PERFORMANCE ANALYSIS OF VARIOUS CONCURRENCY CONTROL MECHANISMS USING HDFS

1 ABSTRACT

This report encapsulates a comprehensive study regarding the performance analysis of various Concurrency Control mechanisms using Hadoop Distributed File System (HDFS). As Concurrency Control is one of the most important functions provided by DBMS, and is very essential for good DBMS performance, this report intends to capture the way in which contents of the files are not left in inconsistent state due to interleaving actions of multiple users accessing same file. We strategically chose HDFS since it is suitable for applications with large datasets and allows us to store and process large datasets in a reliable, distributed, cost-effective manner.

2 TABLE OF CONTENTS

3 INTRODUCTION & PROBLEM STATEMENT

Concurrency Control is the process of allowing multiple users to perform simultaneous execution of transactions (i.e., sequence of reads and writes) in a shared database, preserving ACID (Atomicity, Consistency, Isolation, Durability) properties. Concurrency Control is one of the most important functions provided by DBMS and is very essential for good DBMS performance. However, interleaving actions of different transactions from multiple users might lead to inconsistency and various anomalies, few of which are WR conflict, RW conflict, WW conflict.

DBMS ensures such problem don't arise and users can pretend as they are using a single user system.

Concurrency Control mechanisms are very powerful constructs and there are many ways to implement them. One of the best way to achieve concurrency is to request a lock on an object before performing any transaction on it. In this project, we primarily focus on one of the well - known lock based concurrency control mechanism, namely Strict Two-Phase Locking (Strict 2PL) protocol.

In Strict 2PL each transaction must obtain a lock(s) on an object before performing any read/write actions on it. Each transaction must obtain a shared lock (S) on an object before reading, and an exclusive lock (X) on an object before writing. If a transaction holds an exclusive lock (X) on an object, no other transactions can get a lock (either shared or exclusive) on that object. Also, whenever transaction requests an exclusive lock (X) on object there should be no lock held by other transaction for that object. All locks for the object held by transaction are released only after transaction is completed. Strict 2PL allows only serializable schedules whose resulting execution is equivalent to some serial execution.

So how do we implement this Strict 2PL Concurrency Control mechanism? Again, this report serves to answer this question regarding implementation of Strict 2PL using Hadoop to make data consistent.

4 TECHNICAL DESCRIPTION

4.1 HADOOP:

It has been widely known that Hadoop is an Apache open source, Java-based programming framework that allows us to store and process very large datasets, specifically Big data in a real time distributed computing environment, protecting data privacy and security. A Big data is a collection of extremely huge volume, high velocity, and extensible variety of datasets produced from different devices and applications. While it is infeasible to store and process, this huge complex data using traditional storage and computing techniques, and with large amount of data streaming in every second from countless sources, it is very crucial to analyze them for insights that lead to concrete decisions and strategic business moves with reduced risks. However, the invention of new, innovative technologies (such as Hadoop) have eased the burden of managing and processing this Big data.

Hadoop is usually supported by Linux platform and its flavors. In our case, we have a Windows OS other than Linux, and hence we proceeded with installation of Virtualbox software package (which supports guest OS installation on host OS) and having Ubuntu-14.04 (Linux flavor) in it. Thereafter we carefully followed the steps mentioned in (Running Hadoop on Linux) and had our Hadoop-2.6.2 environment setup.

4.2 DATA ACQUISITION

The original datasets were obtained from the Kaggle website, a platform where researchers from all over the world post their data and compete to produce the best models. The original data is of competition "*Facebook V Results: Predicting Check Ins*". There are two files of original data: *teams.csv*; *submissions_metadata.csv*. The *teams.csv* file is a table of team final rankings and it includes the following fields: (*TeamId*, *TeamName*, *FinalScore*, *Ranking*, *FinalScoreFirstSubmittedDate*, *Medal*). The *submissions_metadata.csv* file is a table of metadata on competition submissions and it includes the following fields: (*SubmissionId*, *TeamId*, *DateSubmitted*, *PublicScore*, *PrivateScore*, *IsSelected*). However, in our project we are interested only in the fields (*SubmissionId*, *TeamId)* from *submissions_metadata.csv* and the fields (*TeamId, Ranking)* from *teams.csv* file.

