Chapter 21: Parallel and Distributed Storage

Database System Concepts, 7th Ed.

Introduction

- Parallel machines have become quite common and affordable
  - prices of microprocessors, memory and disks have dropped sharply
- Data storage needs are growing increasingly large
  - user data at web-scale
    - 100's of millions of users, petabytes of data
  - transaction data are collected and stored for analysis.
  - multimedia objects like images/videos
- Parallel storage system requirements
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing
  - Very high demands on **scalability** and **availability**

Parallel/Distribted Data Storage History

- 1980/1990s
  - Distributed database systems with tens of nodes
- 2000s:
  - Distributed file systems with 1000s of nodes
    - Millions of Large objects (100’s of megabytes)
    - Web logs, images, videos, …
    - Typically create/append only
  - Distributed data storage systems with 1000s of nodes
    - Billions to trillions of smaller (kilobyte to megabyte) objects
    - Social media posts, email, online purchases, …
    - Inserts, updates, deletes
  - **Key-value stores**
- 2010s: Distributed database systems with 1000s of nodes

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning the relations on *multiple disks*, on *multiple* ***nodes*** (computers)
  - Our description focuses on parallelism across nodes
  - Same techniques can be used across disks on a node
- **Horizontal partitioning** – tuples of a relation are divided among many nodes such that some subset of tuple resides on each node.
  - Contrast with **vertical partitioning**, e.g. *r*(*A,B,C,D*) with primary key *A* into r1(*A,B*) and r2(*A,C,D*)
  - By default, the word partitioning refers to horizontal partitioning

I/O Parallelism

- Partitioning techniques (number of nodes = n):

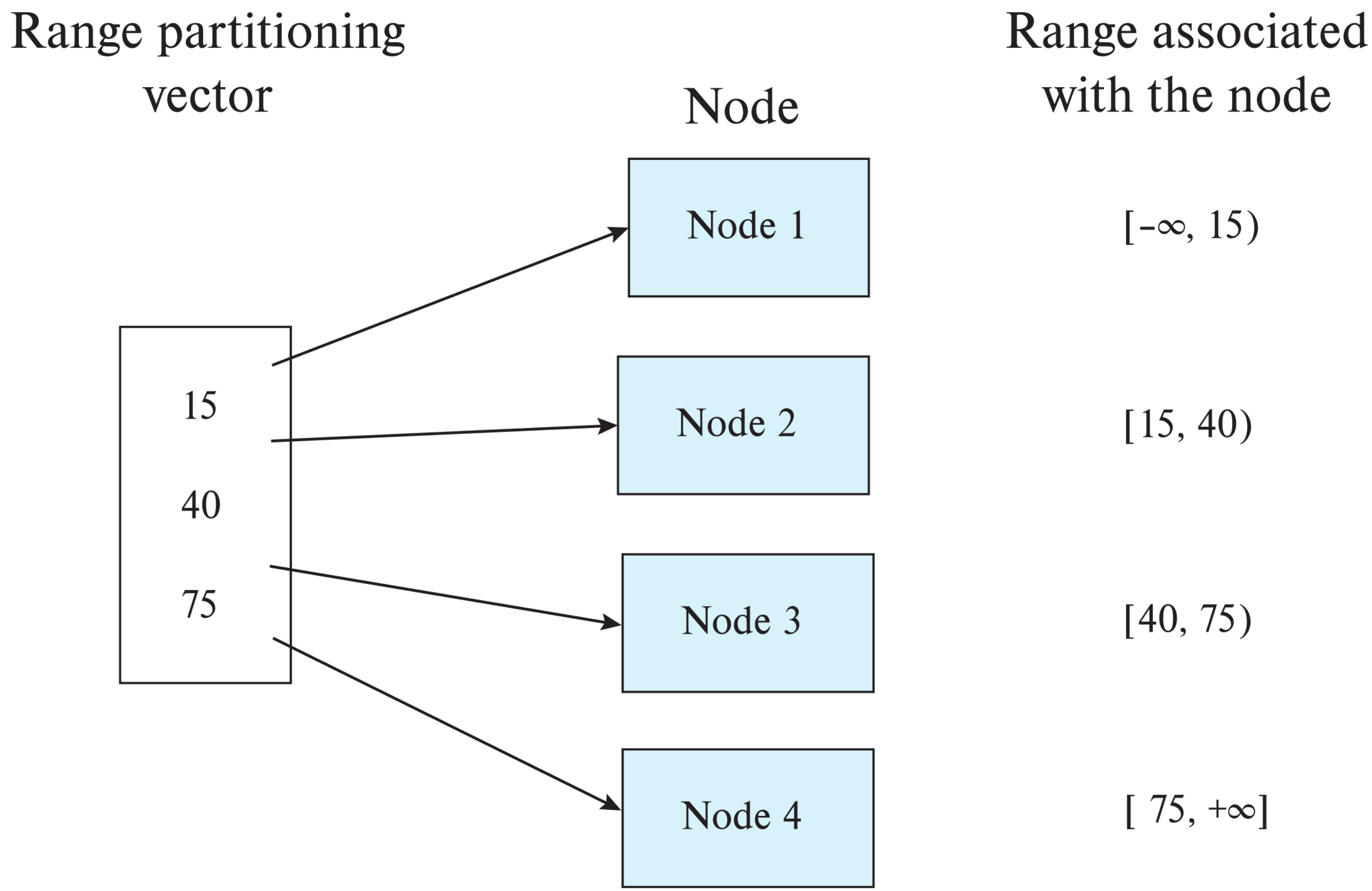
  Round-robin:

  Send the i^{th} tuple inserted in the relation to node i mod n.

  Hash partitioning:

  - Choose one or more attributes as the partitioning attributes.
  - Choose hash function h with range $0 \ldots n - 1$
  - Let i denote result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to node i.

Range Partitioning



I/O Parallelism (Cont.)

Partitioning techniques (cont.):

- **Range partitioning**:
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector [v_o, v_1, ..., $v_{n\text{-}2}$] is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v_{i+1}$ go to node $I + 1$. Tuples with $v < v_0$ go to node 0 and tuples with $v \geq v_{n-2}$ go to node n-1.

  E.g., with a partitioning vector [5,11]

    - a tuple with partitioning attribute value of 2 will go to node 0,
    - a tuple with value 8 will go to node 1, while
    - a tuple with value 20 will go to node2.

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation.
  2. Locating a tuple associatively – **point queries**.
     - E.g., *r.A* = 25.
  3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
     - E.g., $10 \leq r.A < 25$.
- Do above evaluation for each of
  - Round robin
  - Hash partitioning
  - Range partitioning

Comparison of Partitioning Techniques (Cont.)

Round robin:

- Best suited for sequential scan of entire relation on each query.
  - All nodes have almost an equal number of tuples; retrieval work is thus well balanced between nodes.
- All queries must be processed at all nodes

Hash partitioning:

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between nodes
- Good for point queries on partitioning attribute
  - Can lookup single node, leaving others available for answering other queries.
- Range queries inefficient, must be processed at all nodes

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

- Provides data clustering by partitioning attribute value.
  - Good for sequential access
  - Good for point queries on partitioning attribute: only one node needs to be accessed.
- For range queries on partitioning attribute, one to a few nodes may need to be accessed
  - Remaining nodes are available for other queries.
  - Good if result tuples are from one to a few blocks.
  - But if many blocks are to be fetched, they are still fetched from one to a few nodes, and potential parallelism in disk access is wasted
    - Example of **execution skew**.

Handling Small Relations

- Partitioning not useful for small relations which fit into a single disk block or a small number of disk blocks
  - Instead, assign the relation to a single node, or
  - Replicate relation at all nodes
- For medium sized relations, choose how many nodes to partition across based on size of relation
- Large relations typically partitioned across all available nodes.

