COS221 - L35 - Big Data

Linda Marshall

University of Pretoria

26 May 2023

Introduction

- Big data is prevalent in applications such as social media, Web links, user profiles, marketing and sales, posts and tweets, road maps and spatial data, and e-mail.
- Big data storage systems were developed to manage large amounts of data in organisations such as Google, Amazon, Facebook, and Twitter.
- Many applications need systems other than traditional relational SQL systems to augment their data management needs.
- The McKinsey report defines the term *big data* as datasets whose size exceeds the typical reach of a DBMS to capture, store, manage, and analyse that data.
- The term **NOSQL** is generally interpreted as Not Only SQL - rather than NO to SQL.

What is Big Data?

Big data is characterised by the three V's:

- **Volume:** refers to the size of data managed by the system.
- **Velocity:** the speed at which data is created, accumulated, ingested, and processed.
- **Variety:** Big data includes structured, semistructured, and unstructured data in different proportions based on context.

A more recent addition to the three V's:

- **Veracity:** has two built-in features: the credibility of the source, and the suitability of data for its target audience.

Emergence of NOSQL Systems

Why a structured relational SQL system may not be appropriate:

- SQL systems offer too many services (powerful query language, concurrency control, etc.), which these applications may not need.
- a structured data model such the traditional relational model may be too restrictive.

Some examples:

- Google developed a proprietary NOSQL system known as **BigTable**, which is used in many of Google's applications that require vast amounts of data storage, such as Gmail, Google Maps, and Web site indexing.
- Amazon developed a NOSQL system called **DynamoDB** that is available through Amazon's cloud services.
- Facebook developed a NOSQL system called **Cassandra**, which is now open source and known as Apache Cassandra.

Characteristics of NOSQL Systems

- NOSQL characteristics related to distributed databases and distributed systems:
  - Scalability: mostly **horizontal scalability**
  - Availability, Replication and Eventual Consistency: impact of replication on read and write performance
  - Replication Models: master-slave and master-master replication
  - Sharding of Files: serves to distribute the load of accessing the file records to multiple nodes (horizontal partitioning)
  - High-Performance Data Access: to find individual records or objects (data items) from among the millions of data records or objects in a file using hashing or range partitioning on object keys.

Characteristics of NOSQL Systems

- NOSQL characteristics related to data models and query languages:
  - Not Requiring a Schema: by allowing semi-structured, self-describing data
  - Less Powerful Query Languages: typically provide a set of functions and operations as a programming API
  - Versioning: Some NOSQL systems provide storage of multiple versions of the data items, with the timestamps of when the data version was created.

Categories of NOSQL Systems

- **Document-based** NOSQL systems: These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation).
- NOSQL **key-value stores**: These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.
- **Column-based** or wide column NOSQL systems: These systems partition a table by column into column families (a form of vertical partitioning), where each column family is stored in its own files.
- **Graph-based** NOSQL systems: Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions.
- **Hybrid** NOSQL systems: These systems have characteristics from two or more of the above four categories.
- Object databases
- XML databases

The CAP Theorem

The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **consistency** (among replicated copies), **availability** (of the system for read and write operations) and **partition tolerance** (in the face of the nodes in the system being partitioned by a network fault).

The CAP Theorem

It is not possible to guarantee all three of the desirable properties - consistency, availability, and partition tolerance - at the same time in a distributed system with data replication.

Weaker consistency levels are often used in NOSQL systems instead of guaranteeing serializability. In particular, a form of consistency known as **eventual consistency** is often adopted in NOSQL systems.

