happiest minds

Leveraging Simplified Anomaly Detection Solutions for Increased Analytics Insights

Happiest People. Happiest Customers

Contents

Abstract..... 3

Introduction..... 3

High Level Overall Architecture Diagram..... 4

API Server..... 4

Ingestion Engine..... 4

Metadata Store..... 5

Analytics and Distributed Data Handling Core..... 5

Data Store..... 6

Data Aggregators..... 6

Messaging Layer..... 6

ML System..... 7

Query Engine and Spark Layer..... 7

Data Transformation Layer & Processed Data..... 7

ML Worker Nodes..... 8

Generated Model and CMI (Common Model Interface)..... 8

API, SDK, and Workers..... 8

Architectural Choices..... 9

About the Author..... 10

Abstract

In this paper, we present our home grown solution on Anomaly Detection along with its architecture. We will discuss the features and capabilities of the existing platform and project our future plans for it as well.

Introduction

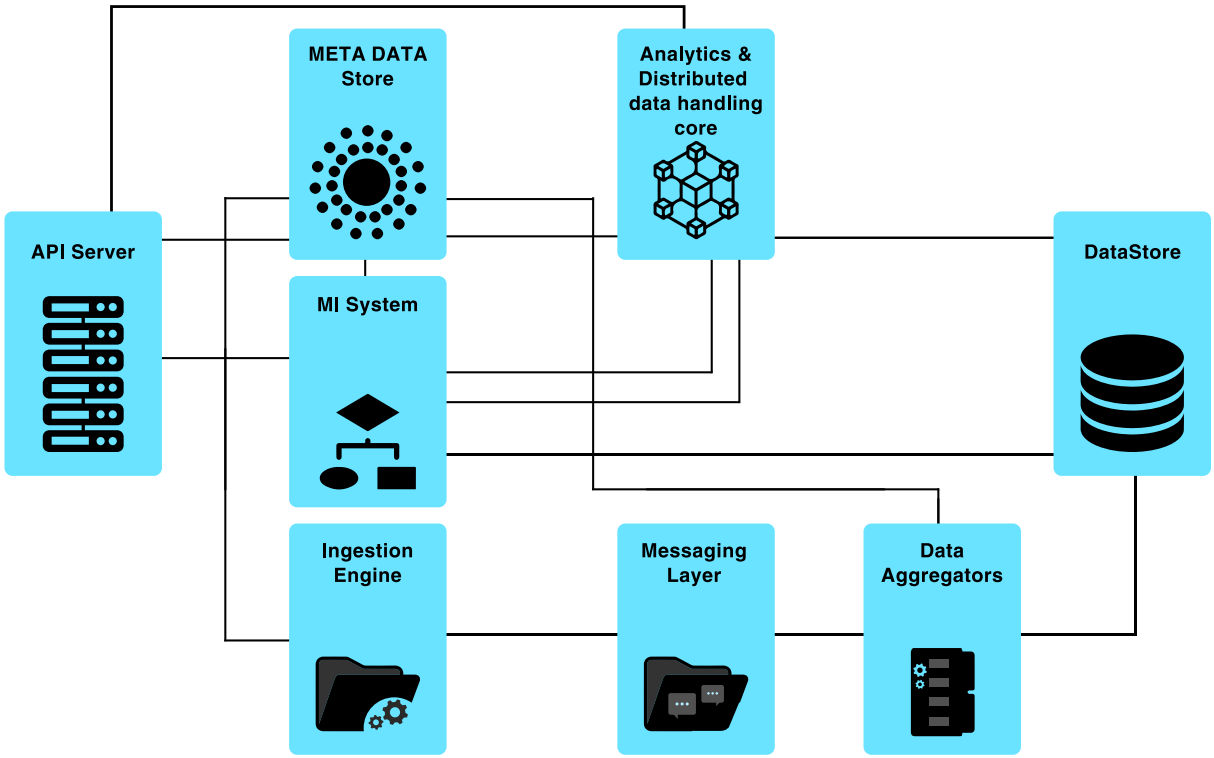
Anomaly detection (or outlier detection) is a process of the identifying items, events or observations which do not conform to an expected pattern of data in a dataset. Generally, the data in a given data pool is related/correlated with each other in a certain fashion- known as data patterns. Anomaly Detection is about finding out deviations from these patterns by using various statistical and machine learning algorithms. Typically the anomalous items will translate to some kind of problems such as bank fraud or a structural defect, medical problems or errors in a text. Anomalies are also referred to as outliers, novelties, noise, deviations and exceptions.