4.3 DESIGN AND ARCHITECTURE

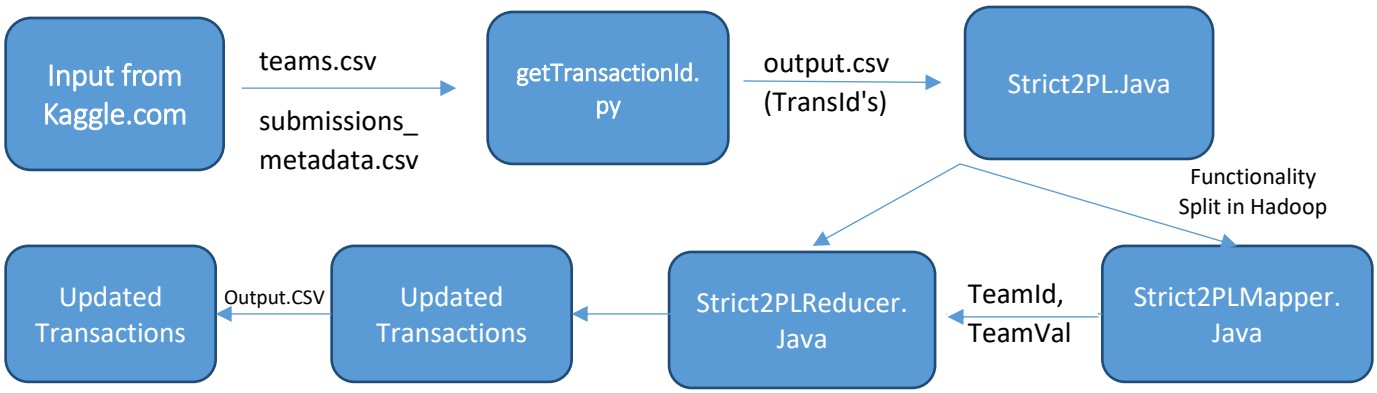


Figure 1: Architectural Design

The first and foremost inputs to our application are the two files *teams.csv* and *submissions_metadata.csv* obtained from the Kaggle website. Firstly, we parse these two input files to obtain a set of transactions (here set of SubmissionId's) for each object TeamId. We keep a log record of number of transactions, lock status ("unlocked" or "locked"), number of Shared locks held, and number of Exclusive locks held on each object TeamId. As a core functionality of Hadoop, our application is split between corresponding Mapper and Reducer class.

Given an input data Mapper class breaks down it into key-value pairs. Each transaction performs a read action on an object TeamId through the corresponding Mapper class. Given a TeamId, to retrieve corresponding Team data from the *teams.csv* file, transaction must request a Shared lock (S lock) on that Team object. Before providing a lock the corresponding lock status, count of X lock, count of S locks are checked and S lock is provided only if the corresponding Team object's X lock count is zero, thereby preserving consistency. Once S lock is obtained transaction reads corresponding Team's data.

Thereafter, reduce task, which takes the output from a map as an input and combines them into smaller set of tuples. In our application, Mapper class provides output <TeamId, {TeamName, FinalScore, Ranking, FinalScoreFirstSubmittedDate, Medal} > in the form of key-value pairs to the corresponding Reducer class. Each transaction performs a write action on an object TeamId through this Reducer class. To perform any updates on Team data, transaction must request an Exclusive lock (X lock) on that object. Again, before granting a lock request, the corresponding lock status, count of X lock, count of S locks are checked and X lock is provided only if the corresponding Team object's lock status is either "unlocked" or count of X lock is zero. Once X lock is obtained transaction updates the corresponding Team's data by modifying its Ranking field.

During each read or write actions on an object, the corresponding lock status, S lock count, and X lock count is updated simultaneously. All locks for the Team object held by transaction are released only after that transaction is completed (committed/aborted), thereby following very strict schedules.

The major advantage of employing MapReduce in our Concurrency Control application is that it allows us to process (i.e., to perform sequence of read and writes) this huge data in a scalable, flexible, and cost-effective manner.

4.4 Algorithm Implemented

In our project, we use predominantly JAVA and Python programming languages for entire implementation. The algorithm implementing Strict 2PL using Hadoop MapReduce technique is as follows.

StrictTwoPhaseLocking ()

A. Input: *teams.csv, submissions_metadata.csv*
B. Configure different job specific parameters to submit map/reduce job to Hadoop.
C. Parse the given two input files to extract a set of transactions (set of SubmissionId's) for each object Team (TeamId).
D. Create an initial log record on each object which includes lock status ("unlocked"/" locked"), number of transactions, number of Shared locks (S locks) count, and an Exclusive lock (X lock) count.

