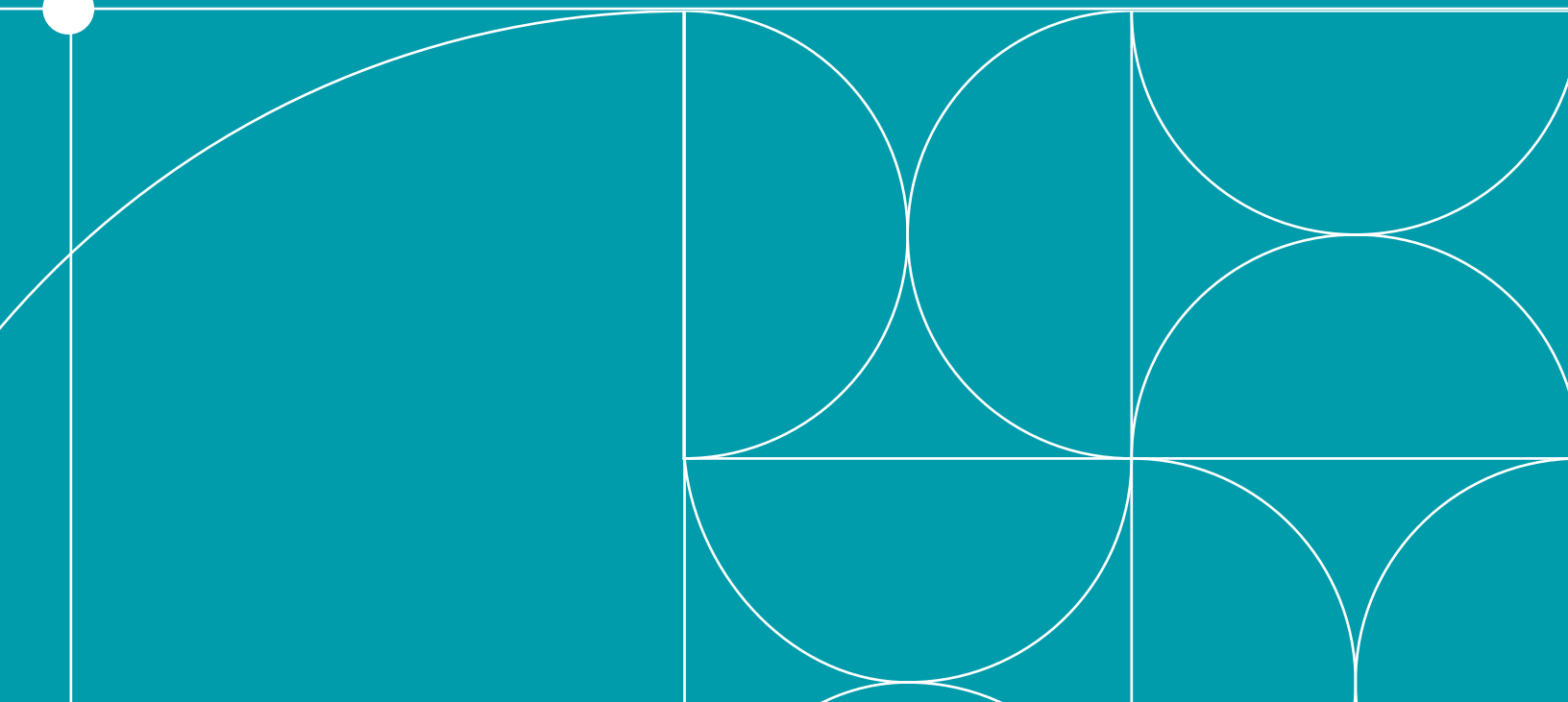
prognos health

The Search for a "Perfect" Healthcare Data Store

Building an Interactive Analytics Platform for Big Healthcare Data



Pushing the boundaries of speed

At Prognos, we regularly manage more than 45 billion lab data records. Additionally, we recently added pharmacy and medical claims data to our data lake — doubling the number of records and increasing the number of patients to more than 325 million. Our repository of data continues to grow with frequent updates from our existing data sources and the addition of new data sources. As the number of records grew, our requirements for how we manage this data also began to change.

