Parallel and Distributed Storage

Virtual Node Partitioning

- Key idea: pretend there are several times (10x to 20x) as many **virtual nodes** as real nodes
  - Virtual nodes are mapped to real nodes
  - Tuples partitioned across virtual nodes using range-partitioning vector
    - Hash partitioning is also possible

- **Example:** There are 4 nodes: N1, N2, N3 and N4 in a parallel database system. The schema of the relation person is as follows:

Person(NID, Name, Thana, District, Age)

The minimum age is 1 and maximum is 110. Perform partitioning of person relation on age into 4 nodes using virtual partitioning method.

Consider 60 virtual nodes.

Virtual Node Partitioning

- **Example:** There are 4 nodes: N1, N2, N3 and N4 in a parallel database system. The schema of the relation person is as follows:

Person(NID, Name, Thana, District, Age)

The minimum age is 1 and maximum is 110. Perform partitioning of person relation on age into 4 nodes using virtual partitioning method.

Consider 60 virtual nodes.

Step 1: Define partition vector for 60 virtual nodes.

Step 2: Partition person relation into virtual nodes.

Step 3: Map 60 virtual nodes to 4 real nodes using **Round-robin method.** Virtual node *i* mapped to real node (*i* mod *4*)+1

Virtual Node Partitioning

- **Mapping table**: mapping table *virtual_to_real_map*[] tracks which virtual node is on which real node

- Allows skew to be handled by moving virtual nodes from more loaded nodes to less loaded nodes

- Both data distribution skew and execution skew can be handled

Question 7-1: Explain how data distribution skew and execution skew can be handled using virtual node partitioning.

Dynamic Repartitioning

- Virtual node approach with a fixed partitioning vector cannot handle significant changes in data distribution over time

- Complete repartitioning is expensive and intrusive

- **Dynamic repartitioning** can be done incrementally using virtual node scheme
  - Virtual nodes that become too big can be split
    - Much like B+-tree node splits
  - Some virtual nodes can be moved from a heavily loaded node to a less loaded node

- Virtual nodes in such a scheme are often called **tablets**

Dynamic Repartitioning

- Virtual nodes in such a scheme are often called **tablets**
- Example of initial **partition table** and partition table after a split of tablet 6 and move of tablet 1

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node1
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
MaxDate	Tablet6	Node1

Tablet move

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node0
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
2018-01-01	Tablet6	Node1
MaxDate	Tablet7	Node1

Tablet split

Routing of Queries

- Partition table typically stored at a **master** node, and at multiple routers
- Queries are sent first to **routers**, which forward them to appropriate node
- **Consistent hashing** is an alternative fully-distributed scheme
  - without any master nodes, works in a completely peer-to-peer fashion

(Self study)

- **Distributed hash tables** are based on consistent hashing
  - work without master nodes or routers; each peer-node stores data and performs routing

(self study)

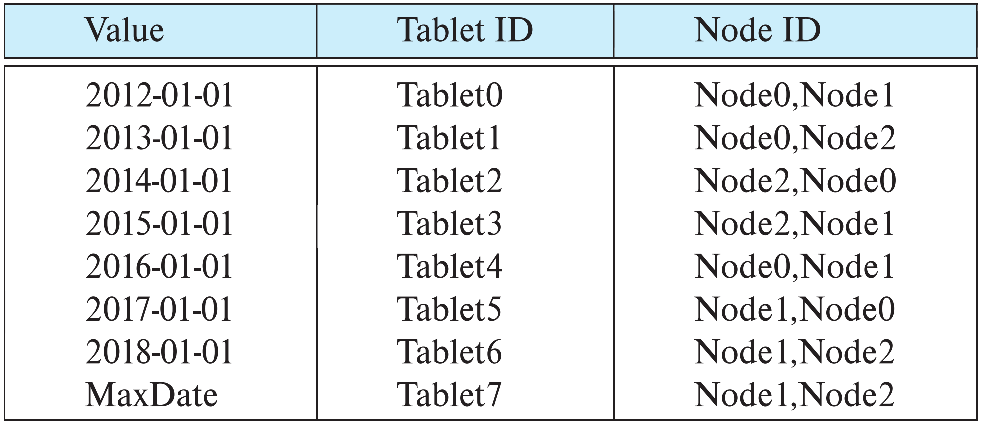
Routing of Queries

- Most parallel data storage systems store the partition table at a master node. However, to support a large number of requests each second, the partition table is usually replicated, either to all client nodes that access data or to multiple routers.
- Routers accept read/write requests from clients and forward the requests to the appropriate real node containing the tablet/virtual nodes based on the key values specified in the request.

