THE HAIL SYSTEM: COMPUTING FOR DATA-INTENSIVE SCIENCE

Daniel King
The Broad Institute of MIT and Harvard
Cambridge, MA, USA
dking@broadinstitute.org

ABSTRACT

PVLDB Reference Format:
Daniel King. THE HAIL SYSTEM:
COMPUTING FOR DATA-INTENSIVE SCIENCE. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hail-is/hail.git.

1 INTRODUCTION

The Hail System is an open-source, multi-tenant, spot-tolerant, elastic, horizontally scalable, cost-metered workflow engine, relational algebra engine, and linear algebra engine. We describe its unique composition of capabilities that enable the analysis of very large human genome sequencing datasets by computational scientists without expertise in distributed systems or data engineering.

1.1 Genetics and Genomics

Genetics is the study of inheritance and genomics is the study of the genome. These two fields endeavor to mechanistically explain the effect on phenotypes of variation in the genome. For example, in the aughts, researchers discovered that certain mutations in the gene PCSK9 lead to an increased risk of coronary artery disease and death [10][1]. Other researchers discovered healthy adult individuals in which PCSK9 was essentially disabled [3]. This evidence lead to the development of new drugs that inhibit PCSK9 providing an alternative to statins and other drugs for lowering LDL levels.

At the time PCSK9 was discovered, the human genome project was just finishing. The assembly and analysis of just one sequence was a mammoth undertaking in cost and time. In the twenty years since, we have scaled up from one sequence to one million. This represents a near doubling in the number of samples every year for twenty consecutive years.

As the rate of sequencing grew, the challenge of deriving insight from a sequencing dataset transformed from a biological engineering challenge into a software engineering one. The Hail System represents our approach to tackling these challenges.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1.2 The Three Phases of Data-Intensive Science

In 2007, Jim Gray exhorted the computer science community to do a better job of producing tools to support the whole of scientific research. He identified four paradigms of science: empirical, theoretical, computational, and "data exploration", "eScience", or, our preferred term, "data-intensive science" [12]. Data-intensive science is characterized the use of automated machines and computers to capture, process, store, and analyze large datasets. Experientially, data-intensive science begins where Excel breaks down.

Within genomics, and we hypothesize analogous phases exist in other fields, data-intensive science has three analysis phases:

- Primary. The physical-digital interface: sequencers, telescopes.
- Secondary. Recode technical measurements into model-relevant values: alignment.
- Tertiary. Interactive & iterative knowledge making: regression, machine learning.

Data flows from the physical world through primary, second, and finally into tertiary analysis. The product of tertiary analysis is scientific insight, encoded both in research papers and released datasets. The phases live in conversation with one another. For example, more sophisticated tertiary analysis may enable primary analysis to use cheaper or higher-throughput methods [14]. Moreover, as tertiary analysis advances scientific understanding, secondary analysis ought to adopt new data representations which reflect this new understanding.

Unfortunately, secondary and tertiary analysis have historically used different computational primitives. Providing one set of primitives that serves both phases empowers tertiary scientists to upstream the advances into secondary analysis, thus benefiting all consumers of secondary analysis products.

1.3 Genomic Data Types

A typical human genome has about three billion locations, called loci, across 24 contigs (22 pairs of autosomes, one pair of sex chromosomes, and the mitochondrial genome). Each locus is identified by its contig and 1-indexed position on the contig. The human *reference* genome is, in essence, a three billion character string over the alphabet of *DNA nucleobases* $\{A, G, T, C\}$. An allele is a sequence of nucleobases observed at a locus. An allele is not necessarily length one; indeed, small and large regions of a particular sequence can be inserted or deleted relative to the reference. A *variant site* is a locus and the list of alleles observed at that locus.

A sequenced human genome is represented as a sparse vector indexed by locus. At each locus is an experimental observation with a point-estimate, the genotype, and quality metadata about that experiment. A genotype is the tuple of alleles observed at this locus.

Sequenced human genomes are sparse because most genotypes comprise only the reference allele.

Sequenced human genomes are usually stored in the plain-text Variant Call Format (VCF) [7], compressed by Block Gzip, and represented as Genomic VCF (GVCF) [6], which uses run-length encoding. The VCF is a tab-separated representation of genome-major, sample-minor matrices of integers, floats, strings, and arrays thereof. In addition to the elements of the matrix, the VCF permits an arbitrary number of genomic metadata columns.

Exome sequencing is the process of measuring a subset of the human genome that covers all the protein-coding genes. An sequenced exome is typically one-hundredth the size of a sequenced genome. We use the term "sequence" to generically refer to a genome or an exome.

1.4 Genomic Tertiary Analysis

Tertiary analysis begins by combining one or more sequences into a variant-site by sample matrix. This operation is effectively transposition. Analysis of this matrix proceeds as follows:

- Quality control. Identify a set of high quality genotypes, variants, and samples. Quality is determined by interrogation of the per-genotype quality metadata; statistical aggregates across each variant-site, each sample, variant-regions, and sample groups; and comparison to variant-site and sample metadata.
- Annotation. In addition to their use for quality control, variant-site and sample metadata are joined for use in the analysis of phenotypes.
- Correlation. Many statistical genetics methods assume uncorrelated samples and features. Satisfying this assumption requires scientists to first compute variant-site and sample correlation matrices and then filter the original dataset to sets deemed sufficiently uncorrelated.
- Regression analyses. Geneticists are interested in a litany of statistical models including:
  - Per-variant-site linear, logistic, & Poisson regression.
  - Per-genomic-region linear, logistic, Poisson regression as well "kernel" methods.
  - Mixed models, which extend the above methods with non-diagonal sample covariance.
- Legacy tools. Genetics and genomics has a large library of specialized and optimized tools. They are written in a litany of languages from Java to C++ to Perl to Python to R. These tools are almost universally single-machine and require the user to manually partition datasets along the genomic axis.
- Visualization. A regression analysis may yield 30 million to one billion results.

Following the example of [9], we explore a few simple queries of interest to the genetics and genomics community and their realization in different query languages. The queries are significantly influenced by the chosen representation. In BigQuery, we use a "coordinate" or "tall" representation where each row corresponds to a locus, allele, sample triplet.

