Dynamic File System STATISTICS TOOL

Users of file systems can easily run a simple script (something like "find . -ls") to collect a snapshot of the attributes of all files in a subtree. A recent CMU student did a project on this for supercomputing file systems (www.pdsi-scidac.org/fsstats). Great data for some uses, but the distribution of files at rest is not necessarily the distribution of files in motion (being accessed) and performance is almost always about the files in motion.

In this project you will build a tool that computes dynamic statistics of files recently accessed on the OpenCloud Hadoop cluster to help users understand their application and cluster performance. Your tool will process logs from HDFS processes (the NameNode and the DataNodes, and maybe other logs if you are creative) to generate useful statistical output. The tool should alternatively also save intermediate results in a database so that the output can be incrementally updated as new log data arrives. Your goal is to build a tool that is robust enough to be released as an open source project and deployed as part of the OpenCloud software support infrastructure.

What dynamic statistics should be computed?

Please compute a graph showing the distribution of work done (bytes accessed) as a function of file, per user, and for all users, over the time period of the logs available to you. By distribution we mean a set of percentiles referring to the work a user requests pertaining to a set of files.

The Xth percentile (X %tile) of a set of numbers is the element of the set for which X% of the set have smaller or equal value and (100-X)% have larger or equal value.

The most interesting 15 percentiles are:

- the quartiles: smallest (0%tile), 25%tile, median (50%tile), 75%tile, largest (100%tile), and
- the first 5 nines: 90%tile, 99%tile, 99.9%tile, 99.99%tile, 99.999%tile, and (100-90)%tile, (100-99)%tile, (100-99.9)%tile, (100-99.99)%tile, (100-99.999)%tile.

Given that the X%tile is always less than or equal to the Y%tile whenever X<Y, these 15 numbers will be ordered, and can be simply graphed as a single bar with 15 ticks, making 15 regions in the bar, one stacked on top of another.

The set of numbers that should be used to compute percentiles, per user, are the total bytes accessed (read or written) for each file the user accessed at all (whether they own the file or not). That is, the set has one number per file accessed, and each number is the total bytes written to a single file plus the total bytes read from that file. Not that if the file is replicated, please record all the bytes written, not just one copy per write.

One additional number should be stacked on top of these 15 numbers: the total work (bytes accessed) by all files during the period of the logs, for this user. Since the percentiles refer to the total work done for one file, the total for all files will at least as large as the total work done by the file that did the largest amount of work; that is, it will be the top number in the stack.

The graph's y-axis should be log base 10 of the "bytes accessed".

The graph's x-axis should be a bar chart, one bar per user and one for all users combined, sorted (ascending) by the total bytes accessed by each user. This will put the bar for all users last, on the far right of the graph.

Label the X-axis with the user's account name, or ALL.

Where are the HDFS logs?

Visible on every compute node and the head (login) node in HDFS, a set of logs from OpenCloud has been provided for you to use. The logs contain data from 2010 to 2013.

NameNode logs are stored in the following directory (in HDFS):

/data/opencloud-logs/namenode/hadoop3/

The NameNode log files of interest have names in the format:

hadoop-global-namenode-HOST.log.YYYY-MM-DD.bz2

where HOST is either “hdfsname” or “hadoop3” (the NameNode’s hostname was changed during collection, but all the data is from the same HDFS filesystem), and YYYY-MM-DD is the date (e.g. 2013-09-21). The average size of a NameNode file is about 60 MB.

DataNode logs are stored in per-node host subdirectories:

/data/opencloud-logs/datanode/HOST/

with filenames of the format:

hadoop-global-datanode-HOST.log.YYYY-MM-DD.bz2

where HOST is the hostname of the DataNode (cloud1 to cloud64) and YYYY-MM-DD is the date. The average size of a DataNode log file is about 2MB.

Logs are stored bzip2 compressed. Hadoop has a "codec" library to decompress bzip2 files as it reads them.

In reality, sometimes we lose a node at the wrong time and lose information, often a datanode log file. You should think about what effect the missing data has on your

results. Ideally you indicate the scope of the "error", or "clean" the data by estimating or bounding the missing values, though this project is not about this problem (so if you do nothing that is okay). Your thoughts on this would be good material for your writeup.

How does one interpret the HDFS logs?

