Synechron

Digital / Business Consulting / Technology

Architecting a Unified Data Platform

- Data lake to Analytics

Table of Contents

Introduction

The financial industry today is facing new challenges like increasing regulatory and compliance pressures, improving their control and governance capabilities, and enhancing oversight and transparency to prevent reputation damage and financial loss, all while still driving performance and profitability. Financial institutions are looking for innovative ways to ensure compliance, enhance performance and maintain a competitive advantage.

Some of the challenges financial institutions have been faced with due to increasing regulatory obligations include:

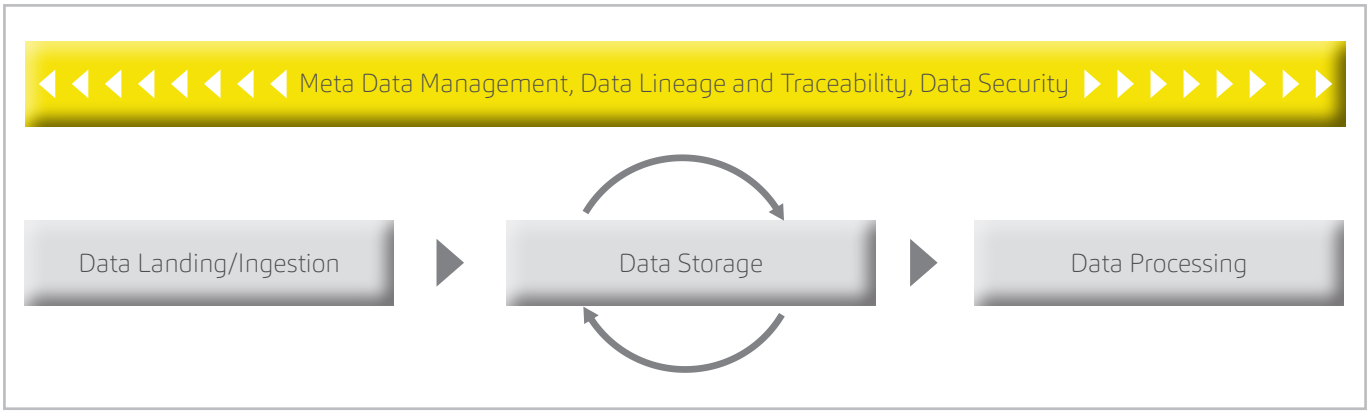
- Increasing regulatory requirements forcing financial institutions to disclose data at an increased frequency and granularity (e.g., BCBS 239, daily LCR)
- Increased analytical and compute workloads (e.g., FRTB, CCAR)
- Increased transparency and reporting needs where banks should report pre-trade and post-trade data to regulators (e.g., MiFID II)
- Predictive analytics and machine learning implementations (e.g., AML and fraud detection)

All these challenges are forcing financial institutions to evaluate their:

- System architecture, infrastructure capability
- Data strategy, integration pattern
- Data analytics platform, Business Intelligence solutions
- Storage and compute stability

Organizations are looking at building a unifying platform to run analytics and machine learning models on data. To address this, this paper explores the options of using some of the open source tools available today. As part of this, there is a focus on creating data lake, and building and operationalizing (deployment and cluster management) the analytics/ML platform for creating a reference implementation for Fundamental review of the trading book (FRTB). In our example, we can use the data lake for landing trade and reference data from the various trading systems, and then transform them into a common format for creating a successful analytics solution.

Data lake: The data lake creation consist of data ingestion, storage, and processing. The following diagram illustrate the stages along with governance and security requirements.



- Landing data receiving raw source data that supports key set of features and capability like (Batch vs streaming), push vs pull and different endpoint support
- Ability to handle different data types (CSV, JSON, Database, log files etc.)

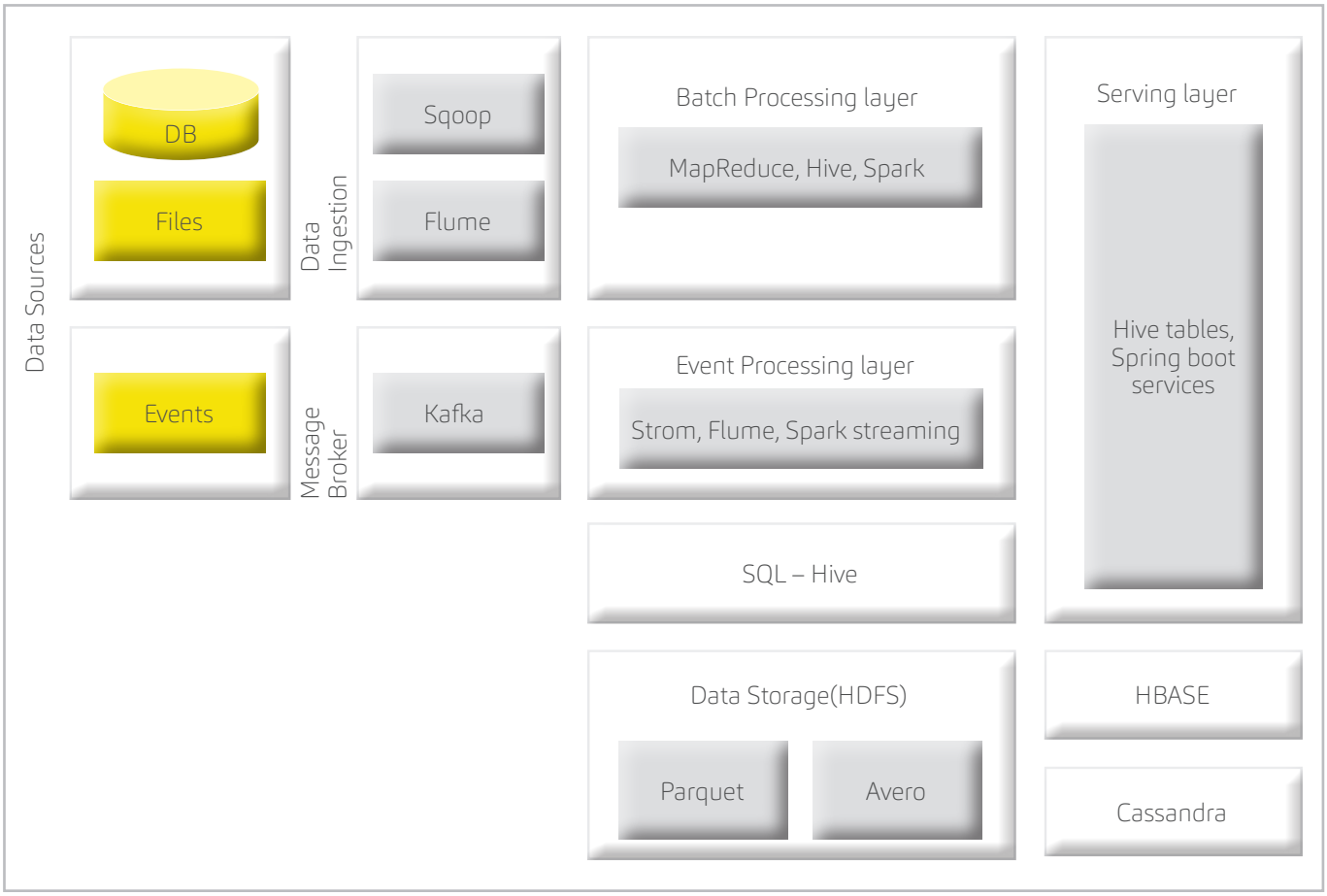
- Provide a flexible access mechanism and be highly optimized for both batch as well as near real-time operations
- Raw data (unaltered data you capture from the source)
- Derived data and aggregated data

- Process the data acquired to derive a normalized format/meaningful insight

We have considered data lake design based on Lambda Architecture (Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batches- and stream-processing methods), and have considered following two approaches depending on their state of current state of data acquisition and storage.

Approach 1 for Data Lake Build

As per our experience, we have seen the following technology mapping for organizations which have implemented Hadoop based data lake build out.