(1) For each variant-site, calculate the allele frequency for each allele.
(2) For each variant-site, calculate the allele frequency for each allele for each user-defined sample group.
(3) For each interval of variant-sites, regress the count of predicted-loss-of-function non-reference alleles against the phenotypes.

```
----------------------------------------
Global fields:
    None
----------------------------------------
Column fields:
    'sample_id': str
    'sample_group': int32
----------------------------------------
Row fields:
    'locus': locus<GRCh37>
    'alleles': array<str>
----------------------------------------
Entry fields:
    'GT': call
----------------------------------------
Column key: ['sample_id']
Row key: ['locus', 'alleles']
----------------------------------------
```

(a) Hail explicit

```
create table mydataset.mt (
  locus: int64,
  ref_allele: string,
  alt_allele: string,
  sample_id: string,
  sample_group: int64,
  n_alt_alleles: int64,
  primary key (locus, ref_allele, alt_allele)
)
cluster by locus, ref_allele, alt_allele
```

(b) BigQuery

Figure 1: Schema in each system. In Hail, we use a Matrix Table which is a wide representation. Each physical record contains an array of entries. In BigQuery, we use a tall representation. Each physical record holds one entry.

```
mt.annotate_variants(
  af = hl.agg.call_stats(mt.GT, mt.alleles).AF)
```

(a) Hail

```
n_alleles = hl.len(mt.alleles)
mt.annotate_variants(
  af = hl.range(n_alleles).map(
    lambda i: hl.agg.mean(mt.GT.one_hot_alleles()[i])))
```

(b) Hail explicit

```
select locus, ref_allele, alt_allele,
       avg(n_alt_alleles) as af
from mt
group by locus, ref_allele, alt_allele
```

(c) BigQuery

Figure 2: Calculating allele-frequencies.

```
mt.annotate_variants(
  af = hl.agg.group_by(
    mt.sample_group,
    hl.agg.call_stats(mt.GT, mt.alleles)))
```

(a) Hail

```
n_alleles = hl.len(mt.alleles)
mt.annotate_variants(
  af = hl.range(n_alleles).map(
    lambda i: hl.agg.group_by(
      mt.sample_group,
      hl.agg.mean(mt.GT.one_hot_alleles()[i]))))
```

(b) Hail explicit

```
select locus, ref_allele, alt_allele,
       sample_group, avg(n_alt_alleles) as af
from mt
group by locus, ref_allele, alt_allele, sample_group
```

(c) BigQuery

Figure 3: Calculating allele frequencies by sample group.

```
mt = mt.filter_rows(mt.predicted_lof)
mt = mt.annotate_rows(
  interval = interval_table.index(
    mt.locus, all_matches=True))
mt = mt.explode_rows(mt.interval)
mt = mt.group_rows_by(
  mt.interval
).aggregate(
  n_plof = hl.agg.sum(mt.GT.n_alt_alleles()))
mt = mt.annotate_rows(
  result=hl.agg.linreg(y=mt.phenotype, x=[1.0, mt.n_plof]))
```

(a) Hail

```
create model `result`
options ( model_type='LINEAR_REG' ) as
select
  sum(mt.n_alt_alleles) as n_plof,
  mt.phenotype as label
from mt
left join interval_table
on interval_table.left_endpoint >= mt.locus
and mt.locus <= interval_table.right_endpoint
where mt.predicted_lof
group by interval_table.left_endpoint,
         interval_table.right_endpoint

select *
from ml.advanced_weights(
  model `result`, struct(false as standardize))
```

(b) BigQuery

Figure 4: Regression of intervals.

1.5 The Hail System

For the past eight years, we have developed the Hail System to enable tertiary analysis of sequencing datasets. It comprises several components:

(1) Two high-level interfaces.
   (a) Hail Query. Dataframes supporting relational and linear algebra on tables and matrices.
   (b) Hail Batch, a library for defining "workflows", directed acyclic graphs of arbitrary GNU/Linux binaries.
(2) Hail Scheduler. Elastic, multi-tenant, scalable, container-based cloud cluster scheduler.
(3) Cloud-native file API supporting local files and cloud object storage.
(4) In-memory analysis-oriented data formats.
(5) At-rest data formats.
(6) At-rest indexed, partitioned dataset formats.
(7) Query planner and optimizer for relational operations.
(8) Expression compiler for compiling scalar operations.
(9) Spot-tolerant, distributed sort.

Tertiary analysis typically makes use of every component. Specifically:

(1) Relational and linear algebra computes statistics and correlations.
(2) Workflows execute critical legacy software massively in parallel.
(3) The entire analysis is scheduled across the same multi-tenant cluster. Cost, logs, resource usage, and profiling information are all available through a single web user interface.
(4) The cloud-native file API is used both in Hail Query data processing code as well as client-side to load small results onto the user's machine.
(5) Immutable and cheap-to-decode in-memory formats are used.
(6) ??? at-rest data
(7) Hail's indexed, partitioned formats enable fast subsetting on the genomic axis.
(8) Expression compiler compiles Python-like operations into efficient JVM bytecode.
(9) Distributed sort powers both joins and grouped aggregations.

1.6 The Importance of Dataframes

Dataframe APIs embedded in general purpose programming languages empower users to automate and build custom abstractions. In particular, the Genome Aggregation Database, the largest public and non-commercial human sequencing dataset, curates a pair of Python libraries totalling 48,000 lines of code built on top of the Hail library without the direct involvement of the Hail team.

1.7 Contributions

The Hail System is unique primarily for its composition, not its particular components. Few other systems provide a unified API for workflows of binaries, dataframe-based relational algebra, and linear algebra on partitioned datasets executed by a serverless backend. Of the major distributed, serverless data engines, Databricks supports user-provided container images, Snowflake has a closed beta for executing user-provided container images, and BigQuery has no such support. Databricks has long supported a dataframe interface. Snowflake introduced dataframes in 2022. In 2023, BigQuery introduced a pre-GA dataframe API.

As far as we know, the Hail System is the only public and open-source system supporting workflows of binaries, dataframe-based relational algebra, and linear algebra on partitioned datasets.

2 BACKGROUND AND RELATED WORK

2.1 Workflow Languages, Workflow Engines, and Serverless Engines

Serverless engines execute tasks at least once. A task is typically defined by an OCI container image, an execve invocation, and resource requirements.

A workflow engine executes a directed acyclic graph of tasks. Some systems allow tasks to read files produced by tasks on which they depend. Workflow engines may or may not serverless. Non-serverless workflow engines either require manual provisioning of VMs or automatically start and stop VMs for each task.

There are many workflow languages. A few of interest to the bioinformatics community include: Common Workflow Language (CWL), Workflow Description Language (WDL), Apache Airflow, Galaxy, and Nextflow. CWL, WDL, Galaxy, and Nextflow all define custom languages for describing workflows. In contrast, Apache Airflow simply exposes a Python API.

FIXME: are dags static or dynamic, looping/vector jobs.

