Techniques and Data Structures to Extend Database Management Systems into Genomics Platforms

A thesis submitted in fulfilment of the requirements for the
degree of Doctor of Philosophy in the School of Computer Sciences at
The University of Sydney

This research reported in this thesis was supported by the award
of a Research Training Program scholarship to the PhD candidate.

Sidath Randeni Kadupitige
July 2024

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been certified for any other degree or other purposes.

I certify that the intellectual content of this thesis is a product of my own work and that all the assistance received in preparing this thesis and the sources of said assistance have been acknowledged.

Name: Sidath Randeni Kadupitige

Signature: SIDATH RANDENI KADUPITIGE

Date: 5 JULY 2024

Abstract

The recent coronavirus pandemic has shown that there is a great need for data velocity and collaboration between institutions and stakeholders regarding the sequencing and identification of variants in organisms. This data collaboration is hindered by the disparate nature of data schemes, metadata recording methods and pipelines between labs. This could be solved if there was an easy way to share and query data using methods and technologies that are common to most people involved in this field.

This thesis aims to provide a guideline on how to adapt an off the shelf database system into a genomics platform. Leaning on the concept of data and processing co-location, we propose a list of requirements and a prototype implementation of said requirements in the scope of a next generation sequencing (NGS) pipeline. The data and the processes involved can be easily queried and invoked using a commonly known language such as SQL.

Our implementation builds bio-data types and user-defined indexes to develop NGS related algorithmic logic inside a database system. We then leverage these algorithms to build a complete sequencing pipeline - from data loading to consensus sequence generation and variant identification. We also assess each stage of the pipeline to show how effective our methods are compared to existing command line tools.

Acknowledgements

While a thesis maybe submitted on it's own, it is only possible to complete it with the collaboration and support of many people. Those who I would like to acknowledge not only assisted me with completion of this thesis, but with my development as an individual and a researcher.

First off, I would like to thank the tremendous support of my supervisor Uwe Röhm. He went above an beyond in providing support and guidance both with how to approach a thesis and research life in general. My co-supervisor Alan Fekete was also extremely helpful with advice on how to polish the thesis text.

Next I would like to acknowledge the friends and support of the Database Research Group at the School of Computer Sciences. Many of them, especially Chenhao Huang, Pouya Salehpour and Michael Cahill, provided invaluable insight and feedback throughout the duration of my candidature.

I would also like to thank the administrative team of the School of Computer Sciences, particularly Evelyn Riegler and Katie Yang, for the ongoing support and assistance they provided.

Finally, I would like to give a heartfelt thanks and gratitude to my parents for their support during my candidature. I truly could not have completed this without them.

Authorship Attribution Statement

This thesis contains published material in the following publications, in which I am the first and corresponding author:

1. **Sidath Randeni Kadupitige** and Uwe Roehm. ”Using SIMD Instructions to Accelerate Sequence Similarity Searches Inside a Database System”. In ADC 2018. pages 81-93.

My particular contributions to this have been:

- Adapting Farrar’s algorithm for sequence alignment to be used with SIMD implementation in .NET in Chapter 4.
- Benchmarks and evaluation for SIMD implementation in Chapter 5.
- Participating in writing the paper, as the lead author.

Candidate Signature: SIDATH RANDENI KADUPITIGE
Date: 5 JULY 2024

Supervisor Signature: ASSOCIATE PROFESSOR UWE RÖHM
Date: 5 JULY 2024

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Genomics, like many other sciences, is facing a data revolution. The invention of high-throughput genomic sequencing technology has made it possible to sequence genomes in a matter of hours, but at the same time this means gigabytes of genomic data that have to be processed, stored, and shared with many other users for further analysis.

As can be witnessed in recent years with the pandemic from the Novel Coronavirus-19, there is clearly a widespread utility for genome sequencing services. The ability to quickly identify a variant or a viral lineage was of great help in tracking down potential clusters and assisting stakeholders to contain local outbreaks. The ability to manage and share the generated genomic in publicly accessible databases was instrumental in allowing a global response to the pandemic.

While this pandemic provides one of the most evident use cases for genome sequencing, the usefulness of such sequencing services extends far beyond. Using massively parallel and increasingly more cost and time efficient next generation sequencing technologies, researchers and medical personnel are able to find the causes and facilitate treatments and cures for many diseases.

Given the ongoing need for data management and processing, the question arises how a database system can be effectively used as a platform for these tasks? In this thesis, we explore several design alternatives in detail. We further develop a prototypical implementation on the Microsoft SQL Server platform, and we measure its performance and scalability (as well as comparing to performance for some of the other design possibilities, and to existing tools).

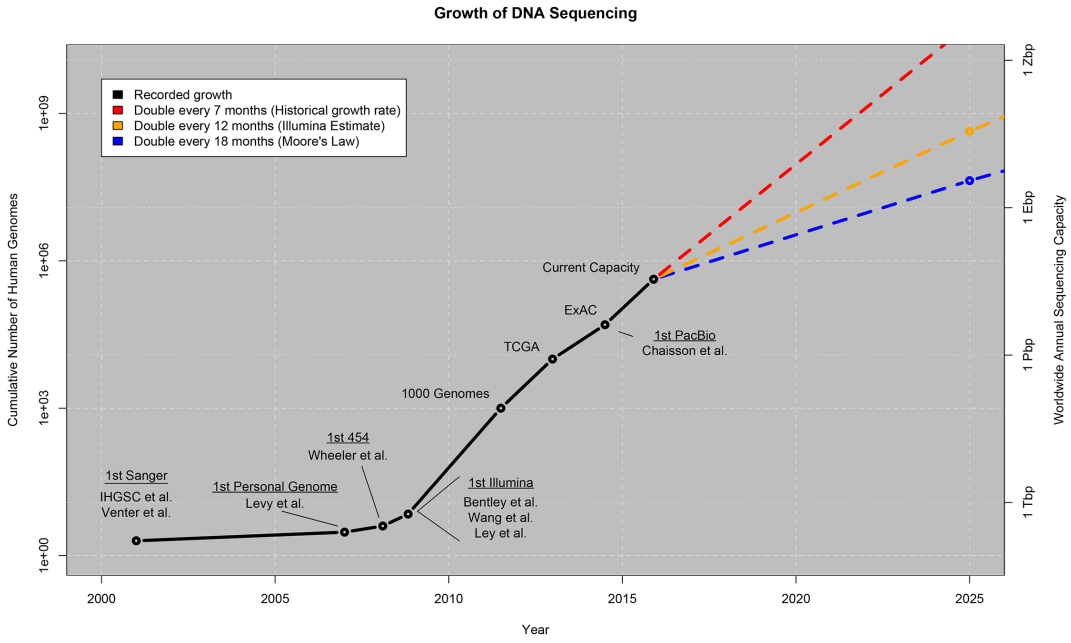

Figure 1.1: Trajectories of sequence growth compared to existing estimates [99]

1.1 Why database support for bioinformatics is required or preferred

With the advent of more and more powerful sequencing and diagnostic machines, the depth and complexity of available biological data has increased many-fold. In computer science, Moore's law is a commonly held standard to describe the increase in complexity of processing power. If the rate of data generation outstrips the rate of processing power complexity increases, then data processing will always lag behind generation unless some algorithmic advances are made. Given this understanding, the most alarming trend is that the amount of genomic data being generated has increased at an exponent on Moore's Law - as seen in Figure 1.1. As such, the primary challenges with dealing with this data are twofold.

Firstly, we must correctly manage this data so that it is correctly annotated and readily available to those individuals who require it, be they biologists in lab, a clinician in a hospital or a patient looking to peruse their medical history. Secondly, we must be able to efficiently analyse this torrent of raw data generated to derive appropriate conclusions. This analysis may present itself in terms

of algorithmic or representation. Since representational analysis (which relies heavily on data visualisation as described in [80]) is based on the initial results from algorithmic analysis, we will focus on the former. With the aim of addressing these two objectives in one stroke, this thesis aims to propose a catalogue of tools and methods that may be used and adapted to both manage the data and carry out popular bioinformatics tasks inside a database engine while the data itself is being generated directly from the source. We will focus in particular on Next Generation Sequencing technologies which generate so much data that it is economically cheaper to place the data on a storage drive and physically ship it to destination labs due to restrictions on information architecture.

1.1.1 Current state of data management for bioinformatics

Most data management options for bioinformatics come in the form of either specialized database engines which require specialized training (such as SciDB [18]) or large flatfile datastores with limited interaction provided by web interfaces (such as GenBank [6]). Ideally, we would like to see programmatic access to these databases to perform analyses (commonly provided by stored procedures or user-defined functions in a database system), however the common approach towards analysis on these platforms is to run a command line tool and provide the output to the user. No options currently exist that provide stored procedures for working directly on the data. While many services exist that provide extensive bioinformatics libraries and operations [16], none of these are inherently coupled with any sort of data management. Most output comes in the forms of flat files - usually CSVs. These inevitably end up in a spreadsheet manager with exceedingly slow data management operation [108].

1.1.2 Why our version of data management would be an improvement

Our approach is to follow a data-centric model where we provide the compute as close to the data as possible. We propose using a database engine as the primary data management and data processing tool.

We propose the following features as our approach:

- Our approach couples both the management and the analysis of bioinformatics data. Data does not need to be moved for analysis to be carried out. In some circumstances, such as sequence analysis, our analysis can even begin before we have a complete dataset as we align each read as it is generated by a sequencing machine and fed into the database engine for storage.

- On the user side - by extending an existing database engine, the training time for new users and maintenance time is much lower as a great wealth of information and tutorials are already available.

- As our approach uses stored procedures which mimic the parameters of command line tools, new users do not even have to learn SQL to carry out their analyses.

- On the development side - As we extend an existing database and not creating a new one, our own development time (and thus error management time) is also greatly reduced as we need only consider the elements of a database engine which are pertinent to our task - i.e. data types, indices, stored procedures and storage.

1.1.3 Next Generation Sequencing

Next Generation Sequencing (NGS, also called high-throughput sequencing or second generation sequencing - with the first generation being Sanger sequencing, and the third generation being non-fragmented sequencing approaches like nanopore sequencing) is a set of technologies that emerged commercially in 2005. It is a massively parallel approach to sequencing based on a method called resequencing.

1.1.3.1 What is Resequencing?

Once a genome for a particular species has been sufficiently characterised (as is the case for human genomes), then it can be generalized and used as a scaffold or pattern by which all specific sequences for that organism can be measured against. The Human Genome Project [25] established a generic sequence for the

species homo sapiens. By using this as a baseline, massively parallel sequencing technologies do not need to sequence large contiguous lengths of any new homo sapiens genome. They just need to sequence a small sufficiently unique sub sequence which can then be aligned against the scaffold genome.

Resequencing, especially with human samples, is one of the most popular applications of next-generation sequencing. It is used to determine the genomic variations of a sample in relation to a common reference sequence. The generated sequence is aligned to the reference sequence and mined for single nucleotide polymorphisms (SNPs) and copy number variations (CNVs) as well as common mutations such as genomic rearrangements and indels.

This application benefits primarily from generating as much sequence data as possible with the smallest budget possible. Longer reads can definitely be beneficial, but the total genomic output (in terms of pure number of bases covered) is more important. Therefore, the flagship platforms from the more established providers tend to be used most heavily in this space.

1.1.3.2 What is the NGS Pipeline?

An NGS pipeline is the process by which an NGS platform arrives at a sequence. While extensive background in provided in Section 2.3, here is a small overview:

- Sample Preparation. This process varies between platforms but for the most part it involves fragmenting the DNA sample, carrying out PCR amplification and adding it to the substrate lanes/gel/beads for beginning the sequencing operation. This process only generates metadata associated with the sample and the platform settings. Any changes made at this stage require the entire pipeline to be restarted.

- Running the Sample. This generates our first accessible data in the form of images (or an electrical signal time series data in the case of nanopore sequencing). This is still preliminary data and is usually discarded once it is processed. Since most of the NGS platform technologies involve fluorescence detection of some kind, these images must be then processed to generate reads.

- Basecalling. The images from running the sample are then fed into a software called a basecaller which derives the first data set we are interested in - the reads. Nanopore sequencers can generate sequences directly based on the strength of the electrical signal for basecalling.

- Short Read Alignment. Once the reads are aggregated, they are aligned against a reference genome. This generates the second data set we are interested in - the alignments.

- Consensus Sequence Generation. Using the aligned short reads, the most common base is pulled at each position to generate the third data set we are interested in - a consensus sequence.

- Further analysis. Once reads, alignments and a consensus sequence have been generated, the task of the sequencing platform is over and the data is sent back to the client for further analysis. The most common of which is:

  - Variant calling. Once a consensus sequence is available, the differences between the consensus sequence and the reference sequence are identified as possible variants.

The current expected pipeline for the above steps is illustrated in Figure 1.2.

1.1.3.3 Hardware Development of NGS Pipeline

As the platforms associated with NGS technology get updated with newer sequencing chemistry, better workflow, higher throughput, and increased parallelism, this is bound to affect the data we are interested in. A key driving force for NGS technology is to decrease the cost of sequencing below the $1000 barrier which is seen as the viability threshold for a diagnostic test [26]. The primary ways to do this are to reduce reagent materials and the number of runs required by a machine to complete a sequence. In turn, the easiest way to reduce reagent costs is to reduce amplification and increase the length of each strand of processed DNA. This will hence result in longer and longer reads until we reach the ideal of DNA nanopore sequencing which would not require fragmentation at all

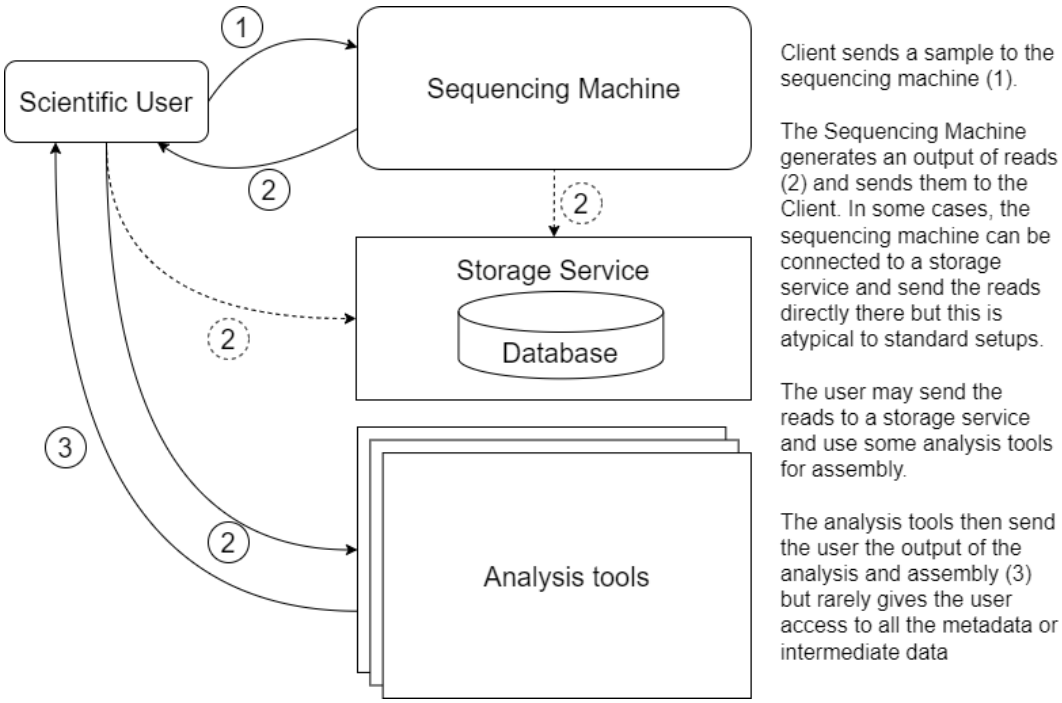

Figure 1.2: A typical NGS sequencing pipeline execution of command-line programs running on flat files

(as described in Section A.5). As such, we must accommodate these expected changes into our project:

- Increased read sizes
- Reduction in read number with a greater reliance on quality.

Given the above trends, if we wish to best affect performance, the best option is to focus on improving short read alignment that improves with an increasing read size (as shown with our dynamic programming approach in Chapter 5.5).

1.1.4 Tasks involved in moving the NGS pipeline into a database

Given that we have three data sets of interest from the NGS pipeline (namely the reads, alignments and consensus sequence), we must find ways to move these into a database engine. If we take the most naive approach and settle on just data management, then we have the following scenario:

1. The sequencer generates reads from the basecaller, we load reads into the database. If we wish to preserve some basecaller information, then we must also have a metadata table for each group of reads as well as a table for the reads themselves. We can use either native DBMS storage and compression options or we can define our own **user-defined data types** or **UDT**s to store the data inside each table. We also need to be able to parse the format in which the reads are presented by the basecaller.

2. An external tool pulls the reads from the database and aligns them to a reference sequence. These alignments are then loaded into a table of their own. We must be able to process the format of the alignments provided by the external tool. Again we can use either native storage or own UDTs.

3. An external tool pulls the alignments from the database and generates a consensus sequence. This sequence and its associated metadata is loaded into tables of their own. Again, the possibility of UDTs arise and again, the format of the data needs to be considered.

In the above scenario we can see several operations relating to data loading and storage which need to be carried out. These are as follows:

1. Identify formats generated by the basecaller and the external tools and write parsers to interpret these formats for data loading

2. Create possible UDTs to store the data

One downside of this approach is evident - as we are using an external tool, we are a level of abstraction away from the data and so we take a performance hit. Another problem is that if any of the tools change or the file format changes, we must alter our database storage and loading procedures. This could result in fracturing our table schemas as new formats may be incompatible with older schemas. So we try a different approach where we load the processing logic into our RDBMS as well:

1. The sequence generates reads from the basecaller, we load reads into the database using parsers and UDTs.

2. We run a stored procedure that aligns the reads and pushes the alignments into a new table as a new UDTs. In order to improve our alignment process, we use an **index**. This can either be the native index provided by the DBMS or a new one we define.

3. We run a stored procedure that takes the alignments and generates a consensus sequence. The generated sequence and associated metadata is then stored in tables of their own using UDTs.

In the above scenario we can see several additional operations from before, these ones mostly relating to data processing, which need to be carried out. These are as follows:

4. Identify formats generated by the external tools and possibly incorporate them as UDTs.

5. Create new stored procedures to carry out the processing logic for short read alignment and consensus sequence generation.

6. Create new indices that can increase the efficiency of the tasks carried out by the stored procedures.

7. Create stored procedures that can parse internal UDTs into formats suitable for external tools in case we wish to use or test them at certain stages in the pipeline.

Now that we have a list of tasks that need to be achieved to provide an NGS pipeline complete database support, we can proceed to design and implementation. We envision our new pipeline diverging from the typical pipeline offered in Figure 1.2, with a new interaction paradigm offered in Figure 1.3.

1.2 Contributions

The contributions of this thesis are:

- I define the design space and, based on this, an approach by which to adapt a database management system into a genomics platform, including

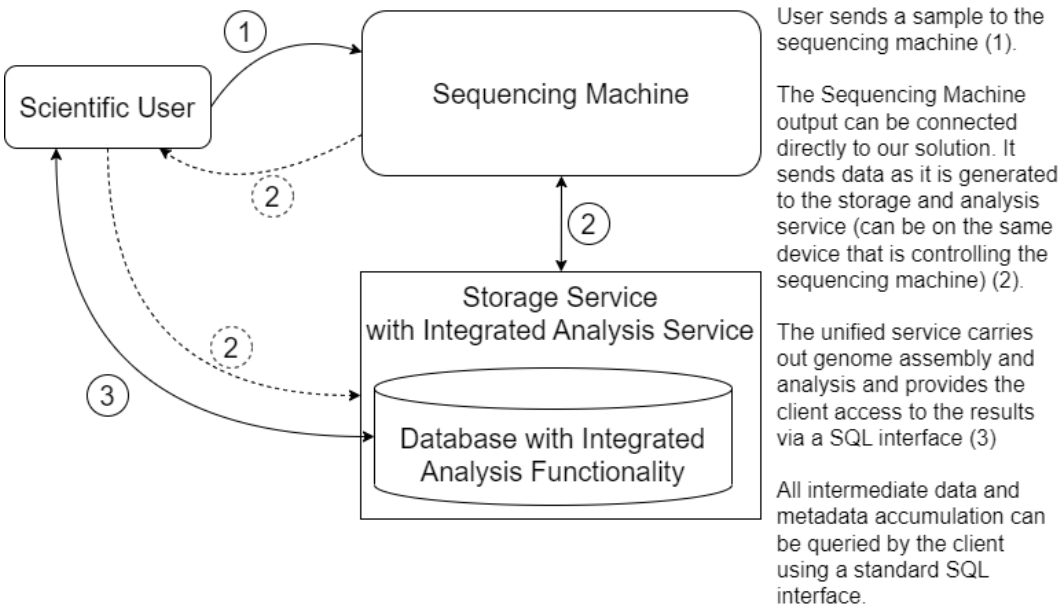

Figure 1.3: A typical NGS sequencing pipeline execution of command-line programs running on flat files

the data types, analysis functionality and indexes required to facilitate this transformation. Our focus is on Next Generation Sequencing pipelines.

- I use the defined approach to develop a sample implementation on Microsoft SQL Server 2018. This involves implementing the data types, indexes and analysis functionality using C#.

- I evaluate the implementation against existing state of the art and common tools. We use Human Genome data (GRCh37.1) for the purposes of this evaluation. We find that our implementation performs better overall for the duration of the pipeline and specifically for consensus sequence generation and single nucleotide polymorphisms identification.

1.3 Thesis Structure

In order to deliver on the contributions, we must provide some background before proceeding with the design, implementation and evaluation. As such, the structure of this document is as follows:

Chapter 2 contains detailed information regarding the theory and technologies used in our implementation:

- An overview of the initial pilot project
- An overview of each major sequence technology from the major vendors (Illumina, Roche, ABI)
- Existing NGS Pipelines
- Data types and file formats involved with NGS related bioinformatics algorithms
- An overview of NGS related bioinformatics algorithms
- Indices that are used to speed up alignment algorithms
- An overview of the development platforms and data sets used for this project

Chapter 3 looks at what design decisions are open to us when considering adapting a database system into a genomics platform.

Chapter 4 goes into the implementation specific details:

- How data loading is implemented, particularly FASTA/FASTQ and SRF
- How we create user-defined indices to assist in alignment tasks
- How short read alignment tasks are implemented
- How dynamic programming is implemented with SIMD accelerated algorithms
- How we generate a consensus sequence
- How we call possible SNPs when we generate a consensus sequence

Chapter 5 shows the performance evaluation of our implementation, along with any optimizations we did. We also compare to existing state of the art command line options.

Chapter 6 edifies our conclusions from our approach and the performance evaluation results of our sample implementation. We also look into possible future work such as how well this code can be adapted to nanopore sequencing and how further optimizations can be made by integrating GPU processing completely into the database engine.

Finally, the Appendix contains extra details that were not included in the main chapters – such as details of technologies from the background (Appendix A) and addition result tables from evaluation (Appendix B).

Chapter 2

Background

In this chapter we will look at existing background material that will assist the adaptation of a database engine into a genomics platform.

Initially we must look at the normal structure of a database engine - which components will be of interest to us and how they work. Once we have a place to start, we look at the core processes involved with where we wish to end up - namely a genomic sequencing pipeline. Subsequently we will look into mapping a genomic pipeline to a database engine and a pilot project that looked at doing this for the BLAST algorithm.

Now that we have some idea of the tasks involved, we will look deeper into the genomics related technologies, particularly those involved with Next generation Sequencing **(NGS)**. This includes NGS platforms themselves, the data types involved with these platforms and the algorithms and indices involved with the NGS pipeline.

As such, we establish a baseline of tasks that need to be accomplished to achieve our goal. Next up we looking into some possible optimisations that can be carried within our pipeline.

Finally we close out this chapter with a list of platforms, technologies and data sets we used to complete this thesis.

2.1 Database Internals

A database engine has traditionally been designed as a storage management system. Data is organised in a structured manner (tables) and improvements are primarily focused on allowing multiple users to retrieve as much data as fast as possible. Here we will look at some of the internals of the database engine to see which areas are of interest to us and why.

2.1.1 Components

A typical database system comprises of components to manage the access of data, to ensure storage of the data and to ensure a record of transactions that have affected the data.

These three functions result in further interactions that create more and more components, but typically we have:

- Storage Engine
  - Buffer Manager
  - Data Access Methods
  - Lock Manager
- Query Processor
  - Query Parser
  - Query Optimizer
  - Query Executor
- Transaction Manager
- Log Manager

2.1.2 Storage Engine

A storage engine deals with how data would be retrieved and stored inside a database system. Data will usually reside in archival storage (typically on disk) and the buffer (typically in memory or a faster disk). Data is usually stored in

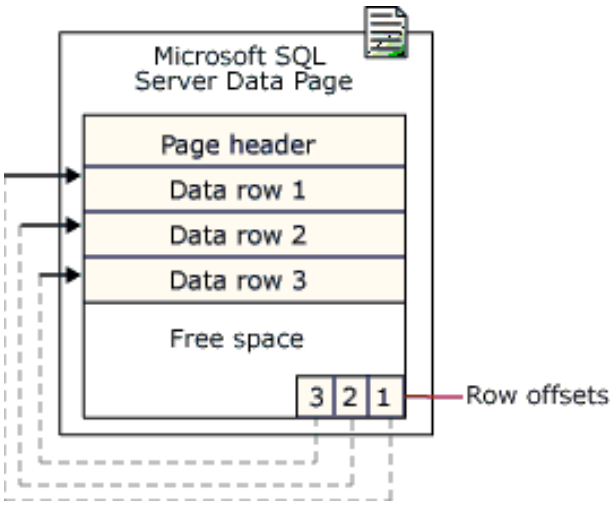

Figure 2.1: SQL Server Page Architecture [95]

files with each file organised via a page structure (an example of SQL Server Page structure can be seen in Figure 2.1) and then sometimes grouped into a higher organisational structure (such as an extent of 8 pages) for efficient access. There are also many specialised pages that keep track of metadata such as free space, object identifiers etc. When talking about performance and efficiency of data access in a database, it is typically referring to the number of page related operations that are necessary. When data needs to be accessed or worked on, it is retrieved from the physical data file and copied into the buffer. The buffer is a fixed size and usually resides in memory, so its occupancy is controlled by a buffer manager that maintains objects in the buffer depending on some sort of schema. This schema is usually determined by freshness of the query asking for the data and the frequency of requests for the data. It is also important to note that data will only be operated on once it is in the buffer.

2.1.3 Database Platforms

In our pilot projects and this thesis, we primarily focused on Microsoft SQL Server. The basis for this was the development advantages of programming in a .NET language for any stored procedure.

2.1.3.1 Microsoft SQL Server

Microsoft SQL Server is a popular database platform that is widely used in enterprise environments. Lately it has also become deployable in both Windows and Linux [70] operating systems. It also enjoys the advantage of allowing its stored procedures to be written in any .NET-based language. This gives it a great deal of flexibility in algorithmic complexity compared to the commonly deployed programming extensions of SQL (such as a PL/pgSQL in PostgreSQL, PL/SQL in Oracle, and even Microsoft's own version - TSQL).

MS SQL Server achieves this by running an instance of its Common Language Runtime (CLR) to manage the execution of this code. Stored procedures, user-defined functions and even user-defined types that interact with the database can be defined in any .NET CLR-compliant language (such as C#) and then called and used through any SQL client that can connect to the server.

2.1.4 File System Access and External Data Wrappers

Scientific data can at times be very large. Sometimes, it would be inappropriate to load that data into a database. In cases like these, it is important to look at existing database technologies that would facilitate better access to this scientific data without having to overly burden the database system. Different database vendors have different systems to allow for storing and or managing file system objects. Usually they use a binary large object (BLOB) data type such as the **lo** in PostgreSQL [86], **BFILE** in Oracle RDBMS [82], **BLOB** in MySQL [83] and **FILESTREAM** in Microsoft SQL Server [69].

2.1.5 Lifecycle of a Typical Query

A query sent to a database engine is typically sent to the query processor where it is first processed by the query parser. This then identifies the tables or objects that the query would need to touch and retrieves statistics from the metadata regarding those tables or objects (such as indices) and constructs a query plan or set of query plans. The optimiser then selects an appropriate query plan and sends it to the query execution engine.

The query execution engine will then initiate a transaction and begin a log entry. It will check the buffer to see if the appropriate data can be found, otherwise it will use data access methods to pull or stream data into the buffer. We will assume for this query that there are no locks or writes on this data and that it is fresh. Once the data is retrieved successfully and the results materialised, the transaction will complete and a log entry will be finalised.

Clearly this is the most simplistic approach and the scenario changes with more users, writes and distributed data. However this approach can be used as a comparative model to the processes we have for a sequencing pipeline.

2.2 Pilot Project

As a pilot project, our research group conducted the dbBLAST project to investigate the suitability of today's database systems for scientific computing, and to identify the possible shortcomings of database engines for those applications [93]. The project concentrated on relational database systems, and as an example application, the basic local alignment search tool (BLAST) for gene sequence databases [2] was chosen.

The central idea is to store the sequence data in a database and to implement the standard BLAST algorithm within the same database systems using stored procedures. Originally, stored procedures have been introduced to database systems as enhanced transaction processing capability that allows to execute a complete transaction including business logic within the database engine. Their main benefits are reduced communication between client and server, and better maintainability of the central code. All major commercial database engines support stored procedures. The traditional stored procedure capability is in the spirit of the SQL/PSM standard, which is basically an extension of SQL with statement grouping, local variables, control flow and condition statements [24; 65]. In addition there is also the possibility of externally defined stored procedures written in a 3GL programming language such as C or in a 4GL language such Java or any language belonging to the common language runtime of .NET.

In the context of scientific computing, stored procedures are interesting because they allow the implementation of data analysis algorithms beyond the

capabilities of SQL queries, and to run those tasks near the data which is to be analysed. In addition to the previously mentioned benefits of code maintainability and improved communication, this would also introduce set oriented processing and declarative querying, both of which are very beneficial in scientific programs.

The main contributions of this pilot project and its corresponding paper [93] are as follows:

- The development of a database-centric version of the BLAST algorithm, called dbBLAST, using stored procedures and an optimised physical design for relational databases; and
- The implementation of dbBLAST within different relational database systems including Oracle 10g, Microsoft SQL Server 2005 and PostgreSQL 8.

In a quantitative evaluation, three dbBLAST implementations were compared with a state of the art, file based BLAST program from NCBI. Based on the experiences and evaluation results, several suggestions can be made for further improvements of database engines to better support scientific database applications. The main results are that the BLAST algorithm could be implemented much faster and short using stored procedure than using a traditional programming language such as C. Also in some particular cases, such as for short query sequence length, the dbBLAST algorithm was faster than NCBI BLAST. This speed increase was a trade off with scalability though as larger inputs were much slower. This scalability issue can be attributed to the limited support for extensive string processing and the weak integration of stored procedures within the query processors in database engines of that time. While the implementation parameters remained the same, there were significant differences between the performance of the various RDBMSs which indicates that each RDBMS approaches table lookups in a different manner. While the scalability of such a solution must be addressed, the initial pilot project dbBLAST did demonstrate the power of declarative querying for scientific computing and how to enrich pure NCBI BLAST searches with access to all the metadata of a stored sequence database.

In dbBLAST, a database-centric version of the popular BLAST algorithm for sequence alignment search was proposed. The approach was based heavily on

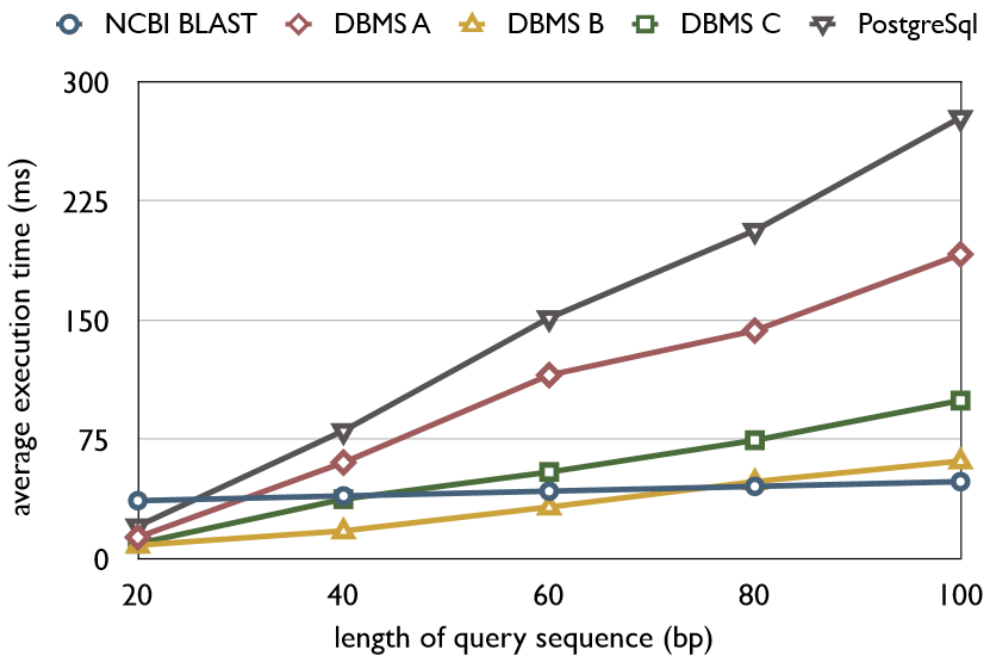

Figure 2.2: dbBLAST and NCBI BLAST comparisons

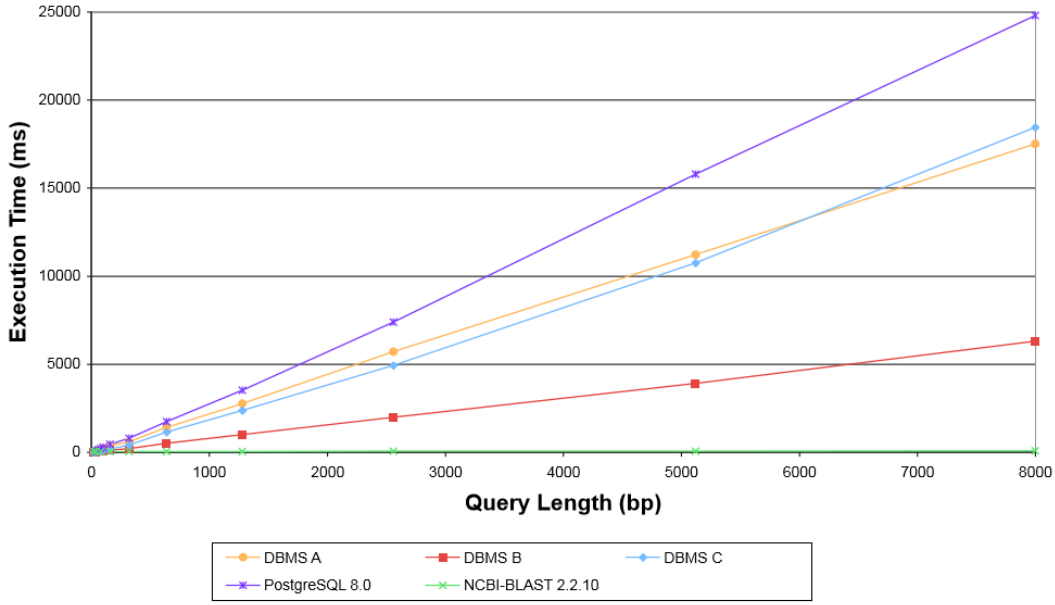

Figure 2.3: dbBLAST and NCBI BLAST comparisons with longer sequence length

leveraging stored procedures capabilities of several relational database engines of the time and optimised physical design with an inverted index over the (segmented) sequence database. The database implementations of dbBLAST were compared against themselves and the file based NCBI BLAST with regards to implementation complexity and runtime performance. For smaller query sizes of up to 60 base pairs, dbBLAST was faster than the file-based NCBI-BLAST as shown in Figure 2.2. However, as the query length increases, the performance of dbBLAST falls compared to NCBI-BLAST as shown in Figure 2.3.

Table 2.1 shows the different development times associated with the various implementations of BLAST. As can be seen, the BLAST algorithm is much faster and more compact using stored procedures in a RDBMS.

	DBMS A	**DBMS B**	**DBMS C**	**PostgreSQL**	**NCBI-BLAST**
Code Length	451	443	549	320	around 33287
Development Time (hours)	28	18	8	11	around 7 years

Table 2.1: Code length and development time of dbBLAST and BLAST implementation

2.2.1 BLAST and dbBLAST Algorithms

The Basic Local Alignment Search Tool, or BLAST, as its more commonly known is one of the most prominent algorithms in bioinformatics. It is a heuristic approach that searches for the best local alignment of a query sequence against a pool of sequences. In NCBI-BLAST, the algorithm is spread into two phases:

1. **Finding Hotspots**. The query sequence is broken down into smaller subsequences or **words** of length k (where k has a default value of 11). These subsequences can also be denoted as **k-mers**. A sequence database is then searched for **hot spots**. Hot spots are local subsequences which are exact matches to these words.

2. **Hot Spot Extension**. For each hot spot found, the alignment comparison is extended in both directions for both the query and database sequences,

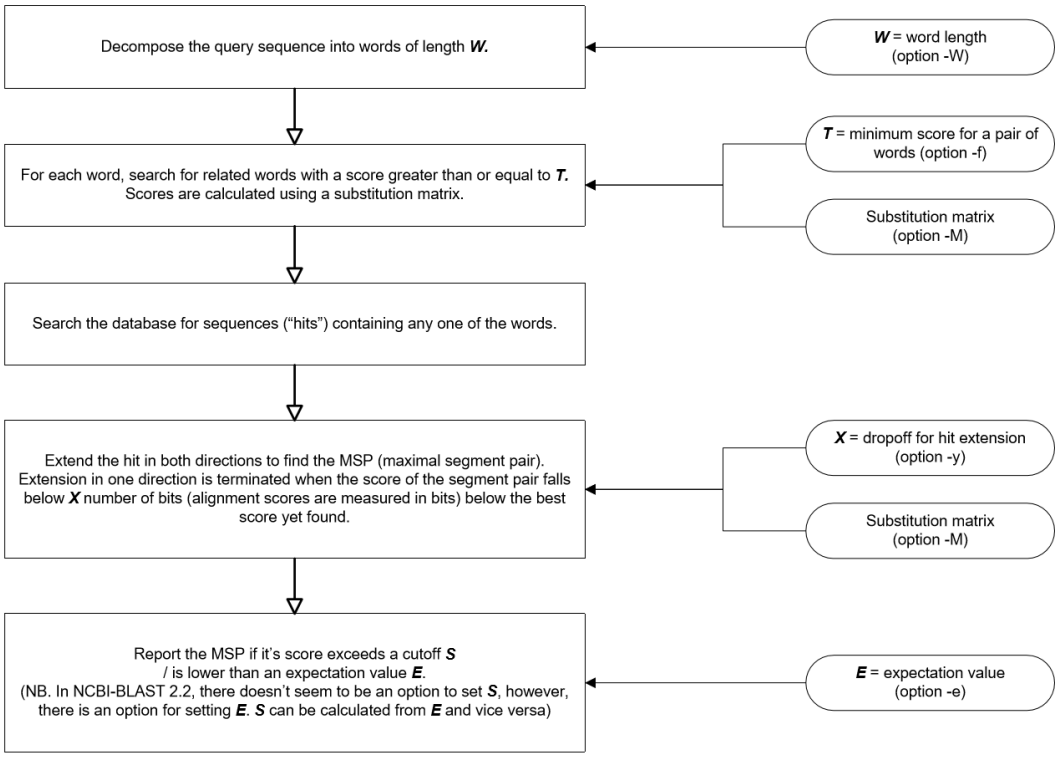

Figure 2.4: BLAST Algorithm

attempting to find a **maximal segment pair** or **MSP**. A MSP is a segment pair with a maximum score over all segment pairs of sequences in comparison. The extension of hot spots in one direction is terminated when either one of the sequences reached its ends, or when the score of the segment pair falls below x number of arbitrary units below the best score yet found. On completions, the algorithm reports all MSPs found during this process with substitution matrix score above a given threshold score s, called **high-scoring pairs** (HSP). This extension step is the most computationally expensive step and accounts for over 90% of NCBI-BLAST's execution time [3].

The BLAST algorithm steps are shown in in Figure 2.4. The parameters used are those for NCBI-BLAST 2.2.

2.2.2 What were the determined factors for creating this functionality inside a RDBMS

Having shown that database functionality could be leveraged to perform the BLAST algorithm, some extra investigations were carried out. These investigations showed that DBMS B (SQL Server 2005) was very effective in short sequence alignment/mapping. As NGS technologies also focused on short sequence alignment, SQL Server was chosen as the platform for further investigation.

2.2.2.1 Data Types

One of the best ways to support bioinformatics algorithms inside a database would be to provide relevant data types to support those algorithms. Inevitably, the best starting point for such a task would be at the sequence level. Hence the first follow on investigation was regarding how best to represent a sequence inside a database. This particular project focused on creating a datatype called **BinSeqData** to store sequence data. This was a UDT written in C# that could be used as a column type in SQL Server 2005. The development took into account the page size of the database engine (8 kB) and developed the sequence storage format with this in mind. The standardized implementation of a basic or **naive** base is used, i.e. an alphabet of { A, C, G, T } represented as 2-bits each: { A:00, C:01, G:10, T:11 }. Then if an unknown base, such as N, appears, it goes through a randomized substitution. Each BinSeqData UDT is represented as:

- constant int32 MAX_SIZE - set to 7986
- byte[] seq_seq
- int32 length
- int32 array_size

2.2.2.2 Indices

The second investigation was into the possible use of an alternative indexing method to the reverse word index. In particular, the use of a suffix tree index [31]. An implementation of database-centric suffix tree construction for

Query Pattern Length	Inverted Word Index (s)	Suffix Index (s)
11	0.01	18
20	0.15	26
40	0.15	48
80	0.3	90

Table 2.2: Comparison of search performance for different indices in dbBLAST [31]

long nucleotides using the TDD algorithm [102] was deployed on SQL Server 2005. A search algorithm for navigating the tree structure to return query pattern matches within the nucleotide sequence. This search algorithm was integrated into dbBLAST. The construction time and scalability of the index was measured against the inverted index, as well as the performance, storage requirements and scalability. As shown in Table 2.2, the results indicate that the search of the suffix tree index suffers because each tree in the index needed to be scanned independently. A single tree would compare favourably to the inverted word index, but the dbBLAST algorithm requires results to be returned for matches in individual GINum sequences. This resulted in multiple tree traversals per search.

2.3 NGS Technology

Next Generation Sequencing technology refers to technology that uses pre-existing knowledge of a generic sequence to identify the sequence of a particular sequence string of DNA, RNA or protein. This use of prior knowledge for sequencing is known as re-sequencing, so all NGS-based machines could be called re-sequencing machines. The premise behind this technology is that if we have sufficient knowledge of a template with which our sequences can be compared *in silico*, then we can remove the cost from the wet lab side of the experimental process.

2.3.1 Main NGS Platforms

There are several major vendors of sequencing machines. Each vendor uses a different biochemical process, which in turn results in different result sets, raw output formats and error rates.
Table 2.3 on page 25 shows a very broad description of most NGS platforms and their various feature sets.

The primary methodology for any sequencing technology is as follows:

1. Prepare the biological sample that contains DNA.
2. Unpackage the DNA as much as possible, usually results in small fragmented portions of the DNA.
3. Attached signed (usually fluorescent) oligonucleotide primers to as many pieces of DNA as possible.
4. Create reads by extending the primers one or more DNA molecules at a time via an endonuclease protein.
5. Read the signed output in some manner (usually a fluorescent frequency).

Each particular NGS technology has some benefits and some detriments. Ironically, the ideal end goal of a re-sequencing technology is to eventually not need to do re-sequencing at all. This means eliminating or minimizing Step 2 listed above as much as possible. As a result of minimizing DNA fragments to that end, we have seen an inevitable increase in read length.

In addition to fluorescent signal detection, newer technologies use hydrogen ion release in step 5 as a method for detecting DNA bases. This methodology leads to drastically longer potential read lengths as less DNA shearing is required. Consequently, the longer reads are easier to assemble against an existing DNA scaffold of a reference sequence.

A detailed dive into the technologies can be found in Appendix A.

Platform	Instrument	Unit	Reads per unit (1000s)	Read size	Read Type	Error Type
Illumina	HiSeq X	Lane	375, 000	300	PE	substitution
Illumina	HiSeq 3000/4000	Lane	312, 500	300	SR & PE	substitution
Illumina	NextSeq 500 High-Output	Run	400, 000	300	SR & PE	substitution
Illumina	NextSeq 500 Mid-Output	Run	130, 000	300	PE	substitution
Illumina	HiSeq High-Output v4	Lane	250, 000	250	SR & PE	substitution
Illumina	HiSeq High-Output v3	Lane	186, 048	250	SR & PE	substitution
Illumina	HiSeq Rapid Run	Lane	150, 696	300	SR & PE	substitution
Illumina	HiScanSQ	Lane	93, 024	200	SR & PE	substitution
Illumina	GAIIx	Lane	42, 075	300	SR & PE	substitution
Illumina	MiSeq v3	Lane	25, 000	600	SR & PE	substitution
Illumina	MiSeq v2	Lane	16, 000	250	SR & PE	substitution
Illumina	MiSeq	Lane	5, 000	250	SR & PE	substitution
Illumina	MiSeq v2 Micro	Lane	4, 000	300	SR & PE	substitution
Illumina	MiSeq v2 Nano	Lane	1, 000	500	SR & PE	substitution
Ion	Proton I	Chip	60, 000	200	SR	indel
Ion	PGM 318	Chip	4, 000	400	SR	indel
Ion	PGM 316	Chip	2, 000	400	SR	indel
Ion	PGM 314	Chip	400	400	SR	indel
PacBio	PacBio RS II	SMRT Cell	47	14, 000	SR	indel
PacBio	PacBio RS	SMRT Cell	22	4600	SR	indel
Roche 454	GS FLX+ FLX	1 PTP	700	700	SR	indel
Roche 454	GS FLX+ FLX	12 PTP	350	700	SR	indel
Roche 454	GS FLX+ FLX	14 PTP	125	700	SR	indel
Roche 454	GS FLX+ FLX	18 PTP	50	700	SR	indel
Roche 454	GS FLX+ FLX	116 PTP	20	700	SR	indel
Roche 454	GS FLX+ FLX	1 PTP	70	400	SR	indel
SOLiD	5500xl W	Lane	266, 667	100	SR & PE	AT Bias
SOLiD	5500 W	Lane	266, 667	100	SR & PE	AT Bias
SOLiD	5500	Lane	81, 500	100	SR & PE	AT Bias
SOLiD	5500xl	Lane	81, 500	100	SR & PE	AT Bias

Table 2.3: List of the most popular NGS platforms

2.4 NGS Whole Genome Analysis pipelines

While NGS pipelines are varied between labs, the basic premise of a sequencing experiment is fairly standard:

1. Acquire the sequenced read data from the sequencing machine.
2. Align sequenced reads to a reference sequence.
3. Generate a consensus sequence from aligned reads.

Steps 1-3 can be considered the primary processes involved with pre-processing and making the sequencing reads ready for analysis, which can involve steps such as:

- Analyze the differences between the reference and the consensus to identify Single Nucleotide Variants (SNV) or Indels (insertions or deletions of multi-nucleotide subsequences).
- Map phylogeny compared to other experiments.

In an attempt to standardize these pipelines, the Broad Institute proposed a series of 'Best Practice Workflows' [106] for particular sequencing and analysis experiments. This initially was based on their Genome Analysis Toolkit (GATK) [20], but extended from there. These pipelines / workflows can be used as a guide for our decisions regarding how to build our own sequencing pipeline.

2.4.1 WARP Pipelines

To facilitate scientific research, the Broad Institute released a new series of workflows that are specifically designed to be deployed on cloud resources. This initiative was labeled as WARP (WDL Analysis Research Pipelines [107]. The idea was to provide containerized environments with the tools required for whole genome assembly and analysis. The pipelines themselves are written as workflows in Workflow Description Language (WDL) and cover experiments such as:

- Germline Variant Discovery
- Genotyping Arrays

- Single-cell Transcriptomics and Epigenomics
- Somatic alignment

While designed for cloud and provisioned on the Terra cloud platform, these pipelines can be used as guidelines for our own project.

2.4.2 Sample Pipelines

Here we can look at some of the sample pipelines that have been provided by the Broad Institute.

2.4.2.1 Common Data Pre-processing for variant discovery

This pipeline is either integrated or carried out prior to almost all of the pipeline examples provided in the 'Best Practice Worflows'. It describes the common tasks of whole genome assembly that we describe at the start of Section 2.4.

First the reads data from the sequencer is mapped to a reference sequence, then a clean-up step of marking duplicates and re-calibrating quality scores is carried out and the final reads are ready to go through assembly and variant calling. Figure 2.5 outlines the process.

A reference implementation of this workflow is also provided at https://github.com/gatk-workflows/broad-prod-wgs-germline-snps-indels. The tools involved are BWA-MEM [53] for aligning reads to the reference sequence, Picar [42] for marking duplicates and GAT [64] for quality score re-calibration.

2.4.2.2 WARP Whole Genome Germline Single Sample Pipeline

This particular pipeline is of the most interest to us as it uses newer technologies and is similar to what we envision as the steps we are likely to require. Figure 2.6 is placed along side the pre-processing pipeline to show the similarities.

This time read alignment to the reference is carried out by the DRAGMAP Aligner that replaces BWA-MEM. It has a similar runtime, but is more sensitive and has a lower false positive rate [12]. Marking duplicates is again handled by GATK.

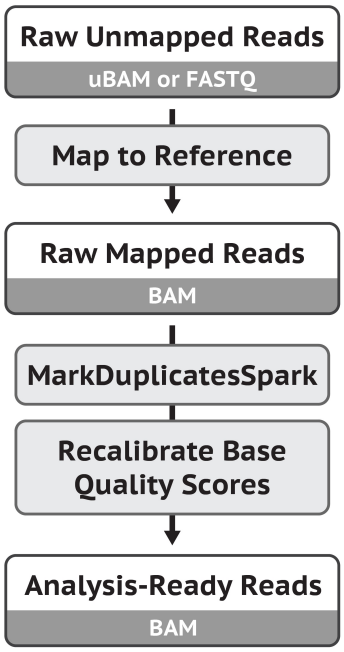

Figure 2.5: GATK Pre-processing pipeline

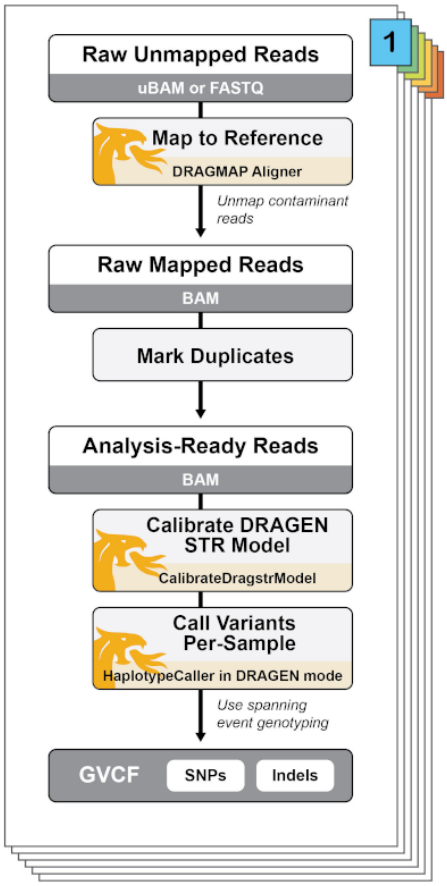

Figure 2.6: WARP Whole Genome Germline pipeline

A few extra steps are now added after the data is 'variant calling ready'. This is the variant calling section which identifies differences between the sample and the reference genome. Here this section is shown as a two step process. First, a model is created based on metrics from the aligned reads, then variants are called using the HaplotypeCaller function in the DRAGEN-GATK toolkit. This new toolkit DRAGEN-GATK [105] is a collaboration between Illumina (the largest sequencing machine vendor) and the Broad Institute to provide some industry standard functionality between vendors and end-users. It is important to note that this process outputs both SNPs and Indels.

2.5 Data Types

In this section we will look at the data types and data formats associated with data generated by next generation sequencing technologies. Our primary focus will be on sequencing data.

2.5.1 Formats for biological data storage outside Native data storage

Most commonly biological data for next generation sequence files comes in formats associated with the vendors of the sequencing technology. Illumina data comes in SRF and FASTQ, Solid data comes in colour space files and 454 data comes in SFF files. After initial processing, all read data usually ends up in a single or multiple FASTQ or binary FASTQ files. Once reads are aligned, the output is in SAM or BAM format to represent sequence alignments. Consensus sequences are again represented as FASTA files and at times with FASTQ files to indicate the confidence in each consensus base.

We can look at other platforms to see what type of file format support is expected. In Table 2.4, a sample file format and parser schema is provided from .NET Bio. Of note is that some file formats, while supported by parsers, can not be generated by the platform.

File Format	Usage	Parser	Output
BED	Sequence Format	Yes	Yes
FASTA	Sequence Format	Yes	Yes
FASTQ	Sequence Format	Yes	Yes
GenBank	Sequence Format	Yes	Yes
GFF	Sequence Format	Yes	Yes
SAM	Sequence Alignment	Yes	Yes
BAM	Sequence Alignment	Yes	Yes
ClustalW	Sequence Alignment	Yes	No
Nexus	Sequence Alignment	Yes	No
Newick	Phylogenetics	Yes	Yes
PhylipParser	Phylogenetics	Yes	No

Table 2.4: File formats that are available for .NET Bio

2.5.1.1 FASTA

FASTA format is a text-based format for representing either nucleotide sequences or peptide sequences, in which nucleotides or amino acids are represented using single-letter codes. The format also allows for sequence names and comments to precede the sequences. The format originates from the FASTA software package, but has now become a standard in the field of bioinformatics.

A sequence in FASTA format is represented as a series of lines, each of which should be no longer than 120 characters and usually do not exceed 80 characters. This probably was to allow for pre-allocation of fixed line sizes in software: at the time most users relied on DEC VT (or compatible) terminals which could display 80 or 132 characters per line. Most people preferred the bigger font in 80-character modes and so it became the recommended fashion to use 80 characters or less (often 70) in FASTA lines. Also, the width of a standard printed page is 70 to 80 characters.

The first line in a FASTA file starts either with a "$>$" symbol or, less frequently, a ";" and was taken as a comment. Subsequent lines starting with a semicolon would be ignored by software. Since the only comment used was the first, it quickly became used to hold a summary description of the sequence, often starting with a unique library accession number, and with time it has become commonplace use to always use "$>$" for the first line and to not use ";" comments (which would otherwise be ignored).

Following the initial line (used for a unique description of the sequence) is the actual sequence itself in standard one-letter code. Anything other than a valid code would be ignored (including spaces, tabulators, asterisks, etc). Originally it was also common to end the sequence with an "*" character (in analogy with use in PIR formatted sequences) and, for the same reason, to leave a blank line between the description and the sequence.

2.5.1.2 FASTQ

FASTQ format is a text-based format for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores. Both the sequence letter and quality score are each encoded with a single ASCII character for brevity [15].

Figure 2.7: An example of a single read FASTQ file [13].

It was originally developed at the Wellcome Trust Sanger Institute to bundle a FASTA sequence and its quality data, but has recently become the de-facto standard for storing the output of high-throughput sequencing instruments such as the Illumina Genome Analyzer.

A FASTQ file normally uses four lines per sequence:

1. Line 1 begins with a ‘ @ ’ character and is followed by a sequence identifier and an optional description (like a FASTA title line).

2. Line 2 is the raw sequence letters.

3. Line 3 begins with a ‘ + ’ character and is optionally followed by the same sequence identifier (and any description) again.

4. Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in the sequence.

A FASTQ file containing a single sequence might look like Figure 2.7.

A quality value Q is an integer mapping of p (i.e., the probability that the corresponding base call is incorrect). Two different equations have been in use. The first is the standard Sanger variant to assess reliability of a base call, otherwise known as Phred quality score:

$Q_{sanger} = -10 \log_{10} p$

The Solexa pipeline (i.e., the software delivered with the Illumina Genome Analyzer) earlier used a different mapping, encoding the odds p/(1-p) instead

of the probability p:

Q_{solexa} = -10 $\log_{10}$ p /(1-p)

Although both mappings are asymptotically identical at higher quality values, they differ at lower quality levels (i.e., approximately p >0.05, or equivalently, Q <13).

2.5.1.3 SRF

Sequence Read Format (SRF) was designed as a generic container format for DNA sequences [9]. It has primarily been used as a container format for short read sequences that are outputted from Next Generation Sequencing (NGS) machines. SRF files store reads as portions of ZTR (while no official expansion for this acronym is provided by the authors, it can be inferred to be Zero-loss Trace Record) files inside their low-level containers called data blocks.

Format is as follows:

1. A SRF data file is comprised of one or more containers.
2. Each container is comprised of several blocks.
3. Each container starts with a Container Header block.
4. This may be followed by an XML block.
5. This is followed by one or more Datablock Header blocks which contains a ZTR header.
6. Each Datablock Header blocks is followed by one or more Data blocks which a single read in the ZTR format.
7. Finally a SRF data file is terminated with a 64-bit unsigned long that designates Index Block Size. If this value is non-zero, then the Index Block Size is preceded immediately by an Index Block.

2.5.1.4 ZTR

ZTR was a format specification designed initially for storing trace or read data from Sanger sequencing experiments [10]. Each ZTR file conventionally contains a single read.