We had a clear vision of creating a platform to provide an interactive, real-time experience to answer complex healthcare questions using our apps, APIs and development environments. To create the platform we envisioned — one that automates data harmonization, linking, and data management while providing lightning-speed access to vast amounts of data — it needed to support a variety of personas interacting with the system from clinicians to data scientists to commercial end users. To support the development of this platform, which later became known as prognosFACTOR®, we first had to identify a healthcare data store to make our data accessible at web speeds with queries completed in less than one second.

Searching for a "better" healthcare data store

At Prognos, we rely heavily on Apache Spark™ for many of our day-to-day operations. Historically Spark's flexibility and power allowed us to tackle even the most challenging data tasks with ease. However, our work with Spark often required large clusters and hours of run time in order to get the information we needed. As we embarked on creating the prognosFACTOR platform, it was clear from the beginning that Spark would not meet our requirements for a scalable and interactive system for the platform.

To meet our needs, we explored moving to a different technology stack. Our first consideration was open source online analytical processing (OLAP) technologies such as Apache Druid, ClickHouse and Apache Pinot. We were able to setup Druid and ClickHouse with our full datasets and ran some of our most common queries. While the speed improvements relative to Spark were significant, they failed to approach the sub-second requirement on complex queries.

We also discovered other challenges. Due to the nature of the end user and the datasets, we were unable to make strict assumptions on the types of queries the platform will execute. By design, prognosFACTOR users have programmatic access to the data, which we were willing to develop if the technology didn't support it. Yet even users who interact with the data through the apps (i.e. cohort designer), have access to a visual query builder interface creating the potential for many different ways to slice and dice the data. This made it challenging to fine-tune these OLAP technologies for pre-aggregations to achieve sub-second query times.

In parallel, we experimented with graph databases. With a promise of fast query times, we tested AWS Neptune™. Despite our efforts to try various schemas to represent our data in graph form, we found that Neptune was slow to ingest data (taking days) and expensive. The cost grew unreasonable as we increased the size of the cluster to reach our performance requirements.

The most promising technology — and the one that we went the furthest with — was Pilosa, an in-memory bitmap index. We created a set of indices for each field of our data (ex. lab test, drug, ICD code, etc.) where each bitmap in the set corresponded to a particular value in that field and each bit, or column, in the bitmap represented a patient. Pilosa allowed us to run set operations (ex. intersection, union, not) on these bitmaps in milliseconds. Using Pilosa, we were able to release a minimum viable product of a cohort designer for our internal users in just a few weeks.

Pilosa's efficiency in slicing and dicing the data was not the only aspect that initially made it the core technology for our back-end. Pilosa is also light (i.e. a single binary), easy to set up and manage, and quickly ingests data.

Birth of a new data store

Pilosa worked well for queries where we could ignore the temporal aspect of the data. Queries identifying patients based on a specific test, drug, or ICD in their history ran efficiently (in milliseconds) and required little effort because they aligned with the way Pilosa operates. However, when we wanted to limit the queries to a specific date range, the process became more involved, requiring us to keep bitmaps for different date granularity. In response we were able to change our schema to keep data at day-granularity which was enough to satisfy our querying requirements.

This experience made it clear that we would need to extend Pilosa to run queries with relative time constraints. For example, finding all patients with a certain diagnosis followed by starting a certain drug within 90 days of that diagnosis. These types of queries are simply not suitable for bitmaps. In order to match complex patterns, we needed the full history of the patients, not just its compressed bitmap representation.

To achieve this we decided to extend Pilosa with Redis, another open source in-memory technology. We posited that we could use Pilosa to identify the initial population from bitmaps and then utilize Redis with patients' full history to determine the final population. While the initial experiments were effective, large queries required hundreds of millions of patient data points to be transferred back and forth between the two technologies. More importantly, we were not happy with the memory requirements of Redis. Unlike the memory efficiency of Pilosa, Redis required us to use two terabytes of RAM scattered around multiple large nodes. We eventually replaced Redis with our own simple key-value data store which allowed us to maintain data with one-fifth of the memory requirements. The same codebase later grew into a standalone data store we named FACTOR Logic™.

