# Introducing Hermes: Executing Clinical Quality Language (CQL) at over 66 Million Resources per Second (inexpensively)

Angelo Kastroulis,‡ Paolo Bonfini,‡ Anastasios Litsas, ‡

‡*Carrera Group, Inc.*

*nature@carrera.io , angelo@carrera.io*

1 November 2022

**ABSTRACT**

Clinical Quality Language (CQL) has emerged as a standard for rule representation in Clinical Decision Support (CDS) and Electronic Clinical Quality Measurement (eCQM) in healthcare. While open-source reference implementations and a few commercial engines exist, there is still a market need for high performance engines that can execute CQL queries on the scales of millions of patients. We introduce the Hermes engine as the world's fastest commercial CQL execution engine.

**Key words:** CDS – eCQM – Clinical Rules – CQL – FHIR

**CONTENTS**



## 1 INTRODUCTION

Clinical Decision Support (CDS) provides the right information to the right decision maker *at the right point of decision time* [2]. Electronic Clinical Quality Measurement (eCQM), on the other hand, is a mechanism for measuring and assessing the outcomes and process *after the fact* [1]. CDS informs *before* something happens, and eCQM measures *what happened*. Thus, they are two sides of the same process.

Computationally, CDS and eCQM are somewhat fundamentally at odds. CDS requires low-latency for intervention at the point of care, while eCQM requires execution at very high Throughput for measurement of clinical results. eCQM rule authoring relies on *a priori* knowledge in their creation. One may be tempted to think that the time it takes to execute against a population is irrelevant in a measurement scenario, however the reality is that computations on conventional systems can take days. It would be far more advantageous to execute in seconds, allowing organizations to inspect and adapt for better outcomes. In the time between reporting periods, a variety of technical, organizational, and care changes may have come into play. The results of the reports become stale and unactionable almost as soon as they are computed.

An ideal approach would allow a single solution to compute eCQM *and* replace aging rules systems (such as prior authorization). Custom rules could be written in CQL or another language (such as SQL) and run on the same platform.

Big data has been characterized in terms of (at least) 4 'V's — *volume*, *velocity*, *variety*, and *veracity*. This is especially applicable to health data [10]. Big data solutions require a high level of computer science expertise to operate effectively. The healthcare informatics requires a high level of expertise in the domain and relevant standards. That presents a challenge to Health IT experts who must possess both skills to build expensive systems. Health IT-specific methods of encoding rules such as CQL have also emerged, but they are not inherently able to operate at the scale of big data solutions. Thus, both of these approaches suffer from a disparity between the knowledge required to operate each of them.

CQL has proven effective in quality reporting initiatives in the United States, as well as in various clinical settings [15, 7, 8]. But, CQL only works with structured data, while around 80% of data is unstructured [4, 17]. An ideal solution would leverage the capability of big data tools to allow options for unstructured data and machine learning.

The Hermes engine bridges the gap between the domains by compiling CQL into code that is executable on big data and streaming pipelines, while optimizing and encoding health-specific knowledge directly into the computations, allowing superior performance. Moreover, by allowing for a variety of execution runners, Hermes also decouples from the underlying technology.

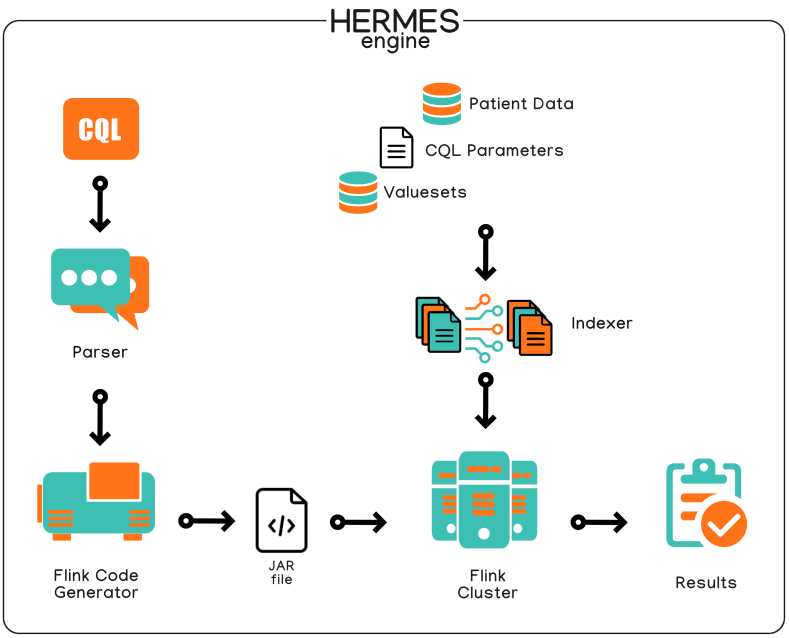


**Figure 1.** Schematic representation of the rationale of the HERMES engine.

## 2 BACKGROUND AND RELATED WORK

Rule-based approaches (such as RETE) require that rules and data fit into working memory. As rule sets and data increase, the amount of memory required increases more than linearly (a phenomenon known as "memory explosion") [9]. Eventually, the working set becomes sufficiently large and the system breaks down. Similarly, small, fast interpreters fail when data sets become exceedingly large [19, 14]. Such algorithms are not appropriate for execution against large populations, making them ineffective on eCQM and rules executed against many patients.

Technologies in the big data ecosystem such as Apache Spark, Kafka, Flink, and Cassandra execute computations on large sets of data by partitioning the data and moving the computation to the data [4, 19, 14]. Data movement has become the new bottleneck and open research shows that systems that carefully minimize movement perform better [11]. Apache Flink has emerged as one of the fastest computational platforms available [18, 5].

