NoSQL Databases

ACS 575: Database Systems

Instructor: Dr. Jin Soung Yoo, Professor
Department of Computer Science
Purdue University Fort Wayne

References

- W. Lemanhieu, et al., Principles of Database Management: The Practical Guide to Storing, Managing and Analyzing Big and Small Data, Ch 11
- Dan Sullivan, NoSQL for Mere Mortals

Outline

- The NoSQL Movement
- What is NoSQL?
  - Features of NoSQL
  - CAP Theorem, Base Principle
  - NoSQL vs. Relational Databases
- NoSQL Categories
  - Key-Value Stores
  - Document-oriented Stores
  - Column-oriented Databases
  - Graph-based Databases

The World Before Codd

- Before Edgar Codd introduced the relational model, databases operated on different structural paradigms.
- The **hierarchical approach**, akin to our modern file systems, organized data in a tree-like structure, with parent-child relationships.
- The **network approach**, which still influences modern graph databases, allowed for more complex data relationships by connecting records in a network-like structure.

Limitations of Early Database Systems

- Early database approaches lacked clear separation between conceptual relationships and physical storage.
- They failed to offer adequate *data abstraction* and *program-data independence*.
  - Data abstraction, essential for simplifying development, was lacking.
- Developers needed in-depth knowledge of database implementation details.

Limitations of Early Database Systems (Cont.)

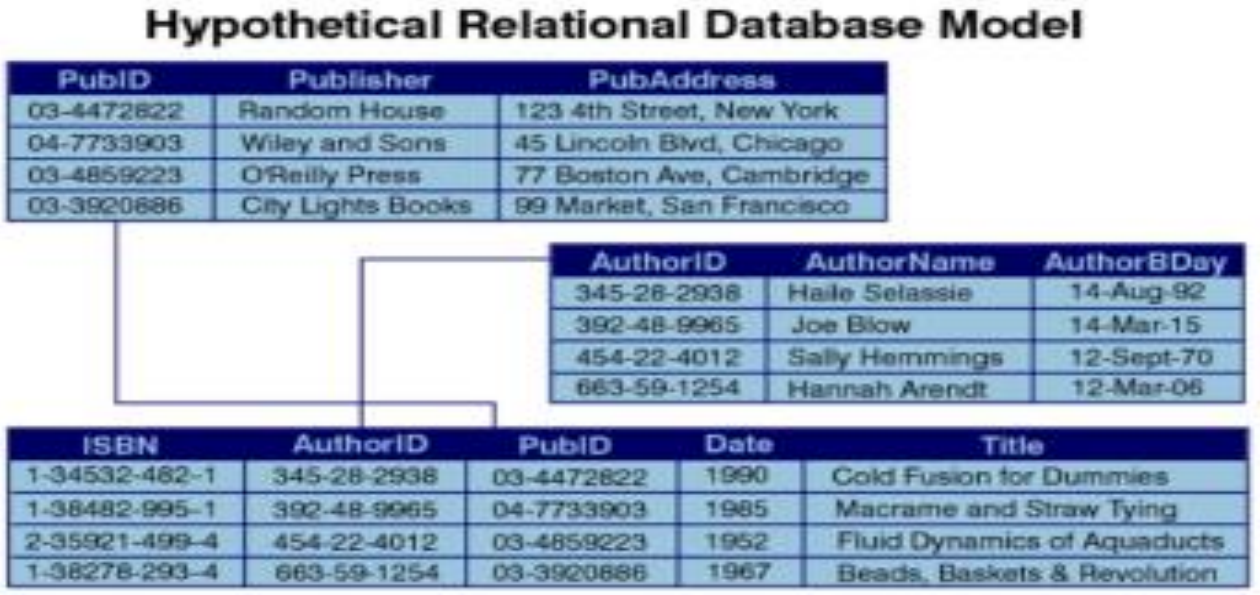
- Reorganizing the database in response to application requirement changes was labor-intensive.
- While efficient for original queries and transactions, they lacked flexibility for new ones.
- They lacked dedicated database query languages, offering only general programming language interfaces.
- Implementing new queries and transactions was time-consuming and costly.

The World After Codd

- The **relational approach**, as proposed by Codd, separates data representation from its implementation.
- Relational database management systems (RDBMS) hide all implementation details, including data storage.
- The query language is independent of how data is stored, ensuring changes in physical database structure don't affect data queries.

Organization in RDBMS

- In RDBMS, everything is organized neatly and logically.
- Data can be defined and structured within a set of tables that establish relationships between them.



Hypothetical Relational Database Model

PubID	Publisher	PubAddress
03-4472822	Random House	123 4th Street, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920686	City Lights Books	99 Market, San Francisco

AuthorID	AuthorName	AuthorBDay
345-28-2938	Haile Selassie	14-Aug-92
392-48-9965	Joe Blow	14-Mar-15
454-22-4012	Sally Hemmings	12-Sept-70
663-59-1254	Hannah Arendt	12-Mar-06

ISBN	AuthorID	PubID	Date	Title
1-34532-462-1	345-28-2938	03-4472822	1990	Cold Fusion for Dummies
1-38482-995-1	392-48-9965	04-7733903	1985	Macrame and Straw Tying
2-35921-499-4	454-22-4012	03-4859223	1952	Fluid Dynamics of Aquaducts
1-38278-293-4	663-59-1254	03-3920686	1967	Beads, Baskets & Revolution

Features of RDBMS

- RDBMS provides a robust set of features:
  - **Relational model with schemas**, offering a mathematical foundation for data representation and querying.
  - **Powerful and flexible query language** known as Structured Query Language (SQL), capable of supporting both ad hoc and predefined queries.
  - **Transactional semantics**, ensuring Atomicity, Consistency, Isolation, and Durability (ACID) of data operations.
  - **Rich ecosystem** with extensive tool support for various database management tasks.
  - Typically **scales up vertically** rather than horizontally, making it suitable for handling growing data volumes within a single server.