There are also many workflow engines. Each of the aforementioned languages has its own workflow engine. Some of these systems support multiple languages.

Scaling serverless execution systems to thousands of VMs and millions of containers is challenging enough that Hashicorp and OpenAI have collectively written four blog posts about it. Hashicorp described using Nomad to schedule one million containers across 5,000 VMs in less than five minutes [11] and two million containers across 6,100 VMs in 22 minutes and 14 seconds. OpenAI described scaling Kubernetes-based workloads to 2,500 VMs [2] and then 7,500 VMs [19].

2.2 Analytical Query Languages and Engines

The bioinformatics community uses many different query languages including Excel, grep, awk, Pandas DataFrames, R DataFrames, SQL, and Spark DataFrames. Before the last ten years of rapid data size growth, bioinformaticians largely embraced the Unix philosophy. Use of grep, awk, cut, and paste was pervasive. More complex analyses were written in C, C++, perl, or R. Since the mid 2010s, trainees are largely fluent in Python or R, but have rarely written a SQL query.

The Pandas DataFrame interface is the *de facto* standard for dataframes in the Python ecosystem. Two notable alternatives to Pandas are Ibis, which aims to provide a single interface to many query engines, and xarray, which generalizes from single-axis, tabular dataframes to multiple-axis tensorial dataframes.

Commonly used single-machine query engines include R DataFrames, Pandas DataFrames, sqlite, MySQL, and PostgreSQL. The last ten years have seen an explosion of embedded, single-machine, analytical query engines (though some optionally integrate with a distribution system). Vaex [4], originally developed for astronomical datasets; Modin [16]; Velox [15], a library for building query engines; Apache DataFusion [8]; DuckDB [18], an analytical analogue of `sqlite`; and Polars.

The high energy physics community maintains ROOT, a custom, column-oriented query engine suited to their needs [5]. Physicists prefer to store their data denormalized because the data for a given "event" is always analyzed in the context of that event.

Commonly used distributed analytical query engines include Snowflake, Databricks/Apache Spark, BigQuery, Azure Synapse Analytics, Amazon Athena, Dask, and Databend.

2.3 Linear Algebra Engines

Distributed linear algebra is historically the domain of HPC, in particular OpenMPI-based solutions. Elemental [17] is a relatively new system which revisits the design choices of ScaLAPACK and PLAPACK in light of hardware developments since the 90s, particularly the rise of multi-core machines. Development of Elemental has ceased but a fork focused on GPGPUs named Hydrogen is under active development by Lawrence Livermore National Laboratories [13].

Along the lines of rethinking linear algebra as the non-uniform memory hierarchy grows, Smith and van de Geijn propose a systematic way to derive efficient algorithms for matrix multiplication for systems with any number of caches [20]. They note and we agree that: "Hard drives and other similarly slow storage devices can be thought of as another layer of the memory hierarchy. Because of this, we believe the methodology in this paper can be used to instantiate out-of-core algorithms for MMM."

Shankar et al. developed numpywren to demonstrate the promise of distributed linear algebra on serverless engines. A key difference between Numpywren and traditional distributed relational algebra engines like Apache Spark is the need for directed acyclic graphs of tasks rather than simple fan-out-fan-in graphs.

2.4 Relational Algebra on Tensors

FIXME: GenomicsDB, TileDB, SciDB

2.5 Partitioned Datasets

The authors are aware of only one file format (besides Hail's own) with first-class support for partitioning: Zarr. Zarr stores partitions of the dataset in distinct files, which makes it appropriate for use with cloud object stores like S3. A Zarr array conceptually is a multi-dimensional array. The set of possible element types include integers, floats, structures, strings, and arrays thereof.[1]

Parquet and ORC are efficient column-oriented formats. Tools, such as Apache Spark, support partitioning a into a collection of Parquet or ORC files. While the Parquet docs note that multiple Parquet data files could use a single metadata file, in practice, query engines do not create such a file. As a result, collecting the key bounds or the number of records in each partition requires $O(N_{partitions})$ object storage requests.

FIXME: TileDB ...

HDF5 is not a partitioned format; however, Highly Scalable Data Service (HSDS) permits the partitioned storage of HDF5 in cloud object stores. HSDS is usually a separate process which sits between the object store and the HDF5 client application. There is at least one Python implementation of HSDS in Python which does not require a separate process.

[1] Variable length data types require extra user-specified configuration.

2.6 The Essence of the Cloud

The two essential features of the cloud are spot instances charged with a granularity of seconds and object storage.

3 HAIL QUERY

Hail Query has two data types: big data types and small data types. Big data types are too large to fit in the main memory of a computer. Small data types fit in main memory. There are three big data types: tables, matrix tables, and block matrices. Big data types have *fields*, identified by Unicode strings, indexed by zero or more axes, and taking on one value for each present coordinate of the axes. An axis has zero or more key fields. The values of a key field need not be unique.

Tables are analogous to SQL tables and dataframes. Tables have two field sets: global fields, which are indexed by no axes and have exactly one value, and row fields, which are indexed by the row axis. Matrix tables are analogous to "wide" dataframes in which a subset of the columns have the same data type and logically comprise a two-dimensional matrix. Matrix tables have four field sets: global fields, row fields, column fields, which are indexed by the column axis, and entry fields, which are indexed by both the row and column axes. Block matrices are analogous to linear algebraic matrices. They one unique field, the element field, which must have a numeric data type.

Every small data type includes the unique missing value, has a total ordering, has at least one in-memory format, and at least one at-rest format. The data types include 32- and 64-bit integers, 32- and 64-bit floats, arrays, sets, dictionaries, structs, tuples, UTF-8 strings, genotype calls, genomic loci (parameterized by a reference genome), ndarrays, and intervals.

3.1 At-rest big data type format

All the big data types use a compressed, indexed, row-key-ordered, row-partitioned, row-oriented at-rest format. The row-orientation makes decoding fairly expensive, in particular, a single-threaded Hail decoder does not saturate a 10gbps link.

Tables and matrix tables additionally store two metadata files per field set containing, per-partition: the minimum and maximum key and the number of records. These metadata files also include the field set's schema and the at-rest small data type format.

3.2 At-rest small data type format

The small data type use common at-rest formats such as variable length integers, IEEE 754, UTF-8 strings, and C-like struct layouts. Dictionaries are stored as sorted arrays of key-value pairs. Sets are stored as sorted arrays. Arrays are sparsely stored with a bitmask indicating which elements are the missing value.

3.3 In-memory format

Big data types, by their nature have no in-memory format. A single partition of a big data type is represented as a *stream* which is a pull-based, code-generated, lazy sequence of values.