Detecting an outlier needs a rich domain expertise and the entire process is entirely dependent on a variety of data. Automating Anomaly Detection would mean attaining the power to dealing with any variety of data, running it against any algorithms and judging the efficiency of each algorithm.

This is intended to empower the domain experts with an automated engine to play with.
In this tool, we have made the work easy for Business analyst to simply use our tool and find out anomalies without the requirement of any dedicated Data Science or Big-data engineer.

Our solution not only covers all of above-said flexibility but also scalable for any new algorithm to be added to the framework. The best part it's a feedback based system which facilitates analyzing and making decisions based on your algorithms performance on a given data set.

High Level Overall Architecture Diagram



API Server

The API server is the major interface between the user and the other components in the system. Details on the design:

- API layer is designed to facilitate the simple and quick transition between changing the rest layer from one framework to the other, without much code change.
- API server can run using any of the two frameworks being Undertow.io and Spray.io. These options can be configured from the configuration file.
- Task segregation and execution strategy facilitates Fast Response time

The API Server is also the initiator of all jobs and tasks. These are maintained and coordinated by a central system making sure that all tasks are run and executed properly. This also works as a fail-safe mechanism.

Ingestion Engine

The ingestion Engine has one primary task accept the data sent from the client system, process the request, check for common validations and forward the data to the messaging system. Same can be processed later by consumers in a Batch mode.

There are two ingestion Engines in place:

- First ingestion engine is directly integrated into the API server itself which reads the requests and directly loads the data into the data store without using Kafka queue to distribute the task. This is capable of handling small payloads from about 10 to 15 thousand updates per second.
- The Second one is the dedicated ingestion engine built using the in-house HTTP-server **Vega–HTTP**, which can handle a very large number of requests and push the data back to a Kafka cluster without any delay or failover in the system. The ingestion engine can process data and send the requests out with minimal CPU usage and performance lag. This system is used for extremely large data systems which would push data scaling from 100000 requests to 170000 requests per second for a basic 2 core server.

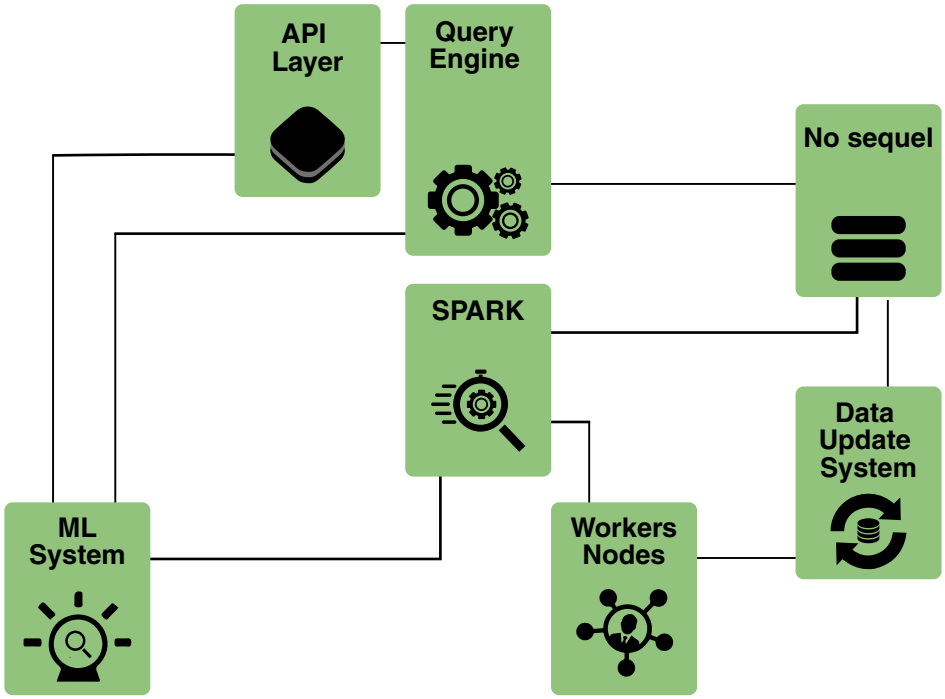
Metadata Store

The solution is metadata based and all our metadata is stored currently. The Metadata Store contains all the information for data in a given data pool. All the data structures, jobs, statuses, etc. These metadata definitions are required by all the component in the system. The Metadata Store is not tightly coupled with the database which means we use no joins or any other form of relationships in the table thus making migrations to an NOSQL database easier.