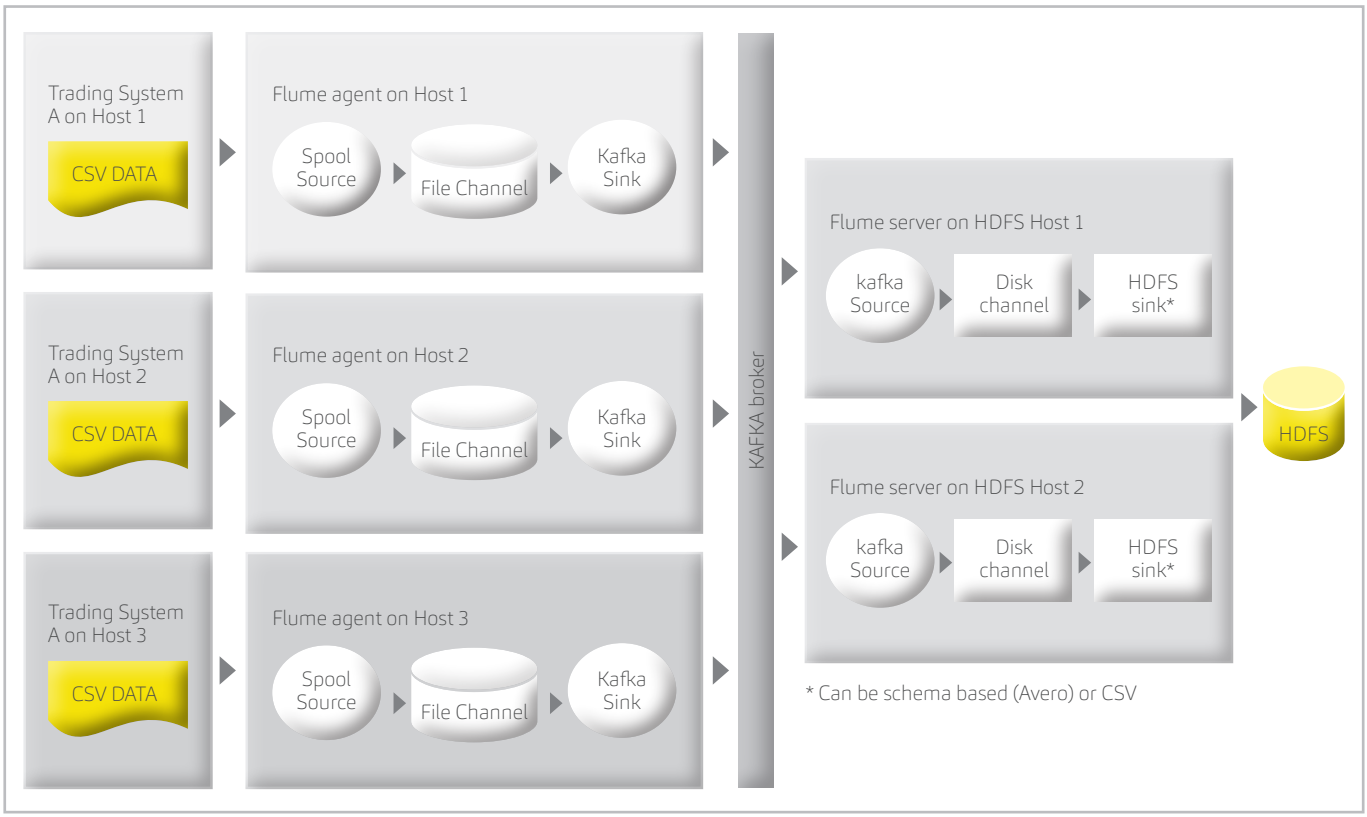


Data Ingestion

Ingesting Data from Files

Apache Flume is a framework designed to collect log data generated in near real-time from multiple application sources. From an FRTB perspective, this could be used to position data generated by trade execution systems and stored as comma separated [CSV] text files.

The design goal here is to ingest position data from multiple trade execution systems and land them into HDFS. A Flume setup that uses load balancing and fan-in would be ideal for this purpose. The diagram below illustrates a Flume server topology: Fan-in based pipeline with load balancing.

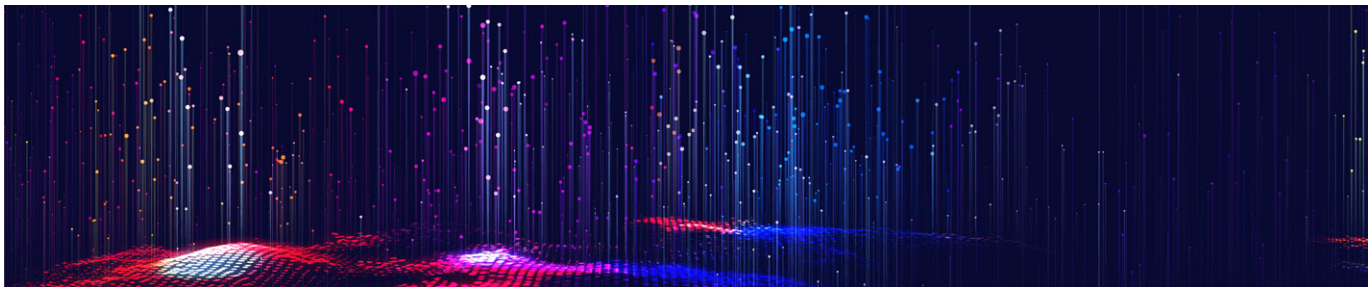


A Flume agent runs on respective host [host1, host2, host3] machines, where the Trade Execution systems are generating the position data as comma delimited files. Each line in the file is a position record. Each Flume agent is configured as follows:

- Set up a spool source that periodically polls for new records in the source and then writes them to a file channel
- The file channel provides reliability guarantees in the event of a Flume server failure. The channel capacity is configured higher than the rate of transactions
- Kafka sink is used to drain the file channel

On the HDFS side, Flume servers are set up on each respective HDFS host. [Host1, Host 2….]. Each Flume agent is configured as:

- A Kafka Source is set up, as an event source, and it is configured to point to Kafka broker setup for respective topics. This setup enables load balancing and concurrent consumption of position data. The data is written to the File Channel
- A configured HDFS sink drains the File Channel. An event serializer that converts the CSV data into a compressed Avro schema format in HDFS Sink

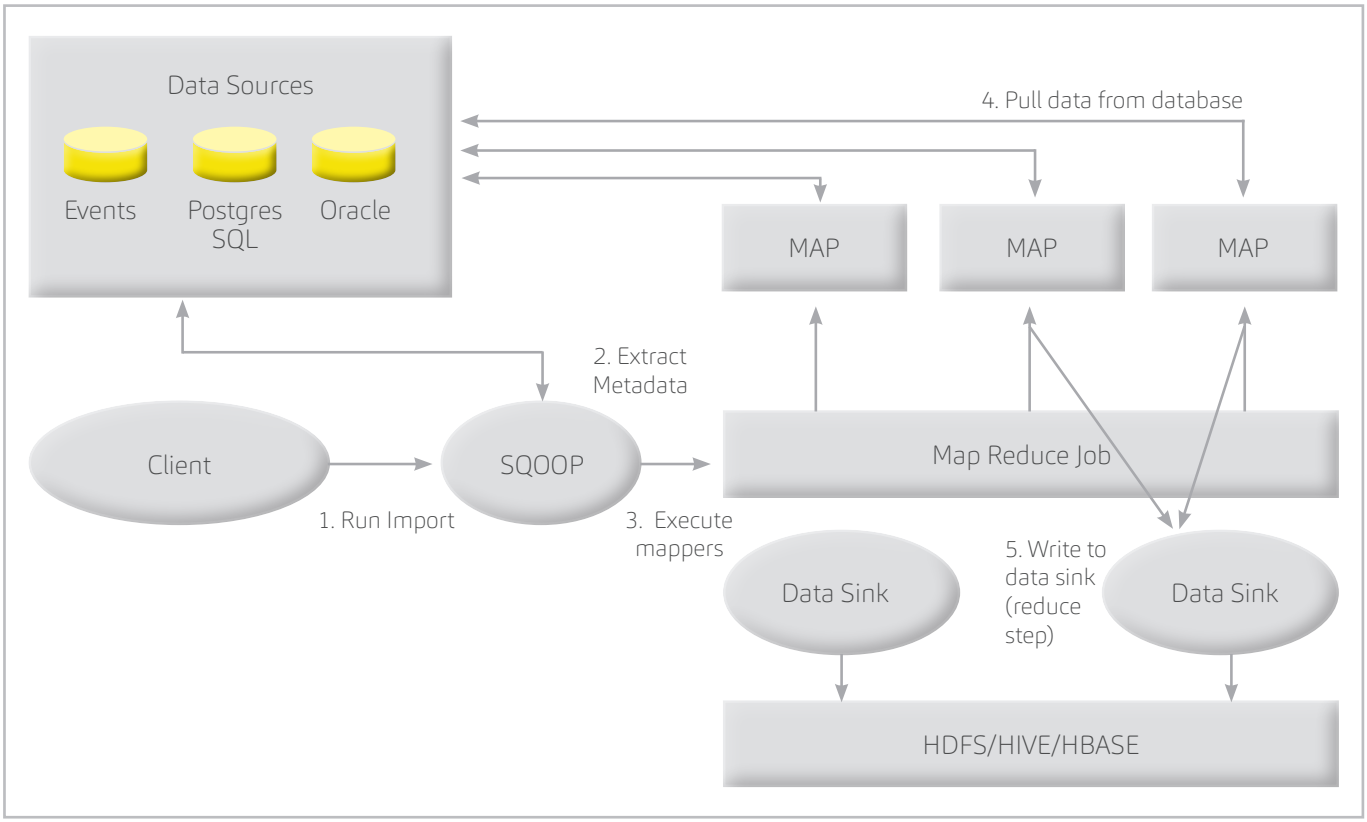


| Ingesting Data from a relational store

Traditionally, trade and position data are maintained in RDBMS stores. For running risk analytics in an optimized and concurrent manner using Spark, the prerequisite step is to store and organize this data into an HDFS cluster.

Sqoop has been designed to address the above-outlined problem. Apache Sqoop is an open source tool that allows users to extract data from a structured data store into Hadoop for further processing. Sqoop has the notion of connectors, which contains the specialized logic needed to read and write to external systems. Sqoop ships with connectors for working with a range of popular databases, including MySQL, PostgreSQL, Oracle, SQL Server, DB2, and Netezza.

