OpsAnalytics: Building a Scalable Analytics System

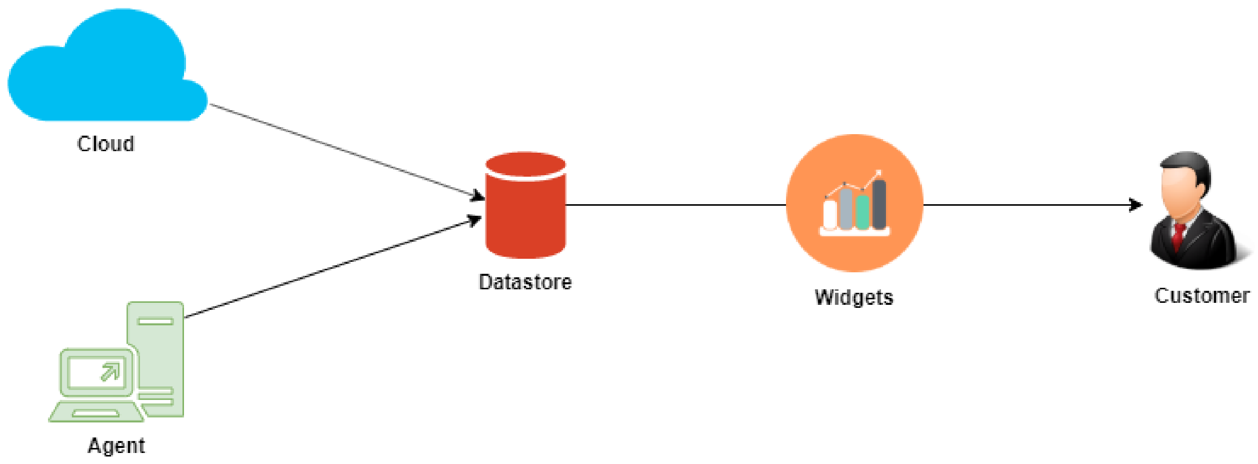
Customers subscribe to various services offered by IICS and design assets to solve certain business problems. These 'assets' are then executed as per Business requirement, in both *Cloud environment* and on-premise *Secure Agents*. One big opportunity just waiting to be seized here is to capture all this execution metadata and use it to:

- Notify users when they go over the licensed execution limit.
- Analyze customer behavior to get insights into customer satisfactions & needs and use it to improve functionality.
- Empower customers to optimize their processes by enabling them to analyze their own usage patterns.

But with execution numbers running in billions per day across services, sheer volume of data to be collected and queried is quite daunting. Also, collecting this data from the Agents running in customer environments where we have less control, is a much harder problem to solve than the Cloud nodes which run entirely under Informatica supervision.

Requirement

We needed a system that could gather all this execution metadata on cloud and agents, ship it to a central data store which then can be queried for reporting and visualizations. And to be of any value, it must be performant, scalable and reliable.



So with these goals in mind, we set out on a journey that was perplexing at times, but quite rewarding in the end.

Design

Without doing much research, few things that emerged from initial discussions were:

- Executions can be seen as "events" and metadata captured when they happen.
- Some buffering can happen at source as we're not building a real time system.
- Reporting queries aggregating over billions of executions, shouldn't take more than 5 seconds.
- Choosing the right datastore was important, let's start with simple Relational Database.

1. Beginnings

As a jumping-off point, we considered relational databases like MySQL cluster. Some of the advantages of such a system are: Stability and familiarity, High availability, ACID guarantees, simplicity. But as we started listing the concerns, they quickly seemed to outweigh the advantages:

- *Management*: Running and managing a cluster of partitioned MySQL instances could be challenging.
- *Query Performance*: MySQL cluster is built for write-scalability but performs badly for range queries.
- *Optimizations*: Traditional databases are built for general use and optimized for OLTP, whereas we are building an Analytics system.

