Chapter 10: Big Data

Database System Concepts, 7th Ed.

Motivation

- Very large volumes of data being collected
  - Driven by growth of web, social media, and more recently internet-of-things
  - Web logs were an early source of data
    - Analytics on web logs has great value for advertisements, web site structuring, what posts to show to a user, etc
- Big Data: differentiated from data handled by earlier generation databases
  - **Volume**: much larger amounts of data stored
  - **Velocity**: much higher rates of insertions
  - **Variety**: many types of data, beyond relational data

Querying Big Data

- Transaction processing systems that need very high scalability
  - Many applications willing to sacrifice ACID properties and other database features, if they can get very high scalability
- Query processing systems that
  - Need very high scalability, and
  - Need to support non-relation data

Big Data Storage Systems

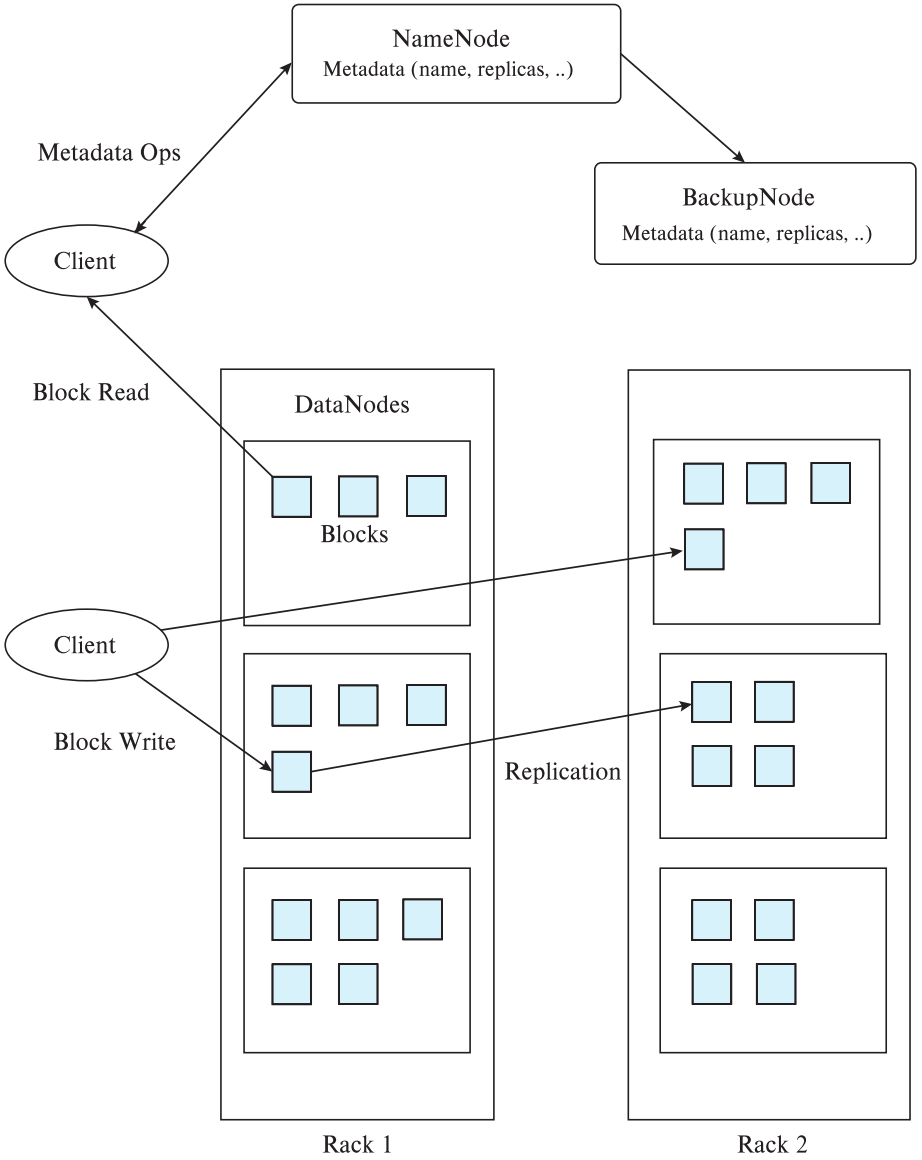
- Distributed file systems
- Sharding across multiple databases
- Key-value storage systems
- Parallel and distributed databases