Decoupling the underlying execution technology from the query engine (as it is done by Apache Beam [16]) allows for a variety of "runners". In addition to decoupling, the next generation of systems will interpret queries by their intent and optimize accordingly [13]. These systems should not only work with batch loads, but also streams. Big data streams are generated continuously at unprecedented speed, yet allocating the cloud resources necessary to make them successful has emerged as a major research problem [12].

Ideally, one would compile CQL into some code that can run on top of another widely-accepted computational platform (runner). Such a solution would also meet the needs of optimizing stream *vs* batch workloads. Additionally, purpose-built systems are faster than general purpose systems because they can exploit certain characteristics of their domain that general-purpose systems cannot. For example, Health data encodes semantic meaning within it and has standardized components which are immutable and hence represent good candidates for optimization (e.g., a versioned, standardized Value Set). Whereas a general-purpose system is not able to infer this from a generic structure.

**Table 1.** Table of Workloads. Columns represent *millions of records* for each Workload id (e.g., Workload 100M has 100 million patients and 2.25 billion resources in the set).

| | 1M | 10M | 50M | 100M |
|---|---|---|---|---|
| Patients | 1 | 10 | 50 | 100 |
| Conditions | 2 | 20 | 100 | 200 |
| Encounters | 3 | 30 | 150 | 300 |
| Medications | 5 | 50 | 250 | 500 |
| Procedures | 1 | 10 | 50 | 100 |
| Observations | 5 | 50 | 250 | 500 |
| Coverages | 5.5 | 55 | 275 | 550 |
| Total | 22.5 | 225 | 1,125 | 2,250 |

While open source CQL execution engines do exist, they suffer from serious performance problems because they are not horizontally scalable. Horizontal scalability is essential in reducing costs and increasing performance since vertical systems quickly run into high costs and hardware limitations [4]. Those engines' only recourse is micro-optimization, but that cannot achieve improvements of orders of magnitude.

A purpose-built CQL compiler/execution engine that has the characteristics mentioned above would be far superior to other systems. In the following section we will demonstrate how HERMES characterizes such a system by means of realistic workloads.

## 3 METHODS

Our HERMES engine is structured as depicted in Figure 1: the CQL query is parsed in order to provide a proper input for the Flink Code Generator, which creates the code that is in turn used to spun the Flink Cluster. This cluster does the actual heavy job by utilizing as an input the CQL Parameters, the Valuesets, and the Patient Data.

To test HERMES' computational ability, we decided to assess its performance when evaluating a quality measure chosen from the

HEDIS 2022 FHIR set. The HEDIS measures are a set of standard rules that hospitals and payers must report annually. We selected BCS for its diverse workload (complex temporal logic) and variety of resources consumed, but was also a fair representation of the "average" complexity. In order to assess the engine under a variety of situations, we considered different *workloads*, as shown in Table 1. Moreover, we also considered different *configurations*, corresponding to different hardware capabilities (see Appendix A). Therefore, in practice, we run a grid of tests where the engine was executed for each combination of *Workload* and *configuration*.

```
1   define "Numerator":
2     exists ([Observation: "Mammogram"] m
3         where m.effectiveTime
4           ends during <time window>)
5
6   define "Denominator":
7      AgeInYearsAt(date from
8       end of <time window>
9     )in Interval[52, 74]
10      and Patient.gender.value = 'female'
11      and <Member coverage in
12          time window with only allowable gaps>
13
14  define "Exclusions":
15    exists([Encounter: "Hospice Encounter"] e
16      where e.status='completed'
17          and <in time window>)
18    or exists([Procedure: "Hospice Intervention"] p
19      where p.status='finished'
20          and <in time window>)
21    or exists([Procedure: "Mastectomy"] p
22      where p.status='completed')
23    ...
24    or exists([Condition: "Absence of Breast"] c
25      where c.prevalencePeriod
26          <in time window>)
27
```

Listing 1: Example of implementation for the BCS CQL

Hermes has both a **streaming** and **batch** mode, but tests were performed in batch mode to determine computing power and Throughput achievable on 100 million patients and 2.25 billion resources (observations, conditions, medications are all resources). The engine must have sufficient work so that it has substantial computation (we want it to work intensively). That is measured in two dimensions: 1) *difficulty* of the computation and 2) large enough *selectivity*. Selectivity refers to the number of matches in a computation (in this case, the number of matching patients in the set). Hermes excels at eliminating non-matching records using its internal indexing scheme and Flink's partitioning and worker semantics. A more naive approach would require each record to be unpacked, deserialized, and computed only to find it does not match. Hermes recognizes immediately that a record does not have a matching code, for example. A set with small selectivity (100 million patients with 1% matches) would return too quickly, playing into the Hermes's strength. While this would show a massive performance advantage over other systems (three or more orders of magnitude) — and low selectivity is a typical scenario — it wouldn't tax the system sufficiently.

What makes Hermes exceptionally well suited for execution on large sets is that it rewrites computations into relational algebra, then performs set operations on them (complete with predicate pushdown and other optimizations). In order to see the effect of such optimizations, it is important to have exaggeratedly high selectivity. We chose an arbitrary value of 20% for our test scenarios.

**Let's explore an example of the engine's behavior when applied to a specific CQL search.** A common function in CQL is *retrieve*. Retrieve is essentially a join between ValueSets and another resource. For example, line 2 of Listing 1 is a join between *Observations* and the *Mammogram Valueset* on their respective `code` properties. The Mammogram ValueSet can contain hundreds of breast cancer codes. In relational algebra, this would be known as a *left outer join*. In order to perform this operation, Flink would need to load the smaller relation (Valuesets) into memory and stream the larger (Observations).

