**Unit- IV: Data Ingestion & Other Hadoop Eco-Systems**

**TOPICS COVERED IN THIS SESSION:**

**Avro**: Avro Data Types and Schemas –

In-Memory Serialization and Deserialization - The Specific API - Avro Datafiles - Avro Tools

**Parquet:**

Data Model - Nested Encoding - Parquet File Format - Parquet Configuration - Writing and Reading Parquet Files - Avro, Protocol Buffers, and Thrift

**INTRODUCTION**

* Avro is an open source project that provides data serialization and data exchange services for Apache Hadoop. These services can be used together or independently. Avro facilitates the exchange of big data between programs written in any language. With the serialization service, programs can efficiently serialize data into files or into messages. The data storage is compact and efficient. Avro stores both the data definition and the data together in one message or file.
* Avro stores the data definition in JSON format making it easy to read and interpret; the data itself is stored in binary format making it compact and efficient. Avro files include markers that can be used to split large data sets into subsets suitable for [***Apache MapReduce***](https://www.ibm.com/in-en/analytics/hadoop/mapreduce)***processing***.
* Some data exchange services use a code generator to interpret the data definition and produce code to access the data. Avro doesn't require this step, making it ideal for scripting languages.
* A key feature of Avro is robust support for data schemas that change over time — often called ***schema evolution***. Avro handles schema changes like missing fields, added fields and changed fields; as a result, old programs can read new data and new programs can read old data. Avro includes APIs for Java, Python, Ruby, C, C++ and more. Data stored using Avro can be passed from programs written in different languages, even from a compiled language like C to a scripting language like Apache Pig.
* An Avro datafile has a metadata section where the schema is stored, which makes the file self-describing. Avro datafiles support compression and are splittable, which is crucial for a MapReduce data input format. Avro supports all data processing frameworks like Pig, Hive, Crunch, Spark which can read and write Avro datafiles. Avro can be used for RPC also.

**Avro Data Types and Schemas**

* Avro defines a small number of primitive data types, which can be used to build application-specific data structures by writing schemas. For interoperability, implementations must support all Avro types.

Avro’s primitive types are listed below:

|  |  |  |
| --- | --- | --- |
| **Avro’s primitive data types** | | |
| **TYPE** | **DESCRIPTION** | **SCHEMA** |
| null | The absence of a value | "null" |
| boolean | A binary value | "boolean" |
| int | 32-bit signed integer | "int" |
| long | 64-bit signed integer | "long" |
| float | Single-precision (32-bit) IEEE 754 floating-point number | "float" |

Avro also defines the complex types which are listed below:

|  |  |  |
| --- | --- | --- |
| **Avro’s complex data types** | | |
| **TYPE** | **DESCRIPTION** | **SCHEMA** |
| array | An ordered collection of objects. All  objects in a particular array must have  the same schema. | {  **"type"**: "array",  **"items"**: "long"  } |
| map | An unordered collection of key-value  pairs. Keys must be strings and values  may be any type, although within a  map, all values must have the  same schema. | {  **"type"**: "map",  **"values"**: "string"  } |
| record | record A collection of named fields of any type. | {  **"type"**: "record",  **"name"**: "WeatherRecord",  **"doc"**: "A weather reading.",  **"fields"**: [  {**"name"**: "year", **"type"**: "int"},  {**"name"**: "temperature", **"type"**: "int"},  {**"name"**: "stationId", **"type"**: "string"}  ]  } |
| enum | A set of named values. | {  **"type"**: "enum",  **"name"**: "Cutlery",  **"doc"**: "An eating utensil.",  **"symbols"**: ["KNIFE", "FORK", "SPOON"]  } |
| fixed | A fixed number of 8-bit unsigned bytes. | {  **"type"**: "fixed",  **"name"**: "Md5Hash",  **"size"**: 16  } |
| union | A union of schemas. A union is  represented by a JSON array, where each element in the array is a schema. Data represented by a union must match one of the schemas in the union. | [  "null",  "string",  {**"type"**: "map", **"values"**: "string"}  ] |

**AVRO API mapping to language types**

Each Avro language API has a representation for each Avro type that is specific to the language. This is known as mapping of Avro types to the language data types. This mapping is of 3 types:

* All languages support a ***dynamic mapping***, which can be used even when the schema is not known ahead of runtime. Java calls this the ***Generic mapping.***
* The Java and C++ implementations can generate code to represent the data for an Avro schema. This Code generation is called the ***Specific mapping*** in Java.
* Java has a third mapping, the ***Reflect mapping***, which maps Avro types onto pre-existing Java types using reflection.