Types of Skew

- **Data-distribution skew:** some nodes have many tuples, while others may have fewer tuples. Could occur due to
  - **Attribute-value skew**.
    - Some partitioning-attribute values appear in many tuples
    - All the tuples with the same value for the partitioning attribute end up in the same partition.
    - Can occur with range-partitioning and hash-partitioning.
  - **Partition skew**.
    - Imbalance, even without attribute –value skew
    - Badly chosen range-partition vector may assign too many tuples to some partitions and too few to others.
    - Less likely with hash-partitioning

Types of Skew (Cont.)

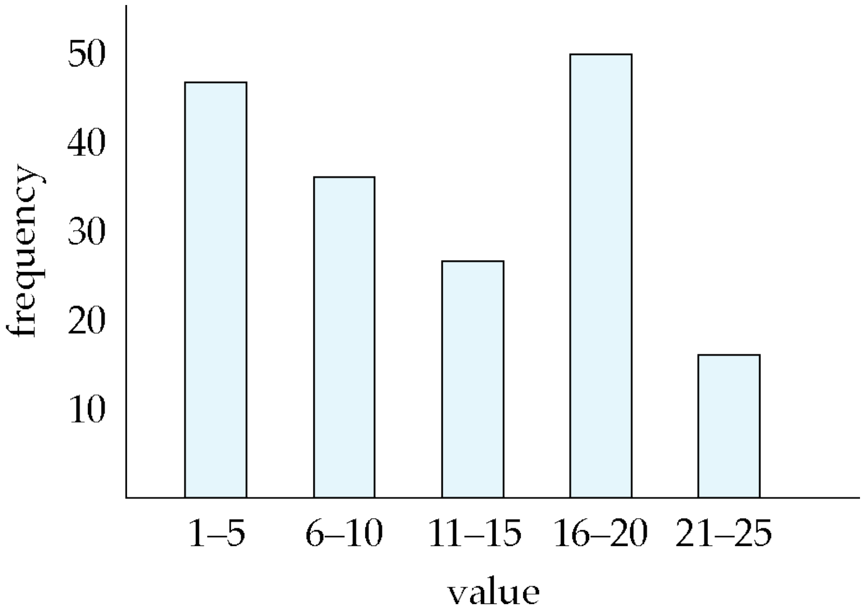
- Note that **execution skew** can occur even without data distribution skew
  - E.g. relation range-partitioned on date, and most queries access tuples with recent dates
- Data-distribution skew can be avoided with range-partitioning by creating **balanced range-partitioning vectors**
- We assume for now that partitioning is **static**, that is partitioning vector is created once and not changed
  - Any change requires **repartitioning**
  - **Dynamic partitioning** once allows partition vector to be changed in a continuous manner
    - More on this later

Handling Skew in Range-Partitioning

- To create a balanced partitioning vector
  - Sort the relation on the partitioning attribute.
  - Construct the partition vector by scanning the relation in sorted order as follows.
    - After every $1/n^{th}$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector.
  - *n* denotes the number of partitions to be constructed.
  - Imbalances can result if duplicates are present in partitioning attributes.
- To reduce cost
  - Partitioning vector can be created using a random sample of tuples
  - Alternatively histograms can be used to create the partitioning vector

Histograms

- Histogram on attribute *age* of relation *person*



- **Equi-width** histograms
- **Equi-depth** histograms
  - break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Assume uniform distribution within each range of the histogram
- Create partitioning vector for required number of partitions based on histogram

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
- Mapping of virtual nodes to real nodes
  - **Round-robin**: virtual node *i* mapped to real node (*i* mod *n*)+1
  - **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

Handling Skew Using Virtual Node Partitioning

- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions
  - Skewed virtual partitions tend to get spread across a number of nodes, so work gets distributed evenly!
- Virtual node approach also allows **elasticity of storage**
  - If relation size grows, more nodes can be added and virtual nodes moved to new nodes

