***HDFS reads, writes, access concurrency explained.***

If your client is running on a server that is not part of the Hadoop cluster, then there is almost no practical limit to the number of open HDFS files.

Each Datanode can have several thousand simultaneous open connections, and the open files will be distributed randomly among all Datanodes.

Of course you should also consider if there are other loads on your cluster that might also be enthusiastic readers or writers.

If your client is actually running on a Datanode, which is not unusual in small operations or in laboratory setups, you should be aware that the first

copy of all blocks of the files being written will be directed to the local Datanode as an optimization. In this case, you might want to limit the

number of open HDFS files to 1000 or so, and/or distribute the ingest among several client instances on multiple Datanodes.

You should probably be more concerned about resources on your client, which will be subject to the local OS limit on the number of simultaneously open

files being read for ingestion, and the number of simultaneously open connections to HDFS streams being written.

The append operation is used to extend previously closed files. Multiple client instances could potentially try to simultaneously append

to the same HDFS file. Of course, in all cases (create, write and append), only one client at a time has write access to any given file.

any number of simultaneous readers are allowed, including during write.

Regarding buffering, for each output stream, the HDFS client buffers 64kb chunks of data locally in RAM before flushing to the Datanode,

unless you use explicit hflush calls. When the buffer fills, hflush is automatically called.

After an hflush call returns successfully, the data is guaranteed to be in the Datanodes, safe from client failures, and available for reading by other clients.

The file browser UI is served by the Namenode. As long as it takes to fill up a block, the Datanodes handle all the communication with the client,

and replication. It is too expensive to update the Namenode every time the Datanode receives a write operation.

Only when the block fills up and a new block allocation is needed, or when the file is closed and the block is terminated, then the Namenode will be updated.

Blocks are usually 128MB or more.

The Namenode is aware of all complete blocks and the block in progress for each open file, and will deliver that information to clients that ask for it.

The clients can successfully read as much data as has been hflush'ed so far, even if it goes beyond the length the Namenode knows at that point.

***HBase access concurrency explained***

Apache HBase + Apache Hadoop + Xceivers

[March 14, 2012](http://blog.cloudera.com/blog/2012/03/hbase-hadoop-xceivers/)

[By Lars George](http://blog.cloudera.com/blog/author/lars-george/)

Introduction

Some of the configuration properties found in Apache Hadoop have a direct effect on clients, such as Apache HBase. One of those properties is called “dfs.datanode.max.xcievers”, and belongs to the HDFS subproject. It defines the number of server side threads and – to some extent – sockets used for data connections. Setting this number too low can cause problems as you grow or increase utilization of your cluster. This post will help you to understand what happens between the client and server, and how to determine a reasonable number for this property.

The Problem

Since HBase is storing everything it needs inside HDFS, the hard upper boundary imposed by the “dfs.datanode.max.xcievers” configuration property can result in too few resources being available to HBase, manifesting itself as IOExceptions on either side of the connection. Here is an example from the HBase mailing list [1], where the following messages were initially logged on the RegionServer side:

2008-11-11 19:55:52,451 INFO org.apache.hadoop.dfs.DFSClient: Exception in createBlockOutputStream java.io.IOException: Could not read from stream  
2008-11-11 19:55:52,451 INFO org.apache.hadoop.dfs.DFSClient: Abandoning block blk\_-5467014108758633036\_595771  
2008-11-11 19:55:58,455 WARN org.apache.hadoop.dfs.DFSClient: DataStreamer Exception: java.io.IOException: Unable to create new block.  
2008-11-11 19:55:58,455 WARN org.apache.hadoop.dfs.DFSClient: Error Recovery for block blk\_-5467014108758633036\_595771 bad datanode[0]  
2008-11-11 19:55:58,482 FATAL org.apache.hadoop.hbase.regionserver.Flusher: Replay of hlog required. Forcing server shutdown

Correlating this with the Hadoop DataNode logs revealed the following entry:

ERROR org.apache.hadoop.dfs.DataNode: DatanodeRegistration(10.10.10.53:50010,storageID=DS-1570581820-10.10.10.53-50010-1224117842339,infoPort=50075, ipcPort=50020):DataXceiver: java.io.IOException: xceiverCount 258 exceeds the limit of concurrent xcievers 256

In this example, the low value of “dfs.datanode.max.xcievers” for the DataNodes caused the entire RegionServer to shut down. This is a really bad situation. Unfortunately, there is no hard-and-fast rule that explains how to compute the required limit. It is commonly advised to raise the number from the default of 256 to something like 4096 (see [1], [2], [3], [4], and [5] for reference). This is done by adding this property to the hdfs-site.xml file of all DataNodes (note that it is misspelled):

<property>    <name>dfs.datanode.max.xcievers</name>  
    <value>4096</value>  
  </property>

Note: You will need to restart your DataNodes after making this change to the configuration file.

This should help with the above problem, but you still might want to know more about how this all plays together, and what HBase is doing with these resources. We will discuss this in the remainder of this post. But before we do, we need to be clear about why you cannot simply set this number very high, say 64K and be done with it.

There is a reason for an upper boundary, and it is twofold: first, threads need their own stack, which means they occupy memory. For current servers this means 1MB per thread[6] by default. In other words, if you use up all the 4096 DataXceiver threads, you need around 4GB of heap to accommodate them. This cuts into the space you have assigned for memstores and block caches, as well as all the other moving parts of the JVM. In a worst case scenario, you might run into an OutOfMemoryException, and the RegionServer process is toast. You want to set this property to a reasonably high number, but not too high either.

Second, having these many threads active you will also see your CPU becoming increasingly loaded. There will be many context switches happening to handle all the concurrent work, which takes away resources for the real work. As with the concerns about memory, you want the number of threads not grow boundlessly, but provide a reasonable upper boundary – and that is what “dfs.datanode.max.xcievers” is for.

Hadoop File System Details

From the client side, the HDFS library is providing the abstraction called Path. This class represents a file in a file system supported by Hadoop, represented by the FileSystem class. There are a few concrete implementation of the abstract FileSystem class, one of which is the DistributedFileSytem, representing HDFS. This class in turn wraps the actual DFSClient class that handles all interactions with the remote servers, i.e. the NameNode and the many DataNodes.

When a client, such as HBase, opens a file, it does so by, for example, calling the open() or create() methods of the FileSystem class, here the most simplistic incarnations

  public DFSInputStream open(String src) throws IOException  
  public FSDataOutputStream create(Path f) throws IOException

The returned stream instance is what needs a server-side socket and thread, which are used to read and write blocks of data. They form part of the contract to exchange data between the client and server. Note that there are other, RPC-based protocols in use between the various machines, but for the purpose of this discussion they can be ignored.