Question 7-2: Explain how a query is executed in parallel storage system with dynamic partitioned storage of a relation. The query and the partitioning is on the same attribute.

Replication

- Goal: **availability** despite failures
- Data replicated at 2, often 3 nodes
- Unit of replication typically a partition (tablet)
- Requests for data at failed node automatically routed to a replica
- Partition table with each tablet replicated at two nodes



Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0,Node1
2013-01-01	Tablet1	Node0,Node2
2014-01-01	Tablet2	Node2,Node0
2015-01-01	Tablet3	Node2,Node1
2016-01-01	Tablet4	Node0,Node1
2017-01-01	Tablet5	Node1,Node0
2018-01-01	Tablet6	Node1,Node2
MaxDate	Tablet7	Node1,Node2

Basics: Data Replication

- Location of replicas
  - **Replication within a data center**
    - Handles machine failures
    - Reduces latency if copy available locally on a machine
    - Replication within/across racks
  - **Replication across data centers**
    - Handles data center failures (power, fire, earthquake, ..), and network partitioning of an entire data center
    - Provides lower latency for end users if copy is available on nearby data center

Updates and Consistency of Replicas

- Replicas must be kept consistent on update
  - Despite failures resulting in different replicas having different values (temporarily), reads must get the latest value.
  - Special concurrency control and atomic commit mechanisms to ensure consistency

- **Master replica** (**primary copy**) scheme
  - All updates are sent to master, and then replicated to other nodes
  - Reads are performed at master
  - But what if master fails? Who takes over? How do other nodes know who is the new master?

Question 8-1: Write advantages and disadvantages of replication

Protocols to Update Replicas

- *Two-phase commit*
  - Assumes all replicas are available
- *Persistent messaging*
  - Updates are sent as messages with guaranteed delivery
  - Replicas are updated asynchronously (after original transaction commits)
    - **Eventual consistency**
  - Can lead to inconsistency on reads from replicas
- *Consensus protocols*
  - Protocol followed by a set of replicas to agree on what updates to perform in what order
  - Can work even without a designated master

Distributed File Systems

- The goal of first-generation distributed file systems was to allow client machines to access files stored on one or more file servers.

- In contrast, later-generation distributed file systems, which we focus on, address distribution of file blocks across a very large number of nodes.

- Such distributed file systems can store very large amounts of data and support very large numbers of concurrent clients.

- A landmark system in this context was the **Google File System (GFS),** developed in the early 2000s, which saw widespread use within Google.

- The open-source **Hadoop File System (HDFS)** is based on the GFS architecture and is now very widely used.

Distributed File Systems

- Distributed file systems are generally designed to efficiently store large files whose sizes range from tens of megabytes to hundreds of gigabytes or more.

- However, they are designed to store moderate numbers of such files, of the order of millions; they are typically not designed to stores billions of different files.

- In contrast, the parallel data storage systems we have seen earlier are designed to store very large numbers (billions or more) of data items, whose size can range from small (tens of bytes) to medium (a few megabytes).

Distributed File Systems

- As in parallel data storage systems, the data in a distributed file system are stored across a number of nodes.

- Since files can be much larger than data items in a data storage system, files are broken up into multiple **blocks.**

- The blocks of a single file can be partitioned across multiple machines.

- Further, each file block is replicated across multiple (typically three) machines, so that a machine failure does not result in the file becoming inaccessible.