Centralized service management system is also an integral metadata for us where we store information on which servers are live showing what are they doing, etc.

With respect to the CAP theorem of all distributed systems, our architecture prefers Availability and Partition Tolerance over Consistency. Therefore, a pull based Mechanism for updating cache and a common Cache Consistency Server ensures that every other component knows that there have been updates in the system.

Analytics and Distributed Data Handling Core



System comprises of two main components

- **The analytics core has SQL wrapper on top of NOSQL at its epicenter. It powers the UI reports and all other key KPI. It supports parallel query run and has caching features enabling a faster delivery of information near real-time.**

  Query Engine : The Query engine is a query interface on top of NOSQL for the API server or other systems. The Query engine has inbuilt functions to parallelize querying for optimal speed. It also enables users to dynamically fetch records that are required by them for Analysis. The Query Engine also performs Statistical calculations if required by the API server. The Query has an aggregation layer for reducing run-time calculation time. This layer in the system makes it well suited for advanced analytics in a distributed manner without loss in performance. The Query Engine is not directly availa ble to the user but he can use it via the API-Server for viewing reports or for other functions.

The ML-System is used to train the data and generate models for prediction or score calculation (elaborated in a section below). The generated CMI (Common Model Interface) model is used to compute the scores on available data sets. We use spark for parallelizing all data operations from fetching, aggregating to running it against the new model and the resultant output. The Worker Nodes are distributed independent nodes that take the tasks available and processes them. They are also responsible for initiating the Spark job and computing the scores respectively.

- The ML-System is used to train the data and generate models for prediction or score calculation (elaborated in a section below). The generated CMI (Common Model Interface) model is used to compute the scores on available data sets. We use spark for parallelizing all data operations from fetching, aggregat ing to running it against the new model and the resultant output. The Worker Nodes are distributed inde pendent nodes that take the tasks available and processes them. They are also responsible for initiating the Spark job and computing the scores respectively.

Spark for Distributed Data Handling and Transformation layer. The Distributed Data handling used for these tasks:

- Data transformation for preprocessing as per the algorithm's requirements.
- Passing the data to the respective worker node to train the data and to generate the model.
- Data transformation for algorithm score computations: the Data Update System is used to update the Data in the Data Store. It generally updates the tables when new records are inserted it also updates the scores if a generated model is available for processing. If a generated model is available for processing it checks if the model is real-time scoring capable, if so it also computes the new record against the model and updates the score along with the new record.

  This makes the Values available for reporting near real-time.

Data Store

We have a basic need to ingest a variety of data from various domain. Hence, the data store should be capable of handling multiple schemas. The main data store is currently a Columunar NoSQL with a SQL wrapper which is distributed , support high performance and is reliability. This also supports secondary indexes. SQL-like queries and statistical functions and even UDT (User Defined Functions).

Data Aggregators

We have a basic need to ingest a variety of data from various domain. Hence, the data store should be capable of handling multiple schemas. The main data store is currently a Columunar NoSQL with a SQL wrapper which is distributed , support high performance and is reliability. This also supports secondary indexes. SQL-like queries and statistical functions and even UDT (User Defined Functions).