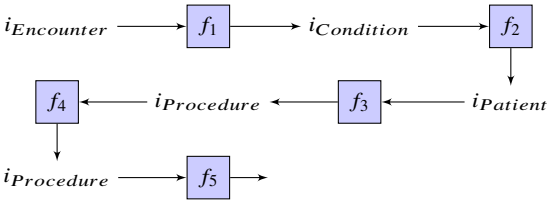


**Figure 2.** CQL `Exclusions` executed as written by an interpreter, i.e., the scans are executed in sequence for some function $f$ given an input $i$.

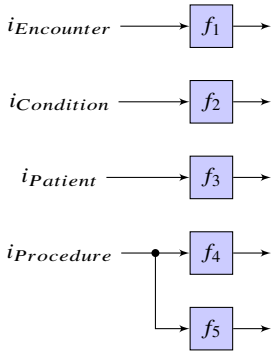


**Figure 3.** CQL `Exclusions` optimized for parallelism for the same function $f$ given input $i$. Notice that the graph for Procedure has multiple ramifications because Procedure may be scanned multiple times.

Still, all matches would need to be remembered, which, over tens or hundreds of millions of matches, consumes gigabytes of RAM.

Hermes' VS HyperCache feature eliminates this by creating a join operation that pre-compiles Valusets and distributes them to each Flink Taskmanager, substantially reducing the memory footprint by **a factor of** 5–10×.

### 3.1 Configurations

We run our test adopting different system setups on Amazon Web Services cloud (AWS), in order to explore the dependency of the engine's performance on the specifics of the hardware. In this context, a "*configuration*" represents the set of parameters which uniquely describes a given setup. A variety of configurations were explored, possessing different computational cores, virtual memory, etc. — see configurations details in Appendix A.

We chose EC2 instances that were optimized for compute and memory. We then varied the amount of resources given to each worker and varied the number of workers.

A *taskmanager* is an independently operating worker node that can run either on the same machine as other worker nodes, or on independent machines in a cluster. Taskmanagers are coordinated by a *jobamanger*, whose primary responsibility is to distribute and collect work. Taskmanagers can themselves divide work into smaller components called *slots*. *Parallelism* refers to the number of slots available to each Taskmanager. Thus a cluster may have hundreds or thousands of slots (e.g., 10 Taskmanagers with 10 slots each yield a total of 100 slots cluster-wide).

### 3.2 Rules

For this test, we selected the Breast Cancer Screening measure (BCS). A common convention for CQL quality measurement is to divide the top level rules into three main rules: `Numerator`, `Denominator`, and

`Exclusions`. This specific nomenclature arises from the fact that quality measurements are intended to be reported as a ratio of the number of patients in which a target action was performed, over the size of the population eligible for that action. In other words, the proportion of patients that *had* something done to those that *should* have.

**For example, in the case of the breast cancer screening (BCS) introduced above** (see Listing 1):

- `Numerator` represents the number of women who had at least one mammogram within the previous 27 months (the time window),
- `Denominator` is the number of women eligible (between the ages of 52 and 74 who visited the office),
- `Exclusions` are the women who shouldn't be accounted for, presumably because they are already under treatment or a care plan (e.g., women who had a mastectomy, are in hospice, have advanced illness, are in long term care, etc.).

Instead of reporting the total ratio, however, the current state of the art is to pre-compute values atomically, allowing downstream systems to total them in a report.

In BCS, `Numerator` is a set operation that scans observation codes within a time window. Since Hermes indexes both time and codes, the operation computes extremely quickly.

`Denominator` adds the complexity of a join between resources (*coverage* and *computing age*). The *coverage* computation is particularly expensive since it aggregates coverage rows grouping, reducing, and eliminating entries in search of gaps. `Exclusions` is the most complex and must scan a number of tables: Procedures (a few times), Medications, Encounters, Conditions, Patients, and Observations. Ultimately, `Exclusions` performs a union and distinct operations on all of those tables for a positive match. Distinct Union is notoriously expensive because a system must remember the keys of the entire table. Hermes' VS HyperCache was designed overcome this scenario.

Executing the three operations together, while intensive, may also present an opportunity to reuse computation. For example, if a system could recognize that Procedures is scanned many times, it could combine the operations for substantial performance savings.

Most interpreter-based execution engines will translate the CQL to Expression Logical Model (ELM) and execute it sequentially as shown in Figure 2. ELM is a logical specification standard (an Abstract Syntax Tree), but it is not compact enough to execute directly in a performant manner. A better approach is shown in Figure 3: the functions related to the resource scans could all be done at the same time on multiple slots. Note the $f_5$ and $f_6$ could actually operate on Procedures at the same time (or perhaps even become the same computation).

### 3.3 Workload: Synthetic Data Generation

So far we have presented the CQL rules and explained their rationale, but how were our tests performed, *in practice*?

First, we needed a dataset on which to apply the rules. For this purpose, we needed be to run Hermes against a synthetic dataset modeled over the structure of the fields expressed in the BCS measure (see Figure 3.3). This artificial dataset shall satisfy the following 2 properties. First, it contains between millions to hundreds of millions of synthetic patients; this is desirable because a large dataset provides a stress test that is sufficiently challenging for our engine. Second, it should allow us to control the volume of matches produced by executing a given search (namely, the one in Listing 1). This is an extremely significant parameter, because an engine's workload scales, at the first order, with the number of expected matches. One key factor in a successful engine design is the ability to quickly discard irrelevant information, focusing on the potential matches in order to save computational time. What we sought was a database containing plenty of irrelevant information which would overload a sub-optimal engine. As previously mentioned, we aimed at having about 20% matches.