**In-Memory Serialization and Deserialization**

* Avro provides APIs for serialization and deserialization that are useful when you want to integrate Avro with an existing system, such as a messaging system where the framing format is already defined. In other cases, consider using Avro’s datafile format.

simple Avro schema for representing a pair of strings as a record:

{

**"type"**: "record",

**"name"**: "StringPair",

**"doc"**: "A pair of strings.",

**"fields"**: [

{**"name"**: "left", **"type"**: "string"},

{**"name"**: "right", **"type"**: "string"}

]

}

If this schema is saved in a file on the classpath called *StringPair.avsc* (***.avsc***is the conventional extension for an ***Avro schema***), we can load it using the following two lines of code:

Schema.Parser parser = **new** Schema.Parser();

Schema schema = parser.parse(

getClass().getResourceAsStream("StringPair.avsc"));

We can create an instance of an Avro record using the Generic API as follows:

GenericRecord datum = **new** GenericData.Record(schema);

datum.put("left", "L");

datum.put("right", "R");

Next, we serialize the record to an output stream:

ByteArrayOutputStream out = **new** ByteArrayOutputStream();

DatumWriter<GenericRecord> writer = **new** GenericDatumWriter<GenericRecord> (schema);

Encoder encoder = EncoderFactory.get().binaryEncoder(out, **null**);

writer.write(datum, encoder);

encoder.flush();

out.close();

There are two important objects here: the ***DatumWriter*** and the ***Encoder***. A ***DatumWriter*** translates data objects into the types understood by an ***Encoder***, which the latter writes to the output stream.

**Parquet:**

Data Model - Nested Encoding - Parquet File Format - Parquet Configuration - Writing and Reading Parquet Files - Avro, Protocol Buffers, and Thrift

**INTRODUCTION**

Apache Parquet is a columnar storage format that can efficiently store nested data.

Columnar formats are attractive since they enable greater efficiency, in terms of both

*file size* and *query performance*. File sizes are usually smaller than row-oriented equivalents since in a columnar format the values from one column are stored next to each other, which usually allows a very efficient encoding. A column storing a timestamp can be encoded by storing the first value and the differences between subsequent values. Query performance is improved too since a query engine can skip over columns that are not needed to answer a query.

***Example: To store client ID, Name, Address and Mobile Number in a table called Clients. Take these data for an instance:***

***Row-oriented data representation***

100, Dylan Jay, 137 Cairns Street, 555-1212  
112, Tara Jay, 137 Dairns Road, 444-2222

***Column-oriented data representation***

ID Number: {100,112}

Name: {Dylan Jay,Tara Jay}  
Address: {137 Cairns Street, 137 Dairns Road}  
Mobile: {555-1212,444-2222}

* A key strength of Parquet is its ability to store data that has a deeply *nested* structure in true columnar fashion. This is important since schemas with several levels of nesting are common in real-world systems. Parquet uses a novel technique for storing nested structures in a flat columnar format with little overhead.
* Nested fields can be read independently of other fields, resulting in significant performance improvements.
* Another feature of Parquet is the large number of tools that support it as a format. The engineers at Twitter and Cloudera who created Parquet wanted it to be easy to try new tools to process existing data, so to facilitate this they divided the project into a specification (*parquet-format*). Which defines the file format in a language-neutral way, and implementations of the specification for different languages (Java and C++) , made it easy for tools to read or write Parquet files.

**Data Model**

* Parquet defines a small number of primitive types, listed below

|  |  |
| --- | --- |
| **Type** | **Description** |
| boolean | Binary value |
| int32 | 32-bit signed integer |
| int64 | 64-bit signed integer |
| int96 | 96-bit signed integer |
| float | Single-precision (32-bit) IEEE 754 floating-point number |
| double | Double-precision (64-bit) IEEE 754 floating-point number |
| binary | Sequence of 8-bit unsigned bytes |
| fixed\_len\_byte\_array | Fixed number of 8-bit unsigned bytes |

The data stored in a Parquet file is described by a schema, which has at its root a message containing a group of fields. Each field has a repetition (required, optional, or repeated), a type, and a name. Here is a simple Parquet schema for a weather record:

**message WeatherRecord {**

**required int32 year;**

**required int32 temperature;**

**required binary stationId (UTF8);**

**}**

Notice that there is no primitive string type. Instead, Parquet defines logical types that specify how primitive types should be interpreted, so there is a separation between the serialized representation (the primitive type) and the semantics that are specific to the application (the logical type). Strings are represented as binary primitives with a UTF8 annotation.

