CPSC 436C
Cloud Computing for Data Science

Data store – Part 3

Maryam R.Aliabadi

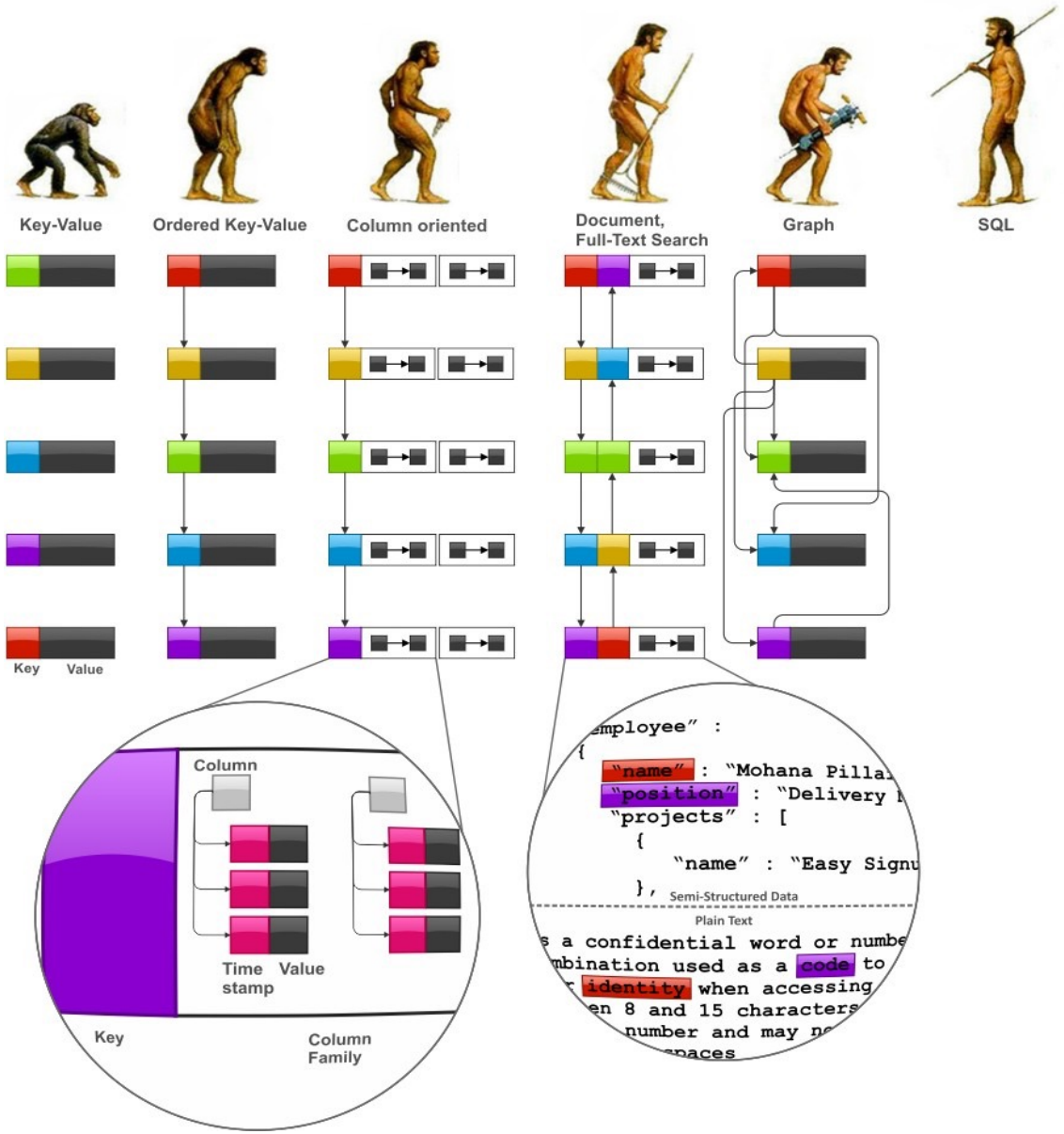
mraiyata@cs.ubc.ca

Spring 2024

Last Class‘ Review

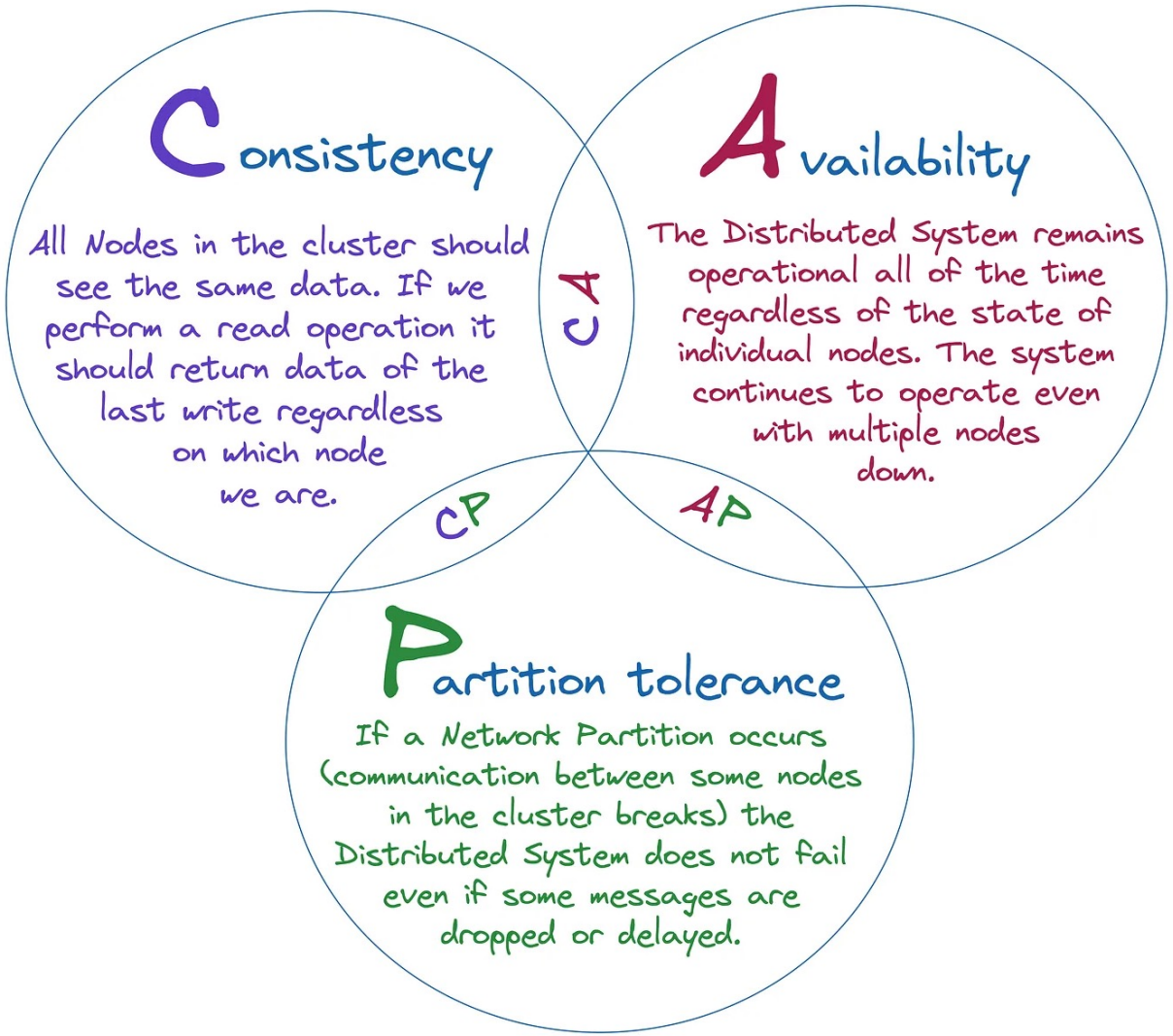
- SQL data bases
  - Structured data