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

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?
  - Details in Chapter 23

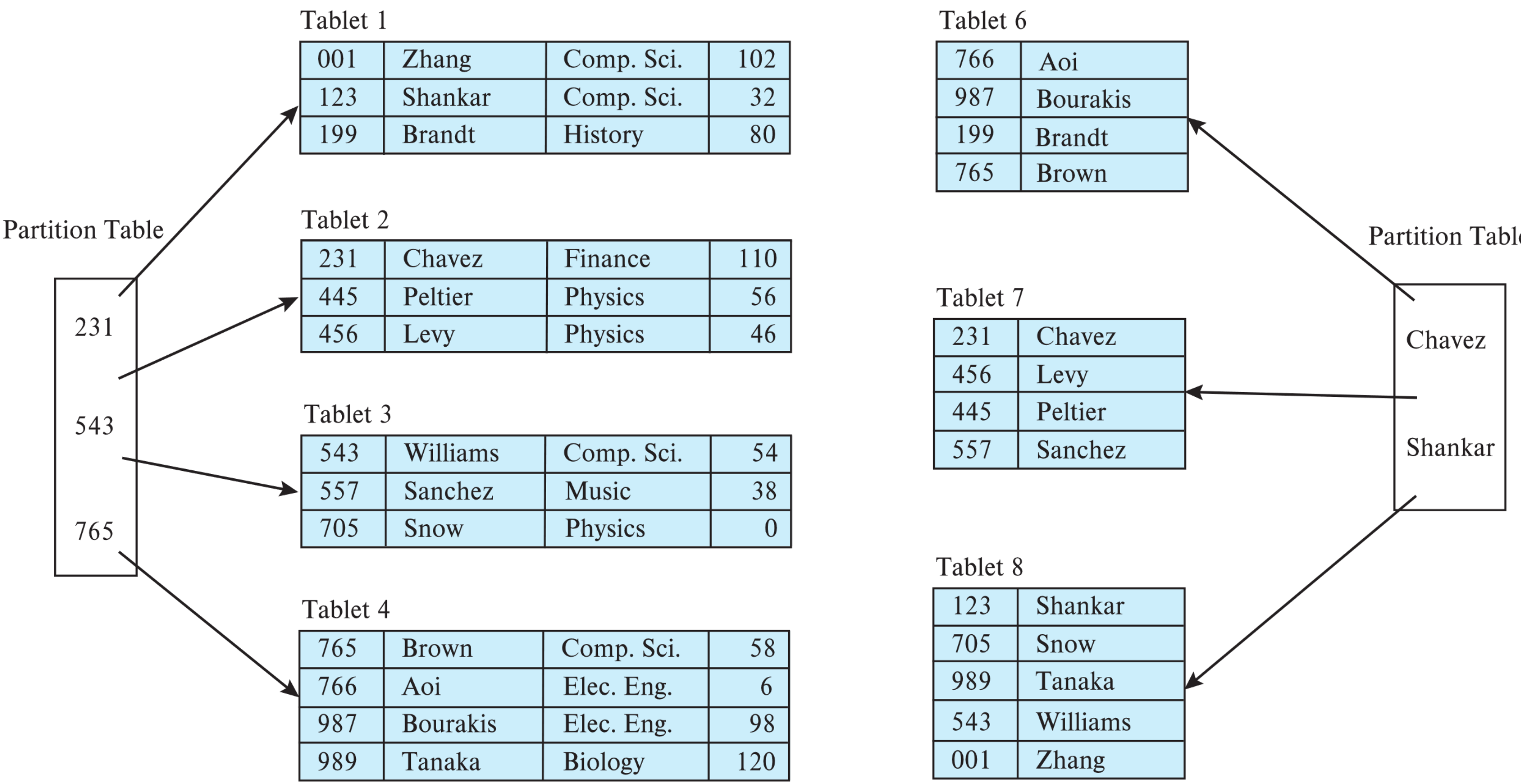
Protocols to Update Replicas

- *Two-phase commit*
  - Coming up in Chapter 23
  - 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