Small data types use common in-memory formats such as two's complement integers, IEEE 754, UTF-8 strings, and C-like struct layouts. Dictionaries are arrays of key-value pairs. Sets are sorted arrays. Arrays are densely stored with a bitmask indicating which elements are the missing value.

We use region-based[2] memory management for every type other than ndarrays. There are always at least two rows: one per-record row and one per-partition row. Ndarrays are reference counted because they tend to be large and are used inside loops, whose body's correspond to a region.

3.4 Expression language

The Hail Query expression language is a mutation-free and Turing-complete language supporting all the aforementioned small data types. Switches, ifs, array maps, array folds, array zips, and loops are all present. We call the expression language ValueIR and consider it one of the four sub-languages of the wider IR[3] language, described below.

3.5 Expression language DSL

The expression language is exposed in Python as an embedded domain specific language which looks and feels like Python. The Python DSL is lazy. In our experience, users find the lazy client API frustrating and confusing; however, implementing an efficient eager API on partitioned datasets is challenging due to the need to store potentially large intermediates.

Operations which are not lazy are called *actions*. The table actions include: `write`, `collect`, `aggregate`, and `export`. Matrix table actions include: `write`, `aggregate_rows`, `aggregate_cols`, an `aggregate_entries`.

3.6 Randomness

Hail ensures nearly complete[4] pipeline execution determinism. Every random sampling function is passed a sequence of identifiers, starting with a unique static identifier. Samples which appear inside one or more iteration contexts are passed variable references which are bound by the iteration context to a dynamic identifier of this particular iteration. These identifiers are passed to counter-based random number generators, in particular Threefry, to produce a deterministic sequence of samples. This scheme frees the query planner to change the dataset partitioning and reorder statements without changing the observed random samples.

3.7 Relational and linear algebra

Whereas small data types are manipulated by the ValueIR expression language, large data types are manipulated by three relational and linear algebraic sub-languages of IR: TableIR, MatrixIR, and BlockMatrixIR. TableIR roughly matches SQL, see Figure 5 for a translation between SQL and the Hail Query Python API.

The MatrixIR, for manipulating matrix tables, is a two-axis generalization of TableIR. There are two interesting issues: filtered entries and joins.

3.7.1 Filtered entries. Filtering a row (respectively, a column) removes that coordinate of the axis entirely along with all the corresponding row (respectively, column) fields and entry fields. In

[2] Also known as arenas or zones.
[3] Intermediate representation.
[4] Sorting currently involves some randomness. We consider this a bug.

SQL	Hail Query Python API	Hail Query TableIR

`SELECT a, b`	`t1.select(t1.a, t1.b)`	`TableMapRows(!t1) { (%g, %row) =>`
		`  SelectFields(%row) [(key a b)]`
		`}`

`WHERE a == b`	`t1.filter(t1.a == t1.b)`	`TableFilter(!t1) { (%g, %row) =>`
		`  !1 = GetField(%row) [a]`
		`  !2 = GetField(%row) [b]`
		`  !3 = ApplyComparisonOp(!1, !2) [EQ]`
		`  !false = False`
		`  Coalesce(!3, !false)`
		`}`

`X JOIN t2`
`ON t1.a = t2.b`
	`t1.key_by(t1.a) \`
	`  .join(t2.key_by(t2.b), how='X')`
		`!t1_1 = TableKeyBy(!t1) [(a), False]`
		`!t2_1 = TableKeyBy(!t2) [(b), False]`
		`TableJoin(!t1_1, !t) [X, 1]`

`SELECT SUM(a)`	`t1.aggregate(hl.agg.sum(t1.a))`	`TableAggregate(!t1) { (%g, %row) =>`
		`  !1 = GetField(%row) [a]`
		`  ApplyAggOp(!1) [Sum]`
		`}`

`SELECT SUM(a) AS a`
`GROUP BY b`
	`t1.group_by(t1.b) \`
	`  .aggregate(a=hl.agg.sum(t1.a))`
		`TableKeyByAndAggregate(!t1) [None, 50] { (%g, %row) =>`
		`  !1 = GetField(%row) [a]`
		`  !2 = ApplyAggOp(!1) [Sum]`
		`  MakeStruct(a: !2)`
		`} { (%g2, %row2) =>`
		`  SelectFields(%row2) [(b)]`
		`}`

`ORDER BY a`	`t1.order_by(t1.a)`	`TableOrderBy(!t1) [(Aa)]`
`LIMIT n`	`t1.head(n)`	`TableHead(!t1) [1]`
`PRIMARY KEY k1, k2`	`t1.key_by(k1, k2)`	`TableKeyBy(!t1) [(a b), False]`

`HAVING a == b`	`t1.filter(t1.a == t1.b)`	`TableFilter(!t1) { (%g, %row) =>`
		`  !1 = GetField(%row) [a]`
		`  !2 = GetField(%row) [b]`
		`  !3 = ApplyComparisonOp(!1, !2) [EQ]`
		`  !4 = False`
		`  Coalesce(!3, !4)`
		`}`

Figure 5: Translating between SQL and Hail Query tables.

contrast, filtering an entry does not modify the axes at all: it creates a hole in the matrix. Filtered entries are excluded from aggregations. For example, the 2x2 identity matrix has a mean value of $\frac{1+1+0+0}{4} = 0.5$; however, when the zero valued entries are filtered out, the mean value is $\frac{1+1}{2} = 1$.

Filtered entries are used by scientists to indicate values that should be treated as "not observed". Consider the case of genome sequencing. A genotype is derived from measurements called "reads". Each read is evidence of certain alleles. If we believe that, for a particular genotype, all the reads are invalid due to some technical error (such as contamination), we would filter that entry. In contrast, if we have successfully collected reads for a given entry but the collected reads do not confidently support any particular genotype, we might mark the genotype as missing but keep the entry in the dataset. In the former case, we assert no observation was made, whereas in the latter case we keep the observation and choose to model missing data in our downstream statistical methods.

3.7.2 Matrix table joins. There are sixteen possible joins between two matrix tables formed by the product of the four possible joins on each axis. Hail only provides two joins: left join on rows with an exact match of columns and outer-join on both axes. In practice, geneticists rarely join two matrix tables together.

3.8 Architecture

Hail Query has two distributed computing backends: Apache Spark and Hail Batch. Hail Query has three execution components: the *client*, the *driver*, and the *workers*. The client is often a process on the scientist's laptop. The client to driver communication is backend-dependent, but usually an HTTP socket. The driver to worker communication is also backend-dependent. In Hail Batch, the driver writes data into object storage which is read by the workers.