The stream instance returned is a specialized DFSOutputStream or DFSInputStream class, which handle all of the interaction with the NameNode to figure out where the copies of the blocks reside, and the data communication per block per DataNode.

On the server side, the DataNode wraps an instance of DataXceiverServer, which is the actual class that reads the above configuration key and also throws the above exception when the limit is exceeded.

When the DataNode starts, it creates a thread group and starts the mentioned DataXceiverServer instance like so:

  this.threadGroup = new ThreadGroup(“dataXceiverServer”);  
  this.dataXceiverServer = new Daemon(threadGroup,  
      new DataXceiverServer(ss, conf, this));  
  this.threadGroup.setDaemon(true); // auto destroy when empty

Note that the DataXceiverServer thread is already taking up one spot of the thread group. The DataNode also has this internal class to retrieve the number of currently active threads in this group:

  /\*\* Number of concurrent xceivers per node. \*/  
  int getXceiverCount() {  
    return threadGroup == null ? 0 : threadGroup.activeCount();  
  }

Reading and writing blocks, as initiated by the client, causes for a connection to be made, which is wrapped by the DataXceiverServer thread into a DataXceiver instance. During this hand off, a thread is created and registered in the above thread group. So for every active read and write operation a new thread is tracked on the server side. If the count of threads in the group exceeds the configured maximum then the said exception is thrown and recorded in the DataNode’s logs:

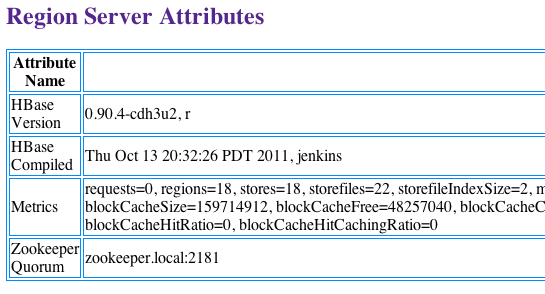
  if (curXceiverCount > dataXceiverServer.maxXceiverCount) {  
    throw new IOException(“xceiverCount ” + curXceiverCount  
                          + ” exceeds the limit of concurrent xcievers “  
                          + dataXceiverServer.maxXceiverCount);  
  }

Implications for Clients

Now, the question is, how does the client reading and writing relate to the server side threads. Before we go into the details though, let’s use the debug information that the DataXceiver class logs when it is created and closed

  LOG.debug(“Number of active connections is: ” + datanode.getXceiverCount());  
  …  
  LOG.debug(datanode.dnRegistration + “:Number of active connections is: ”     + datanode.getXceiverCount());

and monitor during a start of HBase what is logged on the DataNode. For simplicity’s sake this is done on a pseudo distributed setup with a single DataNode and RegionServer instance. The following shows the top of the RegionServer’s status page.

[](http://blog.cloudera.com/wp-content/uploads/2012/05/HadoopHBaseXceiverScreen1.png)

The important part is in the “Metrics” section, where it says “storefiles=22”. So, assuming that HBase has at least that many files to handle, plus some extra files for the write-ahead log, we should see the above logs message state that we have at least 22 “active connections”. Let’s start HBase and check the DataNode and RegionServer log files:

Command Line:

$ bin/start-hbase.sh  
…

DataNode Log:

2012-03-05 13:01:35,309 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 1  
2012-03-05 13:01:35,315 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 2  
12/03/05 13:01:35 INFO regionserver.MemStoreFlusher: globalMemStoreLimit=396.7m, globalMemStoreLimitLowMark=347.1m, maxHeap=991.7m  
12/03/05 13:01:39 INFO http.HttpServer: Port returned by webServer.getConnectors()[0].getLocalPort() before open() is -1. Opening the listener on 60030  
2012-03-05 13:01:40,003 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 1  
12/03/05 13:01:40 INFO regionserver.HRegionServer: Received request to open region: -ROOT-,,0.70236052  
2012-03-05 13:01:40,882 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 3  
2012-03-05 13:01:40,884 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 4  
2012-03-05 13:01:40,888 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 3  
…  
12/03/05 13:01:40 INFO regionserver.HRegion: Onlined -ROOT-,,0.70236052; next sequenceid=63083  
2012-03-05 13:01:40,982 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 3  
2012-03-05 13:01:40,983 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 4  
…  
12/03/05 13:01:41 INFO regionserver.HRegionServer: Received request to open region: .META.,,1.1028785192  
2012-03-05 13:01:41,026 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 3  
2012-03-05 13:01:41,027 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 4  
…  
12/03/05 13:01:41 INFO regionserver.HRegion: Onlined .META.,,1.1028785192; next sequenceid=63082  
2012-03-05 13:01:41,109 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 3  
2012-03-05 13:01:41,114 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 4  
2012-03-05 13:01:41,117 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 5  
12/03/05 13:01:41 INFO regionserver.HRegionServer: Received request to open 16 region(s)  
12/03/05 13:01:41 INFO regionserver.HRegionServer: Received request to open region: usertable,,1330944810191.62a312d67981c86c42b6bc02e6ec7e3f.  
12/03/05 13:01:41 INFO regionserver.HRegionServer: Received request to open region: usertable,user1120311784,1330944810191.90d287473fe223f0ddc137020efda25d.  
…

2012-03-05 13:01:41,246 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 6  
2012-03-05 13:01:41,248 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 7  
…  
2012-03-05 13:01:41,257 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 10  
2012-03-05 13:01:41,257 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 9  
…  
12/03/05 13:01:41 INFO regionserver.HRegion: Onlined usertable,user1120311784,1330944810191.90d287473fe223f0ddc137020efda25d.; next sequenceid=62917  
12/03/05 13:01:41 INFO regionserver.HRegion: Onlined usertable,,1330944810191.62a312d67981c86c42b6bc02e6ec7e3f.; next sequenceid=62916  
…  
12/03/05 13:01:41 INFO regionserver.HRegion: Onlined usertable,user1361265841,1330944811370.80663fcf291e3ce00080599964f406ba.; next sequenceid=62919  
2012-03-05 13:01:41,474 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 6  
2012-03-05 13:01:41,491 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 7  
2012-03-05 13:01:41,495 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 8  
2012-03-05 13:01:41,508 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 7  
…  
12/03/05 13:01:41 INFO regionserver.HRegion: Onlined usertable,user1964968041,1330944848231.dd89596e9129e1caa7e07f8a491c9734.; next sequenceid=62920  
2012-03-05 13:01:41,618 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 6  
2012-03-05 13:01:41,621 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 7  
…  
2012-03-05 13:01:41,829 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 7  
12/03/05 13:01:41 INFO regionserver.HRegion: Onlined usertable,user515290649,1330944849739.d23924dc9e9d5891f332c337977af83d.; next sequenceid=62926  
2012-03-05 13:01:41,832 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 6  
2012-03-05 13:01:41,838 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 7  
12/03/05 13:01:41 INFO regionserver.HRegion: Onlined usertable,user757669512,1330944850808.cd0d6f16d8ae9cf0c9277f5d6c6c6b9f.; next sequenceid=62929  
…  
2012-03-05 14:01:39,711 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 4  
2012-03-05 22:48:41,945 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 4  
12/03/05 22:48:41 INFO regionserver.HRegion: Onlined usertable,user757669512,1330944850808.cd0d6f16d8ae9cf0c9277f5d6c6c6b9f.; next sequenceid=62929  
2012-03-05 22:48:41,963 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 4

You can see how the regions are opened one after the other, but what you also might notice is that the number of active connections never climbs to 22 – it barely even reaches 10. Why is that? To understand this better, we have to see how files in HDFS map to the server-side DataXceiver’s instance – and the actual threads they represent.