  - ACID properties
- NoSQL databases
  - Unstructured/Semi-structured data
  - BASE properties
  - Data models

NoSQL Data Models

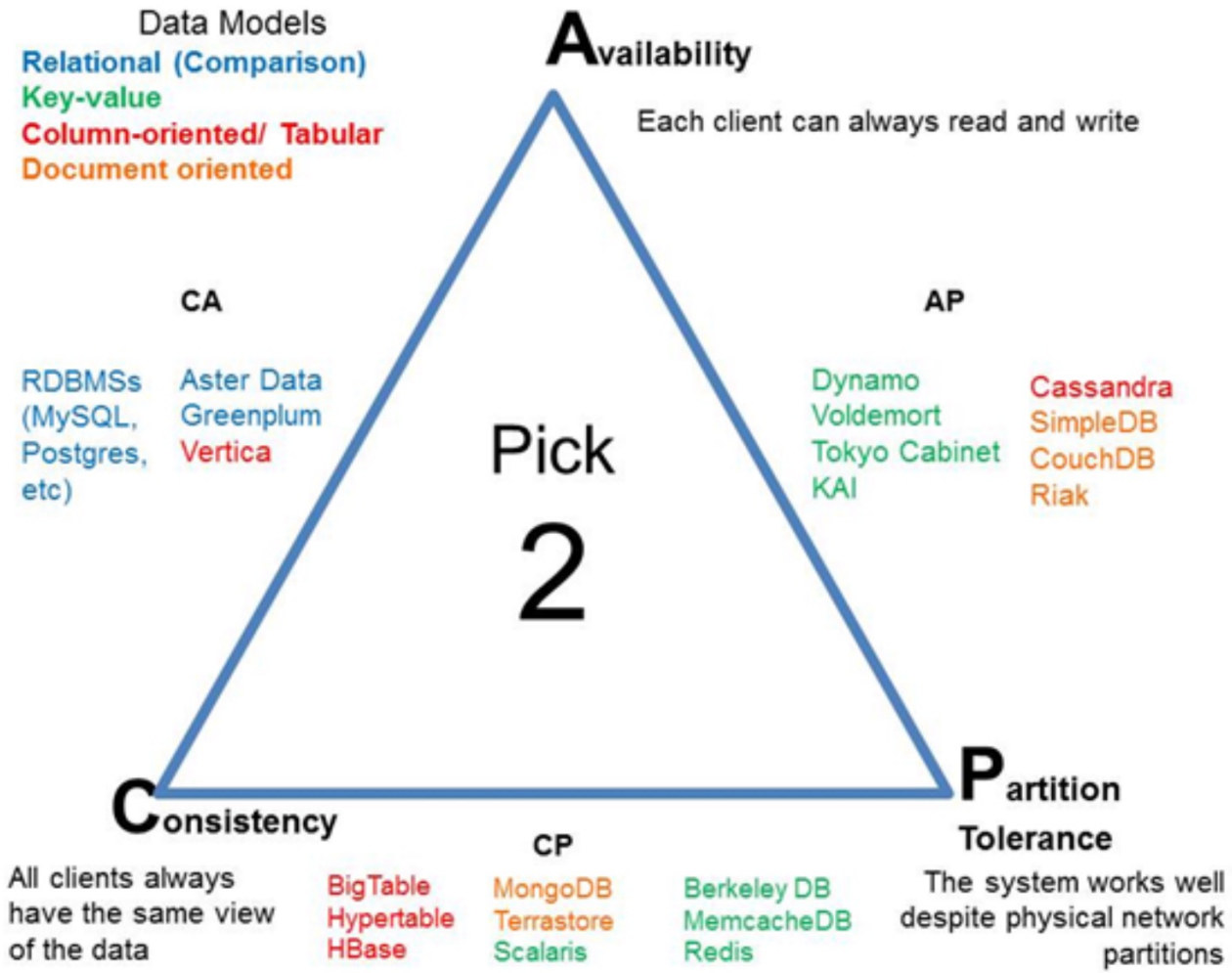


[http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques]