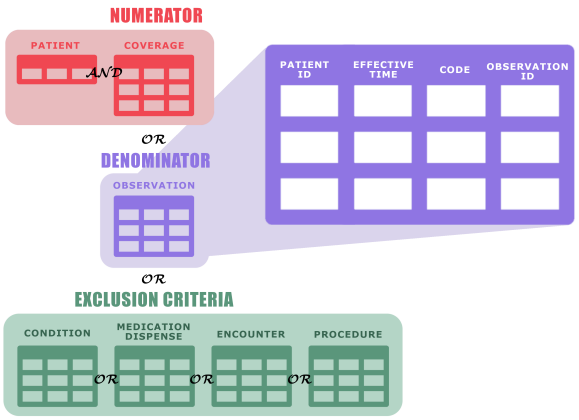


**Figure 4.** Structure of the synthetic dataset used for the performance test. The figure shows the entries for a single patient, which pertain to different tables, i.e., only the Patient table contains one single row per patient, all the other tables may contain multiple rows indexed by the same patient ID. The magnified table shows an example of the possible column contents. The different tables are organized in 3 major groups: Numerator, Denominator, and Exclusion Criteria. The 'And' and 'Or' logical relations signify which conditions shall be satisfied for a match to happen.

The generation of artificial structured (tabular) data is an active field of research [e.g. 6], which includes advanced techniques such as generative-adversarial networks [GANs; e.g., 3]. Yet, to our knowledge, there exists no simple tool to generate data which can guarantee a preset number of matches given a [CQL] rule — the most important structural factor we had to account for. In fact, for the purposes of this test, it is not necessary for the data to be sampled according to a *realistic* distribution. It is more important to determine whether results fell within/outside the matching ranges set by the search rules.

Therefore, we ultimately resorted to creating our own data with a more direct approach. Given that we wanted a 20% match rate, we simply sampled random dataset entries such that they fell within/outside the search ranges accordingly. We can categorize the synthetic entries into 2 types, according to this definition:

**valid** entry → within the *allowed* range
**in-valid** entry → outside the *allowed* range

where the allowed range is set by the rules (specified in §3.2). Notice, though, that a valid entry does not automatically yield 'a match'! Because of the database format of Figure 3.3, a match is returned whenever there is a match between the search query and *either* of the Denominator, Numerator, or Exclusion entries *as a whole*. More specifically:

- For the Denominator, 1 valid match is yielded when both the Patient and Coverage Tables are valid in all their entries

- For the Numerator and Exclusion, 1 valid match is yielded when either of their composing Tables is valid in all their entries

The generation problem then becomes:

> *How shall we distribute valid and in-valid entries across the dataset, so that they collectively yield to matches for 20% of the data volume?*

The number of data per patient are given by the sum of Denominator ($D$), Numerator ($N$), and Exclusion ($E$) data per patient. Note, though, that we want to generate a variable number of such data. Therefore, let's consider their *average* numbers: $\mu_C$, $\mu_N$, and $\mu_E$, and estimate the *data volume per patient*, $T_p$, as:

$$T_p = \mu_D + \mu_N + \mu_E \tag{1}$$

and, from there, derive the amount of *valid data per patient* $V_{T_p}$ (proportional to the number of computations that will result in a match for that patient), given a ratio $r$ of desired matches (e.g. 20%):

$$\begin{aligned} V_{T_p} &= r\, T_p = r\, (\mu_D + \mu_N + \mu_E) \\ &= r\, \mu_D + r\, \mu_N + r\, \mu_E \end{aligned} \tag{2}$$

Let's now recollect that Denominator, Numerator, and Exclusion are actually composed by multiple tables, each having a set of entries (their columns). E.g., Numerator is composed of a single *Patient* ($P$) table, and a $Coverage$ table, both of which shall host valid entries to return a match. We can therefore re-write Equation 2 as:

$$\begin{aligned} V_{T_p} &= r\, \mu_D + r\, \mu_N + r\, \mu_E \\ &= r\, (P \times \mu_C) + r\, \mu_N + r\, \mu_E \\ &= r\, (\mathbb{1} \times \mu_C) + r\, \mu_N + r\, \mu_E \end{aligned} \tag{3}$$

where $\mathbb{1}$ is just a placeholder to remind us that even if we have 1 single *Patient* table per patient, that must be actually sampled. We can redistribute $r$ between *Patient* and *Coverage* tables:

$$V_{T_p} = (\sqrt{r}\, \mathbb{1} \times \sqrt{r}\mu_C) + r\, \mu_N + r\, \mu_E \tag{4}$$

We can read this equation as:

> " *At generation time, we have to sample a matching Patient table row $\sqrt{r}$ of the times, a matching Coverage table row $\sqrt{r}$ of the times, and any of the N or E tables rows r of the times".*

We are just left with defining what is a "*matching table row*", but that is trivial: a row whose entries are all within the ranges allowed by the rules. In other words, when a row is to be generated:

| | | |
|---|---|---|
| as matching | → | all of its entries are sampled as valid |
| as *not* matching | → | at least 1 of its entries is sampled as in-valid. |

With this rationale, we created 4 synthetic datasets of different sizes, hosting 1, 10, 50, or 100 million (respectively labelled $1M$, $10M$, $50M$, and $100M$) synthetic patients[1]. In the remainder, we refer to these test datasets as '*Workloads*'.