Hadoop Deep Dive

The aforementioned DFSInputStream and DFSOutputStream are really facades around the usual stream concepts. They wrap the client-server communication into these standard Java interfaces, while internally routing the traffic to a selected DataNode – which is the one that holds a copy of the current block. It has the liberty to open and close these connection as needed. As a client reads a file in HDFS, the client library classes switch transparently from block to block, and therefore from DataNode to DataNode, so it has to open and close connections as needed.

The DFSInputStream has an instance of a DFSClient.BlockReader class, that opens the connection to the DataNode. The stream instance calls blockSeekTo() for every call to read() which takes care of opening the connection, if there is none already. Once a block is completely read the connection is closed. Closing the stream has the same effect of course.

The DFSOutputStream has a similar helper class, the DataStreamer. It tracks the connection to the server, which is initiated by the nextBlockOutputStream() method. It has further internal classes that help with writing the block data out, which we omit here for the sake of brevity.

Both writing and reading blocks requires a thread to hold the socket and intermediate data on the server-side, wrapped in the DataXceiver instance. Depending what your client is doing, you will see the number of connections fluctuate around the number of currently accessed files in HDFS.

Back to the HBase riddle above: the reason you do not see up to 22 (and more) connections during the start is that while the regions open, the only required data is the HFile’s info block. This block is read to gain vital details about each file, but then closed again. This means that the server-side resource is released in quick succession. The remaining four connections are harder to determine. You can use JStack to dump all threads on the DataNode, which in this example shows this entry:

“DataXceiver for client /127.0.0.1:64281 [sending block blk\_5532741233443227208\_4201]” daemon prio=5 tid=7fb96481d000 nid=0x1178b4000 runnable [1178b3000]  
   java.lang.Thread.State: RUNNABLE  
   …

“DataXceiver for client /127.0.0.1:64172 [receiving block blk\_-2005512129579433420\_4199 client=DFSClient\_hb\_rs\_10.0.0.29,60020,1330984111693\_1330984118810]” daemon prio=5 tid=7fb966109000 nid=0x1169cb000 runnable [1169ca000]  
   java.lang.Thread.State: RUNNABLE  
   …

These are the only DataXceiver entries (in this example), so the count in the thread group is a bit misleading. Recall that the DataXceiverServer daemon thread already accounts for one extra entry, which combined with the two above accounts for the three active connections – which in fact means three active threads. The reason the log states four instead, is that it logs the count from an active thread that is about to finish. So, shortly after the count of four is logged, it is actually one less, i.e. three and hence matching our head count of active threads.

Also note that the internal helper classes, such as the PacketResponder occupy another thread in the group while being active. The JStack output does indicate that fact, listing the thread as such:

 “PacketResponder 0 for Block blk\_-2005512129579433420\_4199” daemon prio=5 tid=7fb96384d000 nid=0x116ace000 in Object.wait() [116acd000]  
   java.lang.Thread.State: TIMED\_WAITING (on object monitor)  
     at java.lang.Object.wait(Native Method)  
     at org.apache.hadoop.hdfs.server.datanode.BlockReceiver$PacketResponder \  
       .lastDataNodeRun(BlockReceiver.java:779)  
     – locked (a org.apache.hadoop.hdfs.server.datanode.BlockReceiver$PacketResponder)  
     at org.apache.hadoop.hdfs.server.datanode.BlockReceiver$PacketResponder.run(BlockReceiver.java:870)  
     at java.lang.Thread.run(Thread.java:680)

This thread is currently in TIMED\_WAITING state and is not considered active. That is why the count emitted by the DataXceiver log statements is not including these kind of threads. If they become active due to the client sending sending data, the active thread count will go up again. Another thing to note its that this thread does not need a separate connection, or socket, between the client and the server. The PacketResponder is just a thread on the server side to receive block data and stream it to the next DataNode in the write pipeline.

The Hadoop fsck command also has an option to report what files are currently open for writing:

$ hadoop fsck /hbase -openforwrite  
FSCK started by larsgeorge from /10.0.0.29 for path /hbase at Mon Mar 05 22:59:47 CET 2012  
……/hbase/.logs/10.0.0.29,60020,1330984111693/10.0.0.29%3A60020.1330984118842 0 bytes, 1 block(s), OPENFORWRITE: ………………………………..Status: HEALTHY  
 Total size:     2088783626 B  
 Total dirs:     54  
 Total files:    45  
 …

This does not immediately relate to an occupied server-side thread, as these are allocated by block ID. But you can glean from it, that there is one open block for writing. The Hadoop command has additional options to print out the actual files and block ID they are comprised of:

$ hadoop fsck /hbase -files -blocks  
FSCK started by larsgeorge from /10.0.0.29 for path /hbase at Tue Mar 06 10:39:50 CET 2012

…  
/hbase/.META./1028785192/.tmp <dir>  
/hbase/.META./1028785192/info <dir>  
/hbase/.META./1028785192/info/4027596949915293355 36517 bytes, 1 block(s):  OK  
0. blk\_5532741233443227208\_4201 len=36517 repl=1

…  
Status: HEALTHY  
 Total size:     2088788703 B  
 Total dirs:     54  
 Total files:     45 (Files currently being written: 1)  
 Total blocks (validated):     64 (avg. block size 32637323 B) (Total open file blocks (not validated): 1)  
 Minimally replicated blocks:     64 (100.0 %)  
 …

This gives you two things. First, the summary states that there is one open file block at the time the command ran – matching the count reported by the “-openforwrite” option above. Secondly, the list of blocks next to each file lets you match the thread name to the file that contains the block being accessed. In this example the block with the ID “blk\_5532741233443227208\_4201” is sent from the server to the client, here a RegionServer. This block belongs to the HBase .META. table, as shown by the output of the Hadoop fsck command. The combination of JStack and fsck can serve as a poor mans replacement for lsof (a tool on the Linux command line to “list open files”).