MongoDB Data Model

- MongoDB documents are stored in BSON (Binary JSON) format, which is a variation of JSON with some additional data types and is more efficient for storage than JSON.
- Individual **documents** are stored in a **collection.**
- For example, the following command can be used to create a collection called project to hold PROJECT objects from the COMPANY database
  - *db.createCollection("project", { capped : true, size : 1310720, max : 500 })*
- Each document in a collection has a unique **ObjectId** field, called **_id**, which is automatically indexed in the collection unless the user explicitly requests no index for the _id field.
- Documents can be inserted into their collection using the following example command:
  - *db.project.insert({_id: "P1", Pname: "ProductX", Plocation: "Bellaire"})*

MongoDB CRUD Operations

- Documents can be created and inserted into their collections using the **insert** operation, whose format is:
  - *db.* $<$ *collection_name* $>$ *.insert*($<$ *document*(*s*) $>$)
- The *delete* operation is called **remove**, and the format is:
  - *db.* $<$ *collection_name* $>$ *.remove*($<$ *condition* $>$)
- There is also an **update** operation, which has a condition to select certain documents, and a *$set* clause to specify the update.
- For *read* queries, the main command is called **find**, and the format is:
  - *db.* $<$ *collection_name* $>$ *.find*($<$ *condition* $>$)

MongoDB Distributed Systems Characteristics

- **Replication in MongoDB.** The concept of replica set is used in MongoDB to create multiple copies of the same data set on different nodes in the distributed system, and it uses a variation of the master-slave approach for replication.
- **Sharding in MongoDB** divides the documents into disjoint partitions known as **shards.** This allows the system to add more nodes as needed by a process known as **horizontal scaling** of the distributed system, and to store the shards of the collection on different nodes to achieve load balancing.

DynamoDB Overview

The DynamoDB system is an Amazon product and is available as part of Amazon's AWS/SDK platforms.

DynamoDB data model

A **table** in DynamoDB does not have a schema; it holds a collection of self-describing items. Each **item** will consist of a number of (attribute, value) pairs, and attribute values can be single-valued or multivalued.
When a table is created, it is required to specify a **table name** and a **primary key**; the primary key will be used to rapidly locate the items in the table. Thus, the primary key is the key and the item is the value for the DynamoDB key-value store.

DynamoDB Distributed Characteristics

Because DynamoDB is proprietary, we will discuss the mechanisms used for replication, sharding, and other distributed system concepts in an open source key-value system called Voldemort.

Voldemort Key-Value Distributed Data Store

Voldemort has been used by LinkedIn for data storage. Some of the features of Voldemort are as follows:

- **Simple basic operations:** get, put, and delete.
- **High-level formatted data values.** The values v in the (k, v) items can be specified in JSON, and the system will convert between JSON and the internal storage format.
- **Consistent hashing for distributing (key, value) pairs.** A hash function h(k) is applied to the key k of each (k, v) pair, and h(k) determines where the item will be stored.
- **Consistency and versioning.** Concurrent writes are allowed, but each write is associated with a vector clock value. When a read occurs, it is possible that different versions of the same value (associated with the same key) are read from different nodes.

Examples of Other Key-Value Stores

Major Examples...

- **Oracle key-value store.** Called the **Oracle NoSQL Database.**
- **Redis key-value cache and store. Redis** differs from the other systems discussed here because it caches its data in main memory to further improve performance.
- **Apache Cassandra. Cassandra** is a NOSQL system that is not easily categorised into one category; it is sometimes listed in the column-based NOSQL category or in the key-value category. It offers features from several NOSQL categories and is used by Facebook as well as many other customers.

Hbase Data Model and Versioning

Hbase data model

The data model in Hbase organises data using the concepts of *namespaces, tables, column families, column qualifiers, columns, rows, and data cells.* A cell holds a basic data item in Hbase. A namespace is a collection of tables.

Versions and Timestamps

Hbase can keep several versions of a data item, along with the timestamp associated with each version.

Hbase CRUD Operations

The *create* operation creates a new table and specifies one or more column families associated with that table. The *put* operation is used for inserting new data or new versions of existing data items. The *get* operation is for retrieving the data associated with a single row in a table, and the *scan* operation retrieves all the rows.