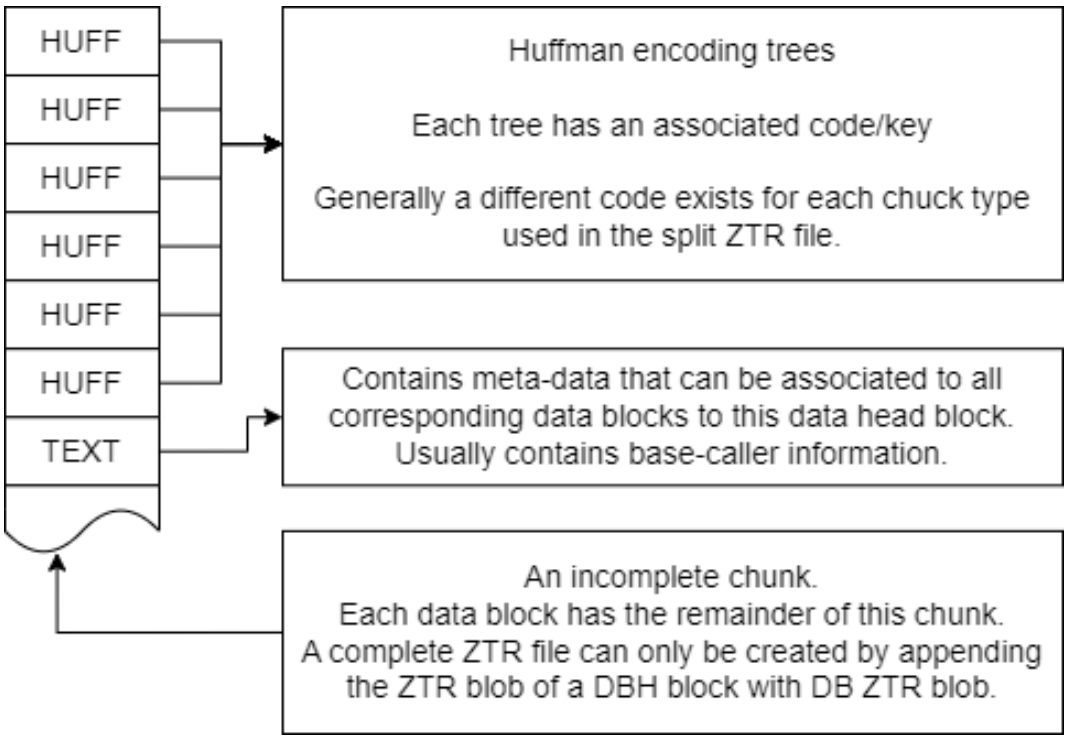

Figure 2.8: ZTR BLOB sample from a Data Block Header (DBH) Block in a SRF file

Format is as follows:

1. Header (1 per file, containing and 8 byte identifier followed by 2 version byte)
2. Chunks (1 or more per header)
   (a) type field (4 bytes)
   (b) Metadata length (4 bytes)
   (c) Metadata
   (d) Data length (4 bytes)
   (e) Data

The different chunk types allow a ZTR file to contain many different types of data, not just sequence data. This is especially useful for metadata storage and provenance information.

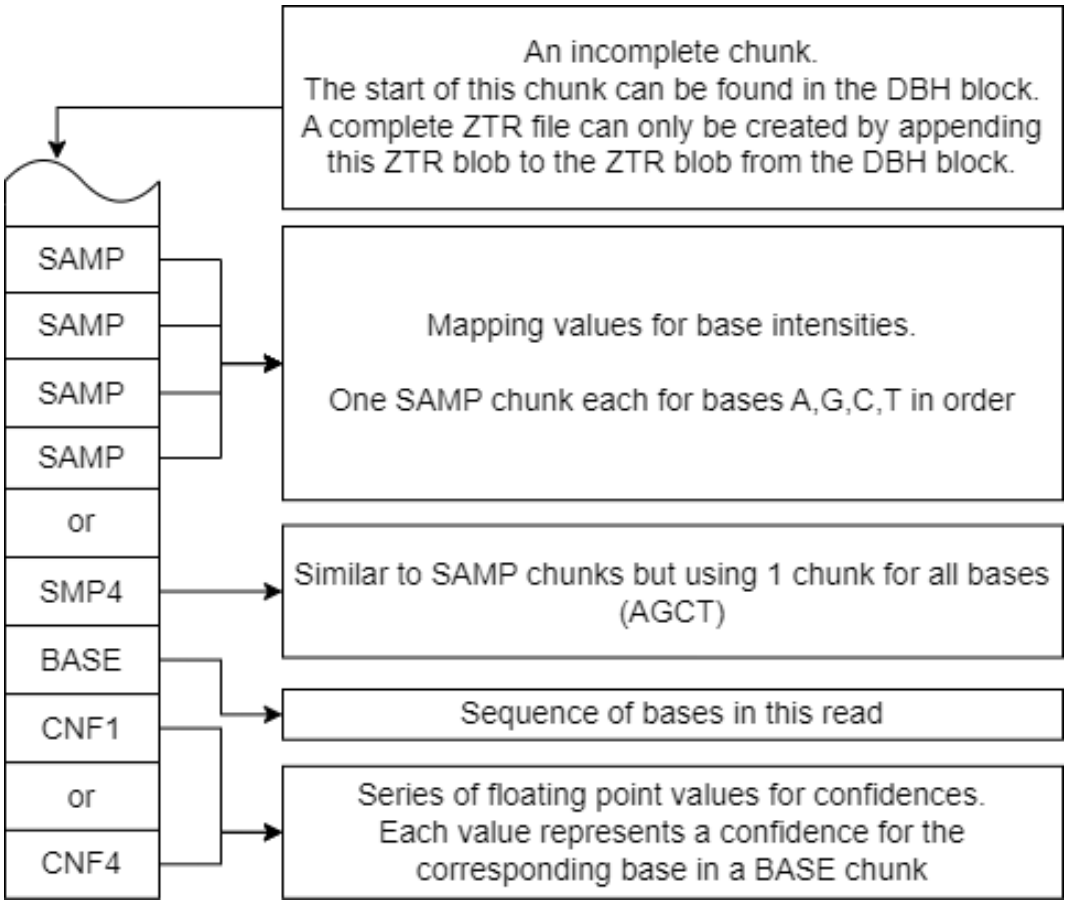

Figure 2.9: ZTR BLOB sample from a Data Block (DB) in a SRF file

1st Nucleotide \ 2nd Nucleotide	A	C	G	T
A	0	1	2	3
C	1	0	3	2
G	2	3	0	1
T	3	2	1	0

Examples:
AA is encoded as 0
CG is encoded as 3
AACG is encoded as 0 1 3

Figure 2.10: CSFASTA 2 adjacent bases encoding matrix

```
>1_88_1830_R3
G32113123201300232320
>1_89_1562_R3
G23133131233333101320
```

Figure 2.11: CSFASTA file example

2.5.1.5 SFF

Standard Flowgram Format (SFF) is a file format for holding trace data for 454 reads typically generated by 454 machines. A SFF file is a container file that begins with a common header component followed by one or more read header and read data component pairs. It can also contain an optional index component. There will be fewer of these file types going ahead as the 454 line of sequencers along with the technology will no longer be marketed as of 2016.

2.5.1.6 Colourspace file or CSFASTA

A CSFASTA file comes in the form of a FASTA file, but the encoding for the sequence line is in the set of { 0 1 2 3 } where each number represents a pair of adjacent base calls with the encoding scheme as defined by the matrix in Figure 2.10. Usually an initial primer base of 'G' is provided to allow for error checking for the first two-base call number as seen in the example of file in Figure 2.11.

2.5.1.7 SAM

SAM stands for Sequence Alignment/Map format [55]. It is a TAB delimited text format consisting of an optional header section, and an alignment section. If present, the header must be prior to the alignments. Header lines start with @, while alignment lines do not. Each alignment line has 11 mandatory fields for

```
@HD VN:1.5 SO:coordinate
@SQ SN:ref LN:47
ref    516 ref  1  0 14M2D31M   *  0   0 AGCATGTTAGATAAGATAGCTGTGCTAGTAGGCAGTCAGCGCCAT *
r001    99 ref  7 30 14M1D3M    = 39  41 TTAGATAAAGGATACTG *
*      768 ref  8 30 1M         *  0   0 *                 * CT:Z:.;Warning;Note=Ref wrong?
r002     0 ref  9 30 3S6M1D5M   *  0   0 AAAAGATAAGGATA    * PT:Z:1;4;+;homopolymer
r003     0 ref  9 30 5H6M       *  0   0 AGCTAA            * NM:i:1
r004     0 ref 18 30 6M14N5M    *  0   0 ATAGCTTCAGC       *
r003  2064 ref 31 30 6H5M       *  0   0 TAGGC             * NM:i:0
r001   147 ref 39 30 9M         =  7 -41 CAGCGGCAT         * NM:i:1
```

Figure 2.12: SAM file example

essential alignment information such as mapping position, and variable number of optional fields for flexible or aligner specific information (cf. Figure 2.12).

Each header line begins with the character @ followed by one of the two-letter header record type codes defined in this section. In the header, each line is TAB delimited and, apart from @CO lines, each data field follows a format TAG:VALUE where TAG is a two-character string that defines the format and content of VALUE. A list of predefined header tags can be found at the samtools website [52].

The 11 mandatory fields for each alignment line are described in Table 2.5. These fields always appear in the same order and must be present, but their values can be 0 or * (depending on the field) if the corresponding information is unavailable.

Col	Field	Type	Regexp/Range	Brief description
1	QNAME	String	[!-?A-]1, 254	Query template NAME
2	FLAG	Int	[0, 216-1]	bitwise FLAG
3	RNAME	String	—[!-()+-<>-][!-]*	Reference sequence NAME
4	POS	Int	[0, 231-1	1-based leftmost mapping POSition
5	MAPQ	Int	[0, 28-1]	MAPping Quality
6	CIGAR	String	—([0-9]+[MIDNSHPX=])+	CIGAR string
7	RNEXT	String	—=—[!-()+-<>-][!-]*	Ref. name of the mate/next read
8	PNEXT	Int	[0, 231-1]	Position of the mate/next read
9	TLEN	Int	[-231+1, 231-1	observed Template LENgth
10	SEQ	String	—[A-Za-z=.]+	segment SEQuence
11	QUAL	String	[!-]+	ASCII of Phred-scaled base QUALity+33

Table 2.5: 11 Mandatory Fields in non-header each line for a SAM file

2.5.1.8 BAM

BAM stands for Binary SAM format. It is designed as a compressed SAM file that has an associated index and is designed for random access. The format uses BGZF compression which is an extension of the GZIP file format. The samtools website [52] contains a complete and exhaustive explanation of the BAM format. As the information contained within the format is the same as its SAM counterpart, the format exists purely for ease of machine readability and efficient storage.

2.5.1.9 CRAM

The Compressed Reference Alignment/Map format (CRAM) format is a binary format that is an extension of the BAM format. It is backwards compatible, but has several features that are designed the allow it to store data in various compressed schemes other than the original BGZF/GZIP compression schemes of the BAM format. In particular, it reduces space by storing reads using reference differences [40] instead of bases. A reference difference read only stores the differences from read to reference instead of the all the bases of the reads. This can result in a large reduction of storage space for reads that are similar to the reference. The condition of this type of read storage is that a reference sequence must be provided for an alignment or consensus generation to occur - however this is not so much of a downside as most read alignment and mapping programs will need a reference sequence somewhere in their pipelines regardless of this requirement.

2.5.1.10 VCF

VCF (Variant Call Format) is a text file format. It contains meta-information lines, a header line, and then data lines each containing information about a position in the genome. It is used to describe where possible variations to the reference genome may occur. An example is shown in Figure 2.13. This is also the normal output file for variant/snp and consensus calling from **samtools mpileup** option.

```
##fileformat=VCFv4.0
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=1000GenomesPilot-NCBI36
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=.,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS     ID        REF ALT    QUAL FILTER INFO                              FORMAT      NA00001
     NA00002        NA00003
20     14370   rs6054257 G      A       29   PASS   NS=3;DP=14;AF=0.5;DB;H2           GT:GQ:DP:HQ 0|0:4
8:1:51,51 1|0:48:8:51,51 1/1:43:5:.,.
20     17330   .         T      A       3    q10    NS=3;DP=11;AF=0.017               GT:GQ:DP:HQ 0|0:4
9:3:58,50 0|1:3:5:65,3   0/0:41:3
20     1110696 rs6040355 A      G,T     67   PASS   NS=2;DP=10;AF=0.333,0.667;AA=T;DB GT:GQ:DP:HQ 1|2:2
1:6:23,27 2|1:2:0:18,2   2/2:35:4
20     1230237 .         T      .       47   PASS   NS=3;DP=13;AA=T                   GT:GQ:DP:HQ 0|0:5
4:7:56,60 0|0:48:4:51,51 0/0:61:2
20     1234567 microsat1 GTCT   G,GTACT 50   PASS   NS=3;DP=9;AA=G                    GT:GQ:DP    0/1:3
5:4       0/2:17:2       1/1:40:3
```

Figure 2.13: VCF file example

2.5.2 Formats for biological data in current DBMS scenarios

All publicly available DNA sequence data can be found at the **International Nucleotide Sequence Database Collaboration** or **INSDC** [44]. This initiative is a daily cross-collaboration between the NIH's **National Centre for Biotechnology Information** or **NCBI**, the **DNA Database of Japan or DDBJ** and the **European Molecular Biology Laboratory - European Bioinformatics Institute** or **EMBL-EBI**. This resource provides a web portal for queries against the collective stored information in these repositories. Direct access to the stored data is provided in form of FTP access and Aspera access to flat files which can be used with various provided tools to create local databases. Submissions follow a strict guideline policy as directed by the Nucleotide Sequence Database Policy [11]. Direct DBMS access to this data is not provided. The most accessed data repository is **GenBank** [5]. GenBank is the annotated collection of all publicly available DNA sequences in the INSDC. The flat files are available in two formats:

- **Annotated Flat File records** These comprise of a series of descriptor fields separated by newlines and finally the sequence represented as a FASTA format. Typically, the GINum field found in these files is used as a primary key when inserting the information into a DBMS.

- **Abstract Syntax Notation 1** or **ASN.1** This is an ISO data representation formation used to achieve interoperability between various platforms. It is a computer readable text-based flatfile and is used as a representation format throughout all out NCBI, not just GenBank [84].

2.5.2.1 What file formats/ schemas exist currently in databases storing biological data

As Genbank provides data as a flat file, the NCBI provided tools use basic data types such as integers and strings. For example, the GINum becomes an integer, and the sequence data is inputted as a string.

2.5.3 Existing complex data types for representing biological data inside a DBMS

As commercial databases do not have integrated data types for biological data, the most common other representation apart from integers and strings is a binary array representation for sequence data in a standard 2-bits for a single base representation: **{A: 00, C: 01, G: 10: T: 11}**. The pilot project in chapter 2.2.2.1 looks at implementing a segmented sequence using the above representation in the form of the BinSeqData UDT.

2.6 Algorithmic tasks common to Next Generation Sequencing

The two algorithmic tasks most common to NGS pipelines are sequence alignment and variant identification.

2.6.1 Sequence Alignment

Sequence alignment is the most common algorithm in bioinformatics. It is essentially a string search of a pattern p against a single document or collation of documents S. This is usually achieved by indexing S and carrying out searches for p. Common terms associated with sequence alignment are described in Table 2.6.

Term	Meaning
Edit distance	The number of total character edits required to change a partial match of p against S to an exact match
Indel	An insertion or a deletion of a character to change a partial match of p against S to an exact match
Mutation	A change of a single character of a partial match of p against S to an exact match

Table 2.6: Common terms associated with sequence alignment

2.6.1.1 Aligning short sequences to a larger Reference sequence

Usually a complete pattern p is not matched against S. Most searches are similar to the algorithmic structure of BLAST search, i.e

1. p is broken into short sequences of length $pindex$.
2. Each of these subsequences are then compared against an index of S for exact matches.
3. Each match of these subsequences is called a candidate region.
4. If a sufficient number of consecutive candidate regions are matched over a threshold t regions, then p is considered to be a match of S.

2.6.1.2 Multiple Sequence Alignment

Another common application of sequence searches is multiple sequence alignment. This application is very common in NGS technologies as there are many instances of small length reads or patterns being matched to a very large search space of S. Multiple sequence alignment is commonly defined as one of two methods

- Unbound (No framework or reference sequence)
- Bound (all aligned against a reference sequence)

In most scenarios for multiple sequence alignment in NGS, many sequences are aligned in a bound fashion, then once they have all been aligned to the reference, they are collated to generate a consensus sequence. The changes between the reference genome and the consensus sequence are identified as possible variants. In a population sample, if these variants were commonly present, we could identify them as SNPs. For the purposes of this thesis, we will treat all possible mutations as potential SNPs.

2.6.1.3 Indices

In order to speed up sequence alignment, various indices have been developed to help in this regard. Most of these indices are adapted from the text mining field

with adaptations for the limited alphabet and highly repetitive regions found in biological sequence data. These most commonly used indices are discussed in Section 2.7.

2.6.1.4 Error models

While exact searches are easy to carry out, inexact searches provide the greatest utility for analysis purposes. As such, a robust error model allows for better matches and less false negatives in sequence alignment. It also allows for the identification of SNPs and other important mutations that are biologically relevant. Most error models allow for a certain degree of edit distance difference between p and S. The most robust error models allow for both indels and mutations. This usually comes at a computational cost that is an order of magnitude more expensive than the initial exact search itself, so a great deal of effort goes into making heuristic determinations of $O(n)$ complexity so as not to impact performance.

2.6.1.5 Examples of Sequence alignment algorithms

Below is a list of common sequence alignment algorithms

- Local Alignment tools
  - NCBI-BLAST
  - Dynamic Programming/ Smith-Waterman
- Sequence Assembly tools
  - Sanger Sequence Alignment
  - Short Read Alignment

2.6.2 Multiple Sequence Alignment / Consensus sequence generation

Some current popular tools for Multiple Sequence Alignment (MSA) and Consensus sequence generation:

- Clustal Omega
- Kalign
- MAFFT
- MUSCLE
- TCoffee

While these tools were not initially designed for consensus generation, they have become widely used for that purpose. As a result - many packages that such as samtools and GATK have added this functionality.

2.6.3 Variant Identification

The paper by Mielczarek and Szyda in 2015 [72] gives a good overview of current variant identification steps:

Thus far, genotypes of SNPs were typically identified in microarrays. Recent advances of NGS and complementary analysis programs have provided a new possibility to identify a higher number of variants. While array density is limited to thousands of polymorphisms, sequencing entire genomes potentially allows for the discovery of all existing polymorphisms (in particular, for the genomic regions covered by the reference sequence). Consequently, it allows for identifying not only common, but also rare SNPs, the importance of which for the determination of complex phenotypes has been recently stressed [36]. Moreover, thanks to the availability of all existing polymorphisms, we no longer need to rely on the linkage disequilibrium or recombination rate in candidate gene mapping, since causal mutations are present in SNP panels originating from NGS. However, for a reliable variant detection, a high depth of coverage is a prerequisite, since a high number of reads aligned to each base is used to differentiate between sequencing errors and true polymorphisms. Many algorithms and software packages have been dedicated to identify SNPs in NGS data. When only a single genome is analysed, SNP calling and genotype calling are similar, because heterozygous or homozygous non-reference genotypes imply the presence of an SNP. In the simultaneous analysis of multiple samples, an SNP is identified if at least one individual is heterozygous or homozygous for a nonreference allele. Therefore,

SNP calling may be defined as the process of identifying sites differing from a reference sequence, while genotype calling refers to the estimation of genotypes [77]. SNP and genotype callers are based on either heuristic or probabilistic methods.

2.6.3.1 Heuristic methods

Heuristic methods call variants based on multiple information sources associated with the structure and quality of data. Due to their computational demands, they are less commonly used than probabilistic methods. The heuristic approach is partly adopted in VarScan2 [46]. VarScan2 identifies variants using a heuristic method, as well as a statistical test based on the number of mapped reads covering each allele. For variant detection, a heuristic algorithm determines the genotype based on thresholds for coverage (minimum 33), base quality (minimum 20) and variant allele frequency (minimum 0.08). After that, Fisher's exact test of read counts supporting each allele is applied compared to the expected distribution based on sequencing error alone.

2.6.3.2 Probabilistic methods

Probabilistic methods provide measures of statistical uncertainty for called genotypes, making it possible to monitor the accuracy of genotype calling. Moreover, additional information concerning allele frequencies and linkage disequilibrium patterns may be included in the analysis [77]. Genotype calling is based on genotype likelihood calculations and adopts Bayes' theorem. After preprocessing steps comprising realignment and quality score re-calibration, the next step involves likelihood calculation for each possible genotype at each base (a homozygote for the reference allele, a homozygote for the alternative allele or a heterozygote). It is based on quality scores and allele counts from the reads at the SNP site. In the Bayesian framework, the computed likelihood is combined with a prior genotype probability, which leads to a posterior probability of a genotype. As a result, the genotype with the highest posterior probability is chosen. The ratio between the highest and the second highest probabilities may be used as a measure of confidence. A uniform prior probability may be selected as equal for all genotypes; alternatively, a non-uniform prior can be pre-imposed

based on additional information, such as dbSNP (SNPdatabase) entries, the reference sequence structure or features provided by the analysed sample. SAMtools is an example of software incorporating the probabilistic approach [55]. In particular, **samtools mpileup** collects information stored in input BAM files and computes the likelihood of data given for each possible genotype. Next, **bcftools** applies the prior probability and performs the actual SNP calling [55]. Other examples of programs based on the Bayesian principle are GATK, AtlasSNP2, SOAPsnp and SNVer [109].

2.6.3.3 Single-sample vs. multi-sample SNP calling

A customary approach is to identify polymorphic variants based on a comparison of individually sequenced genomes with the reference. Although such an approach is ideal to assess individual polymorphisms usually expressed by SNP alleles, indels and copy number variants (CNVs), it is difficult to draw inferences on a population-wide level and it requires custom-written programs. This is because sites homozygous for a reference allele are not stored on an individual basis, even if they are polymorphic in other sequenced individuals, which prevents the calculation of population-wide allele and genotype frequencies. Multi-sample calling involves a simultaneous identification of variants in several individuals, but it is much more CPU time- and resource-consuming than individual variant calling, it requires all the samples to be provided at once (which may pose a challenge in large, collaborative sequencing projects) and is not available in all software tools. Multi-sample calling is provided e.g. by SAMtools and GATK. A major advantage of multi-sample calling comprises the fact that priors may be improved based on the information on allele frequency and on verification of whether the obtained genotypes follow the Hardy-Weinberg equilibrium [77]; the latter, however, is only useful if unrelated individuals are considered.

2.7 Indices

In databases engines, an index is created on a table in order to facilitate faster access of records. Similarly, a sequence alignment program might create an index of the input sequences andor reference sequences to facilitate faster alignment.

It is this sequencing index to which we will refer in this document.

2.7.1 What are the typical indices used in sequence alignment

There are three key types of indices used in sequence alignment. The first and most prominent is a hash index. Usually in the form of a hash table or a hash index, this index maps the output of a hash function applied to a given character string to a set of locations in the input or reference files. Sometimes this is based on the input data, but most commonly it is based on the reference data. The second most common index is a reverse word index. Similar in function to a hash table, it keeps the full text of a string instead of the hash function as its key. Finally we have suffix array based indices which typically only index the reference files.

2.7.1.1 Hash Table

A hash table is an associative array structure that is based on a key:value system. The structure is split into groups called buckets and any key:value pair has its key mapped to a given bucket by a mapping function called a hash function. The value (or the key:value pair) is then stored in the mapped bucket. The strength of this data structure is heavily reliant on the effectiveness of the hash function to map each unique key to a unique bucket as seen in Figure 2.14. When the hash function maps two or more unique keys to the same bucket, then it is called a hash collision. In a perfect hash table, no collisions occur and all buckets are used exactly once. While it is possible to construct such a table if all keys are known, in most instances collisions will occur and a collision handling method is required. The most common method to make each bucket into a linked list of items as seen in Figure 2.15. The computational complexity of hash tables is described in Table 2.7. As the number of entries rises, the chance for collisions increases and thus the efficiency decreases as more operations to find a specific value. This level of efficiency is characterised by a term called the *load factor* or l which is a measurement of the number of entries over the number of buckets. Most collision handling methods will give a certain performance guarantee while

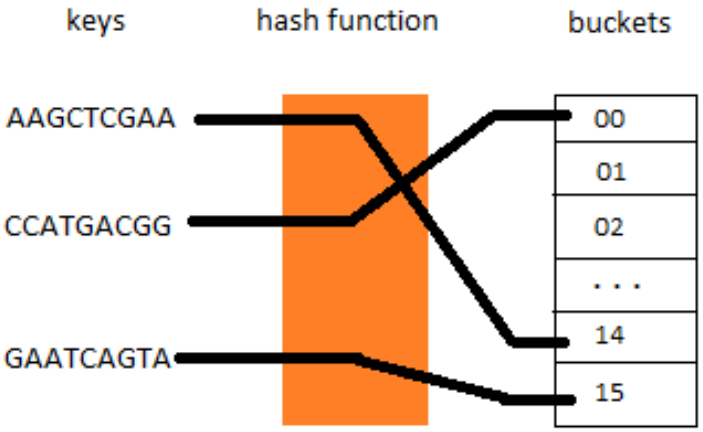

Figure 2.14: Hash Table example

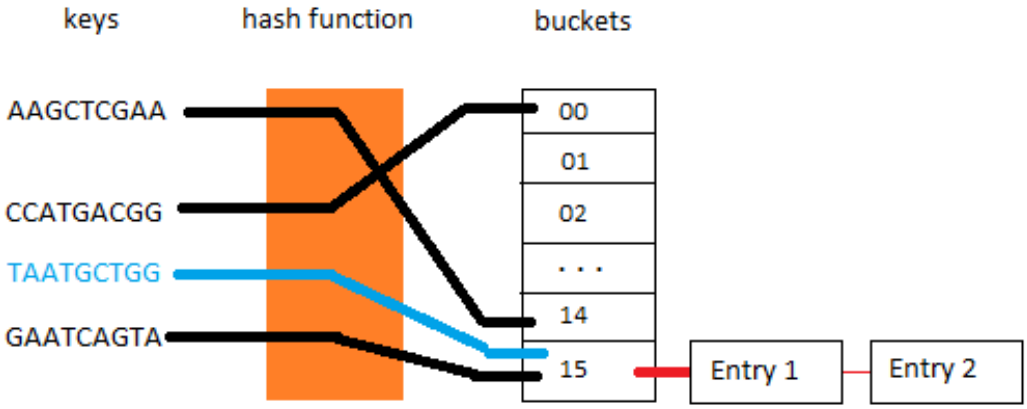

Figure 2.15: Hash collision linked list example

$l <$ some threshold k. Once the k is passed, the hash table itself must be reorganized for a greater number of buckets to ensure efficient lookups.

This data structure is used in NCBI-BLAST [63] and early NGS sequence alignment tools such as MAQ [54] and SOAP [56] for indexing the reads. It has most recently been used in newer NGS sequence alignment tools such as WHAM [57] and SNAP [111] to index the reference sequence. In the field of bioinformatics, an ideal hash function for sequences would map similar sequences to the same bucket.

Metric	Average	Worst case
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Table 2.7: Compute complexity of a Hash Table

2.7.1.2 Spaced-Seed Reads-based Hash Index

The most prominent example of this method is used in the tool called MAQ (while the authors do not officially explain the acronym, it probably stands for Mapping Alignment with Quality) [54]. MAQ can build sequence assemblies by employing read mapping and estimate error probability for the genotype calls, including short indels and SNPs by using quality scores and mate pair information (a mate-pair read is when a read is formed from sequencing two non-contiguous pieces of DNA that are fixed distance apart). MAQ has two stages,

- the alignment stage; and
- the SNP calling stage.

At the alignment phase, MAQ first scans the reference genome sequence and indexes read sequences to identify those potential read alignments. A read is mapped to the position with the minimum threshold sum of quality values according to the mismatched bases. To find the best read alignment, MAQ builds multiple hash tables to ensure that a sequence with no more than two mismatches will be hit. Each hash table corresponds to a non-contiguous seed templates consisting of 1 or 0, where bases at 1 will be indexed. After the read indexing, the reference will be scanned and hashed base by base through the same templates used in the indexing. By looking up in the hash tables, if a potential alignment is found then MAQ will extend out the alignment from the seed without gaps and calculate the sum of qualities of mismatched bases. MAQ then integrates the coordinate of the alignment and the read identifier as a score. If a read has multiple highest scores with different positions on the reference, only one position will be picked randomly. If the paired end reads are available, MAQ keeps a queue to record the last two best alignments of a

read on the forward strand end. When a potential alignment is found on the reverse stage, MAQ looks up the queue storing the mate reads to check whether an alignment exists on the forward strand. The mapping quality is measured as the Phred-scaled probability [27; 28]. MAQ finds short indels by utilising the Smith-Waterman gapped alignment [97] and the mate pair information, given a read pair if one end can be mapped to the reference but the other end cannot. A potential explanation is that an indel interferes the alignment of the unmapped read. Smith-Waterman gapped alignment then can be used to locate these indel regions. At the SNP calling phase, MAQ generates a consensus genotype sequence from the alignments that are inferred from a Bayesian statistical model. By assuming that the sample is diploid by default, MAQ calls the genotype that maximises the posterior probability of genotypes. The likelihood that the consensus genotype is incorrect is measured by Phred score. Potential SNPs can be easily detected by comparing the consensus sequence to the reference. Simulation and read data experiments suggest that MAQ is able to estimate error probabilities of alignments and consensus genotypes accurately, which provide the foundation for the high quality of indels and SNPs [54].

2.7.1.3 Extended Seed Reference-based Hash Index

A good example is in the SNAP program [111]. SNAP's index is a hash table from seed strings of a fixed length s to lists of positions in the reference database where those strings occur. It differs from the indices in BLAST in two ways:

- Longer seeds – SNAP indexes longer seeds than read indexing tools, where s is typically around 20 compared to 10-12 used in BLAST. Longer seeds mean that it is less likely for a seed extracted from a read to exactly match the reference database, e.g., with a sequencing error rate of 2%, the probability that a 20 bp seed is error free is 0.98 to the power of 20 - which is 0.67. This longer seed also reduces the number of false positive seed matches. For example, given the human genome is approximately 4^{16} bp, we expect a seed of length 10 (as seen in BLAST) to match $4^6 = 4096$ locations just by chance, resulting in many unneeded local alignment tests. In contrast, with a seed of length 20, the expected number of hits by chance is $4^{-4} = 0.0039$. This was impractical in initial reads since they were 35

bp in length and an error in the middle would not allow for an s of 20. However now that we have reads of 100 bp or larger, it is possible to draw enough independent long seeds to have a chance of one being error free. For example, from a 100 bp read, up to 5 non overlapping 20 bp seeds can be drawn. Again assuming a 2 percent error rate, the chance that any once seed seed has an error is $1 - 0.67 = 0.33$. However the chance that all five seeds have errors is only $0.33^5 \approx 0.004$.

- Overlapping Seeds – To save memory, many hash based aligned index only non overlapping $s$mers of the reference sequence, so that with a seed length of s, only $1/s$ of the locations in the sequence occur in the index. For example, an index over the human genome with 20 bp seeds would take about 2 GB of memory with non overlapping seeds. However, the disadvantage of this approach is that more seeds per read are required to one that matches the stripe (mapping structure that builds a matrix of current read to each base and performs a read base to any base score) used in the index, e.g. only one in 20 seeds in the read can match the seeds used in the index. While the computational cost of additional hash table lookups seems like it ought to be small, it is in fact substantial because each lookup will result in a CPU cache miss due to the index being much larger than the processor's L3 cache [111]. A read from main memory, without going through the processor cache, takes hundreds of cycles to map in modern processors. SNAP avoids this costs by indexing overlapping s-mers (i.e, a sliding window of length s one base at a time over the database). This requires a larger index but one that is still well within the means of modern servers. For example, SNAP requires 39 GB for an index of the human genome [111].

This memory burden can be alleviated to some extent inside a database system by indexing each chromosome separately. While this does mean that queries need to be run against each chromosome separately, this native split of the query can be leveraged for scalability purposes as it gives a clear delineation for process separation.

2.7.1.4 Reverse Word Index

A reverse word index is a sorted location table that maps the locations of every subsequence of length k (called the *wordlength*) within a sequence or set of sequences. Each row is in form of a tuple *(sequence, linked list of locations)* as seen in Table 2.8. Given that this index is stored in a table, the cost of an index lookup is the same as that of a row lookup. This is similarly for insert and delete operations. This means that if we build an index on the word column of the table, our costs are the same as that of a B-Tree.

This data structure was used in the pilot program dbBLAST to index the reference genome.

Word	Location(s)
AGC	0, 5
GCT	1, 6
CTT	2
TTA	3
TAG	4
TAA	8

Table 2.8: Example of a reverse word index of wordlength 3 for the sequence AGCTTAGCTAA

Metric	Average	Worst case
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Table 2.9: Compute complexity of a Reverse Word Index

2.7.1.5 Suffix Tree and Suffix Array Index

A suffix tree is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. Suffix trees allow particularly fast implementations of many important string operations.

A suffix array is a space efficient way of representing a suffix tree. It can be constructed thus:
Let $S = S[1]S[2]...S[n]$
be a string; and
let $S[i, j]$ denote the substring of S ranging from i to j.
The suffix array SA of S is now defined to be an array of integers providing the starting positions of suffixes of S in lexicographical order. This means, an entry $A[i]$ contains the starting position of the i-th smallest suffix in S and thus for all $1 < i \leq n$:$S[SA[i-1], n] < S[SA[i], n]$.

Consider the text $S = CAGAGA\$$ to be indexed in Table 2.10:

i	1	2	3	4	5	6	7
$S[i]$	C	A	G	A	G	A	$

Table 2.10: Sequence CAGAGA character positions

The sequence ends with the special sentinel letter $ that is unique and lexicographically smaller than any other character. Table 2.11 shows the next stage of suffixes, while Table 2.12 shows the suffixes sorted in ascending order:

Suffix	i
CAGAGA$	1
AGAGA$	2
GAGA$	3
AGA$	4
GA$	5
A$	6
$	7

Table 2.11: Sequence CAGAGA suffixes defined

Suffix	i
$	7
A$	6
AGA$	4
AGAGA$	2
CAGAGA$	1
GA$	5
GAGA$	3

Table 2.12: Sequence CAGAGA suffixes sorted in ascending order

Table 2.13 shows the starting positions of the sorted suffixes from array SA:

i	1	2	3	4	5	6	7
$SA[i]$	7	6	4	2	1	5	3

Table 2.13: Sequence CAGAGA suffixes order

Finally, for more clarity the suffix array with suffixes written out vertically:

i	1	2	3	4	5	6	7
SA[i]	7	6	4	2	1	5	3
1	$	A	A	A	C	G	G
2		$	G	G	A	A	A
3			A	A	G	$	G
4			$	G	A		A
5				A	G		$
6				$	A		
7					$		

Table 2.14: Sequence CAGAGA suffixes sorted in ascending order

So for example, Table 2.14 shows that $SA[3]$ contains the value 4, and therefore refers to the suffix starting at position 4 within S, which in turn is the suffix AGA$.

The suffix tree that is constructed from the string CAGAGA$ can be seen in Figure 2.16. Each substring is terminated with special character $. The six paths from the root to the leaves (shown as boxes) correspond to the six suffixes A$, GA$, AGA$, GAGA$, AGAGA$ and CAGAGA$. The numbers in the leaves give the start position of the corresponding suffix. Suffix links, drawn dashed, are used during construction.

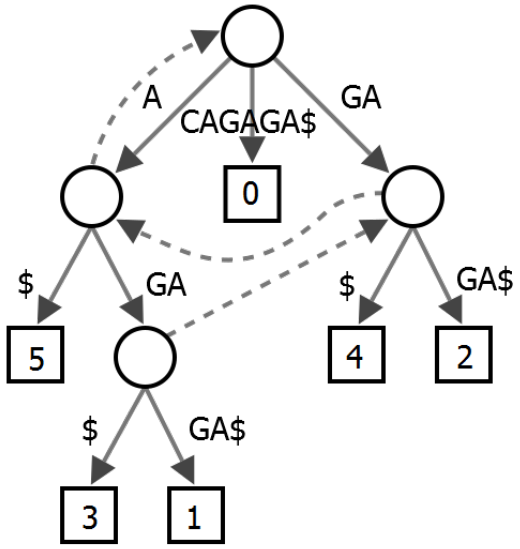

Figure 2.16: Suffix tree for the sequence CAGAGA.

Metric	Average	Worst case
Space	O(n)	O(n)
Construction	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Table 2.15: Compute complexity of a Suffix Tree

2.7.1.6 BWT Explanation

The Burrows-Wheeler transform (BWT) rearranges a character string into runs of similar characters. This is useful for compression, since it tends to be easy to compress a string that has runs of repeated characters by techniques such as move-to-front transform and run-length encoding [53]. More importantly, the transformation is reversible, without needing to store any additional data. The BWT is thus a computationally inexpensive method of improving the efficiency of text compression algorithms. The transform is done by sorting all rotations of the string into lexicographic order, by which we mean that the 8 rotations appear in the second column in a different order, in that the 8 rows have been sorted into lexicographical order. We then take as output the last column and the number k = 7 of the row that the non rotated row ends up in. For example, the sequence "ˆCAGAGA$" in Table 2.16 is transformed into "CGGˆAA$A" in Table 2.17 through these steps.

row	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	ˆ	C	A	G	A	G	A	$
1	C	A	G	A	G	A	$	ˆ
2	A	G	A	G	A	$	ˆ	C
3	G	A	G	A	$	ˆ	C	A
4	A	G	A	$	ˆ	C	A	G
5	G	A	$	ˆ	C	A	G	A
6	A	$	ˆ	C	A	G	A	G
7	$	ˆ	C	A	G	A	G	A

Table 2.16: BWT cycle of all possible rotations of the sequence ˆCAGAGA$

row	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
2	A	G	A	G	A	$	ˆ	C
4	A	G	A	$	ˆ	C	A	G
6	A	$	ˆ	C	A	G	A	G
1	C	A	G	A	G	A	$	ˆ
3	G	A	G	A	$	ˆ	C	A
5	G	A	$	ˆ	C	A	G	A
0	ˆ	C	A	G	A	G	A	$
7	$	ˆ	C	A	G	A	G	A

Table 2.17: BWT sorted cycle of all possible rotations of the sequence ˆCAGAGA$

However, the a slight modification of the process, whereby we instead remember the starting location ^ with a pointer instead of a character, treat the termination character $ as lexicographically less than any of our other characters, and we have a result show in Table 2.18 where the order of the sorted rows of the rotation cycle is equivalent to our suffix array order $SA = [7,6,4,2,1,5,3]$ for a BWT of AGGC$AA.

row	**1**	**2**	**3**	**4**	**5**	**6**	**7**
7	$	C	A	G	A	G	A
6	A	$	C	A	G	A	G
4	A	G	A	$	C	A	G
2	A	G	A	G	A	$	C
1	C	A	G	A	G	A	$
5	G	A	$	C	A	G	A
3	G	A	G	A	$	C	A

Table 2.18: BWT sorted cycle of all possible rotations of the sequence CAGAGA$

2.7.1.7 FM Index

An FM-index is a compressed full-text substring index based on the Burrows-Wheeler transform, with some similarities to the suffix array. It was created by Paolo Ferragina and Giovanni Manzini [30], who describe it as an opportunistic data structure as it allows compression of the input text while still permitting fast substring queries. The name stands for Full-text index in Minute space.

It can be used to efficiently find the number of occurrences of a pattern within the compressed text, as well as locate the position of each occurrence. Both the query time and storage space requirements are sub-linear with respect to the size of the input data.

An FM-Index consists of three structures. First a BWT string of the text, then a count C[c] of all characters lexigraphically smaller than the character c, then a function $Occ(c,k)$ which calculates the occurrences of character c in the prefix of the BWT string $L[1..k]$. These two additional structures on top of the BWT string allow for the construction of the mapping function $LF(i)$ such that $F[i] = L[j]$.

First we start with $C[c]$ of the BWT string as seen in Table 2.19. Then we compute $Occ(c,k)$ which can be done in $O(n)$ time and is as shown in Table 2.20.

c	**$**	**A**	**C**	**G**
c	0	1	4	6

Table 2.19: Count of each character in the string CAGAGA$

c	**A**	**G**	**G**	**C**	**$**	**A**	**A**
i	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$	0	0	0	0	1	1	1
A	1	1	1	1	1	2	3
C	0	0	0	1	1	1	1
G	0	1	2	2	2	2	2

Table 2.20: Occ(c, k) of string CAGAGA$

Once we have both $C[c]$ and $Occ(c,k)$ we can create the last-to-first mapping as such $LF(i) = C[L[i]] + Occ(L[i],i)$.

The primary operations that are carried out on an FM Index are the Count and Locate operations:

- The Count operation returns a count of all the instances of a pattern $P[1..p]$ within the original text.
- The Locate operation returns the index of a given pattern $P[1..p]$ within the original text.

Ferragina and Manzini observe that the Burrows-Wheeler Transform and the LF Mapping can also be used to perform exact matching of a query string P within the text T [30]. With a combination of Count and Locate, any substring of T can be found by using an FM-Index. With a modification of Count to allow for backtracking, an inexact substring $P[1..p]$ with k mismatches can be found in T. Because the rows of the Burrows-Wheeler Matrix are sorted lexicographically, all rows having P as a prefix must appear consecutively, i.e., within a contiguous range of rows. The EXACTMATCH algorithm iteratively calculates ranges of Burrows-Wheeler rows prefixed by successively longer suffixes of the query. At each step, the length of the suffix under consideration grows by one character and the size of the range either shrinks or remains the same. Like UNPERMUTE (Algorithm 2), EXACTMATCH (Algorithm 4) makes use of a helper function based on the LF mapping, called LFC (Algorithm 3). Unlike LF, LFC takes a second argument c, where c is a character drawn from the text alphabet.

Algorithm 1 Last-to-First mapping function

function LF(i)
  return $C[L[i]] + Occ(L[i], i)$
end function

Algorithm 2 BWT T to original string S function

function UNPERMUTE($BWTstringT$)
  $r \leftarrow 1$
  $S \leftarrow emptystring$
  while $T[r] \neq \$$ **do**
    $S \leftarrow prependT[r]toS$
    $r \leftarrow LF(r)$
  end whilereturn S
end function

Algorithm 3 Last-to-First mapping count function

function LFC(c, r)
  return $C[c] + Occ(c, r) + 1$
end function

Algorithm 4 Exact-match search algorithm

function EXACT MATCH($P[1..p]$)
  $c \leftarrow P[p]$
  $sp \leftarrow C[c] + 1$
  $ep \leftarrow C[c+1] + 1$
  $i \leftarrow p - 1$
  while $sp \leq ep$ & $i \geq 1$ **do**
    $c \leftarrow P[i]$
    $sp \leftarrow LFC(c, sp)$
    $ep \leftarrow LFC(c, ep)$
    $i \leftarrow i - 1$
  end whilereturn sp, ep
end function

Metric	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(n)	O(n)
Delete	O(n)	O(n)

Table 2.21: Compute complexity of a FM Index

The correctness of EXACTMATCH is established in appendix B of the paper by Ferragina and Manzini [30]. This particular data structure is used in many tools developed for read alignment including **BWA** and **SOAP2**.

2.8 Hardware Optimisations

As new hardware is released, algorithms are written to leverage this hardware. This is usually facilitated by the hardware manufacturers releasing an API layer that can allow the software designer to interact with the new functions the hardware provides.

2.8.1 CPU

Most CPUs have been augmented with special instruction sets that facilitate common operations with greater speed. These instruction sets are built into the silicon of the CPUs themselves and have special dedicated registers/caches. The particular flavour of instruction sets is determined by the hardware manufacturer - primarily Intel and AMD for x86_64 ISA chipsets. The manufacturer then exposes these instruction sets to developers through an API. A new flavour generally coincides with an increase in the number of instructions or an increase in register size.

2.8.1.1 SIMD

In order to facilitate parallelism for common functions, CPU vendors began to offer certain instructions that allowed a single operation to be carried out on a prepared set of multiple data items. These instruction sets were bundled as

Name	Vendor	Registers
SSE	Intel	64
SSE2	Intel	64
SSE3	Intel	64
SSE4	Intel	64
SSE4.2	Intel	128
AVX	Intel	128
AVX2	Intel	256
AVX512	Intel	512
FMA	AMD	256

Table 2.22: Special CPU instruction sets

extensions and tagged under the common headline – **Single Instruction Multiple Data** or **SIMD**. The most common extension sets in existing CPUs is SSE (Streaming SIMD Extensions) v4 with modern CPUs that were released after 2014, supporting a new set of extensions under the moniker AVX (Advanced Vector eXtensions). The primary difference (apart from certain supported instructions) between these two extension sets is the size of the register upon which these extensions are executed. In SSE v4, the register size is 128-bits, whereas with AVX, the register size is 256-bits. The register size determines how many data points can be operated on at once. For example, if our data is based on 32-bit integers, then a 128-bit register would allow us to carry out operations on four 32-bit integers, while a 256-bit register would allow us to carry out operations on eight 32-bit integers. Figure 2.17 shows a typical SIMD 128-bit and 256-bit register data partition. Figure 2.18 shows an example of a vectorised addition operation carried out on an SIMD register with a vector depth of 8.

From 2014, .NET 4.5 began testing a System.Numerics package that allowed for SIMD accelerated operations. In 2016 with the release of the new compiler, this package was officially supported as a part of .NET 4.6 [17]. As the package deals with managed objects as opposed to direct register operations, some operations such as shifting, shuffling, masks and horizontal max operations are not supported, but a sufficient portion of SSE4 and AVX operations are supported to allow for the development of a SIMD accelerated dynamic programming sequence alignment algorithm.

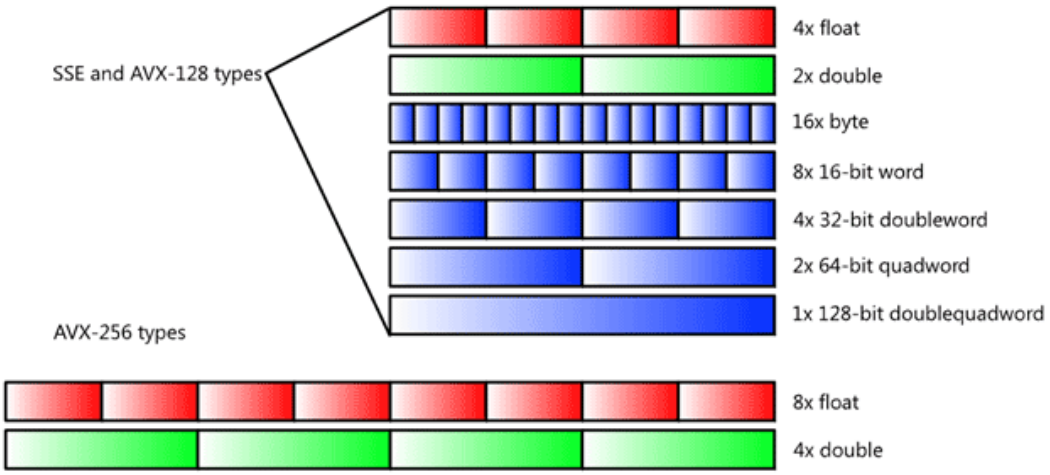

Figure 2.17: Data partitioning in a typical SIMD 128-bit and 256-bit register [60]

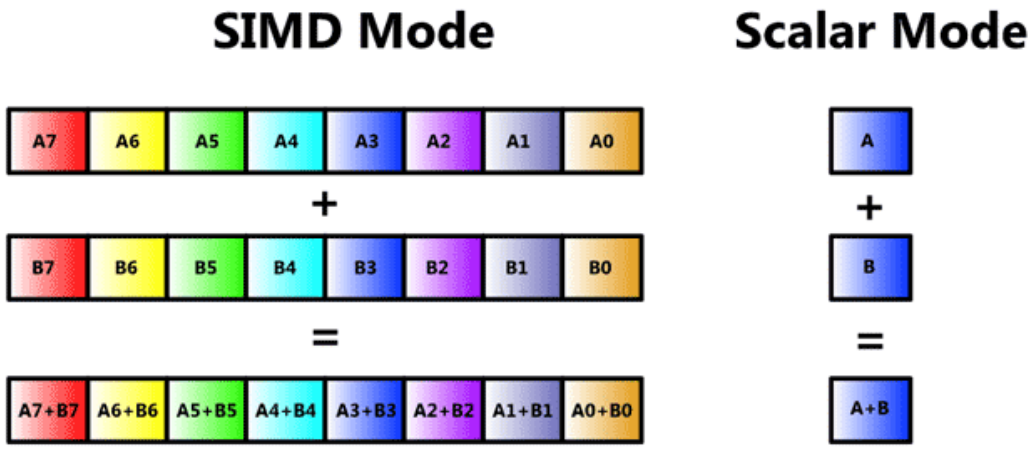

Figure 2.18: Example of a vectorised addition operation carried out on a SIMD register [60]

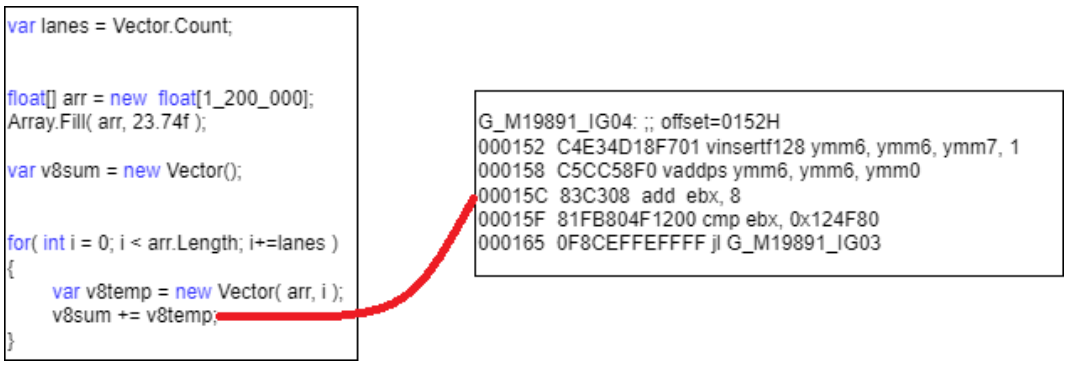

Figure 2.19: Example of C# code to MSIL vectorized code

In Figure 2.19 we can see how C# code from the System.Numerics package is transformed by the just-in-time (JIT) compiler into Microsoft Intermediate Language (MSIL) code. This shows that a high level language levering the benefits of low-level hardware extensions can have a great impact on developer time vs performance gains (as the low level coding would have typically take a lot longer and be a great many more lines).

2.8.1.2 Farrar Method

The Farrar method [29] for carrying out a SIMD accelerated striped Smith-Waterman algorithm is the most efficient for *intrasequence* sequence alignment (some better methods exist for *intersequence* sequence alignment [90]) and it is integrated into some common aligners such as BWA-SW [53], bowtie2 [49], and novoalign [7]. The key innovation of Farrar's approach over previous methods such as Wozniak [110] and Rognes [91] was the use of a striped query profile. Figure 4.13 from [90] outlines the various different query profiles that were considered for SIMD-accelerated SW. Farrar's approach is shown in example C of Figure 4.13 from [90]. By compiling SIMD vectors from every t-th character in the query, Farrar was able to uncouple the dependencies within the two nested loops for iterating across the score matrix. A query profile is a set of pre-computed alignment scores of the query against the alphabet of the target sequence.

2.9 Our Deployment Platforms and Data Sets

This section looks at the various platforms and datasets that will be used to develop and test the methods used in this thesis.

2.9.1 Deployment Platforms

Since our primary development and deployment will occur on Microsoft SQL Server series, we will be using Microsoft Windows series of operating systems. Two primary systems will be used. One will be a personal system, the other will be the testing server. At times we have had to test Linux command line tools, we used the Windows Subsystem for Linux version 2 (WSL2) running Ubuntu 20.04.4. Some items such as NVidia NSight profiler did not run on WSL, so we used the deployment machine running Ubuntu 20.04.4.

2.9.1.1 Windows Deployment Machine

The specifications for this system are shown in Table 2.23:

Component	Description
OS	Microsoft Windows 10 Professional
CPU	Intel Core i7 11700k
RAM	128GB DDR4 3200 Mhz CL16
GPU1	8GB Nvidia RTX 3070
GPU2	8GB Nvidia RTX 3070
HDD	5TB External USB Drive
SSD	120Gb SATA OS Drive
NVME1	PCIe 3.0 480Gb Drive
NVME2	PCIe 4.0 960Gb Drive
RDBMS	Microsoft SQL Server 2018 Developer Edition (v15.0.2000.5)
.NET	.NET 6.0 and .NET Framework 4.8

Table 2.23: Primary testing server specifications

2.9.1.2 Development Machine

The specifications for this system are shown in Table 2.24:

Component	Description
OS	Microsoft Windows 11 Professional
CPU	Intel Core i7 9750H
RAM	16GB
GPU1	6GB Nvidia RTX 1660
SSD1	480GB SATA3 OS Drive
SSD2	PCIe 3.0 2TB Drive
RDBMS	Microsoft SQL Server 2018 Developer Edition (v15.0.2000.5)
.NET	.NET 6.0 and .NET Framework 4.8

Table 2.24: personal machine specifications

2.9.2 IDE

Development was carried out on three primary platforms.

2.9.2.1 MS Visual Studio

The development of command line test modules and integrated stored procedure DLLs were done on Microsoft Visual Studio 2022 Community Edition.

2.9.2.2 MS Visual Studio Code

The development of pre-compiled code for PostgreSQL and unix based pipelines was carried out in Microsoft Visual Studio Code (version 1.72).

2.9.2.3 SQL Server Management Studio

The development of SQL queries and test modules and SQL Server investigations were carried out on Microsoft SQL Server Management Studio (version 18.9.2).

2.9.3 Data Sets

Data sets were sourced from four primary locations. Reference sequences from NCBI, Short Reads from the SRF and Sanger Institute and nCov19 full genomes from Johns Hopkins University.

2.9.3.1 Human Genome Reference Sequence 37.1

The reference genome used was human genome reference sequence version 37.1 sourced from the NCBI **Genomic Reference Consortium** or **GRC** which can be found at https://www.ncbi.nlm.nig.gov/grc/human. While newer versions were available, this version was chosen to allow consistency between comparisons against other programs. This version is also called **hg19**.

2.9.3.2 SRF from Sanger Institute, Cambridge

The initial set of SRF data files were kindly provided by the Sanger Institute for testing purposes (received from archived data Jan 2019 - based on experiments run from 2008-2009). This was a good example of a real world data set. This was used as a query set for the purposes of sequence alignment of short reads from an NGS pipeline.

2.9.3.3 SRF from SRA

A second set of SRF data files were sourced from the NCBI SRA which can be found at https://www.ncbi.nlm.nig.gov/sra for the purposes of testing against the Sanger data set (samples SRX000644 and SRX000711) [51].

2.9.3.4 RepeatMasker files

The files used to determine repeats were from the RepeatMasker website [96] at http://www.repeatmasker.org/species/hg.html. The version used was the latest for hg19 - Repeat Library 20140131.

2.9.3.5 nCov19 files

These files were taken from publicly accessible data gathered by Johns Hopkins University [104] (accession number NC_045512, $MN938384 - MN98390$, $MT642210 - MT642220$).

2.10 User-defined Functions

User-defined functions (UDFs) are critical tools for extending the functionality of a DBMS - particularly the analytics capabilities. Most DBMS vendors offer the ability to write functions and stored procedures in a procedural extension of SQL that has some programmability built in, such as PL/pgSQL for PostgreSQL and TSQL for Microsoft SQL Server. However, these are known to not be very performant [35].

2.10.1 Extensible Language Frameworks

In addition to their own flavour of SQL, vendors will also provide the ability to write UDFs using a more common programming language such as C/C++, Java, Python. Microsoft SQL Server goes further and contains a .NET language run-time environment that can execute code in any .NET language. This allows for a great deal of extensibility in the code for analytics purposes.

2.10.2 UDF Execution Environments

There have been a few approaches to executing stored procedures. Traditionally, in systems such as PostgreSQL, they would be executed as a separate thread call to system operations from inside the DBMS. In this instance, the DBMS would not have control over the execution environment and delays in performance can occur due to resource reallocation requests. In Microsoft SQL Server, a managed environment exists alongside the database engine that is used to execute any .NET UDFs. As such, the memory allocations and data access can be managed via the DBMS engine. An emerging paradigm is containerised execution of UDFs in machine learning-centric data platforms such as Snowflake and Azure Machine Learning instances [41; 94]. These approaches focus on the portability of UDFs between systems and rely on a UDF container management system in addition to provide a client-facing interface on each system.

Chapter 3

Designing a Genomics Platform using Relational Database Systems

Relational database management systems are built with business data processing in mind, such as keeping track of orders, financial transactions, supplier and customer data. This means they assume relatively small, structured records of standard data types (such as short strings, integers, floating point numbers, date and times) which get updated relatively frequently.

In order to develop database systems into a scalable platform for genomics data processing, several aspects have to be adapted. This starts with the storage layer, where we will need to efficiently support the storage of small to very large genome sequences, as well as the efficient indexing of those sequences or corresponding n-grams covering those sequences. In order to support the genomics operations directly in the database system, close to the data, we further will look into the usage of stored procedures for genomics data processing, ideally with hardware acceleration to support the vector- and matrix-based operations common in sequence alignment.