The JStack also reports that there is a DataXceiver thread, with an accompanying PacketResponder, for block ID “blk\_-2005512129579433420\_4199”, but this ID is missing from the list of blocks reported by fsck. This is because the block is not yet finished and therefore not available to readers. In other words, Hadoop fsck only reports on complete (or synced[7][8], for Hadoop version that support this feature) blocks.

Back to HBase

Opening all the regions does not need as many resources on the server as you would have expected. If you scan the entire HBase table though, you force HBase to read all of the blocks in all HFiles:

HBase Shell:

hbase(main):003:0> scan ‘usertable’  
…  
1000000 row(s) in 1460.3120 seconds

DataNode Log:

2012-03-05 14:42:20,580 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 6  
2012-03-05 14:43:23,293 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 7  
2012-03-05 14:43:23,299 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 8  
…  
2012-03-05 14:49:24,332 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 11  
2012-03-05 14:49:24,332 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 10  
2012-03-05 14:49:59,987 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 11  
2012-03-05 14:51:12,603 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 12  
2012-03-05 14:51:12,605 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 11  
2012-03-05 14:51:46,473 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 12  
…  
2012-03-05 14:56:59,420 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 15  
2012-03-05 14:57:31,722 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 16  
2012-03-05 14:58:24,909 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 17  
2012-03-05 14:58:24,910 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 16  
…  
2012-03-05 15:04:17,688 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 21  
2012-03-05 15:04:17,689 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 22  
2012-03-05 15:04:54,545 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 21  
2012-03-05 15:05:55,901 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: DatanodeRegistration(127.0.0.1:50010, storageID=DS-1423642448-10.0.0.64-50010-1321352233772, infoPort=50075, ipcPort=50020):Number of active connections is: 22  
2012-03-05 15:05:55,901 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode: Number of active connections is: 21

The number of active connections reaches the elusive 22 now. Note that this count already includes the server thread, so we are still a little short of what we could consider the theoretical maximum – based on the number of files HBase has to handle.

What does that all mean?

So, how many “xcievers (sic)” do you need? Given you only use HBase, you could simply monitor the above “storefiles” metric (which you get also through Ganglia or JMX) and add a few percent for intermediate and write-ahead log files. This should work for systems in motion. However, if you were to determine that number on an idle, fully compacted system and assume it is the maximum, you might find this number being too low once you start adding more store files during regular memstore flushes, i.e. as soon as you start to add data to the HBase tables. Or if you also use MapReduce on that same cluster, Flume log aggregation, and so on. You will need to account for those extra files, and, more importantly, open blocks for reading and writing.

Note again that the examples in this post are using a single DataNode, something you will not have on a real cluster. To that end, you will have to divide the total number of store files (as per the HBase metric) by the number of DataNodes you have. If you have, for example, a store file count of 1000, and your cluster has 10 DataNodes, then you should be OK with the default of 256 xceiver threads per DataNode.

The worst case would be the number of all active readers and writers, i.e. those that are currently sending or receiving data. But since this is hard to determine ahead of time, you might want to consider building in a decent reserve. Also, since the writing process needs an extra – although shorter lived – thread (for the PacketResponder) you have to account for that as well. So a reasonable, but rather simplistic formula could be:

[http://blog.cloudera.com/wp-content/uploads/2012/03/HadoopHBaseXceiverFormula1.png](http://blog.cloudera.com/wp-content/uploads/2012/03/HadoopHBaseXceiverFormula1.png)

This formula takes into account that you need about two threads for an active writer and another for an active reader. This is then summed up and divided by the number of DataNodes, since you have to specify the “dfs.datanode.max.xcievers” per DataNode.

If you loop back to the HBase RegionServer screenshot above, you saw that there were 22 store files. These are immutable and will only be read, or in other words occupy one thread only. For all memstores that are flushed to disk you need two threads – but only until they are fully written. The files are finalized and closed for good, cleaning up any thread in the process. So these come and go based on your flush frequency. Same goes for compactions, they will read N files and write them into a single new one, then finalize the new file. As for the write-ahead logs, these will occupy a thread once you have started to add data to any table. There is a log file per server, meaning that you can only have twice as many active threads for these files as you have RegionServers.

For a pure HBase setup (HBase plus its own HDFS, with no other user), we can estimate the number of needed DataXceiver’s with the following formula:

[http://blog.cloudera.com/wp-content/uploads/2012/03/HadoopHBaseXceiverFormula2.png](http://blog.cloudera.com/wp-content/uploads/2012/03/HadoopHBaseXceiverFormula2.png)

Since you will be hard pressed to determine the *active* number of store files, flushes, and so on, it might be better to estimate the theoretical maximum instead. This maximum value takes into account that you can only have a single flush and compaction active per region at any time. The maximum number of logs you can have active matches the number of RegionServers, leading us to this formula:

[http://blog.cloudera.com/wp-content/uploads/2012/03/HadoopHBaseXceiverFormula31.png](http://blog.cloudera.com/wp-content/uploads/2012/03/HadoopHBaseXceiverFormula31.png)

Obviously, the number of store files will increase over time, and the number of regions typically as well. Same for the numbers of servers, so keep in mind to adjust this number over time. In practice, you can add a buffer of, for example, 20%, as shown in the formula below – in an attempt to not force you to change the value too often.

On the other hand, if you keep the number of regions fixed per server[9], and rather split them manually, while adding new servers as you grow, you should be able to keep this configuration property stable for each server.

Final Advice & TL;DR

Here is the final formula you want to use:

[http://blog.cloudera.com/wp-content/uploads/2012/05/HadoopHBaseXceiverFormula4.png](http://blog.cloudera.com/wp-content/uploads/2012/05/HadoopHBaseXceiverFormula4.png)

It computes the maximum number of threads needed, based on your current HBase vitals (no. of store files, regions, and region servers). It also adds a fudge factor of 20% to give you room for growth. Keep an eye on the numbers on a regular basis and adjust the value as needed. You might want to use Nagios with appropriate checks to warn you when any of the vitals goes over a certain percentage of change.

Note: Please make sure you also adjust the number of file handles your process is allowed to use accordingly[10]. This affects the number of sockets you can use, and if that number is too low (default is often 1024), you will get connection issues first.

Finally, the engineering devil on one of your shoulders should already have started to snicker about how horribly non-Erlang-y this is, and how you should use an event driven approach, possibly using Akka with Scala[11] – if you want to stay within the JVM world. Bear in mind though that the clever developers in the community share the same thoughts and have already started to discuss various approaches[12][13].