### 3.4 Test Setup

We chose Flink as the underlying runner and compiled BCS to a jar file, distributed on each cluster configuration. We stood up several cluster configurations (see Appendix A) and executed the various Workloads (see Table 1) against each configuration.

We also tested the effect of a variety of index formats (Apache Avro, Orc, and Parquet). Finally, we compared executions with VS HyperCache enabled vs disabled.

[1] Note that, for consistency, we made such that the larger datasets *include* the smaller ones.

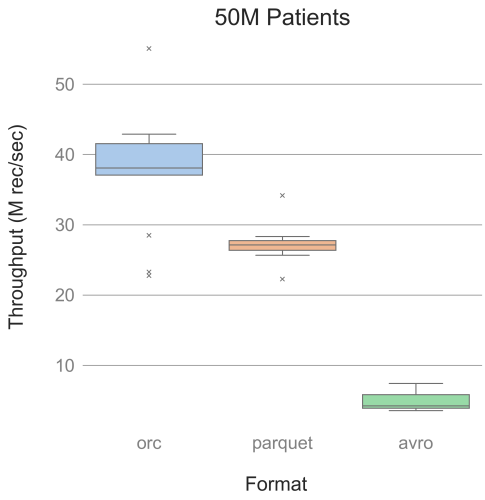


**Figure 5.** Comparing Throughput of the CQL computation using Apache Avro, Parquet, or Orc file types, the 50 million patient Workload illustrates the relative performance indicative of all Workloads we tested. In every Workload, Apache Orc was the fastest.

## 4 OBSERVATIONS AND TEST RESULTS

### 4.1 Index Format

Hermes is capable of ingesting data in a variety of formats. During the compilation phase of CQL, it identifies which elements are referenced and indexes them (such as a patient's age, or a medication's code). These indexes are used at runtime rather than the original FHIR data for computations. This reduces the size of the data scattered across the cluster by two or more orders of magnitude. The index format can be any format specified through format connectors. We tested Apache Avro, Parquet, and Orc. Apache Avro has gained popularity through its use in Apache Kafka and has several advantages (such built-in schema definition, version evolution, and serialization/deserialization performance). Apache Parquet and Orc are columnar formats which allow advanced optimizations such as predicate push down.

As can be seen in Figure 4, Apache Orc was the fastest in all cases, so from this point forward, we will exclude the other formats in discussion for simplification.

### 4.2 Workloads

Hermes' advantage as a Throughput-optimized computational engine can be observed in Figure 4. As workloads increase (10M, 50M, and 100M), so does the Throughput, correspondingly (16, 55, and 66 million resources per second). The advantage of cost amortization of dividing and distributing work among a cluster pays off with more data. But, it also shows the advantage of applying predicates intelligently to reduce the amount of data moving between steps in the topology as early as possible. Engines commonly apply computations across the patient space, thus requiring traversal of all of the data because they never reduce the set. Hermes rewrites the computation to work across the resources *first* (medications, conditions, observations, etc.) reducing the work to be done at later stages.

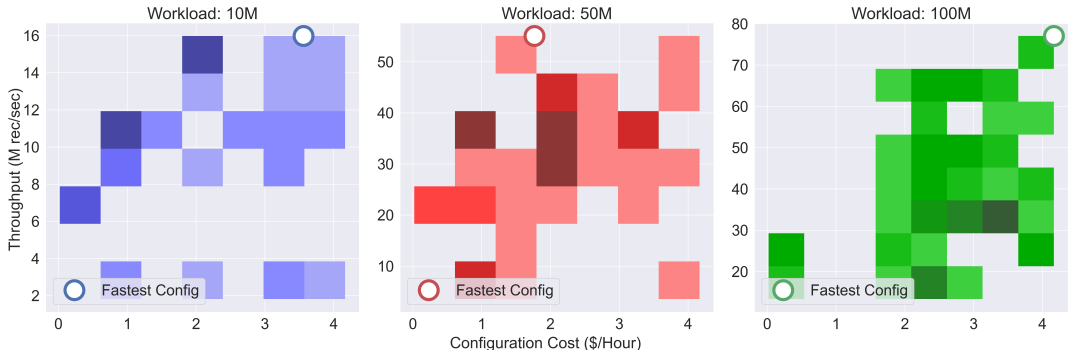


**Figure 6.** Comparing Throughput (millions of resources per second) and Cost (dollars per hour) of Workloads. From *left* to *right*, we show the results for increasing Workloads. The color gradient indicates the number of configurations that occupy that locus of the plot. The data point indicates the best-performing configuration.

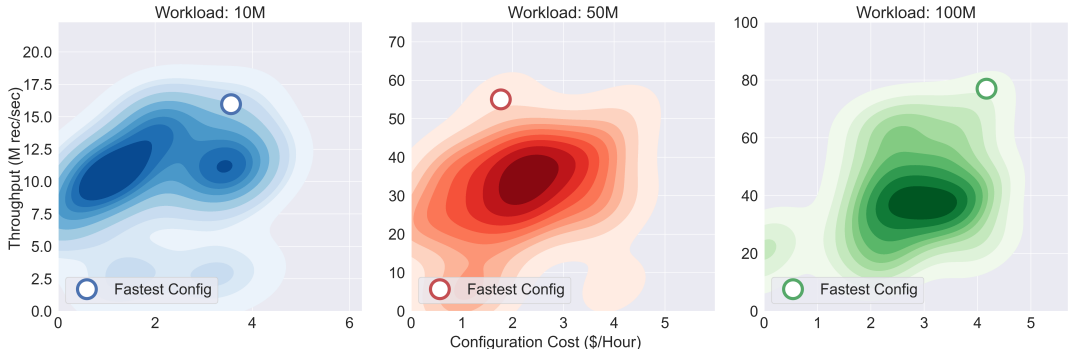


**Figure 7.** Comparing Throughput (millions of resources per second) and Cost (dollars per hour) of Workloads. From *left* to *right*, we show the results for increasing Workloads. The color gradient indicates the density of configurations that occupy that locus of the plot, obtained via a Kernel Density Estimator (KDE). The data point indicates the best-performing configuration.