Parallel Indexing

- **Local index**
  - Index built only on local data
- **Global index**
  - Index built on all data, regardless of where it is stored
  - Index itself is usually partitioned across nodes
- **Global primary index**
  - Data partitioned on the index attribute
- **Global secondary index**
  - Data partitioned on the attribute other than the index attribute

Global Primary and Secondary Indices



(a) Primary index on ID

(b) Secondary index on name

Global Secondary Index

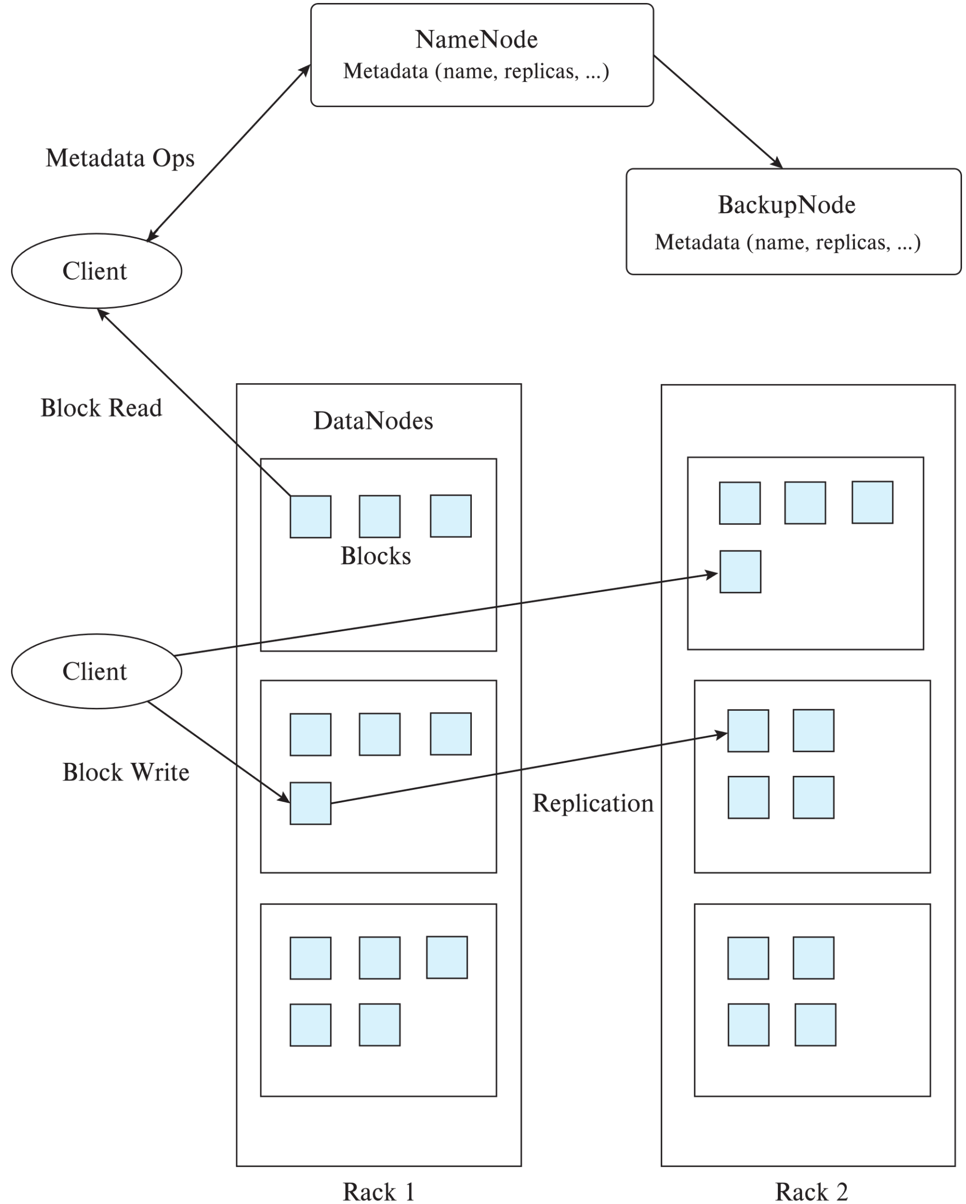
- Given relation r which is partitioned on K_p, to create global secondary index on attributes K_i,
  - create a relation
    - $r_i^s(K_i, K_p)$ if K_p is unique, otherwise
    - $r_i^s(K_i, K_p, K_u)$ where (K_p, K_u) is a key for r
  - Partition r_i^s on K_i
  - At each node containing a partition of r, create index on (K_p) if K_p is a key, otherwise create index on (K_p, K_u)
  - Update the relation r_i^s on any updates to r on attributes in r_i^s