As we built more healthcare specific query features in our new data store, our use for Pilosa was reduced to just its roaring bitmap library. We eventually decided to eliminate Pilosa from our technology stack and instead utilize roaring bitmaps directly in FACTOR Logic through open source libraries. Today roaring bitmaps remain one of the core technologies utilized in FACTOR Logic.

Patient-centric data analytics

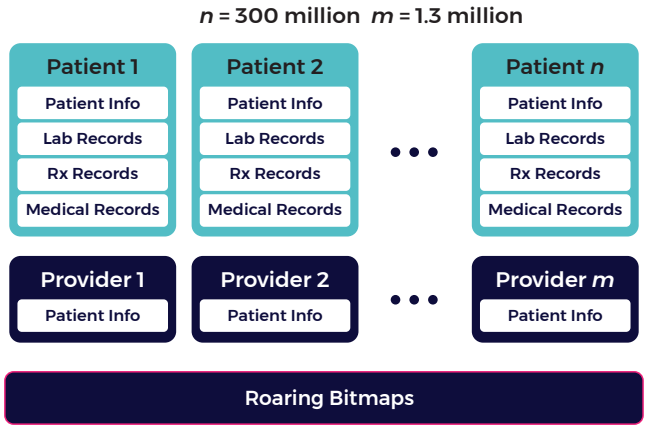
Due to the unique nature of healthcare data, there are a number of the challenges in representing the data using conventional data technologies. At Prognos, one of our core competencies is working with patient-centric data. Our work typically involves looking for a pattern or applying a transformation by strictly looking at the historical data of a patient. While conventional data technologies provide general purpose features like data partitioning, they are not suitable for partitioning of patient data. Most data technologies support partitions in the order of thousands or tens of thousands. These partitioning systems fall apart when used to partition data of 300+ million patients. Since patient-centric analytics is at the core of what we do at Prognos, it was clear to us that we could not depend on common data partitioning technologies available in the industry.

Using the same principles of healthcare data and the way we use this data, we can see that the queries can be easily and massively parallelized. Since analysis of one patient's history has no dependency on that of another patient, we can implement raw concurrency without a synchronization mechanism that would hinder performance. This level of concurrency was not available in any of the technologies we reviewed, so we built it into the FACTOR Logic system.

We designed FACTOR Logic such that the history of a particular patient can be accessed without a scan (i.e. zero look-up). All information related to a patient is stored in one continuous block in memory under that patient's ID, providing a pure patient-centric storage mechanism.

Moreover, investigating our use cases against our data revealed two distinct execution steps to our queries. The first step is identifying a population within the data sets

and the second step is performing various aggregations and transformations on the history of the identified population. Our analyses show that the first step requires the most resources. The second step, while much less resource intensive, is run multiple times to investigate different dimensions of the data. In order to support such use cases, healthcare analytics require fast, lightweight and dynamic views of data. To achieve our goals, we must be able to identify a population in a few seconds and then run various aggregations based on different dimensions of the same population.



FACTOR Logic is designed to detect queries that are running on the same population and is able to use the population that was identified as part of a previous query for future queries. These cached populations are kept in memory efficiently using roaring bitmaps and help to enhance the speed of the query.