Hbase Storage and Distributed System Concepts

Each Hbase table is divided into a number of **regions**, where each region will hold a *range* of the row keys in the table; this is why the row keys must be lexicographically ordered.
Each region will have a number of **stores**, where each column family is assigned to one store within the region. Regions are assigned to **region servers** (storage nodes) for storage.
A **master server** (master node) is responsible for monitoring the region servers and for splitting a table into regions and assigning regions to region servers.
Hbase uses the **Apache Zookeeper** open source system for services related to managing the naming, distribution, and synchronisation of the Hbase data on the distributed Hbase server nodes, as well as for coordination and replication services.

Neo4j Data Model

- The data model in Neo4j organises data using the concepts of **nodes** and **relationships.**
- Both nodes and relationships can have **properties**, which store the data items associated with nodes and relationships.
- Nodes can have **labels**; the nodes that have the *same label* are grouped into a collection that identifies a subset of the nodes in the database graph for querying purposes.
- Relationships are directed; each relationship has a *start node* and *end node* as well as a **relationship type**, which serves a similar role to a node label by identifying similar relationships that have the same relationship type.
- Properties can be specified via a **map pattern**, which is made of one or more 'name : value' pairs enclosed in curly brackets; for example {Lname : 'Smith', Fname : 'John', Minit : 'B'}.

MapReduce and Hadoop - Historical Background

- Hadoop has originated from the quest for an open source search engine.
- The first attempt was made by the then Internet archive director Doug Cutting and University of Washington graduate student Mike Carafella.
- They developed a system called Nutch that could crawl and index hundreds of millions of Web pages.
- Yahoo spun off the storage engine and the processing parts of Nutch as **Hadoop** (named after the stuffed elephant toy of Cutting's son).
- Hadoop was motivated by the Google File System and the MapReduce programming paradigm developed earlier by Google.
- A joint effort by Google, IBM, and NSF used a 2,000-node Hadoop cluster at a Seattle data center and helped further universities' research on Hadoop.

MapReduce

The MapReduce Programming Model

The map and reduce functions have the following general form:
map[K1,V1] which is (key, value) : List[K2,V2] and
reduce(K2, List[V2]) : List[K3,V3]
Map is a generic function that takes a key of type **K1** and a value of type **V1** and returns a list of key-value pairs of type **K2** and **V2**.
Reduce is a generic function that takes a key of type **K2** and a list of values of type **V2** and returns pairs of type **(K3,V3)**. In general, the types K1, K2, K3, etc., are different, with the only requirement that the output types from the Map function must match the input type of the Reduce function.

MapReduce

An Example

Assume that we have a document and we want to make a list of words in it with their corresponding frequencies:

Map (String key, String value):

for each word w in value Emitintermediate (w, “1”);

Here key is the document name, and value is the text content of the document. Then the above lists of (word, 1) pairs are added up to output total counts of all words found in the document as follows:

Reduce (String key, Iterator values) : // here the key is a word and values are lists of its counts //

Int result = 0;

For each v in values :

result += Parseint (v);

Emit (key, Asstring (result));

Hadoop Distributed File System (HDFS) Preliminaries

HDFS was designed with the following assumptions and goals:

- **Hardware failure:** automatic detection and recovery from failures.
- **Batch processing:** HDFS has been primarily designed for batch rather than interactive use.
- **Large datasets:** HDFS was designed to support huge files in the hundreds of gigabytes to terabytes range.
- **Simple coherency model:** HDFS applications need a one writer and many reader access models for files. File content cannot be updated, but only appended. This model alleviates coherency issues among copies of data.

Architecture of HDFS

HDFS has a master-slave architecture.

- **NameNode.** The NameNode maintains an image of the file system comprising i-nodes (index nodes) and corresponding block locations.
- **Secondary NameNodes.** These are additional NameNodes that can be created to perform either the checkpointing role or a backup role.
- **DataNodes:** Blocks are stored on a DataNode in the node's native file system. The NameNode directs clients to the DataNodes that contain a copy of the block they want to read.

DataNodes and NameNodes do not communicate directly but via a so-called **heartbeat mechanism,** which refers to a periodic reporting of the state by the DataNode to the NameNode; the report is called a **Block Report.**

File I/O Operations and Replica Management in HDFS

HDFS provides a single-writer, multiple-reader model. Files cannot be updated, but only appended. A file consists of blocks.

