Rusu Wu
11694764
CPT_s415
Assignment_3

1. **a)**

Speed up means to use more resources to deal with a task, which proportionally uses less time for the task.

Scale up is the factor that expresses how much more work can be done in the same time period by a larger system

Since there are startup cost, interference and skew during the operation, we cannot do better than linear speedup. Specifically, there are startup costs when initializes each process. Also, interference happens when the system compete for shared resources. As for skew, it is difficult to divide a task into exactly equal-sized parts, and the response time (latency) is determined by the largest part.

b)

Shared Memory:

Pros: Efficient Communication via data in memory, accessible by all

Cons: Not Scalable; Shared memory and network become bottleneck; Not scalable beyond 32/64 processors; Cache coherence problem when data is updated.

Shared Disk:

Pros: Have a good sense of Fault tolerance; Scalability

Cons: Disk subsystem is a bottleneck; Not scalable beyond a couple of hundred processors.

Shared Nothing:

Pros: Scalable: Only queries and result relations pass through the network; Cheap to build

Cons: Hard to program.

2.a)

Let U be a set of all users: {U1, U2, ..., Un}.

Then the goal is to find common friends for every pair of (Ui, Uj) where $i \neq j$.

Let {A1,A2,...,Am} be a set of friends for User1 and {B1,B2, ...,Bn} be a set of friends for User2 , then common friends of User1 and User2 can be defined as the intersection (common elements) of these two sets.

Main(){

Input: <$k1$,$v1$>, <$k1$,$v1$> is at the form of {A1,A2,...,Am}

Map: <$k1$,$v1$>$\rightarrow list$<$k2$,$v2$>

Reduce: <$k2$,$list\ v2$>→<$listk3$,$v3$>

Combine all the key by implementing sort algorithm based on the ASCII code to return the lexicographical order of data.

}

Function Map (){

Applied to each pair, computes key-value pairs <$k2$,$v2$>

The intermediate key-value pairs are hash-partitioned based on $k2$

Each partition <$k2$,$listv2$>is sent to a reducer

}

Function Reduce () {

Takes a partition as input and computes key-value pairs <$k3$,$v3$>//(pair of people, count of mutual friends)

}

b)

First round: generate <$k1$,$v1$>

For all r∈R, do counting algorithm.

Output <$k1$,$v1$>, k1 is the word, v1 is the value of counting.

Second round:

Function Map (){

Applied to each pair, computes key-value pairs <$k2$,$v2$>

The intermediate key-value pairs are hash-partitioned based on $k2$

Each partition <$k2$,$listv2$>is sent to a reducer

}

Function Reduce () {

Takes a partition as input and computes key-value pairs <$k3$,$v3$>

}

Based on the output <$k3$,$v3$>, sort the k3 (word) into decreasing order by v3 and then we could get the most frequent keyword.

3.

a) A RDD is a read-only, in-memory partitioned collection of records. Spark RDDs are fault tolerant as they use RDD lineage retrieval. RDDs track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself.

b)

Spark uses random access memory (RAM) instead of reading and writing intermediate data to disks. Hadoop stores data on multiple sources and processes it in batches via MapReduce.

Spark uses "lazy evaluation" to form a directed acyclic graph (DAG) of consecutive computation stages. In this way, the execution plan can be optimized.

One of the main limitations of MapReduce is that it persists the full dataset to HDFS after running each job, which is very expensive. Spark passes the data directly without writing to persistent storage. This is an innovation over MapReduce. The main innovation of Spark was to introduce an in-memory caching abstraction. This makes Spark ideal for workloads where multiple operations access the same input data.

Also, spark can launch tasks much faster. MapReduce starts a new JVM for each task, which can take seconds with loading JARs, JITing, parsing configuration XML, etc. Spark keeps an executor JVM running on each node, so launching a task can me more faster.

c)

Narrow dependencies: Each partition of the parent RDD is used by at most one partition of the child RDD.

Wide dependencies: Each partition of the parent RDD may be depended on multiple child RDD.

Follow is how Spark determines the boundary of each stage. Once you perform any action on an RDD, Spark context gives your program to the driver. The driver creates the DAG (directed acyclic graph) or execution plan (job) for your program. Once the DAG is created, the driver divides this DAG into a number of stages. These stages are then divided into smaller tasks and all the tasks are given to the executors for execution.

Putting operators into stages means that plans are executed in stages rather than performing them all at once. In this case, Spark can reinitiate a task at a specific state rather than from the beginning. Putting operators into stages can benefit to distributed processing, which can help use partitions efficiently. Also, increasing the number of Executors on clusters also increases possibility of parallelism in processing Spark Job.