**CS286 Solving Big Data Problems – Exam #1 Study Guide**

**By: Zayd Hammoudeh**

**Lecture #01 – Introduction to Big Data**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Categories** | | **Data** – Raw values  **Information** – Set of data with meaning  **Knowledge** –Interpretation of the data with meaning.  **Wisdom** –Appropriate application of knowledge. |  |
| **Quantitative**   * Observable and **measureable** * Structured and **objective** * Numerical   **Example:** Income, Height | **Qualitative**   * Observable but not **measureable** * Unstructured and subjective * **Descriptive**   **Example:** Favorite Color |

**Storage Terminology**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Directly Attached**  **Storage (DAS)**   * Storage attached directly to the processing node. * Lowest capacity * Minimal data sharing * **Highest Speed.** | **Network Attached Storage (NAS)**   * Storage accessible via a network connection. * **Capable of using NFS** | **Relational Database  Management System (RDBMS)**   * Traditional database providers. * **Examples:** Oracle, MySQL, IBM DB2 | **Storage Area**  **Network (SAN)**   * Storage accessible via a network connection. * Uses different protocols than NAS. | **Network File System (NFS)**  Allows a computer to view and store data on remote disk as if that disk was directly attached to the local computer.  **Access Transparency** – Access data the same way whether it is remote or local. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Analysis Categories** | | **Four Steps in Traditional Data Mining**   1. Problem Definition 2. Data gathering and preparation 3. Model building and evaluation 4. Knowledge Deployment   Process is **cyclical** and **may repeat multiple times**. |  |
| **Descriptive**   * **Backward** looking. * Hindsight * Explain a previous phenomenon. * **Analysis** | **Predictive**   * **Forward** looking * Foresight * Investigate future trends. * **Mining** |

**Big Data**

|  |  |  |  |
| --- | --- | --- | --- |
| **Big Data** – Data whose scale, diversity, and complexity require new architecture, techniques, algorithms, and analytics to manage it and to extract value and hidden knowledge from it. | **3 V’s of Big Data** | | |
| **Volume** – The amount of data is too large for traditional database software tools to cope with.  **Example:** Image server | **Velocity** – The data is being produced at a rate that is beyond the performance limits of traditional systems.  **Example:** Social media site | **Variety** – Data lacks the structure to make it suitable for storage and analysis in traditional databases and data warehouses.  **Example:** Data organization variety. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data Organization** | | | **Scaling to Process Big Data** | | |
| **Structured** – Every piece of data and its format is known. Fits in a database.  **Example:** RDBMS | **Semi-structured** – For some fields, data may not exist and some fields can have different formats. Not in a typical database but has structure.  **Example:** XML, CSV, JSON | **Unstructured** – Does not fit into a database well. Most data is in this category.  **Examples:** Text document, multimedia content. | **Scale Up**  **Limitations:**   * Large capital and operating expense. * Lower availability and scalability.   **Example:** Monolithic Database | **Scale Out**  **Limitations:**   * Synchronization overhead * Programming Complexity * Specialized hardware.   **Example:** Grid Cluster | **Sampling**  **Limitation:**   * Lower accuracy and precision.   **Example:** Any approach |

|  |  |  |  |
| --- | --- | --- | --- |
| **Exploiting Locality of Reference** – In Big Data, accessing the data can be very time consuming. **Solution:** Keep the data and program close together.  **Distribute Data and Computation** – Map the data to multiple nodes and the program with it to decrease execution time. | **Three Laws of Big Data** | | |
| **Moore’s Law** – Every two years, the number of transistors per chip doubles.  **Kryder’s Law** – Every two years, storage capacity doubles. (**Storage version of Moore’s Law**) | **Amdahl’s Law** – The extent to which a program’s execution can be sped up is dependent on its level of parallelism. | **Murphy’s Law** – What can go wrong will go wrong.  **Big data must be resistant to failures.** |

**Hadoop**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Summary of the Hadoop Strategy** | | | **Core of Hadoop**   1. **Hadoop File System (HDFS)** 2. **Map Reduce** | **Name Node**  Key component in HDFS that **stores the location of distributed data in the file system**. | **Job Tracker**  Manages **computation tasks in the Hadoop system**. |
| **Distribute Data**  Processing nodes share no data. | **Distribute Computation**  Achieve **parallelism without synchronization.** | **Tolerate Failures**  Eliminate **single points of failure**. |