Links:

* [1] <http://old.nabble.com/Re%3A-xceiverCount-257-exceeds-the-limit-of-concurrent-xcievers-256-p20469958.html>
* [2] <http://ccgtech.blogspot.com/2010/02/hadoop-hdfs-deceived-by-xciever.html>
* [3] <https://issues.apache.org/jira/browse/HDFS-1861> “Rename dfs.datanode.max.xcievers and bump its default value”
* [4] <https://issues.apache.org/jira/browse/HDFS-1866> “Document dfs.datanode.max.transfer.threads in hdfs-default.xml”
* [5] <http://hbase.apache.org/book.html#dfs.datanode.max.xcievers>
* [6] <http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#threads_oom>
* [7] <https://issues.apache.org/jira/browse/HDFS-200> “In HDFS, sync() not yet guarantees data available to the new readers”
* [8] <https://issues.apache.org/jira/browse/HDFS-265> “Revisit append”
* [9] <http://search-hadoop.com/m/CBBoV3z24H1> “HBase, mail # user – region size/count per regionserver”
* [10] <http://hbase.apache.org/book.html#ulimit> “ulimit and nproc”
* [11] <http://akka.io/> “Akka”
* [12] <https://issues.apache.org/jira/browse/HDFS-223> “Asynchronous IO Handling in Hadoop and HDFS”
* [13] <https://issues.apache.org/jira/browse/HDFS-918> “Use single Selector and small thread pool to replace many instances of BlockSender for reads”

***Impala access concurrency explained***

<http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html>

# Admission Control and Query Queuing

Admission control is an Impala feature that imposes limits on concurrent SQL queries, to avoid resource usage spikes and out-of-memory conditions on busy CDH clusters. It is a form of "throttling". New queries are accepted and executed until certain conditions are met, such as too many queries or too much total memory used across the cluster. When one of these thresholds is reached, incoming queries wait to begin execution. These queries are queued and are admitted (that is, begin executing) when the resources become available.

In addition to the threshold values for currently executing queries, you can place limits on the maximum number of queries that are queued (waiting) and a limit on the amount of time they might wait before returning with an error. These queue settings let you ensure that queries do not wait indefinitely, so that you can detect and correct "starvation" scenarios.

Enable this feature if your cluster is underutilized at some times and overutilized at others. Overutilization is indicated by performance bottlenecks and queries being cancelled due to out-of-memory conditions, when those same queries are successful and perform well during times with less concurrent load. Admission control works as a safeguard to avoid out-of-memory conditions during heavy concurrent usage.

**Note:**

The use of the Llama component for integrated resource management within YARN is no longer supported with CDH 5.5 / Impala 2.3 and higher.

For clusters running Impala alongside other data management components, you define static service pools to define the resources available to Impala and other components. Then within the area allocated for Impala, you can create dynamic service pools, each with its own settings for the Impala admission control feature.

Continue reading:

* [Overview of Impala Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_intro)
* [Concurrent Queries and Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_concurrency)
* [Memory Limits and Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_memory)
* [How Impala Admission Control Relates to Other Resource Management Tools](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_yarn)
* [How Impala Schedules and Enforces Limits on Concurrent Queries](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_architecture)
* [How Admission Control works with Impala Clients (JDBC, ODBC, HiveServer2)](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_jdbc_odbc)
* [SQL and Schema Considerations for Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_schema_config)
* [Configuring Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_config)
* [Guidelines for Using Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_guidelines)

## Overview of Impala Admission Control

On a busy CDH cluster, you might find there is an optimal number of Impala queries that run concurrently. For example, when the I/O capacity is fully utilized by I/O-intensive queries, you might not find any throughput benefit in running more concurrent queries. By allowing some queries to run at full speed while others wait, rather than having all queries contend for resources and run slowly, admission control can result in higher overall throughput.

For another example, consider a memory-bound workload such as many large joins or aggregation queries. Each such query could briefly use many gigabytes of memory to process intermediate results. Because Impala by default cancels queries that exceed the specified memory limit, running multiple large-scale queries at once might require re-running some queries that are cancelled. In this case, admission control improves the reliability and stability of the overall workload by only allowing as many concurrent queries as the overall memory of the cluster can accomodate.

The admission control feature lets you set an upper limit on the number of concurrent Impala queries and on the memory used by those queries. Any additional queries are queued until the earlier ones finish, rather than being cancelled or running slowly and causing contention. As other queries finish, the queued queries are allowed to proceed.

In CDH 5.7 / Impala 2.5 and higher, you can specify these limits and thresholds for each pool rather than globally. That way, you can balance the resource usage and throughput between steady well-defined workloads, rare resource-intensive queries, and ad hoc exploratory queries.

For details on the internal workings of admission control, see [How Impala Schedules and Enforces Limits on Concurrent Queries](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_architecture).

## Concurrent Queries and Admission Control

One way to limit resource usage through admission control is to set an upper limit on the number of concurrent queries. This is the initial technique you might use when you do not have extensive information about memory usage for your workload. This setting can be specified separately for each dynamic resource pool.

You can combine this setting with the memory-based approach described in [Memory Limits and Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_memory). If either the maximum number of or the expected memory usage of the concurrent queries is exceeded, subsequent queries are queued until the concurrent workload falls below the threshold again.

See [Managing Dynamic Resource Pools](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_resource_pools.html#concept_xkk_l1d_wr) for information about all these dynamic resource pool settings, how to use them together, and how to divide different parts of your workload among different pools.

## Memory Limits and Admission Control

Each dynamic resource pool can have an upper limit on the cluster-wide memory used by queries executing in that pool. This is the technique to use once you have a stable workload with well-understood memory requirements.

Always specify the **Default Query Memory Limit** for the expected maximum amount of RAM that a query might require on each host, which is equivalent to setting the MEM\_LIMIT query option for every query run in that pool. That value affects the execution of each query, preventing it from overallocating memory on each host, and potentially activating the spill-to-disk mechanism or cancelling the query when necessary.

Optionally, specify the **Max Memory** setting, a cluster-wide limit that determines how many queries can be safely run concurrently, based on the upper memory limit per host multiplied by the number of Impala nodes in the cluster.

For example, consider the following scenario:

* The cluster is running impalad daemons on five DataNodes.
* A dynamic resource pool has **Max Memory** set to 100 GB.
* The **Default Query Memory Limit** for the pool is 10 GB. Therefore, any query running in this pool could use up to 50 GB of memory (default query memory limit \* number of Impala nodes).
* The maximum number of queries that Impala executes concurrently within this dynamic resource pool is two, which is the most that could be accomodated within the 100 GB **Max Memory** cluster-wide limit.
* There is no memory penalty if queries use less memory than the **Default Query Memory Limit** per-host setting or the **Max Memory** cluster-wide limit. These values are only used to estimate how many queries can be run concurrently within the resource constraints for the pool.