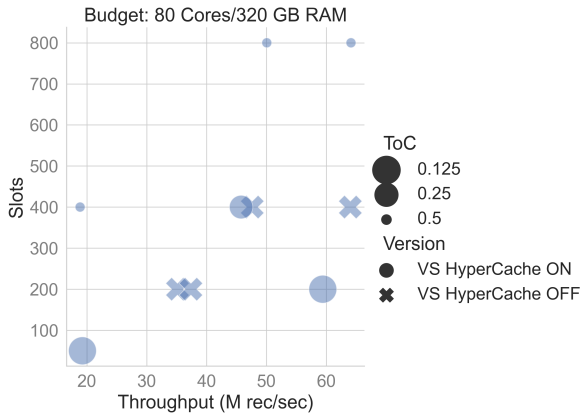


**Figure 8.** Comparing the distribution of resource budgets. The data points are size-coded based on the the ratio of Taskmanagers over Cores ($ToC$), which implies a larger (lower) distribution of work for higher (lowerBecause) values. The two different marker types refer to the activation (or not) of the HyperCache feature.

Thus, the more records to process, the faster Hermes gets.

### 4.3 Resource Arrangement: Budgets and Slots

Within a specific *budget* of resources (a given CPU, RAM, or other constraints), there are still decisions to be made as to how best to use them. For example, multiple configurations use a total cluster-wide consumption of 80 cores and 320 GB of RAM. How should such resources be distributed? Should we favor fewer, larger Taskmanagers (10 with 8 cores and 32GB of RAM each) or many, smaller ones (40 with 1 core and 8GB of RAM each)?

We label "*ToC*" this relationship of Taskmanagers to resources (specifically, cores). A smaller ToC indicates relatively fewer, larger Taskmanagers, while a larger ToC indicates relatively more, smaller Taskmanagers.

*Slots* are the logical execution cores that are parallelized in a computation. The total number of slots in a cluster are computed as

$$n_{slots} = n_{parallelism} \times n_{taskmanagers}$$

Is it better to have a single slot per Taskmanager, or to have multiple slots per Taskmanager?

Figure 4 shows that the way we distribute the resources on the cluster (ToC) and the number of slots cluster-wide are related in the way they affect Throughput. A smaller number of slots (200) performs well (about 56 million resources per second) on a smaller ToC (larger Taskmanagers), whereas increasing the number of slots (to 800) allows the larger quantity of smaller Taskmanagers to do more work. Note that at the extreme end of the chart lie the two fastest configurations, namely:

(i) 800 Slots, 0.5 ToC, VS HyperCache On
(ii) 400 Slots, 0.25 ToC, VS HyperCache Off

The first (with VS HyperCache enabled) allows for many more slots because the memory required for each is far less. The second could not run as many slots and Taskmanagers because each Taskmanager required at least 12GB of RAM. VS Hypercahce effectively reduced the amount of memory required per Taskmanager.

Many smaller Taskmanagers is preferable because the jobmanager may be able to schedule them for other rules sooner. This happens because each of them is doing relatively less work, and hence they complete more quickly individually. VS HyperCases proves effective in reducing the memory footprint, allowing for higher parallelization.

### 4.4 Cost

One key element of the performance analysis is estimating the *cost* of the hardware setup used to execute an engine's search. This is trivial because given a task of any engine — no matter how inefficient such an engine may be—, there will always be a hypothetically more expensive setup that can boost that engine's performance.

The cost of the configurations we tested was estimated by adopting the price-per-hour ($CpH$) of the cloud service that was utilized (Appendix B), and re-normalizing it by the resources that we actually allowed Flink to access (see Appendix A). For example, Image Type m5.24xlarge costs $4.08 per hour with its 384 GB RAM and 96 cores, but e.g. configuration C18B — based on the aforementioned m5.24xlarge — uses only 16 GB RAM and 2 cores.

We can think of the $CpH$ of a given Image Type $i$ as the weighted sum of its CPU and RAM resources times the unit cost $u_i$:

$$CpH = (\alpha N_{c,i} + N_{r,i}) \times u_i \tag{5}$$

where $N_{c,i}$ and $N_{r,i}$ are the CPU and RAM units in Image Type i, respectively, and $\alpha$ is the weighing factor that accounts for the different cost of a unit of 1 GB of RAM and 1 core. This weighting factor can be calculated by comparing the market cost of a 'building block' of RAM (64 GB) and one of CPU (8 cores); we roughly estimated $\alpha \sim 6$.

Notice that $u_i$ is the unit cost per CPU or RAM resource, irrespectively, for Image Type $i$. So we can reverse the above formula, to get the unit cost $u_i$ (which is all we need to estimate the cost of a configuration):

$$u_i = \frac{CpH}{\alpha N_{c,i} + N_{r,i}} \tag{6}$$

Following on our example for Image Type m5.24xlarge, Appendix B gives $CpH_i$ = $4.608/h, $N_{c,i} = 96$, and $N_{r,i} = 384$, hence we obtain $u_i = \$4.608/(6 \times 96 + 384) \sim \$0.005/h/resource$.