Distributed File Systems

- A distributed file system stores data across a large collection of machines, but provides single file-system view
- Highly scalable distributed file system for large data-intensive applications.
  - E.g., 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amounts of data on cheap and unreliable computers
  - Files are replicated to handle hardware failure
  - Detect failures and recovers from them
- Examples:
  - Google File System (GFS)
  - Hadoop File System (HDFS)

Hadoop File System Architecture

- Single Namespace for entire cluster
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode



Hadoop Distributed File System (HDFS)

- **NameNode**
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
- **DataNode**: Maps a Block ID to a physical location on disk
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Distributed file systems good for millions of large files
  - But have very high overheads and poor performance with billions of smaller tuples

Sharding

- **Sharding**: partition data across multiple databases
- Partitioning usually done on some ***partitioning attributes*** (also known as ***partitioning keys*** or ***shard keys*** e.g. user ID
  - E.g., records with key values from 1 to 100,000 on database 1, records with key values from 100,001 to 200,000 on database 2, etc.
- Application must track which records are on which database and send queries/updates to that database
- Positives: scales well, easy to implement
- Drawbacks:
  - Not transparent: application has to deal with routing of queries, queries that span multiple databases
  - When a database is overloaded, moving part of its load out is not easy
  - Chance of failure more with more databases
    - need to keep replicas to ensure availability, which is more work for application

Parallel and Distributed Databases

- Parallel databases run multiple machines (cluser)
  - Developed in 1980s, well before Big Data
- Parallel databases were designed for smaller scale (10s to 100s of machines)
  - Did not provide easy scalability
- **Replication** used to ensure data availability despite machine failure
  - But typically restart query in event of failure
    - Restarts may be frequent at very large scale
    - Map-reduce systems (coming up next) can continue query execution, working around failures

Replication and Consistency

- **Availability** (system can run even if parts have failed) is essential for parallel/distributed databases
  - Via replication, so even if a node has failed, another copy is available
- **Consistency** is important for replicated data
  - All live replicas have same value, and each read sees latest version
  - Often implemented using majority protocols
    - E.g., have 3 replicas, reads/writes must access 2 replicas
      - Details in chapter 23
- **Network partitions** (network can break into two or more parts, each with active systems that can't talk to other parts)
- In presence of partitions, cannot guarantee both availability and consistency
  - Brewer's CAP "Theorem"

The MapReduce Paradigm

- Platform for reliable, scalable parallel computing
- Abstracts issues of distributed and parallel environment from programmer
  - Programmer provides core logic (via map() and reduce() functions)
  - System takes care of parallelization of computation, coordination, etc.
- Paradigm dates back many decades
  - But very large scale implementations running on clusters with 10^3 to 10^4 machines are more recent
  - Google Map Reduce, Hadoop, ..
- Data storage/access typically done using distributed file systems or key-value stores

MapReduce: Word Count Example

- Consider the problem of counting the number of occurrences of each word in a large collection of documents
- How would you do it in parallel?
- Solution:
  - Divide documents among workers
  - Each worker parses document to find all words, map function outputs (word, count) pairs
  - Partition (word, count) pairs across workers based on word
  - For each word at a worker, reduce function locally add up counts
- Given input: “One a penny, two a penny, hot cross buns.”
  - Records output by the map() function would be
    - (“One”, 1), (“a”, 1), (“penny”, 1),(“two”, 1), (“a”, 1), (“penny”, 1), (“hot”, 1), (“cross”, 1), (“buns”, 1).
  - Records output by reduce function would be
    - (“One”, 1), (“a”, 2), (“penny”, 2), (“two”, 1), (“hot”, 1), (“cross”, 1), (“buns”, 1)