Namenode logs contain many lines, but the ones you are most interested in are:

<date> <time> <logLevel> <class> ugi=<ugi> ip=<ip> cmd=create src=<path-to-file> dst=null perm=<perm>

<date> <time> <logLevel> <class> ugi=<ugi> ip=<ip> cmd=open src=<path-to-file> dst=null perm=<perm>

<date> <time> <logLevel> <class> BLOCK* NameSystem.allocateBlock: <path-to-file> <blkname>

<date> <time> <logLevel> <class> BLOCK* NameSystem.addStoredBlock: blockMap updated:
<ip>:<port> is added to <blkname> size <int>

The first indicates that the file is created, the second that the file has been opened for reading, the third that the application's writes have added an HDFS block/chunk to a file, and the fourth indicates the actual size of each replica of that block/chunk.

This information is sufficient to calculate the bytes written for a file since each replica will get its own "blockMap updated" log record.

ugi is user,group. Some log entries may fail to record a ugi. In this case treat the user as "unknown".

e.g.: 2011-09-01 00:00:00,381 INFO org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit:
ugi=iyu,lcs ip=/10.0.0.245 cmd=open
src=/user/ah1z/MED11/dev/SINconcepts/csift_harris_spbof_dist/list.081.39.lineNo
dst=null perm=null

e.g.: 2011-09-01 00:00:00,580 INFO org.apache.hadoop.hdfs.StateChange: BLOCK*
NameSystem.allocateBlock: /user/iyu/MED11/dev/sift/bof/video/DEV/HVC805374.tar.gz.
blk_8215866574467495788_38544776

e.g.: 2011-09-01 00:00:00,947 INFO org.apache.hadoop.hdfs.StateChange: BLOCK*
NameSystem.addStoredBlock: blockMap updated: 10.0.0.49:40010 is added to
blk_8215866574467495788_38544776 size 50725

To determine the bytes read from the logs, use the DataNode log lines:

<date> <time> <logLevel> <class> src: /<ip>:<port>, dest: /<ip>:<port>, bytes: <int>, op:
HDFS_READ, cliID: <clientid>, srvID: <serverid>, blockid: <blkname>

This record indicates that a block/chunk is being read from disk on the src and delivered to an application on the dest node.

e.g.: 2011-09-01 00:01:38,417 INFO org.apache.hadoop.hdfs.server.datanode.DataNode.clienttrace: src: /10.0.0.3:40010, dest: /10.0.0.245:52143, bytes: 15586, op: HDFS_READ, cliID: DFSClient_739810807, srvID: DS-1495917472-10.0.0.3-40010-1267430882598, blockid: blk_1094482346259517430_38492340

If a block/chunk is created before the given log period, you may not be able to determine which file it belongs to. In this case, mark it as "unknown file." Normally, we would use all logs available and try to get file-to-block mapping from the filesystem to solve this problem. For each block read from datanodes, the user needs to open the file first, so there's a log entry in namenode for the open command. The ugi information shows which user opens the file. It doesn't need to match the file's ugi information.

If you want to see file closes, then look in NameNode logs for:

<date> <time> <logLevel> <class> DIR* NameSystem.completeFile: file <path-to-file> is closed by <clientid>

e.g.: 2011-09-01 00:00:01,156 INFO org.apache.hadoop.hdfs.StateChange: DIR* NameSystem.completeFile: file /user/iyu/MED11/dev/sift/bof/video/DEV/HVC805374.tar.gz is closed by DFSClient_379960101

Also, MapReduce typically builds a file in a temporary name, then renames it. So ideally, the work done on the file should be the writes to the temporary name plus the reads to the final name. Renames are also in NameNode logs as:

<date> <time> <logLevel> <class> ugi=<ugi> ip=<ip> cmd=rename src=<path-to-file> dst=<path-to-file> perm=<perm>

e.g.: 2011-09-01 21:27:30,279 INFO org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit: ugi=jbetteri,lcs,clueweb ip=/10.0.0.2 cmd=rename src=/tmp/jbetteri/factzor/nell-357/arg-type-pair-training-examples/_temporary/_attempt_201106031747_8042_m_000111_0/part-m-00111 dst=/tmp/jbetteri/factzor/nell-357/arg-type-pair-training-examples/part-m-00111 perm=jbetteri:supergroup:rw-r--r--

Code structure suggestions

For each reduce task in a MapReduce job there is a different output file. Naturally it will not make sense to try to generate the graph directly in the reduce tasks, unless you use only one reduce task (often not a fast solution). Your code should be a program that submits work to Hadoop, waiting for the Hadoop job to finish, then reads the outputs of the Hadoop job into order to finish the graph. You should strive to do as little as possible in this "serial" code, so that your total run time (measured by timing the entire run) scales well with larger data.