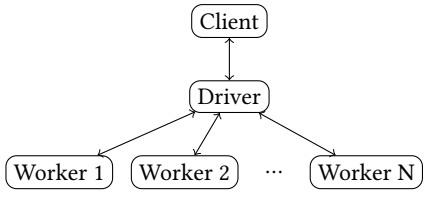

Figure 6: The architecture of Hail Query.

3.9 Query planner

Before executing IR, all TableIR, MatrixIR, and BlockMatrixIR nodes eliminated in favor of ValueIR. Distributed operations are represented in ValueIR as CollectDistributedArray which takes a list of "contexts", a "global" value, and a "body", which is a function of a context and a global.

The LowerMatrixIR pass expresses MatrixIR in terms of TableIR. The LowerTableIR pass expresses TableIR in terms of CollectDistributedArray and other ValueIR. LowerBlockMatrixIR likewise expresses BlockMatrixIR in terms of CollectDistributedArray and other ValueIR.

BlockMatrixIR cannot be efficiently expressed in terms of TableIR because the former explicitly refers to blocks by their two dimensional block coordinates. In contrast, TableIR partitioning is implicitly defined by reads and joins.

Between each pass, a series of analyses and optimizers run include: constant folding, predicate pushdown, field pruning, inlining, and simplification.

CollectDistributedArray requires that body code safely executes concurrently with other executions of the body code, possibly with the same arguments. This requirement simplifies implementation of CollectDistributedArray in terms of distributed frameworks which guarantee at-least-once execution of tasks. For Hail Query, the primary concern is writing to a file name unique to this execution. In Hail Query, a short random string is included in the filename. Filenames are returned to the driver which deletes unexpected files produced by duplicate executions.

FIXME: more discussion of CDA.

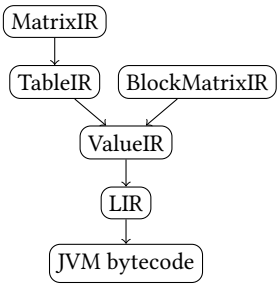

Figure 7: The compilation of flow of Hail Query.

3.10 Expression compiler

ValueIR is compiled into LIR (low-level intermediate representation), which is quite similar to JVM bytecode except it is register-based and has no size limits on methods or classes. On the LIR, we construct a program structure tree [?], split large methods (to avoid downstream JVM limitations), and eliminate dead code. Finally, we compile LIR to JVM bytecode.

Compiling LIR to JVM bytecode is straightforward. In contrast, ValueIR and LIR are very different languages and the compilation step is quite large. There are over one hundred IRs, including twenty stream IRs and sixteen aggregation or scan IRs. There are three specialized components of the compilation: ndarray compilation, stream compilation, and aggregations and scans compilation.

3.10.1 Ndarray compilation. Ndarray compilation is the simplest: all operations that are linear-time in the entries are fused together into one pass. Non-linear operations, such as matmul, svd, eigh, and qr, materialize their arguments and call into the appropriate BLAS or LAPACK routine.

3.10.2 Stream compilation. Streams represent not-necessarily-materialized sequences of values. Every stream IR is either a producer (e.g. `MakeStream`), a consumer (e.g. `ToArray`, `StreamFold`), or both (e.g. `StreamFlatMap`). Streams are compiled to labels and jump instructions. Producers have five key attributes:

(1) A consumer provided *region* into which elements are placed.
(2) A Boolean indicating if this producer *allocates per element.*
(3) A label to which consumers jump to *produce an element.*
(4) A label, provided by the consumer, to which this producer jumps when an element is ready.
(5) A label, provided by the consumer, to which this producer jumps when the stream is empty.

Producers which allocate per element require their consumers to free their region after consuming the element. This introduces $O(N_{elements})$ region free operations and some consumers may need to copy elements into a different region.

In contrast to `java.util.Iterator<T>`, this code-generation pattern avoids boxing primitive values in the JVM and permits passing tuple or struct elements in registers rather than through the heap.

3.10.3 Aggregation and scan compilation. Any associative operation with a zero value could be an aggregator; however, Hail Query permits a more general structure: aggregators may consume one type, internally store another type, and return a third type. An aggregator is defined by four operations: `initOp` which initializes the aggregator state, `seqOp` which consumes one value, `combOp` which combines two aggregator states into one, and `result` which transforms the final aggregator state into the result.

The values consumed by aggregators, the aggregator states, and the returned value typically are stored in different regions.

A scan is like a prefix sum. Each row receives the aggregator result for all previous rows in the dataset.

MatrixIR compiles scans and aggregations into local folds or table aggregations. TableIR aggregations are only permitted in `TableAggregate` and scans are only permitted in `TableMapRows`. Table aggregation compiles into worker-side aggregations over each partition and one driver-side aggregation over the per-partition aggregator states. Table scans compile into worker-side aggregations over each partition, one driver-side scan over the partitions to compute the scan value at the start of each partition, and finally worker-side per-partition scans to compute the scan value for each row.

BlockMatrixIR supports aggregation in `BlockMatrixAgg`. It does not support scans.

3.11 Execution

Compilation produces a set of classes implementing the driver-side or worker-side portions of the pipeline. The main class is classloaded into the JVM and executed. When the driver need to execute a distributed operation, it serializes the relevant worker-side classes

and ships them to the workers. In the Apache Spark backend, the serialized code is serialized along with the RDD object. In the Hail Batch backend, the serialized code is written to and read from object storage. After the pipeline completes, the driver serializes the result to the client, again in a backend-specific way.

3.12 Sorting

Apache Spark's shuffle operator creates an all-to-all dependency. Every output partition relies on a (possibly empty) set of data from every input partition. To achieve this, every Spark worker sends a message to every other worker. The probability of the entire shuffle failing is equal to the probability of any one worker failing. Suppose the shuffle operation will take ten minutes and the probability of spot instance preemption is one in 1,000. Even a relatively small 100 VM cluster will fail to shuffle one out of ten times. Practical shuffle mitigation options are limited:

- Always use non-spot nodes for 5x to 1.1x additional cost.
- Be aware of every shuffle and change cluster composition for shuffles.
- Implement a spot-tolerant shuffle.

We simply implemented a spot-tolerant algorithm: a distribution sort. The upstream query is executed and written to object storage. Keys are sampled from the written data. Key intervals are chosen to evenly partition the data. Each source partition is split into at most 64 partitions. This inflates the number of partitions by a factor of at most 64. The maximum number of splits is configurable. Partitions are recursively split until all partitions are smaller than 512MB. This number is configurable. Each partition is sorted and written to cloud object storage. Downstream operations read and concatenate the sorted partitions.