- **Block Placement.** The ultimate goal of block placement is to minimize the write cost while maximizing data availability and reliability as well as available bandwidth for reading.
- **Replica Management.** Based on the block reports from the DataNodes, the NameNode tracks the number of replicas and the location of each block. A replication priority queue contains blocks that need to be replicated.

The Hadoop Ecosystem

Hadoop ecosystem has a set of related projects that provide additional functionality on top of HDFS and MR.

Important Projects

- **Pig and Hive:** Pig provides a dataflow language. Hive provides an SQL interface on top of MapReduce.
- **Oozie:** This is a service for scheduling and running workflows of Jobs; individual steps can be MR jobs, Hive queries, Pig scripts, and so on.
- **Sqoop:** This is a library and a runtime environment for efficiently moving data between relational databases and HDFS.
- **HBase:** This is a column-oriented key-value store that uses HDFS as its underlying store.

MapReduce Runtime

- **JobTracker.** The master process is called the JobTracker. It is responsible for managing the life cycle of Jobs and scheduling Tasks on the cluster.
- **TaskTracker.** The slave process is called a TaskTracker. There is one running on all Worker nodes of the cluster.

MapReduce Runtime

Overall flow of a MapReduce Job

- **Job submission** A client submits a Job to the JobTracker.
- **Job initialization** The JobTracker accepts the Job and places it on a Job Queue.
- **Task assignment** The JobTracker's scheduler assigns Task to the TaskTracker from one of the running Jobs.
- **Task execution** Once a task has been scheduled on a slot, the TaskTracker manages the execution of the task: making all Task artifacts available to the Task process, launching the Task JVM, monitoring the process and coordinating with the JobTracker to perform management operations like cleanup on Task exit, and killing Tasks on failure conditions.
- **Job completion** Once the last Task in a Job is completed, the JobTracker runs the Job cleanup task.

MapReduce Runtime

Fault Tolerance in MapReduce

- **Task failure** The TaskTracker notifies the JobTracker that the Task has failed. When the JobTracker is notified of the failure, it will reschedule execution of the task.
- **TaskTracker failure** Once the JobTracker marks a TaskTracker as failed, any map tasks completed by the TaskTracker are put back on the queue to be rescheduled.
- **JobTracker failure** In Hadoop v1, JobTracker failure is not a recoverable failure. The JobTracker is a Single Point of Failure. The JobTracker has to be manually restarted. On restart all the running jobs have to be resubmitted. This is one of the drawbacks of Hadoop v1 that have been addressed by the next generation of Hadoop MapReduce called YARN.

MapReduce Runtime

The Shuffle Procedure

- **Map Phase:** During the Map phase, several iterations of partitioning, sorting, and combining may happen. The end result is a single local file per reducer that is sorted on the Key.
- **Copy phase:** The Reducers pull their files from all the Mappers as they become available.
- **Reduce phase:** The Reducer reads all its files from the Mappers. All files are merged before streaming them to the Reduce function.

Job Scheduling

- **Fair Scheduler:** provides fast response time to small jobs in a Hadoop shared cluster.
- **Capacity Scheduler:** The Capacity Scheduler is geared to meet the needs of large Enterprise customers.

Example: Achieving Joins in MapReduce

Let us consider the problem of joining two relations R(A, B) with S(B, C) with the join condition R.A = S.B. Assume both tables reside on HDFS. Here we list the many strategies that have been devised to do equi-joins in the MapReduce environment:

- **Sort-Merge Join.** The broadest strategy for performing a join is to utilize the Shuffle to partition and sort the data and have the reducers merge and generate the output.
- **Map-Side Hash Join.** For the case when one of R or S is a small table that can be loaded in the memory of each task, we can have the Map phase operate only on the large table splits.
- **Partition Join.** Assume that both R and S are stored in such a way that they are partitioned on the join keys. Then all rows in each Split belong to a certain identifiable range of the domain of the join field, which is B in our example.