Hint: worry about the graph generation code last -- a table of the data that belongs in the graph is a pretty good first output.

Hint: when you do worry about the graphing, I recommend using www.gnuplot.info

Incremental Updates using a noSQL Database

Log processing to derive usage patterns is usually run repeatedly every few days. Each run of the tool takes as long as it takes to reproccess the entire scope of the logs. But users don't like waiting, and they might ask for a slightly different report or for an updated report taking into account newly collected log files. A different report, if not anticipated, would require changing the log processing code – something that users will resist doing. An updated report requires processing new log records and incrementally adding that data to data that has already been collected and processed.

A natural reaction to this is to divide the log processing into two stages. The first stage does as much heavy lifting as possible, generating summary data with as much flexibility as possible (so that different questions can be answered from the summary data without reprocessing the original data), and storing the summary data in some convenient intermediate format. Then the second stage is run much later, parameterized by all the different ways the users want to ask questions, and does the much smaller processing from the stored summary data.

A (noSQL or newSQL) database is a "convenient intermediate format" for the summary data. The schema for the database is, informally, the definition of the data that is used for the primary key (first column) and the column definitions (e.g. the headings) for the rest of the database table.

MapReduce programs can "insert rows and columns" into the database instead of writing output data. This is storing the summary data in stage 1.

The second stage program can also be MapReduce, though it can also be just a script or other serial program, perhaps mostly for producing the graphs or other GUI output.

Modify your log processing code to generate intermediate summary data. You will invent a schema, build a HBASE noSQL database to use this schema, and insert the summary data into the HBASE database. Your tool should be able add data from newly collected log files to the HBASE summary data (i.e. it should be able to take one day of logs and add it to the database). Develop a second stage that reads summary data out of HBASE and generates the user's requested output.

What should your database contain?

The design of a database often begins: "what queries do you want to do with this database"? Here are some queries that this project should try to handle:

- Show the percentiles for the amount of work done on a file over a two month period for all users
- Show the total amount of data written and then read back by user XXX last week
- Show the per-user amount of data written and read by files that were created and deleted during the month, sorted by per-user work.

You should think about what data you want in the database so you can easily construct an answer to these questions.

Performance

Hadoop program performance is dependent on many things. There are simple parameters, like the number of map tasks (determined by the amount of input data per map) and the number of reduce tasks (a invocation parameter). These are obvious parameters for you to explore, but not enough. You should think about data efficiency – compression, combining, pre-reducing, network copying, sort efficiency, to name a few. And of course, your program's algorithm might be improved. If you are doing the simplest multi-pass MapReduce, then you are not taking full advantage of the large memories on OpenCloud, and it may be possible to reduce the number of passes or increase the efficiency of each pass.

Hand in

Your final report should describe your tools design and database schema and measure their speed and efficiency. It should include graphs of filesystem work (as described above). For processing newly collected log files, you should evaluate the performance benefits of storing intermediate data in HBASE vs reprocessing all the logs.

You may do only what has been suggested in this document. But your goal is to develop a fast, low cost, log processing tool/framework for helping users and administrators understand what is happening in the cluster storage, in order for them to debug and improve performance. Ambitious projects will invent alternative displays and metrics, and more usable and reliable frameworks.

You should also attach your source code.

Each exam and lab assignment must be the sole work of the student turning it in. You may discuss the project with others, but all the lines of code should be written by you, and you must not copy another person's code, even if they have written the code on a common white board. Both the person copying and the person allowing someone else to copy their code are said to be cheating in these cases, so be sure to protect your work with passwords and non-public storage.

However, we do encourage pointing one another to information online, we encourage advising one another of problems they may face, clarifying ambiguities, discussing high-level design issues, and giving general (not code-specific) debugging advice.

The usual penalty for cheating is to be removed from the course with a failing grade. The University also places a record of the incident in the student's permanent record.