**Lecture #02 – Introduction to HDFS and MapR-FS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **File System**  Like a database. **A system to store data so that the data can be accessed later.**  **Typical Structure:** A rooted tree. | **Storage in a File System**  **Data** – Actual file in the FS.  **Metadata** – Information about the data/file.  **Example:** Size, location | **Block Structure in an ext2 File System** | | | | |
| **Hadoop Block Size:** 64MB | **Inode** – Data structure used to represent a file system object. This includes the location of the disk block location. | **Direct Block** – File block location **pointed to directly by the inode**. | **Indirect Block** – Block **pointed to by the inode through exactly one intermediary block**. | **Double Indirect Block** – Block **pointed to by the inode through exactly two intermediary blocks**. |

|  |  |
| --- | --- |
| **Virtual File System**   * **Transition layer** between a generic file system and a real file system. * Virtualizes different file system types into a single common interface. * **Enables standard POSIX** file access. * **HDFS is not compatible with a virtual file system while MapR-FS is.** | **Distributed File System**   * **Centrally stores metadata** (e.g. **name node**) **and distributes actual data** (e.g. **data node**) * Overcomes **space, performance, and availability limitations of a single machine.** * **Location Transparency** – Abstracts data locality from client access. |

|  |  |  |
| --- | --- | --- |
| **Hadoop Data Write** | **Hadoop Data Read** | **Hadoop Write Pipeline** – Before a write can be acknowledged to the client, it must be acknowledged by the name node.   * Each replicate write is sequential **through a pipeline where one data node writes to the next**.   **Sequential Block Reading** – **Each file block is read sequentially** even if the blocks reside on multiple data nodes and could theoretically be read in parallel.   * **Block size:** 64MB |

**Hadoop Distributed File System (HDFS) Architecture**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **User Process**   * Connected to HDFS through the network. * **Communicates with the name node to know where to read and write data**. | **Name Node** – **Master**   * **Manages file names and locations on disk. Provides metadata information** * **All data is persisted in memory (RAM)** * May have a **secondary name node used to offload processing (e.g. writing logs) off the primary. Secondary is not for high availability.** * All writes must be acknowledged by the name node before they can be acknowledged to the user process. | **Data Node** – **Slave**   * Persistent storage disks for the data. * Data is replicated across multiple data nodes if possible across multiple racks. |

**Limitations of HDFS**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Mutability**  Data is write once, read many. | **Block Size**  Single block size (e.g. **64MB**) for disk I/O, replication and sharding | **POSIX Semantics**  Must use the command “**hadoop fs**” to access the data. **Example POSIX Commands:** Open, close, read, write. | **Availability**  No snapshot or built-in mirroring capability. | **Scalability**  Name node only scales to 100M files. This is **due to the single name node persisting all data in RAM**. | **Performance**  Written in **Java** and runs on a block device |

**Overview of MapR File System (MapR-FS)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Physical Disk** – A single hard drive.  **Storage Pool** – Three striped physical disks. Striping is used to increase write performance. | **Node** – A set of storage pools.  **Topology** – A set of nodes. | **Container** – **Unit of shared storage**. It is the size of replicated data. A **storage pool has multiple containers**. Each container belongs to only one volume. | **Volume** – A tree of files and directories grouped for the purpose of applying a policy or set of policies. |  |

**MapR-FS Volume Features**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Topologies**  Provide data placement policies. | **Compression**  Compress data as it is written to disk. | **Mirroring**  Copy data locally or remotely for protection **in real time for load balancing, backup, and disaster readiness.** | **Snapshots**  Maintain point-in-time data and updates. | **Quotas**  Restrict total capacity per-user or per-group. | **Permissions**  Restrict access to users or groups. | **Replication**  Replicate containers in a volume across the cluster |

**Differences between MapR-FS and HDFS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Block Size**  MapR-FS supports different block sizes for sharding, replication, and performing I/O. | **Mutability**  MapR-FS has full read write capability. | **Access**  MapR-FS volumes can be NFS-mounted. | **POSIX Support**  MapR-FS supports native OS commands to access data. | **Availability**  MapR-FS supports snapshots and local/remote mirroring support. | **Scalability**  No limit to the number of files. | **Performance**  **MapR-FS is written in C** and **runs on a raw device.** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Block Size Comparison**  **between HDFS and MapR-FS**   |  |  |  | | --- | --- | --- | | **Storage Unit** | **HDFS** | **MapR-FS** | | Unit of Sharding | Block=64MB | Chunk=256MB | | Unit of Replication | Block=64MB | Container = 16-32GB | | Unit of I/O | Block=64MB | Block=8KB |   MapR-FS allows for different storage unit sizes to optimize performance. | **Role of a Single Sharding Unit** (e.g. **Block/Chunk**)– In Map Reduce, **each mapper is assigned a single shard** (e.g. block/chunk) to analyze.  **Relationship between Container and Volume** – In MapR-FS, a container is assigned to a single volume and a volume is made up of one or more containers.  **Example Block/Chunk Count Calculation**: If a Map Reduce file has 300MB of data, it will required 5 blocks in HDFS and 2 chunks in MapR-FS. | **Using the “hadoop fs” Command Line Interface (CLI)**  **Format:**  hadoop fs **-**<command> [args]  **Examples:**  **hadoop fs -mkdir newDirectory**  **hadoop fs –rm my\_file.txt**    **Not Supported Command:**  hadoop fs –cd …  **This command has no directory state so must use absolute path.** |