CAP Definitions



CAP Theorem


Data Models
Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented
Availability
Each client can always read and write
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick 2
Consistency
All clients always have the same view of the data
CP
BigTable
Hypertable
HBase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance
The system works well despite physical network partitions

Database Use Cases

1.CA Database (Consistency and Availability):

- Banking Systems

2.AP Database (Availability and Partition Tolerance):

- Social Media Platforms

3.CP Database (Consistency and Partition Tolerance):

- Inventory Management Systems

Today ‘s Topic:

Key-Value Store

Use case: Dynamo

Dynamo

- Distributed key/value storage system
- Scalable
- CAP : Availability and Partition tolerance

Design Consideration

- It sacrifices strong consistency for availability: always writable
- Incremental scalability
- Symmetry: every node should have the same set of responsibilities as its peers
  - Uniform distribution of data
  - Easy system management
- Decentralization
  - Enhances fault tolerance
  - Avoids single point of failure

System Architecture

1. Data partitioning
2. Replication
3. Data versioning
4. Dynamo API
5. Membership management
6. Failure detection and handling

1-Data Partitioning

- If size of data exceeds the capacity of a single machine: Partitioning
- Sharding data (horizontal partitioning).
- Consistent hashing is one form of automatic sharding.

Vertical Partitioning Example

- Original dataset

CustomerID	CustomerName	CustomerEmail	OrderID	OrderDate	ProductName	Quantity
1	Alice	alice@email.com	1001	2023-01-10	Product A	2
2	Bob	bob@email.com	1002	2023-01-11	Product B	1
3	Carol	carol@email.com	1003	2023-01-12	Product A	3
4	Dave	dave@email.com	1004	2023-01-13	Product C	2

- Partition 1

CustomerID	CustomerName	CustomerEmail
1	Alice	alice@email.com
2	Bob	bob@email.com
3	Carol	carol@email.com
4	Dave	dave@email.com

- Partition 2

OrderID	OrderDate	ProductName	Quantity
1001	2023-01-10	Product A	2
1002	2023-01-11	Product B	1
1003	2023-01-12	Product A	3
1004	2023-01-13	Product C	2

Horizontal Partitioning Example

- Original dataset

CustomerID	CustomerName	CustomerEmail	OrderID	OrderDate	ProductName	Quantity
1	Alice	alice@email.com	1001	2023-01-10	Product A	2
2	Bob	bob@email.com	1002	2023-01-11	Product B	1
3	Carol	carol@email.com	1003	2023-01-12	Product A	3
4	Dave	dave@email.com	1004	2023-01-13	Product C	2

- Partition 1

CustomerID	CustomerName	CustomerEmail	OrderID	OrderDate	ProductName	Quantity
1	Alice	alice@email.com	1001	2023-01-10	Product A	2
2	Bob	bob@email.com	1002	2023-01-11	Product B	1

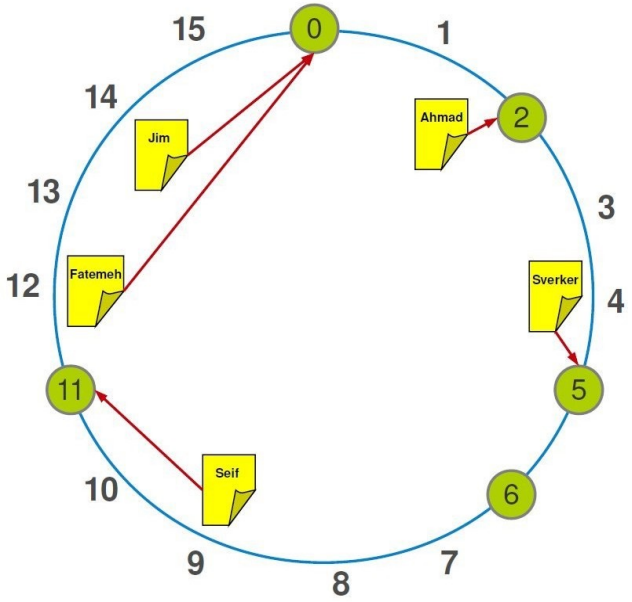
- Partition 2

CustomerID	CustomerName	CustomerEmail	OrderID	OrderDate	ProductName	Quantity
3	Carol	carol@email.com	1003	2023-01-12	Product A	3
4	Dave	dave@email.com	1004	2023-01-13	Product C	2