Pseudo-code of Word Count

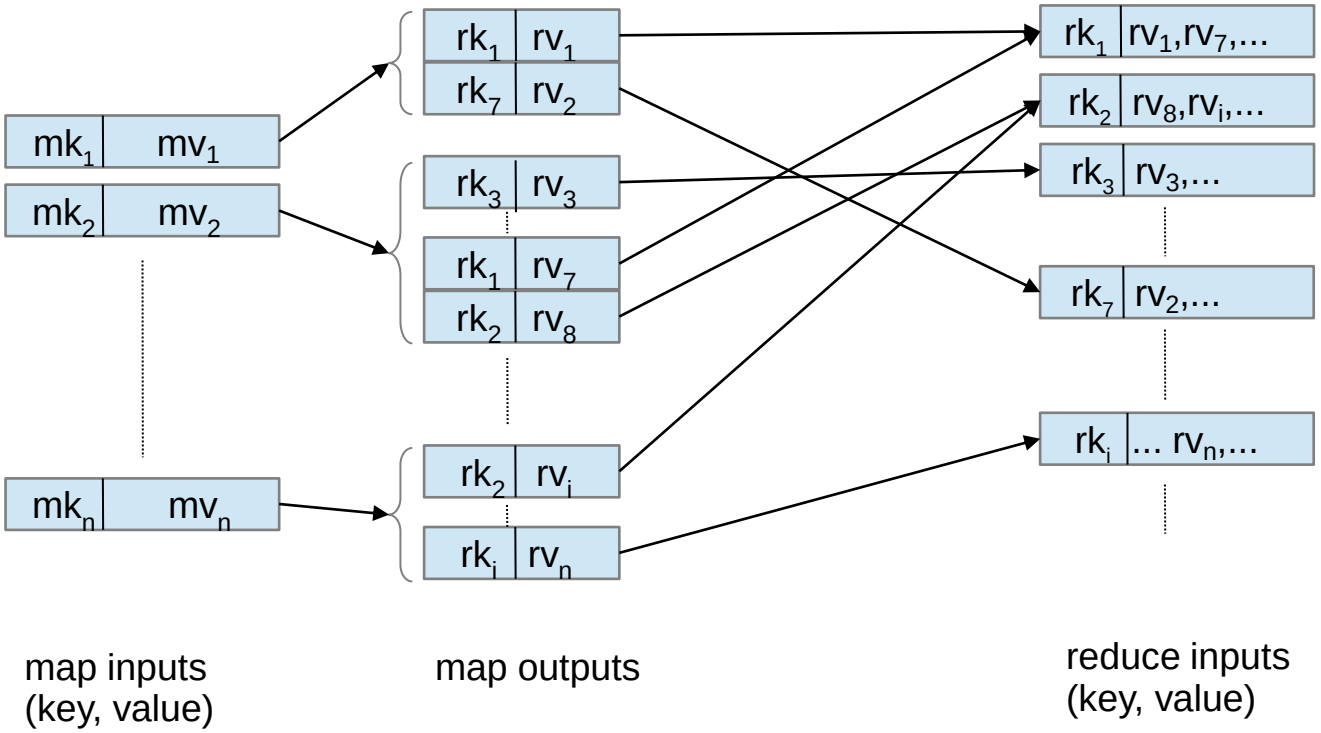
```
map(String record):
    for each word in record
       emit(word, 1);

 // First attribute of emit above is called reduce key
 // In effect, group by is performed on reduce key to create a
 // list of values (all 1's in above code).  This requires shuffle step
 // across machines.
 // The reduce function is called on list of values in each group

reduce(String key, List value_list):
    String word = key
    int count = 0;
    for each value in value_list:
       count = count + value
    Output(word, count);
```

Schematic Flow of Keys and Values

- **Flow of keys and values in a map reduce task**



Map Reduce vs. Databases

- Map Reduce widely used for parallel processing
  - Google, Yahoo, and 100's of other companies
  - Example uses: compute PageRank, build keyword indices, do data analysis of web click logs, ….
  - Allows procedural code in map and reduce functions
  - Allows data of any type
- Many real-world uses of MapReduce cannot be expressed in SQL
- But many computations are much easier to express in SQL
  - Map Reduce is cumbersome for writing simple queries