**Lecture #03 – Introduction to Map Reduce**

|  |  |  |  |
| --- | --- | --- | --- |
| **Map Reduce Underlying Principle**  **Divide and Conquer**  **Derives from Lisp** | **map**(String key, **String** value):  // key: document or shard name  // value: document or shard contents  **for each** word w **in** value:  **EmitIntermediate**((w,”1”)); // key value pair | **reduce**(String key, **Iterator** values):  // key: a word  // values: a list of word counts  **int** results = 0  **for each** v **in** values:  results += **ParseInt**(v)  **Emit**(**AsString**(result))  **Reduce is called one on each key NOT each partition.** | **Key Methods**  **EmitIntermediate** – Output of the mapper function. Writes an intermediary key-value pair to be analyzed by a reducer.  **Emit** – Outputs the result of the reducer. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Three Phases of**  **Map Reduce**   1. **Map** 2. **Sort/Shuffle/Merge** 3. **Reduce** | **Map**   * One mapper per input split. The **“map” function is called once for each key-value pair** (i.e. **record**). * Each mapper processes a local data set and **outputs a set of intermediary key value pairs**. * “**Send the compute to where the data is**.” * **Outputs zero or more key value pairs.** | **Sort/Shuffle/Merge**   * Transfer results from mappers to reducers. * Creates ***n*** partitions where ***n*** is equal to the number of reducers. * Divides intermediary key value pairs into the ***n*** partitions. * May run a “**Combiner**” function to merge results from the Map stage to reduce the amount of data to transfer over the network. * After keys are partitioned and merge, the keys in the partition are sorted. * Partitions are sent over the network to the reducers. Hadoop uses HTTP while MapR-FS uses RPC. | **Reduce**   * One reducer per input partition. The “**reduce” method is called once per key**. * **Outputs zero or more key value pairs.** * Reads one list of values for each key. * **No data locality exploitation in reduce**. |

|  |  |  |
| --- | --- | --- |
| **Responsibilities of the**  **Map Reduce Framework**   * **Split the incoming input file and read the records.** * **Schedules, runs, and reruns map/reduce tasks.** * **Transfers map outputs to reduce inputs.** * **Collects and writes status and results.** | **Map Reduce Block and Record Splitting**   * The Map Reduce framework divides an input file to one or more **splits**\**block**. * A split\block contains one or more (typically many) **records**. **Default record delimiter is “\n”.** * **The map function is called once per record**.   **Map Record Key-Value Format**   * **key** – **Byte offset** for start of record * **value** – Record data in the split. | **Typical Map Reduce Workflow**   1. **Load the data into the cluster.**    * **HDFS** – Uses WORM (write once read many). Preload only.    * **MapR-FS** – POSIX + network file system (NFS) access. Preload or persistent storage. 2. **Analyze the data** 3. **Store the results in the cluster** (e.g. in HDFS/MapR-FS)   **Read the results from the cluster.** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Map Reduce Complete Flow**   1. Data is loaded into HDFS 2. The job decides the input format of the data. 3. Data is split between different mappers running on all the nodes. 4. **Record readers** (RR) parse out the data key-value pairs serve as inputs into the map() methods. | **Map Reduce Complete Flow**   1. The map() method produces key-value pairs that are sent to the **partitioner**. 2. When there are multiple reducers, the partition mapper creates one partition for each reduce task. 3. The key-value pairs are **sorted** by key within each partition. | **Map Reduce Complete Flow**   1. The reduce() method is take the intermediary key value pairs in the partition and reduces them to a **final list of key value pairs**. 2. The job defines the output format of the data. |
| **Example Partition Function** |

**Hadoop Classes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **InputFormat**   * Checks if the input file exists. * Splits the input file into one or more **InputSplit** objects. * Instantiates **RecordReader** to partition splits into records which are turned into key-value pairs.   + **Key is byte offset** of the start of the record. | **Mapper**   * Implements the map() method. * One Mapper object is created for each input split. * Processes keys and/or values. * Updates status in reporter. * Writes output. | **Partitioner**   * Takes the output(s) generated by the map() method and **creates partitions based on the hashed key**. * Each partition is assigned to a single reducer. * **All records with the same key are assigned to the same partition**. | **Combiner (Optional)**   * Has **no default behavior**. * **Motivation**: Reduce the intermediate values of the mappers before they are sent over the network. * **Often the reducer can be repurposed as a combiner**. | **Mapper**   * Implements the reduce() method. * Each Reducer object is assigned one partition. * Executes the reduce method on each key in the partition. * Updates status in reporter. * Writes output. |