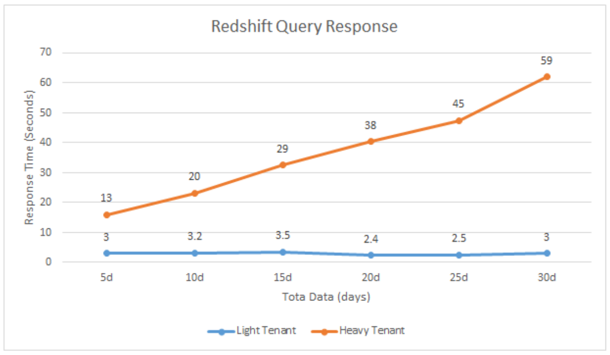
"Is there a database that is relational, scalable, available as a managed service and optimized for analytics?", we asked.

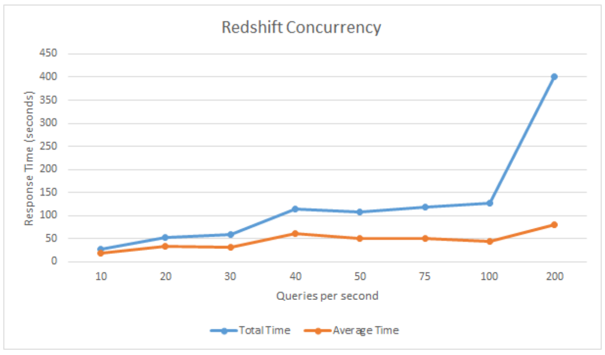
2. Getting smarter

Search for answers to above question lead us to Redshift, an AWS fully-managed, petabyte-scale data warehousing service that is optimized for Data Analytics (OLAP). Some advantages it offers over traditional databases are:

- *Massively Parallel Processing* with an array of compute nodes for processing queries in parallel and a leader node to co-ordinate the efforts, helps achieve parallelization and horizontal scalability.
- *Columnar Data storage* means values of a single column for multiple rows are stored in a single data block on disk. This reduces disk I/O requests and helps attain better compression rates.
- *Zone Maps* created by storing metadata for each block helps speed up queries by reducing data misses.
- *Kinesis Firehose* manged service streamlines the buffering and batch loading of data reliably into Redshift.

But as the clichéd saying goes, *proof of the pudding is in eating*. So we configured a Redshift cluster and inserted test data worth 30 days for benchmarking. Loading test data was smooth enough, apart from occasional load failures. We fired some aggregate queries and noted down the response times:





Test data: *500 Tenants; 100 Processes per Tenant; 100 million executions per day = 30 million executions by one heavy customer + 140K executions each by the rest*
Test Setup*: 2 node cluster with 32 GB RAM and 4 vCPUs on each node.*

Even with all the awesome optimizations, results were disappointing as the response times were quite out of acceptable limits. These numbers could have been improved upon to an extent with cluster expansion and source level aggregations but there was another problem: *Vendor lock-in*. Since Redshift and Firehose are AWS offerings, their counterparts would have had to be found for Azure and GCP Pods. That was unacceptable.

3. New Paradigm:

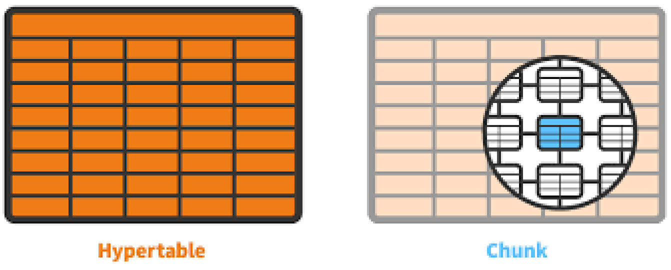
There is a class of systems that work with data which measures how things change over time. Where time isn't just a metric, but a primary axis! If you look closely, that's exactly the kind of data we are dealing with i.e. *how the usage of assets changes with time*. This is referred to as "time-series" data and there are databases that specialize in handling this kind of data. Some aspects of time-series data exploited by these databases are:

- Write workloads are mainly *inserts* which typically arrive in time order.
- Rarely any *updates* or pointed *deletes*.
- Read queries mostly analytical that access data by the window of time.
- Availability with eventual consistency preferred over transactional correctness.

In recent years, this segment has exploded with a lot of interesting options to choose from.

