Topic 3 – Hadoop Distributed File System (HDFS)

**HDFS**

* File system that are hundreds of MB, GB, PB,
* Streaming Data Access – one writer & many readers,
* Throughput – time to read whole file is more important than latency to read the first record
* Available to run on commodity (cheap), unreliable hardware with high failure in each node. HDFS resilient against failure

**Blocks**

* Stores data in fixed-size blocks

A screenshot of a computer

Description automatically generated

* HDFS uses large blocks to minimize overhead time for HDD

**Benefits**

* Blocks on different computer increase capacity
* Blocks can be process in parallel
* Blocks can be replicated for redundancy

**Distributed File System**

* One computer runs the Namenode, holding info about the file system (directory tree)
* Other computer run a Datanode of HDFS and manage the block on that computer
* For small installation, a computer may run both Namenode and Datanode

A diagram of a computer network

Description automatically generated

**Under the Hood**

1. Internal details
   1. Namenode Operation
      1. Keeps in-core view of HDFS for fast performance (dynamic snapshot of whole FS)
      2. When computer shuts down, dynamic data is lost. After restarting, Namenode loads an image of itself from previous checkpoint (initially empty) and combine with entries in Edit Log
      3. All edits are written to Edit Log file before the changes are made to in-core structure. Only changes to file structure is recorded, not changes to file data
      4. As Edit Log becomes long, all entries are copied and written to an updated checkpoint of *fsimage* (new checkpoint)
      5. If computer running Namenode fails:
         1. Default is to start Namenode on a new server with *fsimage and* Edit Log from old Namenode
         2. Have a stand-by secondary backup copying all info from primary in real time
   2. Duplication of Nodes
      1. HDFS typically replicates by a factor of 3 (adjust based on needs) for increase reliability
      2. Parallel write operation when new data is added
      3. If one copy fails, a new copy will be created on a different Datanode

**External View**

1. CLI
   1. Pseudo-Distributed mode (run on 1 computer)
      1. fs.default.name = hdfs://localhost/  
         This setting is a URI indicating where to find the filesystem. This value indicates that it is on  
         your computer (using port 8020).
      2. dfs.replication = 1  
         This means to override the default 3-copies, where each block is replicated for redundancy.  
         On your one computer, you just one one copy of each block
   2. Cluster mode (run across multiple machines)
   3. Java Interface – HDFS has a FileSystem class where Java code is portable across file systems

**Anatony of a Read**

1. The client opens the file by calling open() on the FileSystem object.  
2. The FileSystem calls the namenode (using RPC) to determine the locations of  
the first few blocks of the file.  
3. For info of each block, the namenode returns the address of the 'closest'  
datanode holding a copy of that block.  
4. The FileSystem returns a FSDataInputStream, from which the client can read  
data.  
5. The client then calls read() on this stream. That datanode returns the next  
values to the client.  
6. When one block is exhausted, the FileSystem closes the connection to that  
datanode and opens a connection to the next datanode.

7. When the list of known blocks is exhausted, the namenode is asked for the  
next few blocks of the file.  
8. If, during the reading, an error is encountered, the FileSystem will record that  
this block is not working, and will then try the next closest datanode that has a  
duplicate of this block.  
● Because of this architecture, the network traffic is spread out.  
○ All clients need to talk to the namenode, but they only need a little information.  
○ The bulk of the traffic is with the datanodes, but these are spread out and are replicated

**Anatomy of Write**

● For this example, we will create a new file, write data to it, then close the file.  
1. The client calls create() on the FileSystem object.  
2. The FileSystem calls the namenode (using RPC) to create a new file in the  
HDFS directory tree (the in-core structure, while writing this information to the  
Edit Log). There are no blocks yet, but a new file is created.  
3. The FileSystem returns a FSDataOutputStream, to which the client can write  
data.  
4. As the client writes data, the FSDataOutputStream splits it into packets, which  
are written to the data queue

5. Packets are removed from the data queue and written to the HDFS as  
follows:  
a. The namenode is asked for new blocks to store the data (typically 3 for clusters, 1 for local).  
b. The namenode picks three 'best' datanodes, which will form a pipeline.  
c. The packet is sent to the first of the datanodes, which stores the data in a new block, but then  
sends the packet to the second of the datanodes.  
d. The second datanode does the same, and the third datanode also stores the data.  
e. The client's program meanwhile moves the packet to the ack queue, where it waits for  
acknowledgements.  
f. As each datanode successfully processes the packet, it sends an ack (acknowledge) signal  
back to the source.  
g. When the source receives all three acks, it then discards the packet.

6. If any of the datanodes fail while processing the packet, the following actions  
are performed:  
a. The pipeline is closed.  
b. Any packets in the ack queue are moved back to the data queue (they need to be sent again).  
c. The current block on any good datanode are given a new identity, which is communicated to  
the namenode (so that the namenode can identify the failed datanodes).  
d. The failed datanode is removed from the pipeline.  
e. The remainder of the data packets are sent to the two remaining datanodes.  
f. The namenode will realize that this block is under-replicated, so it will arrange for the block to  
be duplicated from one of the good datanodes.  
7. When the client is done, it calls close(). This then flushes all of the buffers  
and waits for the acks