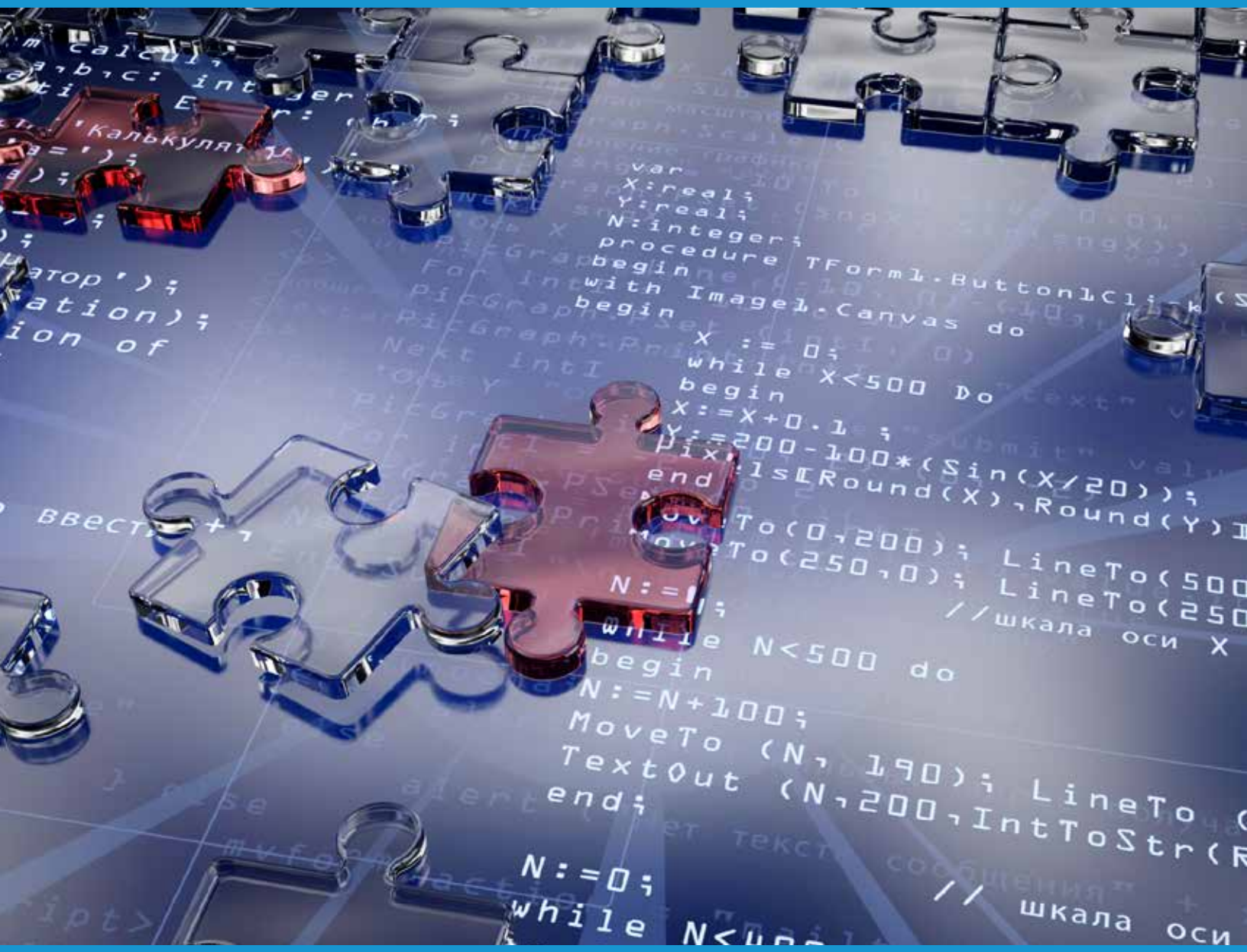
Mphasis
The Next Applied



Error Handling Framework for Data Lakes

Whitepaper by
Hitesh Chauha
Project Lead, Mphasis
Nagaraj Niranji
Sr. Software Engineer, Mphasis