In this chapter we will look at the specific design choices that should be considered when adapting a database engine into a genomics platform.

3.1 Problem Definition

Our primary considerations are the seamless (as much as possible) provision of the functionality associated with a genomics platform while having the advantages of a traditional relational DBMS.

One of the approaches to this is to develop a system from the ground up, such as in the GMQL system by Olha Horlova et al [39]. This requires a great deal of development and training to utilize a new product. Instead, our approach looks to provide a framework for adapting an existing relational engine (such as Microsoft SQL Server, PostgreSQL, Oracle etc.). Specifically, it will require:

- Integrated storage
- Analyses done close to data
- Query acceleration using indices or stored procedure acceleration using user-defined structures (such as user-defined types or user-defined indices)

Now that we have narrowed down our approach to providing a framework for adaptation, we need to consider what types of genomic applications we are targeting, otherwise the scope would again be too large. As such, we choose to focus on the Next Generation Sequencing pipeline.

3.2 Problem Analysis

We can now begin looking at the particular problem set we wish to solve - namely that of providing a NGS pipeline inside a RDBMS.

3.2.1 Pipeline

In order to establish what our platform needs, we first look at existing NGS pipelines to determine our the functions that would be required. We start with the industry standard pipelines discussed in Section 2.4.2 and update them to our requirements.

The reference pipeline can be represented as the following steps:

1. Import raw unmapped reads

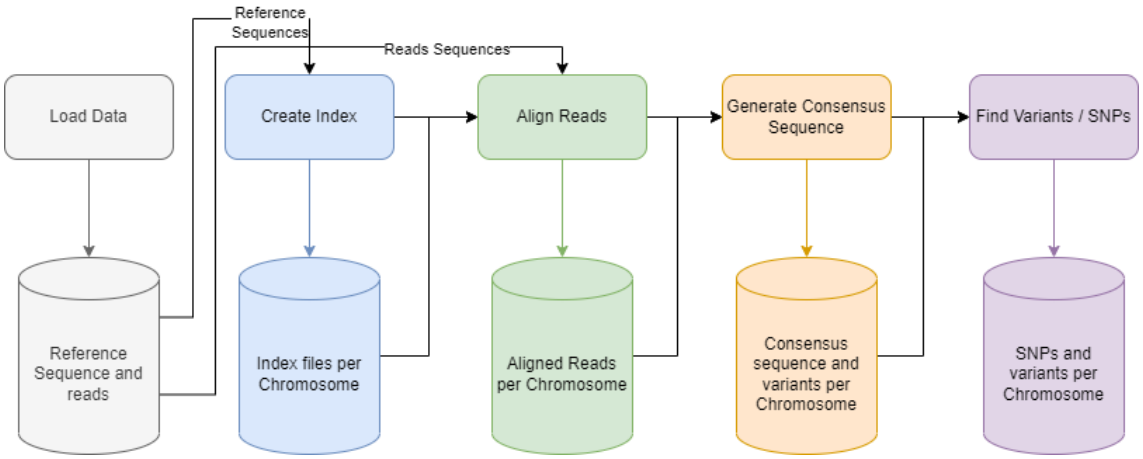

Figure 3.1: Main operations of a sequencing pipeline

2. Map (align) reads to reference (and also needs to create an index for the reference if one does not exist yet)

3. Mark duplicate and poor quality regions in the mapped regions

4. Calibrate the model for quality

5. Call variants (Indels and SNPs)

We decompose data import (step 1 as above) and read mapping (step 2 as above) to a four step process, then we simplify our variant calling to just SNPs (step 5 as above). Our duplicate mapping is carried out during the index creation process and we ignore the base re-calibration step. Finally we end up with with our pipeline which also takes five steps (also shown in Figure 3.1):

1. Data Import

2. Index Creation

3. Sequence Alignment

4. Consensus Sequence Generation

5. Single Nucleotide Polymorphism (SNP) calling

Finally, we can determine what data access is needed by these operations (cf. Table 3.1 on next page).

Step	Data Requirements	Data Sizes	External Data Access?
Data Import	Read existing sequence file formats	Determined by sequence length	Yes
Index Creation	Use an existing file or row as input	Determined by algorithm and sequence length	Yes
Sequence Alignment	UDT inside Tables, Read existing sequence read file formats	Determined by sequence length and read length	Yes
Consensus Sequence Generation	UDT inside Tables, Read existing sequence file formats	Determined by reference sequence length and genome coverage count	Yes
SNP Calling	Table	Determined by reference sequence length and genome coverage count	No

Table 3.1: Data requirements for NGS pipeline steps

3.2.2 Requirements

Based on our expected operations, we can lay out some more requirements for our vision of what a database platform needs in order to be converted to a genomics platform.

3.2.2.1 Biodata Type

There are several different file formats associated with each stage of the sequencing pipeline. A common feature to all of these formats is that they all revolve around the concept of a nucleotide sequence of some length. It could be a small representation like an oligonucleotide or short read of 25 or 35 base pairs (genomic sequence characters), or it could be of larger length like a genomic chromosome of 250 million base pairs.

This means we need a biodata type with at least the ability to represent a sequence. As such any platform we choose to adapt should have the ability for development and integration of robust user-defined types (UDT).

Most often we will be required to process existing files that reside on the underlying file system of the OS upon which the database engine is run. This means that our database engine needs to have the functionality of an external data wrapper that is discussed in Section 2.1.4.

3.2.2.2 Indexing

Two of the operations (create index and align reads) shown in the pipeline Figure 3.1 are associated with user-defined indexes (UDI). This is not indexing that is common to the typical operations of a RDBMS, but rather a specific index file that is created for each reference sequence depending on the error model provided. An error model is typically defined as an 'edit distance' from the exact match to the reference sequence. For example, an error model of edit distance 5 means that each sequence (usually a read) has at most 5 single character changes from the reference sequence to which it is being matched. This UDI can be stored as a UDT within the database, or it can be accessed via an external data wrapper. In addition to the UDIs, the consensus sequence generation and SNP calling stages can be improved by standard database indexes as well. This is because the results of the read alignment stage can be pushed into a table (depending on the design decision), which can benefit from a standard database index. These UDIs are typically accessed via user-defined functions (UDFs) or stored procedures (SP).

3.2.2.3 Analysis Functionality

As we are developing a genomics platform, we need the ability to carry out analyses. This means some sort of programmable function layer must exist in our RDBMS. Typically this would be performed by stored procedures or user-defined functions (UDF). While it is possible to write some functions in platform specific SQL-extensions, this tends not to be performant and is also limited in scope, so we cannot accept SQL-extensions as having our minimum required functionality for analysis.

3.2.2.4 Meta-data Integration

One of the key benefits of a DBMS is that it is designed for the storage and access of structured data. This lines up well with the requirements of a genomics platform to have the facility to manage the storage of metadata. While there exist many file formats that store the outcome of one stage of our sequencing pipeline, there are not any fixed formats for the storage or transfer of metadata

within a pipeline.

By using a database as the cornerstone for our genomics platform, we can provide metadata tables that keep track of all processes and analyses that are involved in a sequencing experiment. This can be set up without user intervention and can be accessed using standard SQL queries.

3.2.2.5 Expected Minimal Support Framework

As we have gone over requirements, we can establish a set of features that we would like to have in our DBMS before it can be adapted into a genomics platform:

- Capability to define and use UDTs
- Ability to provide in-DBMS data analysis via stored procedures or UDFs
- Mechanism to interact with data external to database in a managed fashion via an external data wrapper
- Standard relational model for data interaction between metadata integration and representing relationships between inputs and generated data
- SQL calling mechanism for interacting with the relational engine

Other items that would be nice to have for development or performance purposes, but aren't strictly necessary would be:

- Capability to define UDTs, stored procedures and UDIs using a high level programmable language to minimize development time
- Ability to access existing libraries
- Ability to access hardware acceleration libraries to provide some performance improvement paths

3.2.2.6 Common RDBMS vs. Our Requirements

Looking at the most common RDBMS implementations, we can see how they stack up against our requirements in Table 3.2:

RDBMS	SQL	UDT	External Data Wrapper	Stored Procedures	High level language logic	Execution Environment
MS SQL Server	✓	✓	✓	✓	✓	Managed
PostgreSQL	✓	✓	✓	✓	✓	Precompiled
Oracle	✓	✓	✓	✓	✓	Precompiled
MySQL	✓	✓	✓	✓	✓	Precompiled
SQLite	Limited	✓	X	X	X	X

Table 3.2: Features we need vs. The RDBMS platforms that have them

3.3 Biodata Type

Traditionally, the storage layer of database systems is a so-called *row store* that is optimised for small relational tuples, or records, over atomic data types, such as strings, numbers, or timestamps [95]. Notably, those records are assumed to be small, so that multiple fit onto a single disk page of the storage layer whose typical size as of writing is 4 kB to 8 kB. For example, PostgreSQL 14 usually uses a page size of 8 kB [87], while Oracle traditionally used a page size of 4 kB. Consequently the size of a data record is limited by this page size, and for example Oracle for this reason has a hard limit of 4000 bytes for a VARCHAR attribute [81]. For larger data values such as longer texts or binary large objects (BLOBs), an out-of-line storage has to be deployed, such as for example PostgreSQL's TOAST tables, which require an indirection with every data access [88].

This is a problem for storing gene sequence data which can be small for so-called short reads (a few hundred base pairs), but multiple gigabytes for full reference sequences.

Modern database systems support however the definition of *user-defined types* (UDTs) which should allow us to define a compact representation of gene sequences. We will look into this in Section 4.1.1 in more detail, before we will also look at an alternative approach by keeping the data external to the database storage layer and then access it via specialised data wrappers.

3.3.1 Design Space

When adapting a database for a genomic platform, it is critical to begin with an underlying baseline for how your data will be represented. This baseline determines how your UDFs interact and the amount of I/O necessary for common operations.

As an example, we can look at .NET Bio [89] which uses a single base as the baseline data type for its operations. While this provides a great deal of versatility at the base level (like storing base quality, basecall info etc.), it is simply too much of a datasize and I/O explosion at the sequence level.

Alternatively, another approach is the Genomic Computing (GeCo) approach [39] which uses a sequence range as its baseline. This provides a great versatility in operations such as alignment and windowing operations. However, for this approach to work in terms of I/O, it requires a specific columnar database engine built from the ground up for this purpose. This goes against the concept of adaptation (and also means a much greater developer burden).

Finally, the mainstream approach is using a sequence as the basic building block as it has the easiest representation (that of a character sequence with a pre-defined alphabet).

3.3.2 Design Choices

We choose to use a genomic sequence as the baseline unit. Finer grained objects such as bases and oligonucleotide strings can be derived from this unit - especially using string optimisation functions (such as Span<T> calls in .NET [68]) found in the modern programming languages in which we write our UDTs. We can also represent alignments and quality information as extensions of this base sequence type with different encodings.

The basic structure for our sequence user-defined type (UDT) contains the sequence itself and an identifier. This identifier may be replicated at a table level, but we store it in the UDT for ease of use. The sequence genomic data itself might be in multiple encoding formats or compaction levels, so we store it as a byte array. We call this UDT a **BaseSequence**.

This BaseSequence type is intended for storing a basic genomic sequence

(similar to the FASTA format described in Section 2.5.1.1). So we can collate or extend the sequence type for each of the features that we required. The encoding and storage represent for this UDT and its extensions are described further in Section 4.1.2, for now we will provide an overview of the other common sequence representations expected in our pipeline such as other concepts we need to represent as types such as reads, reference sequences and alignments.

Reads are extensions of sequence types with metadata and quality information. Quality information itself is usually represented as a sequence mapped directly to a sequence, so we stick to the same format.

A reference sequence for an organism is usually a collation of many sequences that have a logical biological split such as chromosomes, mitochondrial DNA, other cellular DNA or RNA etc. We can store a reference sequence as multiple chromosome sequences in a table or as a single user-defined type (UDT) that contains multiple BaseSequence UDTs. A chromosome itself can be directly implemented as a BaseSequence. If used in a table format, each chromosome can contain a foreign key id that points back towards the reference sequence (in this instance a collection of BaseSequences) it belongs to.

3.3.2.1 Biodata Access

In our platform, we would manage biodata access via two approaches:

1. Load sequence data directly into the database and store sequences using the database's storage engine in tables.

2. Use an external data wrapper mechanism such as SQL Server's **FileStream** to manage access to external data but don't increase storage overhead by storing a second copy inside the database.

The first approach is the typical approach for databases. Users load the data, then carry out operations. This is suitable for some of our operations (such as consensus sequence generation and SNP calling) but not for those operations that handle large immutable datasets such as reference sequences and reads. The second approach is more suitable for handling and interacting with such data. In order to facilitate such handling, wrappers need to be designed to

interface with each particular file format and allow communication between each file format and our internal biodata representation.

3.4 Analysis Functionality

Given the pipeline, we have a few requirements from our analysis functionality. We have to accommodate for reading / parsing files into UDTs, creating a user-defined index (UDI), read alignment, consensus sequence generation and SNP calling. All these functions needs to be programmatically encoded into our platform.

3.4.1 Design Space

There are a few methods to provide programming or logic to a database engine:

- stored procedures or user-defined functions (UDF) in the database engines SQL extension;
- stored procedures or UDFs in a high level language as supported by the database engine;
- Extensions of the database engine to integrate new features;
- Calling existing tools in the underlying host system and then importing the results into the database engine

In addition, we may want to look into ways that these analysis functionalities can be improved or accelerated by specific hardware accelerators, such as CPU, GPU or FPGA accelerators.

3.4.2 Design Choices

As we established in Section 3.2.2, we need to interact both with UDTs and UDIs. The best option then would be to design all bespoke solutions in a high level language that is supported by the database engine. This can be as a managed code environment - such as the SQL Server CLR environment; or as a set of pre-compiled binary invocations - such as in PostgreSQL extensions.

3.5 Indexing

As part of our requirements for programmable functionality, we need to be able to create an index for assisting in efficiently aligning reads to the reference sequence.

The index we define here is different to the typical database index. We are indexing a file or a UDT, rather than a typical RDBMS structure such as a table. It has more in common with the full text indexing that is provided by traditional RDBMSs. We refer to it as a user-defined index (UDI).

3.5.1 Design Space

The available options are a flat file index stored outside the database, a UDT stored inside a table or an internal b-tree index that we typically find in a RDBMS.

3.5.1.1 Type of Index

The index types that we can use that are commonly found in sequencing pipelines are:

- Hash Index
- Burrows Wheeler Transform (BWT) based Index
- Reverse Word Index

3.5.1.2 Index Storage

UDIs are generated from reference sequences that are static and very large. As such, it could make sense to store index files as flat files alongside the references in a manner similar to existing command line tools. Access to the UDI file would be managed via an external data wrapper and a parser.

Alternatively, the UDI can be stored inside the database in a table as a UDT.

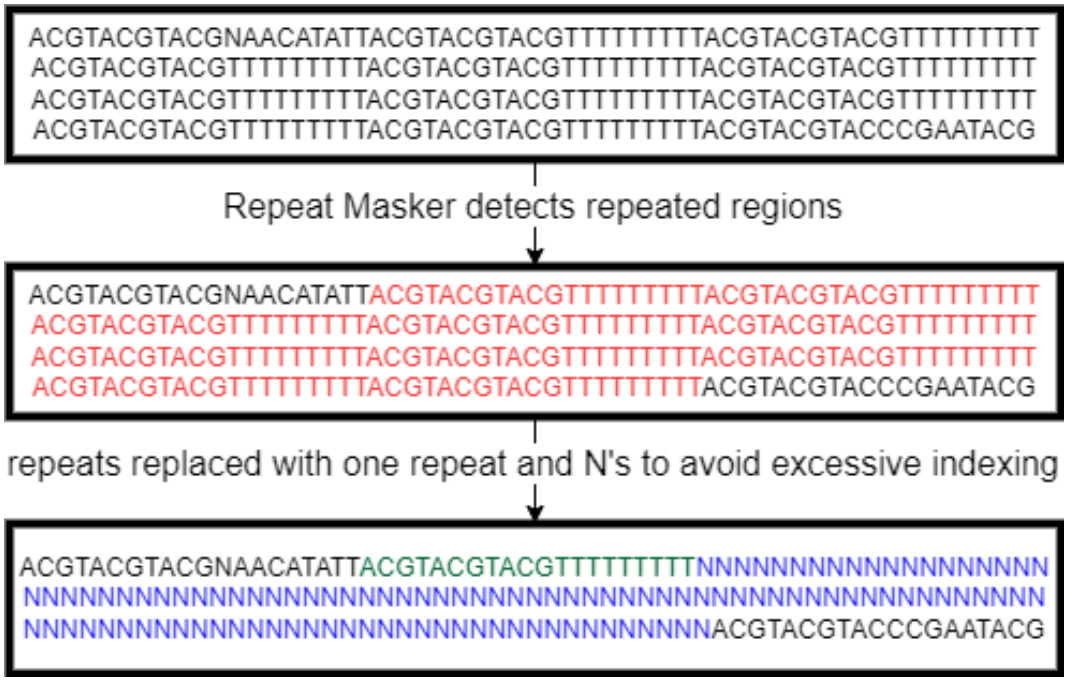

Figure 3.2: How repeat masking works

3.5.1.3 Error Model of the Index

In addition to the type of index, it is possible to have an associated error model. The implementation of how the error model works can be dependent on the index type, algorithm design or both.

3.5.1.4 Reference Sequence Masking

Reference sequences have been identified to have many repeats and low complexity zones. Figure 3.2 shows how the mechanism of repeat masking affects a sequence and Figure 3.3 shows a breakdown of repeats in the reference sequence **hgref37.1**. As can be seen, over 52% of the reference sequence can be masked.

This greatly reduces the areas of the reference sequence that need to be indexed and thus would reduce index sizes. The question is whether to use an existing tool or incorporate this into our pipeline. The existing program takes in the repeat files found at the RepeatMasker website [1] and uses the entries to modify the repeats and low complexity regions into N's.

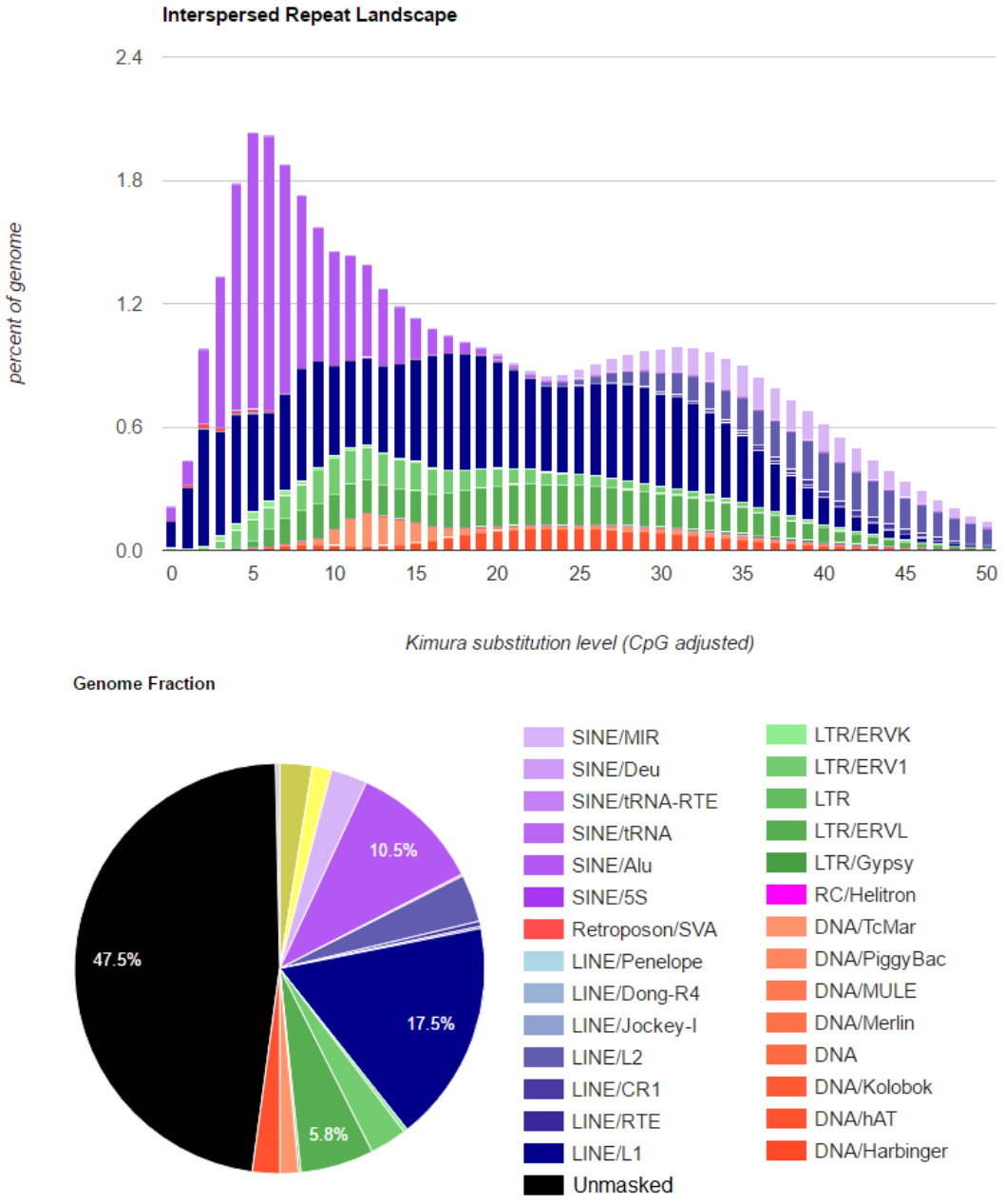

Figure 3.3: Analysis of Human Genome Consortium reference sequence 37.1 showing repeats and low complexity regions [96]. The top histogram shows the percentages of the genome that were substituted at a various Kimura substitution levels by the Repeat Masker program. The bottom pie chart shows the shows the total substitutions by percentage of bases in the genome. The colours represent the various different types of repeats that were detected and masked by the Repeat Masker program

3.5.2 Design Choices

Ultimately, the index choice is dependent on the algorithm we use. During short read alignment, we can look at different alignment algorithms such as BWT-based alignment or hash-based alignment.

A hash index becomes a more effective version of the reverse word index, so we opt not to use a reverse word index.

In terms of index storage, we choose to store the UDI in a table that contains indices as this precludes the overhead of reading and writing an index to disk to store it as a managed file.

In order to simplify the implementation, we recommend incorporating the error model into the index design.

Given that repeat masking is an infrequent operation, it makes sense to execute these once on each reference sequence using existing tools.

3.6 Metadata Integration

One of the advantages of using a database engine is that we can easily facilitate the capture of the processes and metadata involved in any pipeline. If we look to the pipeline in Figure 3.1, we can determine what metadata we need.

The metadata that needs to be captured:

- Metadata about the sequencing experiment
- Metadata on the reference sequence, such as name, source, whether it is has been masked etc.
- Metadata on the set of read sequences used for the experiment.
- Metadata on alignment procedure data such as type of index used, index error model used, which reference sequence the index was based on and information on the read set used for the alignment.

3.6.1 Design Space

In order to provide seamless integration of metadata, we must capture the metadata without having to burden the user with extra queries. There are two ways

to do this:

1. Use triggers on function calls and table modifications or updates; or

2. manage each experiment via a stored procedure. The user just provides the input parameters for the experiment (reference and read files, error model for alignment) and the stored procedure should manage all the metadata acquisition and table inserts.

3.6.2 Design Choices

It is well known that triggers can create a substantial performance overhead. Hence, we choose using a stored procedure to handle the running of an experiment because it gives us both a clear picture for performance analysis and a finer grained control of the metadata allocation process.

3.7 Data Model

Given that the design parameters and requirements have been established, we also need to consider the appropriate choice of a data model for the system. The pilot project discussed in Chapter 2.2 compared various different relational engines to identify their suitability for extension into a genomics platform. However, the question of whether a relational platform itself is the most suitable target for adaptation must be discussed.

The previous design parameters establish the following guidelines:

- The system should take on a data-centric approach with an computation occurring as close to the source / generators of the data as possible;

- The system would benefit from existing architecture like set-based operation to utilise existing algorithmic improvements, minimise development time and minimise user training requirements;

- The system should have some capability for metadata management that is abstracted from user involvement but easily accessible; and

- The system should have a pipelining mechanism that would allow incremental results to be generated while handling an incoming data stream from a data source (such as a sequencing machine).

3.7.1 Design Space

For structured data, the relational data model is most widely used, representing data as structured rows (tuples) following a fixed schema with one or more tables (relations). In fact, most of the existing biodata databases such as GenBank use the relational data model [6]. However, nowadays there exists several other data models for 'Big Data' processing that usually coalesce around three other formats (with most other options being more implementation specific rather than data model dependent):

1. Key-Value store-based systems;
2. Document/Semi-structured based systems; and
3. Graph-based systems;

When comparing these options to a relational data model, some of them do have advantages - particularly in implementation considerations.

3.7.1.1 Key-Value Stores

Key-Value stores (KVs) are very simplistic in their design and offer great benefits in terms of scaling. Access patterns are well defined and the values can store complex data.

However, the typical offerings for these systems are monolithic and do not allow for their operation close to the compute that is required for our data-centric design. The more complex our data types, the less effective a KVs will be due to the required data movements.

Furthermore, the addition of metadata and set operations requires the construction of complex schemas and data management that is very akin to a relational model, so it would make more sense to choose an already existing system that has those features built-in.

Despite the above limitations, KVs do very well in pipelining applications and have very efficient join mechanics for data aggregation. Ultimately, this is insufficient to choose them over a relational model based system.

3.7.1.2 Semi-Structured and Document Stores

A substantial amount of data movement and representation occurs in semi-structured or document data thanks to the prevelance of the need to interact with the architecture of internet. A large portion of data movement is API traffic which occurs in document format [14]. This means that several robust systems have been developed to handle this type of data at scale. Document-type data in particular is especially suited towards metadata capture and associated tagging.

Other semi-structured data sources that would be common the a genomics scenario, such as image data, are less well managed and require substantial bootstrapping to both transfer, manage and analyse. The situation arises that multiple systems working together to handle different types of genomic data would again begin to resemble the core features of a relation model system.

3.7.1.3 Graph-based Systems

Graph-based systems are a more recent focus than the previous two data models. These are now emerging as the premier API expression option as it allows a versatility not found in document stores and a rich expression of connections and relationships very similar to a relational data model. This helps facilitate set-type operations and joins at scale.

These systems tend to be very effective at data slicing and extraction, but would be poorly suited for the complex analytics that are required by a genomics data platform. The benefits would not scale as well as a system designed around a relational data model. Graph databases are still a relative new technology [34]. Neo4J [75], one of the most prominent graph database systems, started a mere 15 years ago and many other systems are even newer than this (such as Memgraph [66]), which means that not all required functionality, most prominently stored procedures or UDTs, are supported yet.

3.7.2 Design Choices and Data Model Comparison

Table 3.3 shows a comparison between the above discussed models and a relational model. The choice to choose a relational model largely relied on the features outlined at the start of this Section 3.7. Based on the requirements, a relational model seems best suited for the task (and required the least amount of development overhead to build a proof-of concept).

Requirements	**Key-Value**	**Relational**	**Document**	**Graph**
Data-centric Compute	No	Yes	Limited	Limited
Set-oriented operations	No	Yes	Yes	Yes
Metadata management	Limited	Yes	Yes	Limited
Pipelining	Yes	Yes	Limited	Limited

Table 3.3: Data Models vs Design Requirements

Chapter 4

Implementation of a Genomics Platform on Microsoft SQL Server

In this chapter we describe a reference implementation of the design principles laid out in Chapter 3 using MS SQL Server.

To re-iterate, the operations required to achieve our goals of re-creating a sequence pipeline inside a database engine are as follows:

- Loading and interacting with bio-data.
- Creating indices on said bio-data (usually a well established reference sequence) to accelerate access and processing for our UDFs.
- Aligning incoming input bio-data (usually in the form or reads) to our reference sequence.
- Collating the aligned input bio-data to create a consensus sequence for the input.
- Running a differential comparison between the consensus sequence and the reference sequence to identify single nucleotide polymorphisms (SNPs).

An illustrative version of the pipeline can be seen in Figure 3.1.

We finish off this chapter with two tables (Table 4.13 and Table 4.14) that list the most important UDTs and stored procedures that facilitate our implementation.

4.1 Storage Layer

We start with a look at the implementation choices for the storage layer, namely how to build out our choice of our fundamental biodata type and data access methods via external data wrappers discussed in Section 3.3. We will look into how to design a sequence-based fundamental biodata type, including the compressed base representation and sequence extensions. Access to this data will be two-fold. Firstly, accessing the data stored on the underlying file system as reads and a reference sequence in industry specific file formats. Secondly, how we imagine storing the data in our database both as a UDT object for programmable access and how that UDT might reside within a table schema.

4.1.1 BioData Type

Previously we have established that a sequence is the baseline representation for our pipeline. Following on from that baseline, we define a sequencing experiment as requiring the following: a reference sequence, an indexing type, an error model and a set of reads. Figure 4.1 outlines what happens when a user initiates a sequencing experiment.

4.1.1.1 Data Access

During the design phase, we made the choice of storing our large immutable sequences in the file system and accessing them via an external data wrapper. For MS SQL Server, we have the option of defining a table with a blob reference with a FILESTREAM identifier. This means the reference file is stored as a managed BLOB object inside the file system. The alternative is to use an external data wrapper to create a parser that can stream a file that sits in the file system. While both are valid options, we implemented the latter.

4.1.2 Sequence Storage Format

Many modern database systems support the definition of *user-defined types* (UDTs) which allow to extend the built-in type system of a database. As we mentioned earlier in our design discussion - we decided to build our pipeline around the

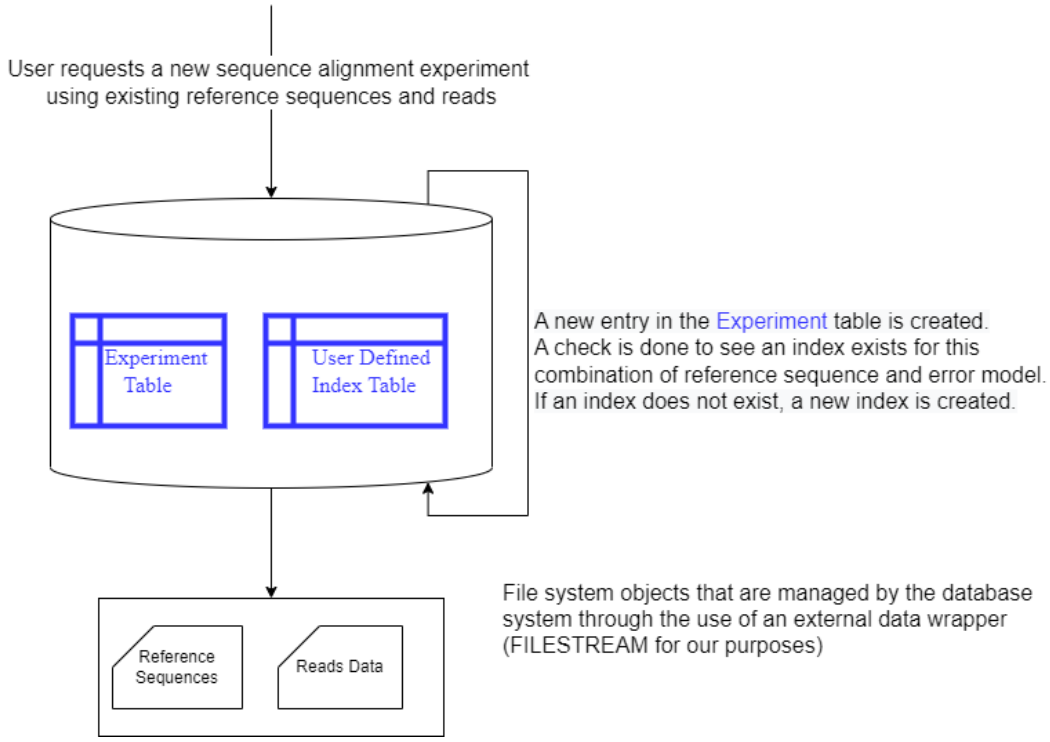

Figure 4.1: Sequencing Experiment Call

concept of a **BaseSequence**. An important aspect of representing this sequence is the different encoding schemes we can use to store the biodata of said sequence. One of the limiting factors of a database is the speed at which it can process data. This has several determining factors such as the size of the page in the storage design, the read/write speed of the underlying hardware upon which the data resides and the speed/interaction of the operation system and database engine with the hardware and each-other. As such, database algorithmic processes are usually discussed with the term Input / Output Operations per second (IOPS). This value is usually associated with a fixed data read rate and page size. Hence, if we can increase the amount of logical data that is processed per page, then we should be able to increase our IOPS. It is based on this premise that we have various compaction and compression schemes for our sequence data.

4.1.2.1 Base Representation

To begin building a biological sequence UDT, we initially start with the basics - base representation. If we restrict our most basic base representation to have a minimal alphabet for DNA of **{A C G T}**, then we can use a minimum of 2 bits to represent each base as shown in Table 4.1. Then we can use two operations on a byte representation to create our 2-bit representation - a bitmask of 0x06, followed by a right bitshift of 1. If we increase the alphabet to include the symbol for an unknown base or an error base **N**, then we need a minimum of 3 bits to represent each base as shown in Table 4.2. This operation also uses two operations - a bitmask of 0x0E followed by a right bitshift of 1. Given that most built-in data operations are byte-based, we need to consider adhering to byte boundaries where possible. While we are using compaction to increase our effective IOPS, we are introducing an overhead by carrying out the compression and decompression process. Three-bit representations can cause extra processing to adhere to the byte boundary as it is not a factor of 8. So we should also consider a 4-bit representation to facilitate less processing as it will only take a single operation to go from byte representation to 4-bit representation - a bitmask of 0x0F as shown in Table 4.3.

Character	ASCII Number	Byte value	2-Bit representation
A	65	0100 0001	00
C	67	0100 0011	01
G	71	0100 0111	11
T	84	0101 0100	10

Table 4.1: Basic 2-bit genomic character representation

Character	ASCII Number	Byte value	3-Bit representation
A	65	0100 0001	000
C	67	0100 0011	001
G	71	0100 0111	011
T	84	0101 0100	010
N	78	0100 1110	111

Table 4.2: Slightly lenient 3-bit genomic character representation

Character	ASCII Number	Byte value	4-Bit representation
A	65	0100 0001	0001
C	67	0100 0011	0011
G	71	0100 0111	0111
T	84	0101 0100	0100
N	78	0100 1110	1110

Table 4.3: Slightly lenient 4-bit genomic character representation

4.1.2.2 Sequence Representation

The most naive approach would be to represent each sequence as an array of bases with a consideration towards database page sizes as displayed with the BinSeqData UDT in the pilot program in Section 2.2.2.1. This is fine for a basic sequence such as a reference sequence but a read would also need to have a field or an array of fields to contain quality scores as well. Once we get to representing alignments, we would also encounter problems with representing gaps. Hence we need extensions of the basic sequence or a different underlying representation of the alphabet in a basic sequence for each scenario.

4.1.2.3 Reference Sequence Representation

A reference sequence can have an alphabet of **A,G,C,T,N** which requires at least 3 bits to represent each base. If we are random accessing this reference sequence, then we should split the sequence into page size chunks to reduce the required number of page reads. This is required for the extend portion of any search algorithm such as those used for dbBLAST or Hash Index search. However, if we are using the complete reference sequence to create our indices, we only need sequential access to the reference sequence. Given that the reference sequence is one of the largest persistent stores, we can manage access via an external data wrapper instead of loading the whole sequence into our database engine. In MS SQL Server, this would be the Filestream wrapper. If we do choose to load a reference sequence into the database engine, then it could be segmented (split into several sub-sequences) due to memory performance considerations. As such, we would by default store reference sequences as a UDT in a SegmentedSequences table in our schema (outlined in Figure 4.16), so our

reference sequence is represented in a two fold manner:

1. A FileStream wrapper on the FASTA flat file that contains the reference sequence.

2. A table of segmented sequences:

    - **REFID** Int64. A foreign key pointing to a unique reference sequence ID which identifies the original reference sequence uniquely.
    - **STARTPOS** Int64. The starting position of this segment of the sequence in the reference sequence.
    - **ENDPOS** Int64. The end position of this segment of the sequence in the reference sequence.
    - **REFSEQSEG** SegmentedSeq. This contains the segmented sequence of the reference sequence represented as a binary block of Page size - 3*8 bytes - overhead bytes worth of bases in 3-bit encoding.
    - The primary key can be generated by using the tuple {REFID, STARTPOS, ENDPOS}, but most accesses will be based on those individual values and not the primary key.

4.1.2.4 Read Representation

Reads require a few fields and have a few restrictions that we can work with:

1. Table 2.3 shows the read sizes we are dealing with: 100 bp to 700 bp for most platforms and 4600 bp to 14000 bp for PacBio platforms, so we are unlikely to require any segmented sequences.

2. Each base in the read sequence has a corresponding quality score which takes up a single byte of information.

3. Reads don't need to have any unknown or **N** bases, just low confidence or low quality base calls, so we can use 2-bit representation for each base.

Based on the above criteria, the ReadSequence type was defined as follows:

- **length** Int32. An integer representing the length of the read in case its not exactly divisible by 4 and the end of a byte needs to be padded to fit the Byte type but does not add extra bases to the actual sequence.

- **reads** Byte[]. A Byte array containing the reads based on a 2-bit representation.

- **quality** Byte[]. A Byte array containing the quality for each base in the read. This is 4 times the size of the reads array as each 2 bits in that array corresponds to 1 byte in this array.

- **metadata** string. Associated metadata such as the base caller, instrumentation etc. When repetitive, this data can be stored at a higher level within a table which references many reads.

4.1.2.5 Alignment Representation

If we wish to extend a sequence representation to represent an alignment, then we need to begin with our reference sequence alphabet and extend it with artefact characters that are introduced during an alignment. The characters that are introduced are twofold:

1. **gap** A gap in the read sequence which indicates an insertion has occurred in the read sequence.

2. **refgap** A gap in the reference sequence which indicates a deletion has occurred the read sequence.

A gap in an alignment is represented by the '-' character. This indicates, for example, to extend the 3-bit representation of a base with a '-' or a gap character, we would add **gap: 101** to the 3-bit representation shown in Table 4.2.

Since we do not want to modify the reference sequence for gaps in the alignment, we also need to assign a second value or a 'reference gap' to indicate a gap in the reference sequence. Similar to with a gap, the 3-bit representation would be **refgap: 110**. The updated 3-bit representation is shown in Table 4.4.

Character	Bit Sequence
A	000
C	001
G	011
T	010
N	111
gap	101
refgap	110

Table 4.4: 3-bit genomic character representation allow for gapped alignments

An example alignment would look like this:

```
readseq: A C - A T A T N A T T C G
refrseq: A C G A T A T N A - T C G
almnseq: A C - A T A T N A refgap T T C G
encoding: 000|001|101|000|010|000|010|111|000|110|010|010|001|011
```

4.1.2.6 Multiple Sequence Alignment Representation

Multiple sequence alignments are represented in the SAM format as described in Section 2.5.1.7. During consensus calling, for ease of use, we use some techniques to make the process more efficient. The above gapped sequence representation along with a start and finish position where it is aligned against the reference sequence is loaded into a table for a pivot command that is called while a sliding window iterates along the reference sequence. As such alignments are reported to a table with the following schema:

- **ExperimentID** Int32. An Identifier to associate the particular sequence alignment and consensus experiment this alignment was a part of.
- **AlignedSequence** GappedSequence. A ReadSequence stripped of quality and metadata, with its sequence parsed into 3 bit, 6 character representation just given an alignment against the reference sequence.
- **Score** Int32. A confidence score for the alignment based on gaps, substitutions and the quality of the read.
- **REFID** Int64. A foreign key pointing to a unique reference sequence ID which identifies the original reference sequence uniquely.
- **STARTPOS** Int64. Start position of this sequence in the reference sequence.
- **ENDPOS** Int64. End position of this sequence in the reference sequence.

4.1.3 External Data Wrappers and Parsers

Many database systems find the need for accessing data files inside the OS file system. This is achieved through the use of external data wrappers. We utilize an implementation of this process called FILESTREAM access to achieve this in Microsoft SQL Server (as described in Section 2.1.4).

4.1.3.1 FileStreams

The massive data volumes of scientific data management require efficient support of binary large objects (BLOBs), an area for which DBMS are known to not necessarily shine [92]. Microsoft SQL Server 2008 introduced a new functionality of storing BLOB data as files in the file system rather than with the normal relational storage engine and future versions have extended this feature. In a SQL Server table, a BLOB can be represented with a varbinary(max) data type. In order to access the FileStream features, that data type is also given the **FILESTREAM** designation. For example:

```
CREATE TABLE Chromosomes (
    id          varchar(max),
    refseqid    varchar(max),
    seqdata     varbinary(max) FILESTREAM
);
```

Those FileStream BLOBs are under full transactional and managerial control of the database system – all shared access is synchronized via the concurrency control of SQL Server and all file data is managed by the DBMS utilities (e.g. backup/restore, database consistency check). All FileStream BLOBs can be accessed directly using standard file system APIs from client programs; this access does not go through the databases buffer pool, but rather supports efficient streaming access of the FileStream content to clients [69]. As the name suggests, this functionality is mainly targeting efficient access to large BLOBs such as images or videos, but can offer an efficient way to access the large amount of data generated from a NGS pipeline without needed to load the data into a table. Another advantage of this approach is that functions can be written as taking input via StreamReader, thus allowing for easier adaption for directly connecting the data management service to the data generator (in this case, an NGS pipeline).

4.1.3.2 Table-valued Functions

Alternatively, we can directly parse a particular file into a table format for the database. Here we can use table-valued functions (TVFs) which provide a mechanism for converting raw data (i.e., in a text file, binary file or an incoming stream via some sort of interface such as a network) into a table. To accomplish this conversion in SQL Server, the Common Language Runtime (CLR as introduced in 2.1.3.1) TVFs use a pull model to stream data as a collection of rows. SQL Server requests (pulls) and processes one row of data from the CLR TVF at a time as opposed to the TVF materializing the entire relation and then the entire relation being processed. This allows the query engine to begin consuming results immediately after the first row is produced instead of having to wait for the entire return table to be populated. To affect this facility in a parser, we implemented the IEnumerator interface for our CLR TVFs [71].

4.1.3.3 Combining FileStream and TVF to create a BLOB Wrapper

Most database engines allow extending the core database functionality with user-defined functions and stored procedures. In SQL Server, one can write user-defined functions in either TSQL, or in any .NET language. User-defined functions can be either scalar or table-valued. The API for providing TVFs follows the standard iterator interface of a relational query engine [92]. This means that programming a table-valued function is much closer to implementing a user-defined relational operator than writing a schema-level UDT method. Figure 4.2 shows a sequence diagram of the interaction between the query processor (QP) and a table-valued file wrapper function (TVF) that parses data stored in a FileStream BLOB. The actual TVF does not do much but rather all data access (and file parsing) logic concentrates in the TVFs result iterator. The query processor uses this iterator to scan through the result set and calls for each row an explicit data conversion function (FillRow) that converts the internal CLR data into the equivalent SQL data types. Given this iterator-oriented contract between the query engine and the file wrapper function, fast streaming access to data stored in a FileStream is done by separating the file parsing logic from the filestreaming. Instead of reading individual lines from a text file with each MoveNext() call, we wrote a paging function that scans through the file in

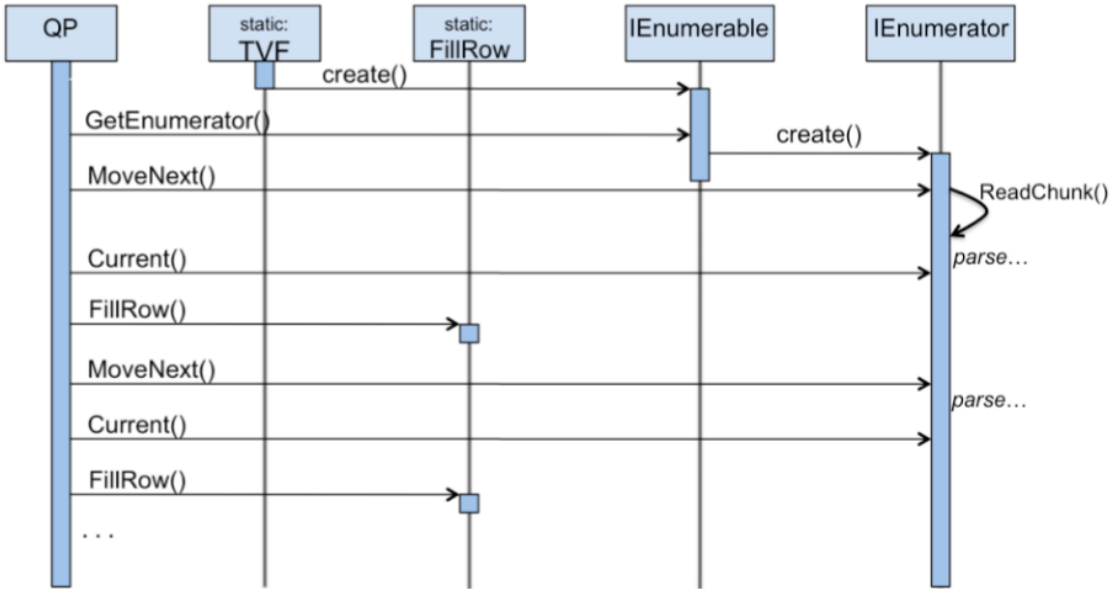

Figure 4.2: Calling sequence for parsing a FASTA file with short reads with a TVF

larger chunks of data (cf. the ReadChunk() call in Figure 4.2). The parsing code works on an internal byte buffer of the iterator that has been filled by the previous ReadChunk() call. An additional buffer-wrap check is needed in the parsing code, because the chunks are not guaranteed to be multiple of file entries; if the end of the chunk is reached before the parsing of the current entry is finished, a paging algorithm moves the current incomplete entry to the start of the buffer and fills the remaining buffer with the next chunk. Finally, we can stream fast through a FileStream BLOB by using the GetBytes() function together with the SequentialAccess flag that implements pre-fetching on FileStream data. The updated FileStream and TVF calls to pull data from an SRF file vs a FASTA file are shown in Figure 4.3 and Figure 4.4.

4.1.3.4 SRF Parser

Once we can directly access files using the FileStream handle above, we can write a parser for each specific file format we wish to interact with=, such as the SRF file format described in Section 2.5.1.3 that is used in older Short Read Archive files. Specifically, we wish to access the read data inside the files. Particularly, the sequence of bases found in a BASE chunk of a ZTR file and the

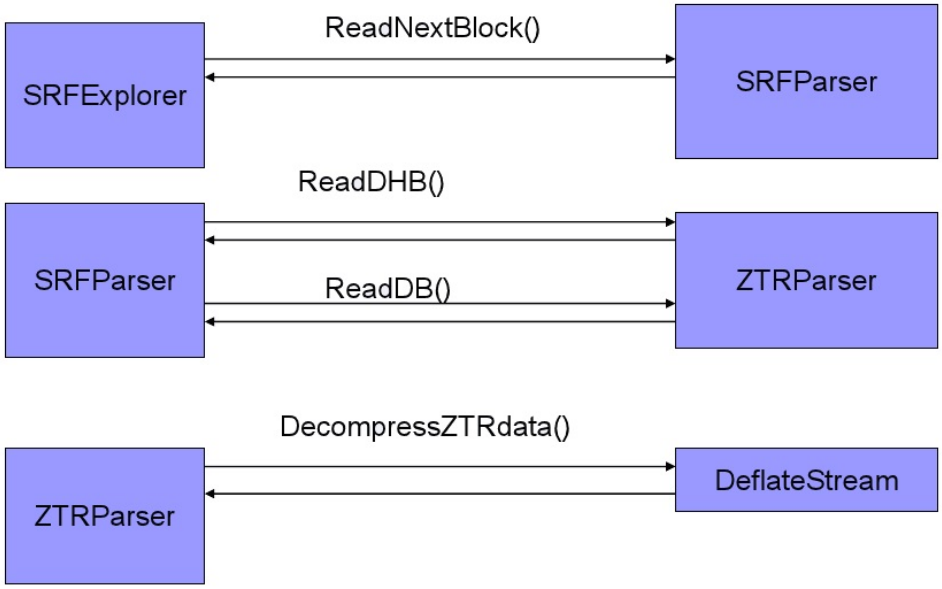

Figure 4.3: Calling sequence for parsing SRF files

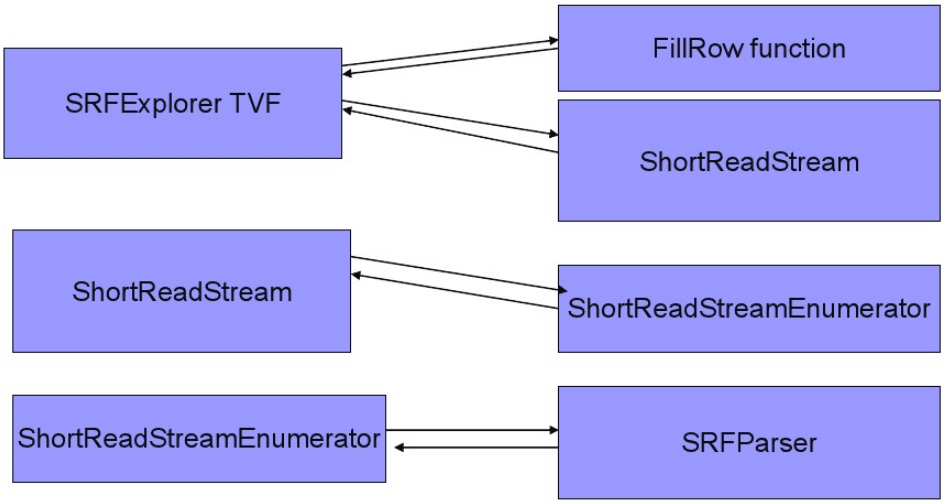

Figure 4.4: Updated calling sequence for parsing SRF files with a TVF

sequence of qualities found in the CNF1 chunk of a ZTR file. In order to access these data fields, we need to be able to read at least the container header and the data header blocks and data blocks of a SRF file. Then we need to parse the ZTR blobs inside the data blocks and decompress the data inside each blob. Our function call tree for reading an SRF file is shown in Figure 4.5. The **readDataBlock()** function in the SRF Parser subsequently calls the ZTR Parser, whose function call tree can be seen in Figure 4.6.

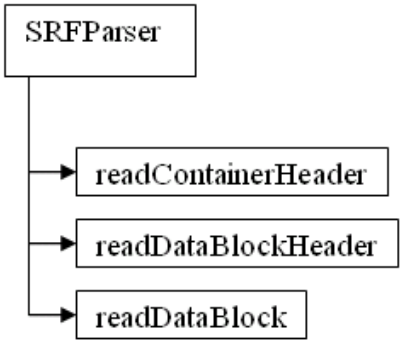

Figure 4.5: SRF Parser

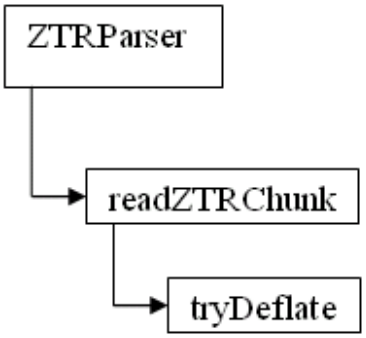

Figure 4.6: ZTR Parser

4.1.3.5 ZTR Parser

Given that an SRF file is a container for multiple ZTR (as described in Section 2.5.1.4) files, we also need a ZTR parser. This particular parser needs to be able to read compressed chunks as defined by the ZTR file format. However, for our purposes, we do not have to read all of them, so we can discard SAMP/SMP4, CNF4, TEXT chunk types when we extract the reads from these files. The chunks we are interested in are the following:

- HUFF chunks contain the Huffman coding trees.
- BASE chunks contain the read sequence in ZTR deflate format.
- CNF1 chunks contain the read qualities in ZTR deflate format.

Once the chunks have been retrieved, they have to be converted from the ZTR deflate format into the RFC1951 specification [21]. This involves melding the ZTR prefix found in the header blobs with the ZTR suffix blob found in the BASE or CNF1 chunk as seen in Figures 2.8 and 2.9. After the converted blob conforms to the RFC1951 specification, it can be decompressed using the HUFF chunk and the DeflateStream .NET class library. The ZTR Parser call tree is shown in Figure 4.6.

4.1.4 Reference Sequence Masking

The effect of repeat masking in reducing the size of the index can be seen in Table B.15 in the Appendix.

4.2 Indexing

Typically in a database system, the concept of indexing is applied to tables. In our implementation, we instead describe a **User-defined Index** for each reference file. An index is created based on our short read alignment method. Our Short read alignment method is relatively direct:

- **For Loop** The read file(s) are iterated through one read at a time. For each read, do the following steps:

    1. An alignment attempt for the read is carried out using the specified index according to a specified error model.
    2. If one or more matches are found, then this is reported to the aligned reads table with the start location of the alignment.

The specifics that come into play in step 1 are determined by:

1. The type of index used.
2. The type of error model used.

The two indices we use are the FM-Index and the Hash Index. For the FM-Index our error model is based on the number of backtracks we allow. For the Hash Index our error model is based on the initial definition of edit distance that we allow for during index construction. As such, our index creation behaviour becomes different:

- **FM-Index** One index created per reference sequence. This can then be stored and re-used.
- **Hash Index** One index created per edit distance per reference sequence. These can then be stored and re-used.

The schema for storing these indices is outlined in Tables 4.5 and 4.6. A separate storage table is kept which can have multiple entries per index. This is because we use a fragmented index for larger reference sequences as described in Section 4.3.2.2.

Value	Data type	Description
IndexID	Int32	Primary key for the index id, used in the
ReferenceID	Int32	Foreign key for reference sequence table. Can be retrieved for analysis and representation purposes
IndexTypeID	Int32	Foreign key for index type table.
EditDistance	Int32	Edit distance specified

Table 4.5: Index Table description schema

Value	Data type	Description
IndexID	Int32	Foreign key for the SNP as the tuple (RefID, ExperimentID, RefPos) may not be unique
indexfile	Filestream or varchar(max)	storage path of the index

Table 4.6: Index Table storage description schema

If Step 1 has one or more results, then these hit alignments are pushed into a table. This table is then used to generate a consensus sequence and identify possible SNPs. The table schema is outlined in Table 4.7. Note that the handling of quality is done before the alignment is pushed into this table.

Value	Data type	Description
ID	Int64	ID for this read alignment
ExpID	Filestream or varchar(max)	Foreign key for the experiment, used to identify all the alignments in a particular run
SR_RefPos	Int64	Position of the Read in the reference sequence. ExpID can identify which reference sequence, but we have duplicated the data into the alignment
SR_RefSeg	varchar(max)	Section of the reference sequence used for this alignment.
SR_Read	varchar(max)	Read sequence only.

Table 4.7: Read Alignments Table description schema

4.2.1 FM Index

The FM Index implementation we chose is based on the induced sorted suffix array that currently has three primary data structures:

1. BWT-string T
2. $C[c]$
3. $Occ(c,k)$

For exact search we also need to add checkpoint matrix, which calculates $LF[i]$ every chk distances. For k mismatch search, we need to add backtracking when we hit a false hit, with an implementation modification similar to bowtie and crossbow [50].

4.2.2 BWT Construction

BWT takes in input string, output string, temporary array and input length. BWT invokes Suffix Array Induced Sort method (SAIS) main a total of i times, where i is $log2(inputlength)$ SAIS main uses three steps. Algorithm 5 shows the full pseudo-code with the parameters defined in Table 4.8:

1. Step 1: reduce the problem in half by sorting all the lms sub-strings.
2. Step 2: solve the reduced problem, recurse if required.
3. Step 3: induce the result for the original problem.

Parameter	**Designation**	**Explanation**
Input String	S	string of n characters in an array $[0..n-1]$
Alphabet of S	$\|S\|$	sum of all the unique characters in S, usually A,G,C,T,N and control characters such as $
Suffix Array	SA	

Table 4.8: SAIS parameters

Algorithm 5 Suffix Array construction using Induced Sorting algorithm

function SAIS(S, SA)
  $t \leftarrow array[0..n-1]$ of boolean values
  $P1 \leftarrow array[0..n1-1]$ of integer values
  $S1 \leftarrow array[0..n1-1]$ of integer values
  $B \leftarrow array[0..||\Sigma(S)||-1]$ of integer ▷ $n1 = ||S1||$
  Scan S once to classify all characters as L- type or S- type int t
  Scan t once to find all the LMS-substrings in S into $P1$
  Induced sort all the LMS-substrings using $P1$ and B
  Name each LMS-substring in S by its bucket index to get a new shortened string $S1$
  if Each character in $S1$ is unique **then**
    Directly compute $SA1$ from $S1$
  else
    $SA\text{-}IS(S1, SA1)$ ▷ Fire a recursive call
  end if
  Induce SA from SA1;
  return SA
end function

4.2.3 Hash Index

If we do not have a specified error model, then a hash index search will be assumed to be a zero mismatch search. As such, we disregard normal indexing methods and index only non overlapping kmers. This way we reduce the total size of the index and can quickly discard inexact searches. The only concession to an error model of any kind is the use of a hit threshold h where we report an alignment of a read p against a reference S if we have h or more candidate regions within a given span. E.g. if we have read of length 100 bp, a kmer length of 25 bp, then for an exact non overlapping match, h should be 4. If we relax this threshold to make h be 3, then we still can include likely biologically relevant matches with gapped alignments of 1 to 25 bp, indicating a possible transposon incident where some genomic information has been partially copied by a biological process.