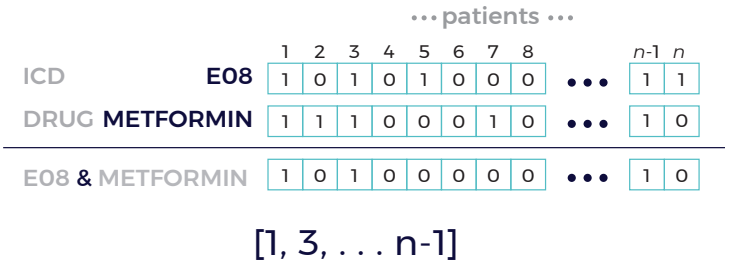
Roaring bitmaps

The versatility and suitability of roaring bitmaps for healthcare data make it a core technology in FACTOR Logic and, in general, for our work at Prognos. FACTOR Logic keeps major fields of the data as roaring bitmaps. For example, prognosFACTOR uses FACTOR Logic to keep the following bitmaps on de-identified data, mapping various aspects of the data to patients:

Demographics - Gender, State, Birth Year
Lab Tests - Test Type, ICD
Rx/Medical Claims - Drug Name, ICD
Providers - NPI, Provider Specialty

These bitmaps allow FACTOR Logic to quickly identify patients who have a certain test, drug or diagnosis in their history without looking at their history.

Each query first goes through a bitmap-only execution pipeline. This pipeline ignores certain aspects of the query such as date constraints, test values, and drug days-supply. The identified patients are then passed to a query interpreter that uses the full patient history with all its requirements to identify the final population. This population is then cached for future queries and passed to the analytics step.



Relationships

Roaring bitmaps are also utilized to represent relationships between data, much like a graph database. For example, using bitmaps all patients associated with a provider can be identified through roaring bitmaps without data scanning.

Similarly, due to the patient-centric nature of FACTOR Logic, all providers associated with a patient can be identified without scanning the data. That means FACTOR Logic can be used as an undirected graph database where the dimensions of the data are connected through patient nodes allowing us to, for example, explore the relationship between a particular lab test and a population of providers without scanning the data.

Data compression

Considering the size of our data, it was essential that we find a way to store the data in-memory and optimize not just for its size, but also for speed. Performance concerns, again, ruled out general purpose in-memory compression algorithms. Instead we decided to store all values as unsigned integers (indices).

During ingestion, the FACTOR Logic translation store translates the data into a set of integers. All incoming queries are also translated into integers before execution. That means both the data and the query executor only deals with integer numbers, on which logical comparisons (ex. an equivalence check) are more efficient than, for example, strings. This not only reduced the size of the data in memory, but also increased the performance of our queries.

To further optimize the size of the data in memory, each field specifies its own integer size. For example, fields like gender and state are represented using unsigned 8-bit integers, while larger cardinality fields such as provider NPI are represented as unsigned 32-bit integers. Dates in the records are also translated to "number of days since 2010" and kept as unsigned 16-bit integers.

It should be noted that unstructured data (ex. free text fields) is not suitable for FACTOR Logic's in-memory storage. Unstructured fields can be stored on disk and included as part of a query response, but currently FACTOR Logic does not provide any means to query unstructured data.

FACTOR Logic Query Language (fLQ)

Data scientists, in general, are mostly familiar with SQL as the query language. The challenges with SQL and other conventional query languages are:

1. They are not easy to construct for healthcare questions. These query languages are designed for general purpose, relational or graph data.

2. They are not optimized for common and complex healthcare questions because the query language is not aware of distinct properties of the data.

fLQ, a query language created by Prognos for FACTOR Logic, is an intuitive, simple to write and fast to execute query language specifically designed and optimized for complex healthcare questions.

Here is a simple query we commonly deal with:

> *All patients who were diagnosed with X within the last 12 months; and started on drug Y within 60 days of diagnosis; and did not change their regimen since that time.*

The new query language makes this and other healthcare questions easier to represent and efficiently executed. It would be represented as outlined below:

```
{
    "predicate":"SEQUENCE","contains":[
      {"predicate":"ICD10","contains":["X"],
       "start-date":"2019-01-01","end-
       date":"2019-12-31"},
      {"predicate":"RX","contains":["Y"],
                "time-relation":"before", "time-days":60},
      {"predicate":"NOT","predicates":[
        {"predicate":"RX", "contains":["*"],
         "time-relation":"before", "time-days":3650}]}]
}
```

The above example is a sequence query that lets the user explore the existence of a sequence of events in the patients' history. Within a sequence query, predicates can be time constrained relative to the occurrence of previous events.

