MS4S21 - Big Data Engineering and Applications

Supplementary Resources

Moizzah Asif, J418, moizzah.asift@southwales.ac.uk

University of South Wales

Contents

Week 1

1.1 Sharding

Sharding is a method for distributing data across multiple machines.

Database systems with large data sets or high throughput applications can challenge the capacity of a single server. For example, high query rates can exhaust the CPU capacity of the server. Working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

There are two methods for addressing system growth: vertical and horizontal scaling.

1.1.1 Vertical Scaling

Vertical Scaling involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space. Limitations in available technology may restrict a single machine from being sufficiently powerful for a given workload. Additionally, Cloud-based providers have hard ceilings based on available hardware configurations. As a result, there is a practical maximum for vertical scaling.

1.1.2 Horizontal Scaling

Horizontal Scaling involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required. While the overall speed or capacity of a single machine may not be high, each machine handles a subset of the overall workload, potentially providing better efficiency than a single high-speed high-capacity server. Expanding the capacity of the deployment only requires adding additional servers as needed, which can be a lower overall cost than high-end hardware for a single machine. The trade off is increased complexity in infrastructure and maintenance for the deployment.

1.1.3 Shard Keys

The shard key consists of a field or fields that exist in every document in the target collection.

You choose the shard key when sharding a collection. The choice of shard key cannot be changed after sharding. A sharded collection can have only one shard key.To shard a non-empty collection, the collection must have an index that starts with the shard key.

Week 6

Please note most of the content this week has been adapted from AWS resources.
You may want to revisit the lecture videos or handouts from week 4 to revisit the basics of AWS cloud and where does EMR sit with in the big data services.

6.1 Amazon EMR

6.1.1 Introduction and overview

Amazon EMR is a managed cluster platform that simplifies running big data frameworks, such as Apache Hadoop on AWS to process and analyze vast amounts of data. The main processing frameworks available for Amazon EMR are Hadoop MapReduce and Spark.
By using such frameworks and related open-source projects, like Apache Hive, you can process data for analytics purposes and business intelligence workloads.
Additionally, you can use Amazon EMR to transform and move large amounts of data into and out of other AWS data stores and databases, such as Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB.

Deja vu - Overview

The central component of Amazon EMR is the cluster. A cluster is a collection of Amazon Elastic Compute Cloud (Amazon EC2) instances.
Each instance in the cluster is called a node. Each node has a role within the cluster, referred to as the node type.
Amazon EMR also installs different software components on each node type, giving each node a role in a distributed application like **Apache Hadoop**. (The text in bold and onward should ring a bell/s and if doesn't, please revise concepts from weeks 1 - 3).

The node types in Amazon EMR as follows:

- **Master node**:

  A node that manages the cluster by running software components to coordinate the distribution of data and tasks among other nodes for processing. The master node tracks the status of tasks and monitors the health of the cluster. Every cluster has a master node, and it's possible to create a single-node cluster with only the master node.

- **Core node**:

  A node with software components that run tasks and store data in the Hadoop Distributed File System (HDFS) on your cluster. Multi-node clusters have at least one core node.

- **Task node**:

  A node with software components that only runs tasks and does not store data in HDFS. Task nodes are optional.

6.1.2 Understanding the cluster life cycle

A successful Amazon EMR cluster follows the lifecycle specified in the steps below and the following flow chart shown in Fig **??**.

1. Amazon EMR first provisions EC2 instances in the cluster for each instance according to your specifications.For all instances, Amazon EMR uses the default AMI for Amazon EMR or a custom Amazon Linux AMI that you specify.

   During this phase, the cluster state is ***STARTING***.

2. Amazon EMR runs bootstrap actions that you specify on each instance. You can use bootstrap actions to install custom applications and perform customisation that you require. For more information on bootstrapping in an EMR cluster you may find this resource useful. However, please bear in mind that this section is created to give you a brief overview before starting any practical work.

   During this phase, the cluster state is ***BOOTSTRAPPING***.

3. Amazon EMR installs the native applications that you specify when you create the cluster, such as Hive, Hadoop, Spark, and so on.

4. After bootstrap actions are successfully completed and native applications are installed, the cluster state is ***RUNNING***.

   At this point, you can connect to cluster instances, and the cluster sequentially runs any steps that you specified when you created the cluster. You can submit additional steps, which run after any previous steps complete.

5. After steps run successfully, the cluster goes into a ***WAITING*** state.

   If a cluster is configured to auto-terminate after the last step is complete, it goes into a ***TERMINATING*** state and then into the ***TERMINATED*** state.

   Whereas if the cluster is configured to wait, you must manually shut it down when you no longer need it. After you manually shut down the cluster, it goes into the ***TERMINATING*** state and then into the ***TERMINATED*** state.