Messaging Layer

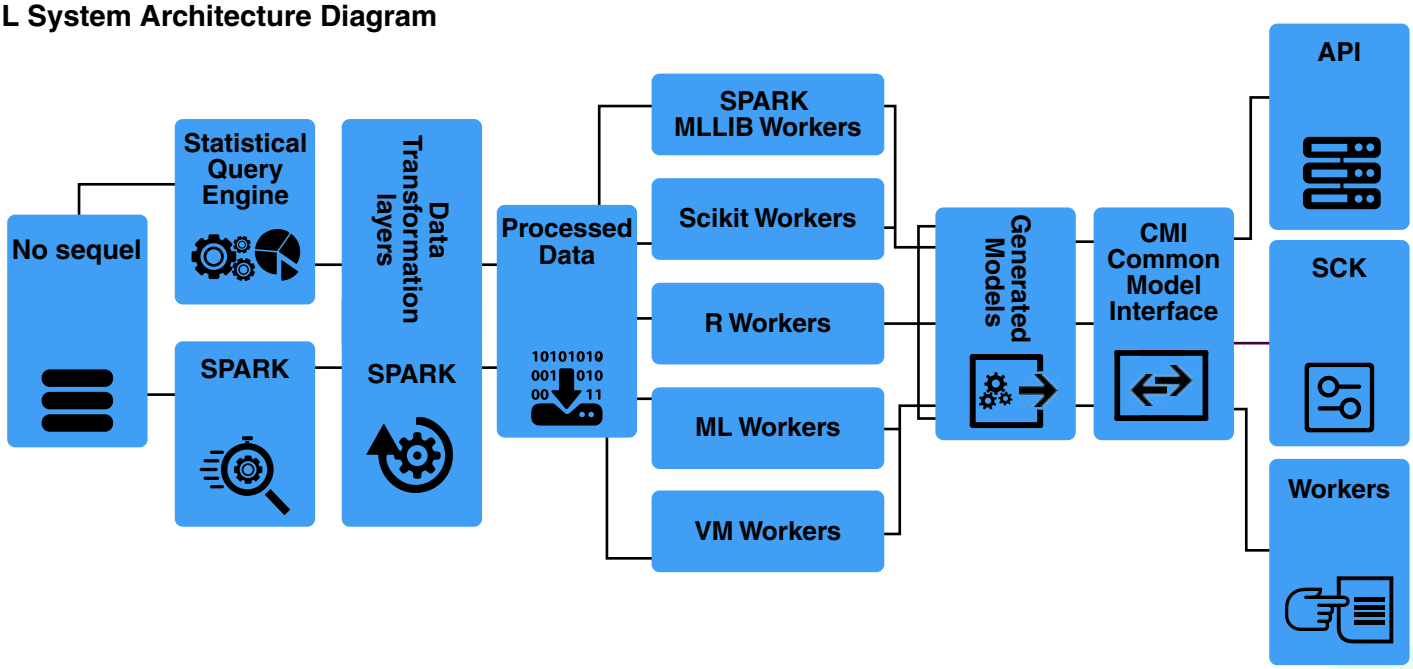
The messaging layer or the transport layer is responsible for load balancing the incoming data to the system. The messaging layer comprises of apache Kafka. Messages are pushed into topics with multiple partition queues thus making it possible to consume the data in a highly distributed manner. The multiple producers and multiple consumer models is effective in balanc- ing load and peak surges in traffic without causing any resultant issues in the system.

ML System

The ML System generates machine learning models. These models are in turn consumed by API's to predict or calculate if a given record is an anomaly or not.

The well-defined UI allows to play with parameters for each of Algorithms. The ML System also is distributed in nature that enables various features and the ability to add custom algorithms. These can be tried and tested on the fly against your data set.

ML System Architecture Diagram



Query Engine and Spark Layer

Query Engine is mainly used to query statistical values that are required for certain algorithms to work. This is using the existing functions available in Phoenix and can also be extended by writing custom UDF (User Defined Functions) over Phoenix.

Apache Spark is exclusively used for Transformation. Any training done across the training system like VW (Vowpal Wabbit) or scikit-learn is done on transformed data passed back by Spark.

Data Transformation Layer & Processed Data

The Data Transformation layer makes common libraries and functions available to the data specialist that is to set the Transformation steps. Scalper Function has accessibility to all the data and custom functions, hence allows you to scale or normalize your data based on your requirements. Supported scaling methods are z-score, Min-Max normalization. For other methods, the required data can be made available using the Query Engine even before entering the Transformation layer.

It is not required to use the DTL Layer for every Algorithm. If the Algorithm purely depends on other factors like statistical function. An example of such would be our HBOS (Histogram Based Outlier Scoring) implementation where all the calls for building the model are done directly using the query engine.

The processed Data can be made available either as Flat files either in local storage or in HDFS for the respective worker. The worker can then pick up the data and Train the model from there onwards.