Consistent Hashing

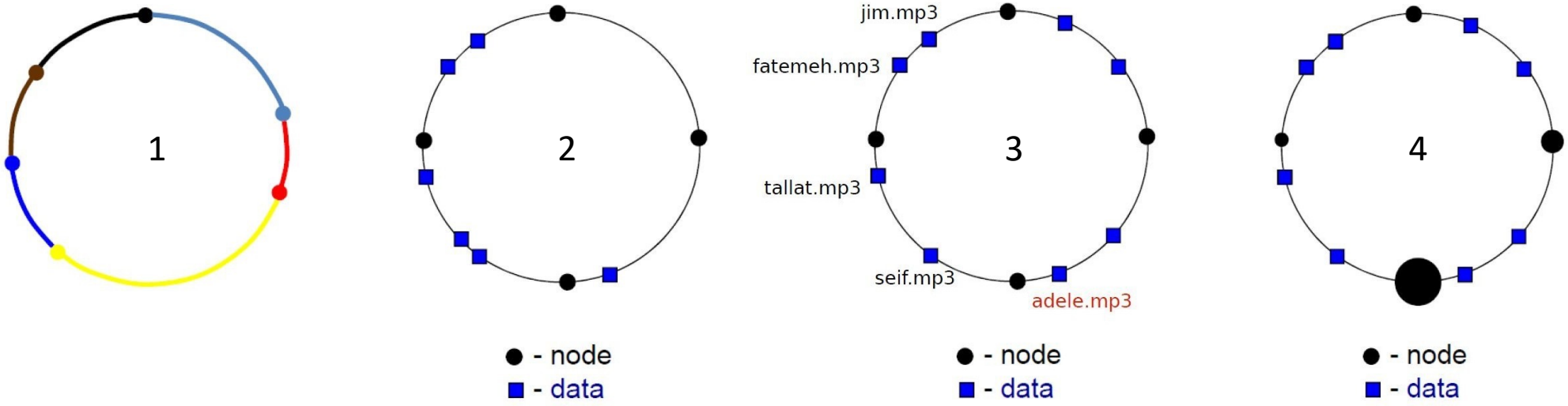
- Hashing
  - Hash both data and nodes using the same hash function in a same id space.
- Mapping to a Ring
  - partition = hash(d) mod n, d: data, n: number of nodes
- Data Placement

hash("Fatemeh") = 12
hash("Ahmad") = 2
hash("Seif") = 9
hash("Jim") = 14
hash("Sverker") = 4



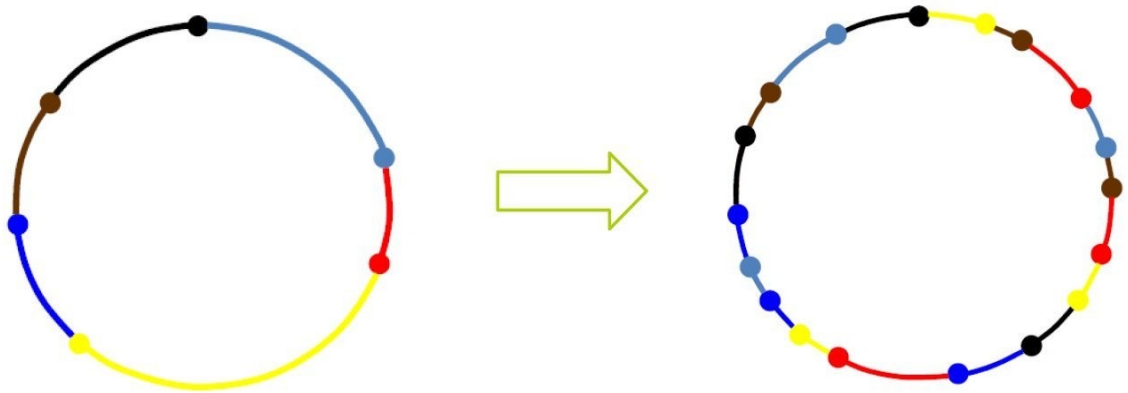
Challenge: Load Imbalance

- Consistent hashing may lead to imbalance.
  - 1-Node identifiers may not be balanced.
  - 2-Data identifiers may not be balanced.
  - 3-Hot spots.
  - 4-Heterogeneous nodes.



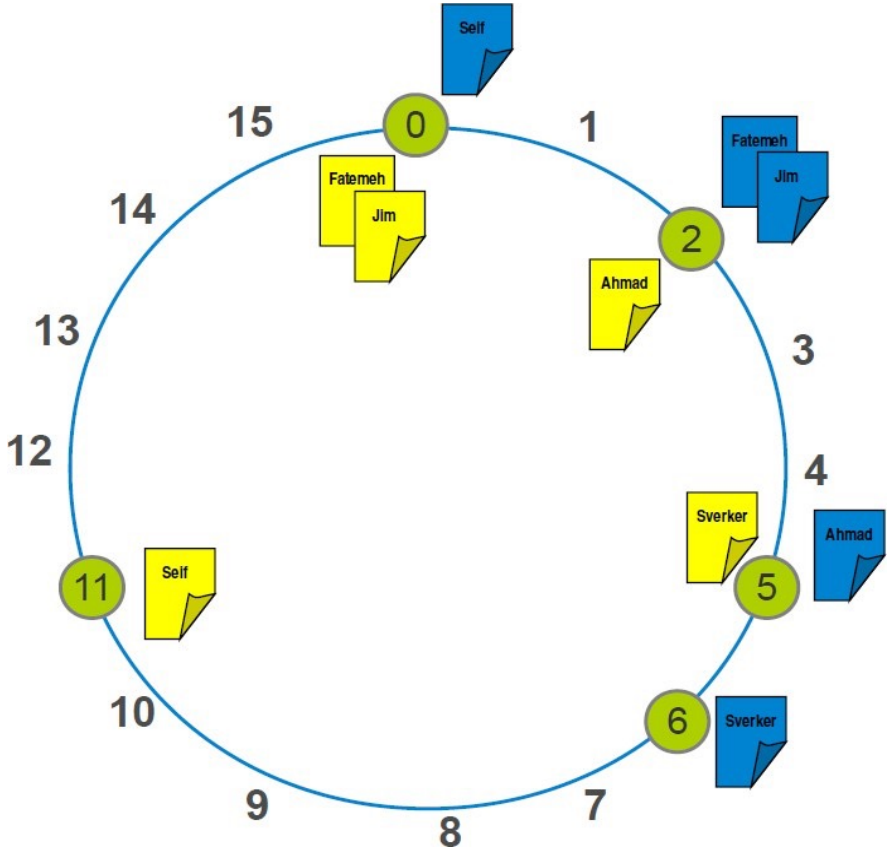
Load Balancing Vs. Virtual Nodes

- Each physical node picks multiple random identifiers.
- Each identifier represents a virtual node.
- Each node runs multiple virtual nodes.



