-tableA using intervals(expr1, expr2, [both_mates]), Input-tableB using intervals(expr3, expr4, [both_mates])*.

  Both input tables can be of any type and the resulting table contains the attributes of Input-tableA followed by attributes of Input-tableB. In case there are naming conflicts we append suffix 1 to the resulting entry that originates from table A and suffix 2 to the resulting table from table B.

Print Statement A print statement consists of the keyword *print* followed by a table name. If the table name is of type *Reads* it can be followed by the *both_mates* keyword to let GQL know that reads need to be printed as pair ends. Below we summarize the output of GQL depending on the type of table.

- Reads. The output consists of a .bam file that contains all reads that a table contains, a .bam.short file that is a text file that contains only the starting, ending mapping coordinate and the orientation of each read, a .hist file which is used by our GUI to plot histogram related information and a .bam.links.txt which displays the correspondence of the current output with the initial bam file. The latter contains pairs of integers where the first integer corresponds to the location of a read in the output bam file and the other integer corresponds to the location of the same read in the original bam file. In case keyword *both_mates* is present the output .bam.short file is organized such that a line contains the starting and ending mapping coordinates of both reads.

- Intervals. The output contains a .interval text file where each row represents an interval. It also contains a .interval.links.txt file with the same functionality as in Reads.

- Text table. The output contains a .txt file which is a subset of the initial input and a .links.txt which connects the output entries with the original ones of the source.

- Joined table, which is any table that has been the result of one or more *IntervalJoin*operations. The output products consist of individual entrees of tables of the above basic types and also a .links.txt file which shows what entries of the leftmost entry of the *IntervalJoin*pairs with the rightmost ones.

A.4 Supplemental Figures

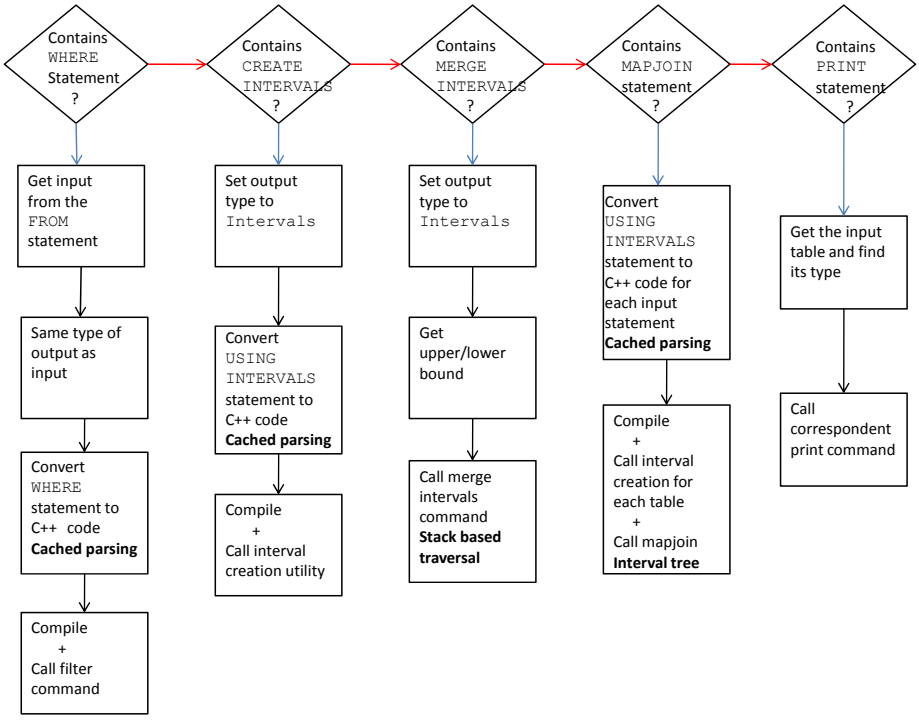


Figure A.1. The query planner. The output of the GQL parser is passed to the query planner which assigns the proper routines of the back end and implements the cached parsing optimization by generating custom C++ files.

Bibliography

[1] Iloprost. http://en.wikipedia.org/wiki/Iloprost.

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

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

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

[5] 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.

