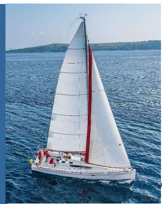
فصل دهم: داده های حجیم
(Big Data)
(زیر بخش های ۳ و ۴ از فصل ۱۰ حذف می باشد.)

درس پایگاه داده
دانشگاه صنعتی نوشیروانی بابل
مهدی عمادی
m.emadi@nit.ac.ir



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

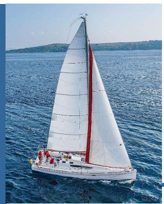
- **Transaction processing systems that need very high scalability**
  - **Many applications willing to sacrifice ACID properties and other database features, if they can get very high scalability**
- **Query processing systems that**
  - **Need very high scalability, and**
  - **need to support non-relation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

Big Data Storage Systems

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

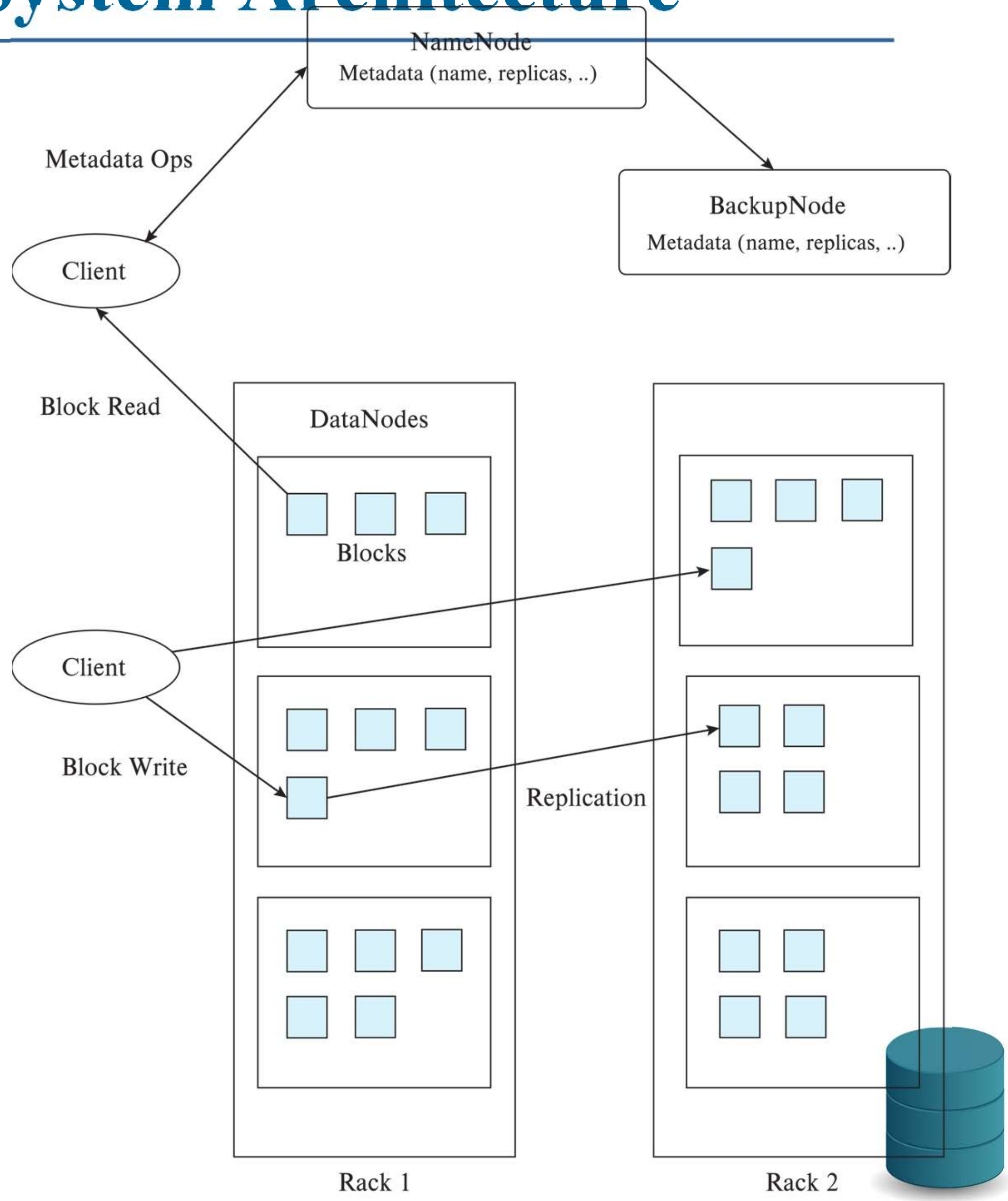
Distributed File Systems

- 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)