For example,

**LIST An ordered collection of values. Annotates group.**

**message m {**

**required group a (LIST) {**

**repeated group list {**

**required int32 element;**

**}**

**}**

**}**

**MAP An unordered collection of key-value pairs. Annotates group.**

**message m {**

**required group a (MAP) {**

**repeated group key\_value {**

**required binary key (UTF8);**

**optional int32 value;**

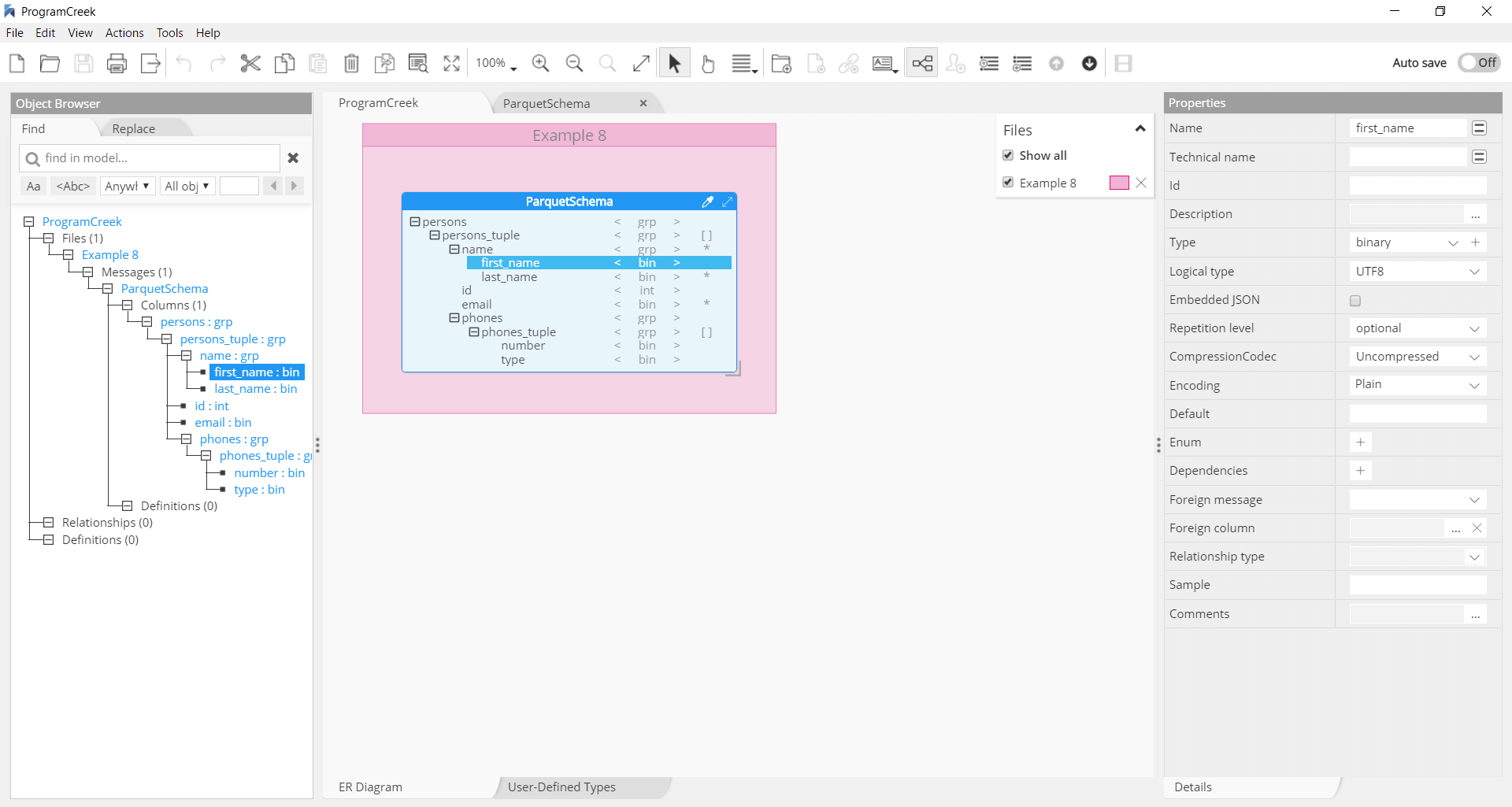
**}**

**}**

**}**

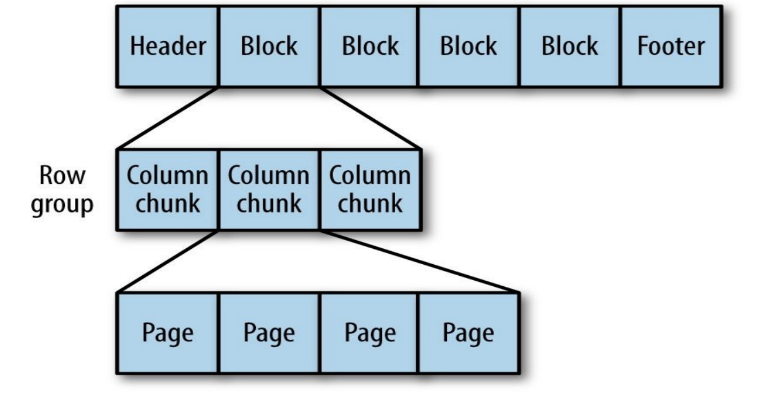
Complex types in Parquet are created using the ***group*** type, which adds a layer of nesting.

Another Example for Parquet group type schema representation:



Parquet File Format

* A Parquet file consists of a header followed by one or more blocks, terminated by a footer. The header contains only a ***4-byte magic number, PAR1,*** that identifies the file as being in Parquet format, and all the files in metadata are stored in the footer, which includes the format version, the schema, any extra key-value pairs, and metadata for every block in the file. ***The final two fields in the footer are a 4-byte field encoding the footer metadata length & the magic number again (PAR1).***
* The consequence of storing the metadata in the footer is that reading a Parquet file requires an initial seek to the end of the file (minus 8 bytes).
* Unlike sequence files and Avro datafiles, where the metadata is stored in the header and sync markers are used to separate blocks, Parquet files don’t need sync markers since the block boundaries are stored in the footer metadata.
* Parquet files are splitable, since the blocks can be located after reading the footer and can then be processed in parallel (by MapReduce, for example).