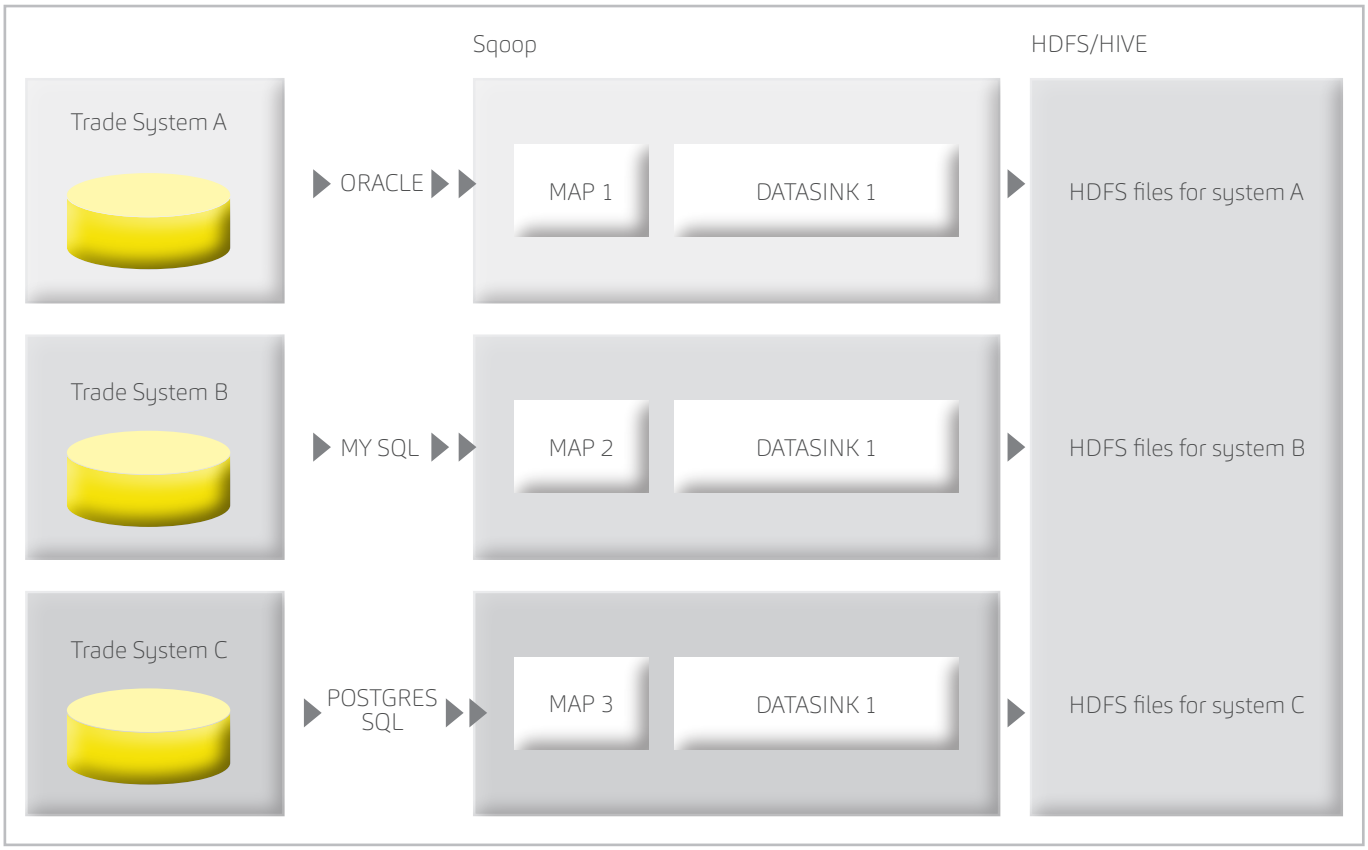
Sqoop comes with two classes of connectors: common connectors for regular reads and writes, and fast connectors that use database proprietary batch mechanisms for efficient imports. The schematic below shows how a Sqoop client can be configured to connect to an RDBMS store and import the data to an HDFS destination.



The workflow outlined in the diagram above can be summarized as follows:

- Sqoop client connects via a JDBC connector to the respective RDBMS store
- Metadata is extracted from the store [No of tables, type of columns, data types ....]
- Executes Mappers – trigger the Map process (multiple and concurrent) for respective tables
- A reduce step is executed to transform the individual records into the corresponding HDFS record
- In the reduce step, the data can be serialized as a text file or an Avro or parquet schema based binary file

Based on the above understanding, the following schematic outlines an approach for ingesting position data from multiple trade stores:



The approach above shows that for each trade store Sqoop client is configured to read specific tables, and then run a map reduce job to produce the required HDFS files. For an FRTB use-case, the generated file should be a simple comma or tab separated text file.

Sqoop creates four mappers for each specific table. We should consider increasing this for larger tables.

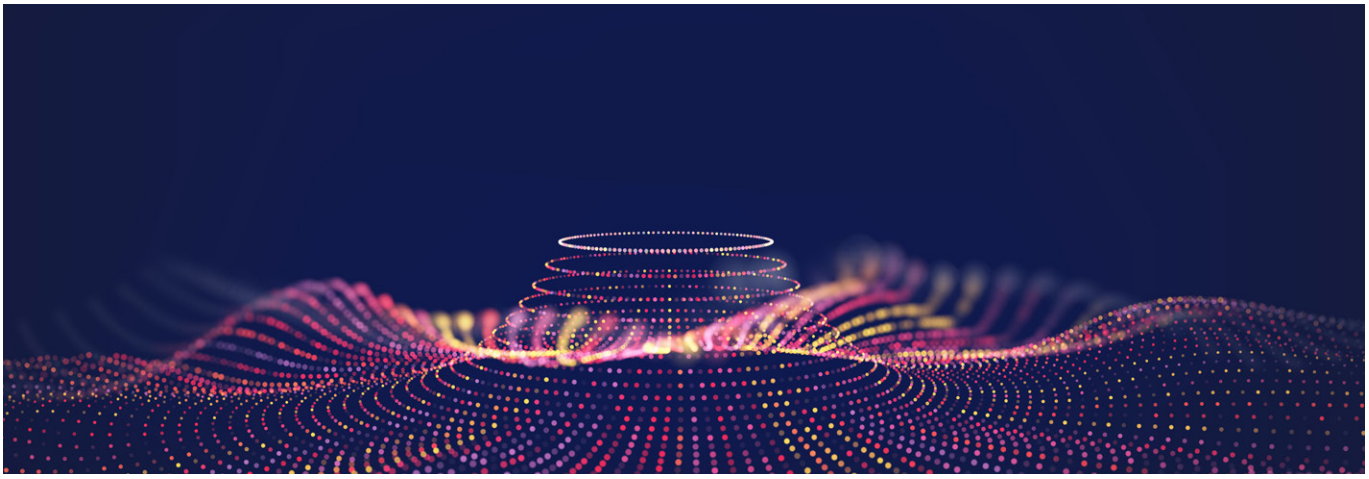
| Ingesting Stream Data

Stream data can be described as data generated by multiple sources, continuously at a fast rate. The payload is usually small, and data acquisition in small batches constructed over a configured time window. For our use case, this could be the position data for a given desk for calculating the risk charge or for backtesting.

| Comparing Kafka and Flume for Stream data

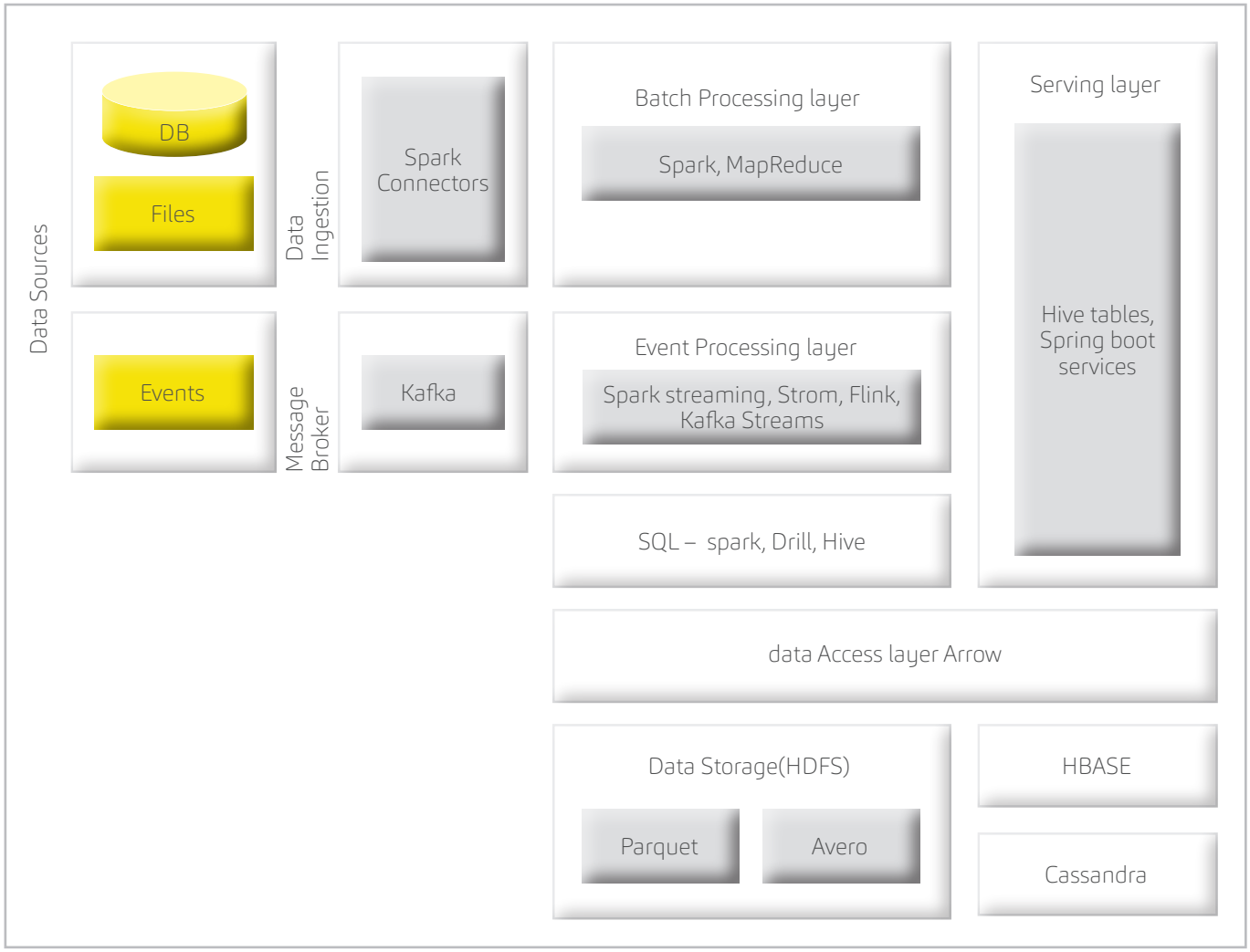
When compared with Kafka, Flume lacks the following capabilities:

- Flume does not provide ordering guarantees for events
- Scalability and reliability are largely dependent on the choice of backing store (the throughput of Flume depends on the backing store)
- Kafka was designed to provide ordering guarantees. Messages are written to a topic's partition always provide ordering guarantees (as indices are maintained for each data item)
- Kafka's store's data on a distributed partition log, which is designed to scale across the physical machines

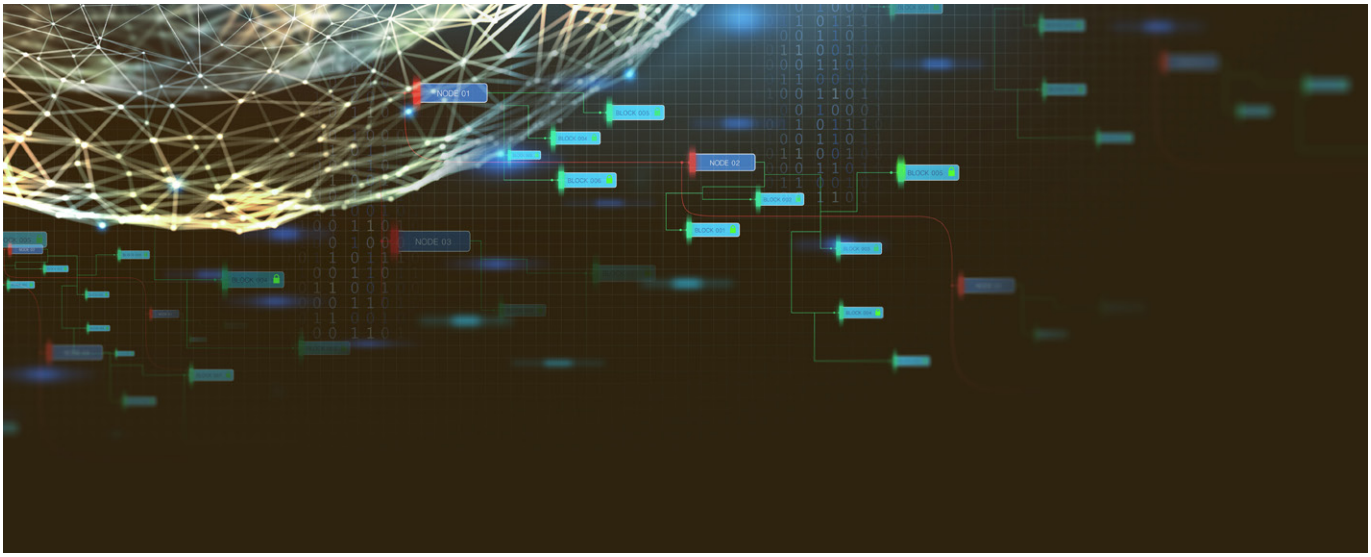


Approach 2 for Data Lake Build

"As advanced tools become more mature and there is better support available, we would recommend implementing a more unified data lake approach based on spark ecosystem

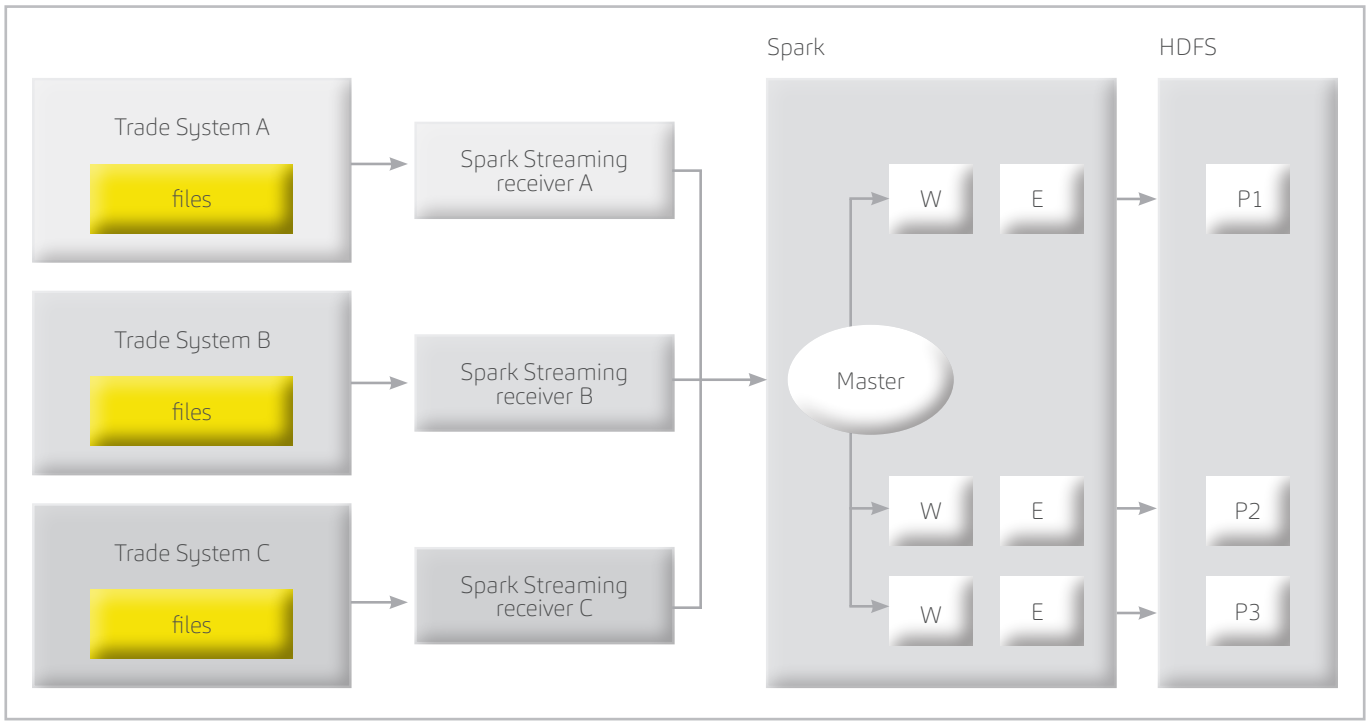


As shown in the above schematic, the goal is to make use of a unified technology stack for data ingestion. This section describes how Spark can be leveraged to ingest data from Files, Relational Store and Real-time event source and eventually land them in an HDFS cluster.



Ingestion from Files

The schematic below outlines the topology for ingesting position data from multiple trading systems in real time.



A spark streaming application is configured to run on respective trading host machines on configured intervals, and then the streaming application periodically polls the input file-based data source and creates micro batches.