With $u_i$ at hand, we can calculate the cost-per-hour for any configuration ($CpH_{config}$) derived from Image Type $i$, which utilizes the actual resources $\bar{N_c}$ and $\bar{N_r}$[2]. simply as:

$$CpH_{config} = (\alpha N_c + N_r) \times u_i \tag{7}$$

For example, let's calculate $CpH_{config}$ for configuration C18B, which is based on Image Type m5.24xlarge. From Appendix A we have $N_c = 2$, $N_r = 16$, and using the previously calculated $u_i$, we obtain $CpH_{config} = (6 \times 2 + 16) \times \$0.005 = \$0.14\ /hour$.

As Figure 4 and 4 show, the fastest configuration was also one of the least expensive at 1/3 the cost.

[2] Do not confuse $\bar{N_c}$ and $\bar{N_r}$ with $N_{c,i}$ and $N_{r,i}$: the former refer to the configuration-restricted resources, the latter refer to the maximal capacity of Image Type $i$ that the configuration is built upon.

## 5 CONCLUSIONS

### 5.1 Conclusion

**Performance.** Hermes performs exceptionally well with large data sets. In fact, the more data it is given, the more records per second it can compute because the cost of distributing the workload is quickly recovered. It is also extremely efficient at computation sharing.
**Cost.** Hermes provides an excellent trade-off between top performance and cost, operating at approximately 66 million resources computed per second at less than $2 per hour. More aggressive pricing models (such as spot pricing) would cost far less ($0.89 per hour). While not specifically tested, the cluster creation time in a kubernetes-style deployment (as is supported by Hermes) allows new pods to be spun up nearly instantaneously, further reducing costs by resource-sharing.
**Resource utilization.** Internal features such as predicate push down and VS HyperCache, proved extremely effective at reducing memory and compute consumption. Hermes is highly parallelizable, working best with many small workers.
**Eco-system friendly.** Finally, Hermes is effective at using a variety of big data technologies such as Apache Flink, Orc, Kafka, Parquet, etc., and orchestration technology such as Kubernetes.

### 5.2 Future Work

The goal of this test was *not* to tune Hermes, but rather understand its performance characteristics and trade-offs. Undoubtedly, additional tuning (such as determining the optimal partitioning of data for maximizing parallelization) would have lead to higher Throughput or lower cost. What is the trade-off between partition size and Throughput?

Future tests should also measure the effect of executing **multiple rules simultaneously**, and the increased global Throughput resulting from combining rules on the same data in the same execution job. Additionally, durably caching **intermediate state** in future versions of Hermes is also an area for improvement. Could performance be gained by analyzing the entire *rule space* and determining which portions of data could be shared between them, thus amortizing the cost of caching to increase Throughput?

## REFERENCES

[1] 2013. What are clinical quality measures? (Jan 2013). https://www.healthit.gov/faq/what-are-clinical-quality-measures
[2] 2018. Clinical decision support. (Apr 2018). https://www.healthit.gov/topic/safety/clinical-decision-support
[3] Masoud Abedi, Lars Hempel, Sina Sadeghi, and Toralf Kirsten. 2022. GAN-Based Approaches for Generating Structured Data in the Medical Domain. *Applied Sciences* 12, 14 (2022), 7075.
[4] Ahmed Hussein Ali. 2019. A survey on vertical and horizontal scaling platforms for big data analytics. *International Journal of Integrated Engineering* 11, 6 (2019), 138–150.
[5] Sonia Bergamaschi, Luca Gagliardelli, Giovanni Simonini, and Song Zhu. 2017. BigBench workload executed by using Apache Flink. *Procedia Manufacturing* 11 (2017), 695–702.

[6] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. 2021. Deep neural networks and tabular data: A survey. *arXiv preprint arXiv:2110.01889* (2021).

[7] Pascal S Brandt, Jennifer A Pacheco, Prakash Adekkanattu, Evan T Sholle, Sajjad Abedian, Daniel J Stone, David M Knaack, Jie Xu, Zhenxing Xu, Yifan Peng, et al. 2022. Design and validation of a FHIR-based EHR-driven phenotyping toolbox. *Journal of the American Medical Informatics Association* 29, 9 (2022), 1449–1460.

[8] Mark L Braunstein. 2022. Public and Population Health. In *Health Informatics on FHIR: How HL7's API is Transforming Healthcare*. Springer, 347–379.

[9] Yi Chen and Behzad Bordbar. 2016. Dress: A rule engine on spark for event stream processing. In *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. 46–51.

[10] Anil Jain. 2022. The 5 V's of big data. (Jun 2022). https://www.ibm.com/blogs/watson-health/the-5-vs-of-big-data/

[11] Angelo Kastroulis. 2019. *Towards Learned Access Path Selection: Using Artificial Intelligence to Determine the Decision Boundary of Scan vs Index Probes in Data Systems*. Ph.D. Dissertation. Harvard University.

[12] Navroop Kaur and Sandeep K Sood. 2017. Efficient resource management system based on 4vs of big data streams. *Big data research* 9 (2017), 98–106.

[13] Martin L Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. 2011. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *Proceedings of the VLDB Endowment* 4, 12 (2011), 1474–1477.

[14] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[15] Binh-Phi Nguyen, Thomas Reese, Stefen Decker, Daniel Malone, Richard D Boyce, and Oya Beyan. 2019. Implementation of clinical decision support services to detect potential drug-drug interaction using clinical quality language. In *MEDINFO 2019: Health and Well-being e-Networks for All*. IOS Press, 724–728.

[16] Chat Room. 2020. Apache Beam. *system* 11, 17 (2020), 24.