For any Anomaly detection solution Nearest neighbors selection process is must . We have developed Multilevel unique Hashing algorithms to deal with categorical and numerical data in order to find out nearest neighbors

ML Worker Nodes

The ML worker nodes are independent systems that help us train and prepare models independently. Implementing this will enable us to have our own library which further enables users to write custom algorithms on Scala using our in-house Machine Learning Engine.

We have used right combination of density based , ML based and AI algorithms to analyze the compute outliers. The workers that are being built are using Spark MLLIB. The worker node has mainly two tasks; use the prepared data to train the users custom or existing algorithm and generate the model that has to then be converted to a CMI (Common Model Interface) which can be used by other systems to calculate and compute the scores based on its output

Generated Model and CMI (Common Model Interface)

The Generated model from any Language or System doesn't have to follow any pattern. It can be different for different languages and even for algorithms. While preparing the Algorithm it is also required to write the transformation steps from the generated model to the CMI.

CMI (Common Model Interface) is a simple Interface with two functions Initialize () and Predict (). Where Initialize () is used to prepare or set up the model and the Predict () function is used to predict the values for the given record. The CMI is going to be used across all the systems, therefore, it is required that it does not have any dependencies on third party libraries.

Example: Let us consider a simple two-dimensional linear regression.

$Y'i = b0 + b1X1i + b2X2i$

If the model generated by Spark MLlib has to be reverse engineered to fit the equation for the given set of input parameters the same output generated by the spark model will also be generated by the CMI model.

API, SDK, and Workers

All the other components use the generated CMI model to predict and reuse the model for prediction instead of recreating or running the algorithms against the native library for prediction. This makes it easier for Distributing and reusing it in various environments for quicker predictions. Algorithms that cannot be reused in this manner like the ones depending on other data like its neighbors or relative requirements are not built using the CMI. Because it cannot be run outside the system as they are not accessible to other devices.

Since CMI is serializable, it can be easily distributed and used for parallel computation like in the worker mode while re-computing scores. The worker nodes parallelize the task, computes scores for all the systems simultaneously

Architectural Choices

This is a Big Data solution using NoSQl , SQL wrappers , real time transformations ND streaming analytics for apt phases of the solutions to facilitate

- Random and consistent Reads/Writes access in high volume request
- Atomic and strongly consistent row-level operations
- Auto failover and reliability
- Flexible, column-based multidimensional map structure
- Variable Schema: columns can be added and removed dynamically
- Integration with Java client, Thrift and REST APIs
- Auto Partitioning and Sharding
- Low latency access to data
- In-memory caching via block cache and bloom filters for query optimization
- No sequel allows data compression and is ideal for sparse data
- Aggregation Enhancements

On initial performance comparison with other SQL layers like Hive and Pig, we came to the conclusion that not only is No sequel faster but also has the required features we were looking for in terms of statistical functions, UDF's and the ability to have custom Hash Functions for indexes.

About the Author

Bhawna Manchanda

Bhawna is a Big Data Architect at Happiest Minds. She has 11 + years experience in conceptualizing, building large/complex end to end solutions and implementations. This includes Modeling & designing Enterprise Application, Solution Architecture , building Enterprise Data Hubs/Data Lake and creating Strategic value through Business Intelligence and Data Analytics.

Happiest Minds

Happiest Minds enables Digital Transformation for Enterprises and Technology providers by delivering seamless Customer Experience, Business Efficiency and Actionable Insights through an integrated set of Disruptive Technologies: Big Data Analytics, Internet of Things, Mobility, Cloud, Security, Unified Communications, etc. Happiest Minds offers domain centric solutions applying skills, IPs and functional expertise in IT Services, Product Engineering, Infrastructure Management and Security. These services have applicability across industry sectors such as Retail, Consumer Packaged Goods, Ecommerce, Banking, Insurance, Hi-tech, Engineering R&D, Manufacturing, Automotive and Travel/Transportation/ Hospitality. Headquar-tered in Bangalore, India, Happiest Minds has operations in the US, UK, Singapore, Australia and has secured $52.5 million Series-A funding. Its investors are JPMorgan Private Equity Group, Intel Capital and Ashok Soota.

E-mail: Business@happiestminds.com
Visit us: www.happiestminds.com

Follow us on