**Lecture #04 – Writing a Map Reduce Program**

**MapReduce Program Imports**

|  |  |  |  |
| --- | --- | --- | --- |
| **org.apache.hadoop.mapreduce.\***  Includes the definition of the“**Mapper**”, “**Reducer**”, “**Job**”, and “**Context**”classes. | **org.apache.hadoop.io.\***  Includes the definition of the“**Text**”, “**LongWritable**”, and “**IntWritable**”classes. | **org.apache.hadoop.conf.\***  Includes the definition of the“**Configured**”and “**Configuration**” classes. | **org.apache.hadoop.util.\***  Includes the definition of the“**Tool**” interface and “**ToolRunner**”class. |
| **org.apache.hadoop.mapreduce.lib.input.\***  Includes the definition of the“**TextInputFormat**” and “**FileInputFormat**”classes. | **org.apache.hadoop.fs.\***  Includes the definition of the“**Path**”class. | **java.util.\***  Includes the definition of the“**StringTokenizer**”class. | **java.io.\***  Includes the definition of the“**IOException**”class. |
| **org.apache.hadoop.mapreduce.lib.output.\***  Includes the definition of the“**FileOutputFormat**”class. |  |  |  |

**MapReduce Class Definitions**

|  |  |  |
| --- | --- | --- |
| **Mapper Class Format**  import java.util.\*;  import java.io.\*;  import org.apache.hadoop.mapreduce.\*;  import org.apache.hadoop.io.\*;  public class *MyMapper*  extends **Mapper** <InputKeyClassName,  InputValueClassName,  OutputKeysClassName,  OutputValuesClassName> {  }  Must override the “**map**” method. | **Reducer Class Format**  import java.util.\*;  import java.io.\*;  import org.apache.hadoop.mapreduce.\*;  import org.apache.hadoop.io.\*;  public class *MyReducer*  extends **Reducer** <InputKeyClassName,  InputValue**s**ClassName,  OutputKeysClassName,  OutputValuesClassName> {  }  Must override the “**reduce**” method. | **Driver Class Format**  import org.apache.hadoop.conf.\*;  import org.apache.hadoop.io.\*;  import org.apache.hadoop.mapreduce.\*;  import org.apache.hadoop.mapreduce.lib.input.\*;  import org.apache.hadoop.util.\*;  public class ReceiptsDriver extends Configured  implements Tool {  …  public static void main(String[] args) throws Exception{  Configuration conf = new Configuration();  System.exit(**ToolRunner.run**(conf, new ReceiptsDriver(), args));  }  }  Must override the “**run**” method. |

**MapReduce Class Method Definitions**

|  |  |  |
| --- | --- | --- |
| **map Function Format**  **@Override**  public void map(Text key, Text value, Context context)  throws IOException, InterruptedException {  StringTokenizer strToken = new StringTokenizer(value, splitCriteria);  // Iterate through all the tokens in the record  while(strToken.hasTokens()){  }  // Emit any intermediate <key, value> pairs  // Optional to emit any pairs.  **context.write**(new OutputKeysClassName(…),  new OutputValuesClassName (…));  }  Called once per input record. | **reduce Function Format**  **@Override**  public void reduce(Text key, **Iterable**<Text> value, **Context** context)  throws IOException, InterruptedException {  // Parse the Iterable object  **for**(Text value: values)  }  // Emit any intermediate <key, value> pairs  // Optional to emit any pairs.  **context.write**(new OutputKeysClassName(…),  new OutputValuesClassName (…));  }  Called once per intermediate key. | **run Function Format**  **@Override**  public int run(String[] args) throws Exception {  if(args.length != 2){  **System.err.printf**("usage: %s [general options] <inputfile> <outputfile>\n",  getClass().getSimpleName());  **System.exit**(1);  }    // Configure the job  Job job = new Job ( **getConf**(), “job name”);    job.setJarByClass(*MyDriver*.class);  job.setMapperClass(*MyMapper*.class);  job.setReducerClass(*MyReducer*.class);    // Define input file's format (e.g. text file)  job.setInputFormatClass(TextInputFormat.class);    // Setup the mapper output classes.  // Mapper Input class are a LongWritable by default and Text  job.setMapOutputKeyClass(*MapperOutputKeysClassName*.class);  job.setMapOutputValueClass(*MapperOutputValuesClassName*.class);    // Set the reducer's output class.  job.setOutputKeyClass(*ReduceOutputKeysClassName*.class);  job.setOutputKeyClass(*ReduceOutputValuesClassName*.class);  // Set the reducer's output class.  FileInputFormat.**addInputFormat** (job, new Path(<*inputfilepath*>);  FileOutputFormat.**setOutputFormat** (job, new Path(<*outputfolderpath*>);  // Wait for the job to finish.  return **job.waitForCompletion**(true) **?**  0 **:** 1;  } |