But what about failure??

A failure during the cluster lifecycle causes Amazon EMR to terminate the cluster and all of its instances unless you enable termination protection. If a cluster terminates because of a failure, any data stored on the cluster is deleted, and the cluster state is set to ***TERMINATED_WITH_ERRORS***. If you enabled termination protection, you can retrieve data from your cluster, and then remove termination protection and terminate the cluster

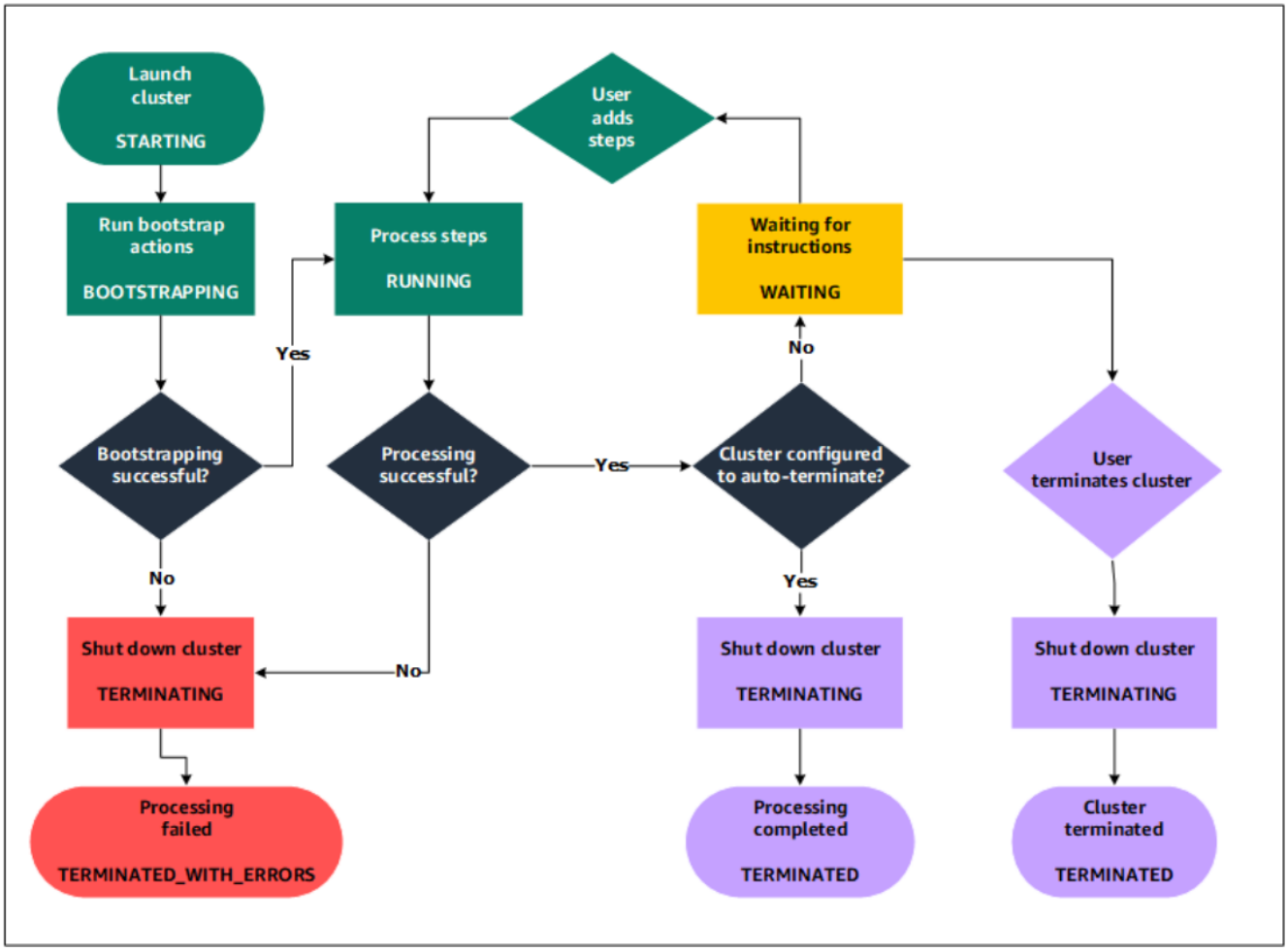

Figure 6.1: EMR lifecycle flow chart

Week 7 & 8

7.1 Apache Hive

This chapter is adapted from chapters in recommended readings for MS4S21 and AWS tutorials and blogs.

7.1.1 Why and how Hive became popular

Apache Hive is a framework for data warehousing on top of Hadoop. Hive grew from a need to manage and learn from the huge volumes of data that Facebook was producing every day from its burgeoning social network. After trying a few different systems, the team chose Hadoop for storage and processing, since it was cost effective and met the scalability requirements.