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

[7] 1000 Genomes Project Consortium, R.M. Durbin, G.R. Abecasis, D.L. Altshuler, A. Auton, L.D. Brooks, R.M. Durbin, R.A. Gibbs, M.E. Hurles, and G.A. McVean. A map of human genome variation from population-scale sequencing. *Nature*, 467:1061–1073, Oct 2010.

[8] 1000 Genomes Project Consortium, R.M. Durbin, G.R. Abecasis, D.L. Altshuler, A. Auton, L.D. Brooks, R.M. Durbin, R.A. Gibbs, M.E. Hurles, and G.A. McVean. A map of human genome variation from population-scale sequencing. *Nature*, 467:1061–1073, Oct 2010.

[9] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, 2007.

[10] G. J. Annas. HIPAA regulations - a new era of medical-record privacy? *N. Engl. J. Med.*, 348:1486–1490, Apr 2003.

[11] Anonymous2013. File *countoverlapswalker.java* that we shared with the pc chair.

[12] Anonymous2013. File exomehighcnv_gql.txt that we shared with the pc chair.

[13] Anonymous2013. File helloworld_gql.txt that we shared with the pc chair.

[14] 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.

[15] A. Bashir, Y.T. Liu, B.J. Raphael, D. Carson, and V. Bafna. Optimization of primer design for the detection of variable genomic lesions in cancer. *Bioinformatics*, 23:2807–2815, Nov 2007.

[16] J. C. Bellamy. *Digital Telephony, 3rd Edition*. Wiley, 2000.

[17] K. Benitez and B. Malin. Evaluating re-identification risks with respect to the HIPAA privacy rule. *J Am Med Inform Assoc*, 17:169–177, 2010.

[18] D. R. Bentley et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, Nov 2008.

[19] M. C. Brandon, D. C. Wallace, and P. Baldi. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 25:1731–1738, Jul 2009.

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

[21] X. Chen, M. Li, B. Ma, and J. Tromp. DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, 18:1696–1698, Dec 2002.

[22] S. Christley, Y. Lu, C. Li, and X. Xie. Human genomes as email attachments. *Bioinformatics*, 25:274–275, Jan 2009.

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

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

[25] M. A. DePristo et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat. Genet.*, 43:491–498, May 2011.

[26] R. H. Dolin and L. Alschuler. Approaching semantic interoperability in Health Level Seven. *J Am Med Inform Assoc*, 18:99–103, 2011.

[27] M. Dublin. So Long, Data Depression. http://www.genomeweb.com/informatics/so-long-data-depression, 2009.

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

[29] L. Feuk, A.R. Carson, and S.W. Scherer. Structural variation in the human genome. *Nat Rev Genet*, 7(2):85–97, Feb 2006.

[30] M. Gardiner-Garden and M. Frommer. CpG islands in vertebrate genomes. *J. Mol. Biol.*, 196(2):261–282, Jul 1987.

[31] D. Haussler. David Haussler. *Nat. Biotechnol.*, 29:243, Mar 2011.

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

[33] D. Haussler et al. Genome 10K: a proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J. Hered.*, 100:659–674, 2009.

[34] 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.

[35] 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.

[36] K. L. Hudson, M. K. Holohan, and F. S. Collins. Keeping pace with the times–the Genetic Information Nondiscrimination Act of 2008. *N. Engl. J. Med.*, 358:2661–2663, Jun 2008.

[37] A.J. Iafrate, L. Feuk, M.N. Rivera, M.L. Listewnik, P.K. Donahoe, Y. Qi, S.W. Scherer, and C. Lee. Detection of large-scale variation in the human genome. *Nat Genet*, 36(9):949–951, Sep 2004.

[38] 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.

[39] 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.

[40] A. N. Kho et al. Electronic medical records for genetic research: results of the eMERGE consortium. *Sci Transl Med*, 3:79re1, Apr 2011.