**Note:** If you specify **Max Memory** for an Impala dynamic resource pool, you must also specify the**Default Query Memory Limit**. **Max Memory** relies on the **Default Query Memory Limit** to produce a reliable estimate of overall memory consumption for a query.

You can combine the memory-based settings with the upper limit on concurrent queries described in[Concurrent Queries and Admission Control](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_admission.html#admission_concurrency). If either the maximum number of or the expected memory usage of the concurrent queries is exceeded, subsequent queries are queued until the concurrent workload falls below the threshold again.

See [Managing Dynamic Resource Pools](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_resource_pools.html#concept_xkk_l1d_wr) for information about all these dynamic resource pool settings, how to use them together, and how to divide different parts of your workload among different pools.

## How Impala Admission Control Relates to Other Resource Management Tools

The admission control feature is similar in some ways to the Cloudera Manager static partitioning feature, as well as the YARN resource management framework. These features can be used separately or together. This section describes some similarities and differences, to help you decide which combination of resource management features to use for Impala.

Admission control is a lightweight, decentralized system that is suitable for workloads consisting primarily of Impala queries and other SQL statements. It sets "soft" limits that smooth out Impala memory usage during times of heavy load, rather than taking an all-or-nothing approach that cancels jobs that are too resource-intensive.

Because the admission control system does not interact with other Hadoop workloads such as MapReduce jobs, you might use YARN with static service pools on CDH 5 clusters where resources are shared between Impala and other Hadoop components. This configuration is recommended when using Impala in a**multitenant** cluster. Devote a percentage of cluster resources to Impala, and allocate another percentage for MapReduce and other batch-style workloads. Let admission control handle the concurrency and memory usage for the Impala work within the cluster, and let YARN manage the work for other components within the cluster. In this scenario, Impala's resources are not managed by YARN.

The Impala admission control feature uses the same configuration mechanism as the YARN resource manager to map users to pools and authenticate them.

Although the Impala admission control feature uses a fair-scheduler.xml configuration file behind the scenes, this file does not depend on which scheduler is used for YARN. You still use this file, and Cloudera Manager can generate it for you, even when YARN is using the capacity scheduler.

## How Impala Schedules and Enforces Limits on Concurrent Queries

The admission control system is decentralized, embedded in each Impala daemon and communicating through the statestore mechanism. Although the limits you set for memory usage and number of concurrent queries apply cluster-wide, each Impala daemon makes its own decisions about whether to allow each query to run immediately or to queue it for a less-busy time. These decisions are fast, meaning the admission control mechanism is low-overhead, but might be imprecise during times of heavy load across many coordinators. There could be times when the more queries were queued (in aggregate across the cluster) than the specified limit, or when number of admitted queries exceeds the expected number. Thus, you typically err on the high side for the size of the queue, because there is not a big penalty for having a large number of queued queries; and you typically err on the low side for configuring memory resources, to leave some headroom in case more queries are admitted than expected, without running out of memory and being cancelled as a result.

To avoid a large backlog of queued requests, you can set an upper limit on the size of the queue for queries that are queued. When the number of queued queries exceeds this limit, further queries are cancelled rather than being queued. You can also configure a timeout period per pool, after which queued queries are cancelled, to avoid indefinite waits. If a cluster reaches this state where queries are cancelled due to too many concurrent requests or long waits for query execution to begin, that is a signal for an administrator to take action, either by provisioning more resources, scheduling work on the cluster to smooth out the load, or by doing [Impala performance tuning](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_performance.html#performance) to enable higher throughput.

## How Admission Control works with Impala Clients (JDBC, ODBC, HiveServer2)

Most aspects of admission control work transparently with client interfaces such as JDBC and ODBC:

* If a SQL statement is put into a queue rather than running immediately, the API call blocks until the statement is dequeued and begins execution. At that point, the client program can request to fetch results, which might also block until results become available.
* If a SQL statement is cancelled because it has been queued for too long or because it exceeded the memory limit during execution, the error is returned to the client program with a descriptive error message.

In Impala 2.0 and higher, you can submit a SQL SET statement from the client application to change theREQUEST\_POOL query option. This option lets you submit queries to different resource pools, as described in[REQUEST\_POOL Query Option](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_request_pool.html#request_pool).

At any time, the set of queued queries could include queries submitted through multiple different Impala daemon hosts. All the queries submitted through a particular host will be executed in order, so a CREATE TABLE followed by an INSERT on the same table would succeed. Queries submitted through different hosts are not guaranteed to be executed in the order they were received. Therefore, if you are using load-balancing or other round-robin scheduling where different statements are submitted through different hosts, set up all table structures ahead of time so that the statements controlled by the queuing system are primarily queries, where order is not significant. Or, if a sequence of statements needs to happen in strict order (such as an INSERT followed by a SELECT), submit all those statements through a single session, while connected to the same Impala daemon host.

Admission control has the following limitations or special behavior when used with JDBC or ODBC applications:

* The other resource-related query options, RESERVATION\_REQUEST\_TIMEOUT and V\_CPU\_CORES, are no longer used. Those query options only applied to using Impala with Llama, which is no longer supported.

## SQL and Schema Considerations for Admission Control

When queries complete quickly and are tuned for optimal memory usage, there is less chance of performance or capacity problems during times of heavy load. Before setting up admission control, tune your Impala queries to ensure that the query plans are efficient and the memory estimates are accurate. Understanding the nature of your workload, and which queries are the most resource-intensive, helps you to plan how to divide the queries into different pools and decide what limits to define for each pool.

For large tables, especially those involved in join queries, keep their statistics up to date after loading substantial amounts of new data or adding new partitions. Use the COMPUTE STATS statement for unpartitioned tables, and COMPUTE INCREMENTAL STATS for partitioned tables.

When you use dynamic resource pools with a **Max Memory** setting enabled, you typically override the memory estimates that Impala makes based on the statistics from the COMPUTE STATS statement. You either set the MEM\_LIMIT query option within a particular session to set an upper memory limit for queries within that session, or a default MEM\_LIMIT setting for all queries processed by the impalad instance, or a defaultMEM\_LIMIT setting for all queries assigned to a particular dynamic resource pool. By designating a consistent memory limit for a set of similar queries that use the same resource pool, you avoid unnecessary query queuing or out-of-memory conditions that can arise during high-concurrency workloads when memory estimates for some queries are inaccurate.

Follow other steps from [Tuning Impala for Performance](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_performance.html#performance) to tune your queries.

## Configuring Admission Control

The configuration options for admission control range from the simple (a single resource pool with a single set of options) to the complex (multiple resource pools with different options, each pool handling queries for a different set of users and groups). Cloudera recommends configuring the settings through the Cloudera Manager user interface.