4.3 Short Read Alignment

As indicated in the previous section, we need to create indices to use them in short read alignment. This process refers to taking a read as input and positioning it alongside the reference genome. Depending on the index that is chosen, this alignment can be done in one step or two.

If using an FM-Index, then a match for each read can be carried out in a single process using an exact match, or k-distance (or k mismatches allowed) search as specified in Section 4.3.1.

If using a hash index, then the read is fragmented into fixed length k-mers (dependent on the k-mer length of the index) and a hash of each k-mer is looked up in the hash index. This gives a list of positions to each match. The best match is then calculated using the procedure in Algorithm 6.

4.3.1 Alignment using FM-Index

The FM-Index is an extension of the Burrows Wheeler Transformed file with a few extra structures, as explained in Section 2.7.1.7.

4.3.1.1 BWT Exact Search

Given large texts, a great deal of computation time would be spent on $LF(r)$ and $LFC(r,c)$ (described in chapter 2.7.1.7 with Algorithms 1 and 3, respectively) calling $Occ(c,r)$. To reduce the computation time, an extra data structure is created which keeps checkpoints for the $LF(c)$ calculation so that an $Occ(c,r)$ call only needs to iterate a maximum of *chk* times where *chk* is the checkpoint distance. Bowtie used a *chk* of 448 [50] and that is what we propose as well. This gives $Occ(c,r)$ a linear time calculation with means Exactmatch will take a linear time with respect to the length of the initial query p.

4.3.1.2 BWT k-Distance Search

Exactmatch is not sufficient for aligning genomic short reads because the best local alignment of a read may contain differences. Such differences may occur for several reasons, such as sequencing errors, mutations between the reference sequence and the subject organism or a combination of the two. To allow for such

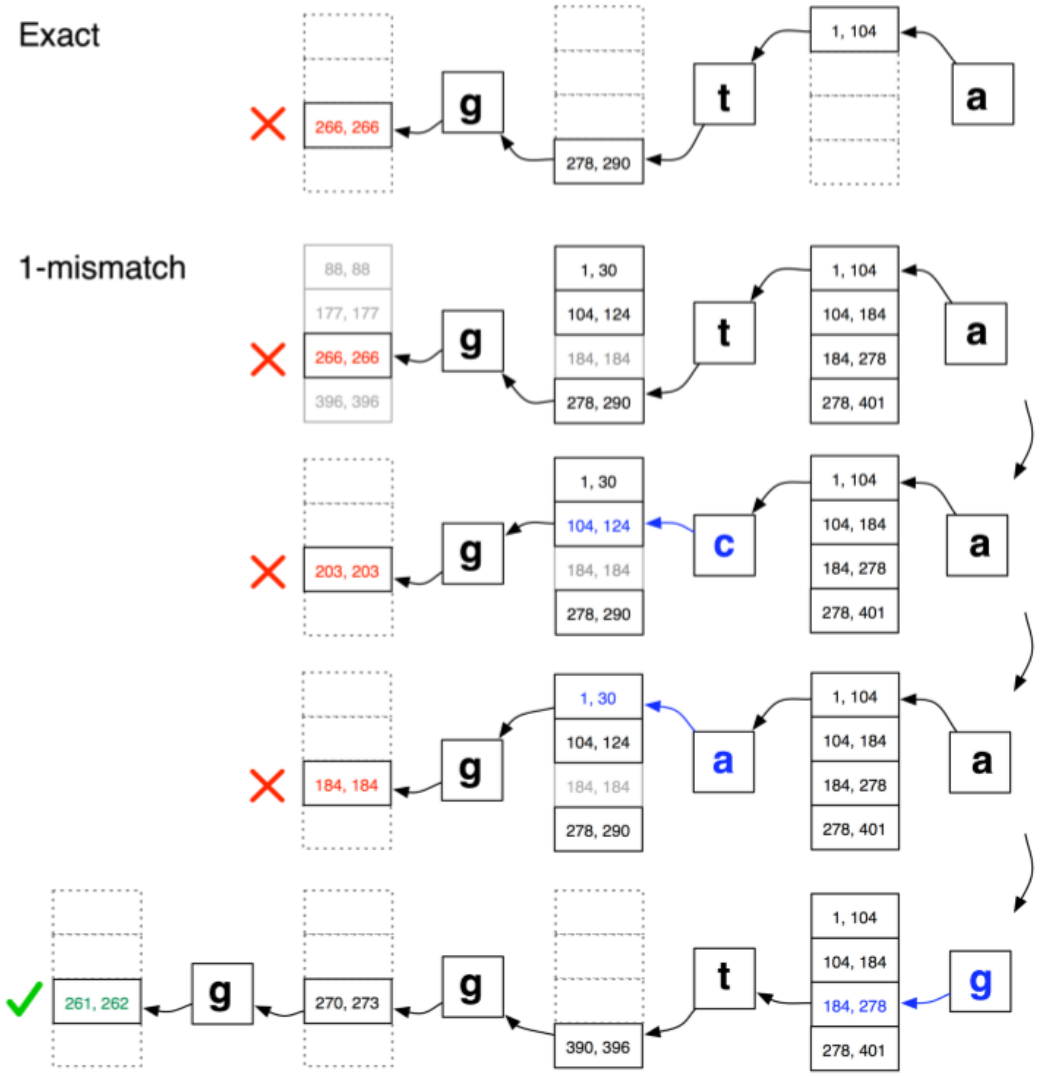

Figure 4.7: Example of how exact and 1-mismatch algorithms might proceed. We begin with an exactmatch search that fails to find a match and stops. Then the algorithm rewinds back and tries each possible base until it finds a match. Once a match is found, the algorithm proceeds to the next base to match.

differences, a sequence alignment algorithm must allow for some errors. Ideally, such a search algorithm must be able to support a robust and malleable error model. If the entire possible search space for a given error model is considered it would be exponential relative to the length of the read, so in practice, extensive pruning is required. Initially we will begin with an initial extension of the Exactmatch algorithm to allow for mismatches in a bowtie-like implementation. Then we will extend that further to allow for indels (i.e. gapped alignments) as well. In addition, if we have extra metadata about the sequence, such as quality data, this too should be considered as it can discern whether we have a genuine mismatch or sequencing error. Inexact search proceeds similarly to Exactmatch, calculating the BWT ranges for successively longer query suffix. The difference is that when the search arrives at an empty BWT range (indicating that the corresponding suffix does not occur in the initial text), the algorithm may select a previously examined query position and substitute a different base there, introducing a potential mismatch into the alignment at that position. This is called a **backtrack**. After executing a backtrack, the Exactmatch search resumes from just to the left of the substituted position. The number or the quality of these backtracks can be defined by the user. The user defines a **pruning policy** that is consulted whenever a backtrack occurs. An example pruning policy would be a maximum of 2 percent errors (i.e, 1 errors per every 50 bases) or 4 percent errors but only similar molecule mutations (i.e A to G, C to T and vice versa). In addition to the pruning policy, the search will never backtrack to points in the search space that are already known to be associated with empty BWT ranges. Doing so is pointless, since the emptiness of the range implies that no alignments of the hypothesized type are possible in the reference. This policy is called **empty range pruning**.

Figure 4.7 illustrates an example of a pruning policy that allows a maximum of 1 mismatch. In this case, the query string is GGTA. It does not have an exact match to the text, but it does have a 1-mismatch alignment where a G in the reference is substituted for an A in the read. ParIDs of number represent the sp, ep parids calculated in an iteration of EXACTMATCH (Algorithm 4). Vertical sets of four parids represent the pairs calculated for each character A, C, G and T (from top to bottom). Blue numbers and letters represent the potential mismatch introduced as part of a backtrack. Empty ranges are shown in red if they trigger a

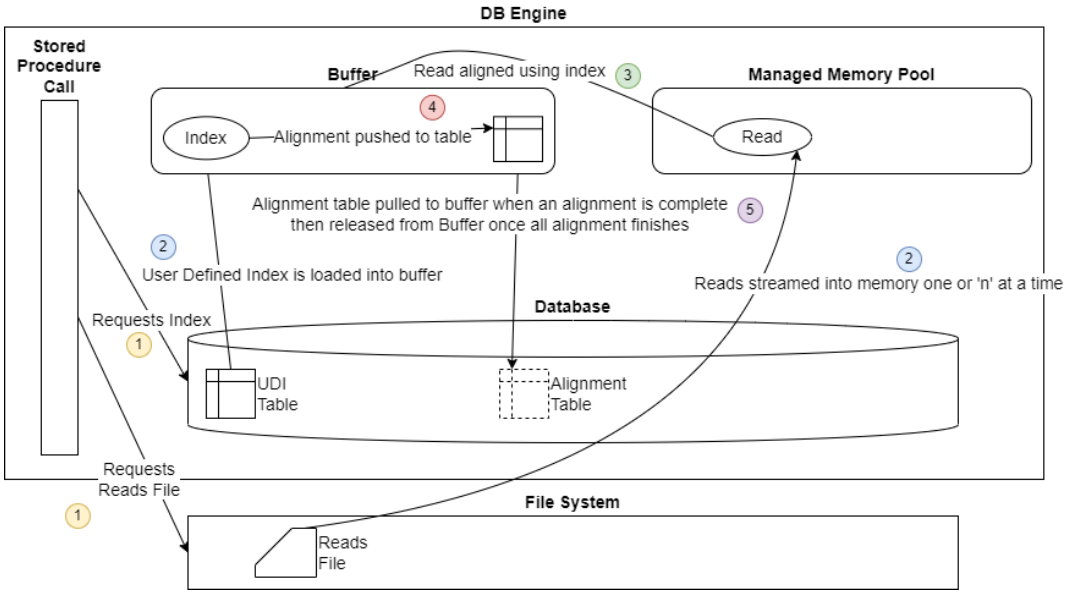

Figure 4.8: Data movement between system components during Short Read Alignment

backtrack, or in grey if they do not. Empty boxes show ranges left un-calculated due to policy pruning. The final reported range is shown in green.

4.3.1.3 Expected Data movement during Short Read Alignment

The stored procedure pulls the index into memory a chromosome at a time, then opens the reads file from disk as a streaming option. Each read is pulled into memory, then aligned against the index. If an alignment is successful within the given error model specified for the index, then an alignment entry is inserted into the alignment Table. Given the frequency of the hits to the Alignment table, it will be resident in memory. Each read is discarded from memory after it has been aligned. This procedure is outlined in Figure 4.8.

4.3.2 Alignment using Hash Index

A hash index is a simpler alignment task, but it takes two steps. First a look-up of the hash index to identify possible hits, then a dynamic programming extension of viable hits.

4.3.2.1 Hash Index Error Model Search

Here we adapt the approach taken by SNAP as described in Section 2.7.1.3. As this is usually the most expensive portion of the alignment, a few techniques were required to reduce the cost of local alignment checks:

1. Firstly, with the longer seeds, as explained previously, it is less likely for there to be false positive hits, so we will find a good alignment quickly. We chose a length of 25 bp.

2. Secondly, most applications only need to find the first or second best candidate locations for an alignment, i.e. the positions where the edit distance between the read and the reference sequence is smallest and second smallest. If the best alignment is sufficiently above some threshold better than the second best, then the read unambiguously maps to the first location. However, if the edit distance difference is below another threshold then the read cannot be confidently mapped to either place and may be dealt with differently.

3. Thirdly, we can eliminate some locations solely on the number of seeds that match there, without performing an edit distance check. For example, if three of the non-overlapping seeds from a read do not match with a particular candidate location, then that candidate location must be at least edit distance 3 from the read, because there must be at least one difference in each of the non matching seeds [4].

The primary algorithm is shown in pseudocode in Algorithm 6 which uses the three observations described above. The algorithm takes in five parameters which control what is considered a good alignment and which seeds it tries. These parameters are shown in in Table 4.9. The algorithm returns one of three values for each read:

1. Single hit if it finds only one good alignment for the read

2. Multiple hits if finds two or more alignments that are too close in score

3. Not found if no appropriate hits are found

Algorithm 6 Sequence alignment based on Hash Index

function HASH INDEX BASED ALIGNMENT(*read*)
  $d_{best} \leftarrow unbound$
  $d_{second} \leftarrow unbound$
  for $i = 1...n$ **do**
    $S \leftarrow i$-th seed of read
    if # of index entries for $S \leq h_{max}$ **then**
      for l belongs to set of locations of S in index **do**
        $p \leftarrow l-$ offset of seed i from start of read
        $SeedsHitting[p] \leftarrow SeedsHitting[p] + 1$
      end for
      $p \leftarrow$ unscored location with most seeds hitting
      if $d_{best} > d_{max}$ **then**
        $d_{limit} \leftarrow d_{max} + c - 1$
      else if $d_{second} \geq d_{best} + c$ **then**
        $d_{limit} \leftarrow d_{best} + c - 1$
      else
        $d_{limit} \leftarrow d_{best} - 1$
      end if
      $g \leftarrow EditDistance(Read, Reference[p], d_{limit})$
      update d_{best} and d_{second} based on newscored d
      if $d_{best} < c$ and $d_{second} < d_{best} + c$ **then**
        return multiple hits (we have 2 hits within distance c and no better hit can be confidant)
      else
        if # non overlapping seeds tests $\geq d_{best} + c$ **then**
          score remaining locations and break (any unscored locations will have too high a distance)
        end if
      end if
    end if
  end for
  if $d_{best} \leq d_{max}$ and $d_{second} \geq d_{best} + c$ **then**
    return single hit with location with best score as defined by d_{best}
  else if $d_{best} \leq d_{max}$ or all seeds had $> h_{max}$ entries **then**
    return multiple hits
  else
    return not found
  end if
end function

Parameter	Designation	Explanation
Seed Size	s	Length of seeds in bp
Seeds to try	n	Number of seeds to try for each read
Max distance	d_{max}	Maximum edit distance from reference to allow for an alignment
Confidence	c	Difference in edit distance between the best and second best alignments needed to report it as confidently aligned
Max hits	h_{max}	Maximum index locations to check for a seed

Table 4.9: Hash Index based search parameters

Algorithm 7 Edit distance calculation function

```
function EDITDISTANCE(Read, Reference[p], d_limit)
    d ← 0
    for i = 1...Read.length do
        if Read[i] ≠ Reference[p][i] then
            d ← d + 1
        end if
    end for
    return d
end function
```

Algorithm 8 Edit distance function - Dynamic Programming

```
function EDITDISTANCE(Read, Reference[p], d_limit)
    d ← 0
    (alignment, mismatches) ← SmithWaterman(Read, Reference[p])
    d ← mismatches
    return d
end function
```

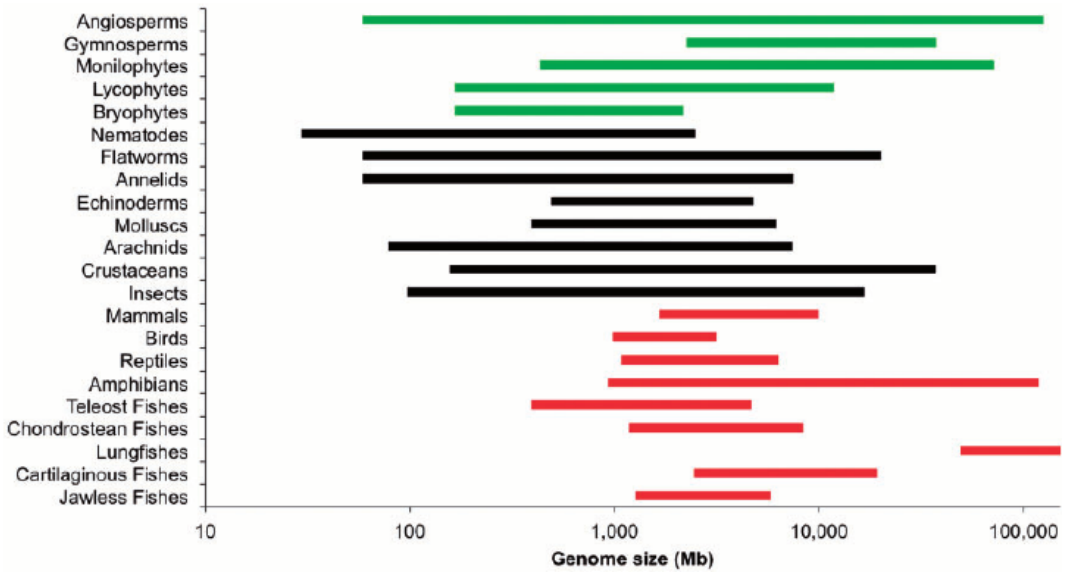

Figure 4.9: Log scale representation of genome sizes for various organism types

4.3.2.2 Index Fragmentation

We noticed that some indexes were taking far longer to build than others. After running the evaluation tests shown in Chapter 5.3.3, we found that it was a memory restriction at around the 130 million character mark during index creation and 60 million during read alignment. As can be seen in Figure 4.9, genome sizes can be a hundred times larger than the human genome, and even the largest currently published genome of the White Spruce or *Picea glauca* already reaches 20.8 Gbp [67]. In order to avoid system thrashing due to page migration from RAM to disk we determined a different approach was needed.

The approach we chose was to fragment the index by splitting the input file by a defined limit of basepairs. The limit we chose was 60 Mbp, but this can be adjusted depending on the amount of memory available in this system. An index is then created on each fragment of the input file and when a query is carried out, it is carried out on each fragment, as seen in Figure 4.10. When a query is carried out on any segment past the first, the result is adjusted to reflect the position of the returned hit in the original reference sequence. This way hits will align properly along the reference sequence. Since queries are run multiple times, we expect to see a linear increase in runtime relative to the number of fragments. This behaviour can be observed in Chapter 5.3.3.

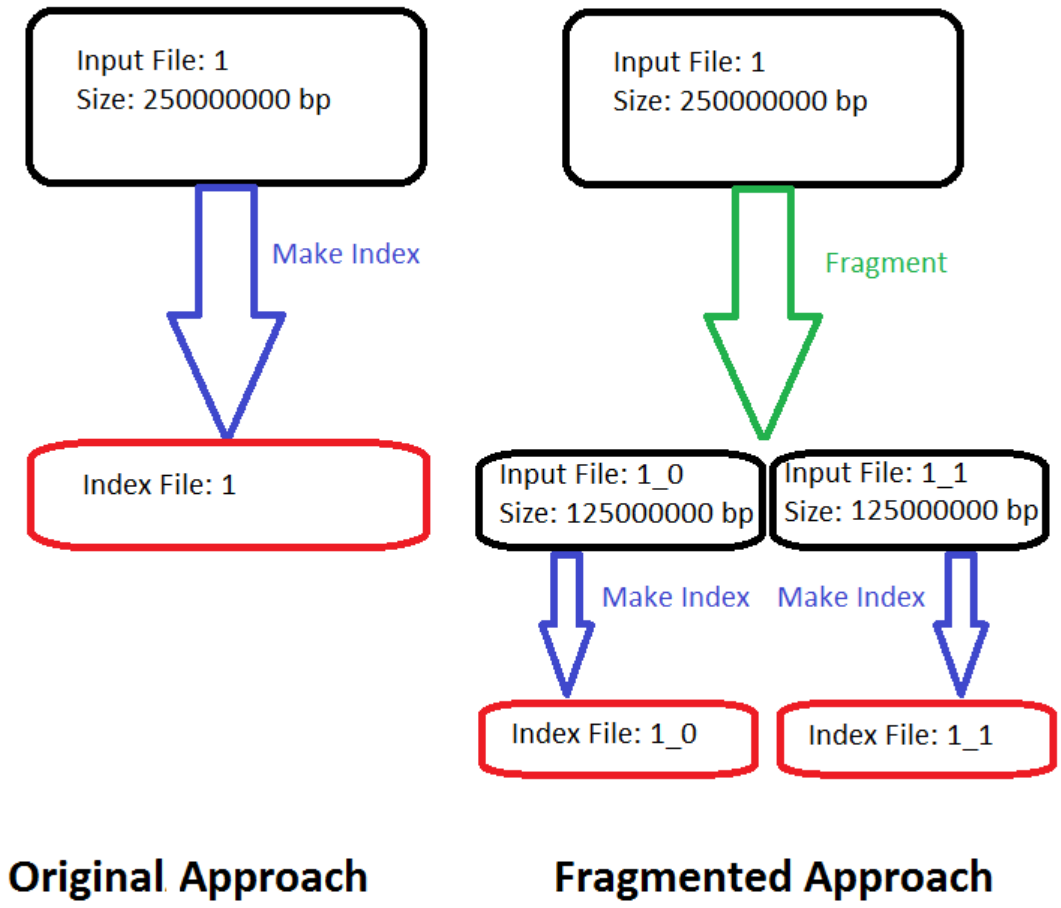

Figure 4.10: Original and new approach to indexing larger genomes or genome fragments

4.3.3 Dynamic Programming

Dynamic programming is a technique used in many fields to solve complex problems by dividing the problem into several smaller problems and computing and storing the solutions to each of the smaller sub-problems. The solutions for the sub-problems are then combined and a solution to the initial complex problem is then found. In bioinformatics, it is used for sequence alignment. Its primary representation is in the Smith-Waterman (SW) and Needleman-Wunsch (NW) alignment algorithms.

4.3.3.1 Smith-Waterman

The SW algorithm is a local alignment algorithm that finds the optimal local alignment between two sequences based on a given scoring matrix.
The algorithm itself rests on the use of a two dimensional scoring matrix H with one axis being the reference sequence d and one axis being the query sequence q. Each sequence string is prepended with a zero character which allows the first row and first column of the scoring matrix to be set to zero. Then each position in the matrix is calculated based on the following equation:

$$H_{i,j} = max \begin{Bmatrix} 0 \\ H_{i-1,j} - G_{init} \\ H_{i,j-1} - G_{init} \\ H_{i-1,j-1} + W(q_i, d_j) \end{Bmatrix}$$

Each horizontal or vertical movement computation is based on a gap score. Biologically speaking, it is more common that once an insertion or deletion has occurred, an extension of said insertion or deletion in more likely.

There are two commonly used gap models:

1. **Linear Gap Penalty**: Each consecutive gap of length k has a penalty score of $k * G_{init}$, where $k > 0$.

2. **Affine Gap Penalty**: Each gap of length 1 has a penalty score of G_{init} and each consecutive gap of length $k > 0$ has a penalty score of $G_{init} + (k-1) * G_{ext}$

	A	**C**	**G**	**T**	**N**
A	3	1	-3	-1	-3
C	-1	3	-3	1	-3
G	-1	-3	3	1	-3
T	-3	-1	1	3	-3
N	-3	-3	-3	-3	-3

Table 4.10: Simple scoring matrix based on DNA structure for our DNA alphabet of **A, C, G, T, N**

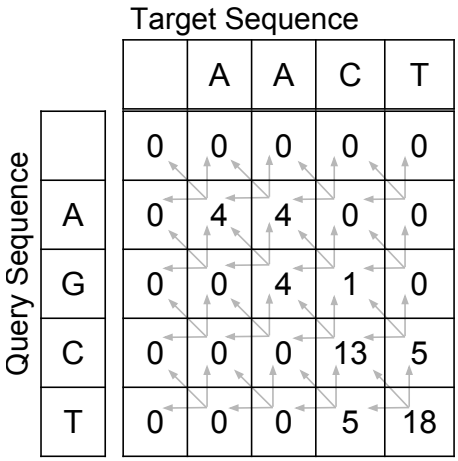

Figure 4.11: A sample of matrix computation in Smith-Waterman

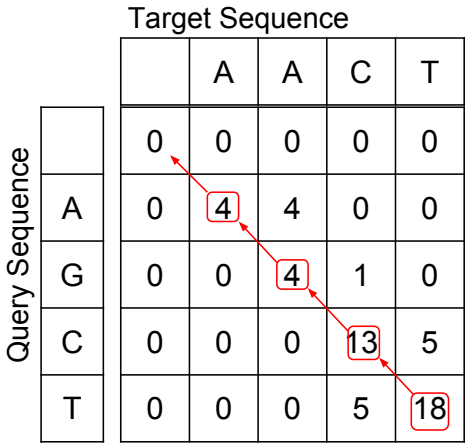

Figure 4.12: A sample of matrix traceback in Smith-Waterman

Each diagonal movement computation is based on a score profile W which contains all the characters found in the alphabet of q and d. We used a basic scoring matrix for matching bases derived from the stucture of bases themselves as seen in Table 4.10.

Once the scoring matrix has been fully computed, a traceback operation is carried out from the maximum score until a score of zero is reached. Figure 4.11 shows a completed score matrix H and Figure 4.12 shows an example traceback operation on the same completed score matrix H.

4.3.3.2 Gotoh Extension

While the SW algorithm works fine for a linear gap penalty, some modification is required to compute for an affine gap penalty. An affine gap penalty means that an initial gap carries an initiation penalty and the continuation of that gap

carries a lower penalty. This reflects the biological fact that once an error is introduced during a genomic interaction event, further errors are more likely. These modifications were introduced by Gotoh [33] and involve adding two new scoring matrices E and F to keep track of vertical gaps and horizontal gaps. The score computation for $H_{i,j}$ is modified as follows:

$$H_{i,j} = max \begin{Bmatrix} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + W(q_i, d_j) \end{Bmatrix}$$

where $E_{i,j}$ and $F_{i,j}$ are computed as follows:

$$E_{i,j} = max \begin{Bmatrix} E_{i,j-1} - G_{ext} \\ H_{i,j-1} - G_{init} \end{Bmatrix}$$

$$F_{i,j} = max \begin{Bmatrix} F_{i-1,j} - G_{ext} \\ H_{i-1,j} - G_{init} \end{Bmatrix}$$

The values for $H_{i,j}$, $E_{i,j}$ and $F_{i,j}$ are equal to 0 when $i < 1$ or $j < 1$.

4.3.3.3 Needleman-Wunsch Differences

The first widely used dynamic programming algorithm for sequence alignment was the Needleman-Wunsch (NW) algorithm [74]. The NW algorithm looks for a global alignment between two sequences. As the SW algorithm is based on the NW algorithm, it uses similar principles. The key differences are as follows:

1. **Initialization**: Only the origin position $H_{0,0}$ of the scoring matrix is set to 0. The remainder of the row can be computed.

2. $H_{i,j}$ **calculation**: The requirement for a minimum score of zero is removed.

3. **Traceback**: The starting position for the traceback is set to the final point of the scoring matrix $H_{|q|,|d|}$, not the maximal point. The termination condition for the traceback is now when both $i = j = 0$.

4.3.3.4 SIMD Acceleration of Dynamic Programming

The process of performing a dynamic programming calculation is one of the most expensive parts of aligning a read (see Figure 5.22). As a demonstration of the optimisations we can pursue using stored procedures, we used SIMD processes to acceleration our dynamic programming calculations. In our implementation, we adapt Farrar's method described in Section 2.8.1.1 to create a query profile and perform our dynamic programming alignment.

Usually the alphabet for both the query and the target sequence are the same. In our (and most other) implementation, our expected alphabet is {**A**, **C**, **G**, **T**, **N**, **-**} for DNA sequences. So, our query profile would take the form of the reference alphabet size of 6 times t vectors of size determined by the query size shown in Table 4.11. The pseudocode for the Farrar method is outlined in Algorithm 9 for query profile creation and Algorithm 10 for score matrix calculation.

Algorithm 9 Query Profile creation

```
function CreateQueryProfile(Query, Reference, bitDepth)
    segLen ← (Query.length + bitDepth − 1)/bitDepth          ▷ integer value set
    for all a in Reference.Alphabet do
        h ← 0
        for i = 0..segLen do
            j ← i
            for k = 1..bitDepth do
                if j > Query.length then
                    queryProfile[a][h] ← 0
                else
                    queryProfile[a][h] ← W(a, q[j])
                end if
                h ← h + 1
                j ← j + segLen
            end for
        end for
    end for
    return queryProfile
end function
```

Algorithm 10 Score Matrix Calculation

```
function TraverseScoreMatrix(Query, Reference, vQueryProfile, segLen)
    for i = 0..Reference.length do
        vF ← < 0, ..., 0 >                                   ▷ Zero Vector
        vH ← vHStore[segLen − 1] << 1
        Swap(vHLoad, vHstore)
        for j = 0..segLen do
            vH ← vH + vQueryProfile[i][j]
            vMax ← max(vMax, vH)
            vH ← max(vH, vE[j])
            vH ← max(vH, vF)
            vHStore[j] ← vH
            vH ← vH − vGapOpen
            vE[j] ← vE[j] − vGapExtend
            vE[j] ← max(vE[j], vH)
            vF ← vF − vGapExtend
            vF ← max(vF, vH)
            vH ← vHLoad[j]
        end for
        vF ← vF << 1
        j ← 0
        while AnyElement(vF > vHStore[j] − vGapOpen) do
            vHStore[j] ← max(vHStore[j], vF)
            vF ← vF − vGapExtend
            if ++ j >= segLen then
                vF ← vF << 1
                j ← 0
            end if
        end while
    end for
    return vHStore
end function
```

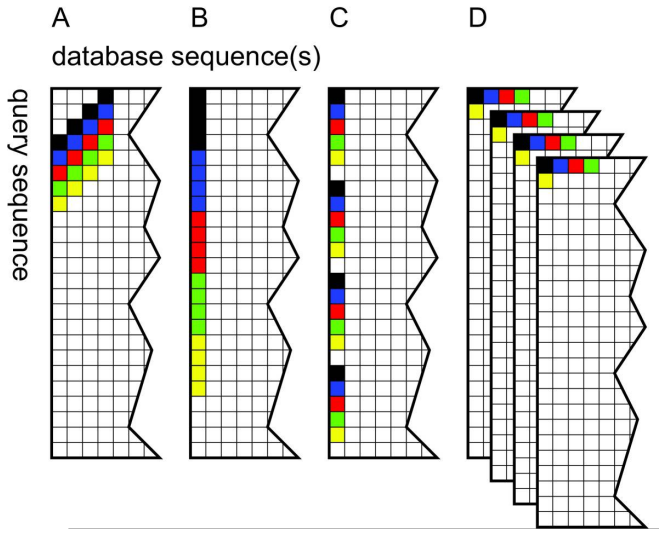

Figure 4.13: Examples of various query profiles. **A**: Vectors along a diagonal [110]; **B**: Vectors along the query [91]; **C**: Striped Vectors along the query [29]; **D**: Vector across multiple queries [90]

bp	max. score	bit-depth	vector type	elements	
				sse	avx
15	135	8	byte	16	32
35	315	16	ushort	8	16
50	450	16	ushort	8	16
75	675	16	ushort	8	16
100	900	16	ushort	8	16
150	1350	16	ushort	8	16
300	2700	16	ushort	8	16
500	4500	16	ushort	8	16
1000	9000	16	ushort	8	16
1500	13500	16	ushort	8	16
3000	27000	16	ushort	8	16
5000	45000	16	ushort	8	16
10000	90000	32	uint	4	8
20000	180000	32	uint	4	8
40000	360000	32	uint	4	8
$1x10^5$	$9x10^5$	32	uint	4	8
$1x10^6$	$9x10^6$	32	uint	4	8
$1x10^7$	$9x10^7$	32	uint	4	8
$1x10^8$	$9x10^8$	32	uint	4	8
$1x10^9$	$9x10^9$	64	long	2	4
$3x10^9$	$27x10^9$	64	long	2	4

Table 4.11: SIMD query profile assigned bit-depths, vector types and elements

4.3.3.5 Our Implementation

Our implementation follows Farrar's approach, but has a few differences due to the limitations imposed by the System.Numerics.Vector library in .NET 4.6. Our approach is outlined in Algorithm 11 on page 117.

4.3.4 Development Challenges with using GPUs in SQL Server

As part of my investigation into the feasibility of hardware extensions, I attempted to do not only the SIMD acceleration that I present in my work, but also a GPU extension based on existing .NET libraries. This was partially successful in that I was able to make some implementation headway, but unable to run a comparative analysis with some of the existing tools due to implementation issues. However, I believe that specifics regarding the challenges (and potential solutions) faced would be beneficial and so I have included them here.

- **.NET Compatibility** Most CUDA API frameworks on .NET that also work with the latest CUDA APIs require .NET version 6. This is problematic as SQL Sever 2018 and SQL Server 2022 only support .NET Framework (the latest of which is .NET Framework 4.8).
- **GPU kernel compilation** GPU kernels compiled from **.cu** to **.ptx** with **nvcc** are not perfect and need to be edited to replace the internally generated process name with the assigned kernel name.
- **Assembly loading** Due to the nature of the CLR, some resources are locked up by the SQL Server internal processes. This includes particular library calls. Any assembly that works with an external resource, such as a GPU or GPU kernels stored in the filesystem, must be added to a list of trusted assemblies otherwise the standard method of publishing stored procedures from Visual Studio will not work and any stored procedures developed in the IDE will not be deployed to the SQL Server instance.

Solving these problems, the following contributions were made:

- Given the lack of a proper CUDA API for .Net Framework 4.x, a more bare-bones solution must be found. This is why I opted to use the **ManagedCuda** framework. It has a port of all the CUDA API calls, but it does

Algorithm 11 Our Score Matrix Calculation Implementation

```
function TraverseScoreMatrixNew(Query, Reference, vQueryProfile, segLen, gapPenalty)
    vZero ← < 0, ..., 0 >                                   ▷ Zero Vector
    vGap ← < gapPenalty, ..., gapPenalty >                  ▷ GapCost Vector
    for i = 0..Reference.length do
        for j = 0..segLen do
            if i == 0 then
                vH ← max(vZero, vQueryProfile[j][Reference[i]])
            else
                if j == 0 then
                    vH ← matrixProfile[segLen − 1][i − 1]
                    vH ← vH >> 1
                    vH ← max(vZero, vH + vQueryProfile[j][Reference[i]])
                else
                    vH ← max(vZero, matrixProfile[j − 1][i − 1] + vQueryProfile[j][Reference[i]])
                end if
                vE ← matrixProfile[j][i − 1] − vGap
                vH ← max(vH, vE)
            end if
            matrixProfile[j][i] ← vH
        end for
        changeFlag ← false
        for j = 0..segLen do
            if j == 0 then
                vH ← matrixProfile[segLen − 1][i]
                vH ← vH >> 1
                vF ← vH − vGap
            else
                vF ← matrixProfile[j − 1][i] − vGap
            end if
            greaterThanFlag ← greaterThanAny(vF, matrixProfile[j][i])
            changeFlag ← changeFlag || greaterThanFlag
            vH ← max(vF, matrixProfile[j][i])
            matrixProfile[j][i] ← vH
            if (j == (segLen − 1)) ∧ changeFlag then
                j ← −1
                changeFlag ← false
            end if
        end for
    end for
    return matrixProfile
end function
```

not provide a framework or API of its own to interact with CUDA-capable GPUs. This results in some very obtuse error messages at times which can cause delays in development as they have to be debugged.

- As part of the basics for kernel usage in gpGPU programming, each function is assigned a particular symbol or kernel name. When loading a kernel as a resource, this name has to be provided so that it can be invoked. If the provided name does not match the invoked name, then the kernel will not load and an error message with 'ErrorNotFound' will be output when trying to created a context with said kernel. In order to use the CUDA code in this scenario, it must first be compiled into device specific **.ptx** files. While documentation states that the compiler will match the kernel label in the **.ptx** file [79] to the provided name in the **.cu** code this is sometimes not the case and the compiler adds extra special characters as prefixes and suffixes. A series of screenshots of this issue can be seen in Figure 4.14. Thus, after compilation, all instances of the kernel name in the **.ptx** file must be changed to match the kernel name in the **.cu** file.

- .NET based stored procedures are handled through the loading of DLL assemblies into the database engine. This process is somewhat exhausting to set up, but can be done with some assistance from the Visual Studio IDE.

    1. Identify the dependency tree for the assemblies about to be loaded. If the dependency has DLLs that are outside of the .NET framework libraries, they must be loaded first before the target assembly using these same methods.
    2. For each assembly DLL file generate a SHA512 checksum to act as a unique id for the assembly.
    3. Use the **sp_add_trusted_assembly(checksum, idstring)** function to add the assembly to a list of trusted assemblies in SQL Server.
    4. Use the Visual Studio IDE to publish the assembly to the target SQL Server instance database. This will generate and execute a script that will register the assembly contents and the defined stored procedures, functions and data types that the assembly describes as usable objects in SQL Server.

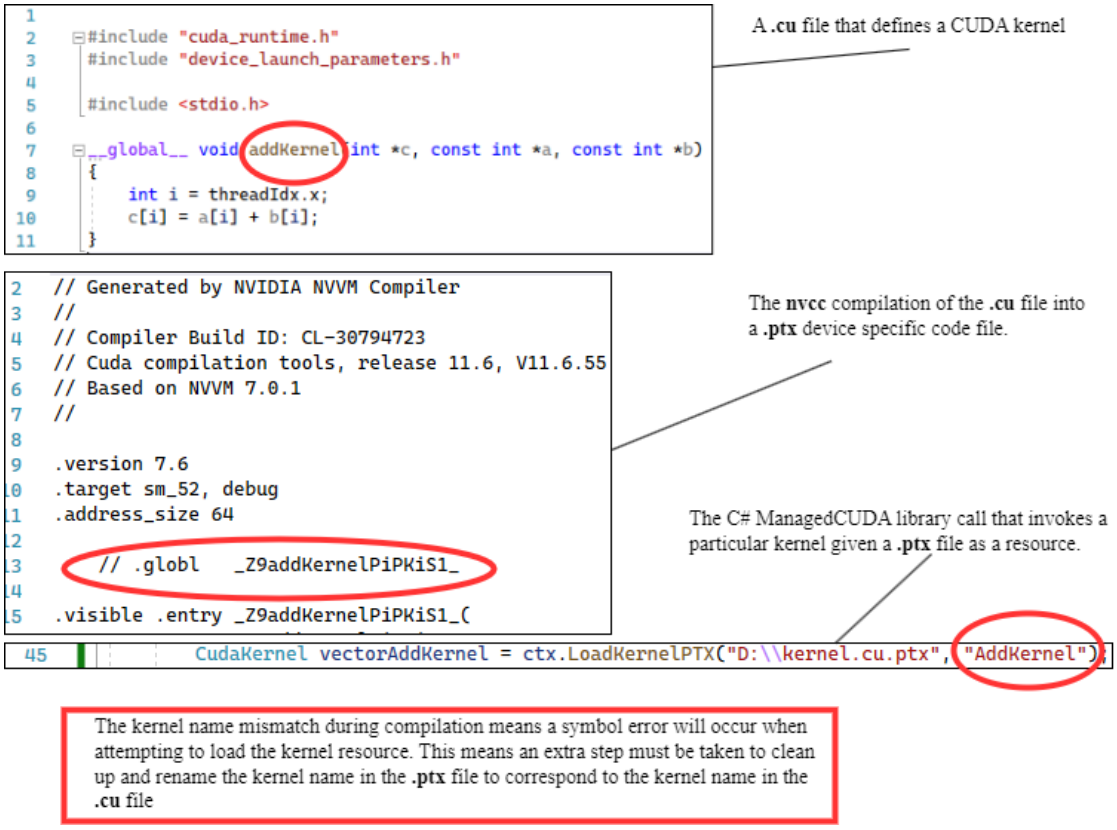

Figure 4.14: Screenshot of what happens during compilation of a .cu file into a .ptx file

4.4 Consensus Sequence Generation

Most of the examples of MSA or consensus sequence generation before NGS occurred using unbound instances, but with the advent of NGS technologies, the most prominent example of consensus sequence generation is the sliding window approach. Since the sequences have already been aligned, a single sliding window approach can be used along with quality scores to generate a consensus sequence as shown in Figure 4.15.

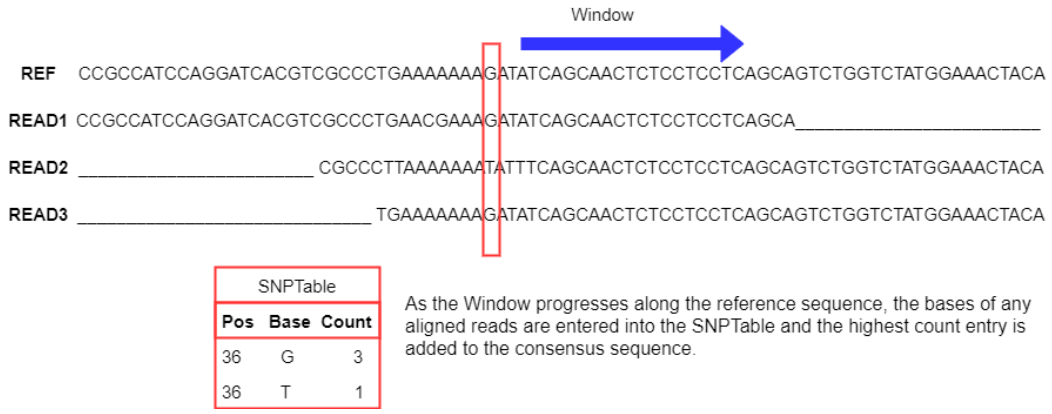

Figure 4.15: Example of how a consensus sequence is generated from a series of reads

4.4.1 Sliding Window

The default behaviour of the sliding window approach is described in Algorithm 12. In order to get a reads data set appropriate for the sliding window, a select statement is carried out on the Read Alignments table:

```
SELECT *
FROM ShortReadAlignments
WHERE ExpID = $expid
ORDER BY SR_RefStartPos;
```

Once the aligned reads are ordered, the sliding window approach can begin. For each position in the reference genome, a check is made against any available aligned reads. We start at the first position and continue until the last position of the reference sequence, each time advancing the window by 1 position. In most cases, the window is of length 1. The reason for a different window length would be for performance improvements, but would involve extra checks or boundary decisions attached to read alignment stage.

At each position, check all the available read bases at that position. This is achieved by maintaining a list of current reads based on the current window position on the reference sequence, and a start position and end position check on the currently available reads (which never needs backtracking because the reads are ordered by their SR_RefStartPos field). The counts of each different

viable base (A,G,C,T) in the current window that is over a quality threshold is tallied. Whether or not quality is checked depends on the reads input. If reads come from a FASTQ format, then quality values are present and are checked. If reads come from a FASTA format, then quality values are not present and are not checked. A default quality threshold of Q30 (1 incorrect base in every 1000 accepted error rate) is used as it is the baseline for NGS sequencing. The algorithm can be modified to accept a custom quality threshold if needed. Once all bases have been tallied, the highest count base is selected as the consensus base. This base is appended to end of the consensus sequence. If that base does not match the reference sequence base, then it is also added to the SNP Table. A toggle can be set to also add the lower count bases to the SNP Table with attached lower quality values (to indicate their lower confidence). In this instance, the aggregate quality is also recorded if further quality thresholds need be applied at the SNP calling level.

Once the process is complete, the completed consensus sequence is serialized and saved to a given location. This is by default a FASTA file with the ExpID and experiment details in the ID field. The reason why it saved directly to a file is that so it can immediately be used in other downstream applications and analyses. Alternatively, it can be saved in one of two other methods. The first is a managed FILESTREAM varbinary(max) (which has a maximum size of 4gb). The second is an internal BaseSequence representation (which has a maximum blob size of 2^{31} - 1 bytes). These can be triggered by a toggle and an enum (indicating un-managed FASTA file, managed Filestream BLOB, or BaseSequence: [FASTA, BLOB, BASESEQ]).

4.4.2 Optimisations

Some optimisations are possible for this process. As each position of the consensus sequence is independent of each other position, it is possible to open multiple windows. This does create extra accesses on the AlignedReads table however. In addition, a non-clustered index can be created to speed up the result of the ShortReadAlignments table:

```
    CREATE INDEX sra_expid_refpos
ON ShortReadAlignments(exp_id, sr_refpos);
```

Algorithm 12 Consensus sequence generation

```
function AGGREGATEALIGNMENTS(Reads, Reference)
    ConsensusSequence ← empty
                                        ▷ Sliding window along the reference genome
    for i = 1..Reference.Length do
        count ← 0
        Count[] ← empty ▷ Array of Reads, assume starting position has already
been defined, iterate through all reads in this position
        for m..Reads.Size do
            if Reads[m][i] = Reference[i] then ▷ The Quality of the read is above
the expected value
                if Reads[m][i].quality > ThresholdQuality then
                    count++
                end if
            else            ▷ The Quality of the read is above the expected value
                if Reads[m][i].quality > ThresholdQuality then        ▷ We did not
match the reference, but we may match with all the other reads
                    Count[Reads[m][i]]++
                end if
            end if
        end for
        ConsensusCharacter ← max(Count[])      ▷ The Final consensus character
        ConsensusSequence[i] ← ConsensusCharacter
        if Reads[m][i] ≠ Reference[i] then
            Update SNP Table with Reads[m][i]          ▷ Update the SNP Table
        end if
    end for
    return ConsensusSequence
end function
```

4.5 SNP Calling

While SNP calling is exhaustively investigated in Section 2.6.3, we do not have to go through many of the processes described in that section. In our implementation, we actually carry out our SNP calling during consensus sequence generation. Each time our sliding window passes over a particular position in the reference sequence, the aligned reads are inspected. If a base that is different to the reference sequence is seen to have occured in multiple reads, then it is inserted into the SNP table. In order to retrieve our SNPs, we just invoke a SQL call for that particular experiment as described in the next section. Since we use quality as part of finding the alignment, it is not usually reported into the Read Alignments table and thus does not usually make its way to the SNP table. The SNP table schema is described in Table 4.12:

Value	**Data type**	**Description**
SNPID	Int64	Primary key for the SNP as the tuple (RefID, ExperimentID, RefPos) may not be unique
ExperimentID	Int32	Foreign key for experiment. Can be used to get the error model
RefSeqID	Int32	Foreign key for reference sequence table. Can be retrieved for analysis and representation purposes
RefPosition	Int32	Position of the SNP in the reference sequence
OriginalChar	Byte	Original character found in the reference sequence
ReportedChar	Byte	Reported character in the consensus sequence
ReportedQuality	Byte	Quality of the reported character (optional)
AggregateQuality	Double	A weighted aggregate function computation of the reported quality of this SNP (optional)

Table 4.12: SNP Table Schema

4.5.1 Retrieving SNP Results

When an experiment is run and a consensus sequence is generated, there may be some entries added into the SNPTable.

These can be retrieved by using the **ExpID** PK for the experiment. The SQL Statement for this query is:

```
SELECT *
  FROM SNPTable
 WHERE ExpID = $expid;
```

An 'ORDER BY RefSeqID, RefPosition' clause can be included to get the SNPs in order for each inspection or for pushing the results to some downstream analysis like phylogenetic graph creation.

4.5.2 Indels

As aspect of the original pipeline that we chose to not implement is the detection of insertions or deletions of subsequences (indels) of length greater than one. Existing Indel calling tools are not as accurate as SNP calling tools [32] (99+% for SNPs vs 80 % for Indels within 1–10 bp), so we chose omit this step to streamline our sample implementation. The methodology typically used to identify indels is similar to the windowing function during consensus sequence calling. A model of base likelihood is used (either a hidden Markov model or a naive Bayes model) to determine with if the current base or base sub-sequence are likely to have occurred at the given position. This could be carried out during the windowing function of consensus sequence generation to reduce the overhead of having to go through the aligned reads twice.

4.6 Running a Sequencing Experiment

As discussed in Section 3.6, we can now look at the process for running an entire sequencing pipeline and collating the associated metadata. To initiate an experiment, the user provides three inputs:

1. **Reads File** - a flat file location for the reads.
2. **Reference File** - a flat file location for the reference sequence, or an integer representing the REFID in the RefSeq table.
3. **Error Model** - an edit distance **k** definition for how far a read mapping can deviate from the reference.

This calls an 'RunExperiment' stored procedure which collects all the required data about the experiment.

The stored procedure has the following steps:

1. Create an Experiment given a set of reads, a reference file and an error model index file:
   (a) Check Index Table to see if an index file exists for the given reference and given error model
      - **(exists)** Use existing error model index file for this experiment; or
      - **(does not exist)** Create and store a new index file for this reference and this error model. Insert a new entry into the Index Table. Then use that newly created file for this experiment.
   (b) Insert an new entry into Experiment table with appropriate foreign key ids for Error Model Index and Reference Sequence. Include the reads file location. Allocate a new ExpID for this experiment.
2. Load Reads file and Error Model Index file and carry out an alignment of each read to the reference sequence. For each read:
   - Insert an entry to the ShortReadAlignments Table with ExpID, reference sequence starting position and an alignment.
3. Filter the ShortReadAlignments Table by ExpID, then sort the resulting tuples by reference sequence starting position.
4. Carry out a Consensus Sequence Generation Window along the aligned reads. For each position on the reference sequence:
   (a) Tally the bases at that position with sufficient quality.
   (b) Add the most common base to the consensus sequence
   (c) Add any other bases to the SNPTable using ExpID, position, base and quality.
5. Return SNPs for this experiment by filtering the SNPTable by ExpID

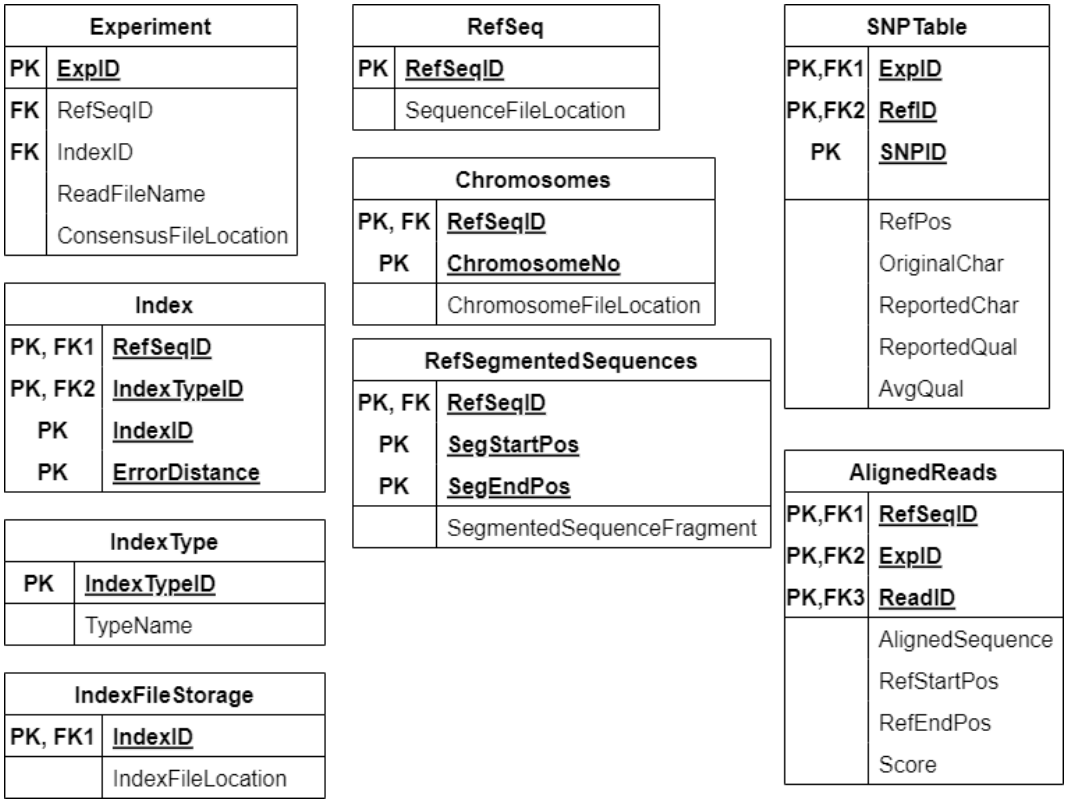

Figure 4.16: Schema Diagram

4.6.1 Table Schemas involved with an Experiment

Here we collate all the schemas for the tables involved with running an experiment. Figure 4.16 shows a schema diagram of the tables described so far in this thesis.

4.7 Implementation Specifics Summary

Herein I will present a summary of the various architecture extensions and functionality extensions that were required to implement our design.

4.7.1 Data Loading

Data loading begins with interfacing with the various data formats, so I built UDTs that could hold the data formats for FASTA and FASTQ sequence files.

Next, I required an external data wrapper that could access those files formats directly and stream them during processing without having to convert and load them into the database as part of the processing pipeline.

Having completed these two implementations, now a genomic sequence such as a reference sequence or a set if reads can be interpreted by our platform.

The sequence itself can be stored as a UDT in a table, or a reference to the base file can be recorded as a varchar field in a table.

4.7.2 Index Generation

Next is index generation. This requires a UDT to represent an index and a stored procedure to generate an index from a reference sequence file.

Similar to the sequence, an index can be stored as a UDT or materialized as a flat file in the file system that is managed by the database using the FILESTREAM keyword.

4.7.2.1 Read Alignment

After we have a set of reads, a reference sequence and an index for said sequence, we can begin the process of read alignment. The identification of alignment is a multi-step process. First, a method of alignment has to be selected. I implemented two approaches: seed and extend (similar to BLAST) and hotspot (similar to SNAP). This means three new stored procedures, one for selection and one for each alignment approach. Results of the alignment are pushed into a table.

4.7.3 Consensus Sequence Generation

Like read alignment, consensus sequence generation does not require any new data types. Consensus can be achieved with a single stored procedure accessing the aligned reads from the previous step.

The resultant consensus sequence is stored as a sequence UDT in a table. All significant variants are stored in another table as native datatypes (no UDT required).

4.7.4 Variant Calling

As most of the computation for this step was carried out during Consensus Sequence Generation, this step can be carried out with a SQL query rather than a stored procedure. However, for the sake of completeness, this query was encapsulated as a stored procedure to allow for consistency in pipeline calls. Results are already in their storage format and only require retrieval by the user.

4.7.5 Running a Sequencing Pipeline

In order to provide the user with a full pipeline along with their individual functionality, a single overarching management stored procedure is also provided. This doesn't require any new UDTs, but it does push results references and settings to a metadata table.

4.7.6 List of Major UDTs and Stored Procedures

As a reference for all the actions discussed above, I have collated the major UDTs in Table 4.13 and stored procedures in Table 4.14.

Name	**Functionality**	**Purpose**
BaseSequence	Data Loading	Baseline type for storing a sequence
RefSequence	Data Loading	Extension of BaseSequence for storing a reference sequence
Read	Data Loading	Extension of BaseSequence for storing a sequence read. Can contain quality data
FASTAEntry	Data Loading	Extension of BaseSequence for storing a FASTA entry
FASTAQEntry	Data Loading	Extension of Read for storing a FASTQ entry
ExternalFileParser	Data Loading	Parser for reading a file using an external data wrapper
FASTAParser	Data Loading	Extension of ExternalFileParser for parsing a FASTA file into multiple FASTAEntry objects
FASTQParser	Data Loading	Extension of ExternalFileParser for parsing a FASTQ file into multiple FASTQEntry objects
SRFParser	Data Loading	Extension of ExternalFileParser for parsing a SRF file
ZTRParser	Data Loading	Parses ZTR Chunks in SRF Files
SimpleIndex	Genomic Index	Baseline Genomic Index type, extension of HashTable
ErrorIndex	Genomic Index	Extension of SimpleIndex, contains an index with variability for a given edit distance.

Table 4.13: Core User-defined Types in the final implementation

Name	Functionality	Purpose
LoadFASTA	Data Loading	Reads a FASTA file entry by entry as a UDT into the database
StreamFASTA	Data Loading	Parses a FASTA file entry by entry without storing it in the database
LoadFASTQ	Data Loading	Reads a FASTQ file entry by entry as a UDT into the database
StreamFASTQ	Data Loading	Parses a FASTQ file entry by entry without storing it in the database
LoadSRF	Data Loading	Reads a SRF file entry by entry as a UDT into the database
StreamSRF	Data Loading	Parses a SRF file entry by entry without storing it in the database
CreateErrorIndex	Genomic Index	Given an Reference Sequence UDT or FASTA file location, will create an ErrorIndex UDT for that sequence
LoadErrorIndex	Genomic Index	Given a materialized ErrorIndex file, will create an ErrorIndex UDT from that file
AlignReads	Read Alignment	Given a SimpleIndex and a set of reads, will perform a read alignment
FindSeeds	Read Alignment	Given a SimpleIndex and a read, will find index matches for seedlength sized fragments of the read.
ExtendSeeds	Read Alignment	Given a RefSequence and an index match, will extend the match for k possible distance.
SWGAlign	Read Alignment	Will perform the Smith-Waterman Gotoh dynamic programming algorithm to find the best alignment between an read fragment and a reference sequence fragment.
SWGAlignSIMD	Read Alignment	Will use SIMD accelerations to perform the Smith-Waterman Gotoh dynamic programming algorithm to find the best alignment between an read fragment and a reference sequence fragment.
GenerateConsensus	Consensus	Using a windowing approach, retrieve the aligned reads in order of appearance in the the reference sequence and retrieve the most common aligned base to make a consensus sequence for the set of reads. Less common aligned bases are put into the SNP table
CallSNPs	Variant Calling	Collate the results from the SNP Table to generate a list of variants for this read alignment and consensus operation
CreateExperiment	Experiment	A management stored procedure for running a pipeline experiment. Calls the other stored procedures and collates metadata regarding inputs and outputs for each operation in the pipeline.

Table 4.14: Core Stored Procedures in the final implementation

Chapter 5

Evaluation

In the following, we evaluate how well the proposed approaches enable a database system as a platform for genomic data processing.