Hive was created to make it possible for analysts with strong SQL skills (but meagerJava programming skills) to run queries on the huge volumes of data that Facebook stored in HDFS. Today, Hive is a successful Apache project used by many organisations as a general-purpose, scalable data processing platform.

In normal use, Hive runs on your workstation and converts your SQL query into a series of jobs for execution on a Hadoop cluster. Hive organizes data into tables, which provide a means for attaching structure to data stored in HDFS. Metadata, such as table schemas, is stored in a database called the metastore.

Hive was originally written to use MapReduce as its execution engine, and that is still the default. It is now also possible to run Hive using Apache Tez and Spark as its execution engine, too.

7.1.2 Overview of Hive

Hive is a standard for SQL queries over petabytes of data in Hadoop. It provides SQL-like access to data in HDFS, enabling Hadoop to be used as a data warehouse. The Hive Query Language (HQL) has similar semantics and functions as standard SQL in the relational database. Hive's query language can run on different computing engines, such as MapReduce, Tez, and Spark.

Hive's metadata structure provides a high-level, table-like structure on top of HDFS. It supports three main data structures, tables, partitions, and buckets. The tables correspond to HDFS directories and can be divided into partitions, where data files can be divided into buckets. Hive's metadata structure is usually the Schema of the Schema-on-Read concept on Hadoop, which means you do not have to define the schema in Hive before you store data in HDFS. Applying Hive metadata after storing data brings more flexibility and efficiency to your data work. The popularity of Hive's metadata makes it the de facto way to describe big data and is used by many tools in the big data ecosystem.

7.1.3 Hive configuration in a Hadoop cluster

Hive is configured using an XML configuration file like Hadoop's. The file is called hive-site.xml and is located in Hive's conf directory. This file is where you can set properties that you want to set every time you run Hive. The same directory contains hive-default.xml, which documents the properties that Hive exposes and their default values.

The execution engine is controlled by the hive.execution.engine property, which defaults to mr (for MapReduce).

7.1.4 Using Hive in the cloud

Right now, all major cloud service providers, such as Amazon, Microsoft, and Google, offer matured Hadoop and Hive as services in the cloud.

Using the cloud version of Hive is very convenient. It requires almost no installation and setup. Amazon EMR (http://aws.amazon.com/elasticmapreduce/) is the earliest Hadoop service in the cloud. However, it is not a pure open source version since it is customised to run only on Amazon Web Services (AWS).

Hadoop enterprise service and distribution providers, such as Cloudera and Hortonworks, also provide tools to easily deploy their own distributions on different public or private clouds. Cloudera Director (http://www.cloudera.com/content/cloudera/en/ products-and-services/director.html) and Cloudbreak (https://hortonworks.com/ open-source/cloudbreak/), open up Hadoop deployments in the cloud through a simple, self-service interface, and are fully supported on AWS, Windows Azure, Google Cloud Platform, and OpenStack.
As discussed in live sessions, we will be working with Hive on AWS EMR.

7.1.5 HiveQL

Hive's SQL dialect, called HiveQL, is a mixture of SQL-92, MySQL, and Oracle's SQLdialect. The level of SQL-92 support has improved over time, and will likely continueto get better. HiveQL also provides features from later SQL standards, such as windowfunctions (also known as analytic functions) from SQL:2003. This chapter does not provide a complete reference to HiveQL; for that, see the Hive documentation.

7.1.6 Data Types

Hive supports both primitive and complex data types. Primitives include numeric,Boolean, string, and timestamp types. The details of primitive types are provided in Table 7.1. The complex data types include arrays, maps, and structs. String and Int are the most useful primitive types, which are supported by most HQL functions.

Primitive Type	Description	Example
TINYINT	It has 1 byte, from -128 to 127. The postfix is Y. It is used as a small range of numbers.	10Y
SMALLINT	It has 2 bytes, from -32,768 to 32,767. The postfix is S. It is used as a regular descriptive number.	10S
INT	It has 4 bytes, from -2,147,483,648 to 2,147,483,647.	10
BIGINT	t has 8 bytes, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The postfix is L.	100L
FLOAT	This is a 4 byte single-precision floating-point number, from 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative). Scientific notation is not yet supported. It stores very close approximations of numeric values.	1.2345679
DOUBLE	This is an 8 byte double-precision floating-point number, from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative). Scientific notation is not yet supported. It stores very close approximations of numeric values.	1.2345678901234567
BINARY	This is a TRUE or FALSE value.	TRUE
STRING	This includes characters expressed with either single quotes (') or double quotes ("). Hive uses C-style escaping within the strings. The max size is around 2 G.	'Books' or "Books"
CHAR	This is available starting with Hive 0.13.0. Most UDF will work for this type after Hive 0.14.0. The maximum length is fixed at 255.	'K' OR "UK"
VARCHAR	This is available starting with Hive 0.12.0. Most UDF will work for this type after Hive 0.14.0. The maximum length is fixed at 65,355. If a string value being converted/assigned to a varchar value exceeds the length specified, the string is silently truncated	'Books' or "Books"
DATE	This describes a specific year, month, and day in the format of YYYY-MM-DD. It is available starting with Hive 0.12.0. The range of dates is from 0000-01-01 to 9999-12-31.	2021-05-14
TIMESTAMP	This describes a specific year, month, day, hour, minute, second, and millisecond in the format of YYYY-MM-DD HH:MM:SS[.fff...]. It is available starting with Hive 0.8.0.	2021-05-14 21:00:00.00