**Important:** Although the following options are still present in the Cloudera Manager interface under the**Admission Control** configuration settings dialog, Cloudera recommends you not use them in CDH 5.7 / Impala 2.5 and higher. These settings only apply if you enable admission control but leave dynamic resource pools disabled. In CDH 5.7 / Impala 2.5 and higher, prefer to set up dynamic resource pools and customize the settings for each pool, as described in [Creating an Impala Dynamic Resource Pool](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_resource_pools.html#concept_xkk_l1d_wr__section_p15_mhn_2v) and[Editing Dynamic Resource Pools](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_resource_pools.html#concept_xkk_l1d_wr__section_gph_tnk_lm).

### Impala Service Flags for Admission Control (Advanced)

The following Impala configuration options let you adjust the settings of the admission control feature. When supplying the options on the impalad command line, prepend the option name with --.

**queue\_wait\_timeout\_ms**

**Purpose:** Maximum amount of time (in milliseconds) that a request waits to be admitted before timing out.

**Type:** int64

**Default:** 60000

**default\_pool\_max\_requests**

**Purpose:** Maximum number of concurrent outstanding requests allowed to run before incoming requests are queued. Because this limit applies cluster-wide, but each Impala node makes independent decisions to run queries immediately or queue them, it is a soft limit; the overall number of concurrent queries might be slightly higher during times of heavy load. A negative value indicates no limit. Ignored iffair\_scheduler\_config\_path and llama\_site\_path are set.

**Type:** int64

**Default:** -1, meaning unlimited (prior to CDH 5.7 / Impala 2.5, the default was 200)

**default\_pool\_max\_queued**

**Purpose:** Maximum number of requests allowed to be queued before rejecting requests. Because this limit applies cluster-wide, but each Impala node makes independent decisions to run queries immediately or queue them, it is a soft limit; the overall number of queued queries might be slightly higher during times of heavy load. A negative value or 0 indicates requests are always rejected once the maximum concurrent requests are executing. Ignored if fair\_scheduler\_config\_path and llama\_site\_path are set.

**Type:** int64

**Default:** unlimited

**default\_pool\_mem\_limit**

**Purpose:** Maximum amount of memory (across the entire cluster) that all outstanding requests in this pool can use before new requests to this pool are queued. Specified in bytes, megabytes, or gigabytes by a number followed by the suffix b (optional), m, or g, either uppercase or lowercase. You can specify floating-point values for megabytes and gigabytes, to represent fractional numbers such as 1.5. You can also specify it as a percentage of the physical memory by specifying the suffix %. 0 or no setting indicates no limit. Defaults to bytes if no unit is given. Because this limit applies cluster-wide, but each Impala node makes independent decisions to run queries immediately or queue them, it is a soft limit; the overall memory used by concurrent queries might be slightly higher during times of heavy load. Ignored iffair\_scheduler\_config\_path and llama\_site\_path are set.**Note:** Impala relies on the statistics produced by the COMPUTE STATS statement to estimate memory usage for each query. See [COMPUTE STATS Statement](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_compute_stats.html#compute_stats) for guidelines about how and when to use this statement.

**Type:** string

**Default:** "" (empty string, meaning unlimited)

**disable\_admission\_control**

**Purpose:** Turns off the admission control feature entirely, regardless of other configuration option settings.

**Type:** Boolean

**Default:** false

**disable\_pool\_max\_requests**

**Purpose:** Disables all per-pool limits on the maximum number of running requests.

**Type:** Boolean

**Default:** false

**disable\_pool\_mem\_limits**

**Purpose:** Disables all per-pool mem limits.

**Type:** Boolean

**Default:** false

**fair\_scheduler\_allocation\_path**

**Purpose:** Path to the fair scheduler allocation file (fair-scheduler.xml).

**Type:** string

**Default:** "" (empty string)

**Usage notes:** Admission control only uses a small subset of the settings that can go in this file, as described below. For details about all the Fair Scheduler configuration settings, see the [Apache wiki](http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html#Configuration).

**llama\_site\_path**

**Purpose:** Path to the configuration file used by admission control (llama-site.xml). If set,fair\_scheduler\_allocation\_path must also be set.

**Type:** string

**Default:** "" (empty string)

**Usage notes:** Admission control only uses a few of the settings that can go in this file, as described below.

### Configuring Admission Control Using Cloudera Manager

In Cloudera Manager, you can configure pools to manage queued Impala queries, and the options for the limit on number of concurrent queries and how to handle queries that exceed the limit. For details, see[Managing Resources with Cloudera Manager](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_managing_resources.html).

### Configuring Admission Control Using the Command Line

If you do not use Cloudera Manager, you use a combination of startup options for the Impala daemon, and optionally editing or manually constructing the configuration files fair-scheduler.xml and llama-site.xml.

For a straightforward configuration using a single resource pool named default, you can specify configuration options on the command line and skip the fair-scheduler.xml and llama-site.xmlconfiguration files.

For an advanced configuration with multiple resource pools using different settings, set up the fair-scheduler.xml and llama-site.xml configuration files manually. Provide the paths to each one using theimpalad command-line options, --fair\_scheduler\_allocation\_path and --llama\_site\_path respectively.

The Impala admission control feature only uses the Fair Scheduler configuration settings to determine how to map users and groups to different resource pools. For example, you might set up different resource pools with separate memory limits, and maximum number of concurrent and queued queries, for different categories of users within your organization. For details about all the Fair Scheduler configuration settings, see the [Apache wiki](http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html#Configuration).

The Impala admission control feature only uses a small subset of possible settings from the llama-site.xml configuration file:

llama.am.throttling.maximum.placed.reservations.queue\_name

llama.am.throttling.maximum.queued.reservations.queue\_name

impala.admission-control.pool-default-query-options.queue\_name

impala.admission-control.pool-queue-timeout-ms.queue\_name

The impala.admission-control.pool-queue-timeout-ms setting specifies the timeout value for this pool, in milliseconds. Theimpala.admission-control.pool-default-query-options settings designates the default query options for all queries that run in this pool. Its argument value is a comma-delimited string of 'key=value' pairs, for example,'key1=val1,key2=val2'. For example, this is where you might set a default memory limit for all queries in the pool, using an argument such as MEM\_LIMIT=5G.

The impala.admission-control.\* configuration settings are available in CDH 5.7 / Impala 2.5 and higher.

### Examples of Admission Control Configurations

#### Example Admission Control Configurations Using Cloudera Manager

For full instructions about configuring dynamic resource pools through Cloudera Manager, see [Dynamic Resource Pools](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_resource_pools.html#xd_583c10bfdbd326ba--43d5fd93-1410993f8c2--7ff2).