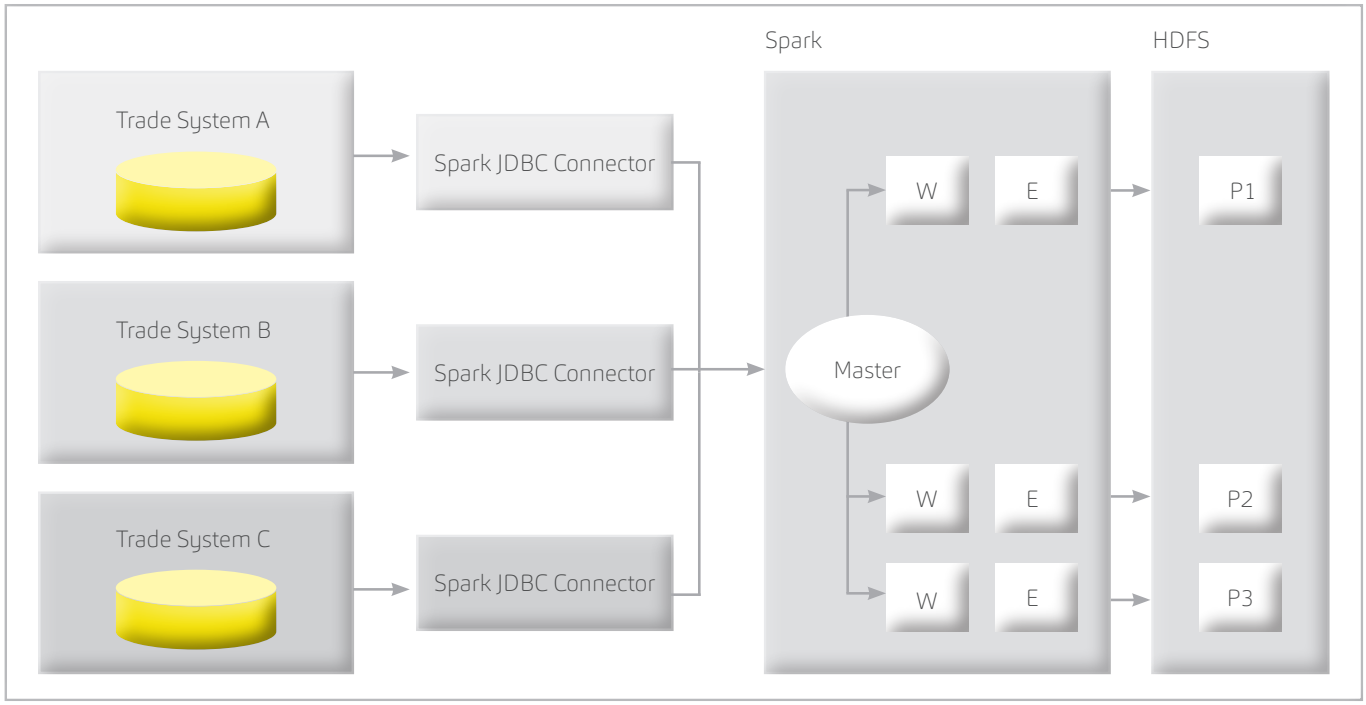
On the Spark cluster backend, each mini-batch is transformed to a delimited text file or an optimized parquet-based schema file based on the use-case. This transformation executes on the executor nodes. The spark executor nodes are collocated with HDFS data nodes.

Note: From an implementation perspective, the transformation logic is implemented in each executor by invoking the logic as a lambda function provided as an input to "foreachpartition" function.

Ingesting Data from a relation store

This section describes the Spark configuration for ingesting position data from disparate relational stores and lands them into HDFS.

The schematic below outlines the topology.



Spark has built-in support for reading data from relational databases, Spark specifically supports

dialects (data-type mappings) for the DB2, Oracle, Derby, MS SQL Server, MySQL, Postgres, and Teradata.

As shown, each Trading system host machine has a Spark submit script. The script launches a driver program. The driver program is configured to connect to the appropriate database. The entire position set is loaded as a Spark DataFrame.

A transformation function is executed on the data frame. The transformation logic transforms each row into a respective record stored on an HDFS file in parquet format. This logic executes on each executor for each respective partition. This scheme provides a scalable and concurrent approach to land data on to the HDFS cluster.

Notes on high-level implementation:

Spark's DataFrameReader API provides connectors that are JDBC compliant.

A typical load call takes the following form: val result = spark.read. jdbc("jdbc:postgresql://postgresrv/mydb", "TableName", Array("predicate"), props)

It accepts a URL, a table name, and a set of properties (in a java.util.Properties object). But you can also narrow the dataset to be retrieved with a set of predicates (expressions that can go into a where clause).

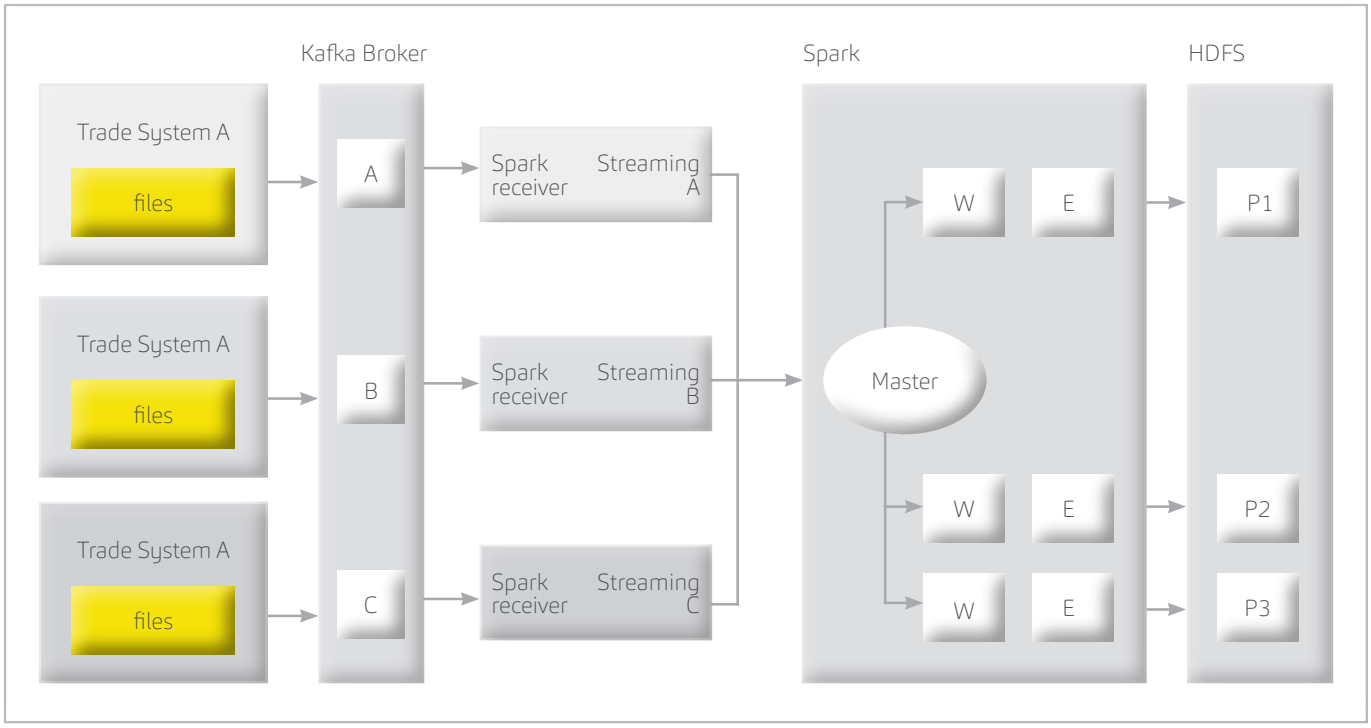
On the HDFS write, Spark provides a DataFrameWriter API. The save method provides options to control the format of the output. By default, the save method saves data on HDFS in parquet format. The method partition by can be used to specify the partitioning scheme for organizing the HDFS data for optimizing the reads.

API Doc = https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/DataFrameWriter.html#partitionBy(scala.collection.Seq)

Ingesting Stream Data

This section describes a Spark based setup for consuming real-time data from a Kafka based messaging system. The assumption here is that the upstream trading systems push the respective position data on the Kafka topics.

The schematic below outlines the topology for ingesting real-time position data from a Kafka broker setup.



As illustrated above, each trading system has a Kafka producer that submits position data on the respective topics.

Spark streaming receivers based on a direct connector are set up on the respective trading system host machines. These receivers subscribe to their respective topics. They are configured to execute functions on a remote Spark Cluster. The Spark cluster's worker nodes are co-located with HDFS nodes.