2-Replication

- To achieve high availability and durability, data should be replicates on multiple nodes.



3-Data Versioning

- Updates are propagated asynchronously.

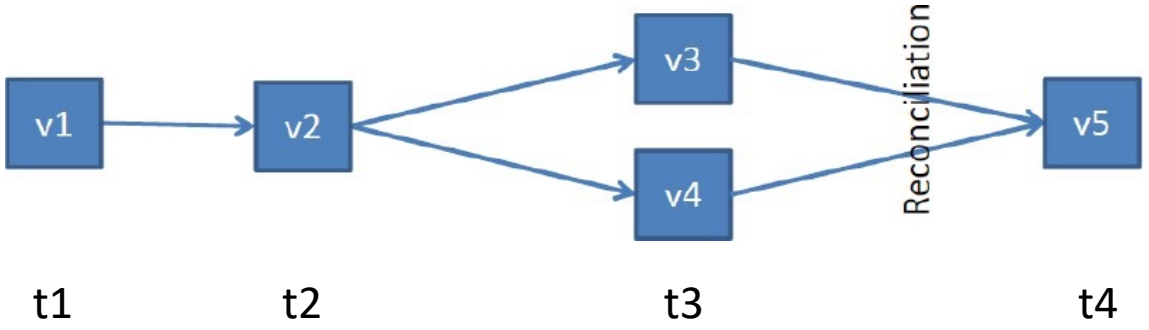
- Each update/modification of an item results in a new and immutable version of the data.
  - Multiple versions of an object may exist.

- Replicas eventually become consistent.

Data Versioning

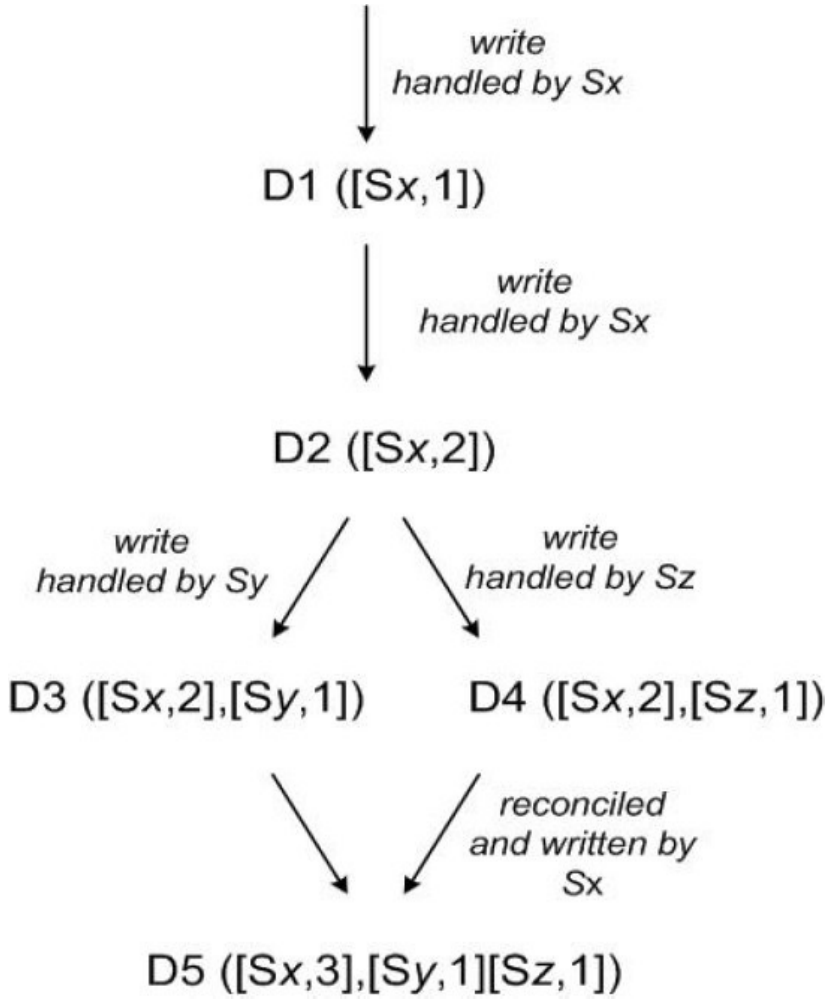
- Version branching can happen due to node/network failures.

- Dynamo uses vector clocks for capturing causality, in the form of [node, counter] pairs
  - If causal: older version can be forgotten
  - If concurrent: conflict exists, requiring reconciliation



Data Versioning

- Client C1 writes new object via Sx.
- C1 updates the object via Sx.
- C1 updates the object via Sy.
- C2 reads D2 and updates the object via Sz.
- C3 reads D3 and D4 via Sx.
  - The read context is a summary of the clocks of D3 and D4: [(Sx, 2), (Sy, 1), (Sz, 1)].
- Reconciliation



4-Dynamo API

- `get(key)`
  - Return single object or list of objects with conflicting version and context.

- `put(key, context, object)`
  - Store object and context under key.
  - Context encodes system metadata, e.g., version number.

Execution of Operations

- Client can send the request:
  - To the node responsible for the data (Partition coordinator)
    - save on latency, code on client
  - To a generic load balancer: extra hop
    - distribute incoming requests evenly across the Dynamo nodes

- The choice depends on the specific configuration and requirements of the Dynamo deployment.