Example: Achieving Joins in MapReduce

- **Bucket Joins.** This is a combination of Map-Side and Partition Joins. In this case only one relation, say the right side relation, is Partitioned. We can then run Mappers on the left side relation and perform a Map Join against each Partition from the right side.
- **N-Way Map-Side Joins.** A join on R(A, B, C, D), S(B, E), and T(C, F) can be achieved in one MR job provided the rows for a key for all small tables can be buffered in memory.
- **Simple N-Way Joins.** A join on R(A, B), S(B, C), and T(B, D) can be achieved in one MR job provided the rows for a key for all small tables can be buffered in memory.

Apache Pig

Whereas it is possible to express very complex analysis in MR, the user must express programs as a one-input, two-stage (map and reduce) process. There is no standard way to do common data transformation operations like Projections, Filtering, Grouping, and Joining. The developers of Pig invented the language **Pig Latin** to fill in the "sweet spot" between SQL and MR.
Pig accommodates user-defined functions (UDFs) extensively. It also supports a nested data model with the following four types:

- **Atoms:** Simple atomic values such as a number or a string
- **Tuples:** A sequence of fields, each of which can be of any permissible type
- **Bag:** A collection of tuples with possible duplicates
- **Map:** A collection of data items where each item has a key that allows direct access to it.

Apache Hive

Hive went beyond Pig Latin in that it provided not only a high-level language interface to Hadoop, but a layer that makes Hadoop look like a DBMS with DDL, metadata repository, JDBC/ODBC access, and an SQL compiler.
The Hive query language HiveQL includes a subset of SQL that includes all types of joins, Group By operations, as well as useful functions related to primitive and complex data types.

Advantages of the Hadoop/MapReduce Technology

- The disk seek rate is a limiting factor when we deal with petabyte-level workloads. The MapReduce model of scanning datasets in parallel alleviates this situation.
- The MapReduce model allows handling of semistructured data and key-value datasets more easily compared to traditional RDBMSs, which require a predefined schema.
- The MapReduce model has linear scalability in that resources can be added to improve job latency and throughput in a linear fashion.

Rationale behind YARN

- **Multitenancy:** Multitenancy refers to accommodating multiple tenants/users concurrently so that they can share resources.
- **JobTracker Scalability.** As the cluster sizes increased beyond 4,000 nodes, issues with memory management and locking made it difficult to enhance JobTracker to handle the workload.
- **JobTracker: Single Point of Failure.** The recovery model of Hadoop v1 was very weak. A failure of JobTracker would bring down the entire cluster.
- **Misuse of the MapReduce Programming Model.** MR runtime was not a great fit for iterative processing; this was particularly true for machine learning algorithms in analytical workloads.
- **Resource Model Issues.** In Hadoop v1, a node is divided into a fixed number of Map and Reduce slots. This led to cluster underutilization because idle slots could not be used. Jobs other than MR could not run easily on the nodes

YARN Architecture

- **Resource Manager (RM).** The Resource Manager is only concerned with allocating resources to Applications, and not with optimizing the processing within Applications.
- **ApplicationMaster (AM).** The ApplicationMaster is responsible for coordinating the execution of an Application on the cluster.
- **NodeManager.** A NodeManager runs on every worker node of the cluster. It manages Containers and provides pluggable services for Containers.
- **Fault tolerance and availability.** The RM remains the single point of failure in YARN. On restart, the RM can recover its state from a persistent store.

Other Frameworks on YARN

- **Apache Tez.** Tez is an extensible framework developed at Hortonworks for building high-performance applications in YARN; these applications will handle large datasets up to petabytes.
- **Apache Giraph.** Apache Giraph is the open source implementation of Google's Pregel system, which was a large-scale graph processing system used to calculate Page-Rank. Giraph is currently used at Facebook to analyze the social network users' graph, which has users as nodes and their connections as edges; the current number of users is approximately 1.3 billion.
- **Hoya: HBase on YARN.** The Hortonworks Hoya (HBase on YARN) project provides for elastic HBase clusters running on YARN with the goal of more flexibility and improved utilization of the cluster.