* Each block in a Parquet file stores a row group, which is made up of column chunks containing the column data for those rows. The data for each column chunk is written in pages.
* Each page contains values from the same column, making a page a very good candidate for compression since the values are likely to be similar.
* Most of the time Parquet files are processed using higher-level tools like Pig, Hive, or Impala.

Avro, Protocol Buffers, and Thrift

* Most applications will prefer to define models using a framework like Avro, Protocol Buffers, or Thrift, and Parquet caters to all these cases. Instead of ParquetWriter and ParquetReader, use AvroParquetWriter, ProtoParquetWriter, or ThriftParquetWriter, and the respective reader classes.
* These classes take care of translating between Avro, Protocol Buffers, or Thrift schemas and Parquet schemas. They also take care of performing the equivalent mapping between the framework types and Parquet types, which means there is no need for the user to deal with Parquet schemas directly.

**Sqoop:**

Getting Sqoop - Sqoop Connectors - A Sample Import - Text and Binary File Formats - Generated Code - Additional Serialization Systems - Imports: A Deeper Look - Controlling the Import - Imports and Consistency - Incremental Imports - Direct-Mode Imports - Working with Imported Data - Imported Data and Hive - Importing Large Objects - Performing an Export - Exports: A Deeper Look - Exports and Transactionality - Exports and SequenceFiles

**INTRODUCTION**

* ***Apache Sqoop*** is an open-source tool that allows users to extract data from a structured data store into Hadoop for further processing. This processing can be done with MapReduce programs or other higher-level tools such as Hive.
* When the final results of an analytic pipeline are available, Sqoop can export these results back to the data store for consumption by other clients.

***You can run Sqoop by simply typing sqoop at the command line.***

Sqoop is organized as a set of tools or commands. ***help*** is the name of one such tool; it can print out the list of available tools, like this:

% **sqoop help**

usage: sqoop COMMAND [ARGS]

Available commands:

codegen Generate code to interact with database records

create-hive-table Import a table definition into Hive

eval Evaluate a SQL statement and display the results

export Export an HDFS directory to a database table

help List available commands

import Import a table from a database to HDFS

import-all-tables Import tables from a database to HDFS

job Work with saved jobs

list-databases List available databases on a server

list-tables List available tables in a database

merge Merge results of incremental imports

metastore Run a standalone Sqoop metastore

version Display version information

See 'sqoop help COMMAND' for information on a specific command. As it explains, the help tool can also provide specific usage instructions on a particular tool when you provide that tool’s name as an argument:

% **sqoop help import**

usage: sqoop import [GENERIC-ARGS] [TOOL-ARGS]

***Common arguments:***

--connect <jdbc-uri> Specify JDBC connect string

--driver <class-name> Manually specify JDBC driver class to use

--hadoop-home <dir> Override $HADOOP\_HOME

--help Print usage instructions

-P Read password from console

--password <password> Set authentication password

--username <username> Set authentication username

--verbose Print more information while working

**...**

**Sqoop Connectors**

Sqoop *connector* is a modular component that uses this framework to enable Sqoop

imports and exports. Sqoop ships with connectors for working with a range of popular

databases, including MySQL, PostgreSQL, Oracle, SQL Server, DB2, and Netezza. There is also a generic JDBC connector for connecting to any database that supports Java’s JDBC protocol. Sqoop provides optimized MySQL & Oracle, and Netezza connectors that use database-specific APIs to perform bulk transfers more efficiently.

**A Sample Import**

We’ll use MySQL, which is easy to use and available for many platforms.

Step-1: **Log in and create a database.**

**Step-2: % mysql -u root -p**

Enter password:

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 235

Server version: 5.6.21 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> **CREATE DATABASE <database name>**

Step-3: **Create a table and insert data into the table**

Step-4**: Let’s use Sqoop to import this table into HDFS:**

% **sqoop import --connect jdbc:mysql://localhost/hadoopguide \**

> **--table widgets -m 1**

...

14/10/28 21:36:23 INFO tool.CodeGenTool: Beginning code generation

...

14/10/28 21:36:28 INFO mapreduce.Job: Running job: job\_1413746845532\_0008

14/10/28 21:36:35 INFO mapreduce.Job: Job job\_1413746845532\_0008 running in

uber mode : false

14/10/28 21:36:35 INFO mapreduce.Job: map 0% reduce 0%

14/10/28 21:36:41 INFO mapreduce.Job: map 100% reduce 0%

14/10/28 21:36:41 INFO mapreduce.Job: Job job\_1413746845532\_0008 completed

successfully

...

14/10/28 21:36:41 INFO mapreduce.ImportJobBase: Retrieved 3 records.