Distributed File Systems

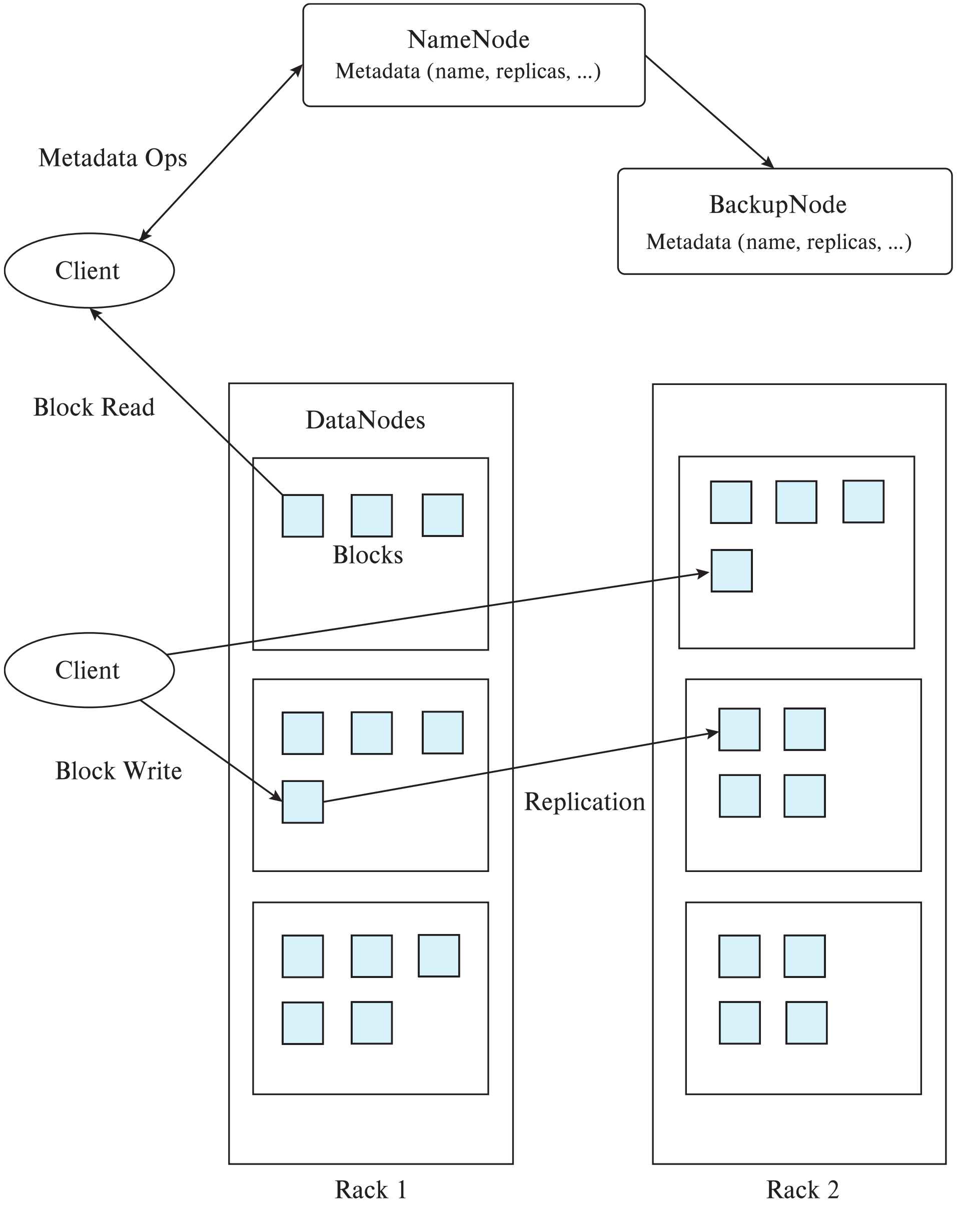
- File systems typically support two kinds of *metadata*:

1. A directory system, which allows a hierarchical organization of files into directories and subdirectories, and

2. A mapping from a file name to the sequence of identifiers of blocks that store the actual data in each file.

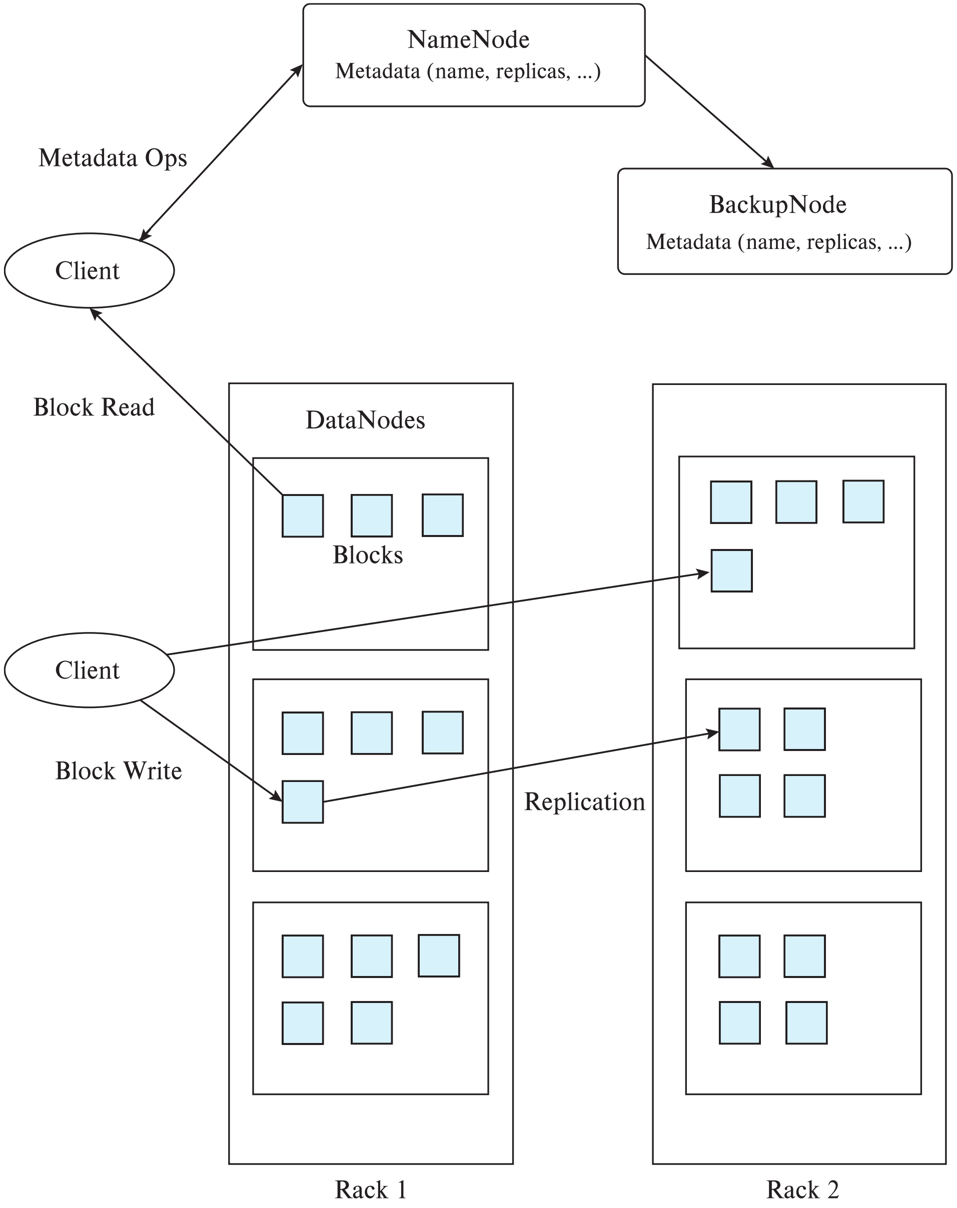
Hadoop File System (HDFS)

- Client: sends filename to NameNode
- NameNode
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
  - Returns list of BlockIDs along with locations of their replicas
- DataNode:
  - Maps a Block ID to a physical location on disk
  - Sends data back to client



Hadoop File System (HDFS)

Question 8-2: Using 64MB block size, how can you store a file named “YourId_HDFS” of size 10GB in Hadoop file system?