We will do so in multiple steps:

1. **Data Loading.** Looking at the storage layer, Genomic data comes in different file formats, most prominently FASTA (described in Section 2.5.1.1), and before it can be access and queried in databases, it has to be loaded into an internal storage representation as discussed in Section 4.1.1. In Section 5.2, we compare the storage efficiency and the data import overhead for various storage formats. We compare this further with out-of-database storage, as well as the hybrid approach of using SQLServer's FileStreams to access native storage formats and allowing the database system to treat them as table-like objects that can be queried with SQL.

2. **Index Generation.** An important optimisation for query evaluation and efficient data processing are indexes which attempt to focus I/O on relevant data entries which are needed for a query. In Section 5.3 we evaluate the storage overhead and the index generation times for indexes over gene sequence data.

3. **Sequence Alignment.** The core of the gene sequence pipelines is the alignment of short reads to a reference genome. In Section 5.4 we evaluate the runtime of short-read alignment within a database as compared to external short read alignment tools.

4. **Consensus Generation and SNP Calling.** Once reads are aligned to a reference sequence, then a consensus sequence and variants can be collated to see how the input reads differed from the reference sequence. Section 5.6 evaluates the performance of in-database consensus sequence generation and SNP calling.

5. **Whole Pipeline Analysis.** In the final Section 5.8.2, we compare the runtime of an overall gene sequence alignment pipeline inside a DBMS with a typical pipeline running out-of-DBMS in the command line.

5.1 Evaluation Platform

Our evaluation platform is as described in Section 2.9.1.1 and again in Table 5.1. We primarily ran on Windows where we could, except when we needed to benchmark platforms other than our own. This required some Linux benchmarks on the same machine. WSL2 on Windows was highly effective in replicating Linux behaviour on Windows 11, but would not work with Nvidia NSight benchmarking platform.

The specifications for this system is summaried in Table 5.1:

Component	Description
OS	Microsoft Windows 10 Professional
CPU	Intel Core i7 11700k
RAM	128GB DDR4 3200 Mhz CL16
GPU1	8GB Nvidia RTX 3070
GPU2	8GB Nvidia RTX 3070
HDD	5TB External USB Drive
SSD	120Gb SATA OS Drive
NVME1	PCIe 3.0 480Gb Drive
NVME2	PCIe 4.0 960Gb Drive
RDBMS	Microsoft SQL Server 2018 Developer Edition (v15.0.2000.5)
.NET	.NET 6.0 and .NET Framework 4.8

Table 5.1: Primary evaluation server specifications

5.2 Data Loading

In the field of genomic data, the FASTA format is the default. Some tools may use alternative, extended or platform-specific versions such as FASTQ or CSFASTA, however all these platforms and tools have the ability to transform any sequence data to the FASTA format.

Generally we find FASTA files representing pure sequences and FASTQ or CSFASTA files representing reads. There are also some downstream files such as the SAM/BAM format for multiple sequence alignments or the VCF format for SNP identification.

As such we chose to test the data loading and internal representation of FASTA files as a UDT in our platform.

The FASTA files we chose are from the GRCh37 reference genome (version p13).

5.2.1 FASTA

Initially we look into the data-loading times for different storage schemes. The genome as a whole is split up into logical separations of chromosomes as representatives of the native biological state of these genomic sequences.

The main premise here is to evaluate the ingestion of the entire reference genome and compare the different storage schemes. Each of the loading scenarios described here is an integer average of 10 loading runs. We use our Filestream interface on .NET CLR to load the data into a table instead of directly querying it. We have data schemes provided in Section 4.1.2.1.

For data-loading, as shown in Figure 5.1, we compare multiple different schemes:

- Byte loading (raw results in Appendix Table B.1).
- two-bit segmented encoding scheme (raw results in Appendix Table B.2).
- three-bit segmented encoding scheme (raw results in Appendix Table B.3).

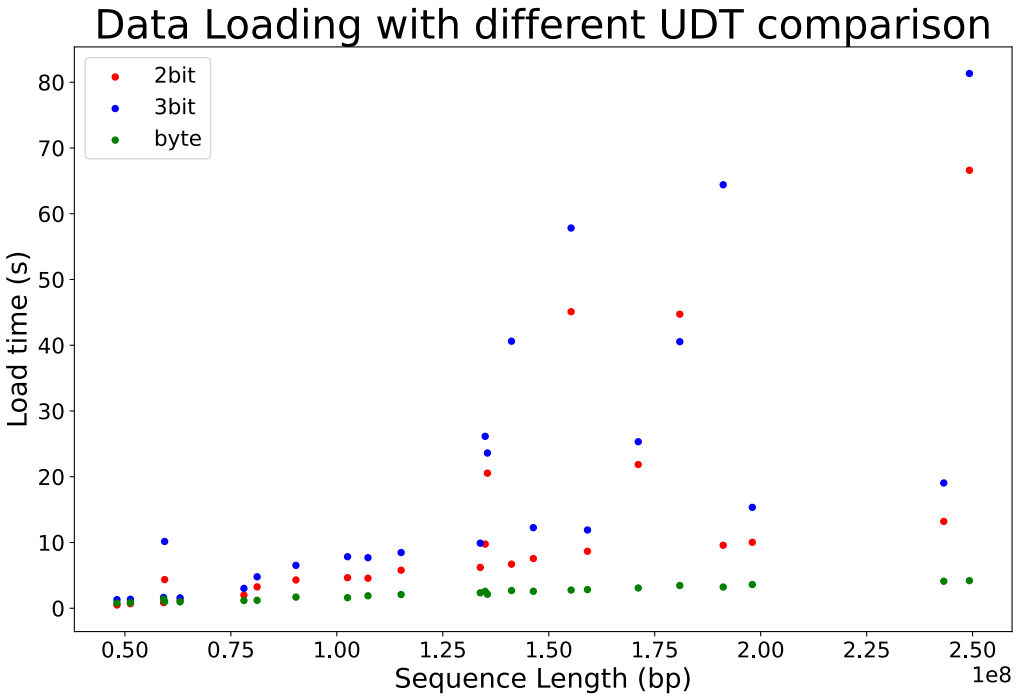

Figure 5.1: Data Loading Comparison between Storage UDTs

As seen by a comparison in Figure 5.1, there is a large discrepancy in loading the larger chromosomal sequences. This is as a result of the overhead of the Filestream access pattern and the 2-bit and 3-bit encoding vs the native character loading for which a DBMS is optimised.

It is clear from Figure 5.1, that the more processing that occurs (2-bit encoding requires only three operations for compaction, while 3-bit encoding requires six operations for compaction), the longer the data takes to load.

There also seems to be two distinct linear lines of fit. We suspect this has to do with the memory capacity boundary with SQL Server memory thresholds affecting the amount of available memory (the evaluation server had 128 GB main memory).

Given that database systems are used to natively process character data, it is no surprise that data loading the sequences directly without any extra encoding (i.e. using byte encoding) will result in the fastest data ingestion. The relationship between sequence length and load-time also looks to be linear - which bodes well for scalability with future sequences.

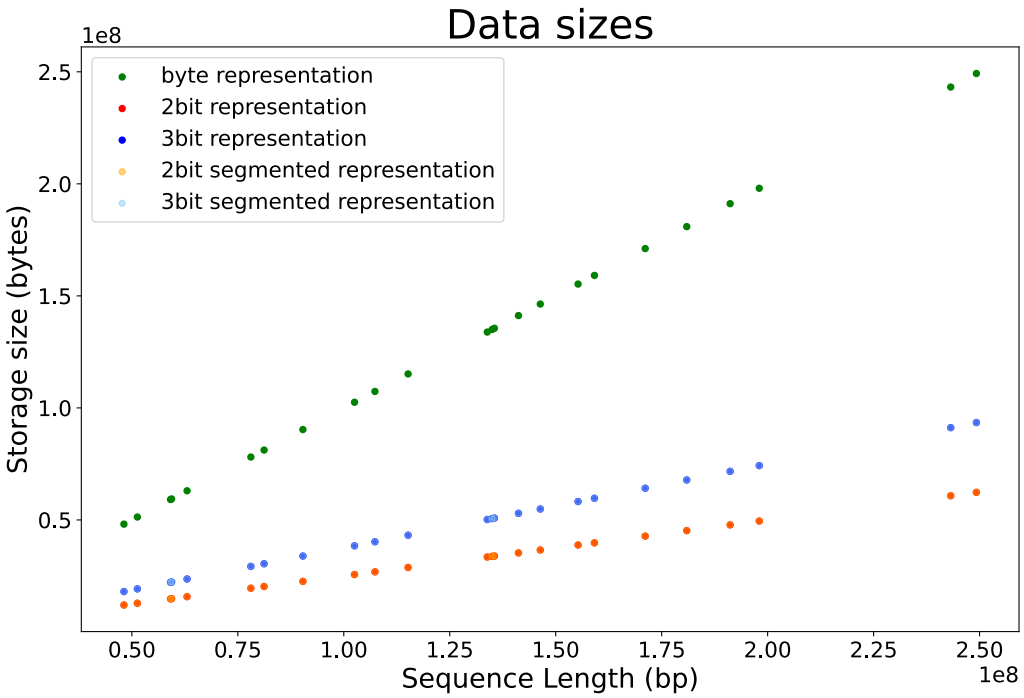

Figure 5.2: Data Loading Comparison between Storage UDTs

This appears to be a space-time trade-off as we see the relative sizes of the loaded data in Figure 5.2.

Another aspect of data storage is how it is affected by the page layout of the DBMS Storage system. In Figure 5.2 we see the resultant file sizes of the chromosomes when stored with or without enforced segmentation by page size (8092 bytes - overhead). The efficacy of enforced segmentation can only be really seen during a comparison of querying the table data.

5.2.2 SRF

Our implementation also allows for data access of specific file formats. We implemented a demonstration data access wrapper for the SRF format. In Table 5.2 we compare our implementation (SRFExplorer and SRFExplorerTVF) against the most commonly used command line file-based program set (srf2fastq) for SRF operations, the Staden package [98; 8] which converts SRF files into FASTQ format.

Reads	srf2fastq	SRFExplorer	SRFExplorerTVF
10	75	3.1	118
100	59	18.7	131
1000	86	187.2	357
10000	437	1847.1	2446
100000	4018	18305.1	23160
1000000	41342	177084.5	220768.9

Table 5.2: Comparison of SRF loading times between srf2fastq, SRFExplorer and SRFExplorerTVF. Measurements are in milliseconds.

Profiling the run indicates the following:

- Most space and function call intensive section is the DeflateStream constructor and DeflateStream.Read() function (90%).
- ReadZTRChunk() takes
  - 0.279 ms on average for reading BASE chunks and,
  - 0.628 ms on average for reading CNF1 chunks.
- the tryDeflate() decompressing function, which would typically be the most time consuming, only takes up 0.054ms for each chunk call.

These results indicate that the overhead for a datawrapper is considerable the larger our read set becomes.

5.2.3 Data loading Comparison

Our **BaseSequence** type showed the best performance when it was loaded in a segmented fashion rather than all at once. This is also ideal for access purposes as unless we need the entire sequence, we only need to load the relevant pages. Figure 5.3 shows that the more complicated our encoding method, the more time it takes to load the sequence. Consequently load times are **3bit** $>$ **2bit** $>$ **byte**. However, we see the obvious outcome in storage size, in which the sizes are as expected for each encoding can be seen in Fig 5.4: **byte** $>$ **3bit** $>$ **2bit**.

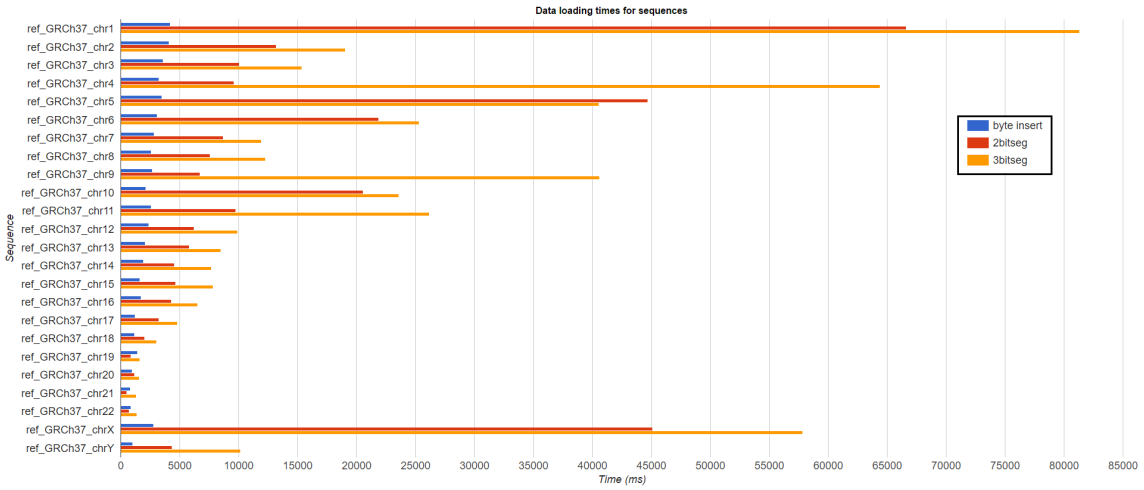

Figure 5.3: Base sequence loading times in seconds with different encodings

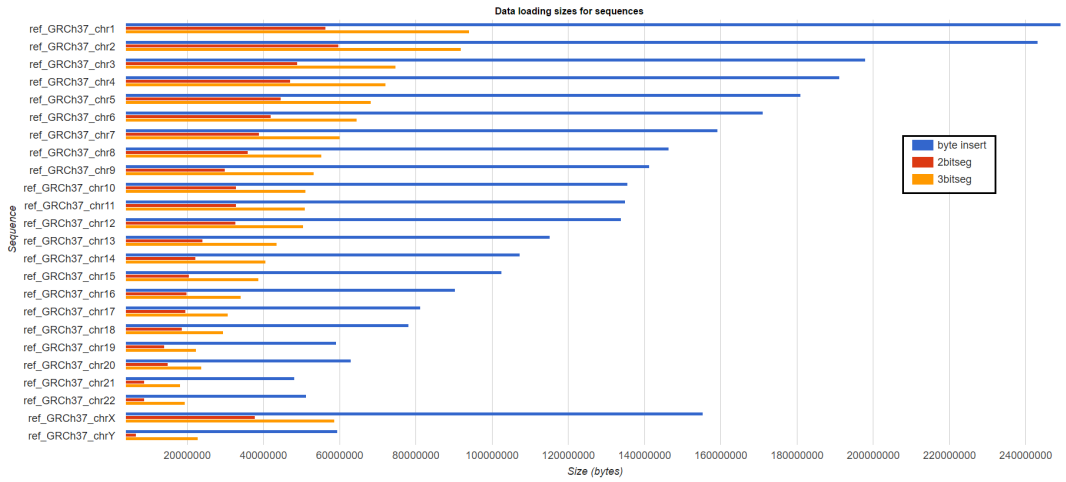

Figure 5.4: Base sequence loading sizes in bytes with different encodings

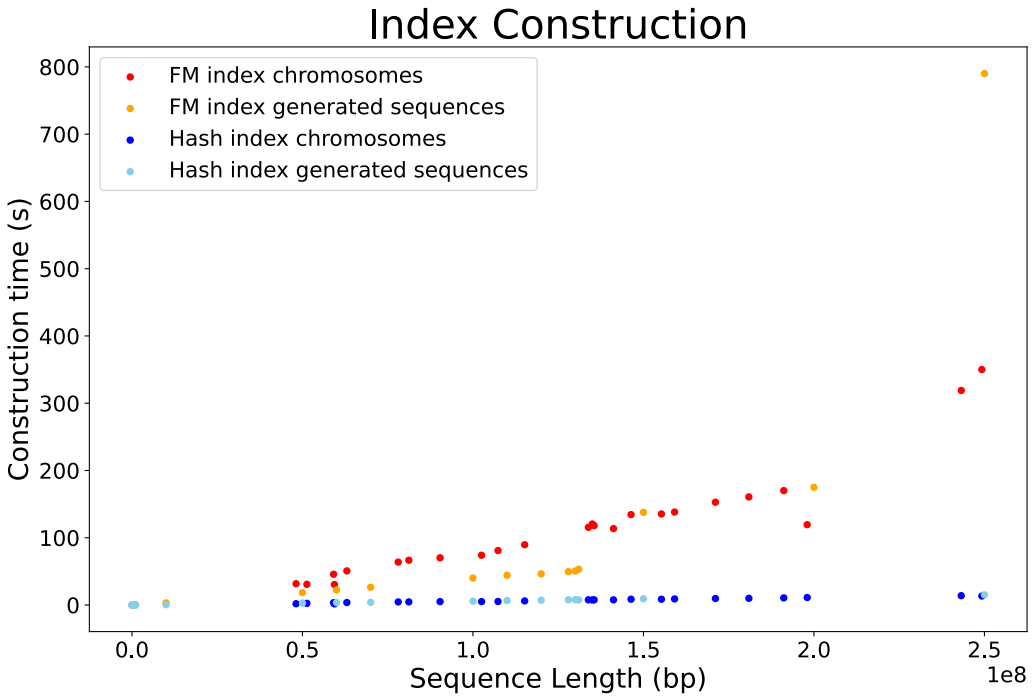

Figure 5.5: Index construction

5.3 Index Generation

5.3.1 Indices

Next we look at the construction times and sizes for the two main index structures: F-M Index (described in Section 4.2.1) and Hash Index (described in Section 4.2.3). These indices are created from the reference genome and stored in the DBMS. We look at creation of the each index using both a randomized sequence of varying readlengths as well as the chromosomes found in the human reference genome GRCh37.

5.3.1.1 F-M Index

The FM index construction takes quite some extra time compared to the hash index. As the FM index is built using new functions and the hash index is built using existing functions, we can consider this an algorithmic optimisation point. Again, we have a space-time trade-off here. The FM index takes about 1.2× the

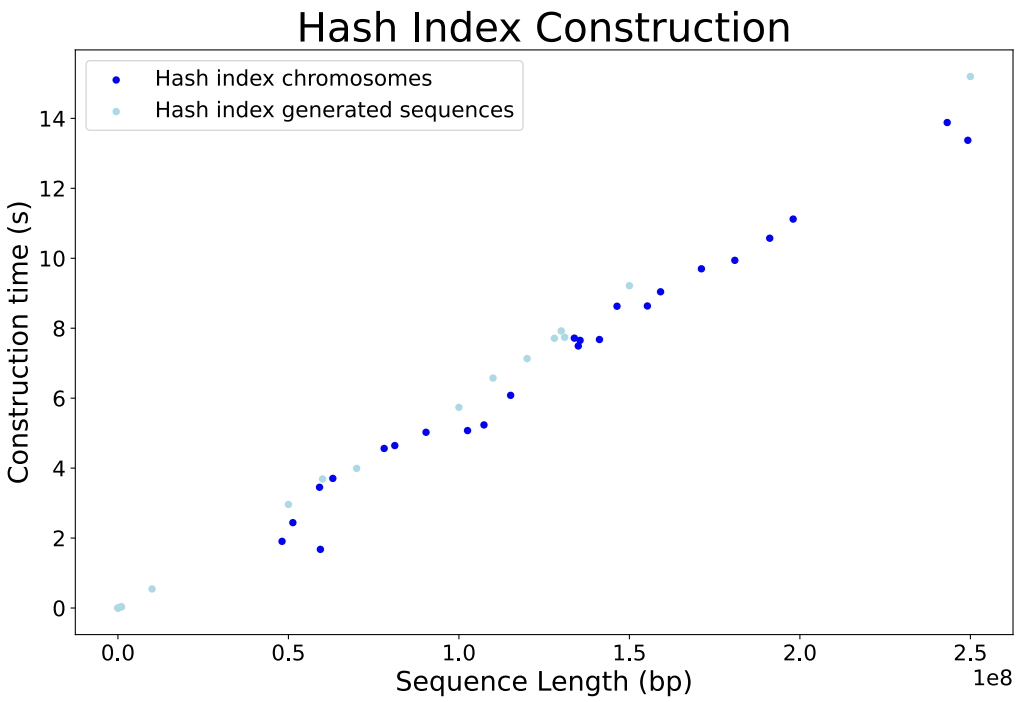

Figure 5.6: Hash Index construction

sequence length in space, while the hash index takes 14× the sequence length in space (on average).

The index construction time relative to sequence length appears linear. Again, this is beneficial for scalability concerns.

5.3.1.2 Hash Index

Construction times for the hash index under normal data generated with no biases or unknowns (N) is shown in Figure 5.6.

5.3.2 Index Types

5.3.2.1 Exact Match

Construction times for the hash index using human reference genome 37 is shown in Table B.9 in the Appendix.

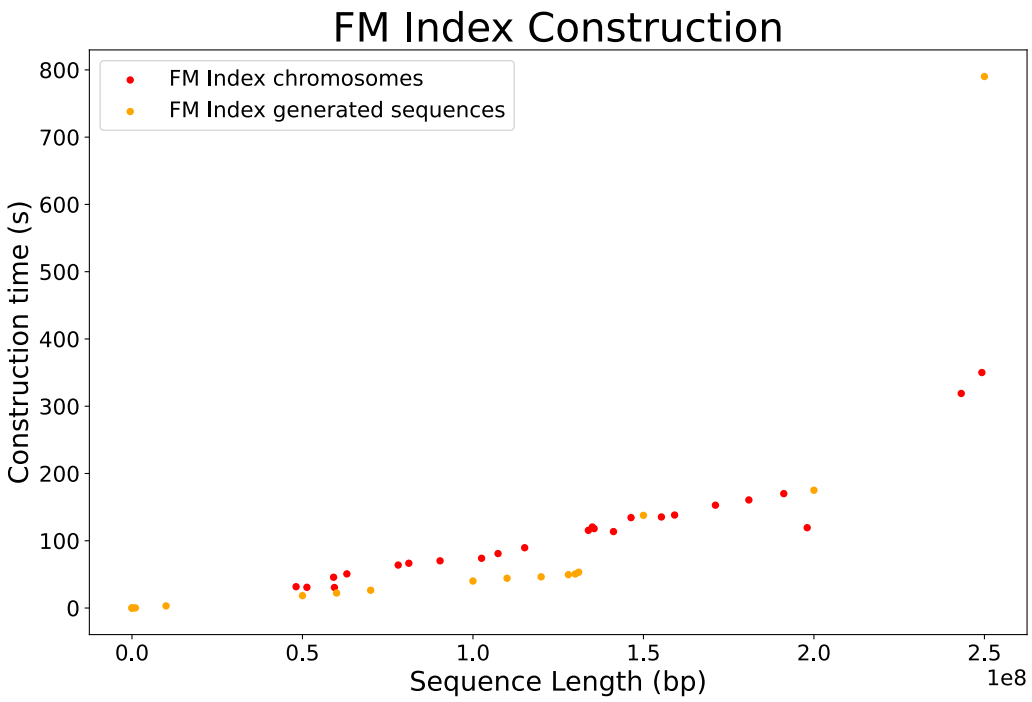

Figure 5.7: FM Index construction

5.3.2.2 Error Model

Construction times for a Hash Index with an error model using human reference genome 37 is roughly similar to Construction time $*seedLength$ for the EXACT-MATCH index.

5.3.3 Data Sizes

Given that these indices are compute once-use many, they are also stored in the database.

When we began to construct k-error model hash index, we found that our memory footprint exceeded our system memory (and so started using the system page file - which reached 1.8× our system memory). This results in thrashing when data is continuously being moved between the page file on disk and system memory. Fortunately, we predicted this outcome and have a methodology in place to deal with this scenario. Here we will use the index fragmentation methodology described in Section 4.3.2.2. However, we first need to establish

a value for fragmentation. In Table 5.3 we set a series of fixed values for the memory available to MS SQL Server. This is achieved by running SQL Server inside a virtual machine with fixed memory using Windows Hyper-V.

Fragment Length (bp)	**Split**	**Max Fragment Length (bp)**	**RAM (GB)** - System	**RAM (GB)** - SQLServer	**Thrashing - Index Creation**	**Thrashing - Sequence** Alignment
250000000	1	249250621	16	11	TRUE	TRUE
250000000	1	249250621	128	16	TRUE	TRUE
250000000	1	249250621	128	32	FALSE	TRUE
250000000	1	249250621	128	64	FALSE	FALSE
200000000	2	200000000	16	11	TRUE	TRUE
200000000	2	200000000	128	16	TRUE	TRUE
200000000	2	200000000	128	32	FALSE	TRUE
200000000	2	200000000	128	64	FALSE	FALSE
150000000	2	150000000	16	11	TRUE	TRUE
150000000	2	150000000	128	16	FALSE	TRUE
150000000	2	150000000	128	32	FALSE	FALSE
150000000	2	150000000	128	64	FALSE	FALSE
125000000	2	125000000	16	11	FALSE	TRUE
125000000	2	125000000	128	16	FALSE	TRUE
125000000	2	125000000	128	32	FALSE	FALSE
125000000	2	125000000	128	64	FALSE	FALSE
100000000	3	100000000	16	11	FALSE	TRUE
100000000	3	100000000	128	16	FALSE	TRUE
100000000	3	100000000	128	32	FALSE	FALSE
100000000	3	100000000	128	64	FALSE	FALSE
80000000	4	80000000	16	11	FALSE	TRUE
80000000	4	80000000	128	16	FALSE	FALSE
80000000	4	80000000	128	32	FALSE	FALSE
80000000	4	80000000	128	64	FALSE	FALSE
60000000	5	60000000	16	11	FALSE	FALSE
60000000	5	60000000	128	16	FALSE	FALSE
60000000	5	60000000	128	32	FALSE	FALSE
60000000	5	60000000	128	64	FALSE	FALSE
40000000	7	40000000	16	11	FALSE	FALSE
40000000	7	40000000	128	16	FALSE	FALSE
40000000	7	40000000	128	32	FALSE	FALSE
40000000	7	40000000	128	64	FALSE	FALSE

Table 5.3: Testing whether memory thrashing occurs at segmentation levels

We found that the limit our system could handle for our index structure was

Sequence Length (x million bp)	250	200	150	125	100	80	60
mem usage index creation	21.0	16.8	12.6	10.5	8.4	6.7	5.0
mem usage sequence alignment	43.7	34.9	26.2	21.8	17.5	14.0	10.5

Table 5.4: Memory usage during index creation and sequence alignment with various fragment sizes

Name	**Size** (bp)	**Entries**	**Storage size** (bytes)	**Construction time** (s)
ref_GRCh37_chr1	249250621	108541679	4148092718	28514.77

Table 5.5: Storage size and run time for Hash Index with robust error modelling for repeat masked chromosome 1 of hg19

around 61000000 entries. This is why we started using repeat masked sequences and introduced index fragmenting as a solution as described in Chapter 4.3.2.2. We ended up running a single repeat masked chromosome to see what the construction time and size might be. This is shown in Table 5.5. After implementing fragmentation, we ran both random data sets and chromosomes for index construction seen in Appendix Table B.14.

5.3.4 Repeat Masked Reference Sequences

Repeat masking the reference sequences that are used for index generation results in a large portion of the sequence being masked. Since this results in long strings of N, these matches would be excluded from the index. Appendix Table B.15 shows a comparison between the number of entries in our Hash Index method with and without repeat masking.

5.4 Short Read Alignment

Our short read alignment performance evaluation is based on the size of the index, type of index, and number of mismatches in the query. The relevant evaluations can be found in the full pipeline run Appendix Table B.24.

5.4.1 F-M Index Exact Match

While testing our FM index, we found that our runtimes were far too high, with our processing rate being about 5 to 27 reads per second, which means any run

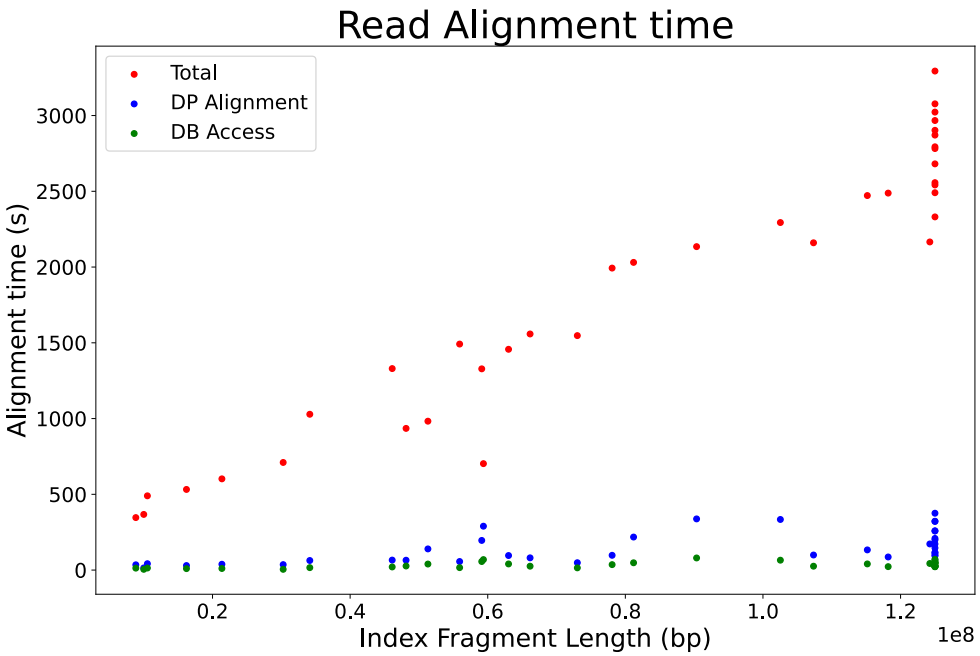

Figure 5.8: Read Alignment using Hash Index that is fragmented at 125 Mbp

for even the smallest chromosome would take nearly half a day, and a complete genome would take approximately 3 months to finish.

5.4.2 Hash Index Error Model

For the purposes of read alignment, we decided to use the Hash Index as it was substantially faster. The trade-off for this was the memory footprint required to run an alignment. Initially when I tried to run read alignment, I encountered a RAM barrier which caused resource thrashing as data was moved between the memory and the hard drive page file. To avoid this occurrence, I created a fragmentation point at 125 million base pairs. That meant each index was recreated with multiple smaller profiles per reference sequence. However, this meant that the input reads would need to be run against each index (increasing run-time). Hence, we can see what appears to be two separate curves in Figure 5.8.

When we extract only the reads for the fragment sizes of just 125 million (Figure 5.9), we see that the read alignment is the same, independent of the sizes of the reads being aligned against it and are more dependent on the size of the index.

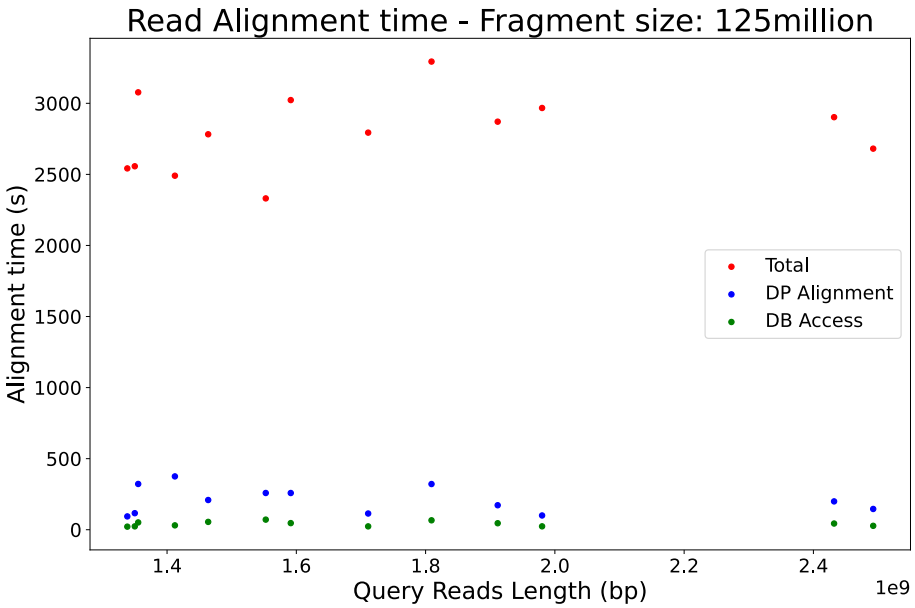

Figure 5.9: Read Alignment using Hash Index 125mill size fragments only

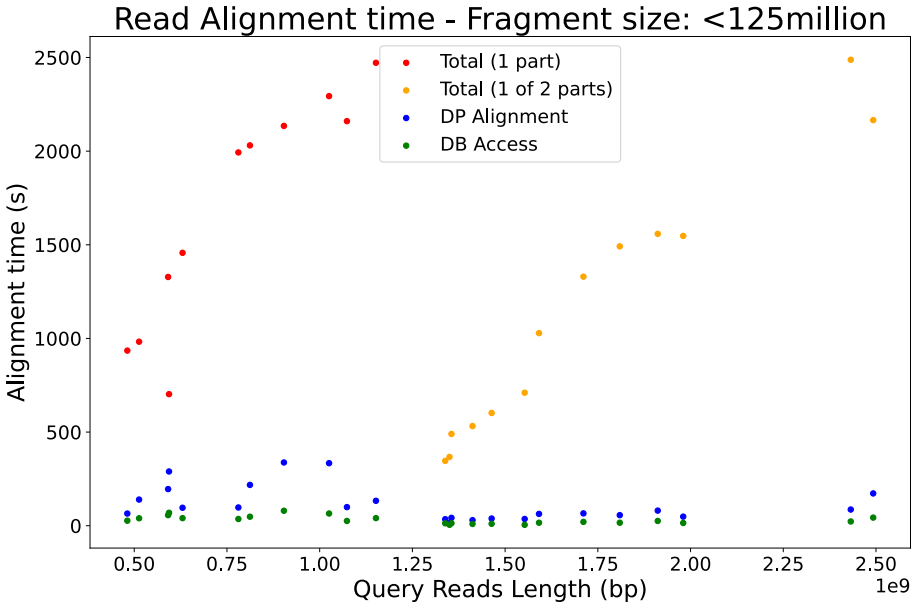

Figure 5.10: Read Alignment using Hash Index not 125mill size fragments only

However, when we look at the comparison without these large index fragments (Figure 5.10), we see that this is not the case. The smaller sequence lengths (those without two index fragments - shown in red) are taking longer,

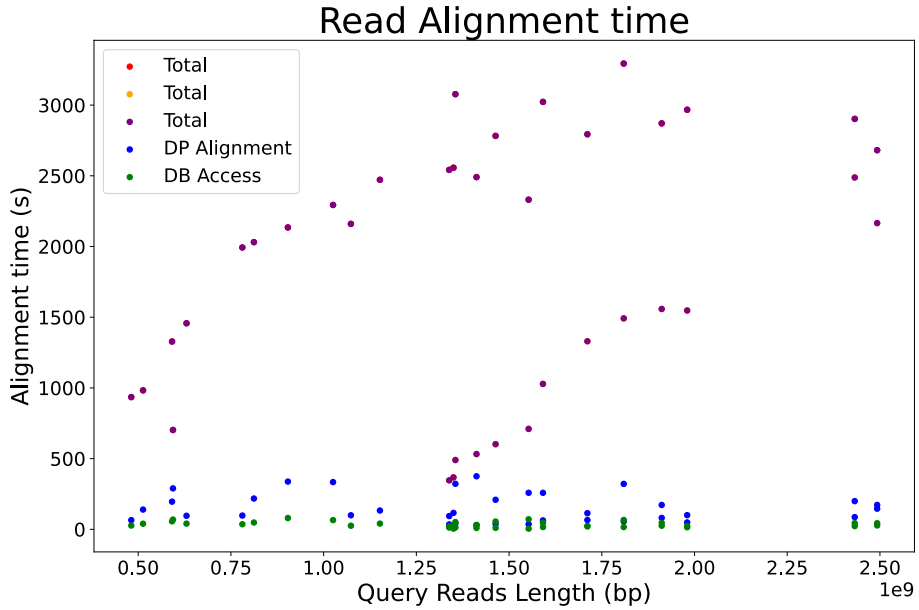

Figure 5.11: Read Alignment using Hash Index not 125mill size fragments only

which can likely mean that we have some sort of read loading overhead. Indeed, when we plot just the initial read run (Figure 5.11), we can see that an asymptotic curve appears.

Our read alignment average runs can be seen in Appendix Table B.24.

5.5 Dynamic Programming

While our Short Read Alignment procedure first uses an index or indices to calculate viable candidate regions for our alignment, we use dynamic programming to align the full read against the relevant section of the reference genome. As this is the most repeated procedure of the alignment after index lookups, this motivated us to speed up our code by using SIMD extensions. Figure 5.12 and Figure 5.13 show an average runtime for various query sizes and edit distances for the generic Smith-Waterman algorithm and the SIMD improved version. In order to get a more fine-grained reading for performance at this level, we used elapsed CPU ticks instead of milliseconds as our counter.

When we chart the values in log scale as seen in Figure 5.12 and Figure 5.13,

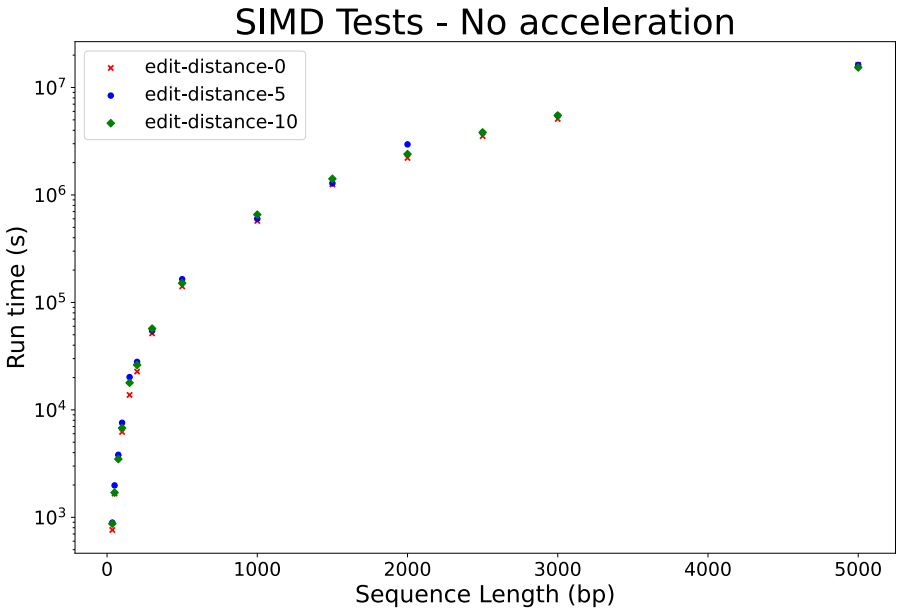

Figure 5.12: Smith-Waterman Algorithm runtimes without any SIMD acceleration.

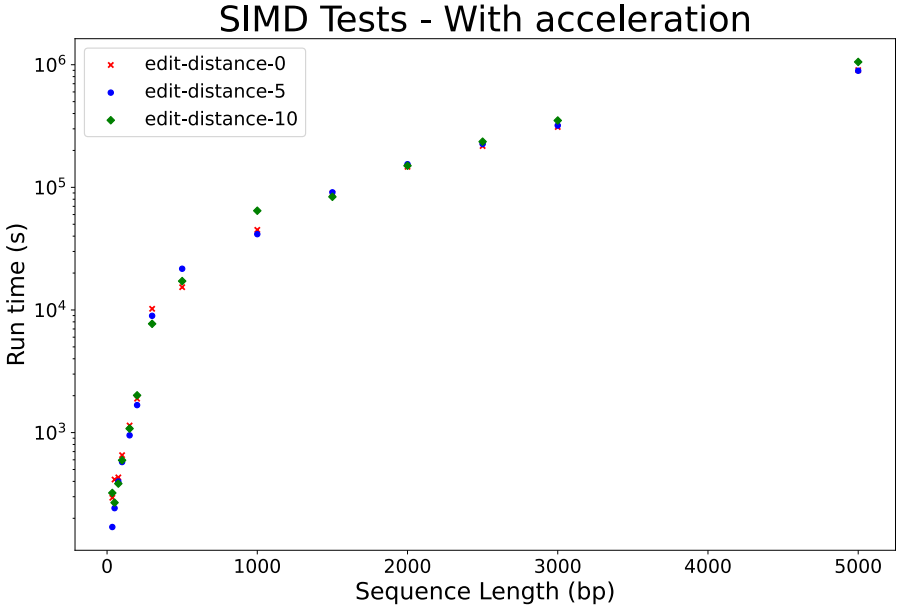

Figure 5.13: Smith-Waterman Algorithm runtimes with SIMD acceleration.

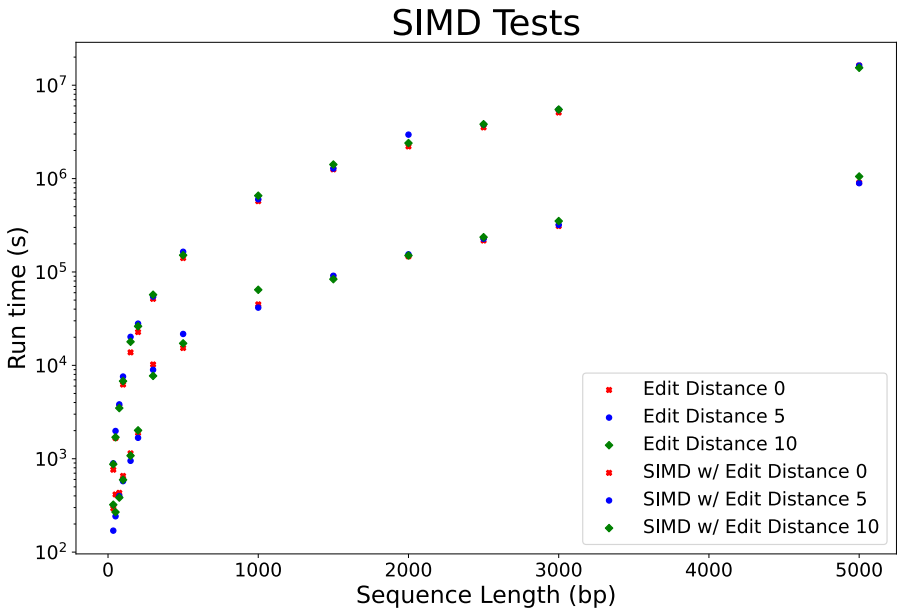

Figure 5.14: Smith-Waterman Algorithm runtimes with comparison between SIMD acceleration and No SIMD acceleration

we can see that the SIMD improvements are resulting in an order of magnitude improvement at the higher query lengths. This can also be nicely seen in the comparison between SIMD acceleration and no SIMD acceleration in Figure 5.14. Since read sizes are continually increasing, this does bode well for the performance of our solution when measured against upcoming read sizes.

5.6 Consensus Calling

This evaluation result can be seen in the whole pipeline analysis in Table B.25 in the Appendix.

Figure 5.15 shows the time taken to generate the consensus sequence within a database engine. Compared to read alignment, the consensus sequence generation algorithm takes substantially less time. This is partially due to ease of access to our reads once we have stored them in a database engine. This makes operations such as sorting faster compared to data structures that command lines tools might generate (as seen in Figure 5.16 and in Figure 5.17).

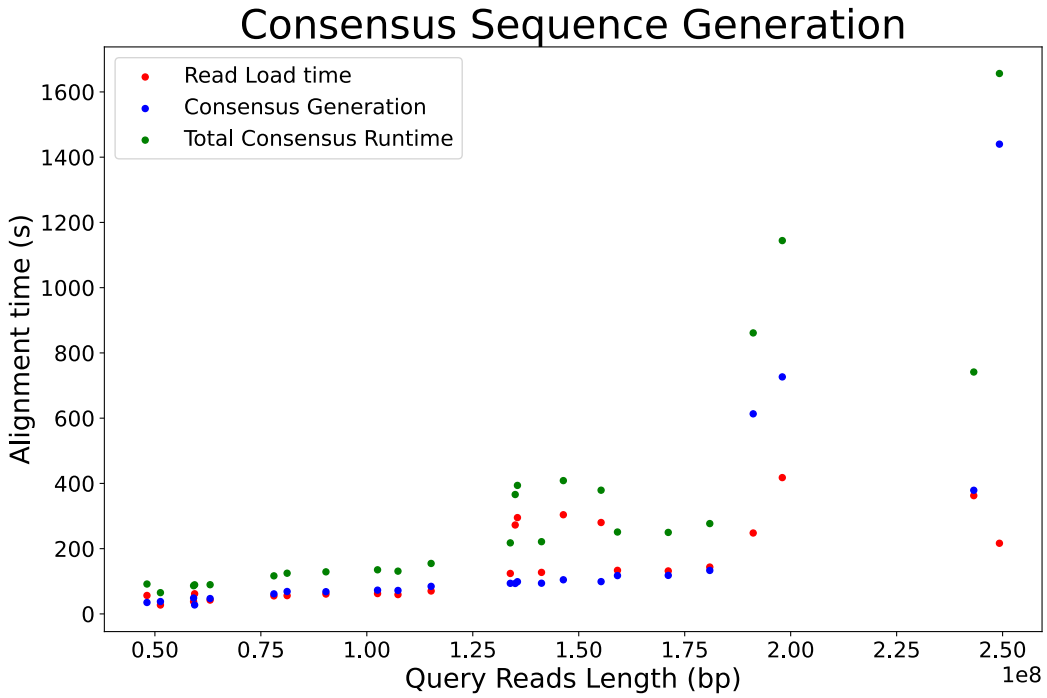

Figure 5.15: Consensus Generation - including SNP Table insertion

Our pipeline in described in Figure 5.16. The pipeline itself is similar to the samtools pipeline described in Figure 5.17.

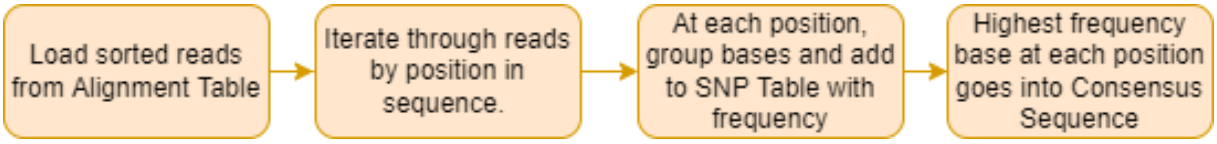

Figure 5.16: Our consensus pipeline

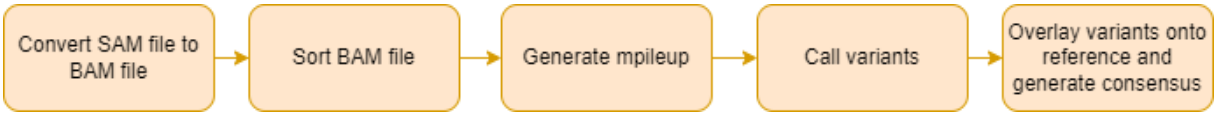

Figure 5.17: Samtools consensus pipeline

5.7 SNP Calling

As with command line tools, SNP Calling and Consensus Sequence Generation is intertwined. As a result the individual SNP Calling function call results seen in Figure 5.18 are simply loading the results that were inserted into the SNP table during Consensus Sequence Generation.

While the relationship between sequence length to SNP calls is roughly linear, we can see some deviation based on the variability of mutations (or variants) in the reads of each chromosome.

It should be noted that the command line tools such as SOAP's snpcaller and GATK's haplotype caller also do a preliminary detection of indels as well as SNPs. However, the accuracy and precision rate for indel prediction is significantly lower compared to SNP calling (see Section 2.6.3) so we made the determination to not include it in our reference implementation.

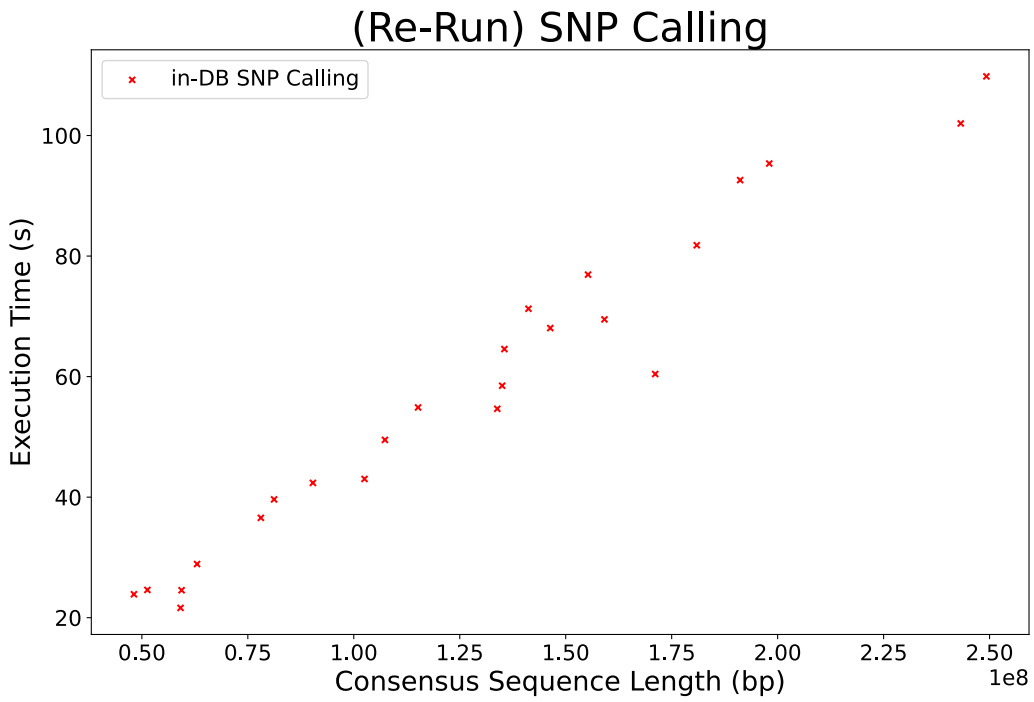

Figure 5.18: SNP Calling runtime

5.8 Whole pipeline analysis

Once all the component pieces are assembled, a full pipeline can be constructed by plugging various outputs of downstream operations into upstream operations.

The full pipeline is as follows:

- index construction
- index serialization
- index deserialization
- query processing
- consensus calling
- SNP identification

The Appendix Tables B.20 through to B.26 show the progression of the pipeline that was based on the repeat masked chromosomes of hg19 using a robust error model hash index:

- Appendix Table B.20 shows the runtimes for the full pipeline for each genome.
- Appendix Table B.21 shows the index construction times. The fragmentation point was set to 125 Mbp.
- Appendix Table B.22 shows the serialization time for each reference sequence index. The fragmentation point is set to 125 Mbp.
- Appendix Table B.23 shows the time taken to load each serialized index back into memory.
- Appendix Table B.24 shows the time taken to process each reads file. Each reads file consisted of 10 0bp reads, sampled from each unmasked version of each chromosome with a coverage of 10$\times$ and the following error profile:
  - Total possible mutations per read: 5

  - Overall mutation rate of : 10%
  - Probability of the mutation being a substitution: 80%
  - Probability of the mutation being a insertion: 10%
  - Probability of the mutation being a deletion: 10%

  In addition, a mandatory substitution mutation at every 1009th location was introduced to test SNP calling.

- Appendix Table B.25 shows the time take to load and process the aligned reads and construct a consensus sequence.
- Appendix Table B.26 shows the number and the time taken to insert the variants into the SNP table.

5.8.1 Compare to state of the art non-database tools

Since a majority of these tools are designed to run on a Linux operating system, we used the Windows Subsystem for Linux which gives is a bash shell using an Ubuntu distribution kernel v.3.4.0 x86_64. The various tools used are show in Table 5.6:

Program	**Version**	**Usage**
bwa	0.7.12	Index, Read Alignment, BWT-based Index
bowtie2	2.2.5	Index, Read Alignment, BWT-based Index
snap	1.0beta23	Index, Read Alignment, Hash Index
wham	0.1.5	Index, Read Alignment, Hash Index
soap2	2.20	Index, Read Alignment, Hash Index
samtools	1.4	Consensus calling, Variant calling
soapsnp	1.0.2	Variant calling

Table 5.6: Comparison tools

5.8.1.1 Index Construction

Appendix Table B.27 shows index construction times for bwt-based indexers. Appendix Table B.28 shows index sizes for bwt-based indexers. Appendix Table B.29 shows index construction times for hash-based indexers. Appendix Table B.30 shows index sizes for hash-based indexers.

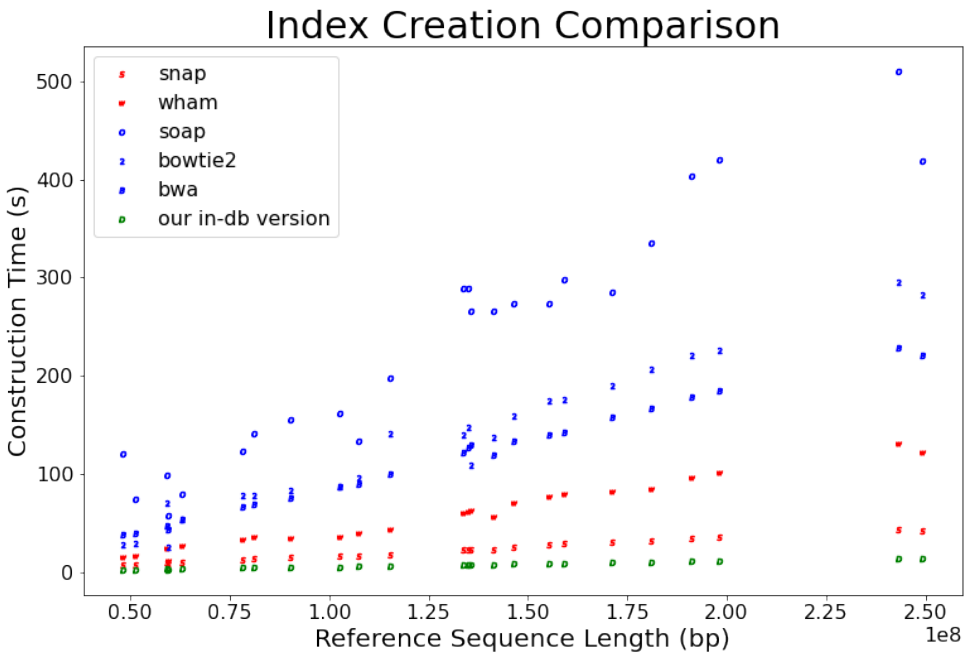

Figure 5.19: Comparison of Index creation times in seconds

Our bwt index construction time is similar but our resultant index file sizes are quite large. This is primarily because we are storing the full FM-Index as opposed to just the bwt transformed and complementary bwt transformed strings that **bwa** and **bowtie** store. As can be seen in Figure 5.19, our bwt index construction time is roughly 20% more than **bowtie2** and 40-50% more than **bwa**. As we store our FM-Index, our filesize ends up roughly 20x the size of **bwa** and **bowtie2** as seen in Figure 5.20. If we were to store just the same data as the comparative programs, we could actually beat them in storage space by around 60% as we store sequences as a **3bit** encoded segmented **BaseSequence**.

Our hash index construction times are quite a bit longer than the comparative programs. Part of the reason is that we use the inbuilt generic .NET **Dictionary**<**Type1**,**Type2**> type which stores an array of key pointers, an array of value pointers, an array of key-values and an array of key to value maps. This means quite a few large memory structures which resulted in our having to fragment the index construction into blocks of size 125 Mbp, but really at 60 Mbp entries. As seen in Figure 5.19, our hash index construction time is nearly

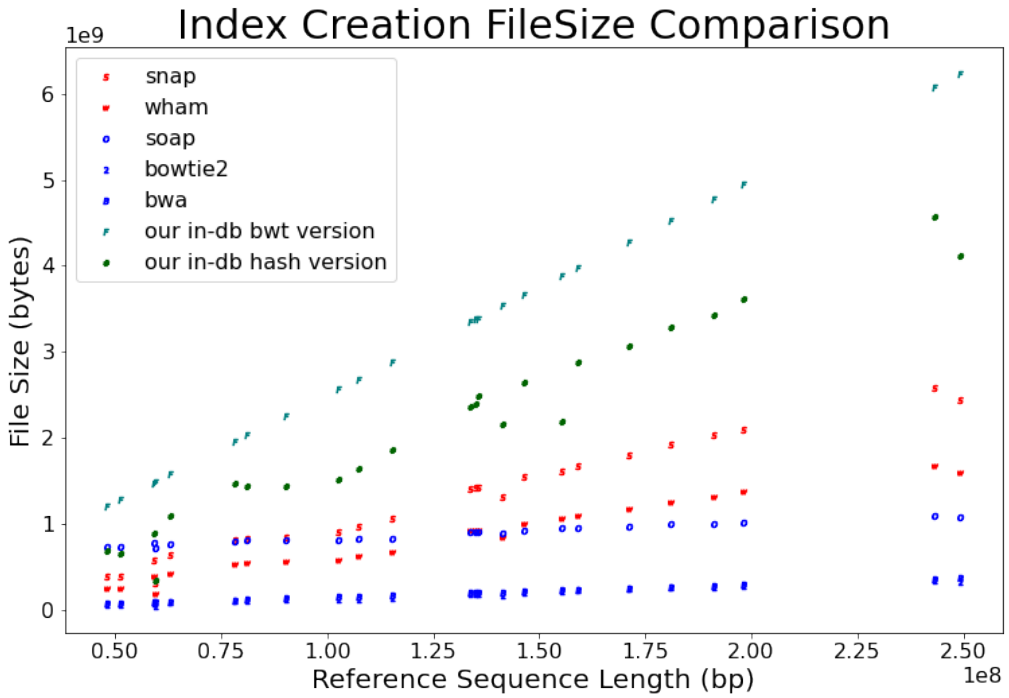

Figure 5.20: Comparison of Index File size in bytes

25× that of **snap**, 6× that of **wham** and 3× that of **soap2**. Conversely, our index size is far smaller than the comparative programs. As seen in Figure 5.20, the faster construction time programs now show a correspondingly larger index size. This usually indicates extra structures inside the implementation that assist with speeding up the index construction and possibly runtime (which we see is true in Figure 5.21). Since we have a naive storage format and a fragmented index, here our index loses out again in size as its 1.6× the size of the **snap** index, 2.5× the size of the **wham** index and nearly 4× the size of the **soap2** index. Given each entry in our storage format for an index is a seed of length 25 followed by a list of integers, we can improve our storage format by storing the seeds in a three-bit encoded format. This should see a saving of at least 16 bytes per entry. Given our largest index has 121 million entries, we should save near 2 GB or a 50% reduction on filesize.