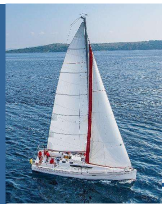
- **NameNode**
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
- **DataNode** : Maps a Block ID to a physical location on disk
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Distributed 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 Databases and Data Stores

- Supporting scalable data access
  - Approach 1: memcache or other caching mechanisms at application servers, to reduce database access
    - Limited in scalability
  - Approach 2: Partition (“shard”) data across multiple separate database servers
  - Approach 3: Use existing parallel databases
    - Historically: parallel databases that can scale to large number of machines were designed for decision support not OLTP
  - Approach 4: Massively Parallel Key-Value Data Store
    - Partitioning, high availability etc completely transparent to application
- Sharding systems and key-value stores don’t support many relational features, such as joins, integrity constraints, etc, across partitions.



Key Value Storage Systems

- **Key-value storage systems store large numbers (billions or even more) of small (KB-MB) sized records**
- **Records are partitioned across multiple machines and**
- **Queries are routed by the system to appropriate machine**
- **Records are also replicated across multiple machines, to ensure availability even if a machine fails**
  - **Key-value stores ensure that updates are applied to all replicas, to ensure that their values are consistent**

Key Value Storage Systems

- Key-value stores may store
  - **uninterpreted bytes**, with an associated key
    - E.g. Amazon S3, Amazon Dynamo
  - **Wide-table** (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g. filtering) to execute on storage node
  - JSON
    - MongoDB, CouchDB (document model)
- **Document stores** store semi-structured data, typically JSON
- Some key-value stores support multiple versions of data, with timestamps/version numbers

Data Representation

- **An example of a JSON object is:**

```
{
    "ID": "22222",
    "name": {
        "firstname: "Albert",
        "lastname: "Einstein"
    },
    "deptname": "Physics",
    "children": [
        { "firstname": "Hans", "lastname": "Einstein" },
        { "firstname": "Eduard", "lastname": "Einstein" }
    ]
}
```

Key Value Storage Systems

- Key-value stores support
  - *put*(key, value): used to store values with an associated key,
  - *get*(key): which retrieves the stored value associated with the specified key
  - *delete*(key) -- Remove the key and its associated value
- Some systems also support *range queries* on key values
- Document stores also support queries on non-key attributes
  - See book for MongoDB queries
- Key value stores are not full database systems
  - Have no/limited support for transactional updates
  - Applications must manage query processing on their own
- Not supporting above features makes it easier to build scalable data storage systems
  - Also called NoSQL systems

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"

Replication and Consistency

- Very large systems will partition at some point
  - Choose one of consistency or availability
- Traditional database choose consistency
- Most Web applications choose availability
  - Except for specific parts such as order processing
- More details later, in Chapter 23

STREAMING DATA

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

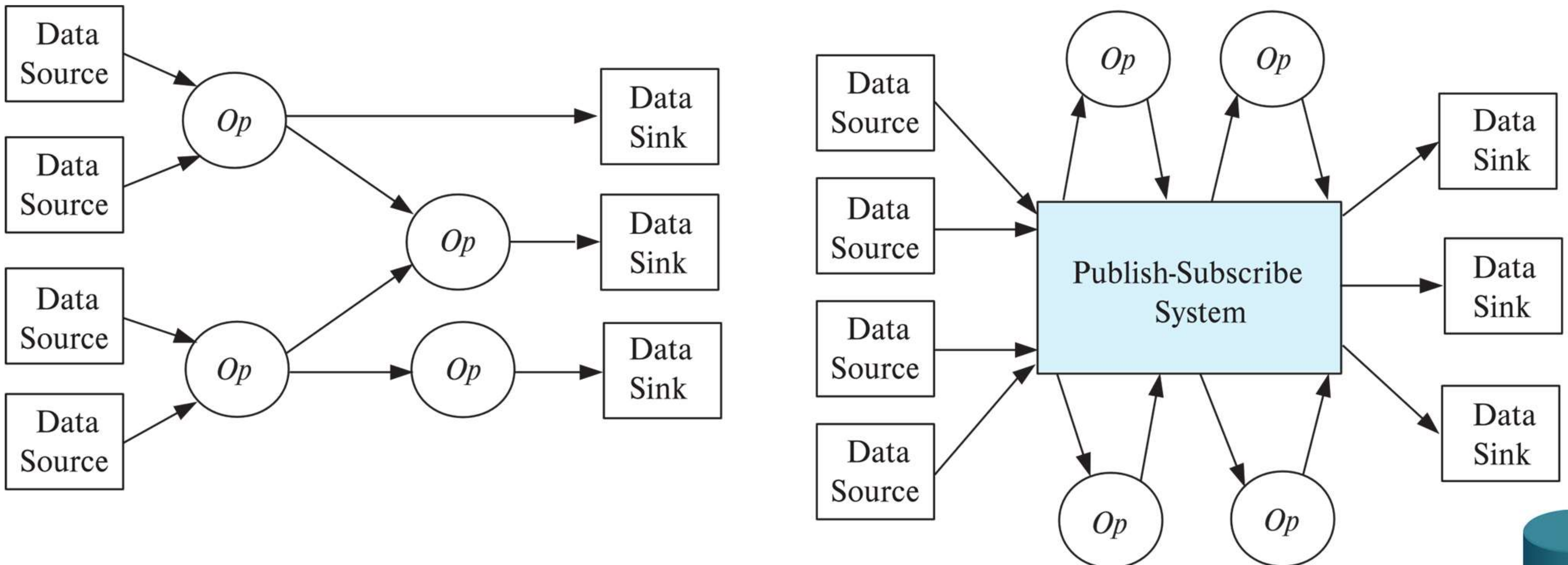
- SQL Window functions described in Section 5.5.2
- Streaming systems often support more window types
  - Tumbling window
    - E.g. hourly windows, windows don’t overlab
  - Hopping window
    - E.g. hourly window computed every 20 minutes
  - Sliding window
    - Window of specified size (based on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