One such option is TimescaleDB , which is essentially an engine purpose-built to handle time-series data, on top of a cluster of good ol' PostgreSQL. It's scalable and available on all major cloud providers. It claims to offer best of both worlds: stability & power of a mature relational database and optimizations of a time-series database.

These optimizations are achieved by breaking down the dataset into "chunks" of data and storing them in individual tables. These chunks are then accessed through a virtual view called "hypertable" which exposes the entire dataset as a single table to the end user.



Hypertables are partitioned by time interval by default and can be partitioned on additional fields. This chunking of the dataset helps achieve two things:

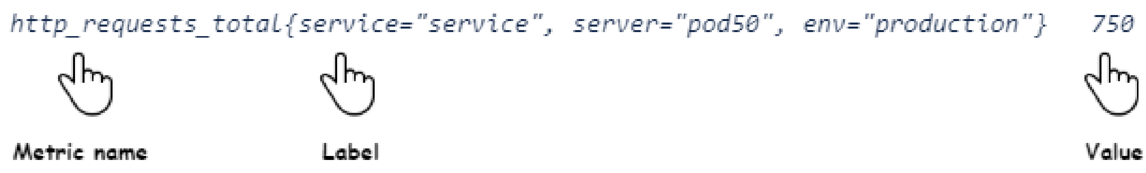
- Reduce disk I/O: Since chunks are disjoint, it helps the query planner to minimize the set of chunks it must touch to resolve a query.
- Parallelization: Partitioning allows the same query to run in parallel on different chunks even on a single node.

As promising as it seemed, the performance fell way short with query response times in the range of 3-7 mins. So that was that.

4. New Hope:

One of the more popular choice in the Time-series segment is an open source toolkit called Prometheus. As it's all the rage in DevOps world and already used extensively for our internal monitoring needs, we had to give it a try and see if it could work for this use case. Prometheus works by regularly "pulling" metrics data from targets over HTTP and keeping it in memory for couple of hours before eventually persisting to local storage. Targets can be configured either through static configurations or dynamic service discovery.

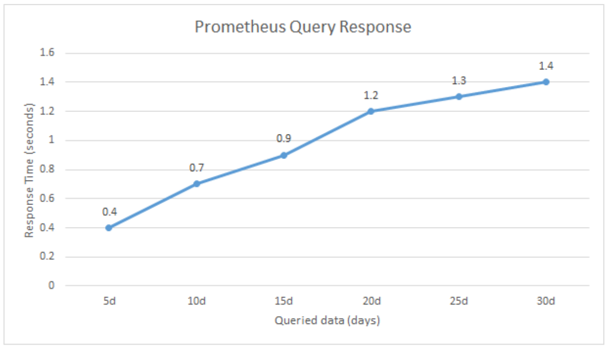
Unlike other databases considered so far, Prometheus is a NoSQL system purpose-built for monitoring metrics on other systems and metrics is all that it understands. A metric is defined by its name, set of associated labels and the value of the metric. Labels can be used to filter and group data to fetch relevant values for visualization.

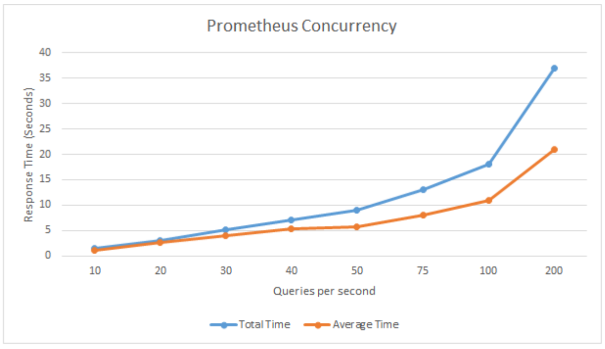


```
http_requests_total{service="service", server="pod50", env="production"}   750
```

Prometheus server is surprisingly easy to configure and run. Apart from the server which does all the heavy lifting, Prometheus ecosystem has many other useful tools like Alertmanager for handling alerts, service discovery, client libraries and out of the box integration with Grafana for visualization.

But enough about tools, where them performance numbers at?