Distributed File Systems

- Google File System (GFS)
- Hadoop File System (HDFS)
- And older ones like CODA
- And more recent ones such as Google Colossus
- Basic architecture:
  - Master: responsible for metadata
  - Chunk servers: responsible for reading and writing large chunks of data
  - Chunks replicated on 3 machines, master responsible for managing replicas
  - Replication is in GFS/HDFS is within a single data center

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 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 and append only data
  - Can be used as underlying storage for a data storage system
    - E.g. BigTable uses GFS underneath

Sharding

Sharding (recall from Chapter 10)

- 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)

Key Value Storage Systems

Recall from Chapter 10

- Key-value stores may store
  - **uninterpreted bytes**, with an associated key
    - E.g. Amazon S3, Amazon Dynamo
  - **Wide-column stores** (can have arbitrarily many attribute names) with associated key
      - Google BigTable, Apache Cassandra, Apache HBase, Amazon DynamoDB, Microsoft Azure Cloud store
      - Allows some operations (e.g. filtering) to execute on storage node
      - Google MegaStore and Spanner and Yahoo! PNUTS/Sherpa support relational schema
  - JSON
    - MongoDB, CouchDB (document model)
- **Document stores** store semi-structured data, typically JSON

Typical Data Storage Access API

- Basic API access:
  - get(key) -- Extract the value given a key
  - put(key, value) -- Create or update the value given its key
  - delete(key) -- Remove the key and its associated value
  - execute(key, operation, parameters) -- Invoke an operation to the value (given its key) which is a special data structure (e.g., List, Set, Map .... Etc.)
- Extensions to add range queries, version numbering, etc.