> **Within:** Two events occurring within X days of each other where order does not matter.
>
> **Before:** Event must occur within X days after the previous event.
>
> **After:** Event must occur at least X days after the previous event.

Note that each predicate in the query can be constrained with specific absolute date ranges. This establishes an intuitive way to identify events within known dates.

Sometimes we want to look for a set of events that are occurring relative to an index event. Event queries allow us to define an index event along with a set of must-have and must-not-have events, each of which are constrained to the index event with relative days where the index event is assumed to be day zero.

*All patients who were diagnosed for the *first time* with X within the last 12 months; and started on drug Y within 60 days of diagnosis.*

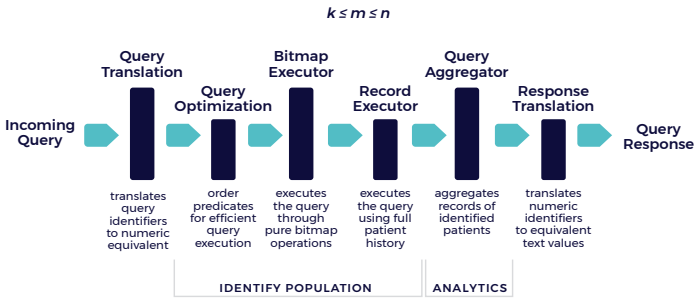
```
{
    “predicate”:”EVENT”,”contains”:[
      {“predicate”:”ICD10”,”contains”:[“X”],
       “start-date”:”2019-01-01”,”end-
        date”:”2019-12-31”},

      {“predicate”:”MUST-NOT-HAVE”,”predicates”:
        [{“predicate”:”ICD10”, “contains”:[“X”]}],
    “start-day”:-3650, “end-day”:-1}

      {“predicate”:”MUST-HAVE”,”predicates”:
        [{“predicate”:”RX”,”contains”:[“Y”]}],
    “start-day”:0, “end-day”:60}
     ]
}
```

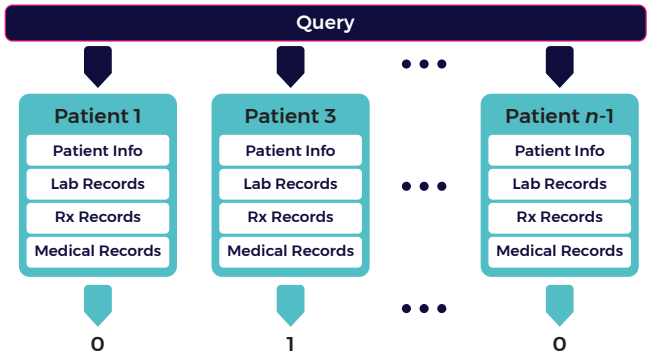
We have built numerous predicate types and helpers to assist our teams to quickly and efficiently answer everyday healthcare analytics questions. These range from event-transition queries that provide insights into pattern transitions to population-comparison queries to trend-aggregation queries and more. A list of all predicate types and use cases are provided as part of FACTOR Logic query language documentation.

Another advantage of fLQ is its structural design. A common FACTOR Logic query request contains two distinct parts:



1. Using the provided pattern in the query (as outlined with examples above), identify a subpopulation
2. Perform aggregations or analytics on the identified subpopulation

The first part is initially executed on bitmaps without looking at the actual records. Then the query is massively parallelized and executed over the identified subpopulation and cached for future aggregation or analytics requests.



Query optimization

Each type of predicate requires a different amount of execution resources. The query optimization step would change the order of predicates to maximize efficiency of its overall execution.

For example, a predicate related to the gender of a patient would execute much faster than a predicate that checks for the existence of a record within a specified date range. Filtering the patient based on gender would require a single lookup or comparison, while existence of a record or event within a given time period would require going over the event history of the patient.