Notes on high-level implementation:

Spark direct receiver DStream of Tuples [Key = position Id , Value=position record]. A ***foreachpartition function*** *is applied on the DStream. This function converts each position record into an HDFS record.*

Following are the advantages of using a Kafka Direct connector:

- *The direct connector makes it possible to achieve exactly-once processing of incoming messages*
- *The direct consumer doesn't require ZooKeeper to store consumer offsets. It stores offset in the Spark checkpoint directory*

Data Management

This section describes the techniques to manage data at the storage layer effectively:

Data Serialization

Some of the key criteria to keep in mind while selecting the right serialization formats that natively support partitioning and have schema evolution features based are:

- Code Generation - the ability to generate code enabling us to create richer data making it easier to interact with data
- Schema Evolution - allows you to add, modify, and in some cases delete attributes, and provides backward and forward compatibility
- Compression Support- should have the ability to compress and decompress if we are dealing with huge volumes of data
- Language – is supported by the programming languages used
- Supports Partitioning – Splitting data into multiple chunks for parallel processing support

Data Organization

Data organization is one of the most challenging aspects as we have competing requirements from different user groups, once we have decided the organization of data in HDFS, we need to take of operational issues like how to partition and compact your data, whether to enable Kerberos to secure your cluster and how to manage and communicate data changes. Some of the aspects to consider are

- Partitioning the data – By date/ version number
- Data Tiering
  - Raw data (1st tier) - Keep the source of data intact
  - Derived/model data (2nd tier) – data organized around the business needs
    - These can include time series data
    - Aggregated data
    - Transformation to graph
- Data quality check (like handling null data)
- Applying Schema (checking for data types, number of columns and handling data errors)

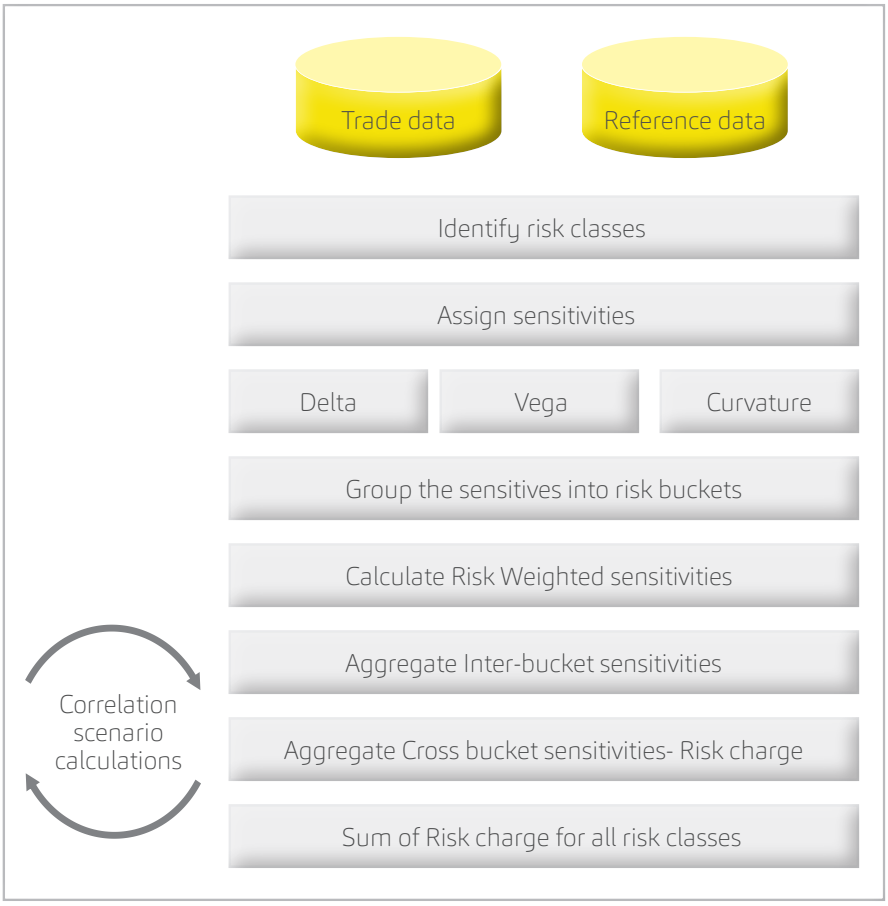
Data Catalog

It's important to create a unified metadata repository to get a unified view of data stored, providing an ability to discover and categorize the data making it searchable, support ad-hoc queries and provision for schema discovery and versioning. Depending on the complexity of the data and organization, we should decide the implementation of the data catalog.

Analytics as a Service - Use case

In this case, we have considered the FRTB (Fundamental review of the Trading Book) standardized approach as the sample use case for building a unified data analytics system. FRTB requires a major overhaul of system architecture, as wells major changes in data integration, processing, and management.

The High-level workflow for the business case is described as



To implement a unified analytics platform for the business process, we need to ensure that the Platform:

- Enables the ability to handle Big data
  - Support for batch and Real-time processing
  - Building data pipelines using functional/relational operations
  - Support a variety of data and data schema (as discussed in data lake section)
- Supports faster processing and scalable compute
  - Different data formats
  - Compaction/encryption/ memory management
  - Caching/Checkpointing
  - Partitioning for data parallelism
- Includes data consistency guarantees
- Provides support for multiple programming languages
- Has the ability to run ML/AI models on the data

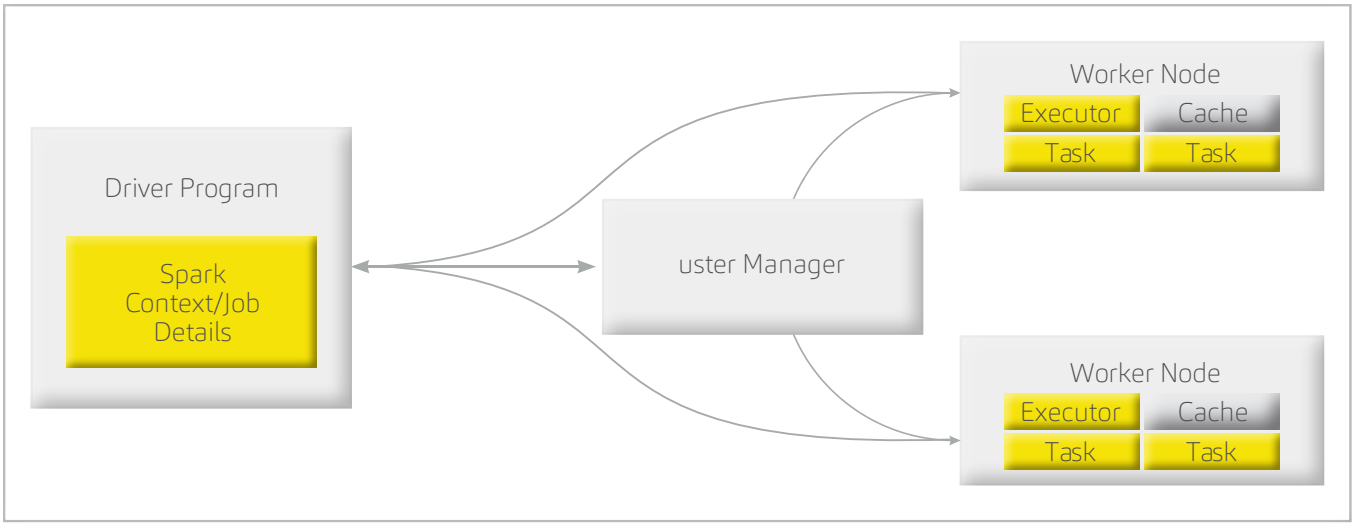
To realize the design goals stated above, the following should be achieved:

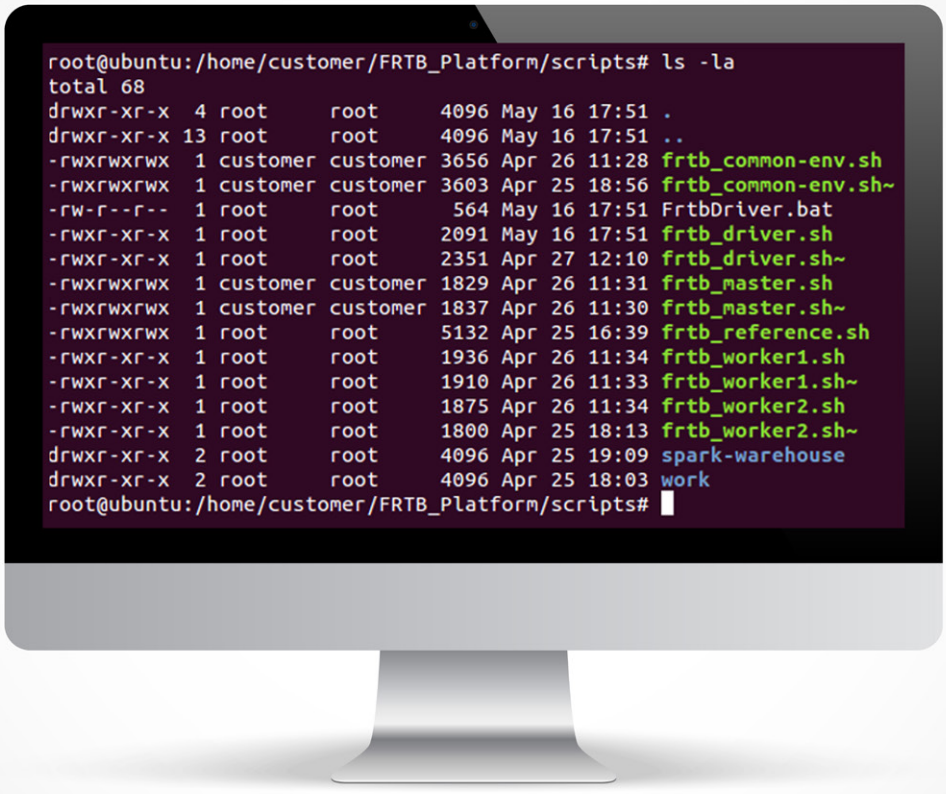
- A driver or client program runs the main application logic for FRTB calculations on a large portfolio.
- The logic implementation as a staged execution pipeline in Spark
- Stage definition as a UDF (User defined function). These UDF's are pluggable and implemented in R, Python, Java or Scala
- Each stage executes these functions concurrently on large partitioned data sets
- The number of tasks is scaled on a single machine to the number of cores on the machine [Vertical scalability]
- Based on the number of the portfolio on a given trading desk, the platform can scale horizontally across physical nodes
- A dedicated spark cluster runs each portfolio's FRTB execution