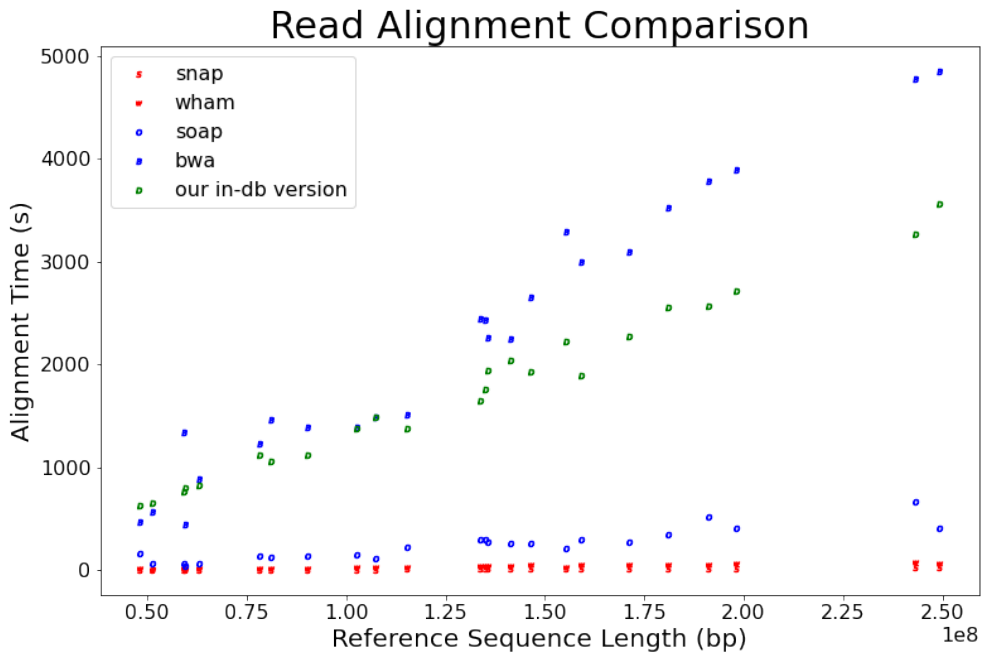

Figure 5.21: Read alignment times in seconds

5.8.1.2 Read Alignment

Here we look at read processing for short read alignment to genomic query strings. We have reads generated from each chromosome with inserted mismatches that we used to get the results in Appendix Table B.24. Of note is that the current industry standard bwt aligners have no issue loading reads - which indicates that our FM Index alignment approach should have solvable issues. Also the smaller number of reads for chromosome 1 is interesting - mainly due to the higher number of repeats.

Appendix Table B.33 shows the size of read alignment outputs for bwt-based indexers. Appendix Table B.34 shows the size of read alignment outputs for hash-based indexers. Appendix Table B.35 shows the total number of reads aligned using hash-based indexers compared to the total reads available. While the bwt aligners can align every read, the hash-based aligners use a semi-global approach and are not guaranteed to align every read. Given sufficient coverage of the genome and high quality reads, this problem is usually overcome on the sequencing pipeline prior to read alignment. However, this approach was mostly solved in **snap** having a better alignment algorithm than **wham** or **soap**.

As seen in Figure 5.21, our read alignment time with a hash index is about the same as **bwa**, but it is twice as slow as **bowtie2** and an order of magnitude slower than **soap2**, and two orders of magnitude slower than **snap** or **wham**. This can mainly be explained due to the overhead of running in the CLR, but it is more likely due to the string manipulations involved with preparing a read for dynamic programming.

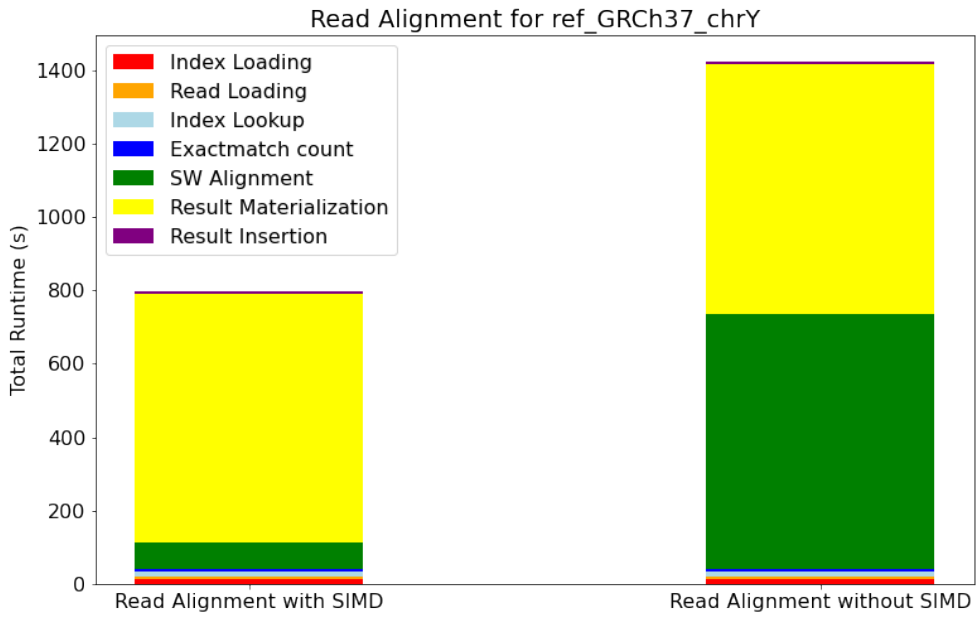

Figure 5.22: Read alignment breakdown for GRCh37_chrY

Figure 5.22 shows a breakdown of the steps in read alignment for a representative sample (in this case, Chromosome Y from GRCh37 with 100bp reads). The most costly of these is the Smith-Waterman alignment and the result materialization. We show that hardware acceleration can improve one of the Smith-Waterman alignment costs substantially but result materialization remains a problem. We leave improving those operations as future work.

We haven't included the results for our bwt alignment because it is far too slow. While our hash index solution had a processing rate of around 900 to 1600 reads per second, our bwt solution was far slower, with a processing rate of 5 to 27 reads per second (greater the sequence size, the slower the processing rate). This is likely due to the lack of pre-computing the LF and LFC components of the

Figure 5.23: Read alignment output sizes in bytes

FM-Index - which would lead to an even larger increase in the index size and over burden the memory footprint further. The read alignments created by our solution are similar in size to those of **wham** and **soap2**, while being about a third the size of **bwa** output and 25% of **bowtie2** output as seen in Figure 5.23. The output for **snap** is far smaller as it is in compressed BAM format, compared to the uncompressed SAM format of the other aligners. However, our aligner aligns nearly all the available reads similar to **snap**, compared to the 70-80% of **wham** and **soap2**.

5.8.1.3 Consensus Sequence Generation

Appendix Table B.36 shows the consensus sequence generation time (including pre-processing) using a piped **samtools mpileup** into **bcftools** into **vcfutils**. The SAM files used for these tests were from the bowtie2 output in Appendix Tables B.31 and B.33.

Appendix Table B.37 shows the runtimes for consensus calling using **bcftools** and **vcfutils** if the **samtools mpileup** VCF output is already available.

Appendix Table B.39 shows the file sizes of the consensus calling pipeline.

5.8.1.4 Consensus Sequence Generation and Variant calling

This is the area in which our solution shows significant advantages over the existing state of the art. As our data can easily be accessed in an ordered manner via the DBMS engine, our runtimes are $12\times - 16\times$ faster than **samtools** and $16\times - 44\times$ times faster than **soapsnp** as seen in Figure 5.24. Similarly, as we only need to store individual variants in the SNP table, we greatly reduce the storage sizes by nearly three orders of magnitude. In Figure 5.25 we can see that we need a log scale to appropriately represent the storage sizes on a single graph, with **samtools mpileup** output taking up nearly $2740\times - 3840\times$ times as much and **soapsnp** taking up nearly $1700\times - 2300\times$ times as much.

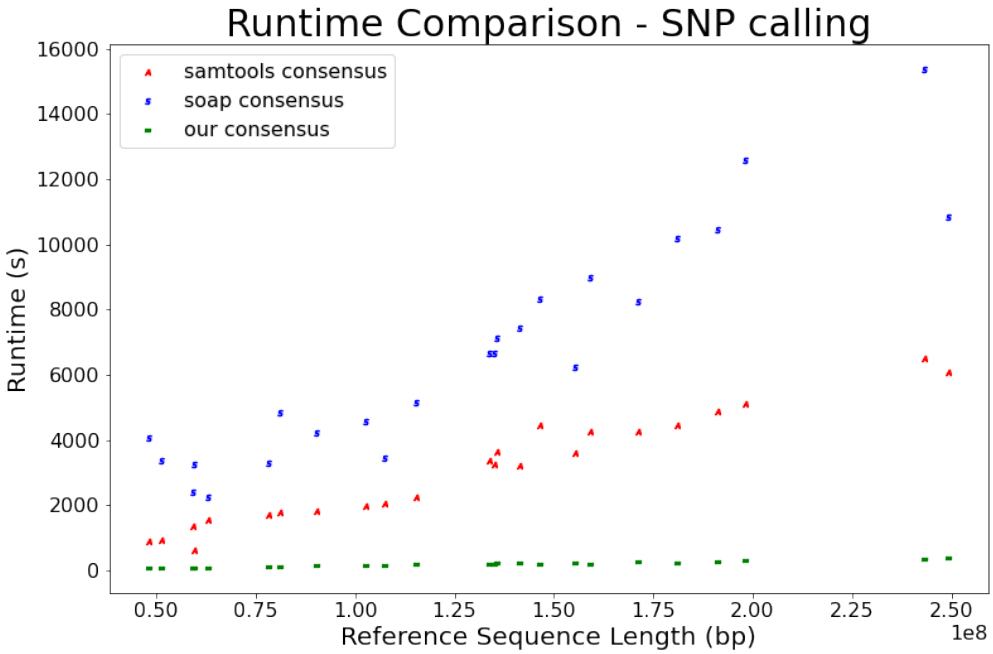

Figure 5.24: Consensus and variant calling times in seconds

It should be noted that **samtools mpileup** can give a compressed output with similar runtimes to uncompressed, and this filesize is only $340\times - 455\times$ larger than ours. Also the compressed output from the **bcftools** variant call once **samtools mpileup** is complete only takes up $3.7\times - 5.2\times$ as much as our solution.

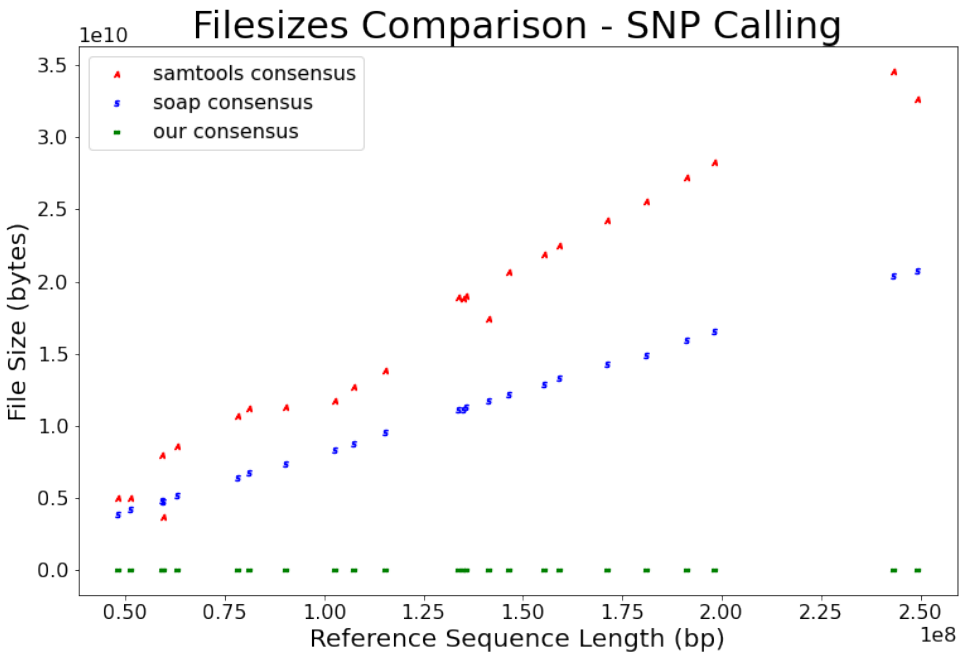

Figure 5.25: Consensus and variant calling sizes in bytes

Appendix Table B.40 shows the time taken to generate variants using **samtools** if the **mpileup** VCF output is available. Appendix Table B.42 shows the corresponding output file sizes for variant calling.

Appendix Table B.41 shows the times taken to call variants (including preprocessing time) using **soapsnp** using the soap2 output. Appendix Table B.43 shows the corresponding output sizes of **soapsnp**. **soapsnp** should be considered as comparable to **samtools mpileup** as it gives a full accounting of each position in the possible sequence, not just the variants.

5.8.2 Full Pipeline Comparison

Given the available tools, we can create some comparative pipelines for executing a complete NGS pipeline. The first of these is the fastest command-line pipeline as shown in Fig 5.26, the most complete single toolbox command-line pipeline as shown in Fig 5.27, and finally our own pipeline as show in Fig 5.28.

When we compare the pipelines, our method shows a clear advantage in runtime as seen in Figure 5.29. The reason for this is two-fold. Firstly, our

Figure 5.26: Fastest Command Line Pipeline

Figure 5.27: Single toolbox Command Line Pipeline

Figure 5.28: Our in-DB Pipeline

sequence alignment resides inside a table in a database engine. The engine provides a multitude of optimisations for the specific operations of sorting and windowing, while a command line has to reconstruct each multiple sequence alignment into a secondary data structure to achieve this. Secondly, most of the work for SNP calling is done in our consensus sequence generation phase and hence we effectively only need to do a table scan to retrieve this information.

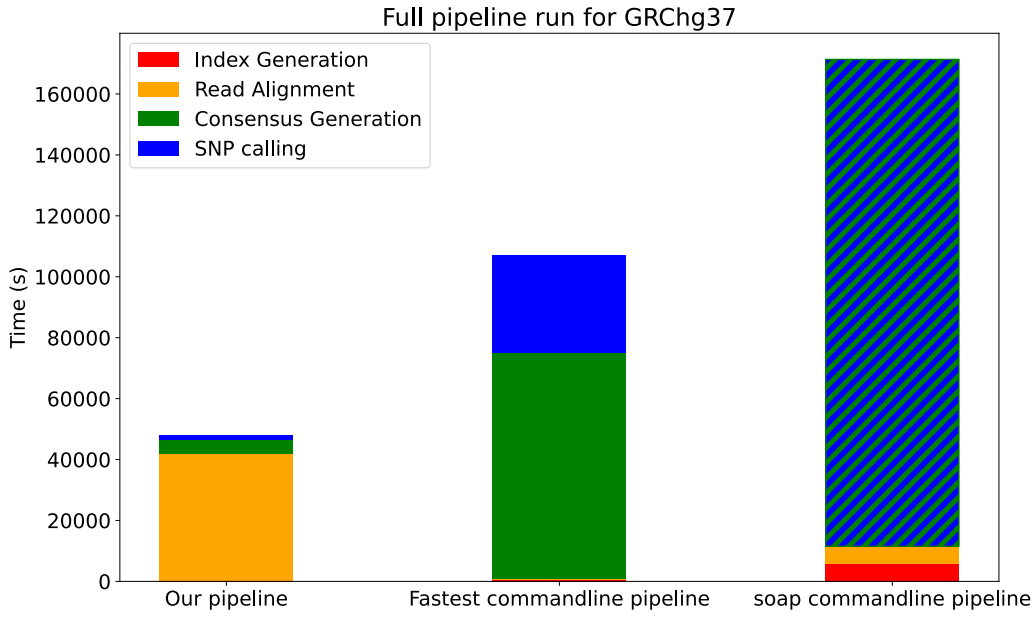

Figure 5.29: Full pipeline comparisons

Chapter 6

Conclusion

This thesis investigated the feasibility of adapting relational database management systems into a genomics platform, with specific focus on Next Generation Sequencing pipelines. The hypothesis was that by applying data management principles to genomics pipelines, in particular by supporting genomics operations directly inside a database engine close to the data, the overall processing performance should be faster than with out-of-database approaches, while at the same time benefiting from the integration of metadata and experimental data in the same schema.

To this end, this thesis presented a series of design alternatives in Chapter 3 which can be applied to different database engines, depending on their capabilities. The overall system design looked into different ways on how to represent large sequence efficiently in database, either integrated as UDTs or via external data wrappers. We also showed how to efficiently integrate the core genomics pipeline functionality using stored procedures, user defined functions and indexes, and with potential hardware acceleration. From this design chapter we can conclude that it is in principle possible to integrate complex genomics data and genomics data analysis into a relational data engine, as long as the engine supports user-defined types and functions.

The sample implementation using Microsoft SQL Server showed some significant areas where this novel approach is ahead of the state of the art of command-line tools based processing. Our approach and implementation provided improvements on the current state of the art including:

- Our index creation time;
- Our consensus sequence generation time; and
- SNP calling time.

This resulted in our total pipeline runtime being faster while also providing experimental data management and integrated metadata capture without user intervention. While our index sizes were larger than the existing methods, our final output file sizes were considerably smaller as well.

6.1 Performance Evaluation – Lessons Learned

We evaluated the prototypical implementation against existing state of the art and common sequence alignment tools using the Human Genome data (GRCh37.1) in order to establish how well the proposed approach enables a data system as a platform for genomic data processing, and also to see which of the design options to choose.

The broad result is that while our approach provides innate data management, this comes at a high cost for the initial alignment stages. The payoff comes in the first analysis stage, which is the consensus sequence generation and variant calling, which is two orders of magnitude faster and requires far less space as we have minimal data duplication.

This result is not too surprising given inherent overhead of a database management system as compared to specialised command-line tools, such as its concurrency control and its row-store architecture. Some of this can be avoided by either using a bit-compressed sequence representation or by using the approach with external data wrappers, which allows to access the sequence data without conversion into the row-store structure. But still a generic data processing platform such as a relational database system has more abstraction layers internally, which induce some processing overhead. In our evaluation this overhead showed clearest during the read alignment phase, due to the dynamic programming nature of the underlying algorithms.

However if we take a step back and look at the performance of an integrated approach over a whole genomics pipeline, then the integrated storage with its

lack of data conversions between different tools starts to pay off. The runtime of the full sequence analysis pipeline for the Human Genome dataset was fastest with the integrated approach, and the SNP calling step was even by far the fastest with a database as genomics platform, as this benefited from the result materialization from the previous steps. The scale of this difference was actually quite surprising and demonstrates clearly the potential of database systems as integrated genomic platforms.

At the same time, the integration of this genomics functionality into the database system also allows for integrated querying of sequence data and metadata about sequences and the corresponding experiments. This makes it possible to easily compare, for example, the results of different experiments, which has been an important task during the recent COVID pandemic in order to be able to identify and trace new virus variants efficiently.

6.2 Adapting to Emerging Hardware

Given the runtime overhead in some of the genomics functionality, especially with the operations during the read-alignment phase, one potential performance optimisation is to make use of hardware acceleration and new emerging hardware. We conducted some experiments with SIMD acceleration which resulted in an order of magnitude of runtime performance.

This is very promising and in future work, it would be great to quantify the parameters for new hardware expansion and demonstrate a methodology that can be used to adapt to emerging hardware. During the course of our system development, the following trends were observed:

- Register size increases for special instruction sets (e.g, extensions of AVX256 such as AVX512 - increases register instructions to 512 bits).
- Data transfer pathway bypasses (e.g. Remote GPU access direct from Network Interface to GPU, Direct storage access direct from GPU to Storage Interface).
- Inclusion of dedicated specialized hardware for matrix operations in more platforms.

- Movement of processors towards a chiplet design, generating more NUMA regions (common access regions of the low level register - instructions between one NUMA region (Non-Uniform Memory Access) requires a single operation, between different regions requires 2 or more instructions).

- Increasing adopting of RISC architecture (ARM chips) as mainstream processors (unlike common CISC architecture that we see in common consumer vendor chips supplied by Intel and AMD).

As such, a production-level implementation of genomics functionality inside a database should ideally take advantage of hardware acceleration as much as possible to take these trends into account. A caveat is however that the standard stored procedure and UDF interfaces of database engines are not very open to hardware acceleration (HW). There is a clear need for better acceleration APIs from database vendors. Such an HW-accelerated implementation also becomes hardware-dependent rather than remaining a generic implementation that can be easily deployed on a variety of hardware configurations.

6.3 Further Development

Based on the background and evaluation, the following ideas would be best suited for immediate implementation.

6.3.1 Read Alignment Improvements

Given that what should be the most expansive portion of read alignment (the dynamic programming portion) only take 5-16% of the runtime, a faster method for string preparation should show a large improvement in performance. Also, finding a way to pre-compute LF and LFC (for FM-Indexing) for BWT-based alignment without causing a bloated index size should greatly speed up BWT-based alignment by reducing the restricting the search cost to a predefined maximum, as opposed to the entire possible search space.

6.3.2 SAM, BAM and CRAM Loading

If we wish to interact with other programs by allowing the user to integrate their output into our pipeline (e.g. run **snap** and take the output BAM file and process it using our consensus sequence generation instead of **samtools mpileup**), then we need to be able to push and pull SAM, BAM and CRAM files to and from our ShortReadAlignments table. To accomplish this, we will need to write parsers for SAM, BAM and CRAM formats similar to our SRF and ZTR parsers.

6.3.3 VCF and BCF Loading

Similar to the situation with BAM or SAM files, if we wish to incorporate outputs and analyses from outside applications, we will need to be able to push and pull VCF or BCF files to and from our SNPTable table. This would require writing a parser for VCF or BCF files similar to our SRF and ZTR parsers.

6.3.4 New BWT-based Index Construction & Search Methods

Another approach for sequence alignment is using a bi-directional BWT string. This is heavily utilized in approaches that rely on GPU Single Instruction Multiple Threads (SIMT) operations to perform alignment. This requires a reorganisation of the FM Index to support bi-directional flow and massively parallel operations. The resulting index structure requires around 2.8 $\times$ the genome length to be indexed [47]. As such it can only fit into GPU memory with a compressed 2 bit representation (3 GB genome $\times$ 2.8 $\times$ 0.25 $=$ 2.1 GB) which means operations must also be carried out with this reduced information precision.

6.4 Future Work

With the continual development of new algorithms and new technology, the sequencing landscape is likely to need further study. In this section, we look at some ways in which this landscape has and will change, along with how we can extend our solution to meet these new challenges. Firstly, we will look at the changing face of sequencing technology, and how it affects some of the assumptions we have made, as well as the changes or extensions we would need

to make to incorporate this into our solution. Secondly, we will look at the possible advancements in algorithms and software that affect the sequencing pipeline.

6.4.1 Technology

Sequencing technology has sufficiently advanced that most of the sequencing technologies referred as NGS are beginning to be referred to as second generation sequencing as we arrive at the next technological spike for sequencing technology (called third generation). While an exact delineation between NGS and third generation is still in contention, the consensus seems to arriving at technologies that satisfy at least some of the following conditions [38]:

1. Real-time reporting of the results. No post processing of the initial detection event is required to identify a base (e.g. base-calling via converting fluorescent images is not required).

2. Single-molecule sequencing. Amplification is not required. (This is a big cost reduction step both in reduction of required reagents and increased size of reads).

3. Significant technological divergence from current methods.

One of the most promising currently available (on a limited basis) third generation technologies is nanopore sequencing, provided by Oxford Nanopore Technologies (**ONT**).

The file format used by ONT is called FAST5. This an extension/implementation of the HDF5 [73; 103] format that is commonly used for scientific data storage. Read sizes average around 5000 to 5500 bp in size, however the technology does not guarantee a fixed read size, just a range. Given these implementation considerations, we would have to add the following tasks to our solution to make it ready for ONT technologies:

8. Parsers for FAST5 format and maybe an internal FAST5 or HDF5 UDT.

9. Extending dynamic programming past the 16-byte barrier (current naive approach would mean at least a 50% performance drop).

6.4.2 Software

Apart from sequencing platforms themselves, the greatest improvements for sequencing generation lie with the applications, indices and algorithms that help align the reads. This is still the most computationally expensive task in creating a consensus sequence. While the majority of indices haven't changed since 2009 (still Hash Indices, FM-Index), the leveraging of new hardware technologies to parallelize processing has made an impact. New algorithms have been developed to take advantage of cloud technology, CPU and GPU acceleration. In our project we explored CPU acceleration through the use of SIMD technology in Chapter 4.3.3.4. Thus, a likely future work goal would be to add GPU acceleration to our solution.

6.4.2.1 GPU Acceleration

A GPU is essentially a collection of specific processing units running a similar task in a massively parallel manner. The downside is that the specific usage of these processing units is far more restrictive than a CPU, as well as the direct memory that is accessible to them. The use of GPUs for general purpose computing (i.e. - anything other than their intended task of rendering graphics) is called General Purpose GPU or gpGPU. This term is also used to describe certain GPUs that are released specifically with gpGPU computing in mind (typically with higher memory capacity than their traditional commodity versions).

In terms of software development for the field of gpGPU, two main platforms for exist:

- **OpenCL** A general purpose programming language based on a C++ kernel for CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators that can handle most GPUs from different vendors (NVIDIA™, AMD™, Intel™) [100].

- **CUDA** The first widespread development platform for gpGPU tasks – released by NVIDIA™ and mainly compatible with just that vendor [76].

Several popular GPU-based aligners exist, the most popular of which are CUSHAW [59], CUSHAW2-GPU [58], BarraCUDA [48] and SOAP3-dp [61].

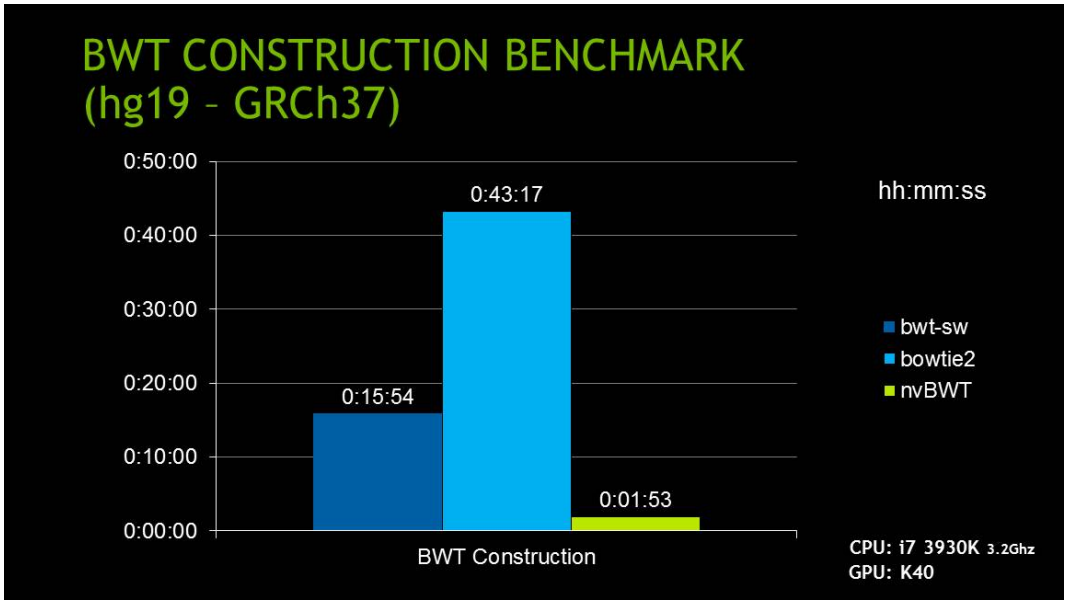

Figure 6.1: nvBIO - BWT construction time comparison [85]

While BarraCUDA and SOAP3-dp provide faster methods for BWT construction (both on reference sequence and on a collection of reads), CUSHAW variants use the GPU to help speed up and re-use parts of the Smith-Waterman computation. These two different approaches have been explored extensively but most of the new aligners being developed are based on the construction of BWT strings for the collection of reads.

As the most widespread platform, CUDA, most new aligners are written with a CUDA-capable GPU in mind. Given their popularity, NVIDIA itself released their own NGS sequence alignment library called NVBIO [78; 85]. Their library provides the most popular gpGPU NGS alignment tools in the form of BWT construction as show in Figure 6.1 and dynamic programming acceleration as shown in Figure 6.2. Given that the tasks we wish to use have already been written and compiled in C++ and CUDA, an approach for integration of these features we can try is to make a C# wrapper that will utilize the function calls and then integrate that C# DLL into the DBMS.

gpGPU frameworks such as CUDA and OpenCL have proven to be valuable in the genomic processing space. However, implementation and tight integration is still haphazard due to the nature of certain RDBMS platforms (e.g, even the latest version of SQL Server 2022 runs on .NET Framework 4.8, while the CUDA

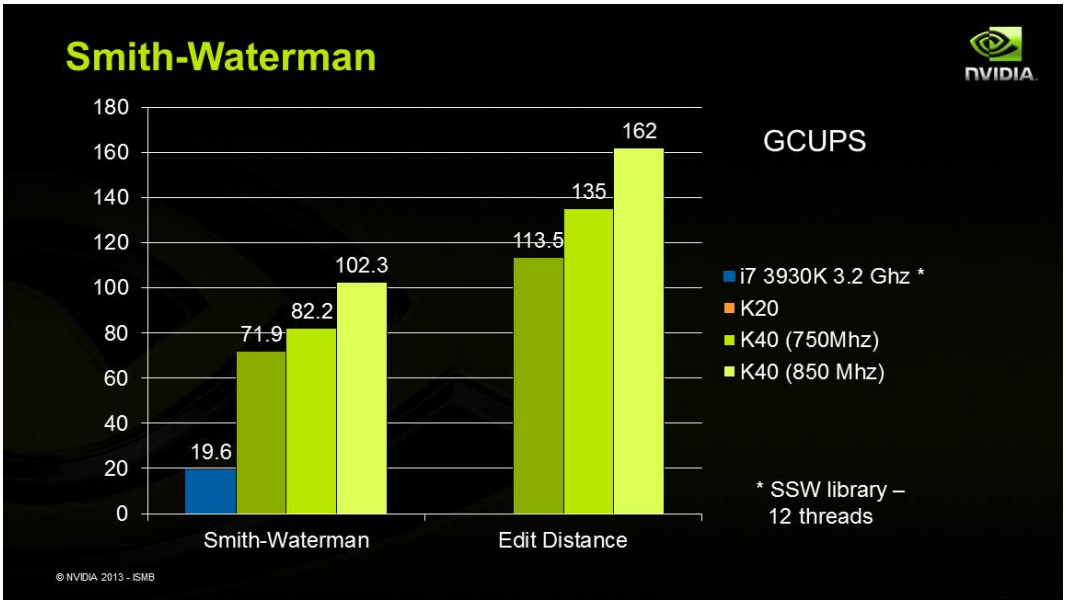

Figure 6.2: nvBIO - Smith-Waterman throughput comparison [85]

and OpenCL framework integrations require .NET 6.0 to function). As such, it might be more realistic to decouple the tight integration we have explored and use a different method such as integrating pipelined results from an existing command line solution that is not managed by the RDBMS.

However, if these implementation issues can be solved, I can draw some valuable speculations given these new algorithms and index formats. This would allow me to add some or all the following tasks to my solution to make it competitive with GPU-accelerated aligners:

10. Create new BWT indices that can utilize GPU-accelerated construction algorithms for **reference sequences**.

11. Create new BWT indices that can utilize GPU-accelerated construction algorithms for **collections of reads**.

12. Use a .NET library such as ManagedCUDA to write updated GPU-accelerated algorithms for dynamic programming.

13. Integrate existing aligners such as NVBIO into the DBMS by constructing wrappers that allow .NET access.

6.5 Take-Aways and Implementation Barriers

- **Dealing with Framework APIs.** A great deal of time was consumed in making arbitrary feature-set *X* work with feature-set *Y* like problems. This was especially true for the .NET standard. While products kept rolling out from the primary vendor that were built on old versions of .NET (such as SQL Server 2018 and even SQL Server 2022 only supporting .NET Framework 4.8), other companies built their products on the latest versions - i.e. .NET 6. It caused a great deal of frustration, especially from a performance mindset, as that meant that a lot of managed code could not be written in a high level language such as C# (utilising optimisations found in .NET 6) and instead had to be written in C++/CUDA and then managed by C# wrappers. This was detrimental from a development standpoint as one of the great benefits of using a high level language running its latest API would be the easy to code use of performance tools.

- **Data and Analysis Velocity.** During the course of development, COVID-19 happened. This showcased the need for rapid data migration and analysis so that different strains could be identified. A great deal of PCR data items were acquired, but alongside this, a great many sequences for the virus were also captured. The need for rapid variant identification and visualisation became readily apparent to all stakeholders. I strongly believe that visualisations of key data points moving between those labs involved with tracking the spread of the virus and with the populace was instrumental in orchestrating a world wide response. Those lines of communication that were pulled together in an ad-hoc manner should now be formalised.

6.6 Final Words

A thesis is a complicated task with an incredible amount of time invested by many parties. The minutiae of systems and solutions that is explored is quite amazing to those involved in the process. I would again wish to re-iterate my thanks to those mentioned in the acknowledgements.

This appendix collates extraneous information which is not necessary but helpful for some extended explanations of background material, as well as tabular data of evaluation results.

Appendix A

Detailed NGS Technology descriptions

In this Appendix chapter we provide a more in depth look into the next generation sequencing platforms that are available. Most of the information in this section is collated materials from vendors and the forums of interested parties, particularly SeqAnswers [22; 23; 62; 13; 38; 37; 101].

A.1 Illumina

One of the first three companies to offer a next generation sequencing solution, Illumina uses a process called **Sequencing by Synthesis** for sequencing [22].

Step 1: Sample Preparation

The DNA sample of interest is sheared to appropriate size (average 800bp) using a compressed air device known as a nebulizer. The ends of the DNA are polished, and two unique adapters are ligated to the fragments. Ligated fragments of the size range of 150-200bp are isolated via gel extraction and amplified using limited cycles of PCR.

This process is a fairly straightforward multi-step molecular biology process, however there are many pitfalls that can result in skewed results downstream.

Steps 2-6: Cluster Generation by Bridge Amplification

In contrast to the 454 and ABI methods which use a bead-based emulsion

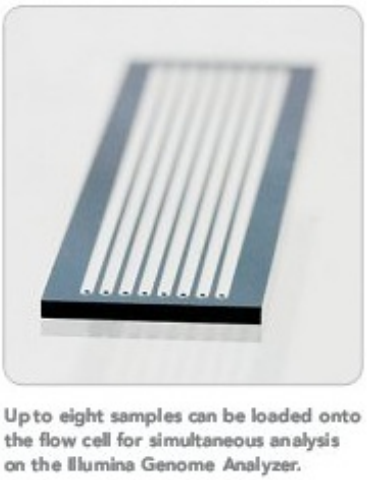

Figure A.1: Illumina Flowcell [22]

PCR to generate **polonies**, Illumina utilizes a unique **bridged** amplification reaction that occurs on the surface of the flow cell.

The flow cell surface is coated with single stranded oligonucleotides that correspond to the sequences of the adapters ligated during the sample preparation stage. Single-stranded, adapter-ligated fragments are bound to the surface of the flow cell exposed to reagents for polyermase-based extension. Priming occurs as the free/distal end of a ligated fragment **bridges** to a complementary oligo on the surface.

Repeated denaturation and extension results in localized amplification of single molecules in millions of unique locations across the flow cell surface. This process occurs in what is referred to as Illumina's **cluster station**, an automated flow cell processor.

Steps 7-12: Sequencing by Synthesis

A flow cell containing millions of unique clusters is now loaded into the 1G sequencer for automated cycles of extension and imaging.

The first cycle of sequencing consists first of the incorporation of a single fluorescent nucleotide, followed by high resolution imaging of the entire flow cell. These images represent the data collected for the first base. Any signal above background identifies the physical location of a cluster (or polony), and the fluorescent emission identifies which of the four bases was incorporated at that position.

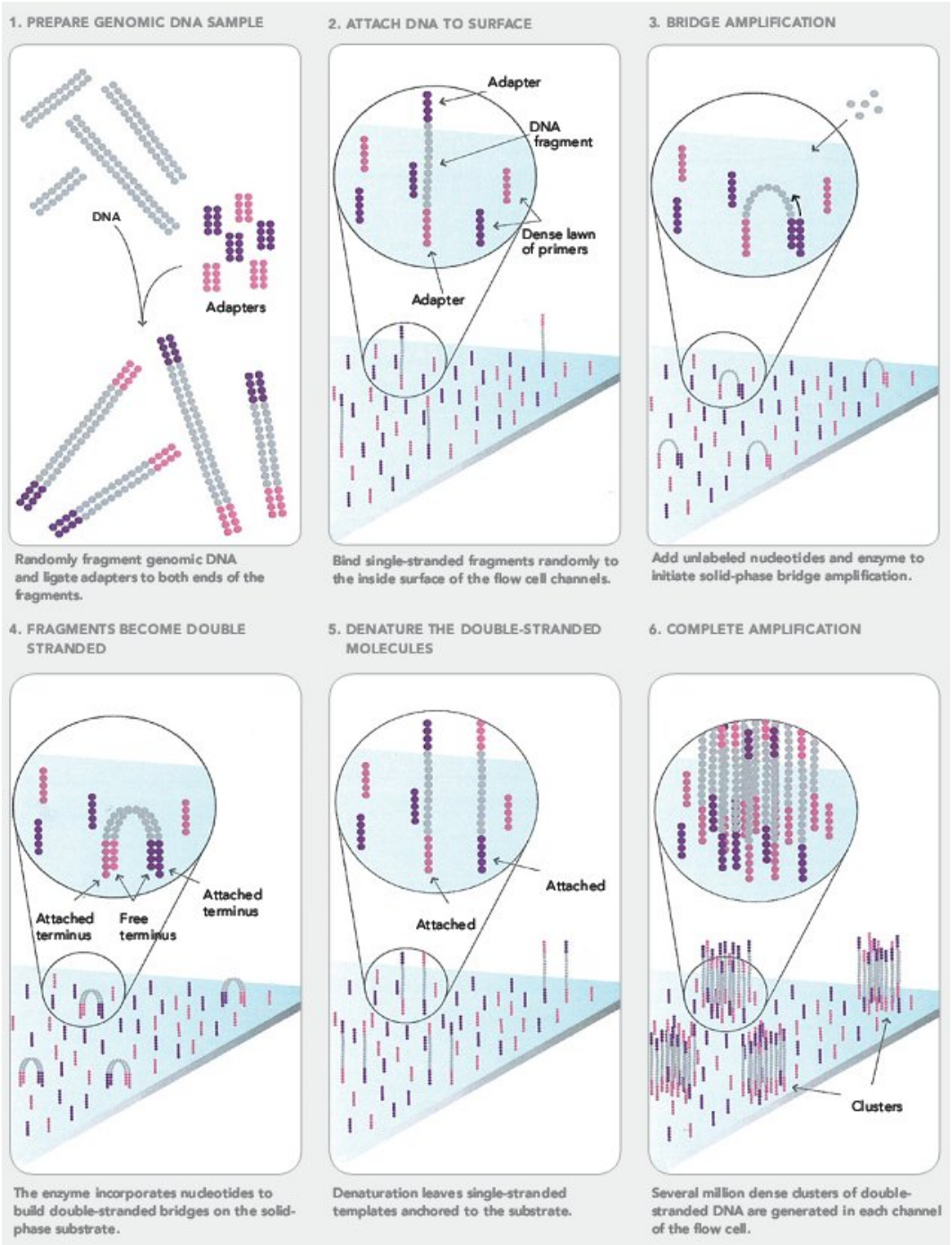

Figure A.2: Illumina Steps 1 to 6 [22]

This cycle is repeated, one base at a time, generating a series of images each representing a single base extension at a specific cluster. Base calls are derived with an algorithm that identifies the emission colour over time. At this time reports of useful Illumina reads range from 26-50 bases.

Steps 7-12: Sequencing by Synthesis

The use of physical location to identify unique reads is a critical concept for all next generation sequencing systems. The density of the reads and the ability to image them without interfering noise is vital to the throughput of a given instrument. Each platform has its own unique issues that determine this number, 454 is limited by the number of wells in their PicoTiterPlate, Illumina is limited by fragment length that can effectively **bridge**, and all providers are limited by flow cell real estate.

Originally, Illumina data output comes in a format called QSEQ that is similar to FASTQ, with different quality values specific to Illumina. However, After Pipeline update 1.8, this Illumina specific format was discarded and FASTQ format was chosen as the default output format.

A.2 ABI SoLiD

Applied Biosystems has launched their instrument, which supports their version of high-throughput sequencing chemistry, termed “SOLiD”. Acquired from Agencourt Personal Genomics in late 2006, SOLiD is a unique parallel chemistry which enables simultaneous sequencing of thousands of individual DNA molecules [23].

Here I will present a brief overview of the technology. Figures and information taken directly from this presentation from ABI’s website.

Sequencing on the SOLiD machine starts with library preparation. In the simplest fragment library, two different adapters are ligated to sheared genomic DNA (left panel of Figure A.5). If more rigorous structural analysis is desired, a “mate-pair” library can be generated in a similar fashion, be incorporating a circularizationcleavage step prior to adapter ligation (right panel of Figure A.5).

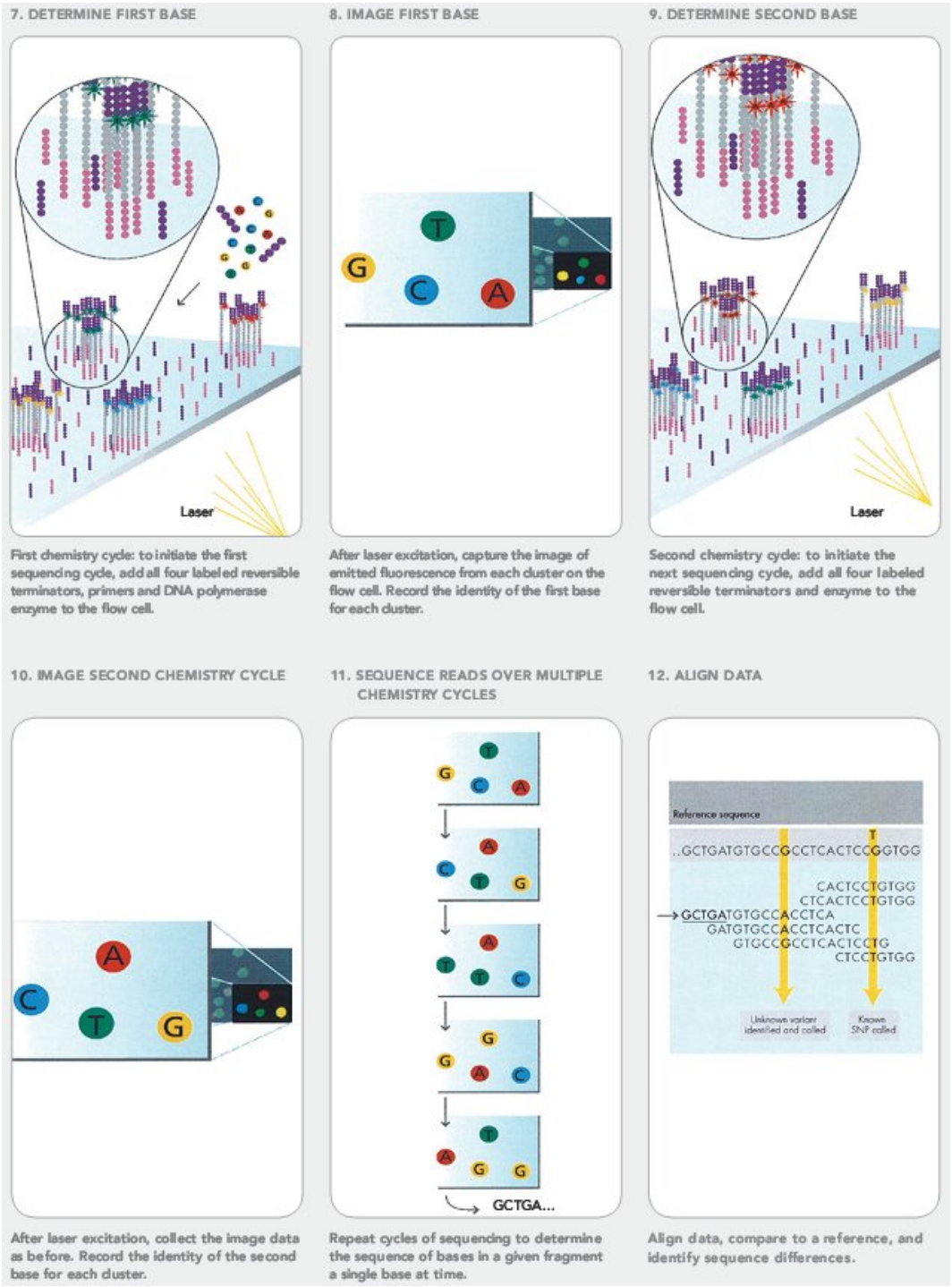

Figure A.3: Illumina Steps 7 to 12 [22]

Figure A.4: SOLiD Machine [23]

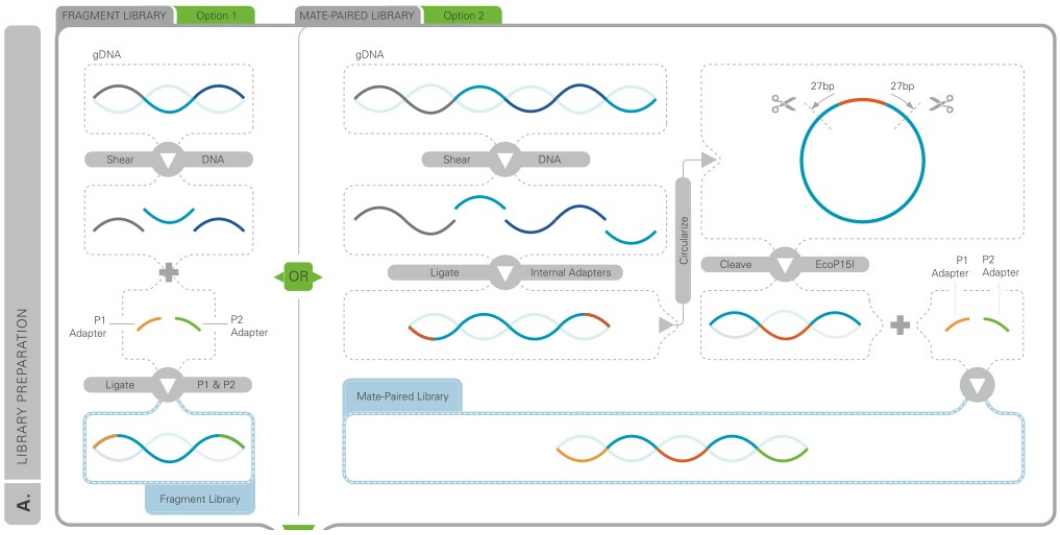

Figure A.5: Library generation schematic. [23]

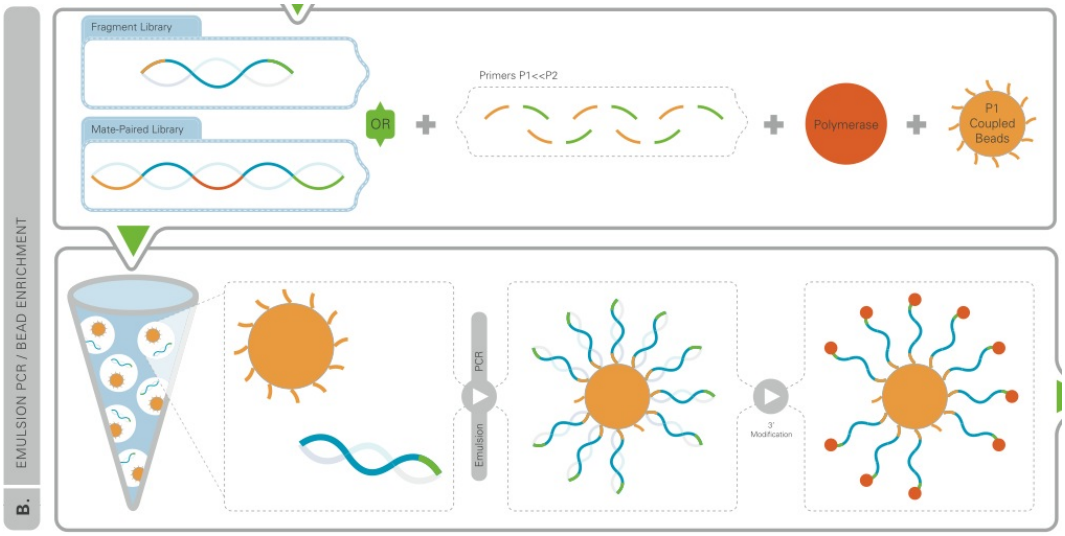

Figure A.6: Clonal bead library generation via emulsion PCR. [23]

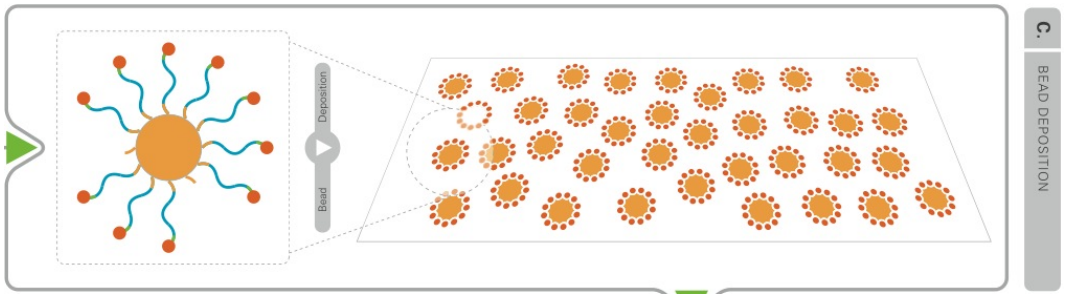

Figure A.7: Depositing beads into flow cell via end modifications. [23]

Once the adapters are ligated to the library, emulsion PCR is conducted using the common primers to generate “bead clones” which each contain a single nucleic acid species.

Each bead is then attached to the surface of a flow cell via 3́modifications to the DNA strands.

At this point, we have a flow cell (basically a microscope slide that can be serially exposed to any liquids desired) whose surface is coated with thousands of beads each containing a single genomic DNA species, with unique adapters on either end. Each microbead can be considered a separate sequencing reaction which is monitored simultaneously via sequential digital imaging. Up to this point all next-gen sequencing technologies are very similar, this is where ABISOLiD diverges dramatically (see Figure A.8).

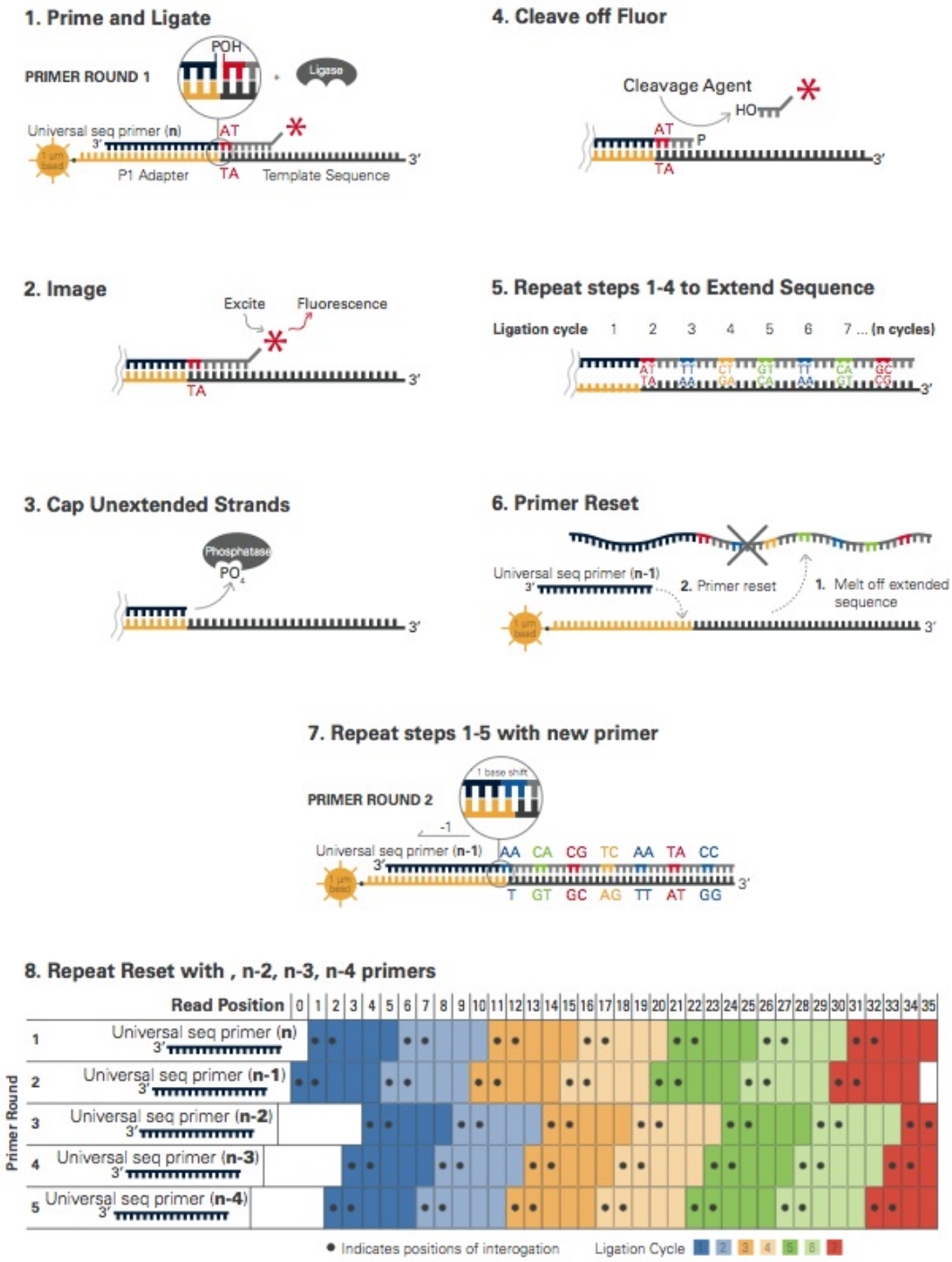

Figure A.8: Schematic of ABI SOLiD v2.0 sequencing chemistry [23]

SOLiD 2.0 chemistry utilizes 12 encoding (meaning bases 12 of the probe are the specific bases linked to the colourspace calls). The original version of the chemistry used 4,5 encoded probes.

The actual base detection is no longer done by the polymerase-driven incorporation of labeled di-deoxy terminators. Instead, SOLiD uses a mixture of labeled oligonucleotides and queries the input strand with ligase. Understanding the labeled oligo mixture is key to understanding SOLiD technology.

Each oligo has degenerate positions at bases 3-5 (Ns), and one of 16 specific di-nucleotides at positions 1-2 (numbered from the 3' end). Positions 6 through the 5' are also degenerate (likely inosine, not confirmed), and hold one of four fluorescent dyes. The sequencing involves:

1. Anneal a primer, then hybridize and ligate a mixture of fluorescent oligos (8-mers) whose 1st & 2nd 3' bases match that of the template

2. Capping un-extended fragments with the same mixture of non-fluorescent probes

3. Phosphatase treatment to prevent any remaining un-extended strands from contributing to out of phase ligation events

4. Detection of the specific fluorescent markers

5. Removal of fluorescent markers via two step chemical cleavage of the three 5' bases. This leaves behind a 5 base ligated probe, with a 5' phosphate

6. Repeat, this time querying the 6th & 7th bases

7. After 5-7 cycles of this, perform a reset, in which the initial primer and all ligated portions are melted from the template and discarded.

8. Next a new initial primer is used that is N-1 in length. Repeating the initial cycling (steps 1-5) now generates an overlapping data set (bases 1/2, 6/7, etc, see Figure A.8, Step 8 above).

Thus, 5-7 ligation reactions followed by 5 primer reset cycles are repeated generating sequence data for 35 contiguous bases, in which each base has been queried by two different oligonucleotides.