[17] Joe Tekli. 2016. An overview on xml semantic disambiguation from unstructured text to semi-structured data: Background, applications, and ongoing challenges. *IEEE Transactions on Knowledge and Data Engineering* 28, 6 (2016), 1383–1407.

[18] Ilya Verbitskiy, Lauritz Thamsen, and Odej Kao. 2016. When to use a distributed dataflow engine: evaluating the performance of Apache Flink. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE, 698–705.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

**APPENDIX A: TABLE OF CONFIGURATIONS**

| ID | Image Family | Taskmanagers | Cores/TM | Ram/TM (GB) | Parallelism | Network BW | EBS BW |
|---|---|---|---|---|---|---|---|
| C1 | c6id | 2 | 1 | 12 | 10 | 12 | 10 |
| C2 | c6id | 5 | 1 | 12 | 10 | 12 | 10 |
| C9 | c6id | 2 | 4 | 12 | 10 | 12 | 10 |
| C10 | c6id | 5 | 4 | 12 | 10 | 12 | 10 |
| C5 | c6id | 2 | 1 | 24 | 10 | 12 | 10 |
| C13 | c6id | 2 | 4 | 24 | 10 | 12 | 10 |
| C13b | c6id | 2 | 12 | 24 | 10 | 12 | 10 |
| C3 | c6id | 10 | 1 | 8 | 10 | 37 | 30 |
| C3b | c6id | 10 | 1 | 15 | 10 | 37 | 30 |
| C6 | c6id | 5 | 1 | 30 | 10 | 37 | 30 |
| C4 | m5 | 10 | 1 | 32 | 10 | 25 | 19 |
| C4B | m5 | 10 | 1 | 32 | 20 | 25 | 19 |
| C4C | m5 | 10 | 1 | 32 | 5 | 25 | 19 |
| C16 | m5 | 10 | 4 | 32 | 10 | 25 | 19 |
| C16B | m5 | 10 | 4 | 32 | 20 | 25 | 19 |
| C16C | m5. | 10 | 4 | 32 | 5 | 25 | 19 |
| C17 | m5 | 10 | 8 | 32 | 10 | 25 | 19 |
| C17B | m5 | 10 | 8 | 32 | 20 | 25 | 19 |
| C17C | m5 | 10 | 8 | 32 | 5 | 25 | 19 |
| C19 | m5 | 10 | 2 | 32 | 10 | 25 | 19 |
| C19B | m5 | 10 | 2 | 32 | 20 | 25 | 19 |
| C19C | m5 | 10 | 2 | 32 | 5 | 25 | 19 |
| C18 | m5 | 20 | 2 | 16 | 10 | 25 | 19 |
| C18B | m5 | 20 | 2 | 16 | 20 | 25 | 19 |
| C8 | m5 | 20 | 1 | 16 | 10 | 25 | 19 |
| C8B | m5 | 20 | 1 | 16 | 20 | 25 | 19 |
| C12 | m5 | 20 | 4 | 16 | 10 | 25 | 19 |
| C12B | m5 | 20 | 4 | 16 | 20 | 25 | 19 |
| C24 | m5 | 20 | 4 | 8 | 10 | 25 | 19 |
| C24B | m5 | 20 | 4 | 8 | 20 | 25 | 19 |
| C22 | m5 | 40 | 2 | 8 | 10 | 25 | 19 |
| C22B | m5 | 40 | 2 | 8 | 20 | 25 | 19 |
| C22C | m5 | 40 | 2 | 8 | 5 | 25 | 19 |
| C23 | m5 | 20 | 2 | 8 | 10 | 25 | 19 |
| C23B | m5 | 20 | 2 | 8 | 20 | 25 | 19 |
| C25 | m5 | 20 | 2 | 4 | 10 | 25 | 19 |
| C25B | m5 | 20 | 2 | 4 | 20 | 25 | 19 |
| C20 | m5 | 12 | 2 | 8 | 10 | 25 | 19 |
| C20B | m5 | 12 | 2 | 8 | 20 | 25 | 19 |
| C21 | m5 | 12 | 2 | 9.5 | 10 | 10 | 4.75 |
| C21B | m5 | 12 | 2 | 9.5 | 20 | 10 | 4.75 |
| C21C | m5 | 12 | 3 | 12 | 10 | 10 | 4.75 |
| C21D | m5 | 12 | 3 | 12 | 20 | 10 | 4.75 |

**APPENDIX B: TABLE OF CHARACTERISTICS OF AWS EC2 TYPES**

| Image Type | Per Hour Cost | Per Hour Spot Cost | Reserved Cost | CPU (Intel) | Memory (GB) | Cores | Network Bandwidth (Gbps) | EBS Bandwidth (Gbps) | Storage |
|---|---|---|---|---|---|---|---|---|---|
| c6id.8xl | 1.61 | 0.7571 | 1.016 | Xeon 8375C (Ice Lake) | 64 | 32 | 12 | 10 | 1x1900 NVMe |
| c5.24xl | 4.08 | 1.836 | 2.57 | Xeon Platinum 8275L | 192 | 96 | 25 | 19 | EBS |
| m5.24xl | 4.608 | 2.163 | 2.903 | Xeon Platinum 8175 | 384 | 96 | 25 | 19 | EBS |
| m5.2xl | 0.384 | 0.1871 | 0.242 | Xeon Platinum 8175 | 32 | 8 | 10 | 4.75 | EBS |
| c5id.24xl | 4.838 | 1.664 | 3.049 | Xeon 8375C (Ice Lake) | 192 | 96 | 37 | 30 | 4x1425 NVMe |