Hadoop Distributed File System

Hadoop Distributed File System (HDFS)

- Modeled after Google File System (GFS)
- Single Namespace for entire cluster
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple (e.g., 3) DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode

Limitations of GFS/HDFS

- Central master becomes bottleneck
  - Keep directory/inode information in memory to avoid IO
  - Memory size limits number of files
  - Colossus file system supports distributed master
    - With smaller (1MB) block size
- File system directory overheads per file
  - Not appropriate for storing very large number of objects
- File systems do not provide consistency guarantees
  - File systems cache blocks locally
  - Ideal for write-once and append only data
  - Can be used as underlying storage for a data storage system
    - E.g., BigTable uses GFS underneath

Sharding

- Divide data amongst many cheap databases (MySQL/PostgreSQL)
- Manage parallel access in the application
  - Partition tables map keys to nodes
  - Application decides where to route storage or lookup requests
- Scales well for both reads and writes
- Limitations
  - Not transparent
    - application needs to be partition-aware
    - AND application needs to deal with replication
  - (Not a true parallel database, since parallel queries and transactions spanning nodes are not supported)

Data Storage Systems vs. Databases

Distributed data storage implementations:

- May have limited support for relational model (no schema, or flexible schema)
- But usually do provide flexible schema and other features
  - Structured objects e.g. using JSON
  - Multiple versions of data items
- Often do not support referential integrity constraints
- Often provide no support or limited support for transactions
  - But some do!
- Provide only lowest layer of database

Data Representation

- In wide-column stores like BigTable, records may be vertically partitioned by attribute (*columnar storage*)
  - (record-identifier, attribute-name) forms a key
- Multiple attributes may be stored in one file (**column family**)
  - In BigTable records are sorted by key, ensuring all attributes of a logical record in that file are contiguous
    - Attributes can be fetched by a prefix/range query
  - Record-identifiers can be structured hierarchically to exploit sorting
    - E.g., url: www.cs.yale.edu/people/silberschatz.html
can be mapped to record identifier
edu.yale.cs.www/people/silberschatz.html
    - Now all records for cs.yale.edu would be contiguous, as would all records for yale.edu

Geographically Distributed Storage

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (close to user), governmental regulations
- Latency of replication across geographically distributed data centers much higher than within data center
  - Some key-value stores support **synchronous replication**
    - Must wait for replicas to be updated before committing an update
  - Others support **asynchronous replication**
    - update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
    - Must deal with small risk of data loss if data center fails.

Transactions in Key-Value Stores

- Most key-value stores don’t support full-fledged transactions
  - But treat each update as a transaction, to ensure integrity of internal data structure
- Some key-value stores allow multiple updates to one data item to be committed as a single transaction
- Without support for transactions, secondary indices cannot be maintained consistently
  - Some key-value stores do not support secondary indices at all
  - Some key-value stores support asynchronous maintenance of secondary indices
- Some key-value stores support ACID transactions across multiple data items along with two-phase commit across nodes
  - Google MegaStore and Spanner

Transactions in Key-Value Stores

- Some key-value stores support concurrency control via locking and snapshots
- Some support *atomic test-and-set* and *increment* on data items
  - Others do not support concurrency control
- Key-value stores implement recovery protocols based on logging to ensure durability
  - Log must be replicated, to ensure availability in spite of failures
- Distributed file systems are used to store log and data files in some key-value stores such as BigTable, HBase
  - But distributed file systems do not support (atomic) updates of files except for appends
  - LSM trees provide a nice way to index data without requiring updates of files
- Some systems use persistent messaging to manage logs

Querying and Performance Optimizations

- Many key-value stores do not provide a declarative query language
- Applications must manage joins, aggregates, etc on their own
- Some applications avoid computing joins at run-time by creating (what is in effect) materialized views
  - Application code maintains materialized views
  - E.g., If a user makes a post, the application may add a summary of the post to the data items representing all the friends of the user
- Many key-value stores allow related data items to be stored together
  - Related data items form an **entity-group**
  - e.g., user data item along with all posts of that user
  - Makes joining the related tuples very cheap
- Other functionality includes
  - Stored procedures executed at the nodes storing the data
  - Versioning of data, along with automated deletion of old versions