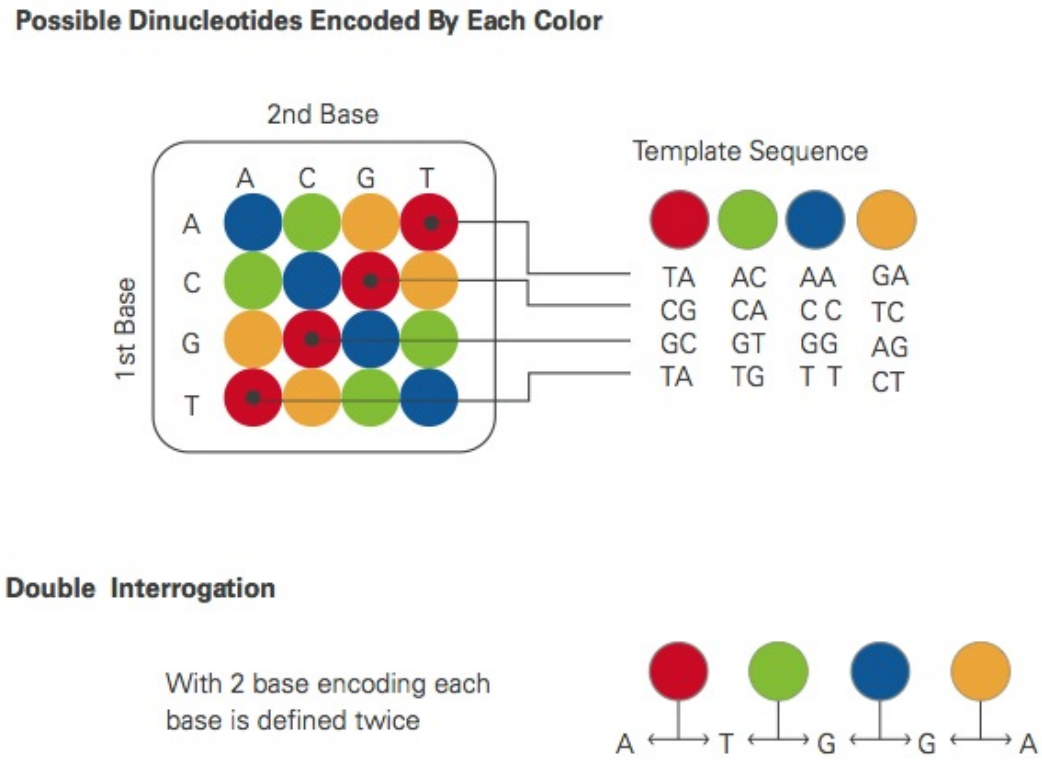

Figure A.9: Schematic of di-base encoding, and how it relates to calling the actual template sequence [23]

If you are doing the math you have realized there are 16 possible di-nucleotides ($4\hat{2}$) and only 4 dyes. So data from a single colour call does not tell you what base is at a given position. This is where the brilliance (and potential confusion) comes about with regard to SOLiD. There are 4 oligos for every dye, meaning there are four di-nucleotides that are encoded by each dye.

For example (see Figure A.9), the di-nucleotides CA, AC, TG, and GT are all encoded by the green dye. Because each base is queried twice it is possible, using the two colours, to determine which bases were at which positions. This two colour query system (known as **colour space**) has some interesting consequences with regard to the identification of errors. A detailed explanation of colour space and it's unique issues can be found in the PDF files attached to this post.

One of the side effects of this dual encoding is that when aligning to a reference and attempting to determine variants, true variants will follow specific colour change rules as defined in Figure A.10.

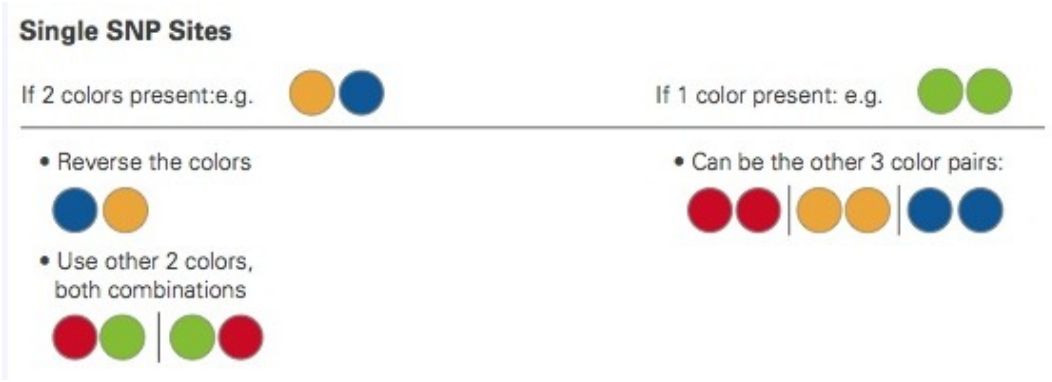

Figure A.10: Colourspace valid variant rules [23]

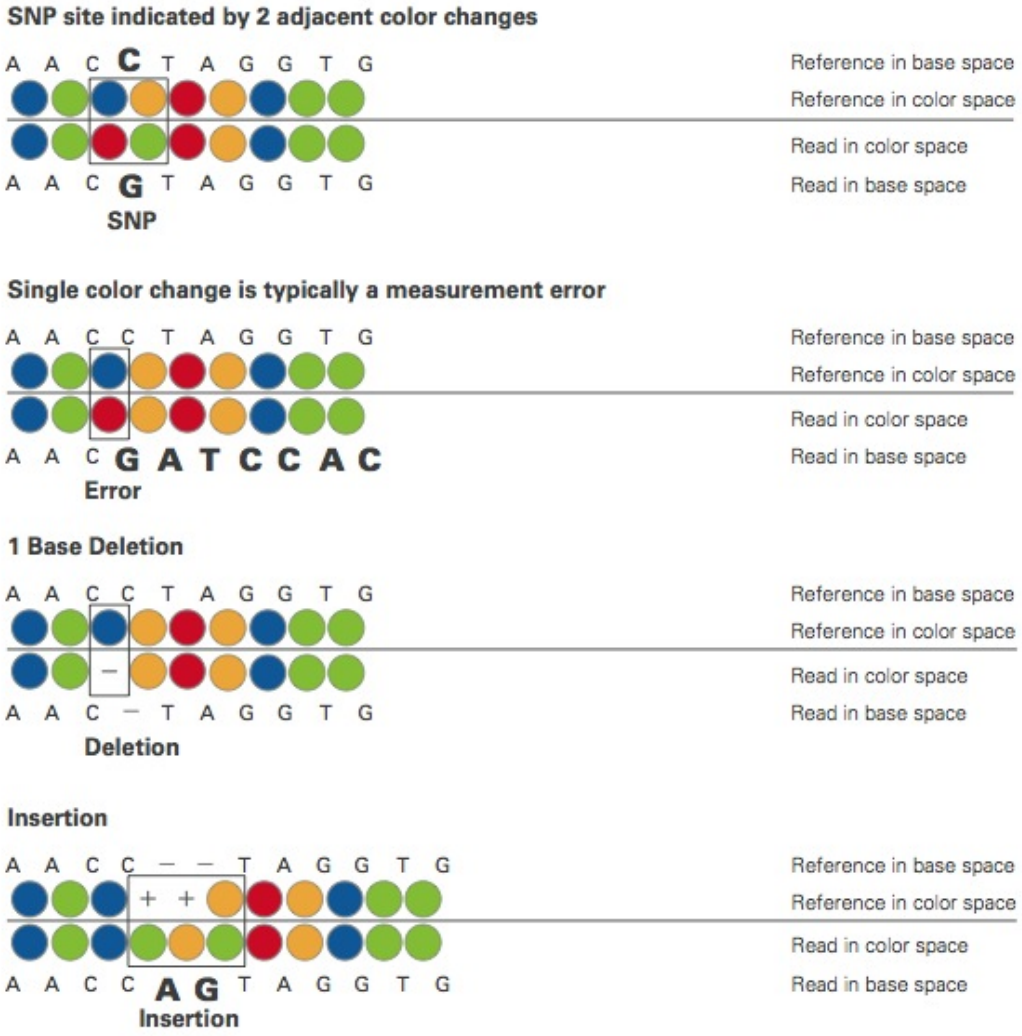

Figure A.11: Colourspace examples [23]

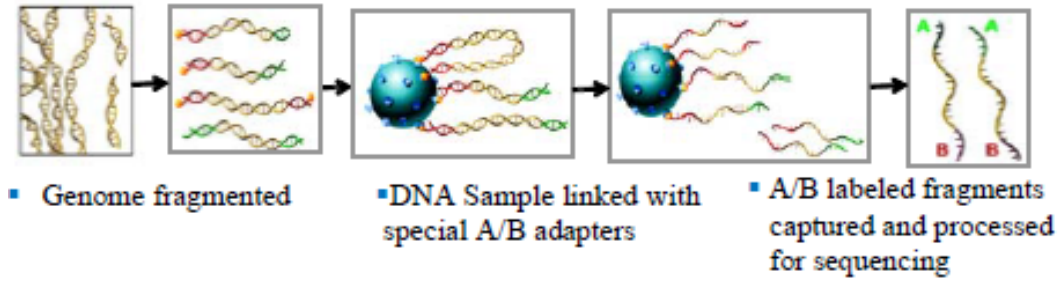

Figure A.12: 454 sequencing method part 1.1 Library preparation: breaking double-stranded DNA into single-stranded DNA and attaching it with an adapter [62]

Detection of a true SNP is reflected by changes in two adjacent colourspace calls, which must follow the rules above. Figure A.11 below gives some examples of this principle in examining alignments.

A.3 Roche 454

Roche 454 sequencing using a sequencing by synthesis method called pyrosequencing. The particular innovation for NGS that 454 Technologies developed was massively parallelizing this pyrosequencing process. The full pipeline is outlined in Figure A.17 [62].

The output of a Roche 454 sequence is in the form of a flowgram that measures the light intensity output of the final stage of sequencing. This flowgram output is in the SFF output.

Step 1. Library preparation. The first step of 454 sequencing is to prepare the library. The user first has long, double stranded DNA prior to the preparation. The user then must break the double-stranded DNA into short, double-stranded DNA. Then, the fragments are joined with an adaptor at either end. The fragments are then separated into single stranded DNA. This can be seen in Figure A.12 and Figure A.13.

Next, the DNA fragments will be joined with the micro-sized beads. After mixing, the library fragment is captured by the specific oligonucleotide on the bead.

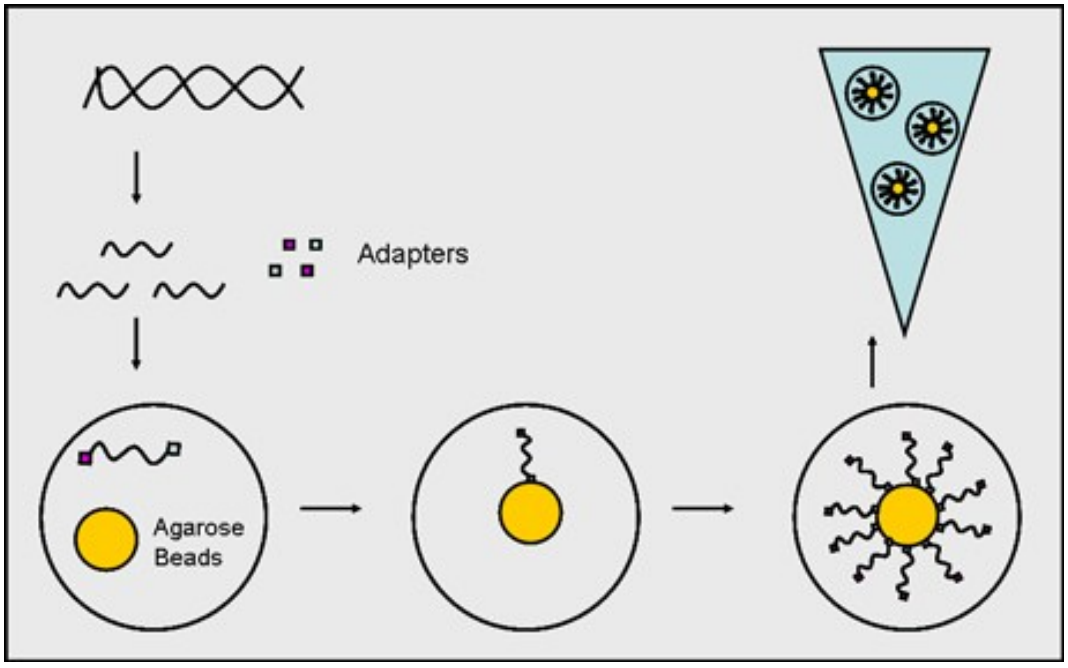

Figure A.13: 454 sequencing method part 1.2 Library preparation and beads loading: attaching library fragments onto beads. Step 2 is also illustrated here as emulsions are formed within the container [62]

Step 2 Emulsion formation. Each of these fragment plus bead complexes is then mixed with emulsion oil. The combination and shaking causes the water to form droplets around the beads, called an emulsion and that is why the subsequent process is called emulsion PCR.

Step 3 Emulsion PCR. As each droplet/emulsion contains only one DNA fragment, the water contains the necessary components to amplify the sequence. This process is referred as emulsion PCR, which begins within these micelles producing one million copies of each DNA fragment on the surface of each bead.

Step 4 Beads loading. Micelles are then broken and enrich for DNA-positive beads. Load all these beads into a picotiter plate, in which one well only fits one bead. Steps 3 and 4 are illustrated in Figure A.14 and Figure A.15.

Basically, the light signal is captured as the product is amplified as per Figure A.16. Using the light signal from the CCD camera, it generates a flowgram as seen in Figure A.18.

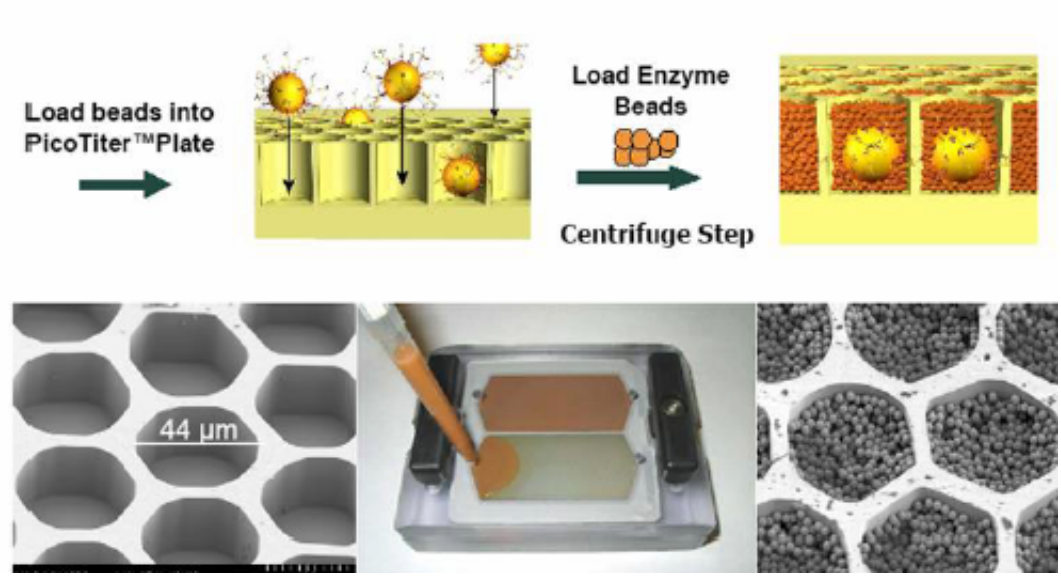

Figure A.14: 454 sequencing method part 3 and 4. Emulsion PCR + Enrich beads [62]

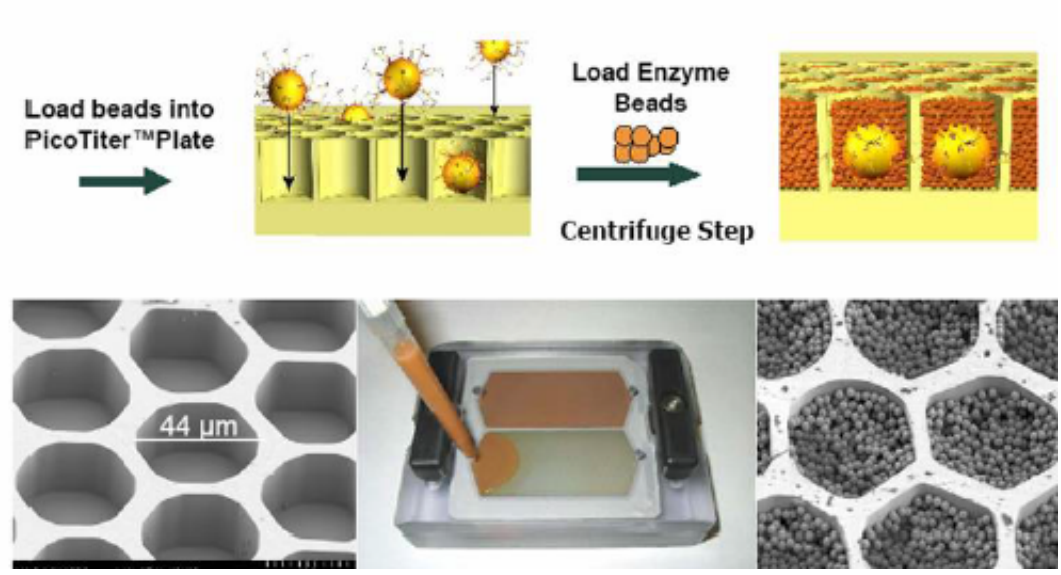

Figure A.15: 454 sequencing method part 4. Loading bead onto picotiter plate [62]

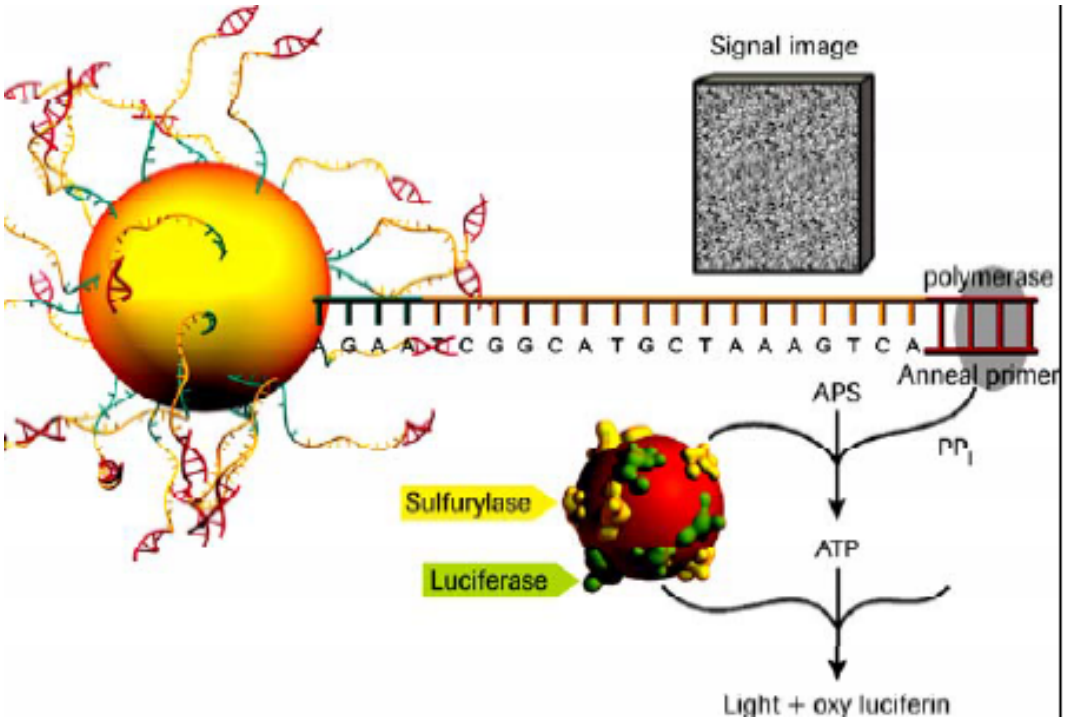

Figure A.16: 454 sequencing method part 5. Actual pyrosequencing and light emission [62]

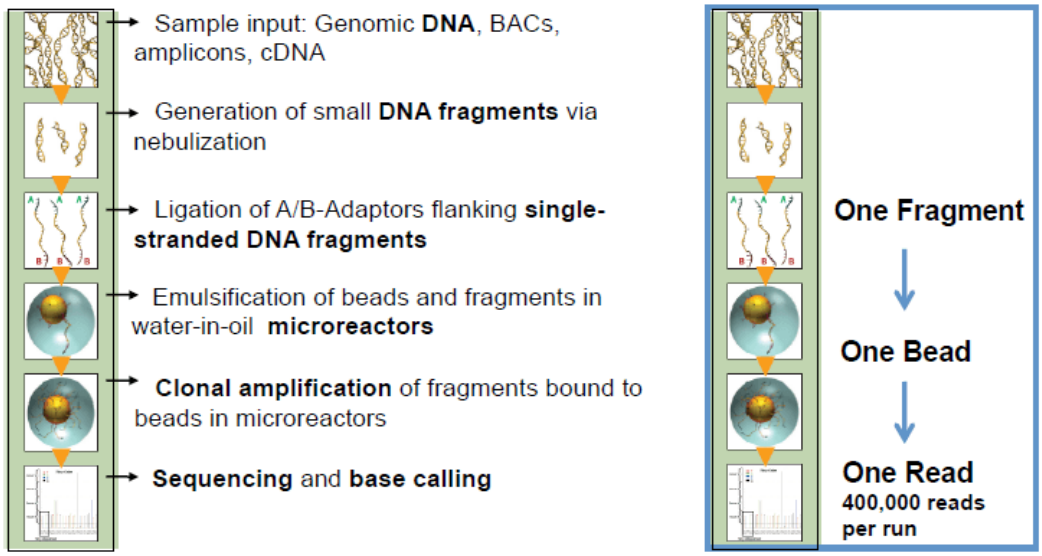

Figure A.17: 454 sequencing Full pipeline description [62]

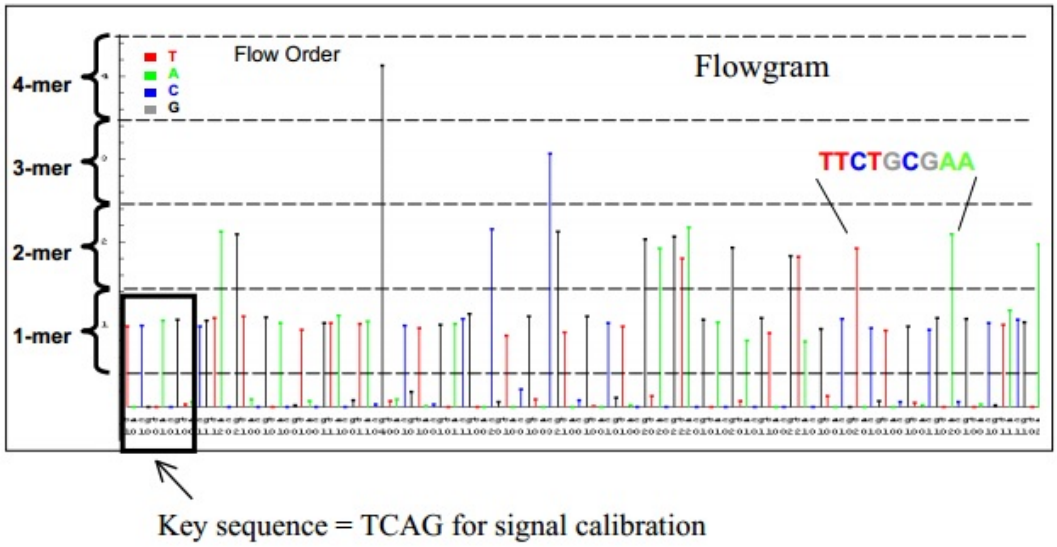

Figure A.18: 454 Flowgram example [62]

A.4 Ion Torrent

Ion torrent and Ion proton sequencing do not make use of optical signals. Instead, they exploit the fact that addition of a deoxyribonucleoside triphosphate (dNTP) to a DNA polymer releases an hydrogen ion.

In nature, the incorporation of a dNTP into a growing DNA strand involves the formation of a covalent bond and the release of pyrophosphate and a positively charged hydrogen ion.A dNTP will only be incorporated if it is complementary to the leading unpaired template nucleotide. Ion semiconductor sequencing exploits these facts by determining if a hydrogen ion is released upon providing a single species of dNTP to the reaction as seen in Figure A.19.

Microwells on a semiconductor chip that each contain many copies of one single-stranded template DNA molecule to be sequenced and DNA polymerase are sequentially flooded with unmodified A, C, G or T dNTP. If an introduced dNTP is complementary to the next unpaired nucleotide on the template strand it is incorporated into the growing complementary strand by the DNA polymerase. If the introduced dNTP is not complementary there is no incorporation and no biochemical reaction as seen in Figure A.20. The hydrogen ion that is released in the reaction changes the pH of the solution, which is detected by an ISFET. The unattached dNTP molecules are washed out before the next cycle

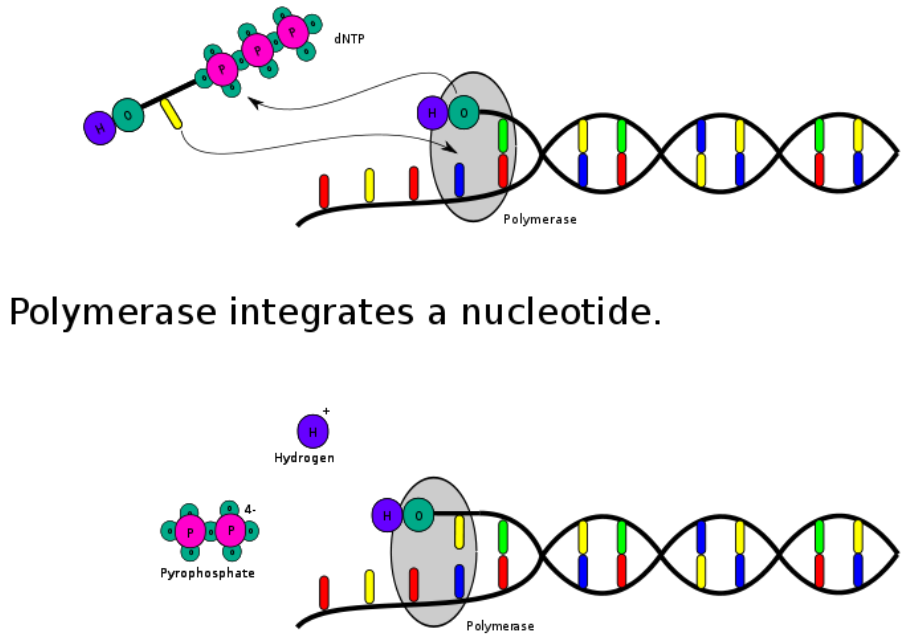

Figure A.19: The incorporation of dNTP into a growing DNA strand causes the release of hydrogen and pyrophosphate [101].

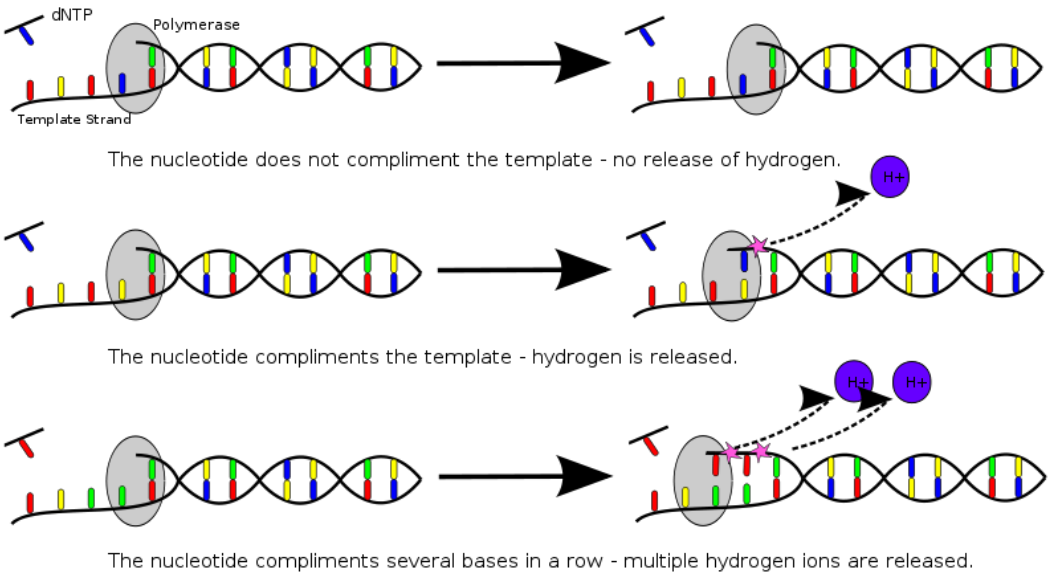

Figure A.20: The incorporation of dNTP into a growing DNA strand causes the release of hydrogen and pyrophosphate [101].

when a different dNTP species is introduced.

Beneath the layer of microwells is an ion sensitive layer, below which is an ISFET ion sensor which is described in Figure A.22. All layers are contained within a CMOS semiconductor chip, similar to that used in the electronics industry.

Each chip contains an array of microwells with corresponding ISFET detectors. Each released hydrogen ion then triggers the ISFET ion sensor. The series of electrical pulses transmitted from the chip to a computer is translated into a DNA sequence in a manner shown in Figure A.21 with no intermediate signal conversion required. Because nucleotide incorporation events are measured directly by electronics, the use of labeled nucleotides and optical measurements are avoided. Signal processing and DNA assembly can then be carried out in software.

The major benefits of ion semiconductor sequencing are rapid sequencing speed and low upfront and operating costs. This has been enabled by the avoidance of modified nucleotides and optical measurements.

Because the system records natural polymerase-mediated nucleotide incorporation events, sequencing can occur in real-time. In reality, the sequencing rate is limited by the cycling of substrate nucleotides through the system. Ion

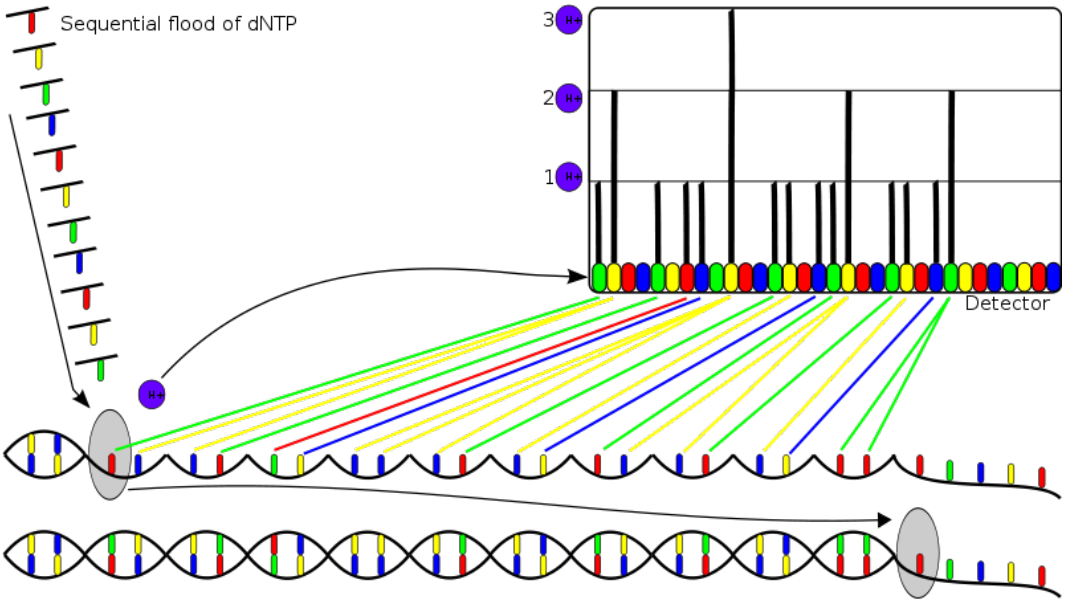

Figure A.21: Released hydrogen ions are detected by an ion sensor. Multiple incorporations lead to a corresponding number of released hydrogen ions and intensity of signal [101].

Torrent Systems Inc., the developer of the technology, claims that each incorporation measurement takes 4 seconds and each run takes about one hour, during which 100 to 200 nucleotides are sequenced. If the semiconductor chips are improved, the number of reads per chip should increase.

If homopolymer repeats of the same nucleotide (e.g. TTTTT) are present on the template strand (strand to be sequenced) then multiple introduced nucleotides are incorporated and more hydrogen ions are released in a single cycle. This results in a greater pH change and a proportionally greater electronic signal. This is a limitation of the system in that it is difficult to enumerate long repeats. This limitation is shared by other techniques that detect single nucleotide additions such as pyrosequencing. Signals generated from a high repeat number are difficult to differentiate from repeats of a similar but different number; e.g., homorepeats of length 7 are difficult to differentiate from those of length 8.

Another limitation of this system is the short read length compared to other sequencing methods such as Sanger sequencing or pyrosequencing. Longer read lengths are beneficial for *de novo* genome assembly. Ion Torrent semiconductor sequencers produce an average read length of approximately 400 nucleotides

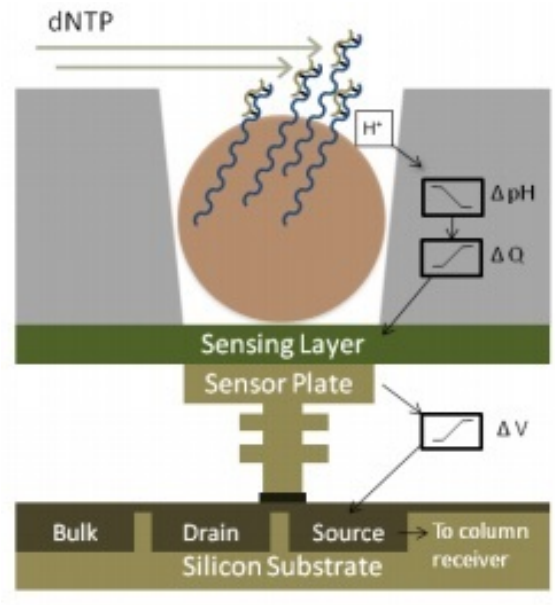

Figure A.22: Schematic cross-section of a single well of an Ion Torrent sequencing chip. The well houses Ion Sphere particles containing DNA template. When a nucleotide incorporates, a proton releases and the pH of the well changes. A sensing layer detects the change in pH and translates the chemical signal to a digital signal [13].

per read.

The throughput is currently lower than that of other high-throughput sequencing technologies, although the developers hope to change this by increasing the density of the chip.

One of the most promising currently available (on a limited basis) third generation technologies is nanopore sequencing, provided by Oxford Nanopore Technologies (**ONT**).

A.5 Nanopore sequencing

The actual proof of concept for nanopore sequencing was established in 1996 [45], long before NGS machines had even hit the market or even the Human Genome Project had been completed. The first breakthrough came in [45] showing that single-stranded DNA or RNA could be driven through large α-hemolysin channels (the nanopore) by electrophoresis. In particular, it was

found that passage through the channel blocks ion flow, decreasing the current for a length of time proportional to the length of the nucleic acid (allowing for the detection of specific bases). The second set of breakthroughs came in showing that non-biological solid state technologies (which would allow for greater stability in manufacture, measurement, cost and throughput) could be used to generate suitable nanopores [43] and even allow for the sequencing of double-stranded DNA [19]. The sequencing process is outlined below and in Figure A.23 [38]:

1. **Step A** - Suspended library molecules are concentrated near nanopores embedded in the membrane. A voltage applied across the membrane induces a current through the nanopores.

2. **Step B** - Schematic of a library molecule, showing dsDNA ligated to a leader adapter pre-loaded with a motor protein and a hairpin adapter and the tethering oligos.

3. **Step C** - Sequencing starts from the 5' end of the leader adapter. The motor protein unwinds the dsDNA allowing single-stranded DNA to pass through the pore.

4. **Step D** - A flow cell contains 512 channels (grey), each channel consisting of 4 wells (white). Each well contains a pore (blue) and a sensor. At any given time, the device is recording the data stream from the wells of the active well-group, in this example, g1.

5. **Step E** - Perturbation in the current across the nanopore is measured 3,000 times per second as ssDNA passes through the nanopore.

6. **Step F** - The 'bulk data' are segmented into discrete 'events' of similar consecutive measurements. The 5-mer corresponding to each event is inferred using a statistical model.

7. **Step G** - The 1D base-calls are inferred separately for the template and complement event signals.

8. **Step H** - Alignment of the 2D base-calls from the event signals from both, and the 1D base-calls are used to constrain the 2D base-calls.

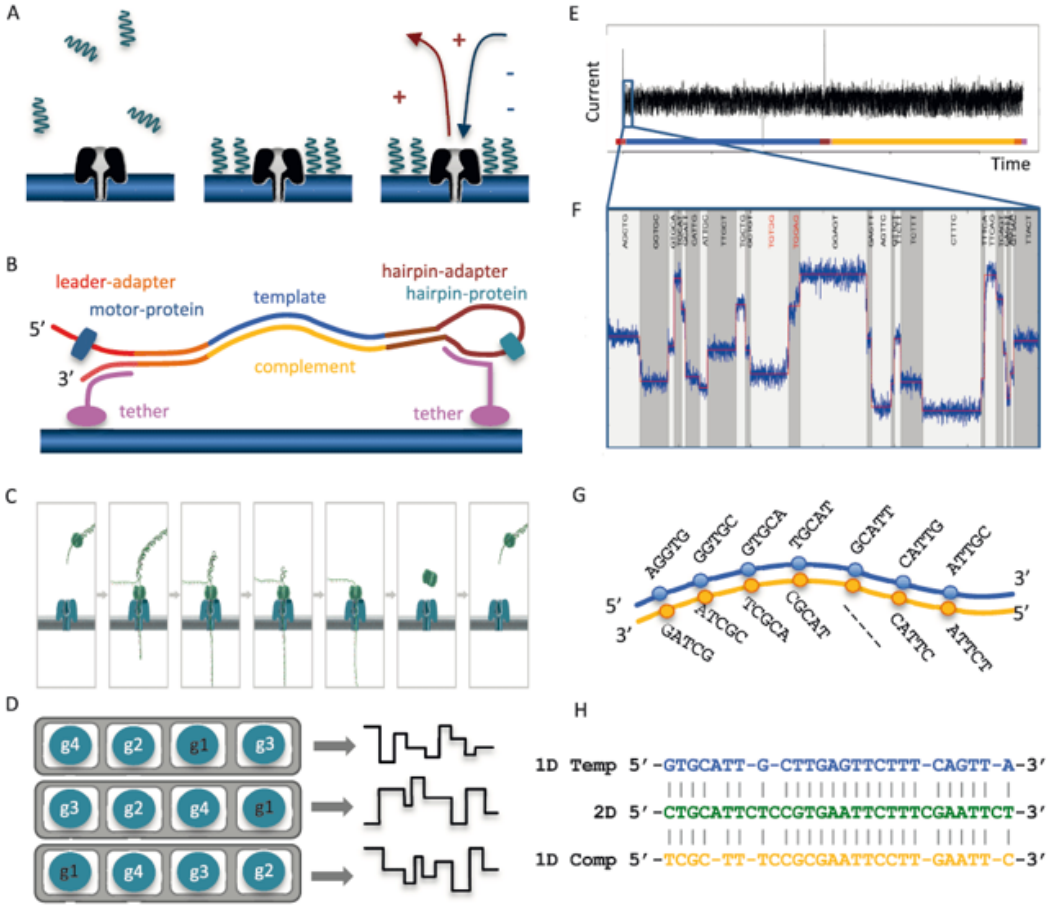

Figure A.23: Oxford Nanopore Sequencing [38]

The currently available machines provided by ONT are the MinION and the GridION (5 aggregated MinIONs plus a compute hub). The MinION is a very small form factor about the size of a modern day mobile phone and can connect directly to a processing machine (such as a laptop) via a USB 3A interface making it extremely portable and great for rapid field operations in emergent scenarios (such as disease outbreaks and remote forensics) [37]. ONT are working on two more machines:

- **PromethION** - A High-throughput, high-sample number and modular benchtop system with up to 48 MinION flow cells.
- **SmidgION** - An even more portable version of the MinION that connects to smartphones via a USB 3.1 Type C connector.

Since all their technology revolves around the MinION flow cell pipeline, data formats and files should be streamlined along with read sizes. The file format used by ONT is called FAST5. This an extension/implementation of the HDF5 [73] [103] format that is commonly used for scientific data storage. Read sizes average around 5000-5500bp in size, however the technology does not guarantee a fixed read size, just a range.

Appendix B

Detailed Evaluation results

This appendix contains full detailed evaluation results in tabular form.

B.1 Evaluation Tables

The following are the averages of the evaluation runs for individual chromosomes in human reference genome 37 with different storage representations.

B.1.1 Data Loading

Table B.1, Table B.2 and Table B.3 show the average data loading times, while Table B.4 and Table B.5 show the corresponding resulting data sizes.

Name	**Size** (bp)	**Load time** (ms)
ref_GRCh37_chr1	249250621	4195
ref_GRCh37_chr2	243199373	4109
ref_GRCh37_chr3	198022430	3605
ref_GRCh37_chr4	191154276	3227
ref_GRCh37_chr5	180915260	3462
ref_GRCh37_chr6	171115067	3085
ref_GRCh37_chr7	159138663	2833
ref_GRCh37_chr8	146364022	2582
ref_GRCh37_chr9	141213431	2689
ref_GRCh37_chr10	135534747	2132
ref_GRCh37_chr11	135006516	2560
ref_GRCh37_chr12	133851895	2350
ref_GRCh37_chr13	115169878	2081
ref_GRCh37_chr14	107349540	1894
ref_GRCh37_chr15	102531392	1609
ref_GRCh37_chr16	90354753	1696
ref_GRCh37_chr17	81195210	1208
ref_GRCh37_chr18	78077248	1180
ref_GRCh37_chr19	59128983	1398
ref_GRCh37_chr20	63025520	980
ref_GRCh37_chr21	48129895	784
ref_GRCh37_chr22	51304566	847
ref_GRCh37_chrX	155270560	2767
ref_GRCh37_chrY	59373566	1023

Table B.1: Data Loading time for individual chromosomes in human reference genome 37

Name	**Size** (bp)	**Load time** (ms)
ref_GRCh37_chr1	249250621	66611
ref_GRCh37_chr2	243199373	13200
ref_GRCh37_chr3	198022430	10029
ref_GRCh37_chr4	191154276	9583
ref_GRCh37_chr5	180915260	44720
ref_GRCh37_chr6	171115067	21856
ref_GRCh37_chr7	159138663	8666
ref_GRCh37_chr8	146364022	7563
ref_GRCh37_chr9	141213431	6705
ref_GRCh37_chr10	135534747	20548
ref_GRCh37_chr11	135006516	9761
ref_GRCh37_chr12	133851895	6212
ref_GRCh37_chr13	115169878	5788
ref_GRCh37_chr14	107349540	4569
ref_GRCh37_chr15	102531392	4646
ref_GRCh37_chr16	90354753	4308
ref_GRCh37_chr17	81195210	3251
ref_GRCh37_chr18	78077248	2000
ref_GRCh37_chr19	59128983	858
ref_GRCh37_chr20	63025520	1169
ref_GRCh37_chr21	48129895	492
ref_GRCh37_chr22	51304566	698
ref_GRCh37_chrX	155270560	45095
ref_GRCh37_chrY	59373566	4363

Table B.2: Data Loading time for individual chromosomes in human reference genome 37 using segmented 2bit representation

Name	**Size** (bp)	**Load time** (ms)
ref_GRCh37_chr1	249250621	81327
ref_GRCh37_chr2	243199373	19054
ref_GRCh37_chr3	198022430	15343
ref_GRCh37_chr4	191154276	64407
ref_GRCh37_chr5	180915260	40542
ref_GRCh37_chr6	171115067	25325
ref_GRCh37_chr7	159138663	11903
ref_GRCh37_chr8	146364022	12259
ref_GRCh37_chr9	141213431	40617
ref_GRCh37_chr10	135534747	23609
ref_GRCh37_chr11	135006516	26146
ref_GRCh37_chr12	133851895	9898
ref_GRCh37_chr13	115169878	8474
ref_GRCh37_chr14	107349540	7697
ref_GRCh37_chr15	102531392	7842
ref_GRCh37_chr16	90354753	6523
ref_GRCh37_chr17	81195210	4783
ref_GRCh37_chr18	78077248	3023
ref_GRCh37_chr19	59128983	1641
ref_GRCh37_chr20	63025520	1576
ref_GRCh37_chr21	48129895	1309
ref_GRCh37_chr22	51304566	1363
ref_GRCh37_chrX	155270560	57825
ref_GRCh37_chrY	59373566	10156

Table B.3: Data Loading time for individual chromosomes in human reference genome 37 using segmented 3bit representation

B.1.1.1 Data Sizes

Name	**Size** (bp)	**Byte Storage** (MB)	**2bit** (MB)	**3bit** (MB)
ref_GRCh37_chr1	249250621	1902	951	634
ref_GRCh37_chr2	243199373	1856	928	619
ref_GRCh37_chr3	198022430	1511	756	504
ref_GRCh37_chr4	191154276	1459	730	487
ref_GRCh37_chr5	180915260	1381	691	461
ref_GRCh37_chr6	171115067	1306	653	436
ref_GRCh37_chr7	159138663	1215	608	405
ref_GRCh37_chr8	146364022	1117	559	373
ref_GRCh37_chr9	141213431	1078	539	360
ref_GRCh37_chr10	135534747	1035	518	345
ref_GRCh37_chr11	135006516	1031	516	344
ref_GRCh37_chr12	133851895	1022	511	341
ref_GRCh37_chr13	115169878	879	440	293
ref_GRCh37_chr14	107349540	820	410	274
ref_GRCh37_chr15	102531392	783	392	261
ref_GRCh37_chr16	90354753	690	345	230
ref_GRCh37_chr17	81195210	620	310	207
ref_GRCh37_chr18	78077248	596	298	199
ref_GRCh37_chr19	59128983	452	226	151
ref_GRCh37_chr20	63025520	481	241	161
ref_GRCh37_chr21	48129895	368	184	123
ref_GRCh37_chr22	51304566	392	196	131
ref_GRCh37_chrX	155270560	1185	593	395
ref_GRCh37_chrY	59373566	453	227	151

Table B.4: Storage size for individual chromosomes in human reference genome 37 using standard and UDT representations

Name	**Size** (bp)	**2bitseg** (MB)	**3bitseg** (MB)
ref_GRCh37_chr1	249250621	56393870	94098230
ref_GRCh37_chr2	243199373	59770020	91813760
ref_GRCh37_chr3	198022430	48979730	74758420
ref_GRCh37_chr4	191154276	47138840	72165520
ref_GRCh37_chr5	180915260	44662420	68300060
ref_GRCh37_chr6	171115067	42055780	64600290
ref_GRCh37_chr7	159138663	38962660	60078910
ref_GRCh37_chr8	146364022	35907280	55256200
ref_GRCh37_chr9	141213431	29909850	53311700
ref_GRCh37_chr10	135534747	32876850	51167900
ref_GRCh37_chr11	135006516	32948860	50968470
ref_GRCh37_chr12	133851895	32747720	50532580
ref_GRCh37_chr13	115169878	24031650	43479690
ref_GRCh37_chr14	107349540	22222440	40527310
ref_GRCh37_chr15	102531392	20494800	38708360
ref_GRCh37_chr16	90354753	19831340	34111370
ref_GRCh37_chr17	81195210	19520730	30653450
ref_GRCh37_chr18	78077248	18742490	29476370
ref_GRCh37_chr19	59128983	13999030	22322930
ref_GRCh37_chr20	63025520	14931720	23793970
ref_GRCh37_chr21	48129895	8753425	18170530
ref_GRCh37_chr22	51304566	8738773	19369030
ref_GRCh37_chrX	155270560	37853410	58618600
ref_GRCh37_chrY	59373566	6567813	22811690

Table B.5: Storage size for individual chromosomes in human reference genome 37 using segmented UDT representations

B.1.2 Index Construction

B.1.2.1 FM Index

Construction times for the FM-Index under normal data generated with no biases or unknowns (N) is shown in Table B.6.

Size (bp)	**Construction time** (s)
1000	0.0281
10000	0.0026
100000	0.0189
1000000	0.1869
10000000	3.1299
50000000	18.4311
60000000	22.2883
70000000	26.4080
100000000	40.0925
110000000	44.2954
120000000	46.3385
128000000	49.5994
130000000	50.5462
131000000	53.1885
150000000	137.7605
200000000	175.1160
250000000	790.1582

Table B.6: FM Index construction time with random datasets

Creation times for the FM-Index using human refence genome 37.1 is shown in Table B.7.

Name	**Size** (bp)	**Construction time** (s)
ref_GRCh37_chr1	249250621	350.0772
ref_GRCh37_chr2	243199373	318.9625
ref_GRCh37_chr3	198022430	119.4313
ref_GRCh37_chr4	191154276	170.0530
ref_GRCh37_chr5	180915260	160.6767
ref_GRCh37_chr6	171115067	152.8675
ref_GRCh37_chr7	159138663	138.2284
ref_GRCh37_chr8	146364022	134.4116
ref_GRCh37_chr9	141213431	113.6374
ref_GRCh37_chr10	135534747	118.0544
ref_GRCh37_chr11	135006516	120.4866
ref_GRCh37_chr12	133851895	115.4634
ref_GRCh37_chr13	115169878	89.6325
ref_GRCh37_chr14	107349540	80.9491
ref_GRCh37_chr15	102531392	73.9622
ref_GRCh37_chr16	90354753	70.1566
ref_GRCh37_chr17	81195210	66.5518
ref_GRCh37_chr18	78077248	63.8510
ref_GRCh37_chr19	59128983	45.7507
ref_GRCh37_chr20	63025520	50.7422
ref_GRCh37_chr21	48129895	31.6656
ref_GRCh37_chr22	51304566	30.8258
ref_GRCh37_chrX	155270560	135.3549
ref_GRCh37_chrY	59373566	30.4971

Table B.7: FM Index construction time for individual genomes in human reference genome 37.1

B.1.2.2 Hash Index

Size (bp)	**Construction time** (s)
1000	0.0001
10000	0.0003
100000	0.0024
1000000	0.0349
1000000	0.0349
10000000	0.5452
50000000	2.9603
60000000	3.6896
70000000	3.9930
100000000	5.7374
110000000	6.5752
120000000	7.1318
128000000	7.7120
130000000	7.9236
131000000	7.7412
150000000	9.2153
250000000	15.1971

Table B.8: Hash Index construction time with random datasets

Name	**Size** (bp)	**Construction time** (s)
ref_GRCh37_chr1	249250621	13.3730
ref_GRCh37_chr2	243199373	13.8812
ref_GRCh37_chr3	198022430	11.1204
ref_GRCh37_chr4	191154276	10.5733
ref_GRCh37_chr5	180915260	9.9411
ref_GRCh37_chr6	171115067	9.6987
ref_GRCh37_chr7	159138663	9.0417
ref_GRCh37_chr8	146364022	8.6288
ref_GRCh37_chr9	141213431	7.6772
ref_GRCh37_chr10	135534747	7.6526
ref_GRCh37_chr11	135006516	7.4922
ref_GRCh37_chr12	133851895	7.7166
ref_GRCh37_chr13	115169878	6.0808
ref_GRCh37_chr14	107349540	5.2353
ref_GRCh37_chr15	102531392	5.0735
ref_GRCh37_chr16	90354753	5.0249
ref_GRCh37_chr17	81195210	4.6451
ref_GRCh37_chr18	78077248	4.5622
ref_GRCh37_chr19	59128983	3.4527
ref_GRCh37_chr20	63025520	3.7056
ref_GRCh37_chr21	48129895	1.9064
ref_GRCh37_chr22	51304566	2.4410
ref_GRCh37_chrX	155270560	8.6361
ref_GRCh37_chrY	59373566	1.6771

Table B.9: Hash Index construction time for individual genomes in human reference genome 37

B.1.2.3 Index storage sizes

Size (bp)	**Storage size** (bytes)
1000	25040
10000	250040
100000	2500040
1000000	25000040
10000000	250000040
50000000	1250000040
60000000	1500000040
70000000	1750000040
100000000	2500000040
110000000	2750000040
120000000	3000000040
128000000	3200000040
130000000	3250000040
131000000	3275000040
150000000	3750000040
200000000	5000000040
250000000	6250000040

Table B.10: Storage for FM-Index with random data sets

B.1.3 Dynamic Programming

Name	**Size** (bp)	**Storage size** (bytes)
ref_GRCh37_chr1	249250621	6231265565
ref_GRCh37_chr2	243199373	6079984365
ref_GRCh37_chr3	198022430	4950560790
ref_GRCh37_chr4	191154276	4778856940
ref_GRCh37_chr5	180915260	4522881540
ref_GRCh37_chr6	171115067	4277876715
ref_GRCh37_chr7	159138663	3978466615
ref_GRCh37_chr8	146364022	3659100590
ref_GRCh37_chr9	141213431	3530335815
ref_GRCh37_chr10	135534747	3388368715
ref_GRCh37_chr11	135006516	3375162940
ref_GRCh37_chr12	133851895	3346297415
ref_GRCh37_chr13	115169878	2879246990
ref_GRCh37_chr14	107349540	2683738540
ref_GRCh37_chr15	102531392	2563284840
ref_GRCh37_chr16	90354753	2258868865
ref_GRCh37_chr17	81195210	2029880290
ref_GRCh37_chr18	78077248	1951931240
ref_GRCh37_chr19	59128983	1478224615
ref_GRCh37_chr20	63025520	1575638040
ref_GRCh37_chr21	48129895	1203247415
ref_GRCh37_chr22	51304566	1282614190
ref_GRCh37_chrX	155270560	3881764040
ref_GRCh37_chrY	59373566	1484339190

Table B.11: Storage size for FM Index with individual genomes in human reference genome 37

Size (bp)	**Storage size** (bytes)
1000	1279
10000	13160
100000	135561
1000000	1395562
10000000	14355563
50000000	73555564
60000000	88355564
70000000	103155564
100000000	147555564
110000000	162755564
120000000	177955564
128000000	190115564
130000000	193155564
131000000	194675564
150000000	223555564
200000000	299555564
250000000	375555565

Table B.12: Storage size for Hash Index with random data sets

B.1.4 Pipeline Run

B.1.5 Comparison to Current Tools

B.1.5.1 Index Construction

B.1.5.2 Read Alignment

Table B.31 shows read alignment times for BWT-based indexers. Table B.32 shows read alignment times for hash-based indexers.