StrictTwoPhaseLockingMapper ()

A. Input: *submissions_metadata.csv* in the form of key-value pairs.
B. Split the tab separated value and obtain Transaction ID (here SubmissionId) and TeamId.
C. Read Team data from *teams.csv*
    A. If lock_status(TeamId) = "unlocked"
        Then lock_status(TeamId) = "S_lock"
        S_count(TeamId) = 1
        Read corresponding Team data from *teams.csv*

B. Else-if lock_status(TeamId) = “S_lock”
Then S_count(TeamId) = S_count(TeamId) + 1
Read corresponding Team data from *teams.csv*
C. Else /* Here lock status will be “X_lock” */
Wait until lock_status(TeamId) = “unlocked”/” S_lock”

End;

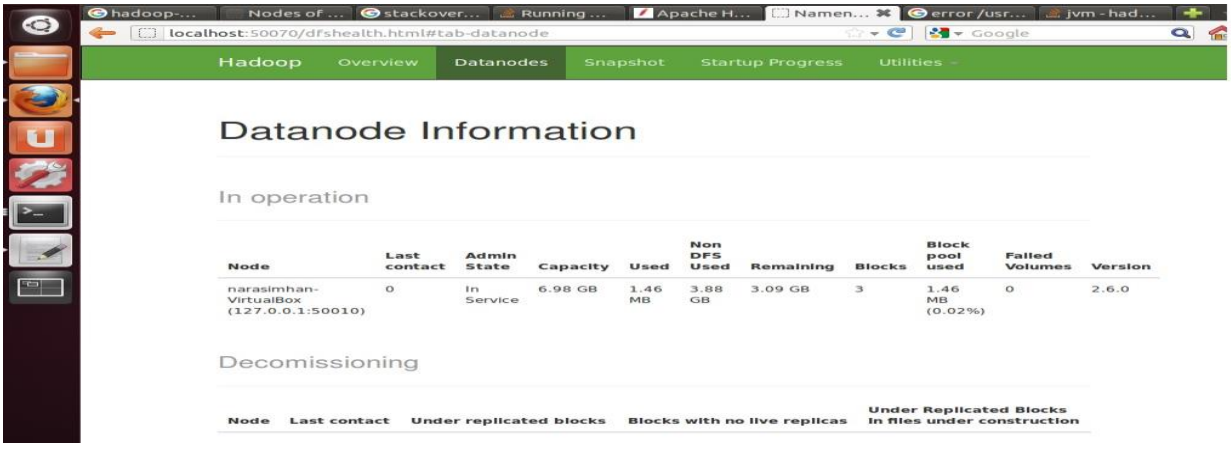
StrictTwoPhaseLockingReducer ()

A. Input: <TeamId, {*TeamName*, *FinalScore*, *Ranking*, *FinalScoreFirstSubmittedDate*, *Medal*}>
B. Write/Modify Team data.
A. If lock_status(TeamId) = “unlocked”
Then lock_status(TeamId) = “X_lock”
X_count(TeamId) = 1
Update the Ranking field
/*Releasing the lock*/
lock_status(TeamId) = “unlocked”
X_count(TeamId) = 0
S_count(TeamId) = S_count(TeamId) – 1
B. Else
Wait until lock_status(TeamId) = “unlocked”/” S_lock”

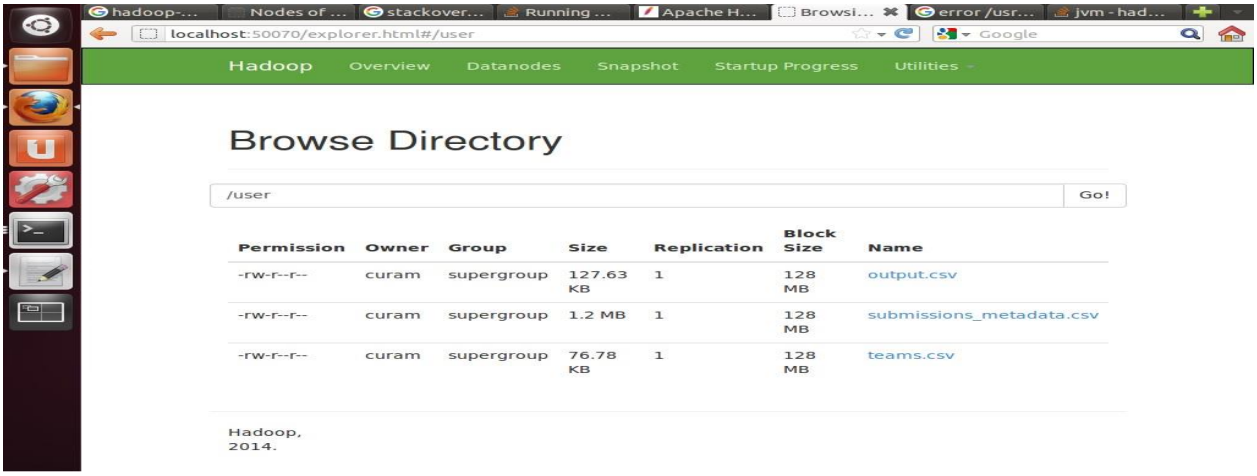
End;

5 RESULTS AND CRITIQUE

After configuring Hadoop environment, we need to start HDFS by starting namenodes and datanodes by issuing commands as shown below.