Table 7.1: Hive Primitive datatypes

Complex Type	Description	Example
ARRAY	This is a list of items of the same type, such as [val1, val2, and so on]. You can access the value using array_name[index], for example, fruit[0] = "apple". Index starts from 0.	["apple","orange","mango"]
MAP	This is a set of key-value pairs, such as key1, val1, key2, val2, and so on. You can access the value using map_name[key] for example, fruit[1] = "apple".	{1:"apple", 2: "orange"}
STRUCT	This is a user-defined structure of any type of field, such as val1, val2, val3, and so on. By default, STRUCT field names will be col1, col2, and so on. You can access the value using structs_name.column_name, for example, fruit.col1 = 1.	{1, "apple"}
NAMED STRUCT	This is a user-defined structure of any number of typed fields, such as {name1, val1, name2, val2, and so on}. You can access the value using structs_name.column_name, for example, fruit.apple = "gala".	{"apple" "gala" "weight kg":1}
UNION	This is a structure that has exactly any one of the specified data types. It is available starting with Hiv 0.7.0. It is not commonly used.	{2:["apple", "orange"]}

Table 7.2: Hive Complex datatypes

7.1.7 Operators

There are four types of operators supported by Hive:

- Relational Operators
- Arithmetic Operators
- Logical Operators
- Complex Operators

Relational Operators

Operator	Operand	Description
A=B	all primitive types	TRUE if expression A is equivalent to expression B otherwise FALSE.
A != B	all primitive types	TRUE if expression A is not equivalent to expression B otherwise FALSE.
A < B	all primitive types	TRUE if expression A is less than expression B otherwise FALSE.
A <= B	all primitive types	TRUE if expression A is less than or equal to expression B otherwise FALSE.
A > B	all primitive types	TRUE if expression A is greater than expression B otherwise FALSE.
A >= B	all primitive types	TRUE if expression A is greater than expression B otherwise FALSE.
A IS NULL	all types	TRUE if expression A evaluates to NULL otherwise FALSE.
A IS NOT NULL	all types	FALSE if expression A evaluates to NULL otherwise TRUE.
A LIKE B	strings	TRUE if string pattern A matches to B otherwise FALSE.
A RLIKE B	strings	NULL if A or B is NULL, TRUE if any substring of A matches the Java regular expression B, otherwise FALSE.
A REGEXP B	strings	Same as RLIKE.

Table 7.3: Hive Relational Operators

Arithmetic Operators

Operator	Operand	Description
A + B	all number types	Gives the result of adding A and B.
A - B	all number types	Gives the result of subtracting B from A
A*B	all number types	Gives the result of multiplying A and B.
A/B	all number types	Gives the result of dividing B from A.
A%	B all number types	Gives the reminder resulting from dividing A by B
A & B	all number types	Gives the result of bitwise AND of A and B.
A | B	all number types	Gives the result of bitwise OR of A and B.
A ∧ B	all number types	Gives the result of bitwise XOR of A and B.
˜A	all number types	Gives the result of bitwise NOT of A.

Table 7.4: Hive Arithmetic Operators

Logical operators

A AND B	boolean	TRUE if both A and B are TRUE, otherwise FALSE.
A&& B	boolean	Same as A AND B.
A OR B	boolean	TRUE if either A or B or both are TRUE, otherwise FALSE.
A || B	boolean	Same as A OR B.
NOT A	boolean	TRUE if A is FALSE, otherwise FALSE.
!A	boolean	Same as NOT A.

Table 7.5: Hive Logical Operators

Complex operators

A[n]	A is an Array and n is an int	It returns the nth element in the array A. The first element has index 0.
M[key]	M is a Map $\langle K, V \rangle$ and key has type 'K'	It returns the value corresponding to the key in the map.
S.x	S is a struct	It returns the 'x' field of 'S'.

Table 7.6: Hive Logical Operators