Bottom line: **Prometheus is lightning fast.**

So we're done right? *Not so fast*! Although Prometheus excels at response times, sadly enough it lacks in some of the other aspects that are essential for a robust and reliable system:

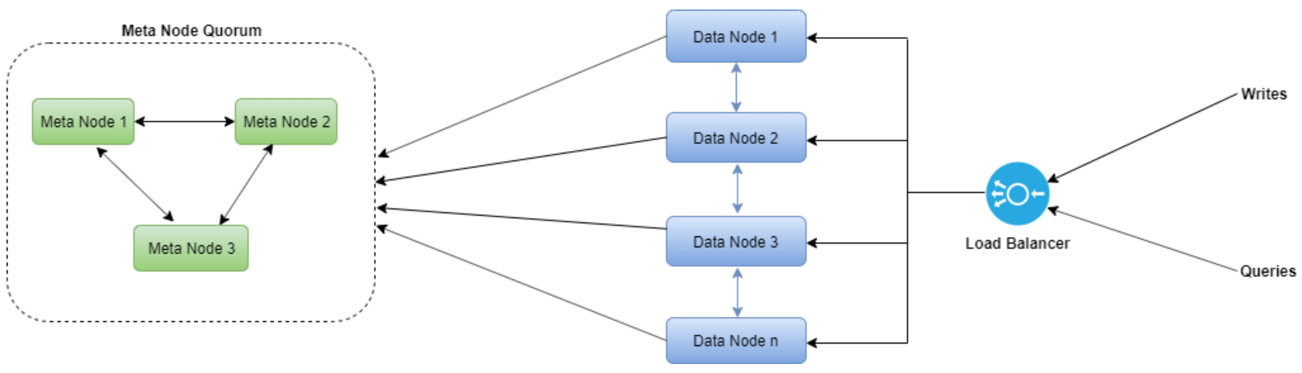
- *Scalability* : Though a single instance of Prometheus can handle large amounts of data efficiently, it has no strategy for scale out and clustering.
- *Long term storage*: Prometheus stores all data on local storage and since there is no out of the box clustering or replication, unwise to use it to handle 6-12 months of data.
- *Pull mechanism*: Prometheus's pull-based metrics collection approach is not ideal for scraping data from loosely coupled on-premise agents with challenges like authentication and service discovery.
- *Managed service*: No reliable managed Prometheus as a service in the market means significant deployment, management and debugging overheads for both Ops and Dev teams.

It's worth mentioning that some third party Prometheus extensions like Thanos, Cortex, Victoria Metrics etc. have attempted to address some of the above concerns but ultimately fail to make a convincing case.

5. Bull's eye

Seeing Prometheus perform so well gave us the confidence that a NoSQL based Time-series database might just be what we needed, provided it came with proper clustering, long term storage and all that jazz. That's when we stumbled upon InfluxDB, another popular choice in the segment. It offers power and performance of a NoSQL Time-series database and Redshift-like elaborate cluster management to provide elastic scalability, high availability, replication, long term retention and eventual consistency.

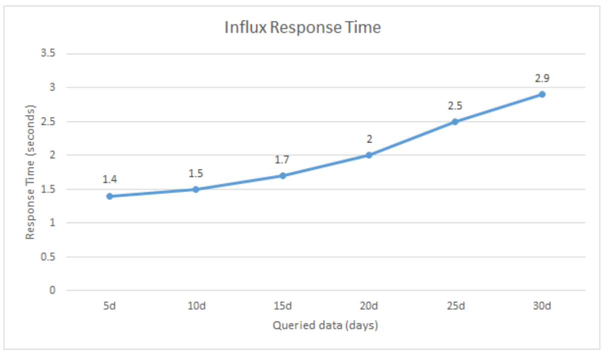
An InfluxDB cluster consists of *Meta nodes* that keep a consistent view of the cluster and *Data nodes* that replicate data and query each other over TCP.