Sqoop’s import tool will run a MapReduce job that connects to the MySQL database

and reads the table. By default, this will use four map tasks in parallel to speed up the import process. Each task will write its imported results to a different file, but all in a common directory.

We specified that Sqoop should use a single map task (-m 1) so we get a single file in HDFS. We can inspect this file’s contents like so:

% **hadoop fs -cat widgets/part-m-00000**

**1,sprocket,0.25,2010-02-10,1,Connects two gizmos**

**2,gizmo,4.00,2009-11-30,4,null**

**3,gadget,99.99,1983-08-13,13,Our flagship product**

By default, Sqoop will generate comma-delimited text files for our imported data. Delimiters can be specified explicitly, as well as field enclosing and escape characters, to allow the presence of delimiters in the field contents. The command-line arguments

that specify delimiter characters, file formats, compression.

**Text and Binary File Formats**

* Sqoop is capable of importing into a few different file formats. Text files (the default) offer a human-readable representation of data, platform independence, and the simplest structure. However, they cannot hold binary fields (such as database columns of type VARBINARY) and distinguishing between null values and String-based fields containing the value "null" can be problematic.
* To handle these conditions, Sqoop also supports SequenceFiles, Avro datafiles, and Parquet files. These binary formats provide the most precise representation possible of the imported data.

**Generated Code**

* In addition to writing the contents of the database table to HDFS, Sqoop also provides you with a generated Java source file (*widgets.java*) written to the current local directory.

**% sqoop codegen --connect jdbc:mysql://localhost/hadoopguide \**

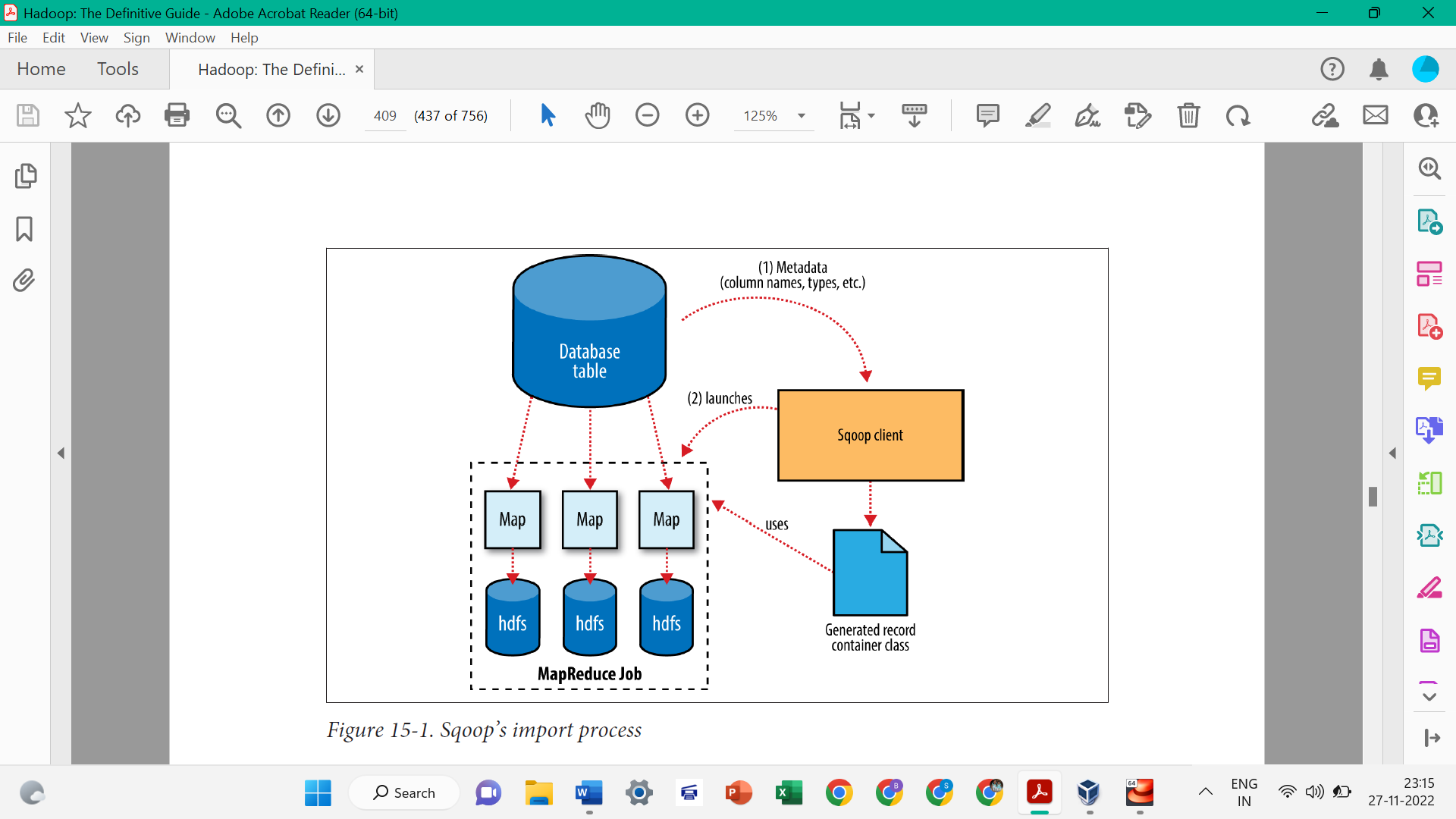
**> --table widgets --class-name Widget**

* The codegen tool simply generates code; it does not perform the full import. We specified that we’d like it to generate a class named Widget; this will be written to *Widget.java*.