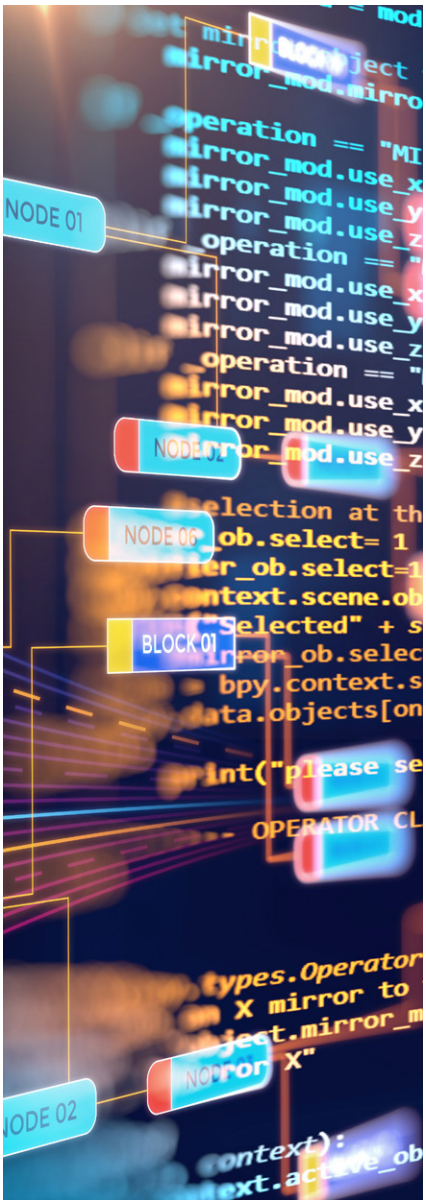
Ability to handle Big data

Based on the business requirement, modeled data pipeline can be implemented using UDF's. Spark SQL has flexible APIs (for handling corrupt data), supports a variety of data sources, and have built-in support for structured streaming. With an improvement in the catalyst optimizer and tungsten execution engine, it has evolved as one of the go-to frameworks for building data pipelines.

Batching processing







| Real-time processing

| Spark or Flink

These frameworks are used to achieve real-time processing, and the selection of the framework depends on respective use case. Below we have outlined the differences between the frameworks.

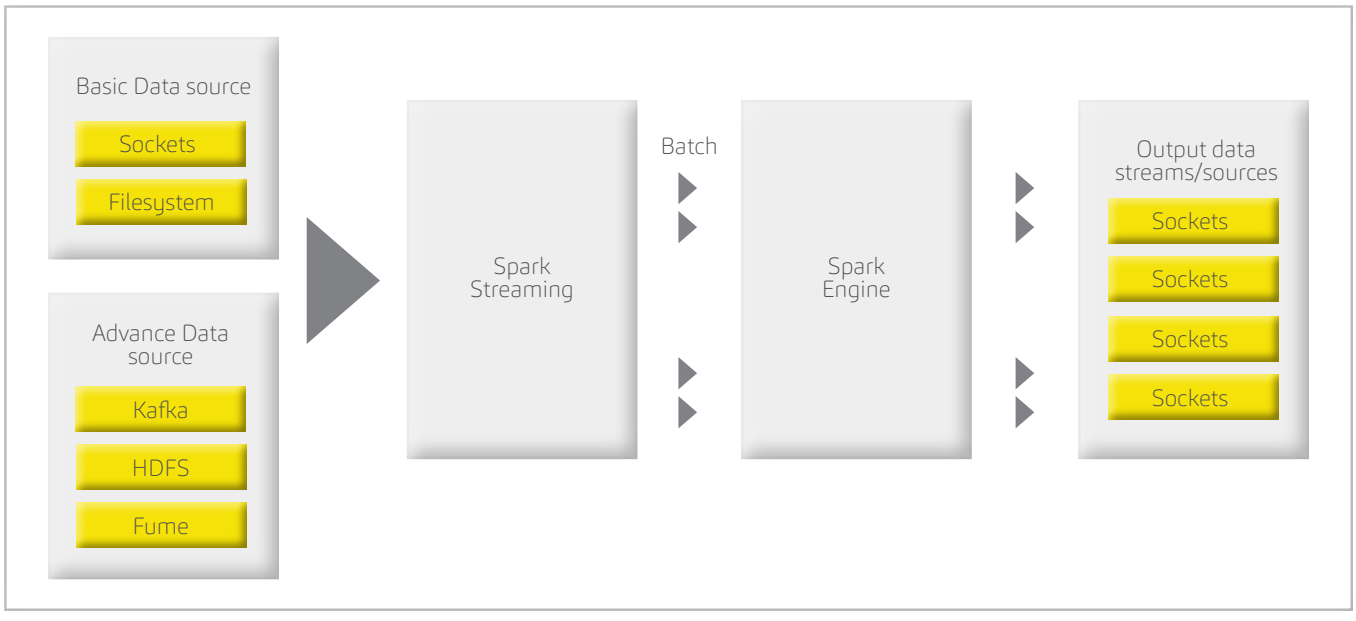
| Spark

- Spark started out as a fast batch processing system as compared to Hadoop [Spark provides in-memory transformations]
- It provides streaming capabilities in the form of micro batches.
  - At an implementation level, the API design approach is to break batches into micro-batches
  - This approach allows one to strike a balance between latency versus throughput
    - Higher the batch size, better the throughput, lower the batch size, better is the per-message processing latency
- Spark support implementing batch and streaming execution pipelines in Java, Scala, Python, and R

| Flink

- Flink's initial design goal was to provide a fast stream processing system with guarantees like reliability and scalability
- Flink's stream processing API acts on each event
  - At an implementation level, the API and data structures are designed to treat a collection of events as a stream of individual events and not batches
  - It also provides batch processing capabilities, but the approach is different and is implemented as a special case of stream processing
- Flink support implementing batch and streaming execution pipelines in Java and Scala
- Flink uses raw bytes as internal data representation, which if needed, can be hard to program.
- Doesn't have matured APIs for querying data (Flink's Table API is not quite there when compared with Spark)
- Data source integration API's are not the best and are limited in options, as compared to Spark.

In our case, we would recommend going with a Spark streaming approach (as shown in the diagram below).



Building a Data pipeline: - we can create a data pipeline as UDF's

Analytical workload – To realize analytical workload, Spark provided functional and relational operators that are used. For example, to calculate a capital risk charge of Interest rate risk for a book consisting of – 1-year Corp. Bond (USD). The inter-bucket correlation step (Aggregate Inter-bucket sensitivities) in this case

1. We have the following positions (in data frame)

BUCKET	CURVE	RWS	TENOR
USD	LIBOR	LIBOR	0.25
USD	LIBOR	-	0.50
USD	LIBOR	-	1.0

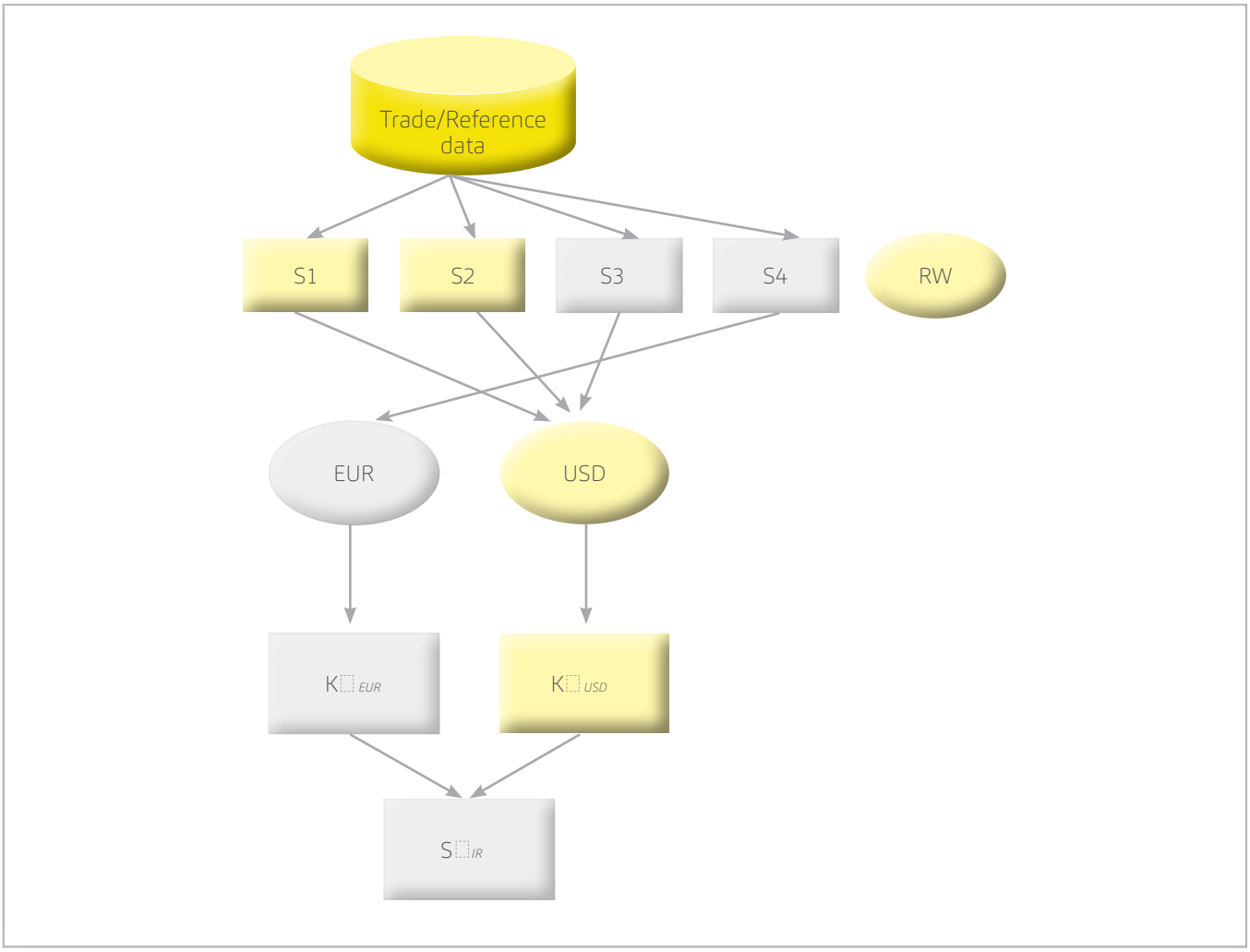
2. Compare each row against other to create a correlation matrix that can be used to realize the risk charge of a given bucket