Once the namenodes and datanodes are started, we need to place our input data files in HDFS as shown.



Followed by which we need to compile our java classes to create class files to build executable jar file. So, that HDFS starts running once we unpack our jar file. However, unfortunately we were not able to perform our last step of compiling and building jar due to Hadoop technical errors as shown.

```
curam@narasimhan-VirtualBox:/usr/local/hadoop$ javac -classpath $HADOOP_HOME/share/common/hadoop-common-2.
6.0.jar:$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.6.0.jar:$HADOOP_HOME/share/hado
op/common/lib/common-cli-1.2.jar -d /usr/local/hadoop/etc/hadoop/src *.java
```

```
StrictTwoPhaseLocking.java:5: package org.apache.hadoop.fs does not exist
import org.apache.hadoop.fs.*;    /* An abstract file system API in hadoop */
^
StrictTwoPhaseLocking.java:6: package org.apache.hadoop.conf does not exist
import org.apache.hadoop.conf.*;  /* Used for configuring system parameters */
^
StrictTwoPhaseLocking.java:7: package org.apache.hadoop.io does not exist
import org.apache.hadoop.io.*;    /* Generic i/o code usage for reading/writing data to network, databases
, files etc. */
^
StrictTwoPhaseLocking.java:8: package org.apache.hadoop.util does not exist
import org.apache.hadoop.util.*;  /* Common utilities */
^
StrictTwoPhaseLocking.java:56: cannot access org.apache.hadoop.io.Closeable
```

Due to limited knowledge on Hadoop, in spite of trying multiple times, unfortunately we were not able to resolve the technical errors issued by Hadoop, which we consider as future work.

6 CONCLUSIONS

Again, this report captured implementation of Strict Two-Phase Locking (Strict 2PL) Concurrency Control mechanism using HDFS.

As illustrated, Hadoop provides native support, to store and process very large datasets in a scalable, flexible, and cost-effective manner. We learned the significance of applying concurrency control mechanisms in various cases, especially when various users want to access the data at

same time. We also learned the way HDFS can be refined by employing concurrency control mechanisms. However, Strict 2PL Concurrency Control mechanism has its own drawback like starvation and deadlock, which could be further avoided by applying deadlock prevention techniques.

As our group members, had zero knowledge with Hadoop prior to the semester, we chose to study and employ Hadoop in our project hoping that we would gain a comprehensive knowledge for future applications.

Each of our group members have contributed equally to all the elements of the project. Kartik Sooji, Kumud Bhat, and Curam Sundaram Narasimhan decided to study and understand the working of Hadoop. Followed by Kartik Sooji proceeded with setting up Hadoop environment (Hadoop 2.7.3) on both Windows and Linux OS. Also, Kumud Bhat and Curam Sundaram Narasimhan installed Hadoop (1.4.0) and (2.6.0) on Ubuntu OS. We understood Hadoop configuration and settings in all the three versions. Regarding programming implementation, we all discussed our own logic and ideas and implemented Mapper and Reducer parts of our application. Similarly, the different parts of the report were written by all of us. Since all the technology employed in this project is new to all of us, we decided to work together on all the elements of the project.

7 REFERENCES

1. Hadoop Tutorials. Retrieved from https://www.tutorialspoint.com/hadoop
2. Hadoop Internals. Retrieved from http://ercoppa.github.io/HadoopInternals/
3. Apache Hadoop-Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Apache_Hadoop
4. Big Data Insights. Retrieved from http://www.sas.com/en_us/insights/big-data.html
5. Running Hadoop on Linux. Retrieved from http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/
6. Hadoop Cluster – Architecture, Core Components and Work-flow. Retrieved from http://saphanatutorial.com/hadoop-cluster-architecture-and-core-components/
7. Bansal, N., Upadhyay, D., & Mittal, U. (2014, September). Concurrency control techniques in HDFS. In *Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference*- (pp. 87-90). IEEE.
8. Transaction Management Overview. Retrieved from Raghu Ramakrishnan slides in the course lectures (Lecture 10).