Introduction

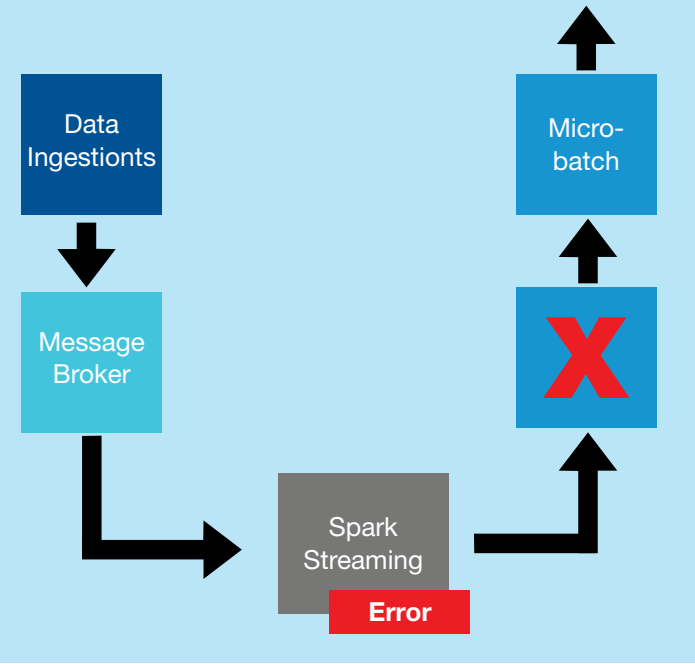
The Data process life cycle of a data lake starts from ingesting the data from multiple sources, processing it in various stages and finally storing the processed data in to multiple No-SQL databases. However, debugging this kind of application is often a tough job. Therefore robust exception and error handing mechanism is required to process the data without any failure or interrupt.
Apache Spark is a lightning fast framework for implementing highly scalable applications. The data and processing logic is spread over the data nodes for parallel processing and faster execution.

> The multiple ways you can avoid an ERROR..

Need for Effective Error Handling

Exceptions need to be treated carefully because a simple runtime exception caused by dirty source data can easily lead to the termination of the whole process. Data gets transformed in order to be joined and matched with other data, which usually gets ingested from multiple sources. While ingesting data from data sources, uncertainty about nature of data and the transformation algorithms that are often provided by the application coder, causes the job to terminate with error.

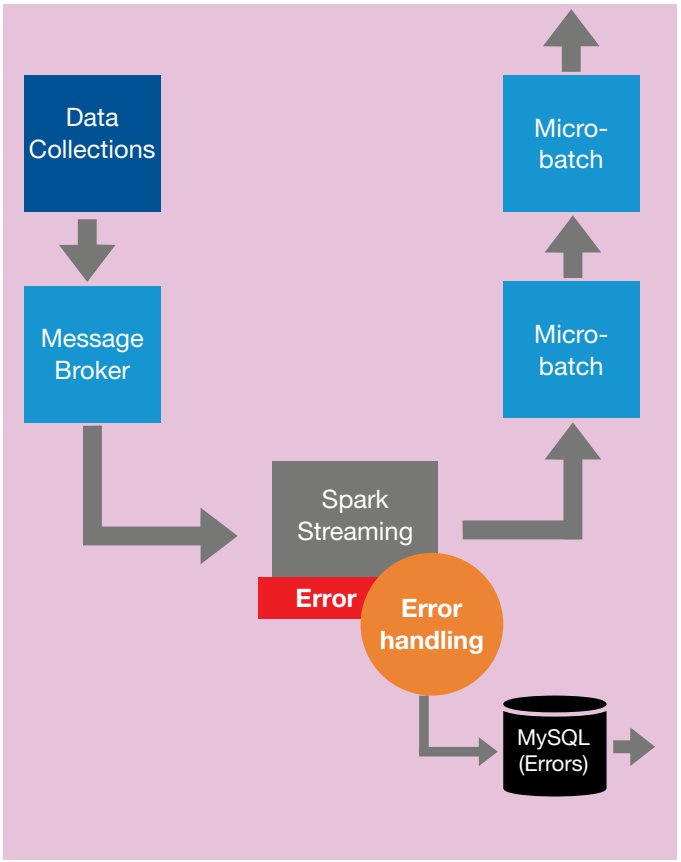
For example, take the following generic data processing model in using Spark/Kafka, without error handling in place.