FACTOR Logic uses short circuiting on AND logic to speed up queries. As a result, when looking at the predicates inside AND logic gate, we first find the predicate that evaluates to false and can stop evaluating the rest of the predicates in that logic. Our query optimizer looks for predicates inside AND logic and orders the predicates that are expected to be executed faster, first in the query.

API interface and authentication

FACTOR Logic has a built-in restful API allowing the user to interact with the data over simple http requests. The user can load, query and maintain the data and control all FACTOR Logic features through its API interface.

An important consideration for any data technology is security. FACTOR Logic comes with a built-in authentication system, which can work with the users own user management system via JSON web tokens.

Unlike most data technologies we regularly work with, it is extremely easy to deploy and maintain FACTOR Logic. It is a single executable binary, and once run, it can be managed through its API interface for administration, for logging and for linking with external services. This allows us to deploy FACTOR Logic for auto-scaling with ease. We can also containerize it with minimal configuration.

Programming language interface (PLI)

One of the common pain points with any data technology is that, however rare it might be, the user will come across cases where their needs do not align with how the technology operates. FACTOR Logic is a patient-centric data technology and, as such, most of its functionality operates around patients. That being said, it is common for our team to do physician-centric analysis, or require a custom pattern to match the patients that is either impossible or hard to represent with fLQ.

To address this, FACTOR Logic comes with its own native programming language interface (PLI) and interpreter through which the user can work with its in-memory data, bitmaps, and caches. The PLI allows the user to bypass the query language and interact with raw in-memory data directly. There are three ways to utilize the FACTOR Logic PLI.

1. Send the program to be executed through the "/execute" API endpoint.

2. Include the program snippet in the query, where the snippet acts as a custom predicate. This method allows the user to embed custom logic at certain parts of the query.

3. Use FACTOR Logic workbench, which is a notebook-like application that is deeply integrated with PLI.

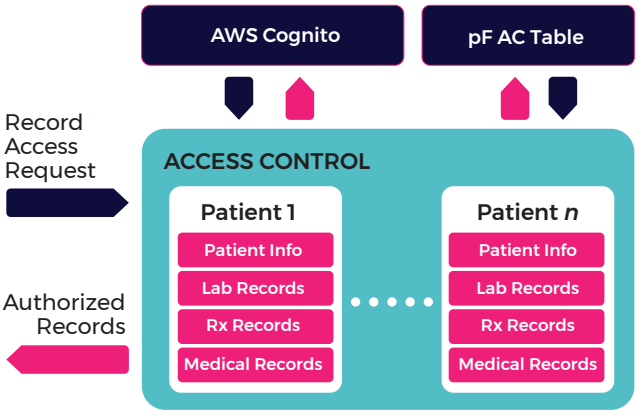
An important aspect of PLI is that it runs natively in the same process as the data and its interpreter is optimized for FACTOR Logic in-memory data structure. The user can take advantage of the same massive parallelization FACTOR Logic uses in the user's own programs. FACTOR Logic enforces access control and safety measures but otherwise the code is running at near native speed, providing answers to even the most complex healthcare questions within seconds.

Access control

Access rights play a major role in healthcare data management. Every record in our data registry comes with its own associated access rights. For each incoming request, FACTOR Logic query executor is allowed to access only the records it is authorized to use.

FACTOR Logic will call the user's access control service via a web hook through which the user can control access to data per request, per user, or per use case. Access control settings are cached by FACTOR Logic for 10 minutes to not to hinder query response performance of the server.

A particular user's access to data can be constrained based on dates on the record, the tags defined on the record, and/or the use case associated with the request. Any change to access-control metadata immediately takes effect within 10 minutes.

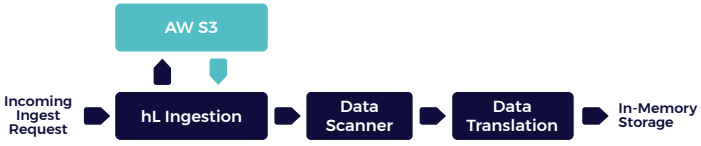


Unlike a conventional record-based access control on a relational database, FACTOR Logic needs to take a broader approach to how it implements access control. Because it is a patient-centric data store and a graph database, FACTOR Logic keeps track of more than just records, but also patients, providers, and their relationships. FACTOR Logic tracks and maintains the patients, providers, and records that each user is allowed to access so that it can provide answers without the overhead of access control.