3.13 Joining

Genomics data has a natural row key ordering, the genomic locus; therefore most datasets are stored in sorted order. For this reason, Hail execlusively uses ordered joins, thus saving the substantial cost of computing hashes and moving data round.

Most Hail join operations are binary: they combine two relational objects. Hail has one special join, the multi-way zip join, which joins an arbitrary number of relational objects using a tournament tree. The stream for each input relational object is converted into an iterator which is used by a leaf of the tournament tree.

These iterators objects are not generic and always return a pointer to a region-allocated value. They simplify the implementation of the tourament tree and the overhead of memory allocation is typically dwarfed by the work of the consumer of this multi-way join.

4 HAILTOP.BATCH

4.1 Most workflow systems designed custom languages.

4.2 In our experience, scientists are familiar with R and Python.

4.3 Exposing an API rather than a language empowers users to build the most-useful-to-them abstractions atop our API.

4.4 The simplest batch is the empty batch:

import hailtop.batch as hb b = hb.Batch(backend=hb.ServiceBackend()) b.run() The next simplest batch is a single job printing hello world: import hailtop.batch as hb b = hb.Batch(backend=hb.ServiceBackend()) j = b.create_job() j.command("echo hello world") b.run()

4.5 Python is used to automate creation of repetitive jobs.

4.6 The lack of higher-level "vector job" abstractions does mean particularly large batches (on the order of 10M jobs) can take tens of minutes to submit.

5 HAIL BATCH

5.1 Cost-metered, multi-tenant, spot, elastic, scalable, cloud compute engine.

5.2 The smallest unit of the system is a job.

- A container image execution. We pervasively assume the existence of bash (or another shell supporting the same semantics).
- Every job is a member of exactly one batch (described next).
- Configurable properties:
- Docker image name
- Bash script
- Core-memory ratio (1:7.5, 1:3.75, 1:0.9)
- Spot-vs-non-spot
- Disk request
- "Always Run"
- Normally a job is canceled if it depends on canceled jobs. "Always Run" jobs when the jobs on which it depends are complete or canceled.
- These are used to cleanup resources created by previous jobs.
- Region
- The cloud-specific region(s) in which a job may execute.
- Accessing data cross region boundaries is typically non-free.
- Timeout
- Maximum runtime of one attempt of the job.
- Cloudfuse mounts
- Zero or more buckets to mount using gcsfuse or blobfuse2.
- Input files
- A list of cloud storage URLs and local paths to which to download them.
- Output files

- A list of local paths and cloud storage URLs to which to upload them.

5.3 Attempt

- Due to network partitions, a job may execute more than once.
  - Each execution has a unique attempt id.
  - Users are charged for every attempt.

5.4 Batch

- A batch contains zero or more jobs.
  - Jobs may depend on zero or more other jobs. There must not be any cycles.
  - A batch is either running, complete, or canceled.
  - A batch starts complete.
  - Jobs only may be added to running or complete batches.
  - A job's dependencies must be created before it is.
  - Concurrent clients may add jobs to the same batch.

5.5 Billing project

- Every batch is a member of exactly one billing project.
  - Billing projects correspond to a single funded scientific project.
  - Any member of a billing project may view or cancel any batch in that billing project.

5.6 Database

- MySQL database. 4 cores. 16 GiB memory.
  - Three years, >8,000,000 batches, >34,000,000 jobs, 900 TiB disk.
  - Goals:
  - O(1) API operations: n_completed jobs in batch, total spend in batch, total spend in billing project, cancel batch.
  - Parallel updates of a job's information.
  - Consistent view of cost across jobs, batches, and billing projects.
  - What is in it?
  - Batches, VMs, attempts, billing.
  - Job metadata (e.g. dependencies, arbitrary key-value attributes, requested resources) and state (pending, running, canceled).
  - Job commands are stored in cloud storage because the bash script are sometimes large.
  - Rollups
  - The database is a single point of failure and also the main bottleneck.
  - Concurrent inserts or updates to the same row in MySQL either serializes or deadlocks the transactions.
  - In order to allow for parallelism, certain tables have a "token" column which is an integer ranging from 0 to some configurable number, inclusive.
  - Multiple transactions can insert or update rows in parallel.
  - For example, if a user_job_states table has an n_running_jobs column we would create this table: CREATE TABLE user_job_states (user_id INT NOT NULL, n_running_jobs INT NOT NULL DEFAULT 0, token TINYINT) When a job starts running we might execute: INSERT INTO user_job_states (user_id, n_running_jobs, token) VALUES(1, 1, FLOOR(RAND() 200)) ON DUPLICATE KEY UPDATE n_running_jobs = n_running_jobs + 1 When a job completes we might execute the same query with -1 in place of 1.
  - The probability of collision is given by the birthday problem.
  - We currently use 200 but this frequently leads to (retryable) deadlocks.
  - Need to balance the cost of aggregation for read against write parallelism.
  - Billing
  - Provides user feedback and informs the cost limiter when a billing project is overspent.
  - Resource use "rolls up" from attempt to job to batch to billing-project-&-user-&-date to billing-project-&-user.
  - MySQL triggers ensure the rollup is recalculated after every database change.
  - All tables use tokens except the attempt and job tables.
  - Active resources
  - Very large batches (e.g. a 10M job batch) cause the auto-scaler to aggressively add machines. We want a canceled batch to immediately stop triggers scale up, so we need O(1) update to the number of pending jobs.
  - The number of pending and running jobs per pool and per-batch-per-pool are stored in tables using tokens.
  - The autoscaler and scheduler use the pending and running jobs per pool.
  - Cancellation updates the per pool table based on the per-batch-per-pool table in O(N_POOLS) time.

5.7 Available resources:

- Pools
  - Three core to memory (in GiB) ratios: 1:7.5, 1:3.75, 1:0.9.
  - Spot and non-spot.
  - Disk space.
  - In GCP, disks are dynamically attached when more disk is needed.
  - In Azure, large disk requests fail.
  - Private instance
  - Effectively a shim over the underlying VM API.
  - Use-cases
  - A set of jobs otherwise amenable to standard core ratios but needing one initial or final job with a very large RAM ratio.
  - Users wanting GPUs (which are not currently supported in any pool).

5.8 Architecture

- MySQL database is a centralized state store.
  - Front-end is a horizontally scalable HTTP API for submission and monitoring.
  - Driver is a centralized scheduler and autoscaler.
  - Workers are VMs.
  - A separate auth service is a thin layer over cloud IAM.
  - Only relies on three simple, robust, and pervasive cloud APIs: VM, Object Storage, and Container Image Repository. (VERTICAL INTEGRATION!)