- Retry micro batch n number of times
- If all the retries are failed, kill the streaming job manually
- Continuous trial and error

Error Handling Framework - Benefits

A data lake is typically sourced with millions of records. The ingestion and processing applications need to process millions of records. The probability of having wrong/dirty data in such scenario is high and non-negligible. Hence, instead of letting the process terminate or loading the bad/dirty data to the data lake, it is more desirable to implement an error handling framework that manages the errors effectively and automates the re-processing adequately. The error handling framework should also classify the errors that can be re-processed so that the process doesn't restart from the source.



Provided above is an overview of the process and its benefits.

- For low error rate, handle each error individually.
- Open connection to storage, save error packages for later processing
- Clog the stream for high error rate streams

The errors can be grouped as Fatal and Non-Fatal Errors. Fatal errors are those that cause a program to abort due to hardware, network or infrastructure issues. Non-Fatal errors are caused by the underlying files and bad data. The rest of this document provides an error handling framework for building a data lake and it uses Spark examples as needed.

> Error handling framework helps you fight against both fatal and non-fatal errors.

Fatal Errors and Exception Handling

A fatal error occurs when the Spark job cannot access the source, target, or repository. There are unexpected errors such as, data-source or target connection errors, missing required configuration with spark context errors, etc., which force the spark job to stop running and abort immediately. To handle fatal errors, we can first log an error in Error table and then allow the job to be terminated. An entry can be made to the error handling table using the JOB_TERMINATED = 'Y', which indicates that the job is failed due to a fatal error.

In case of job terminate or abort because of fatal error, we can restart the spark job. Restartability is the ability to restart the spark job if a processing step fails to execute properly. This will avoid the need of any manual cleaning up before a failed job can restart.

We need the ability to restart processing at the step where it failed as well as the ability to restart the entire

> The best way to handle a fatal error is by restarting a job which avoids the need of manual cleaning.

job. To restart processing at the step where it failed, we can use Spark Check-Point to a fault- tolerant storage system such that it can recover from failures. There are two types of data that are check pointed.

- Metadata check point - Saving the state of streaming computation job to fault-tolerant storage, like Hadoop Distributed File System(HDFS). This is used to recover from failure of the node running the driver of the streaming application (discussed in detail later). Metadata includes:
  - Configuration - The configuration that was used to create the streaming application.
  - DStream operations - The set of DStream operations that define the streaming application.
  - Incomplete batches - Batches whose jobs are queued but have not completed yet.
- Data check pointing - Saving the generated Resilient Distributed Data Sets(RDDs) to reliable storage. This is necessary in some stateful transformations that combine data across multiple batches. In such transformations, the generated RDDs depend on RDDs of previous batches, which causes the length of the dependency chain to keep increasing with time. To avoid such unbounded increase in recovery time (proportional to dependency chain), intermediate RDDs of stateful transformations are periodically check pointed to reliable storage (e.g. HDFS) to cut off the dependency chains.