Window Syntax in SQL

- Windowing syntax varies widely by system
- E.g. in Azure Stream Analytics SQL:
  select *item*, *System.Timestamp* as *window end*, sum(*amount*)
  from *order* timestamp by *datetime*
  group by *itemid*, tumblingwindow(*hour*, 1)
- Aggregates are applied on windows
- Result of windowing operation on a stream is a relation
- Many systems support stream-relation joins
- Stream-stream joins often require join conditions to specify bound on timestamp gap between matching tuples
  - E.g. tuples must be at most 30 minutes apart in timestamp

Algebraic Operations on Streams

- Tuples in streams need to be routed to operators
- Routing of streams using DAG and publish-subscribe representations
  - Used e.g. in Apache Storm and Apache Kafka respective

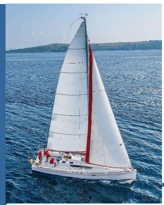


(a) DAG representation of streaming data flow

(b) Publish-subscribe representation of streaming data flow

Publish Subscribe Systems

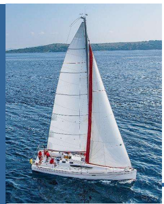
- **Publish-subscribe (pub-sub) systems provide convenient abstraction for processing streams**
  - **Tuples in a stream are published to a topic**
  - **Consumers subscribe to topic**
- **Parallel pub-sub systems allow tuples in a topic to be partitioned across multiple machines**
- **Apache Kafka is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



GRAPH DATABASES

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**

Graph Data Model (Cont.)

- Suppose
  - relations *instructor* and *student* are nodes, and
  - relation *advisor* represents edges between instructors and student
- Query in Neo4J:
  match (*i*:*instructor*)<-[:*advisor*]-(*s*:*student*)
  where *i.dept name*= 'Comp. Sci.'
  return *i*.ID as ID, *i.name* as *name*, collect(*s.name*) as *advisees*
- match clause matches nodes and edges in graphs
- Recursive traversal of edges is also possible
  - Suppose *prereq*(*course_id, prereq_id*) is modeled as an edge
  - Transitive closure can be done as follows:

    match (*c1*:*course*)-[:*prereq* *1..]->(*c*2:*course*)
    return *c*1.*course id*, *c*2.*course id*

Parallel Graph Processing

- Very large graphs (billions of nodes, trillions of edges)
  - Web graph: web pages are nodes, hyper links are edges
  - Social network graph: people are nodes, friend/follow links are edges
- Two popular approaches for parallel processing on such graphs
  - Map-reduce and algebraic frameworks
  - Bulk synchronous processing (BSP) framework
- Multiple iterations are required for any computations on graphs
  - Map-reduce/algebraic frameworks often have high overheads per iteration
  - BSP frameworks have much lower per-iteration overheads
- Google's Pregel system popularized the BSP framework
- Apache Giraph is an open-source version of Pregel
- Apache Spark's GraphX component provides a Pregel-like API

Bulk Synchronous Processing

Bulk synchronous processing framework

- Each vertex (node) of a graph has data (state) associated with it
  - Vertices are partitioned across multiple machines, and state of node kept in-memory
- Analogous to map() and reduce() functions, programmers provide methods to be executed for each node
  - Node method can send messages to or receive messages from neighboring nodes
- Computation consists of multiple iterations, or supersteps
- In each **superstep**
  - nodes process received messages
  - update their state, and
  - send further messages or vote to halt
  - Computation ends when all nodes vote to halt, and there are no pending messages;

پایان فصل دهم