[41] J.M. Kidd, G.M. Cooper, W.F. Donahue, H.S. Hayden, N. Sampas, T. Graves, N. Hansen, B. Teague, C. Alkan, F. Antonacci, E. Haugen, T. Zerr, N.A. Yamada, P. Tsang, T.L. Newman, E. Tzn, Z. Cheng, H.M. Ebling, N. Tusneem, R. David, W. Gillett, K.A. Phelps, M. Weaver, D. Saranga, A. Brand, W. Tao, E. Gustafson, K. McKernan, L. Chen, M. Malig, J.D. Smith, J.M. Korn, S.A. McCarroll, D.A. Altshuler, D.A. Peiffer, M. Dorschner, J. Stamatoyannopoulos, D. Schwartz, D.A. Nickerson, J.C. Mullikin, R.K. Wilson, L. Bruhn, M.V. Olson, R. Kaul, D.R. Smith, and E.E. Eichler. Mapping and sequencing of structural variation from eight human genomes. *Nature*, 453:56–64, May 2008.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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*, 25:2078–2079, Aug 2009.

[47] M. Li, J. H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17:149–154, Feb 2001.

[48] 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.

[49] B. Malin, K. Benitez, and D. Masys. Never too old for anonymity: a statistical standard for demographic data sharing via the HIPAA Privacy Rule. *J Am Med Inform Assoc*, 18:3–10, 2011.

[50] 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.

[51] P. M. Nadkarni, L. Ohno-Machado, and W. W. Chapman. Natural language processing: an introduction. *J Am Med Inform Assoc*, 18:544–551, 2011.

[52] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9:680–710, 1984.

[53] T.L. Newman, E. Tuzun, V.A. Morrison, K.E. Hayden, M. Ventura, S.D. McGrath, M. Rocchi, and E.E. Eichler. A genome-wide survey of structural variation between human and chimpanzee. *Genome Res*, 15(10):1344–1356, Oct 2005.

[54] J. Pathak et al. Mapping clinical phenotype data elements to standardized metadata repositories and controlled terminologies: the eMERGE Network experience. *J Am Med Inform Assoc*, 18:376–386, 2011.

[55] D. Patterson. Computer Scientists May Have What It Takes to Help Cure Cancer. The New York Times, December 5, 2011.

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

[57] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[58] U. I. Schwarz, M. D. Ritchie, Y. Bradford, C. Li, S. M. Dudek, A. Frye-Anderson, R. B. Kim, D. M. Roden, and C. M. Stein. Genetic determinants of response to warfarin during initial anticoagulation. *N. Engl. J. Med.*, 358:999–1008, Mar 2008.

[59] A.J. Sharp, Z. Cheng, and E.E. Eichler. Structural variation of the human genome. *Annu Rev Genomics Hum Genet*, Jun 2006.

[60] 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.

[61] T. A. Sivakumaran, R. P. Igo, J. M. Kidd, A. Itsara, L. J. Kopplin, W. Chen, S. A. Hagstrom, N. S. Peachey, P. J. Francis, M. L. Klein, E. Y. Chew, V. L. Ramprasad, W. T. Tay, P. Mitchell, M. Seielstad, D. E. Stambolian, A. O. Edwards, K. E. Lee, D. V. Leontiev, G. Jun, Y. Wang, L. Tian, F. Qiu, A. K. Henning, T. LaFramboise, P. Sen, M. Aarthi, R. George, R. Raman, M. K. Das, L. Vijaya,

G. Kumaramanickavel, T. Y. Wong, A. Swaroop, G. R. Abecasis, R. Klein, B. E. Klein, D. A. Nickerson, E. E. Eichler, and S. K. Iyengar. A 32 kb critical region excluding Y402H in CFH mediates risk for age-related macular degeneration. *PLoS ONE*, 6(10):e25598, 2011.

[62] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[63] P. Stark. Congressional intent for the HITECH Act. *Am J Manag Care*, 16:P24–28, Dec 2010.

[64] E.A. Steven, S. Mccanne, and E. Vetterli. A Layered Dct Coder For Internet Video. In *In Proceedings of the IEEE International Conference on Image Processing*, pages 13–16, 1996.

[65] K. Vandepoele, N. Van Roy, K. Staes, F. Speleman, and F. van Roy. A novel gene family NBPF: intricate structure generated by gene duplications during primate evolution. *Mol. Biol. Evol.*, 22(11):2265–2274, Nov 2005.

[66] F. F. Wagner and W. A. Flegel. RHD gene deletion occurred in the Rhesus box. *Blood*, 95(12):3662–3668, Jun 2000.

[67] J. Ziv and A. Lempel. Compression of Individual Sequences Via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 1978.