B.1.5.3 Consensus Sequence Generation and SNP Calling

Name	**Size** (bp)	**Storage size** (bytes)
ref_GRCh37_chr1	249250621	324711363
ref_GRCh37_chr2	243199373	346011726
ref_GRCh37_chr3	198022430	282143733
ref_GRCh37_chr4	191154276	272005372
ref_GRCh37_chr5	180915260	257069494
ref_GRCh37_chr6	171115067	242141681
ref_GRCh37_chr7	159138663	223578367
ref_GRCh37_chr8	146364022	206626571
ref_GRCh37_chr9	141213431	173098698
ref_GRCh37_chr10	135534747	189232275
ref_GRCh37_chr11	135006516	188894569
ref_GRCh37_chr12	133851895	187327919
ref_GRCh37_chr13	115169878	139013008
ref_GRCh37_chr14	107349540	127217337
ref_GRCh37_chr15	102531392	117355391
ref_GRCh37_chr16	90354753	112269654
ref_GRCh37_chr17	81195210	109835857
ref_GRCh37_chr18	78077248	107772315
ref_GRCh37_chr19	59128983	77658355
ref_GRCh37_chr20	63025520	85467619
ref_GRCh37_chr21	48129895	51130977
ref_GRCh37_chr22	51304566	50052842
ref_GRCh37_chrX	155270560	216069297
ref_GRCh37_chrY	59373566	36483989

Table B.13: Storage size for Hash Index with individual genomes in human reference genome 37

Name	**Size** (bp)	**Storage size** (bytes)	**Fragments**
ref_GRCh37_chr1	249250621	7726114452	5
ref_GRCh37_chr2	243199373	8274037398	5
ref_GRCh37_chr3	198022430	6774774118	4
ref_GRCh37_chr4	191154276	6518255458	4
ref_GRCh37_chr5	180915260	6148800144	4
ref_GRCh37_chr6	171115067	5811511049	3
ref_GRCh37_chr7	159138663	5334211118	3
ref_GRCh37_chr8	146364022	4956880768	3
ref_GRCh37_chr9	141213431	4071452786	3
ref_GRCh37_chr10	135534747	4506786577	3
ref_GRCh37_chr11	135006516	4536793986	3
ref_GRCh37_chr12	133851895	4493788986	3
ref_GRCh37_chr13	115169878	1438542986	1
ref_GRCh37_chr14	107349540	1427325054	1
ref_GRCh37_chr15	102531392	1331403111	1
ref_GRCh37_chr16	90354753	1612944298	1
ref_GRCh37_chr17	81195210	1890414420	1
ref_GRCh37_chr18	78077248	1984802942	1
ref_GRCh37_chr19	59128983	1799304969	1
ref_GRCh37_chr20	63025520	1961905966	1
ref_GRCh37_chr21	48129895	1238580009	1
ref_GRCh37_chr22	51304566	1183581664	1
ref_GRCh37_chrX	155270560	5136375627	3
ref_GRCh37_chrY	59373566	822304243	1

Table B.14: Storage size for Hash Index with robust error modelling and fragmentation at 60000000bp enabled

Name	**Size** (bp)	**Entries**	**Entries using Masking**	**Reduction**
ref_GRCh37_chr1	249250621	8535399	4479722	52.48%
ref_GRCh37_chr2	243199373	9106537	4958782	54.45%
ref_GRCh37_chr3	198022430	7447687	3920875	52.65%
ref_GRCh37_chr4	191154276	7187254	3725569	51.84%
ref_GRCh37_chr5	180915260	6796776	3584313	52.74%
ref_GRCh37_chr6	171115067	6410142	3329895	51.95%
ref_GRCh37_chr7	159138663	5919341	3142521	53.09%
ref_GRCh37_chr8	146364022	5492183	2875145	52.35%
ref_GRCh37_chr9	141213431	4594763	2414916	52.56%
ref_GRCh37_chr10	135534747	5035237	2740429	54.43%
ref_GRCh37_chr11	135006516	5025886	2601544	51.76%
ref_GRCh37_chr12	133851895	4979982	2560224	51.41%
ref_GRCh37_chr13	115169878	3714650	2024668	54.50%
ref_GRCh37_chr14	107349540	3397911	1786012	52.56%
ref_GRCh37_chr15	102531392	3137298	1679513	53.53%
ref_GRCh37_chr16	90354753	3010925	1604415	53.29%
ref_GRCh37_chr17	81195210	2938070	1599021	54.42%
ref_GRCh37_chr18	78077248	2904945	1601160	55.12%
ref_GRCh37_chr19	59128983	2071424	985241	47.56%
ref_GRCh37_chr20	63025520	2302942	1208076	52.46%
ref_GRCh37_chr21	48129895	1375429	750402	54.56%
ref_GRCh37_chr22	51304566	1338956	727888	54.36%
ref_GRCh37_chrX	155270560	5710538	2394448	41.93%
ref_GRCh37_chrY	59373566	988188	401566	40.64%

Table B.15: Comparison between entries in a hash index using a repeat masked reference sequence

Size (bp)	ed 0	ed 5	ed 10
35	605.6	678	580.8
50	1316.2	1216	1131.4
75	2892.6	2671.8	2582.6
100	4621	4648.6	4874.8
150	10736.2	10805.6	10392.4
200	18824.2	18527.2	18857.8
300	41141.6	41437.2	41468.6
500	113543	114351.6	113188
1000	447720.6	450967.2	452378
1500	1010722.2	1007057.4	1053102
2000	1789257.2	1869152.2	1824046.6
2500	2783786.6	2814603.6	2861773
3000	4034815.8	4134108.4	4040465.8
5000	11191762.2	11373476.2	11378558

Table B.16: Runtimes for running SW algorithm with linear gap penalties

Size (bp)	ed 0	ed 5	ed 10
35	202.4	141	133.6
50	247.2	238	262.6
75	402.4	412.6	396.2
100	682.2	684.8	685
150	1423.6	1296.8	1244.4
200	2076.8	2147.2	2093.4
300	4409.8	5157.6	5215.4
500	14792	14250.4	14353
1000	52660.6	49817.8	49860.4
1500	110102	112048.8	112957.2
2000	185510.2	179349.2	182695.6
2500	308050.2	307851	307417
3000	420564.6	419879.4	407331.2
5000	1131352.4	1112231.8	1132206.2

Table B.17: Runtimes for running Smith-Waterman algorithm with linear gap penalties using SIMD accelerated code

Size (bp)	ed 0	ed 5	ed 10
35	762.6	893	873.2
50	1665	1981.8	1697.4
75	3598.4	3819	3486.2
100	6228.8	7587.6	6778.2
150	13775.2	20191.4	17887.2
200	22734.6	27976.6	26201.2
300	51569.2	54525.4	57098.4
500	140972.2	164566.2	151273.2
1000	573196	600408.6	656190.4
1500	1255819.6	1281211	1414785.8
2000	2209770.8	2956804.6	2401326.2
2500	3533241.8	3797596	3823031.2
3000	5098129.6	5489876.2	5493088
5000	16221085.2	16232273.6	15372186.4

Table B.18: Runtimes for running Smith-Waterman algorithm with linear gap penalties inside SQL Server CLR

Size (bp)	ed 0	ed 5	ed 10
35	294	169.8	322
50	414.2	242	268.4
75	431.2	398.8	383.6
100	654.4	575.2	595.2
150	1142.4	950.2	1078
200	1889.8	1675.6	2013
300	10221.2	8961	7712.8
500	15345.2	21686	17198.2
1000	44999.2	41521.2	64469.6
1500	89420.6	91059.4	83718
2000	147050.4	154474.4	150111.2
2500	217245	225807.6	235809.4
3000	310610	319337	350907.6
5000	903959.8	892981.2	1053578.2

Table B.19: Runtimes for running Smith-Waterman algorithm with linear gap penalties using SIMD accelerated code inside SQL Server CLR

Item	Filename	Full Runtime
1	ref_GRCh37_chr1.masked	8072.348469
2	ref_GRCh37_chr2.masked	8825.918624
3	ref_GRCh37_chr3.masked	6807.033364
4	ref_GRCh37_chr4.masked	6361.141354
5	ref_GRCh37_chr5.masked	6179.842066
6	ref_GRCh37_chr6.masked	5303.972602
7	ref_GRCh37_chr7.masked	5188.622892
8	ref_GRCh37_chr8.masked	4671.72395
9	ref_GRCh37_chr9.masked	3874.194934
10	ref_GRCh37_chr10.masked	4851.132409
11	ref_GRCh37_chr11.masked	4145.180547
12	ref_GRCh37_chr12.masked	3993.249673
13	ref_GRCh37_chr13.masked	3166.530695
14	ref_GRCh37_chr14.masked	2738.217547
15	ref_GRCh37_chr15.masked	2828.841502
16	ref_GRCh37_chr16.masked	2651.281707
17	ref_GRCh37_chr17.masked	2527.521995
18	ref_GRCh37_chr18.masked	2462.031367
19	ref_GRCh37_chr19.masked	1616.62316
20	ref_GRCh37_chr20.masked	1797.310132
21	ref_GRCh37_chr21.masked	1135.206971
22	ref_GRCh37_chr22.masked	1162.871652
23	ref_GRCh37_chrX.masked	4151.082985
24	ref_GRCh37_chrY.masked	860.2709213

Table B.20: Full pipeline run - Complete runtimes

Item	Duplicate Counts	Duplicate N Counts	Length (bp)	Entries	Time
1	3755975	63293758	125000000	57950243	648.7201
	3822904	69300517	124250621	51127176	479.3161
2	3709742	61082172	125000000	60208062	636.1943
	2991096	53768161	118199373	61440092	621.8578
3	3058659	60943195	125000000	60998122	629.682
	1852943	35483476	73022430	35685987	257.3213
4	3176054	63188513	125000000	58635409	611.557
	1558079	31568191	66154276	33027982	203.4286
5	3529589	60992679	125000000	60477708	692.3503
	1494000	27165549	55915260	27255687	177.7868
6	2993878	64297603	125000000	57708495	598.6751
	1091779	20781886	46115067	24241378	115.7228
7	3827883	62019839	125000000	59152254	633.499
	895179	15771302	34138663	17472158	58.7238
8	3397952	62031627	125000000	59570397	640.407
	627213	9940228	21364022	10796557	31.4764
9	5297794	71518530	125000000	48183652	466.5265
	425292	7041136	16213431	8746979	20.245
10	4287140	60681787	125000000	60031049	709.2147
	255676	4002264	10534747	6276783	15.1559
11	3427668	63386514	125000000	58185794	688.2652
	207655	4229531	10006516	5569306	12.6033
12	3351997	63361458	125000000	58286521	716.6256
	262554	4053470	8851895	4535847	10.879
13	2355291	63057362	115169878	49757201	396.2556
14	2303080	61233648	107349540	43812788	335.7328
15	3327856	59054037	102531392	40149475	296.7531
16	3458521	48616174	90354753	38280034	296.0378
17	2905648	39712760	81195210	38576778	276.1112
18	1806274	36900582	78077248	39370368	255.0452
19	2184271	33360834	59128983	23583854	144.7406
20	1717064	31737817	63025520	29570615	180.1714
21	829075	28899743	48129895	18401053	64.0801
22	1375854	32490931	51304566	17437757	70.2836
23	3299287	75593105	125000000	46107584	522.2451
	761406	17274347	30270560	12234783	42.6716
24	1507546	48955588	59373566	8910408	40.8397

Table B.21: Full pipeline run - Index Construction

Item	Serialization Time
1	24.5114
	22.518
2	26.6919
	27.0372
3	28.1266
	16.1879
4	26.9161
	13.3118
5	29.5067
	12.8764
6	30.8238
	10.7886
7	27.2031
	7.7455
8	25.3035
	41.0645
9	21.0316
	3.4018
10	28.6611
	2.5844
11	25.17
	2.3479
12	27.7194
	1.8569
13	36.9378
14	18.8878
15	18.0642
16	16.335
17	17.2613
18	17.0746
19	10.057
20	12.5572
21	7.6755
22	7.5796
23	21.6103
	10.0701
24	3.7889

Table B.22: Full pipeline run - Index Serialization Time

Item	Time
1	100.9664
	84.8338
2	105.7265
	101.3395
3	103.313
	61.6767
4	98.1234
	56.0604
5	115.8159
	50.2624
6	102.1475
	40.2919
7	102.7272
	28.7368
8	98.8214
	18.7314
9	78.113
	15.3284
10	101.6782
	12.1524
11	97.2618
	7.905
12	99.899
	8.9042
13	85.8855
14	76.1562
15	67.6646
16	62.7598
17	64.0971
18	67.166
19	37.848
20	48.0418
21	29.1216
22	29.8702
23	81.875
	22.1465
24	14.8865

Table B.23: Full pipeline run - Index deserialization and Load Time

Item	Query file (bp)	**Total Process Time** (s)	**Aligment Time** (s)	**DB Access Time** (s)
1	2492506210	2681.0843	146.078	27.482
	2492506210	2165.6679	172.427	43.401
2	2431993730	2902.434	199.387	43.618
	2431993730	2488.0692	86.474	22.691
3	1980224300	2967.1445	100.281	23.892
	1980224300	1547.4729	48.601	14.657
4	1911542760	2870.8692	172.28	45.714
	1911542760	1558.3287	80.768	25.588
5	1809152600	3293.6649	321.464	66.285
	1809152600	1491.7095	56.516	16.086
6	1711150670	2793.7637	114.085	23.624
	1711150670	1329.896	65.886	20.607
7	1591386630	3022.7479	258.292	46.513
	1591386630	1028.3592	63.15	16.033
8	1463640220	2782.1274	209.011	55.142
	1463640220	602.1082	38.453	10.441
9	1412134310	2490.6583	375.19	30.747
	1412134310	532.2223	29.806	9.617
10	1355347470	3077.27	321.987	51.298
	1355347470	490.0699	42.044	13.539
11	1350065160	2557.1125	116.338	23.619
	1350065160	367.3149	14.106	5.003
12	1338518950	2542.0847	93.748	21.708
	1338518950	346.4075	34.855	12.819
13	1151698780	2471.8478	133.128	40.638
14	1073495400	2160.1676	99.405	25.541
15	1025313920	2293.8109	334.022	65.23
16	903547530	2134.7485	337.734	79.948
17	811952100	2030.7591	217.92	48.13
18	780772480	1993.152	97.446	35.964
19	591289830	1328.1727	195.651	56.458
20	630255200	1457.1898	95.949	40.472
21	481298950	935.1314	64.998	26.878
22	513045660	982.6206	139.684	39.793
23	1552705600	2331.0406	258.466	71.135
	1552705600	710.2595	36.017	4.875
24	593735660	702.48	289.607	69.141

Table B.24: Full pipeline run - Query Processing

Item	**Read Load Time** (s)	**Total Cons Gen Time** (s)
1	216.5596	1439.9205
2	362.4874	378.9546
3	417.8194	726.493
4	248.0286	613.2848
5	143.6189	133.2861
6	131.7226	118.0712
7	133.5593	117.452
8	303.9629	104.6158
9	127.2792	94.0986
10	295.1841	98.7137
11	272.5593	93.3805
12	124.0059	93.7539
13	70.2366	84.4396
14	59.2068	71.8696
15	62.3666	72.8233
16	60.8795	68.0759
17	56.065	68.686
18	55.2738	61.3807
19	37.6856	48.7071
20	42.4028	47.0788
21	56.5231	35.0605
22	27.2625	37.8937
23	280.1717	99.0232
24	61.7124	27.5018

Table B.25: Full pipeline run - Consensus Sequence Construction

Item	**SNP Insertion Time** (s)	**SNP Count**
1	1091.659	Failed
2	122.107	620385
3	653.918	Failed
4	542.65	Failed
5	75.927	440169
6	64.548	407087
7	66.27	402082
8	59.151	354338
9	51.422	312544
10	55.48	346026
11	53.532	321175
12	51.766	313471
13	49.368	242611
14	39.663	21565
15	41.681	21989
16	40.693	213287
17	43.447	207807
18	36.88	192289
19	31.12	145681
20	28.182	147594
21	20.935	92863
22	22.937	96833
23	50.822	300315
24	11.567	67995

Table B.26: Full pipeline run - SNP Insertion. Some SNP insertion trials failed - These corresponded to Chromosomes 1,3 and 4

Chromosome	**Length** (bp)	**bwa**	**bowtie2**
ref_GRCh37_chr1	249250621	221.049	282.442
ref_GRCh37_chr2	243199373	227.844	295.269
ref_GRCh37_chr3	198022430	184.226	225.549
ref_GRCh37_chr4	191154276	178.08	220.231
ref_GRCh37_chr5	180915260	166.569	206.595
ref_GRCh37_chr6	171115067	157.574	190.219
ref_GRCh37_chr7	159138663	142.869	175.919
ref_GRCh37_chr8	146364022	133.186	158.539
ref_GRCh37_chr9	141213431	119.003	136.891
ref_GRCh37_chr10	135534747	129.469	108.696
ref_GRCh37_chr11	135006516	126.277	147.732
ref_GRCh37_chr12	133851895	121.788	140.168
ref_GRCh37_chr13	115169878	100.339	140.989
ref_GRCh37_chr14	107349540	89.003	95.713
ref_GRCh37_chr15	102531392	86.875	87.136
ref_GRCh37_chr16	90354753	75.957	82.675
ref_GRCh37_chr17	81195210	69.042	78.061
ref_GRCh37_chr18	78077248	66.732	77.276
ref_GRCh37_chr19	59128983	47.155	69.875
ref_GRCh37_chr20	63025520	53.48	53.143
ref_GRCh37_chr21	48129895	37.535	27.491
ref_GRCh37_chr22	51304566	39.108	28.739
ref_GRCh37_chrX	155270560	139.408	173.891
ref_GRCh37_chrY	59373566	42.821	25.037

Table B.27: Index construction times for BWT-based indexers in seconds

Chromosome	bwa	bowtie2
ref_GRCh37_chr1	273876887	327537973
ref_GRCh37_chr2	364799741	345846340
ref_GRCh37_chr3	297034055	284352063
ref_GRCh37_chr4	286731891	274243577
ref_GRCh37_chr5	271373269	260124374
ref_GRCh37_chr6	256673053	245532548
ref_GRCh37_chr7	238708563	228474043
ref_GRCh37_chr8	219546452	210815466
ref_GRCh37_chr9	211821109	178593724
ref_GRCh37_chr10	203302802	194419138
ref_GRCh37_chr11	202510195	194156294
ref_GRCh37_chr12	200778316	193238214
ref_GRCh37_chr13	172755208	143808288
ref_GRCh37_chr14	161024620	133466085
ref_GRCh37_chr15	153797549	124123786
ref_GRCh37_chr16	135532524	120142805
ref_GRCh37_chr17	121793224	118599335
ref_GRCh37_chr18	117116416	114154186
ref_GRCh37_chr19	88693849	87452038
ref_GRCh37_chr20	94538666	92688863
ref_GRCh37_chr21	72195383	58124176
ref_GRCh37_chr22	76957434	57823864
ref_GRCh37_chrX	232906507	222449093
ref_GRCh37_chrY	89060899	44732314

Table B.28: Index sizes in bytes for BWT-based indexers in bytes

Chromosome	snap	wham	soap
ref_GRCh37_chr1	42.172	121.471286	419.5
ref_GRCh37_chr2	43.798	130.916656	510.441
ref_GRCh37_chr3	35.048	101.371298	420.453
ref_GRCh37_chr4	33.627	95.469313	403.061
ref_GRCh37_chr5	31.291	84.371855	334.953
ref_GRCh37_chr6	30.04	81.273429	284.813
ref_GRCh37_chr7	28.643	79.801955	298.4
ref_GRCh37_chr8	24.936	70.653408	274.026
ref_GRCh37_chr9	22.797	56.516008	265.213
ref_GRCh37_chr10	22.988	61.933031	265.751
ref_GRCh37_chr11	22.884	60.807058	288.952
ref_GRCh37_chr12	22.786	60.168931	288.528
ref_GRCh37_chr13	17.51	42.95923	197.912
ref_GRCh37_chr14	16.415	39.767827	133.777
ref_GRCh37_chr15	15.818	35.96809	162.049
ref_GRCh37_chr16	14.92	34.504993	155.667
ref_GRCh37_chr17	14.111	35.04972	141.556
ref_GRCh37_chr18	12.744	32.582517	122.655
ref_GRCh37_chr19	10.339	24.25176	98.748
ref_GRCh37_chr20	10.055	26.140923	78.644
ref_GRCh37_chr21	6.686	15.074281	120.808
ref_GRCh37_chr22	7.143	15.613653	74.222
ref_GRCh37_chrX	28.022	76.275682	273.605
ref_GRCh37_chrY	7.168	10.657888	57.168

Table B.29: Index construction times for hash-based indexers in seconds

Chromosome	snap	wham	soap
ref_GRCh37_chr1	2437345573	1591751308	1088915929
ref_GRCh37_chr2	2575630805	1680416075	1099799113
ref_GRCh37_chr3	2106012642	1373321778	1021021090
ref_GRCh37_chr4	2032437580	1323302232	1007661494
ref_GRCh37_chr5	1919147467	1253857253	995571652
ref_GRCh37_chr6	1805684339	1180517660	971167354
ref_GRCh37_chr7	1672306587	1097251016	954986751
ref_GRCh37_chr8	1551008466	1007819294	925208244
ref_GRCh37_chr9	1311712539	853069822	896242552
ref_GRCh37_chr10	1418656132	927717506	910088282
ref_GRCh37_chr11	1420659768	924882234	905534897
ref_GRCh37_chr12	1402346876	920267736	906560101
ref_GRCh37_chr13	1067440542	673490176	836542957
ref_GRCh37_chr14	973468071	622544320	827625085
ref_GRCh37_chr15	902172127	578154047	820724383
ref_GRCh37_chr16	854124551	558314751	817815682
ref_GRCh37_chr17	824756781	550278473	816485697
ref_GRCh37_chr18	820584457	525939111	799681022
ref_GRCh37_chr19	576611485	395127869	780935449
ref_GRCh37_chr20	649169882	419434039	775112972
ref_GRCh37_chr21	399926092	247399303	730697927
ref_GRCh37_chr22	390957112	246793942	734688946
ref_GRCh37_chrX	1609791104	1067221364	953925935
ref_GRCh37_chrY	309514366	183347196	721070205

Table B.30: Index sizes in bytes for hash-based indexers

Chromosome	Reads	bwa	bowtie2
ref_GRCh37_chr1	8718191	4852.422	2220.186
ref_GRCh37_chr2	9892132	4782.938	2136.993
ref_GRCh37_chr3	7700698	3893.625	2260.485
ref_GRCh37_chr4	7384329	3789.797	1339.066
ref_GRCh37_chr5	7109521	3528.5	1492.708
ref_GRCh37_chr6	6620582	3101.875	1457.669
ref_GRCh37_chr7	6230817	2992.859	1395.416
ref_GRCh37_chr8	5645196	2650.156	1463.031
ref_GRCh37_chr9	4723051	2246.547	952.914
ref_GRCh37_chr10	5423994	2264.828	1288.02
ref_GRCh37_chr11	5061272	2435.484	3984.678
ref_GRCh37_chr12	4943898	2441.516	838.006
ref_GRCh37_chr13	4060713	1518.641	423.956
ref_GRCh37_chr14	3513534	1495.703	543.404
ref_GRCh37_chr15	3300523	1387.609	3841.725
ref_GRCh37_chr16	3053462	1385.641	3087.407
ref_GRCh37_chr17	3083233	1467.984	2856.603
ref_GRCh37_chr18	3198275	1231.297	2745.554
ref_GRCh37_chr19	1845211	1345.703	2593.421
ref_GRCh37_chr20	2292649	894	2547.281
ref_GRCh37_chr21	1504460	473.688	2102.534
ref_GRCh37_chr22	1401384	565.344	1832.017
ref_GRCh37_chrX	4611290	3297.375	2626.185
ref_GRCh37_chrY	798191	443.828	347.234

Table B.31: Read alignment times using BWT-based indexers in seconds

Chromosome	snap	wham	soap
ref_GRCh37_chr1	25.91	69.126057	415.231
ref_GRCh37_chr2	28.933	82.955821	664.046
ref_GRCh37_chr3	22.25	62.576641	411.813
ref_GRCh37_chr4	21.27	57.85265	522.268
ref_GRCh37_chr5	21.484	54.577599	353.824
ref_GRCh37_chr6	19.079	50.002138	272.064
ref_GRCh37_chr7	18.603	56.115942	301.243
ref_GRCh37_chr8	16.059	48.712523	268.608
ref_GRCh37_chr9	13.765	39.397117	267.113
ref_GRCh37_chr10	15.506	43.997775	273.847
ref_GRCh37_chr11	14.736	40.011715	297.393
ref_GRCh37_chr12	14.271	37.674238	294.412
ref_GRCh37_chr13	11.753	31.059752	229.452
ref_GRCh37_chr14	10.559	31.026527	119.06
ref_GRCh37_chr15	9.973	25.208641	155.296
ref_GRCh37_chr16	9.323	23.140813	141.235
ref_GRCh37_chr17	8.947	23.612169	128.217
ref_GRCh37_chr18	9.358	23.652869	135.857
ref_GRCh37_chr19	6.021	13.611906	72.325
ref_GRCh37_chr20	6.908	16.936192	71.61
ref_GRCh37_chr21	4.412	11.812856	161.011
ref_GRCh37_chr22	4.55	10.233691	67.019
ref_GRCh37_chrX	14.206	34.977838	210.38
ref_GRCh37_chrY	2.758	6.173763	37.466

Table B.32: Read alignment times using hash-based indexers in seconds

Chromosome	bwa	bowtie2
ref_GRCh37_chr1	5935876976	8705232721
ref_GRCh37_chr2	6250164711	9204846512
ref_GRCh37_chr3	5074372029	7521416206
ref_GRCh37_chr4	4894296800	7246404812
ref_GRCh37_chr5	4630275088	6843976866
ref_GRCh37_chr6	4360943588	6461120526
ref_GRCh37_chr7	4090186693	5999720512
ref_GRCh37_chr8	3723847578	5511583341
ref_GRCh37_chr9	3291791920	4641502374
ref_GRCh37_chr10	5065796722	5065796729
ref_GRCh37_chr11	3409604728	5046051674
ref_GRCh37_chr12	3391752466	5031684825
ref_GRCh37_chr13	2481497566	3685154881
ref_GRCh37_chr14	2285911636	3394532282
ref_GRCh37_chr15	2155769526	3144239357
ref_GRCh37_chr16	2068637535	3023622254
ref_GRCh37_chr17	2037016035	2989957600
ref_GRCh37_chr18	1914433347	2857514086
ref_GRCh37_chr19	1446624652	2139756802
ref_GRCh37_chr20	1532888026	2283180593
ref_GRCh37_chr21	902400052	1344908934
ref_GRCh37_chr22	913420614	1341284653
ref_GRCh37_chrX	3970376697	5836466762
ref_GRCh37_chrY	718464398	987055732

Table B.33: Read alignment file sizes using BWT-based indexers in bytes

Chromosome	snap	wham	soap
ref_GRCh37_chr1	500027498	2897733654	2319931780
ref_GRCh37_chr2	566915739	3288748480	2636298537
ref_GRCh37_chr3	439846894	2558672139	2049724961
ref_GRCh37_chr4	421198497	2453351400	1964956582
ref_GRCh37_chr5	405694177	2355242357	1886493370
ref_GRCh37_chr6	377097442	2199334586	1762347623
ref_GRCh37_chr7	354979605	2068799443	1657921785
ref_GRCh37_chr8	321022142	1873485250	1500506142
ref_GRCh37_chr9	268728043	1567838604	1255486239
ref_GRCh37_chr10	308557201	1799680092	1441883757
ref_GRCh37_chr11	287870322	1674991844	1340059193
ref_GRCh37_chr12	280755430	1639984039	1312290733
ref_GRCh37_chr13	228446108	1347266419	1079947785
ref_GRCh37_chr14	197283734	1162205514	930390358
ref_GRCh37_chr15	185290526	1091432894	873579306
ref_GRCh37_chr16	171763425	1005611139	803695594
ref_GRCh37_chr17	173144839	1018227594	815203834
ref_GRCh37_chr18	179138256	1053415742	843857418
ref_GRCh37_chr19	103110110	607477710	485863253
ref_GRCh37_chr20	128190894	756782340	605346334
ref_GRCh37_chr21	83020727	495925791	397707975
ref_GRCh37_chr22	77468282	463070486	370767696
ref_GRCh37_chrX	262487963	1530668677	1224630401
ref_GRCh37_chrY	44288533	262571669	210346639

Table B.34: Read alignment file sizes using hash-based indexers in bytes

Chromosome	Total Reads	snap	wham	soap
ref_GRCh37_chr1	8718191	8718116	7186115	6896881
ref_GRCh37_chr2	9892132	9892070	8157047	7835586
ref_GRCh37_chr3	7700698	7700644	6352384	6096088
ref_GRCh37_chr4	7384329	7384273	6087934	5844191
ref_GRCh37_chr5	7109521	7109471	5862965	5630328
ref_GRCh37_chr6	6620582	6620530	5458981	5242285
ref_GRCh37_chr7	6230817	6230783	5136435	4934606
ref_GRCh37_chr8	5645196	5645155	4655138	4468344
ref_GRCh37_chr9	4723051	4723010	3894681	3738280
ref_GRCh37_chr10	5423994	5423959	4471421	4294682
ref_GRCh37_chr11	5061272	5061214	4170411	4003485
ref_GRCh37_chr12	4943898	4943865	4075259	3909497
ref_GRCh37_chr13	4060713	4060683	3348957	3216870
ref_GRCh37_chr14	3513534	3513511	2896500	2781198
ref_GRCh37_chr15	3300523	3300496	2719850	2612279
ref_GRCh37_chr16	3053462	3053429	2515773	2413290
ref_GRCh37_chr17	3083233	3083203	2541372	2439880
ref_GRCh37_chr18	3198275	3198256	2637330	2533538
ref_GRCh37_chr19	1845211	1845191	1520026	1459418
ref_GRCh37_chr20	2292649	2292626	1888399	1812305
ref_GRCh37_chr21	1504460	1504445	1241671	1193202
ref_GRCh37_chr22	1401384	1401373	1154487	1108956
ref_GRCh37_chrX	4611290	4611246	3799786	3645551
ref_GRCh37_chrY	798191	798187	658535	632678

Table B.35: Reads aligned by hash-index based aligners

Chromosome	sam to bam (s)	sort bam (s)	mpileup (s)
ref_GRCh37_chr1	360.863	247.728	8666.54
ref_GRCh37_chr2	383.248	261.677	9132.578
ref_GRCh37_chr3	310.182	216.063	7304.653
ref_GRCh37_chr4	299.51	204.648	6951.302
ref_GRCh37_chr5	273.179	187.73	6396.959
ref_GRCh37_chr6	263.542	175.43	6121.311
ref_GRCh37_chr7	247.301	160.922	5976.649
ref_GRCh37_chr8	230.887	147.402	6037.191
ref_GRCh37_chr9	198.932	124.589	4531.151
ref_GRCh37_chr10	206.19	141.713	5089.944
ref_GRCh37_chr11	207.49	143.824	4718.183
ref_GRCh37_chr12	206.876	143.238	4828.182
ref_GRCh37_chr13	150.965	105.147	3309.319
ref_GRCh37_chr14	142.646	96.573	3037.882
ref_GRCh37_chr15	129.703	89.834	2870.24
ref_GRCh37_chr16	126.868	86.58	2755.954
ref_GRCh37_chr17	125.086	87.215	2690.385
ref_GRCh37_chr18	117.43	81.572	2581.387
ref_GRCh37_chr19	89.854	60.51	1991.752
ref_GRCh37_chr20	94.86	64.488	2223.006
ref_GRCh37_chr21	55.257	38.639	1329.123
ref_GRCh37_chr22	56.363	39.781	1328.338
ref_GRCh37_chrX	250.726	158.079	5365.012
ref_GRCh37_chrY	40.956	28.616	907.399

Table B.36: Time taken to generate consensus using samtools with mpileup and bcftools pipeline

Chromosome	bcftools (s)	vcfutils (s)
ref_GRCh37_chr1	1424.087	1154.113
ref_GRCh37_chr2	1440.359	1205.537
ref_GRCh37_chr3	1185.745	1001.433
ref_GRCh37_chr4	1122.031	949.566
ref_GRCh37_chr5	1064.003	896.777
ref_GRCh37_chr6	1026.908	850.969
ref_GRCh37_chr7	932.382	791.508
ref_GRCh37_chr8	843.246	731.471
ref_GRCh37_chr9	724.985	603.695
ref_GRCh37_chr10	781.977	672.045
ref_GRCh37_chr11	794.803	658.741
ref_GRCh37_chr12	791.766	657.321
ref_GRCh37_chr13	575.124	492.577
ref_GRCh37_chr14	524.627	450.652
ref_GRCh37_chr15	483.99	409.326
ref_GRCh37_chr16	504.052	421.346
ref_GRCh37_chr17	486.872	404.206
ref_GRCh37_chr18	470.349	399.77
ref_GRCh37_chr19	346.33	295.509
ref_GRCh37_chr20	373.537	312.511
ref_GRCh37_chr21	219.576	199.823
ref_GRCh37_chr22	221.376	186.911
ref_GRCh37_chrX	988.106	786.253
ref_GRCh37_chrY	167.632	131.382

Table B.37: Time taken to generate consensus using samtools with bcftools and vcfutils using the mpileup output

Pipeline component	Output
mpileup (1)	vcf (compressed)
mpileup (2)	vcf (uncompressed)
bcftools	vcf
vcfutils	fastq

Table B.38: Pipeline stages and expected output filesizes (bytes) for each stage in Consensus Sequence Generation and variant calling

Chromosome	mpileup1	mpileup2	bcftools	vcfutils
ref_GRCh37_chr1	3859848714	32693427677	29367709797	506789298
ref_GRCh37_chr2	4077650528	34585234491	31058399218	494485094
ref_GRCh37_chr3	3334411072	28245738676	25346705770	402523642
ref_GRCh37_chr4	3213131069	27206072063	24417526623	388456730
ref_GRCh37_chr5	3041287169	25584303197	22953906983	367840729
ref_GRCh37_chr6	2866228033	24255439560	21766153695	347811992
ref_GRCh37_chr7	2666741111	22520079597	20230654181	323561650
ref_GRCh37_chr8	2442734025	20688385548	18568009958	297484860
ref_GRCh37_chr9	2058373858	17428334951	15689816379	287012010
ref_GRCh37_chr10	2245910892	18999905477	17060721532	275567014
ref_GRCh37_chr11	2246341472	18847767039	16902432855	274391283
ref_GRCh37_chr12	2235659744	18883015619	16941065932	272145222
ref_GRCh37_chr13	1633470088	13827646052	12403806479	234056786
ref_GRCh37_chr14	1510711070	12679526337	11367278462	218155431
ref_GRCh37_chr15	1398084082	11732033653	10535126104	208460185
ref_GRCh37_chr16	1350598378	11319406999	10163708442	183599337
ref_GRCh37_chr17	1336184267	11234931872	10087571944	165096944
ref_GRCh37_chr18	1273337234	10699460029	9586472477	158635105
ref_GRCh37_chr19	963762815	7997248704	7170415471	120208627
ref_GRCh37_chr20	1015943917	8586434383	7699826082	128029878
ref_GRCh37_chr21	599687754	5034054376	4510826805	97843819
ref_GRCh37_chr22	598039672	5043924645	4529016481	104197320
ref_GRCh37_chrX	2596788860	21908798198	19683602596	315696486
ref_GRCh37_chrY	440111696	3705804951	3360373774	120705949

Table B.39: File sizes of all files involved in samtools consensus sequence pipeline in bytes

Chromosome	generate variants
ref_GRCh37_chr1	612.652
ref_GRCh37_chr2	579.363
ref_GRCh37_chr3	674.704
ref_GRCh37_chr4	657.87
ref_GRCh37_chr5	413.737
ref_GRCh37_chr6	312.237
ref_GRCh37_chr7	314.852
ref_GRCh37_chr8	344.545
ref_GRCh37_chr9	266.5
ref_GRCh37_chr10	490.877
ref_GRCh37_chr11	467.622
ref_GRCh37_chr12	455.565
ref_GRCh37_chr13	256.708
ref_GRCh37_chr14	259.16
ref_GRCh37_chr15	246.508
ref_GRCh37_chr16	276.558
ref_GRCh37_chr17	172.908
ref_GRCh37_chr18	196.749
ref_GRCh37_chr19	183.191
ref_GRCh37_chr20	195.867
ref_GRCh37_chr21	138.788
ref_GRCh37_chr22	92.294
ref_GRCh37_chrX	300.559
ref_GRCh37_chrY	84.592

Table B.40: Time taken to generate variants using samtools

Chromosome	sort	soapsnp
ref_GRCh37_chr1	29.56	10791.113
ref_GRCh37_chr2	66.688	15299.767
ref_GRCh37_chr3	37.635	12556.709
ref_GRCh37_chr4	33.413	10398.87
ref_GRCh37_chr5	51.359	10133.425
ref_GRCh37_chr6	22.294	8204.728
ref_GRCh37_chr7	44.287	8944.357
ref_GRCh37_chr8	35.039	8279.185
ref_GRCh37_chr9	18.879	7423.659
ref_GRCh37_chr10	20.194	7083.701
ref_GRCh37_chr11	34.028	6610.139
ref_GRCh37_chr12	32.856	6605.182
ref_GRCh37_chr13	17.011	5111.468
ref_GRCh37_chr14	10.678	3422.584
ref_GRCh37_chr15	20.695	4549.153
ref_GRCh37_chr16	19.741	4213.019
ref_GRCh37_chr17	11.732	4812.752
ref_GRCh37_chr18	11.925	3268.314
ref_GRCh37_chr19	10.961	2400.003
ref_GRCh37_chr20	6.471	2245.463
ref_GRCh37_chr21	13.013	4056.657
ref_GRCh37_chr22	6.804	3367.314
ref_GRCh37_chrX	26.815	6206.054
ref_GRCh37_chrY	3.309	3249.514

Table B.41: Time taken to generate variants using soapsnp

Chromosome	variants file size
ref_GRCh37_chr1	44579137
ref_GRCh37_chr2	47661629
ref_GRCh37_chr3	39674910
ref_GRCh37_chr4	37902009
ref_GRCh37_chr5	35241796
ref_GRCh37_chr6	33890530
ref_GRCh37_chr7	30379336
ref_GRCh37_chr8	28848397
ref_GRCh37_chr9	22203783
ref_GRCh37_chr10	26018479
ref_GRCh37_chr11	26081816
ref_GRCh37_chr12	26325276
ref_GRCh37_chr13	19465618
ref_GRCh37_chr14	17643050
ref_GRCh37_chr15	15606193
ref_GRCh37_chr16	15020417
ref_GRCh37_chr17	15131067
ref_GRCh37_chr18	15173541
ref_GRCh37_chr19	10920654
ref_GRCh37_chr20	12120352
ref_GRCh37_chr21	7121979
ref_GRCh37_chr22	6831071
ref_GRCh37_chrX	29470827
ref_GRCh37_chrY	3601307

Table B.42: Size of variant call files for variant calling using bcftools

Chromosome	soapsnp output
ref_GRCh37_chr1	20791228012
ref_GRCh37_chr2	20362790105
ref_GRCh37_chr3	16554148706
ref_GRCh37_chr4	15972946711
ref_GRCh37_chr5	14932658095
ref_GRCh37_chr6	14285797066
ref_GRCh37_chr7	13277373408
ref_GRCh37_chr8	12201501940
ref_GRCh37_chr9	11703955773
ref_GRCh37_chr10	11292438723
ref_GRCh37_chr11	11106162168
ref_GRCh37_chr12	11143399453
ref_GRCh37_chr13	9532340747
ref_GRCh37_chr14	8760303731
ref_GRCh37_chr15	8351822085
ref_GRCh37_chr16	7375489498
ref_GRCh37_chr17	6732982334
ref_GRCh37_chr18	6403717268
ref_GRCh37_chr19	4828055492
ref_GRCh37_chr20	5220348532
ref_GRCh37_chr21	3907175195
ref_GRCh37_chr22	4204966718
ref_GRCh37_chrX	12906286063
ref_GRCh37_chrY	4748614860

Table B.43: Size of variant call files for variant calling using soapsnp

Bibliography

[1] P. G. AFA Smith, R Hubley. Repeatmasker open-4.0., 2015. URL http://www.repeatmasker.org/. 3.5.1.4

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, oct 1990. doi: 10.1016/s0022-2836(05)80360-2. 2.2

[3] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25(17):3389–3402, Sep 1997. 2

[4] R. Baeza-Yates and C. Perleberg. Fast and practical approximate string matching. *Lecture Notes in Computer Science*, 644:185–192, 1992. 3

[5] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. GenBank. *Nucleic Acids Res.*, 39(Database issue):D32–37, Jan 2011. 2.5.2

[6] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. GenBank. *Nucleic Acids Research*, 41(D1): D36–D42, nov 2012. doi: 10.1093/nar/gks1195. 1.1.1, 3.7.1

[7] N. T. S. Bhd. Novoalign, 2014. URL http://www.novocraft.com/products/novoalign/. 2.8.1.2

[8] J. Bonfield. Staden package, 2016. URL http://staden.sourceforge.net/. 5.2.2

[9] J. K. Bonfield. Sequence read format (srf), 2008. URL http://srf.sourceforge.net/. 2.5.1.3

[10] J. K. Bonfield and R. Staden. ZTR: a new format for DNA sequence trace data. *Bioinformatics*, 18(1):3–10, Jan 2002. 2.5.1.4

[11] S. Brunak. Nucleotide sequence database policies. *Science*, 298(5597): 1333b–1333, nov 2002. doi: 10.1126/science.298.5597.1333b. 2.5.2

[12] D. Caetano-Anolles. Introducing dragmap, the new genome mapper in dragen-gatk, 2021. URL https://gatk.broadinstitute.org/hc/en-us/articles/4410953761563. 2.4.2.2

[13] F. Chang, G. L. Liu, C. J. Liu, and M. M. Li. Somatic diseases (cancer). In *Clinical Genomics*, pages 297–319. Elsevier, 2015. doi: 10.1016/b978-0-12-404748-8.00018-6. (document), 2.7, A, A.22

[14] CloudFlare. CloudFlare Radar - Global Internet Traffic Monitor, 2024. https://radar.cloudflare.com/. 3.7.1.2

[15] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The sanger FASTQ file format for sequences with quality scores, and the solexa/illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, dec 2009. doi: 10.1093/nar/gkp1137. 2.5.1.2

[16] N. R. Coordinators. Database resources of the national center for biotechnology information. *Nucleic Acids Research*, 45(D1):D12–D17, nov 2016. doi: 10.1093/nar/gkw1071. 1.1.1

[17] M. Corporation. .net framework 4.7, 4.6, and 4.5, 2017. URL https://msdn.microsoft.com/en-us/library/w0x726c2(v=vs.110).aspx. 2.8.1.1

[18] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. Dewitt, B. Heath, D. Maier, S. Madden, M. Stonebraker, and S. Zdonik. A demonstration of scidb: A science-oriented dbms. In *VLDB'09: Proceedings of the 2009 VLDB Endowment*. VLDB Endowment, August 2009. URL http://database.cs.brown.edu/papers/vldb09/scidb.pdf. 1.1.1

[19] C. Dekker. Solid-state nanopores. *Nature Nanotechnology*, 2(4):209–215, 2007. URL http://www.nature.com/nnano/journal/v2/n4/abs/nnano.2007.27.html. A.5

[20] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, D. Altshuler, and M. J. Daly. A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature Genetics*, 43(5):491–498, 2011. doi: 10.1038/ng.806. URL https://doi.org/10.1038/ng.806. 2.4

[21] P. Deutsch. DEFLATE compressed data format specification version 1.3. Technical report, Aladdin Enterprises, may 1996. 4.1.3.5

[22] ECO. Tech summary: Illumina's solexa sequencing technology, April 2007. URL http://seqanswers.com/forums/showpost.php?p=32&postcount=1. (document), A, A.1, A.1, A.2, A.3

[23] ECO. Tech summary: Abi's solid (seq. by oligo ligation/detection) v2.0, Dec 2007. URL http://seqanswers.com/forums/showpost.php?p=11&postcount=1. (document), A, A.2, A.4, A.5, A.6, A.7, A.8, A.9, A.10, A.11

[24] A. Eisenberg. New standard for stored procedures in sql. *SIGMOD Rec.*, 25 (4):81–88, Dec. 1996. ISSN 0163-5808. doi: 10.1145/245882.245907. URL http://doi.acm.org/10.1145/245882.245907. 2.2

[25] E. S. L. et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, feb 2001. doi: 10.1038/35057062. 1.1.3.1

[26] R. M. D. et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, oct 2010. doi: 10.1038/nature09534. 1.1.3.3

[27] B. Ewing and P. Green. Base-calling of automated sequencer traces Using-Phred.II. error probabilities. *Genome Research*, 8(3):186–194, mar 1998. doi: 10.1101/gr.8.3.186. 2.7.1.2

[28] B. Ewing, L. Hillier, M. C. Wendl, and P. Green. Base-calling of automated sequencer traces UsingPhred. i. accuracy assessment. *Genome Research*, 8(3):175–185, mar 1998. doi: 10.1101/gr.8.3.175. 2.7.1.2

[29] M. Farrar. Striped smith-waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, nov 2006. doi: 10.1093/bioinformatics/btl582. (document), 2.8.1.2, 4.13

[30] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0850-2. URL http://dl.acm.org/citation.cfm?id=795666.796543. 2.7.1.7, 2.7.1.7, 2.7.1.7

[31] S. George. Suffix tree indexing for sequence databases. Master's thesis, University of Sydney, Sydney, NSW, Australia, 2006. (document), 2.2.2.2, 2.2

[32] D. H. Ghoneim, J. R. Myers, E. Tuttle, and A. R. Paciorkowski. Comparison of insertion/deletion calling algorithms on human next-generation sequencing data. *BMC Res. Notes*, 7(1):864, Dec. 2014. 4.5.2

[33] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162(3):705–708, Dec 1982. 4.3.3.2

[34] C. M. D. Group. Database of databases search, filter for graph data model, 2024. URL https://dbdb.io/browse?data-model=graph. 3.7.1.3

[35] S. Gupta and K. Ramachandra. Procedural extensions of SQL: understanding their usage in the wild. *Proc. VLDB Endow.*, 14(8):1378–1391, 2021. doi: 10.14778/3457390.3457402. URL http://www.vldb.org/pvldb/vol14/p1378-ramachandra.pdf. 2.10

[36] A. E. Handel, G. Disanto, and S. V. Ramagopalan. Next-generation sequencing in understanding complex neurological disease. *Expert Review of Neurotherapeutics*, 13(2):215–227, feb 2013. doi: 10.1586/ern.12.165. 2.6.3

[37] E. Hayden. Pint-sized dna sequencer impresses first users. *Nature*, 521: 15–16, 2015. A, A.5

[38] J. M. Heather and B. Chain. The sequence of sequencers: The history of sequencing {DNA}. *Genomics*, 107(1):1–8, 2016. ISSN 0888-7543. doi: http://dx.doi.org/10.1016/j.ygeno.2015.11.003. URL http://www.sciencedirect.com/science/article/pii/S0888754315300410. (document), 6.4.1, A, A.5, A.23

[39] O. Horlova, A. Kaitoua, and S. Ceri. Array-based data management for genomics. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 109–120, 2020. doi: 10.1109/ICDE48307.2020.00017. 3.1, 3.3.1

[40] 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. 2.5.1.9

[41] S. Inc. Snowflake: Introduction to external functions, 2022. URL https://docs.snowflake.com/en/sql-reference/external-functions-introduction.html. 2.10.2

[42] B. Institute. Picard toolkit. https://broadinstitute.github.io/picard/, 2019. 2.4.2.1

[43] D. S. J. Li, C. Mcmullan, and D. Branton. Ion-beam sculpting at nanometre length scales. *Nature*, 412:2–5, 2001. A.5

[44] I. Karsch-Mizrachi, Y. Nakamura, and G. C. and. The international nucleotide sequence database collaboration. *Nucleic Acids Research*, 40(D1): D33–D37, nov 2011. doi: 10.1093/nar/gkr1006. 2.5.2

[45] J. J. Kasianowicz, E. Brandin, D. Branton, and D. W. Deamer. Characterization of individual polynucleotide molecules using a membrane channel. *Proceedings of the National Academy of Sciences*, 93(24):13770–13773, 1996. URL http://www.pnas.org/content/93/24/13770.abstract. A.5

[46] D. C. Koboldt, Q. Zhang, D. E. Larson, D. Shen, M. D. McLellan, L. Lin, C. A. Miller, E. R. Mardis, L. Ding, and R. K. Wilson. VarScan 2: Somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome Research*, 22(3):568–576, feb 2012. doi: 10.1101/gr.129684.111. 2.6.3.1

[47] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. M. Yiu. High throughput short read alignment via bi-directional bwt. In *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pages 31–36, 2009. doi: 10.1109/BIBM.2009.42. 6.3.4

[48] W. B. Langdon and B. Y. H. Lam. Genetically improved barracuda. *BioData Min.*, 10(1):28:1–28:11, 2017. doi: 10.1186/s13040-017-0149-1. URL https://doi.org/10.1186/s13040-017-0149-1. 6.4.2.1

[49] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357–359, mar 2012. doi: 10.1038/nmeth.1923. 2.8.1.2

[50] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009. doi: 10.1186/gb-2009-10-3-r25. 4.2.1, 4.3.1.1

[51] R. Leinonen, H. Sugawara, and M. Shumway. The sequence read archive. *Nucleic Acids Res*, 39(Database issue):19–21, Jan 2011. 2.9.3.3

[52] H. Li. Sam and bam format specification v1.4, 2016. URL http://samtools.github.io/hts-specs/SAMv1.pdf. 2.5.1.7, 2.5.1.8

[53] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, may 2009. doi: 10.1093/bioinformatics/btp324. 2.4.2.1, 2.7.1.6, 2.8.1.2

[54] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, nov 2008. doi: 10.1101/gr.078212.108. 2.7.1.1, 2.7.1.2

[55] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. D. and. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, jun 2009. doi: 10.1093/bioinformatics/btp352. 2.5.1.7, 2.6.3.2

[56] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, jan 2008. doi: 10.1093/bioinformatics/btn025. 2.7.1.1

[57] Y. Li, J. M. Patel, and A. Terrell. WHAM. *ACM Transactions on Database Systems*, 37(4):1–39, dec 2012. doi: 10.1145/2389241.2389247. 2.7.1.1

[58] Y. Liu and B. Schmidt. CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing. *IEEE Des. Test*, 31(1):31–39, 2014. doi: 10.1109/MDAT.2013.2284198. URL https://doi.org/10.1109/MDAT.2013.2284198. 6.4.2.1

[59] Y. Liu, B. Schmidt, and D. L. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the burrows-wheeler transform. *Bioinform.*, 28(14):1830–1837, 2012. doi: 10.1093/bioinformatics/bts276. URL https://doi.org/10.1093/bioinformatics/bts276. 6.4.2.1

[60] C. Lomont. Introduction to intel® advanced vector extensions, 2011. URL https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions. (document), 2.17, 2.18

[61] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, H.-F. Ting, S.-M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T.-W. Lam. Soap3-dp: Fast, accurate and sensitive gpu-based short read aligner. *PLOS ONE*, 8(5):1–11, 05 2013. doi: 10.1371/journal.pone.0065632. URL https://doi.org/10.1371/journal.pone.0065632. 6.4.2.1

[62] E. R. Mardis. Next-generation DNA sequencing methods. *Annual Review of Genomics and Human Genetics*, 9(1):387–402, sep 2008. doi: 10.1146/

annurev.genom.9.081307.164359. (document), A, A.12, A.3, A.13, A.14, A.15, A.16, A.17, A.18

[63] S. McGinnis and T. L. Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Research*, 32(Web Server):W20–W25, jul 2004. doi: 10.1093/nar/gkh435. 2.7.1.1

[64] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, 20(9):1297–1303, 2010. doi: 10.1101/gr.107524.110. URL http://genome.cshlp.org/content/20/9/1297.abstract. 2.4.2.1

[65] J. Melton. *SQL:1999 : understanding relational language components*. Academic Press, San Francisco, 2002. ISBN 9781558604568. 2.2

[66] Memgraph. Memgraph: Open-source graph database, tuned for dynamic analytics environments. easy to adopt, scale and own., 2024. URL https://github.com/memgraph/memgraph. 3.7.1.3

[67] T. P. Michael. Plant genome size variation: bloating and purging dna. *Briefings in Functional Genomics*, 13(4):308, 2014. doi: 10.1093/bfgp/elu005. URL http://dx.doi.org/10.1093/bfgp/elu005. 4.3.2.2

[68] Microsoft. All about span: Exploring a new .net mainstay, 2018. URL https://learn.microsoft.com/en-us/archive/msdn-magazine/2018/january/csharp-all-about-span-exploring-a-new-net-mainstay. 3.3.2

[69] Microsoft. Binary large object (blob) data (sql server), 2020. URL https://learn.microsoft.com/en-us/sql/relational-databases/blob/binary-large-object-blob-data-sql-server?view=sql-server-ver16. 2.1.4, 4.1.3.1

[70] Microsoft. Installation guidance for sql server on linux, 2022. URL https://learn.microsoft.com/en-us/sql/linux/sql-server-linux-setup?view=sql-server-ver16#system. 2.1.3.1

[71] Microsoft. CLR Table-Valued Functions, 2023. https://learn.microsoft.com/en-us/sql/relational-databases/clr-integration-database-objects-user-defined-functions/clr-table-valued-functions?view=sql-server-ver15. 4.1.3.2

[72] M. Mielczarek and J. Szyda. Review of alignment and SNP calling algorithms for next-generation sequencing data. *Journal of Applied Genetics*, 57(1):71–79, jun 2015. doi: 10.1007/s13353-015-0292-7. 2.6.3

[73] NCBI, 2017. URL https://www.ncbi.nlm.nih.gov/sra/docs/submitformats/#HDF5. 6.4.1, A.5

[74] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, mar 1970. doi: 10.1016/0022-2836(70)90057-4. 4.3.3.3

[75] Neo4j. Neo4j - the world's leading graph database, 2012. URL http://neo4j.org/. 3.7.1.3

[76] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500. 6.4.2.1

[77] R. Nielsen, J. S. Paul, A. Albrechtsen, and Y. S. Song. Genotype and SNP calling from next-generation sequencing data. *Nature Reviews Genetics*, 12(6):443–451, jun 2011. doi: 10.1038/nrg2986. 2.6.3, 2.6.3.2, 2.6.3.3

[78] nVidia. NVBIO, 2017. http://nvlabs.github.io/nvbio/. 6.4.2.1

[79] NVidia. Cuda development toolkit - ptx isa, 2022. URL https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. 4.3.4

[80] S. I. O'Donoghue. Visualization of macromolecular structures. *Nat. Methods*, 7:S42–S55, 2010. doi: 10.1038/nmeth.1427. URL http://dx.doi.org/10.1038/nmeth.1427. 1.1

[81] Oracle. Tune the size of database pages, 2005. URL https://docs.oracle.com/javadb/10.6.2.1/tuning/ctunperf10065.html. 3.3

[82] Oracle. Working with lobs and bfiles, 2010. URL https://docs.oracle.com/cd/E18283_01/java.112/e16548/oralob.htm. 2.1.4

[83] Oracle. The blob and text types, 2022. URL https://dev.mysql.com/doc/refman/8.0/en/blob.html. 2.1.4

[84] J. Ostell. Integrated access to heterogeneous biomedical data from NCBI. *IEEE Eng. Med. Biol.*, 14:730–736, 1995. 2.5.2

[85] J. Pantaleoni. A massively parallel algorithm for constructing the BWT of large string sets. *CoRR*, abs/1410.0562, 2014. URL http://arxiv.org/abs/1410.0562. (document), 6.1, 6.4.2.1, 6.2

[86] PostgreSQL. lo, 2022. URL https://www.postgresql.org/docs/15/lo.html. 2.1.4

[87] PostgreSQL. Database page layout, 2022. URL https://www.postgresql.org/docs/current/storage-page-layout.html. 3.3

[88] PostgreSQL. Toast, 2022. URL https://www.postgresql.org/docs/current/storage-toast.html. 3.3

[89] M. Research. .net bio, 2022. URL https://dotnetfoundation.org/projects/netbio. 3.3.1

[90] T. Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12:221, Jun 2011. (document), 2.8.1.2, 4.13

[91] T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, Aug 2000. (document), 2.8.1.2, 4.13

[92] U. Röhm and J. A. Blakeley. Data management for high-throughput genomics. In *Proceedings of the 4th Biennial Conference on Innovative Data*

Systems Research (CIDR 2009), Asilomar, CA, USA, January 4-7, 2009, Jan. 2009. URL http://arxiv.org/abs/0909.1764. 4.1.3.1, 4.1.3.3

[93] U. Röhm and T.-M. Diep. How to BLAST your database — a study of stored procedures for BLAST searches. In *Database Systems for Advanced Applications*, pages 807–816. Springer Berlin Heidelberg, 2006. doi: 10.1007/11733836_58. 2.2

[94] K. Saur, T. Mirmira, K. Karanasos, and J. Camacho-Rodríguez. Containerized execution of udfs: An experimental evaluation. *Proc. VLDB Endow.*, 15(11):3158–3171, 2022. URL https://www.vldb.org/pvldb/vol15/p3158-saur.pdf. 2.10.2

[95] R. Sheldon. Sql server performance tuning: Nine best practices, 2021. URL https://www.red-gate.com/simple-talk/databases/sql-server/database-administration-sql-server/sql-server-performance-tuning-nine-best-practices/. (document), 2.1, 3.3

[96] A. Smit, R. Hubley, and P. Green. Repeatmasker home page, 2017. URL http://repeatmasker.org/. (document), 2.9.3.4, 3.3

[97] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, Mar 1981. 2.7.1.2

[98] R. Staden. The staden sequence analysis package. *Molecular Biotechnology*, 5(3):233–241, jun 1996. doi: 10.1007/bf02900361. 5.2.2

[99] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):1–11, 07 2015. doi: 10.1371/journal.pbio.1002195. URL https://doi.org/10.1371/journal.pbio.1002195. (document), 1.1

[100] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73, 2010. doi: 10.1109/MCSE.2010.69. URL http://dx.doi.org/10.1109/MCSE.2010.69. 6.4.2.1

[101] D. Tack. Ion semiconductor sequencing, Feb 2011. URL https://en.wikipedia.org/wiki/Ion_semiconductor_sequencing. (document), A, A.19, A.20, A.21

[102] S. Tata, R. A. Hankins, and J. M. Patel. Practical suffix tree construction. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 36–47. VLDB Endowment, 2004. ISBN 0-12-088469-0. URL http://dl.acm.org/citation.cfm?id=1316689.1316695. 2.2.2.2

[103] The HDF Group. Hierarchical Data Format, version 5, 1997-2017. http://www.hdfgroup.org/HDF5/. 6.4.1, A.5

[104] J. H. University and Medicine. The search for covid-19 variants, 2022. URL https://coronavirus.jhu.edu/data/variant-data. 2.9.3.5

[105] G. Van der Auwera. Full release of open-source dragen-gatk 1.0, 2021. URL https://gatk.broadinstitute.org/hc/en-us/articles/4411716682011-Full-release-of-open-source-DRAGEN-GATK-1-0. 2.4.2.2

[106] G. A. Van der Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. V. Garimella, D. Altshuler, S. Gabriel, and M. A. DePristo. From fastq data to high-confidence variant calls: The genome analysis toolkit best practices pipeline. *Current Protocols in Bioinformatics*, 43(1):11.10.1–11.10.33, 2013. doi: https://doi.org/10.1002/0471250953.bi1110s43. URL https://currentprotocols.onlinelibrary.wiley.com/doi/abs/10.1002/0471250953.bi1110s43. 2.4

[107] O. B. Van der Auwera GA. *Genomics in the Cloud: Using Docker, GATK, and WDL in Terra (1st Edition)*. O'Reilly Media, 2020. 2.4.1

[108] N. Veltman. Sql: The prequel (excel vs. databases), Nov 2013. URL https://schoolofdata.org/2013/11/07/sql-databases-vs-excel/. 1.1.1

[109] Z. Wei, W. Wang, P. Hu, G. J. Lyon, and H. Hakonarson. SNVer: a statistical tool for variant calling in analysis of pooled or individual next-generation sequencing data. *Nucleic Acids Research*, 39(19):e132–e132, aug 2011. doi: 10.1093/nar/gkr599. 2.6.3.2

[110] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Comput. Appl. Biosci.*, 13(2):145–150, Apr 1997. (document), 2.8.1.2, 4.13

[111] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and more accurate sequence alignment with SNAP. *CoRR*, abs/1111.5572, 2011. URL http://arxiv.org/abs/1111.5572. 2.7.1.1, 2.7.1.3