**Closer look at the import**

Before the import can start, Sqoop uses JDBC to examine the table it is to import. It

retrieves a list of all the columns and their SQL data types. These SQL types (VARCHAR, INTEGER, etc.) can then be mapped to Java data types (String, Integer, etc.), which will hold the field values in MapReduce applications. Sqoop’s code generator will use this information to create a table-specific class to hold a record extracted from the table.



The Widget class from earlier, for example, contains the following methods that retrieve each column from an extracted record:

**public Integer get\_id();**

**public String get\_widget\_name();**

**public java.math.BigDecimal get\_price();**

**public java.sql.Date get\_design\_date();**

**public Integer get\_version();**

**public String get\_design\_comment();**

More critical to the import system’s operation, though, are the serialization methods

that form the DBWritable interface, which allow the Widget class to interact with JDBC:

**public void readFields(ResultSet \_\_dbResults) throws SQLException;**

**public void write(PreparedStatement** **\_\_dbStmt) throws SQLException;**

* JDBC’s ResultSet interface provides a cursor that retrieves records from a query; the readFields() method here will populate the fields of the Widget object with the columns from one row of the ResultSet’s data. The write() method shown here allows Sqoop to insert new Widget rows into a table, a process called *exporting*.
* The MapReduce job launched by Sqoop uses an InputFormat that can read sections of a table from a database via JDBC. The ***DataDrivenDBInputFormat*** provided with Hadoop partitions a query’s results over several map tasks. Reading a table is typically done with a simple query such as:

SELECT *col1*,*col2*,*col3*,... FROM *tableName*

* But often, better import performance can be gained by dividing this query across multiple nodes. This is done using a *splitting column*. Using metadata about the table, Sqoop will guess a good column to use for splitting the table (typically the primary key for the table, if one exists). The minimum and maximum values for the primary key column are retrieved, and then these are used in conjunction with a target number of tasks to determine the queries that each map task should issue.
* For example, suppose the widgets table had 100,000 entries, with the id column containing values 0 through 99,999. When importing this table, Sqoop would determine that id is the primary key column for the table. When starting the MapReduce job, the DataDrivenDBInputFormat used to perform the import would issue a statement such as SELECT MIN(id), MAX(id) FROM widgets. These values would then be used to interpolate over the entire range of data.
* Assuming we specified that five map tasks should run in parallel (with **-m 5**), this would result in each map task executing queries such as SELECT id, widget\_name, ... FROM widgets WHERE id >= 0 AND id < 20000, SELECT id, widget\_name, ... FROM widgets WHERE id >= 20000 AND id < 40000, and so on.

The choice of splitting column is essential to parallelizing work efficiently. If the id column were not uniformly distributed (perhaps there are no widgets with IDs between 50,000 and 75,000), then some map tasks might have little or no work to perform, whereas others would have a great deal. Users can specify a particular splitting column when running an import job (via the --split-by argument), to tune the job to the data’s actual distribution. If an import job is run as a single (sequential) task with **-m 1**, this split process is not performed.

**What is Apache Flume?**

Apache Flume is an open-source tool for collecting, aggregating, and moving huge amounts of streaming data from the external web servers to the central store, say HDFS, HBase, etc. It is a highly available and reliable service which has tuneable recovery mechanisms.

The main purpose of designing Apache Flume is to move streaming data generated by various applications to Hadoop Distributed FileSystem.

**Why Apache Flume?**

A company has millions of services that are running on multiple servers. Thus, produce lots of logs. In order to gain insights and understand customer behaviour, they need to analyze these logs altogether.

In order to process logs, a company requires an extensible, scalable, and reliable distributed data collection service.

That service must be capable of performing the flow of unstructured data such as logs from source to the system where they will be processed (such as in Hadoop Distributed FileSystem). Flume is an open-source distributed data collection service used for transferring the data from source to destination.

It is a reliable, and highly available service for collecting, aggregating, and transferring huge amounts of logs into HDFS. It has a simple and flexible architecture.