Algebraic Operations

- Current generation execution engines
  - natively support algebraic operations such as joins, aggregation, etc. natively.
  - Allow users to create their own algebraic operators
  - Support trees of algebraic operators that can be executed on multiple nodes in parallel
- E.g. Apache Tez, Spark
  - Tex provides low level API; Hive on Tez compiles SQL to Tez
  - Spark provides more user-friendly API

Algebraic Operations in Spark

- **Resilient Distributed Dataset** (**RDD**) abstraction
  - Collection of records that can be stored across multiple machines
- RDDs can be created by applying algebraic operations on other RDDs
- RDDs can be lazily computed when needed
- Spark programs can be written in Java/Scala/R
  - Our examples are in Java
- Spark makes use of Java 8 Lambda expressions; the code

  s - > Arrays.asList(s.split(" ")).iterator()

  defines unnamed function that takes argument s and executes the expression Arrays.asList(s.split(" ")).iterator() on the argument
- Lambda functions are particularly convenient as arguments to map, reduce and other functions

Streaming Data and Applications

- **Streaming data** refers to data that arrives in a continuous fashion
  - Contrast to **data-at-rest**
- Applications include:
  - Stock market: stream of trades
  - e-commerce site: purchases, searches
  - Sensors: sensor readings
    - Internet of things
  - Network monitoring data
  - Social media: tweets and posts can be viewed as a stream
- Queries on streams can be very useful
  - Monitoring, alerts, automated triggering of actions

Querying Streaming Data

Approaches to querying streams:

- **Windowing**: Break up stream into windows, and queries are run on windows
  - Stream query languages support window operations
  - Windows may be based on time or tuples
  - Must figure out when all tuples in a window have been seen
    - Easy if stream totally ordered by timestamp
    - **Punctuations** specify that all future tuples have timestamp greater that some value
- **Continuous Queries**: Queries written e.g. in SQL, output partial results based on stream seen so far; query results updated continuously
  - Have some applications, but can lead to flood of updates

Querying Streaming Data (Cont.)

Approaches to querying streams (Cont.):

- **Algebraic operators on streams**:
  - Each operator consumes tuples from a stream and outputs tuples
  - Operators can be written e.g., in an imperative language
  - Operator may maintain state
- **Pattern matching**:
  - Queries specify patterns, system detects occurrences of patterns and triggers actions
  - **Complex Event Processing** (**CEP**) systems
  - E.g., Microsoft StreamInsight, Flink CEP, Oracle Event Processing

Stream Processing Architectures

- Many stream processing systems are purely in-memory, and do not persist data
- **Lambda architecture**: split stream into two, one output goes to stream processing system and the other to a database for storage
  - Easy to implement and widely used
  - But often leads to duplication of querying effort, once on streaming system and once in database

Stream Extensions to SQL

- SQL Window functions described in Section 5.5.2
- Streaming systems often support more window types
  - **Tumbling window**
    - E.g., hourly windows, windows don’t overlab
  - **Hopping window**
    - E.g., hourly window computed every 20 minutes
  - **Sliding window**
    - Window of specified size (based on timestamp interval or number of tuples) around each incoming tuple
  - **Session window**
    - Groups tuples based on user sessions

Publish Subscribe Systems

- **Publish-subscribe** (**pub-sub**) systems provide convenient abstraction for processing streams
  - Tuples in a stream are published to a topic
  - Consumers subscribe to topic
- Parallel pub-sub systems allow tuples in a topic to be partitioned across multiple machines
- **Apache Kafka** is a popular parallel pub-sub system widely used to manage streaming data
- More details in book

Graph Data Model

- Graphs are a very general data model
- ER model of an enterprise can be viewed as a graph
  - Every entity is a node
  - Every binary relationship is an edge
  - Ternary and higher degree relationships can be modelled as binary relationships

Graph Data Model (Cont.)

- Graphs can be modelled as relations
  - *node*(*ID, label, node_data*)
  - *edge*(*fromID, toID, label, edge_data*)
- Above representation too simplistic
- Graph databases like Neo4J can provide a **graph view of relational schema**
  - Relations can be identified as representing either nodes or edges
- Query languages for graph databases make it
  - easy to express queries requiring edge traversal
  - allow efficient algorithms to be used for evaluation

End of Chapter 10