Put Operation

- Coordinator generates new vector clock and writes the new version locally.
- Send to N nodes.
- Wait for response from W nodes.
- Using W=1
  - High availability for writes
  - Low durability

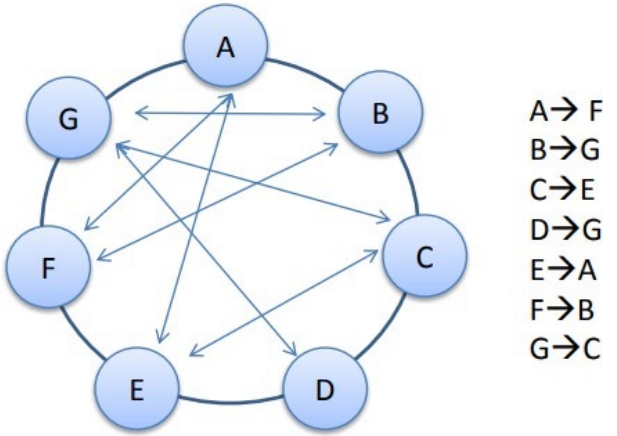
Get Operation

- Coordinator requests existing versions from N.
  - Wait for response from R nodes.

- If multiple versions, return all versions that are causally unrelated.
- Divergent versions are then reconciled.
- Reconciled version written back.
- Using R=1
  - High performance read engine.

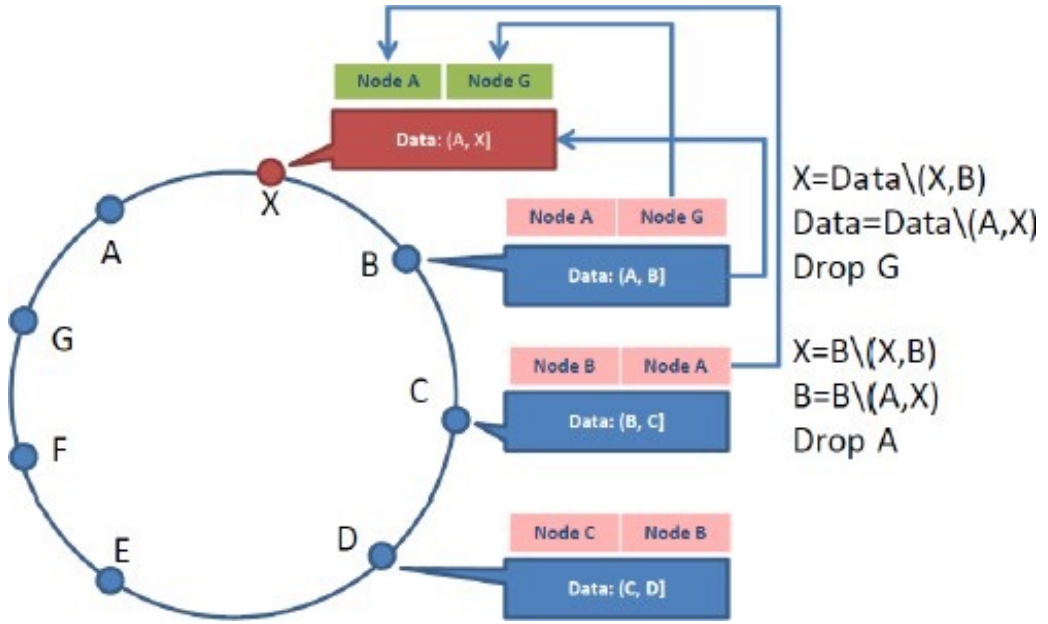
5-Membership Management

- Administrator explicitly adds and removes nodes.
- Gossiping to propagate membership changes.
  - periodically, each node contacts another randomly selected node.
- Eventually consistent view.
- O(1) hop overlay.
  - data access operations can be performed with low latency (constant number of hops).



Adding Nodes

- A new node X added to system.
  - X is assigned key ranges w.r.t. its virtual servers.
  - For each key range, it transfers the data items.



Removing Nodes

- Reallocation of keys is a reverse process of adding nodes.
  - Determining new owners
  - Assigning new key ranges to new owners
  - Data transfer to new owners
  - Synchronization

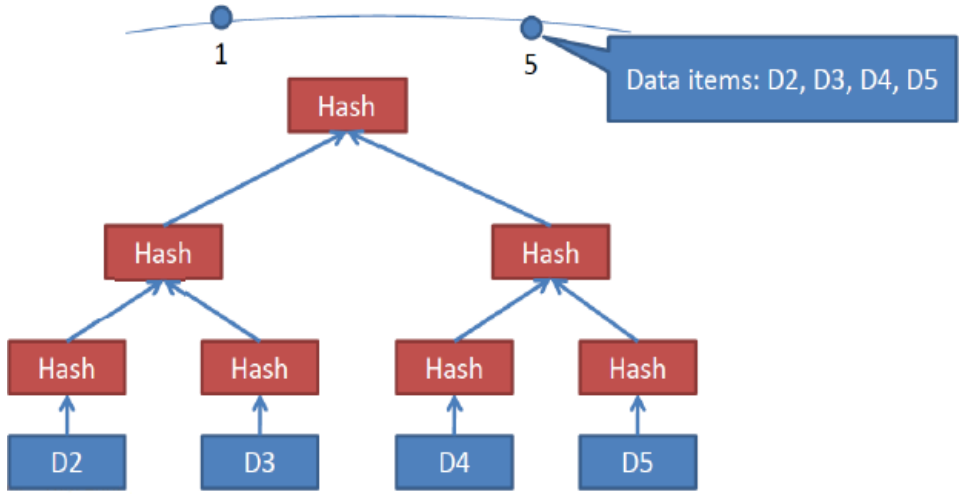
6 - Failure Detection and Handling

- Passive failure detection.
  - Use pings only for detection from failed to alive.

- In the absence of client requests, node A doesn't need to know if node B is alive.