4.1. Examples of some fatal error handling

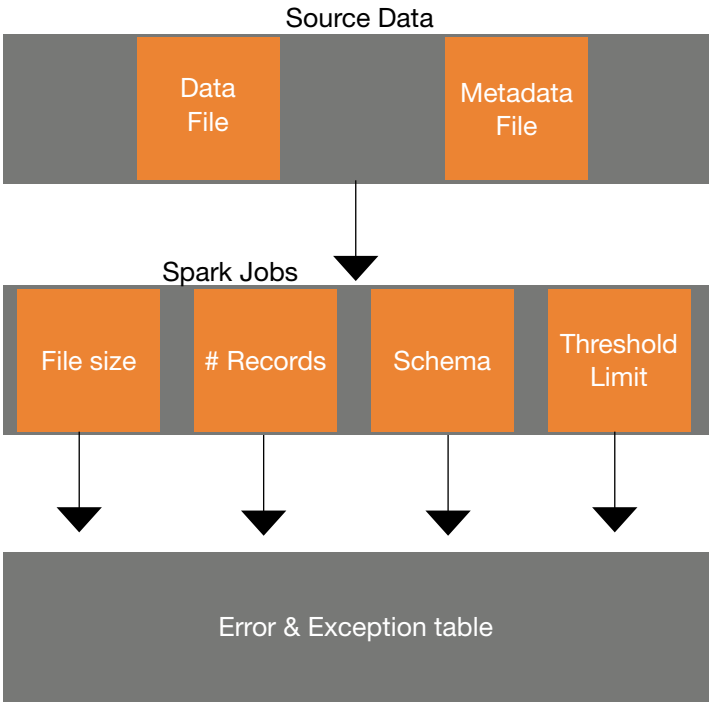
- No Such Field Error:
  - When we use our own build of Spark against an older version of Hive than what's in CDH.
  - There might be dependency issue in the classpath. Maven might be pulling down a very old version of the dependency jar
- java.lang.Null Pointer Exception:
  - This is what shows up when the main method isn't static
  - We might be referring to other class
- java.lang.Out Of Memory Error
  - If your nodes have 6g, then use 6g rather than 4g, spark.executor.memory=6g. Make sure you're using all the memory by checking the UI (it will say how much memory you're using)
  - Try using more partitions, you should have 2 - 4 per CPU. Input method editor (IME) increasing the number of partitions is often the easiest way to make a program more stable (and often faster). For huge amounts of data you may need way more than 4 per CPU, I've had to use 8000 partitions in some cases
  - Decrease the fraction of memory reserved for caching by using spark.storage.memoryFraction. If you don't use cache() or persist in your code, this might as well be 0. The default value is 0.6, which means you only get 0.4 * 4g memory for your heap. IME reducing the memory fraction often makes out of memory exception(OOMs) go away. UPDATE: Apparently with spark 1.6 we will no longer need to play with these values; spark will determine them automatically.
  - Similar to above but it shuffles memory fraction. If your job needs more shuffle memory then set it to a lower value (this might cause your shuffles to spill to disk which can have catastrophic impact on speed). Sometimes when it's a shuffle which is causing out of memory, you need to do opposite i.e. set it to something large, like 0.8, or make sure you allow your shuffles to spill to disk (it's the default since 1.0.0).

- Watch out for memory leaks as these are often caused by accidentally closing over objects you don't need in your lambdas. The way to diagnose is to look out for the "task serialized as XXX bytes" in the logs. If XXX is larger than a few KB or more than a MB, you may have a memory leak

- java.io.IOException: Filesystem closed:
  - If there is a large shuffle, it might be an out-of-memory error that causes executor failure, which then causes the Hadoop Filesystem to be closed in their shutdown hook. So, the RecordReaders in running tasks of that executor throw "java.io.IOException: Filesystem closed" exception.
- java.lang.Illegal State Exception:
  - The problem is caused by missing jars in the spark project; you need to add these jars to your project classpath
  - Hadoop and Spark built into an app, or the cluster's version of Spark are not matched with the Hadoop version.
- java.lang.Unsupported Class Version Error:
  - Setting JAVA_HOME at whole cluster level
  - Add compatible dependencies for spark core & Java in pom.xml
- Error:scala.reflect.internal.Missing Requirement Error:
  - This happens when you don't have all of the dependencies for Scala reflection loaded by the primordial classloader. For running apps from SBT(an open source build tool for Scala) setting fork := true should do the trick.
  - Find your launch configuration and go to "Classpath"
  - Remove Scala Library and Scala Compiler from the "Bootstrap" entries
  - Add (as external jars) scala-reflect, scala-library and scala-compiler to user entries
  - Make sure to add the right version

Non-Fatal Errors and Exception Handling

5.1. File Level Errors



The source data includes different data files and their corresponding metadata files which consists of metadata such as file size, no. of records, schema etc. A series of spark jobs can be designed to run on the source files once the data is ingested in the HDFS. These jobs handle the error exceptions for different scenarios.

Following are the list of file level checks that can be performed without getting into the semantics of the underlying data.

1. File Size
2. # records
3. Schema
4. Threshold Limit

The results of the above file level validations are stored in an error handling table. If a file does not match any one of these validations, then that file is corrupted and can't be reprocessed in its current state. The file needs to be re-ingested and then processed.

5.2. Data Level Errors or Data Quality

The data level errors can be categorized as soft errors and hard errors.