$$(\rho_{kl} \; for \; relaizing \; the \; equation \; K_b = \sqrt{\sum_k WS_k^2 + \sum_k \sum_{(k \neq l)} \rho_{kl} WS_k WS_l})$$

3. As in this case all three rows belong to the same position, the curve quality is the same, but the tenor is different, hence the correlation matrix expected as

$$\begin{pmatrix} 100\% & 97.04\% & 91.39\% \\ 97.04\% & 100\% & 97.04\% \\ 91.39\% & 97.04\% & 100\% \end{pmatrix}$$

The diagram below illustrates the process and the pattern used to realize this.



1 | For 1-year Corporate bond (USD- bucket "USD") - Calculate Risk-weighted sensitivity for vertex .25, .5, and 1 years

Sensitivity	SA_RISK_FACTOR_CLASS	CURRENCY	BUCKET	VERTEX_TENOR	RWS
1_DELTA_VERTEX1	IR_PRICE	USD	USD	0.25	0.00017
1_DELTA_VERTEX2	IR_PRICE	USD	USD	0.5	0.00017
1_DELTA_VERTEX3	IR_PRICE	USD	USD	1	0.01584

2 | $K_b = \sqrt{\sum_k WS_k^2 + \sum_k \sum_{(k \neq l)} \rho_{kl} WS_k WS_l}$

$\rho_{k\,l}$ – *Delta correlation factor*

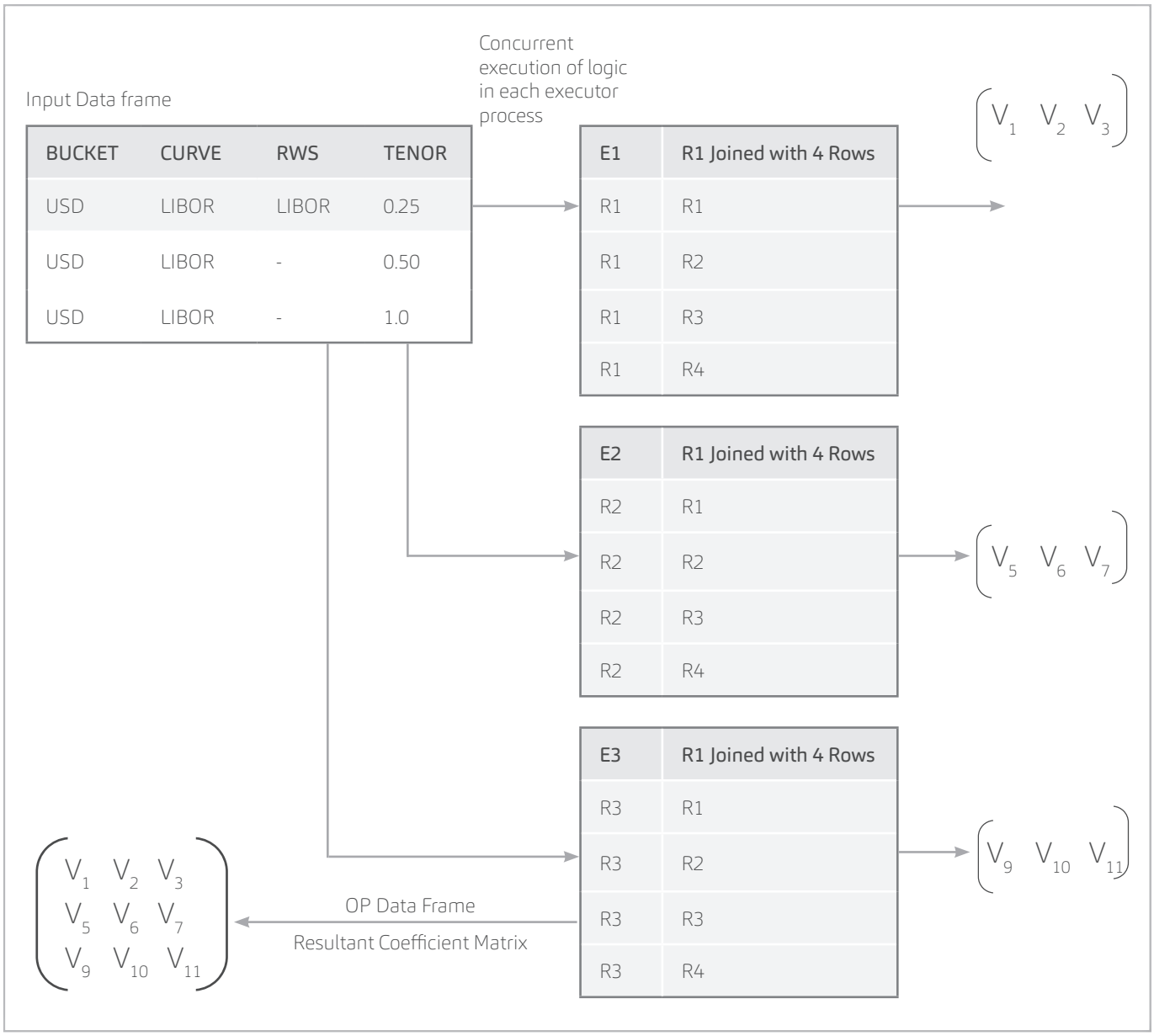
For same tenor different curve – 99.9%

For different tenor same curve - $\max[e\left(-3\% * \frac{abs(T_k - T_l)}{\min(T_K, T_l)}\right), 40\%]$

Correlation matrix for USD bucket

100%	97.04%	91.39%
97.04%	100%	97.04%
91.39%	97.04%	100%

This can be implemented using the following pattern:



Conclusion:

While financial organizations globally grapple with new and upcoming regulatory requirements, and look to move past reactive compliance and into a state of proactivity, it is essential for there to be an investment in technology and innovation to address these complex challenges. By building a unifying platform for data analytics and advanced data models using machine learning and AI, firms will see successful compliance and have better control over their data.

References:

"Hadoop In Practice" – Alex Homes

"Designing data intensive application" – Martin Kleppmann

"Data Lake for Enterprise" - Tomcy John, Pankaj Misra

"Big data -Principles and best practices of scalable real-time data systems" – Nathan Marz

"Spark in Action" - Petar Zecevi´c, Marko Bona´ci

"Apache Spark Documentation" - https://spark.apache.org/documentation.html

Minimum capital requirements for market risk (https://www.bis.org/bcbs/publ/d352.pdf)

About the Authors:

Narsimhan Bramadesam
Sr. Director - Software
Synechron Technology
Narsimhan.Bramadesam@synechron.com

Narsimhan (fondly called as NBA) is a business-focused technology leader with global experience in financial services and insurance specialized in regulatory disclosures, compliance, and risk management. He has successfully architected and managed the delivery of innovative solutions for data analytics, business intelligence, and data warehouse in enterprise environments.

A pure technology evangelist by heart, he enjoys upgrading his technical skills and has recently completed training in Artificial Intelligence and Deep Learning. His academic background includes an undergraduate degree in Mechanical Engineering and a post-graduate certificate in management.

Sandeep Mahendru
Sr. Director - Software
Synechron Technology
Sandeep.Mahendru@synechron.com

An experienced Software Architect and business-focused technology leader with a demonstrated history of working in the banking industry. Sandeep comes with very strong experience and expertise in Requirements Analysis, Enterprise Architecture, Front Office Trading systems, Large Scale Computing frameworks, Agile Methodologies, and Java EE Frameworks. He also possesses an engineering degree in BS Computer Technology focused in Computer Science from YCCE College of Engineering.

Sandeep has been highly instrumental in driving lot of innovative technology initiatives at Synechron and he is a core member of our recently launched RegTech Accelerator Program.

Global Footprint

Synechron

Digital / Business Consulting / Technology

www.synechron.com | Email: info@synechron.com