5.9 Single threaded Python driver can manage about 90 completing jobs per second.

- When a job completes, a worker sends an HTTP request to the driver.
  - The driver marks the job as complete in the database.

- To schedule another job, the driver makes an HTTP request to a worker.

- Scheduler is triggered either once a minute or when a job completes

- Scheduler fetches a subset of ready jobs from database for each users

- Cores are shared fairly among all users with ready jobs.

5.10 90 completing jobs per second

- If job latency is normally distributed with mean 5 minutes, then the driver can handle a steady state cluster of 27,000 jobs. If each job uses one core, that's a 1,700 VM cluster.

- Longer mean job latency permits larger clusters.

- We've operated clusters as large as 5,000 16 core VMs.

5.11 Cost-metering

- Due to the cost constraints on science, enforcing cost limits is important. We also do not want to kill a running job before the user exceeds a cost limit.

- Our approach leaves a lot to be desired but works well enough at our scale.

- Once a minute, a worker sends an HTTP request informing the driver of the list of still running jobs. In response, the driver updates the total cost for this job, the batch containing the job, and the billing project containing the batch.

- Every ten seconds the driver looks for billing projects that are overspent. All running batches in an overspent billing project are canceled which kills the running jobs.

- Assessment:

- A network partition in principle, allows arbitrary overspend.

- Under normal conditions, overspend could be as high as: 16 cores 70 seconds N_VMs usd_per_core.

- Total cost of jobs, batches, and billing projects monotonically increases. The system (and by extension the user) may underestimate cost but never overestimates it.

- In practice, overspend is usually less than a few dollars and limits are thousands of dollars.

5.12 Multi-tenant

Each user job runs in an OCI container started by crun using cgroupsv2.

Each job is in a private network. CPU is controlled by cpu.weight. Memory is controlled by memory.max and memory.high. Swap is prohibited.

Multi-tenancy both mitigates fragmentation (instead of two isolated users fragmenting their own clusters, the users share a single less fragmented cluster) and reduces latency for small & fast jobs (because, in practice, clusters are often fragmented).

5.13 Spot tolerant

A user's job runs to completion at least once.

Users must write jobs which are idempotent. The Hail Python API facilitates downloading inputs, computing on local files, and then uploading results, which ensures that jobs that do not mutate external resources are idempotent.

5.14 Elasticity

Every fifteen seconds, the driver calculates the demanded cores by region by pool by user and starts new instances to meet demand.

There are two scarce resources: cores already leased from the cloud and VM creation operations. The latter is limited by cloud-imposed limits on VM creation operations per minute.

The driver fairly shares these scarce resources across all users accounting for both jobs ready to run and jobs already running.

6 HAIL'S TENETS

6.1 Pervasive preference for spot instances

6.2 Vertical integration: Only rely on slow moving extremely robust third parties (e.g. the Linux kernel)

6.3 Scientists must not be subjected to operational work (cite Occupy the Cloud)

6.4 Client-Driver-Worker, adopted from PySpark, leverages the cloud but preserves the highly customizable local environment when data is small (i.e. transferred to the client).

6.5 Dataframes uber alles, SQL is not well suited to the needs of scientists (cite query languages for hep paper)

6.6 The only durable storage is cloud object storage.

7 FUNDING & ORGANIZATIONAL CHALLENGES

7.1 Contributions require caching in immense knowledge of the system, in practice we see few open source contributions.

7.2 Pharma and BioTech companies use and benefit from the system, but, as of publishing, none have contributed engineering resources or funding.

7.3 Elemental was shutdown due to lack of support for the developers

8 QUALITATIVE IMPACT ON SCIENCE

8.1 Hail Query and the Hail VDS (cite preprint) have been used to combine as many as 955,000 exomes (cite gnomAD v4) and 350,000 genomes (cite All of Us Echo release) into analysis-ready VDSes. Exomes or genomes are sequenced and aligned per-sample. Tertiary analysis prefers a variant-major (aka feature-major) dataset; therefore, producing a VDS from sequences is essentially a very large matrix transpose. See (cite preprint) for details on this process and the VDS format.

8.2 Hail Query (historically atop the Apache Spark backend, but work is shifting to the Hail Batch backend) has powered the analysis of several large sequencing studies including Centers for Common Disease Genomics, gnomAD v2, v3, and v4, All of Us 2023 Data Freeze, Schizophrenia Exome Sequencing Meta-analysis. In all, the repository, github.com/hail-is/hail has been cited 180 times.

8.3 Hail Batch enabled a multi-ancestry

than 10x faster than my previous setup. It literally took me less time to learn Hail Batch, port the workflow to it, and have it run on all data than it took to wait for it to finish elsewhere. I was calling tandem repeat expansions, so we now have some candidate diagnoses to pursue. The development experience is also incomparably better with local mode and Python tooling. All around, this feels like a game changer.

9 QUANTITATIVE EVALUATION

9.1 Single threaded

- Table decoding
  - I expect at least 50MiB/s.
  - Not great but a 16-core VM can manage 6.4 gigabits, 60% of 10gb
  - Matrix table decoding
  - VDS decoding
  - Table & matrix table sorting
  - Linear regression rows
  - Table group-by-aggregate
  - gnomAD frequency calculations

9.2 Multi-threaded

- As above.

10 QUANTITATIVE AND QUALITATIVE COMPARISONS

10.1 MySQL (as a representative of the traditional RDBMS)

10.2 sqllite

10.3 BigQuery

10.4 Apache Spark

- Designed for non-spot instances; ergo, fundamentally not cloud native.
  - Users must either operate their own cluster or use a proprietary serverless Spark platform (e.g. Dataproc Serverless, EMR Serverless).

10.5 DuckDB

10.6 AWS Fargate

10.7 Cloud Run

10.8 Azure Functions

10.9 Numpywren

10.10 Ibis

- Their Python API/DSL looks more like ours

11 FUTURE WORK

11.1 differentiable programming

11.2 import/export Arrow for zero-transcoding data transfer from one process to another.

11.3 Columnar storage

- 50 MiB/s is peak single-threaded decode speed, largely due to the amount of instructions necessary to decode, for example, variable length integers or homogeneous arrays of structures.
  - Columnar storage should enable much higher bandwidth decoding and encoding.
  - Pipelines only reading the genotype field (typically 4 bytes out of 10s) tend to be bottlenecked by the network due to the vast amount of unnecessary data.
  - Repetitive fields should benefit from run-length compression.