#### Example Admission Control Configurations Using Configuration Files

For clusters not managed by Cloudera Manager, here are sample fair-scheduler.xml and llama-site.xml files that define resource pools root.default, root.development, and root.production. These sample files are stripped down: in a real deployment they might contain other settings for use with various aspects of the YARN component. The settings shown here are the significant ones for the Impala admission control feature.

**fair-scheduler.xml:**

Although Impala does not use the vcores value, you must still specify it to satisfy YARN requirements for the file contents.

Each <aclSubmitApps> tag (other than the one for root) contains a comma-separated list of users, then a space, then a comma-separated list of groups; these are the users and groups allowed to submit Impala statements to the corresponding resource pool.

If you leave the <aclSubmitApps> element empty for a pool, nobody can submit directly to that pool; child pools can specify their own <aclSubmitApps> values to authorize users and groups to submit to those pools.

<allocations>

<queue name="root">

<aclSubmitApps> </aclSubmitApps>

<queue name="default">

<maxResources>50000 mb, 0 vcores</maxResources>

<aclSubmitApps>\*</aclSubmitApps>

</queue>

<queue name="development">

<maxResources>200000 mb, 0 vcores</maxResources>

<aclSubmitApps>user1,user2 dev,ops,admin</aclSubmitApps>

</queue>

<queue name="production">

<maxResources>1000000 mb, 0 vcores</maxResources>

<aclSubmitApps> ops,admin</aclSubmitApps>

</queue>

</queue>

<queuePlacementPolicy>

<rule name="specified" create="false"/>

<rule name="default" />

</queuePlacementPolicy>

</allocations>

**llama-site.xml:**

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<property>

<name>llama.am.throttling.maximum.placed.reservations.root.default</name>

<value>10</value>

</property>

<property>

<name>llama.am.throttling.maximum.queued.reservations.root.default</name>

<value>50</value>

</property>

<property>

<name>impala.admission-control.pool-default-query-options.root.default</name>

<value>mem\_limit=128m,query\_timeout\_s=20,max\_io\_buffers=10</value>

</property>

<property>

<name>impala.admission-control.pool-queue-timeout-ms.root.default</name>

<value>30000</value>

</property>

<property>

<name>llama.am.throttling.maximum.placed.reservations.root.development</name>

<value>50</value>

</property>

<property>

<name>llama.am.throttling.maximum.queued.reservations.root.development</name>

<value>100</value>

</property>

<property>

<name>impala.admission-control.pool-default-query-options.root.development</name>

<value>mem\_limit=256m,query\_timeout\_s=30,max\_io\_buffers=10</value>

</property>

<property>

<name>impala.admission-control.pool-queue-timeout-ms.root.development</name>

<value>15000</value>

</property>

<property>

<name>llama.am.throttling.maximum.placed.reservations.root.production</name>

<value>100</value>

</property>

<property>

<name>llama.am.throttling.maximum.queued.reservations.root.production</name>

<value>200</value>

</property>

<!--

Default query options for the 'root.production' pool.

THIS IS A NEW PARAMETER in CDH 5.7 / Impala 2.5.

Note that the MEM\_LIMIT query option still shows up in here even though it is a

separate box in the UI. We do that because it is the most important query option

that people will need (everything else is somewhat advanced).

MEM\_LIMIT takes a per-node memory limit which is specified using one of the following:

- '<int>[bB]?' -> bytes (default if no unit given)

- '<float>[mM(bB)]' -> megabytes

- '<float>[gG(bB)]' -> in gigabytes

E.g. 'MEM\_LIMIT=12345' (no unit) means 12345 bytes, and you can append m or g

to specify megabytes or gigabytes, though that is not required.

-->

<property>

<name>impala.admission-control.pool-default-query-options.root.production</name>

<value>mem\_limit=386m,query\_timeout\_s=30,max\_io\_buffers=10</value>

</property>

<!--

Default queue timeout (ms) for the pool 'root.production'.

If this isn’t set, the process-wide flag is used.

THIS IS A NEW PARAMETER in CDH 5.7 / Impala 2.5.

-->

<property>

<name>impala.admission-control.pool-queue-timeout-ms.root.production</name>

<value>30000</value>

</property>

</configuration>

## Guidelines for Using Admission Control

To see how admission control works for particular queries, examine the profile output for the query. This information is available through the PROFILE statement in impala-shell immediately after running a query in the shell, on the **queries** page of the Impala debug web UI, or in the Impala log file (basic information at log level 1, more detailed information at log level 2). The profile output contains details about the admission decision, such as whether the query was queued or not and which resource pool it was assigned to. It also includes the estimated and actual memory usage for the query, so you can fine-tune the configuration for the memory limits of the resource pools.

Where practical, use Cloudera Manager to configure the admission control parameters. The Cloudera Manager GUI is much simpler than editing the configuration files directly.

Remember that the limits imposed by admission control are "soft" limits. The decentralized nature of this mechanism means that each Impala node makes its own decisions about whether to allow queries to run immediately or to queue them. These decisions rely on information passed back and forth between nodes by the statestore service. If a sudden surge in requests causes more queries than anticipated to run concurrently, then throughput could decrease due to queries spilling to disk or contending for resources; or queries could be cancelled if they exceed the MEM\_LIMIT setting while running.

In impala-shell, you can also specify which resource pool to direct queries to by setting the REQUEST\_POOLquery option.

The statements affected by the admission control feature are primarily queries, but also include statements that write data such as INSERT and CREATE TABLE AS SELECT. Most write operations in Impala are not resource-intensive, but inserting into a Parquet table can require substantial memory due to buffering intermediate data before writing out each Parquet data block. See [Loading Data into Parquet Tables](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_parquet.html#parquet_etl) for instructions about inserting data efficiently into Parquet tables.

Although admission control does not scrutinize memory usage for other kinds of DDL statements, if a query is queued due to a limit on concurrent queries or memory usage, subsequent statements in the same session are also queued so that they are processed in the correct order:

-- This query could be queued to avoid out-of-memory at times of heavy load.

select \* from huge\_table join enormous\_table using (id);

-- If so, this subsequent statement in the same session is also queued

-- until the previous statement completes.

drop table huge\_table;

If you set up different resource pools for different users and groups, consider reusing any classifications you developed for use with Sentry security. See [Enabling Sentry Authorization for Impala](http://www.cloudera.com/documentation/enterprise/latest/topics/impala_authorization.html#authorization) for details.

For details about all the Fair Scheduler configuration settings, see [Fair Scheduler Configuration](https://archive.cloudera.com/cdh5/cdh/5/hadoop/hadoop-yarn/hadoop-yarn-site/FairScheduler.html#Configuration), in particular the tags such as <queue> and <aclSubmitApps> to map users and groups to particular resource pools (queues).