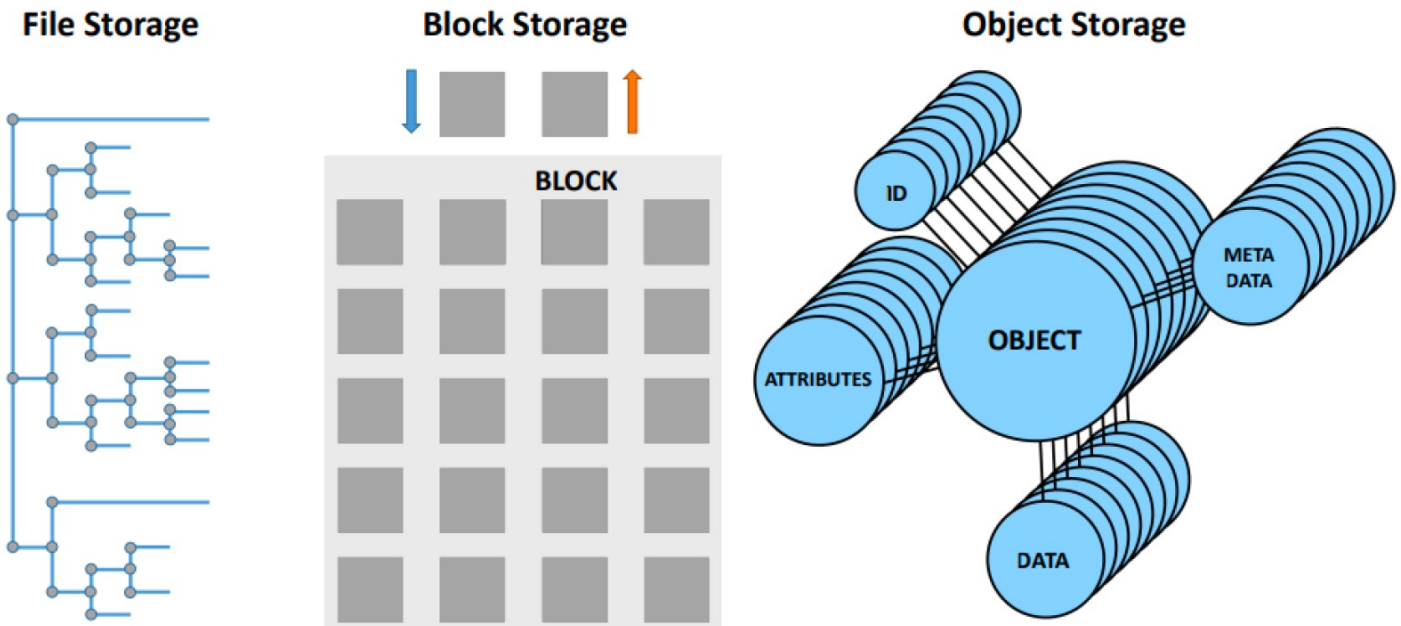
Handling Permanent Failure

- Anti-entropy for replica synchronization.
- Use Merkle trees for fast inconsistency detection and minimum transfer of data.
  - Nodes maintain Merkle tree of each key range.
  - Exchange root of Merkle tree to check if the key ranges are updated.



Cloud Storage Services

- File storage
- Block storage
- Object storage

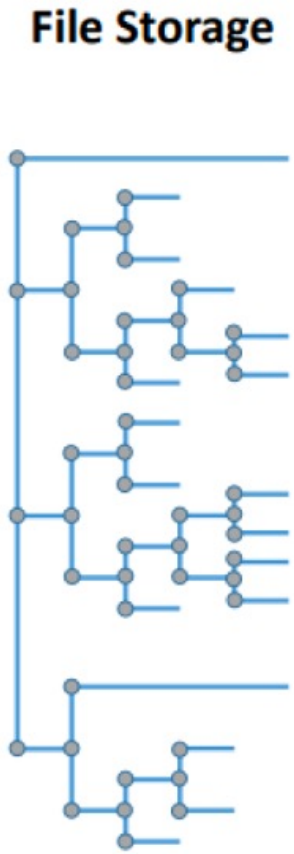


File Storage

- **File Storage**: a method of data storage where data is organized into files and directories, in a hierarchical structure

- **Example:**

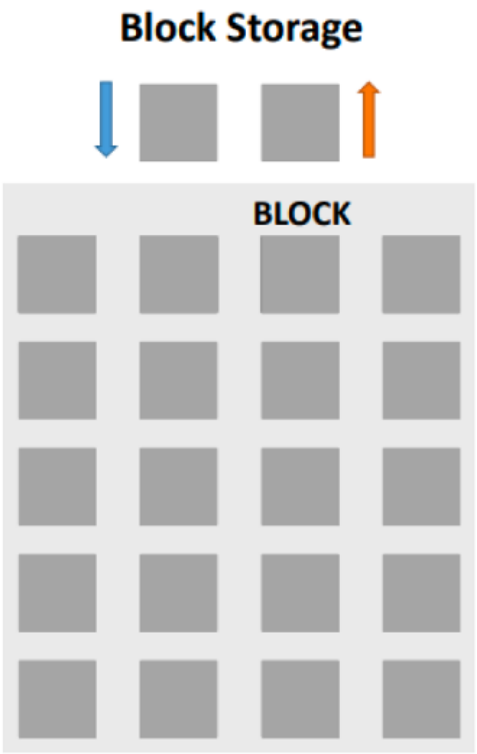
  Distributed File Systems: Examples include Hadoop Distributed File System (HDFS) and Google File System (GFS)



Block Storage

- **Block Storage**: a type of data storage where data is divided into fixed-sized blocks and stored on separate volumes or devices.