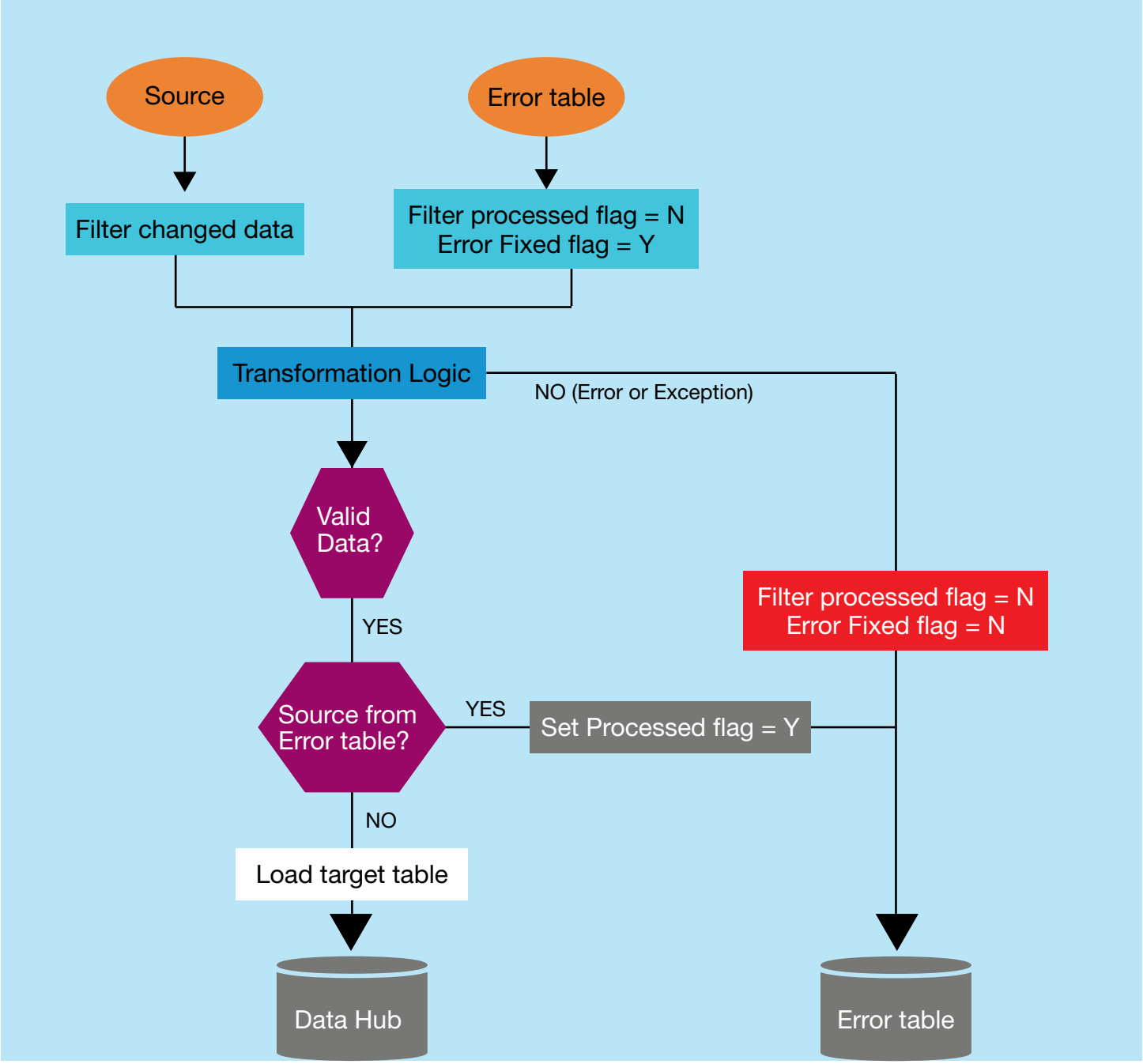
Hard Error/Exceptions: These are the validations that are mandatory to process the record further to the data lake. For example, Employee ID in a salary table can't be blank. A record is not valid if it does not meet this validation and such records are dropped out from further processing and not made to the target data lake. These records with respective error information are loaded to the error table for further analysis and re-processing.

> Non-fatal errors can be handled by introducing file level validations..

Soft Error/Exceptions: The validations that are nice to have or desired on a data record are soft exceptions. These records do not disturb the defined quality/integrity of the data lake, but they do not meet certain pre-defined validation rules. Error handing framework should allow them to be processed further and also loaded to the error table for further analysis, prevention and hence, reduction of the errors.

Typically, the set of data quality validations and checks are defined by the Business and IT Engineers together.