Deployment and ingestion

As the core technology behind prognosFACTOR, one of our primary goals is to make FACTOR Logic feature-rich yet simple and lightweight. We take advantage of the Golang (Go) echo framework where possible. In fact, FACTOR Logic is implemented fully in Go using libraries available in Go repositories. We version and then compile to a single binary, allowing it to be easily transferred to a target server system and launched in a matter of seconds.

FACTOR Logic can ingest data at extremely high speeds. FACTOR Logic commonly ingests data from CSV files stored on AWS’s S3 service and can ingest data directly from S3 either in CSV or Parquet format. In our day-to-day use at Prognos, FACTOR Logic ingestion rates consistently stay well above one million records per second per node range. That means we can load a brand new cluster from raw data files within a few hours. After the initial ingestion, FACTOR Logic servers are incrementally updated with new data periodically. FACTOR Logic is not designed for streaming real-time data. The FACTOR Logic clusters are typically set up for daily or weekly incremental updates.



While FACTOR Logic is an in-memory database, it does keep a snapshot of its state and data on disk. This allows us to quickly restart the server at the speed of transferring data from disk to memory. FACTOR Logic clusters can be scaled up and down, or failed nodes can be replaced in a timely manner with the help of the snapshots.

Memory and compute requirements

The resource requirements for FACTOR Logic are both CPU and memory bound. The amount of memory limits the amount of data that can be stored and the number of cores (clock-speed) impacts the speed at which the queries can be performed on the data.

Depending on the number of records and the schema, FACTOR Logic is typically run on AWS EC2 instances (type R). It is standard to use a cluster where the data is distributed over four nodes, with a number of replicas that scale up and down based on the compute need.

Performance metrics

Test Server and Data

Number of patients: *351,198,746*

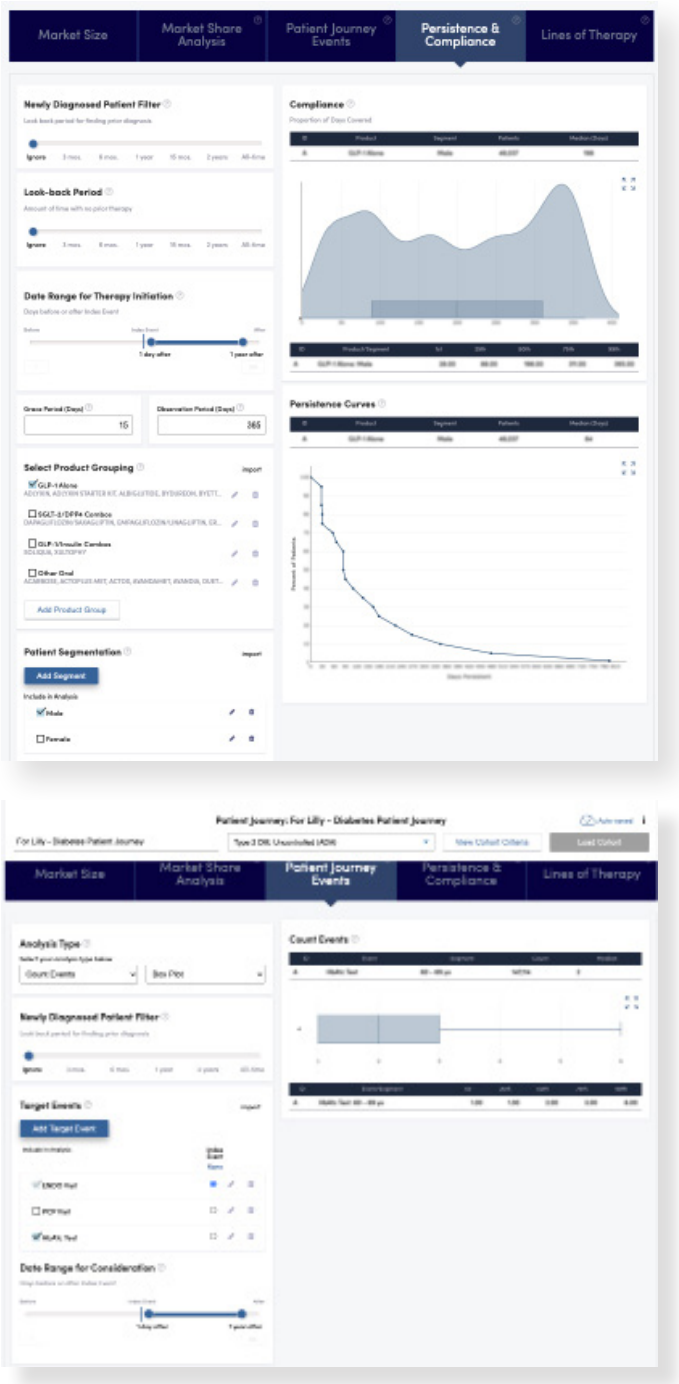
Number of records: *26,672,273,465 (lab records, Rx claims, medical claims)*