11.4 Vectorized processing

11.5 Replace SQL database

- Bad things:
  - Token scheme required to allow parallel updates to hot rows
  - Must write in MySQL SQL rather than Python, C++, or Java.
  - No event queue or other way to notify another process of changes.
  - Not sure what's better though.

11.6 Tighter cost controls

- A better solution would have each worker lease, say, five minutes worth of the cost of a job.
  - Ensures no overspend but would sometimes cancel or prevent from running an otherwise acceptably costing job.

CollectDistributedArray should support dependencies between contexts. This allows for DAGs necessary to express distributed linear algebra (cite numpywren).

A "distinctly keyed" dataset is one where there is at most one row per key value. Hail often works with foreign files and thus has no statistics about the keys. Moreover, Hail does a poor job of opportunistically tracking this information. In theory this information admits faster joins (once a key is processed it is done) and permits warning a user that Hail's default "left join, distinct" join will elide information in the right-hand-side table.

Add TableMaterialize, MatrixTableMaterialize, BlockMatrixMaterialize nodes. These nodes are explicit statements that computing the dataset should be "cheap". The query planner is free to either implement these as a write-read to temporary storage or to eliminate them entirely. Moreover, the query plan can choose encodings best suited for a write-once-read-once.

- For example, sorting a partitioned dataset requires sampling the new keys and then distributing the data into new partitions. In order to avoid computing the upstream query twice, sort explicitly writes the dataset. For computationally cheap queries, the cost of I/O dominates. In this regime, reading, writing, and finally reading twice is twice as slow as just reading twice.

11.7 Faster storage.

- Cloud storage is impressively scalable but cannot rapidly serve a single large object to 5,000 VMs.
- Container image pulling suffers similar problems.
- Cloud storage operations are not free.
- We want to use a fast read-through cache for objects read by large numbers of Hail Query partitions.

11.8 Container image extraction is also shockingly slow.

- We would like to use a pre-extracted format amenable to lazy loading.
- Perhaps a torrent style distribution system.

12 ACKNOWLEDGMENTS

NEU: DVH, Olin, Matthias, Aaron Turon, Dimitrios Vardoulakis, Dee, others? Harvard: Steve, Greg, Eddie, Scott, Dan, Andrew, others? Broad: Ben, Cotton, Mark, DMac, Hail team past and present, Konrad, KC, gnomAD, Andrea G, AUS, Michael F, Leo G, others? Funders: MSFT, Wertheimer, Stanley, NIH(?), NHGRI(?), others?

13 FIGURES

13.1 Job <- Batch <- billing project

13.2 VDS

Two MTs, one with column-sparse reference block runs and one with row-sparse variants

13.3 MT

The four pieces

13.4 hailtop.batch example

REFERENCES

[1] Marianne Abifadel, Mathilde Varret, Jean-Pierre Rabès, Delphine Allard, Khadija Ouguerram, Martine Devillers, Corinne Cruaud, Suzanne Benjannet, Louise Wickham, Danièle Erlich, Aurélie Derré, Ludovic Villéger, Michel Farnier, Isabel Beucler, Eric Bruckert, Jean Chambaz, Bernard Chanu, Jean-Michel Lecerf, Gerald Luc, Philippe Moulin, Jean Weissenbach, Annick Prat, Michel Krempf, Claudine Junien, Nabil G Seidah, and Catherine Boileau. 2003. Mutations in PCSK9 cause autosomal dominant hypercholesterolemia. *Nat. Genet.* 34, 2 (June 2003), 154–156.

[2] Christopher Berner. [n. d.]. Scaling Kubernetes to 2,500 nodes. ([n. d.]). https://openai.com/research/scaling-kubernetes-to-2500-nodes

[3] Roger S. Blumenthal, Seth S. Martin, and Parag H. Joshi. 1999. *The fascinating story of PCSK9 inhibition: Insights and perspective from ACC.* https://www.healio.com/news/cardiology/20150821/10_3928_1081_597x_20140101_08_1343305

[4] Maarten A Breddels and Jovan Veljanoski. 2018. Vaex: big data exploration in the era of gaia. *Astronomy & Astrophysics* 618 (2018), A13.

[5] Rene Brun and Fons Rademakers. 1997. ROOT — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 389, 1 (1997), 81–86. https://doi.org/10.1016/S0168-9002(97)00048-X New Computing Techniques in Physics Research V.

[6] Derek Caetano-Anolles. [n. d.]. GVCF - Genomic Variant Call Format. ([n. d.]). https://gatk.broadinstitute.org/hc/en-us/articles/360035531812-GVCF-Genomic-Variant-Call-Format

[7] Global Alliance for Genomics & Health. [n. d.]. The Variant Call Format Specification VCFv4.3 and BCFv2.2. ([n. d.]). https://samtools.github.io/hts-specs/VCFv4.3.pdf

[8] Apache Foundation. [n. d.]. Apache Arrow DataFusion. ([n. d.]). https://arrow.apache.org/datafusion/

[9] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, and Gustavo Alonso. 2021. Evaluating Query Languages and Systems for High-Energy Physics Data. *Proc. VLDB Endow.* 15, 2 (oct 2021), 154–168. https://doi.org/10.14778/3489496.3489498

[10] Stephen S Hall. 2013. Genetics: a gene of rare effect. *Nature* 496, 7444 (April 2013), 152–155.

[11] Hashicorp. [n. d.]. The Million Container Challenge. ([n. d.]). https://www.hashicorp.com/c1m

[12] Tony Hey. 2009. *The fourth paradigm: Data-intensive scientific discovery.* Microsoft Research, Chapter 1.

[13] Lawrence Livermore National Laboratories. [n. d.]. Hydrogen. ([n. d.]). https://github.com/LLNL/Elemental

[14] Benjamin M Neale. [n. d.]. Blended-Genome Exome. ([n. d.]).

[15] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[16] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya Parameswaran. [n. d.]. Towards Scalable Dataframe Systems. *Proceedings of the VLDB Endowment* 13, 11 ([n. d.]).

[17] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw.* 39, 2, Article 13 (feb 2013), 24 pages. https://doi.org/10.1145/2427023.2427030

[18] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[19] Eric Sigler and Benjamin Chess. [n. d.]. Scaling Kubernetes to 7,500 nodes. ([n. d.]). https://openai.com/research/scaling-kubernetes-to-7500-nodes

[20] Tyler M Smith and Robert A van de Geijn. 2019. The MOMMS family of matrix multiplication algorithms. *arXiv preprint arXiv:1904.05717* (2019). https://arxiv.org/pdf/1904.05717