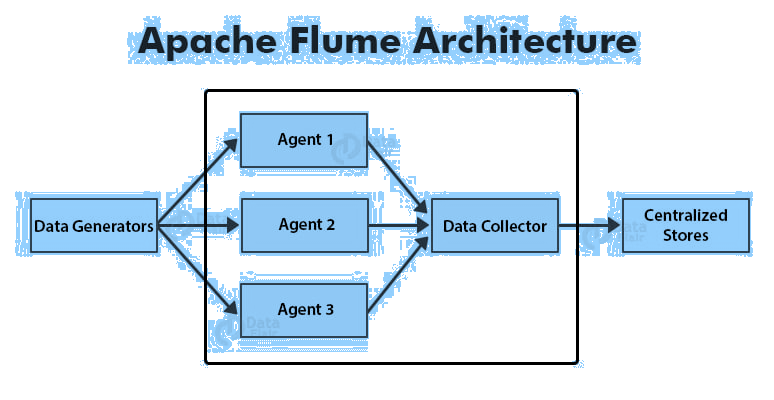
Apache Flume is highly robust and fault-tolerant and has tuneable reliability mechanisms for fail-over and recovery. It allows the collection of data collection in batch as well as in streaming mode.

**Features of Apache Flume**

* Apache Flume is a robust, fault-tolerant, and highly available service.
* It is a distributed system with tuneable reliability mechanisms for fail-over and recovery.
* Apache Flume is horizontally scalable.
* Apache Flume supports complex data flows such as multi-hop flows, fan-in flows, fan-out flows. Contextual routing etc.
* Apache Flume provides support for large sets of sources, channels, and sinks.
* Apache Flume can efficiently ingest log data from various servers into a centralized repository.
* With Flume, we can collect data from different web servers in real-time as well as in batch mode.
* We can import large volumes of data generated by social networking sites and e-commerce sites into Hadoop DFS using Apache Flume.

**Apache Flume Architecture**

Apache Flume has a simple and flexible architecture. The below diagram depicts Flume architecture.



data generators generate huge volumes of data that are collected by individual agents called Flume agents which are running on them. The data generators are Facebook, Twitter, e-commerce sites, or various other external sources.

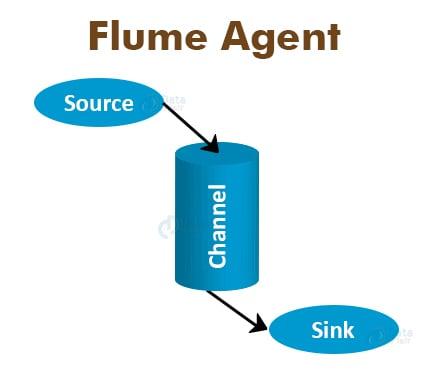
A data collector collects data from the agents, aggregates them, and pushes them into a centralized repository such as HBase or HDFS.

### **Flume Event**

A Flume event is a basic unit of data that needs to be transferred from source to destination.

### **Flume Agent**

Flume agent is an independent JVM process (JVM) in Apache Flume. Agent receives events from clients or other Flume agents and passes it to its next destination which can be sink or other agents.  
Flume Agent contains three main components. They are the source, channel, and sink.



### **Source**

A source receives data from the data generators. It transfers the received data to one or more channels in the form of events.  
Flume provides support for several types of sources.

**Example − Exec source, Thrift source, Avro source, twitter 1% source, etc.**

### **Channel**

A channel receives the data or events from the flume source and buffers them till the sinks consume them. It is a transient store.  
Flume supports different types of channels.

**Example − Memory channel, File system channel, JDBC channel, etc.**

### **Sink**

A sink consumes data from the channel and stores them into the destination. The destination can be a centralized store or other flume agents.

**Example − HDFS sink.**

### **Additional Components of Flume Agent**

There are few more components other than described above that play a significant role in transferring the events.

### **Interceptors**

They alter or inspect flume events transferred between the flume source and channel.

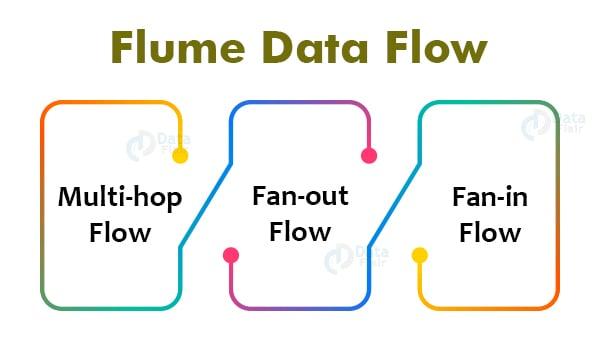
### **Channel Selectors**

They determine which channel is to be chosen for transferring the data when multiple channels exist. Channel selectors are of two types- Default and multiplexing.

### **Sink Processors**

Sink Processors invoke a particular sink from the group of sinks.

### **Apache Flume – Data Flow**



A flume is a tool used for moving log data into HDFS. Apache Flume supports complex data flow. There are three types of data flow in Apache Flume. They are:

### **Multi-hop Flow**

Within Apache Flume, there can be multiple agents. So before reaching the final destination, the flume event may travel through more than one flume agent. This is called a multi-hop flow.

### **2. Fan-out Flow**

The dataflow from one flume source to multiple channels is called fan-out flow. Fan-out flow is of two types − replicating and multiplexing.