Running on a two-node cluster of AWS EC2 r6g.metal *(Total 128 cores 1TB RAM)*

FACTOR Logic v201029.1710 - Memory Utilization: *~750GB (data + pre-allocated query executors)*

	QUERY	RESPONSE TIME
1	*Among diabetes patients, computing the frequency counts for the number of times a patient between the age of 60-69 is tested for HbA1c in the year following an observation of a record indicating a visit to an endocrinologist*	1.25 seconds
2	*Computing the frequency count of the number of days a patient had a supply of GLP-1 monotherapies in the year following initiation of therapy. Initiation of therapy is any fill of the drug up to one year after the index date. The Index date is dates where a patient is observed to satisfy the diabetes cohort criteria.*	650 milliseconds
3	*Computing distribution of the number of days diabetes patients maintain an uninterrupted supply of given therapy, defined as no gaps in availability of the therapy of longer than a 15-day grace period.*	545 milliseconds

Diabetes cohort criteria defined as those with E11* ICD code AND either Hemoglobin A1c/Hemoglobin.total in Blood OR Hemoglobin A1c/Hemoglobin.total in Blood by HPLC lab tests with value greater than 9.



Above are screenshots of the query results on prognosFACTOR’s Patient Journey application which is set up to directly call the single-node test FACTOR Logic instance.

What is next for FACTOR Logic?

Interoperability: We aim to seamlessly integrate FACTOR Logic into the user’s workflow. The program currently supports Tableau and Databricks integrations so that our clients can continue to use the visualization and analytics capabilities they are accustomed to — but with results in seconds. We are currently working on expanding our capabilities for integration with Snowflake and Redshift.

Speed and Scalability: As one of FACTOR Logic’s main differentiating factors, it is important for us to continue to push the boundaries of speed and performance. Our apps, visualizations, and queries are becoming increasingly complex, creating a growing need to interactively drive insights from our full registry.

Deployment and Maintainability: Additionally, we are working on support for Kubernetes Administration through the built in app. Kubernetes is one of the most popular automated deployment systems used by our clients. FACTOR Logic Kubernetes support would allow our clients to integrate FACTOR Logic directly into their existing deployment, autoscaling and management workflow.

To see the power of FACTOR Logic in action, explore the cohort builder.

***Ali Koc, Vice President of Engineering**, leads the design, development and adoption of best-in-class and emerging technologies at Prognos Health. He recently pioneered the development of the prognosFACTOR platform and the supporting technologies, including FACTOR Logic. Ali has led complex technology projects across diverse industries for more than a decade. He also teaches undergraduate and graduate level courses in programming, databases, cloud computing and big data at CUNY-Baruch College.*