5.3. Automating the Data Quality Checks



- The data is ingested to the data lake and initially set Filter processed flag = N & Error fixed flag =Y
- If the data is:
  - Valid (meets the pre-defined validation rules) Data - then it is checked if it came from the error table after fixing the error. If yes then set the processed flag= Y and load the data to Data hub. If data is not from the error table then it is directly loaded to Data hub.
  - Invalid Data - If the data doesn't the validation rules then set the processed flag = N & Error fixed flag = N and load the data along with respective error details to the error table.
- After the error data is corrected the iteration again continuous from validation.

Error Table Defined

Error Table

JOB_ID	Decimal(50)
JOB_SRT_ TIME	TimeStamp(19)
SRC_LIST	VarChar(200)
TRG_LIST	VarChar(200)
JOB_TERMINATED	VarChar(1)
ERR_SEQ_ID	Decimal(10)
ERR_MSG	VarChar(2000)
ERR_TIME	TimeStamp(19)
USR_NAME	VarChar(50)
ERR_DATA_COL_1	VarChar(200)
ERR_DATA_COL_2	VarChar(200)
ERR_DATA_COL_3	VarChar(200)
.	
.	
ERR_DATA_COL_N	VarChar(200)
ERR_FIXED_FLAG	VarChar(1)
ERR_PROCESSED_FLAG	VarChar(1)

Columns description:

1. JOB_ID: Contains the JOB_ID for the corresponding job which processes the data
2. JOB_SRT_TIME : Contains the job start time
3. SRC_LIST : Contains the list of source from where the data is expected as input for job
4. TRG_LIST : Contains list of target storage to which data is expected to be stored by job
5. JOB_TERMINATED : Indicator or a flag that is set to Y, if job is terminated after an exception
6. ERR_SEQ_ID: Contains the ERR_SEQ_ID for the corresponding error or exception thrown.
7. ERR_MSG : Describes Error message in detail
8. ERR_TIME : Time of error thrown by job
9. USR_NAME : User details
10. ERR_DATA_COL_N : Source data elements
11. ERR_FIXED_FLAG : Value is set to Y, if data issue is fixed and ready to process, else N.
12. ERR_PROCESSED_FLAG : Value is set to Y, if data is processed else set to N.

Hitesh Chauhan

Project Lead – Mphasis Analytics Practice

Hitesh has 10 years of experience in designing, developing analytics solutions using technologies like Big Data and Java. Heis well experienced in Hadoop and Spark. He has played multiple roles in the IT industry - developer, architect and team lead. Data management, data analytics are some of the key areas he is interested in. Hitesh is currently working as a project lead at Mphasis and is responsible for developing spark transformations, exception handling and building the presentation layers.

Nagaraj Niranji

Sr. Software Engineer – Mphasis Analytics Practice

Nagaraj has 5 years of experience in designing and developing Java and BigData solutions using Hadoop and Spark technologies. He has played multiple roles in the IT industry; he has worked as a developer and a team lead.

Nagaraj is a Senior Software Engineer at Mphasis for five months now. Prior to this he worked for Cognizant Technology Solutions and HCL Technologies. He is a bachelor of engineering from MS Ramaiah Institute of Technology.

About Mphasis

Mphasis (BSE: 526299; NSE: MPHASIS) applies next-generation technology to help enterprises transform businesses globally. Customer centricity is foundational to Mphasis and is reflected in the Mphasis' Front2Back™ Transformation approach. Front2Back™ uses the exponential power of cloud and cognitive to provide hyper-personalized (C = X2C2™ = 1) digital experience to clients and their end customers. Mphasis' Service Transformation approach helps 'shrink the core' through the application of digital technologies across legacy environments within an enterprise, enabling businesses to stay ahead in a changing world. Mphasis' core reference architectures and tools, speed and innovation with domain expertise and specialization are key to building strong relationships with marquee clients. To know more, please visit www.mphasis.com

For more information, contact: marketinginfo@mphasis.com

USA
460 Park Avenue South
Suite #1101
New York, NY 10016, USA
Tel.: +1 212 686 6655

USA
226 Airport Parkway
San Jose
California, 95110
USA

UK
88 Wood Street
London EC2V 7RS, UK
Tel.: +44 20 8528 1000

INDIA
Bagmane World Technology Center
Marathahalli Ring Road
Doddanakundhi Village
Mahadevapura
Bangalore 560 048, India
Tel.: +91 80 3352 5000

VAS 30/11/16 US LETTER BASIL4187

www.mphasis.com