### **3. Fan-in Flow**

The fan-in flow is the data flow where data is transferred from many sources to one channel.

## Flume Advantages

1. Apache Flume enables us to store streaming data into any of the centralized repositories (such as HBase, HDFS).  
2. Flume provides steady data flow between producer and consumer during reading2/write operations.  
3. Flume supports the feature of contextual routing.  
4. Apache Flume guarantees reliable message delivery.  
5. Flume is reliable, scalable, extensible, fault-tolerant, manageable, and customizable.

## Flume disadvantages

1. Apache Flume offers weaker ordering guarantees.  
   2. Apache Flume does not guarantee that the messages reaching are 100% unique.  
   3. It has complex topology and reconfiguration is challenging.  
   4. Apache Flume may suffer from scalability and reliability issues.

## Apache Flume Applications

1. Apache Flume is used by e-commerce companies to analyze customer behaviour from a particular region.  
   2. We can use Apache Flume to move huge amounts of data generated by application servers into the Hadoop Distributed File System at a higher speed.  
   3. Apache Flume is used for fraud detections.  
   4. We can use Apache Flume in IoT applications.  
   5. Apache Flume can be used for aggregating machine and sensor-generated data.  
   6. We can use Apache Flume in the alerting or SIEM.

**PROTOCOL BUFFERS**

Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible*(****Forward compatibility****or****upward compatibility****is a design characteristic that allows a system to accept input intended for a later version of itself)* and backward-compatible way (***Backward compatibility****(sometimes known as****backwards compatibility****) is a property of an operating system, product, or technology that allows for interoperability with an older legacy system*). It’s like JSON, except it's smaller and faster, and it generates native language bindings.

Protocol buffers are a combination of the definition language (created in .proto files), the code that the proto compiler generates to interface with data, language-specific runtime libraries, and the serialization format for data that is written to a file (or sent across a network connection).

**What Problems do Protocol Buffers Solve?**

Protocol buffers provide a serialization format for packets of typed, structured data that are up to a few megabytes in size. The format is suitable for both ephemeral network traffic and long-term data storage. Protocol buffers can be extended with new information without invalidating existing data or requiring code to be updated.

Protocol buffers are the most commonly-used data format at Google. They are used extensively in inter-server communications as well as for archival storage of data on disk. Protocol buffer messages and services are described by engineer-authored .proto files. The following shows an example message:

**message Person {  
  optional string name = 1;  
  optional int32 id = 2;  
  optional string email = 3;  
}**

The proto compiler is invoked at build time on .proto files to generate code in various programming languages (covered in [Cross-language Compatibility](https://developers.google.com/protocol-buffers/docs/overview#cross-lang) later in this topic) to manipulate the corresponding protocol buffer. Each generated class contains simple accessors for each field and methods to serialize and parse the whole structure to and from raw bytes.

Because protocol buffers are used extensively across all manner of services at Google and data within them may persist for some time, maintaining backwards compatibility is crucial. Protocol buffers allow for the seamless support of changes, including the addition of new fields and the deletion of existing fields, to any protocol buffer without breaking existing services.

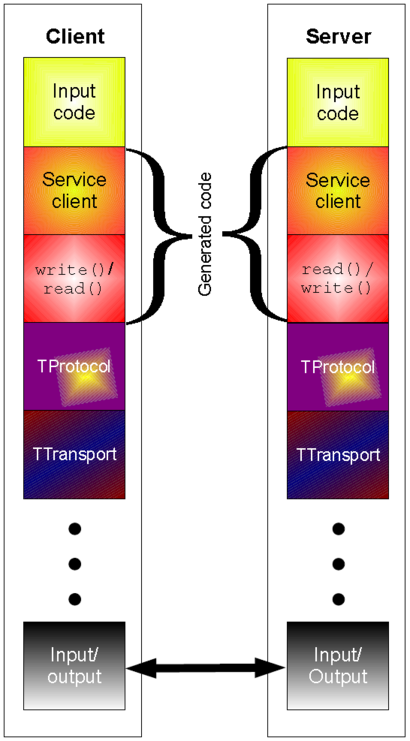
**APACHE THRIFT**

Thrift is an interface definition language and binary communication protocol used for defining and creating services for numerous programming languages. It was developed at Facebook for "scalable cross-language services development" and as of 2020 is an open source project in the Apache Software Foundation.

With a remote procedure call (RPC) framework it combines a software stack with a code generation engine to build cross-platform services which can connect applications written in a variety of languages and frameworks, including ActionScript, C, C++, Go, Haskell, Java etc.,

**Thrift Architecture**

Thrift includes a complete stack for creating clients and servers.



The top part is generated code from the Thrift definition. From this file, the services generate client and processor code. In contrast to built-in types, created data structures are sent as result in generated code. The protocol and transport layer are part of the runtime library. With Thrift, it is possible to define a service and change the protocol and transport without recompiling the code. Besides the client part, Thrift includes server infrastructure to tie protocols and transports together, like blocking, non-blocking, and multi-threaded servers. The underlying I/O part of the stack is implemented differently for different languages.