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

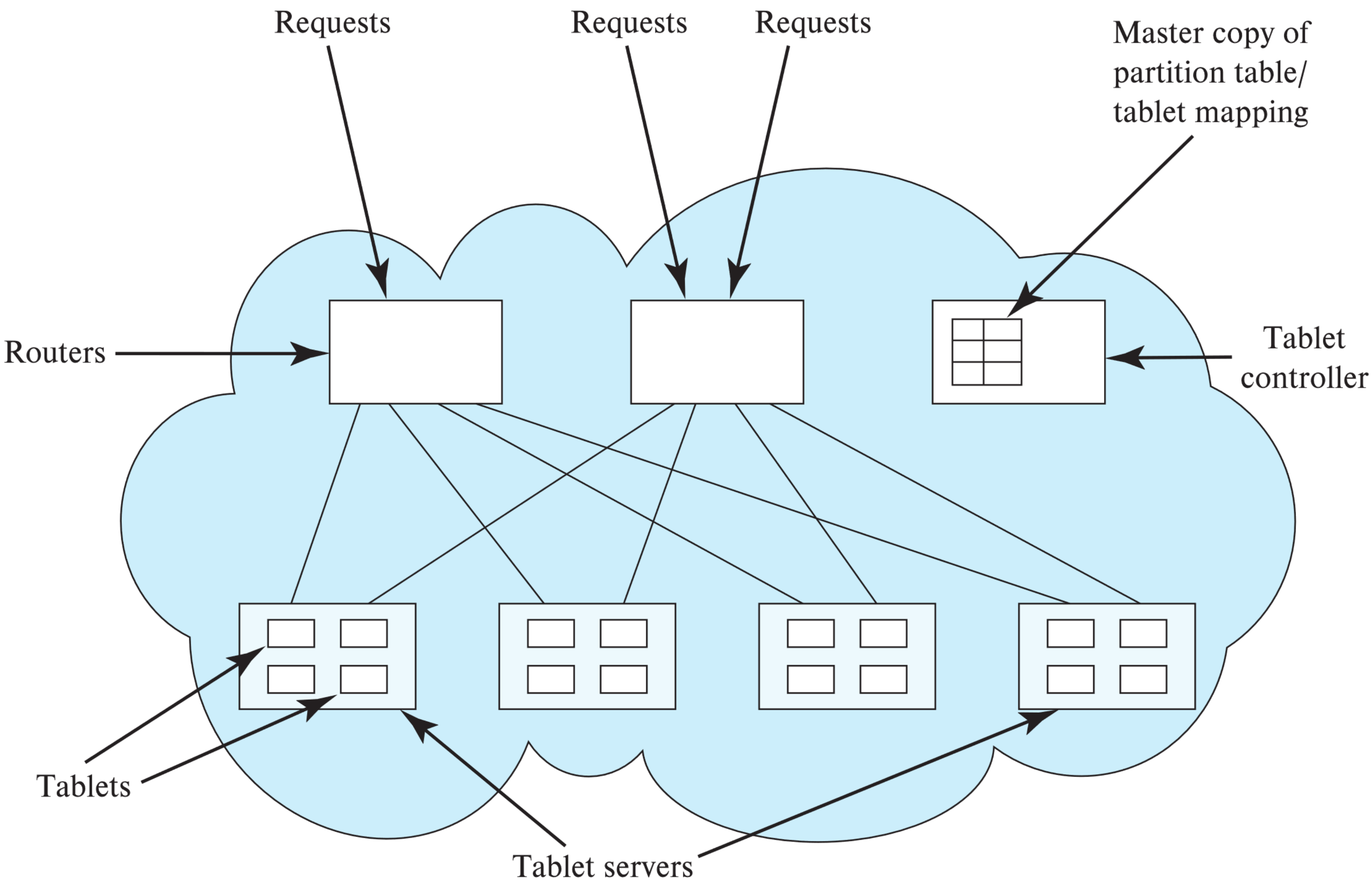
- In wide-column stores like BigTable, records may be vertically partitioned by attribute (*columnar storage*)
  - (record-identifier, attribute-name) forms a key
- Multiple attributes may be stored in one file (**column family**)
  - In BigTable records are sorted by key, ensuring all attributes of 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

Storing and Retrieving Data

Architecture of BigTable key-value store

- Table split into multiple tablets
- Tablet servers manage tablets, multiple tablets per server. Each tablet is 100-200 MB
  - Each tablet controlled by only one server
  - Tablet server splits tablets that get too big
- Master responsible for load balancing and fault tolerance
- All data and logs stored in GFS
  - Leverage GFS replication/fault tolerance
  - Data can be accessed if required from any node to aid in recovery

Architecture of Key-Value Store (modelled after Yahoo! PNUTS)



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.

Index Structures in Key-Value Stores

- Storing data in each tablet in clustered on key benefits range queries
- B^+-tree file organization works well for range queries
- Write optimized trees, especially LSM trees (Section 24.2) work well for updates as well as for range queries
  - Used in BigTable, HBase and many other key-value stores
- Some key-value stores organize records on each node by hashing, or just build a hash index on the records

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
- More details in Chapter 23

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
- Details in Chapter 23

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

END OF CHAPTER