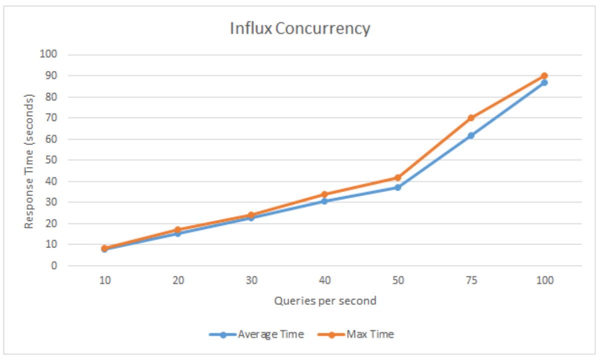


Influx's data model is similar to Prometheus with data points represented as metrics, associated tags and the value. Metrics must be initialized and incremented at every event at the client and this data must be shipped over at regular intervals as InfluxDB works on "push" model.

Push vs Pull : Although it's a subject of fierce debate in the time-series monitoring world, in general the differences are mostly pedantic. But for our use case, push did seem to fit better considering the nature of targets we wanted to collect data from.

With our fingers crossed, we configured a 2 node InfluxDB trial cluster and loaded it up with a month of test data.



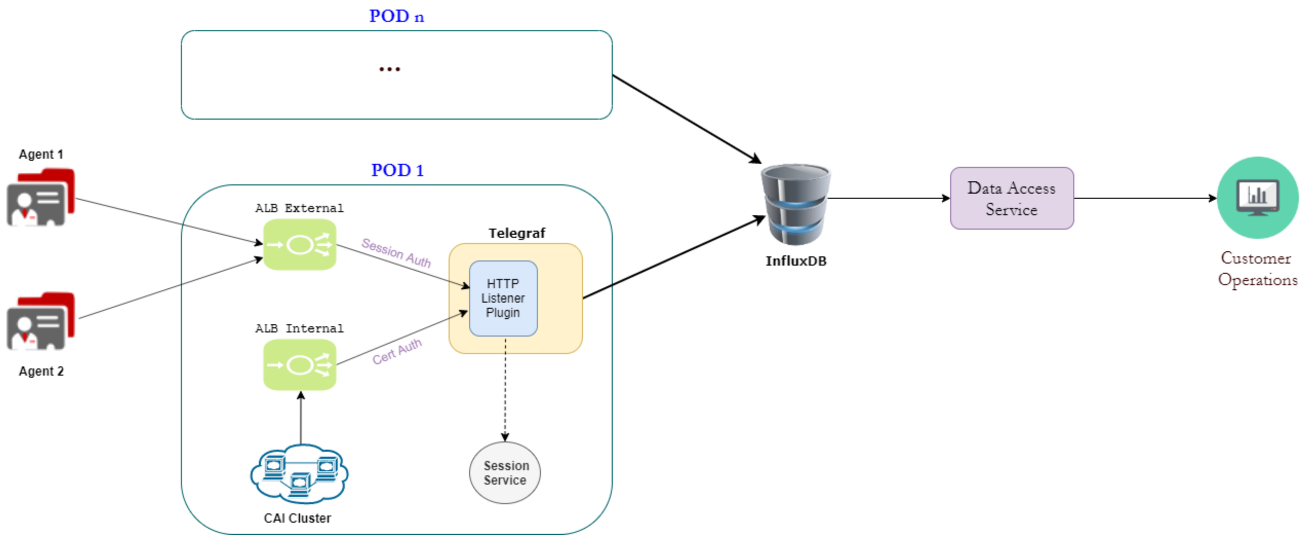


Bingo! All our queries reported back within 5 seconds and our long search was over!

Concurrency numbers here don't look so hot owing to the fact that very light weight trial nodes were used for this test. Only part remaining now was to figure out how the data was going to be loaded into the database.

Influx ecosystem, also referred to as *TICK* stack, offers a tool called Telegraf exactly for this purpose. Telegraf is an open source, plugin-driven server agent written in Golang, for collecting and sending metrics to various databases including InfluxDB. We ended up customizing its http_listener plugin in order to add support for session id based authentication for the agents.

With the final piece of the puzzle in place, the design was finalized.



Conclusion

All products are built to solve a set of specific problems and excel in some areas, at the cost of falling behind in others. Influx worked for us, but may not be the best tool for the system you're building. This blog does not attempt to provide ready made answers but to showcase an example of a journey one might take to get to the answers.