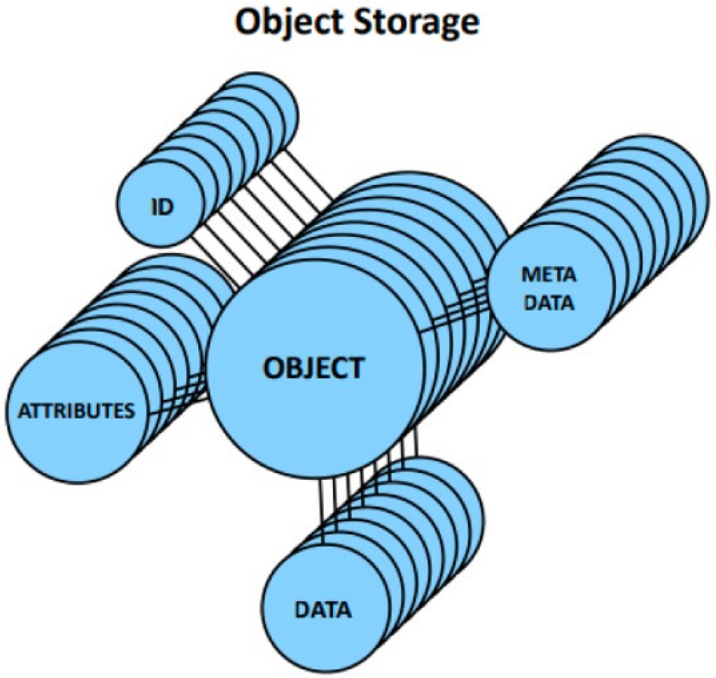
- **Examples:**
  1. Amazon Elastic Block Store (EBS)
  2. Storage Area Network (SAN)



Object Storage

- **Object Storage**: a data storage architecture that manages data as objects, including the data itself, metadata, and a unique identifier.

- **Examples:**
  1. Amazon Simple Storage Service (S3)
  2. Azure Blob Storage



Cloud Storage Vs. Databases

Database	Underlying Storage	Example on AWS
Relational (SQL)	File Storage	Amazon RDS
Document Stores (NoSQL)	Object Storage	Amazon MongoDB
Key-Value Stores (NoSQL)	Block Storage	Amazon DynamoDB
Column-Family Stores (NoSQL)	Custom Storage	Cassandra
Graph Stores (NoSQL)	Custom Storage	Amazon Neptune
Object Stores (NoSQL)	Object Storage	Amazon S3

Object Stores

- An object store is a category of NoSQL database that is designed to store and manage data as discrete objects or blobs.

- Key characteristics:
  - Schema-less
  - Scalability
  - High availability
  - Simplicity
  - Unstructured/semi-structured data

- CAP

How to choose the best option for data storage?

Data Store Evaluation

- To be able to evaluate a data store you need to understand your needs
  - What is the structure of your data?
  - How big is your data?
  - What is the velocity?
  - What kind of processing is needed?
  - What kind of queries do you want to answer?
  - What is the expected latency?
  - What access pattern is needed?
  - What are the cost requirements?

Dimensions

- **Data**
  - What characteristics should be considered with respect to **data**?
- **Processing**
  - What characteristics should be considered with respect to **processing**?
- **Other dimensions**:
  - Cost, Implementation complexity

Data Dimension

- Data related characteristics
  - Structure
  - Size
  - Sink rate
  - Source rate
  - Quality

Data Structure

- What is the type of the data (Variety)?
  - **Structured**: Well defined schema, data types, understandable by machine
  - **Unstructured**: Loosely typed (text, pics)
  - **Semi-structured**: Mix of structured and unstructured. Ex. Well defined schema, but some attributes are unstructured

Size

- What is the size of the data (Volume)?
  - **S**: Megabytes
  - **M**: Gigabytes
  - **L**: Tera Bytes
  - **XL**: 100's of Tera Bytes
  - **XXL**: Peta Bytes

Sink Rate/Speed

- How fast the data are coming in (Velocity)?
  - **Very High**: > hundreds of updates per second
  - **High**:> tens of updates per hours
  - **Medium**: a few updates per hour
  - **Low**: Updates daily or less frequently

Source Rate/Speed

- How updated is the indexing/speed layer?
  - **High**: updated in “real-time” as data arrives
  - **Medium**: Updated on an hourly basis
  - **Low**: Updates on a daily or less frequently

Quality

- How well does the system deal with bad or low-quality data (Veracity)?
  - **High**: can compensate and handle in an automated fashion
  - **Medium**: can handle but results may be unreliable
  - **Low**: can not handle bad or low-quality data. Will not provide any results

Data Store Solutions - Examples

- **RDBMS**: Relational model with powerful querying capabilities
- **HDFS**: Batch oriented system for storing large data sets
- **DynamoDB**: a fully managed NoSQL database with highly available storage for unstructured and semi-structured data.
- **Amazon S3**: a scalable and highly available object storage service

Data Store Solution Comparisons

Perspective	RDBMS	HDFS	S3	DynamoDB
Data Variety	Structured	Semi-structured	Object (Unstructured)	Semi-structured
Data Volume	M (Gigabytes)	XXL (Peta Bytes)	XXL (Peta Bytes)	M (Gigabytes)
Sink Rate (Data Velocity)	Low	Very High	Low	Very High
Source Rate	Medium (Hourly)	Low (Daily)	Low (Daily)	High (Real-time)
Data Veracity	Medium	Medium	Medium	High
Access Pattern	Complex (SQL Queries)	Batch Processing	Random Access	Real-time Queries
Cost	Medium to High	Low to Medium	Pay-as-you-go	Pay-as-you-go

Recap

- Key-Value Store – DynamoDB
  - Consistent Hashing
  - Dynamo API
  - Version Branching
  - Membership management
  - Failure detection and handling
- Data Store trade-offs

Next topic:
Data Management Systems