Popular RDBMSs

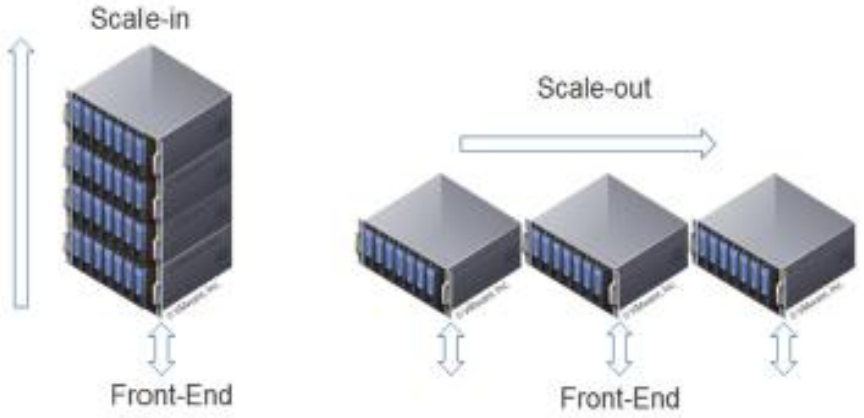
- Some of the most widely used RDBMSs include:
  - Oracle
  - MySQL
  - Microsoft SQL Server
  - PostgreSQL
  - IBM DB2
- RDBMSs held the title of being the "King of the World" in the database world for several decades.
- They dominated the industry for approximately 30 to 40 years, beginning around 1970.

Big Data Challenges

- The rise of Big Data brought new challenges to the forefront.
- The Internet facilitated the collection of vast amounts of data, leading to an explosion in data volume.
- **Three primary characteristics define Big Data**:
  - ***Volume***: The sheer amount of data generated and collected.
  - ***Variety***: The diverse types and sources of data, including structured, semi-structured, and unstructured data.
  - ***Velocity***: The speed at which data is generated, processed, and analyzed.
- This influx of data and its varied nature prompted a new generation to approach data in innovative ways, asking new questions and seeking novel insights.

Scaling Strategies

- To increase system capacity, organizations typically employ two main scaling strategies:
- **Vertical Scaling (Scaling Up)**:
  - Involves upgrading existing hardware components such as CPU and RAM to increase the capacity of a single machine.
- **Horizontal Scaling (Scaling Out)**:
  - Involves adding more machines to distribute the workload across multiple nodes in a network.



- Each scaling strategy has its advantages and considerations, and the choice between them depends on factors such as cost, performance requirements, and scalability goals.

Database Scaling Strategies

- As data volumes or the number of parallel transactions increase, database capacity can be expanded using two main scaling strategies:
- Vertical Scaling (Scaling Up):
  - Involves increasing the storage capacity, memory, and/or CPU power of the database server.
- Horizontal Scaling (Scaling Out):
  - Involves deploying multiple database servers arranged in a cluster to distribute the workload.
  - Each server in the cluster handles a portion of the database operations, allowing for parallel processing and improved scalability.

Database Scaling Strategies (cont.)

- Considerations:
  - Vertical scaling with custom CPUs can be expensive, and there are hardware limitations.
  - Traditional RDBMSs may not scale well horizontally due to limitations in distributed architectures.
  - Distributed RDBMSs may prioritize consistency, which could impact availability if individual servers (nodes) experience issues or fail to communicate.

RDBMS Consistency and Schema Compliance

- RDBMSs place significant emphasis on data consistency and adherence to a formal database schema.
  - Formal Database Schema:
    - RDBMSs require a structured database schema to define the organization and structure of data.
    - New data entries or modifications must conform to this schema in terms of data types, referential integrity, and other constraints.
  - Transactional Consistency:
    - RDBMSs ensure transactional consistency through mechanisms such as ACID properties (Atomicity, Consistency, Isolation, Durability).
    - This ensures that the database remains consistent at all times, even in the event of system failures or interruptions.

RDBMS Consistency and Schema Compliance

- RDBMSs place significant emphasis on data consistency and adherence to a formal database schema.
- While consistency is generally desirable, strict adherence to consistency standards can introduce overhead and limit scalability and flexibility.
- The focus on maintaining consistency may sometimes hinder the system's ability to scale efficiently and adapt to changing requirements.

Simplified Data Operations in Big Data Settings

- In many Big Data environments, the complex querying capabilities offered by traditional databases may not be necessary.
- Instead, the primary requirement is often the ability to efficiently store and retrieve large volumes of data with simple "put" and "get" operations.
- Big Data scenarios often involve semi-structured or highly volatile data types such as sensor data, images, or audio data, where the data's structure may evolve rapidly or lack a predefined schema altogether.
- Traditional relational database management systems (RDBMSs) with rigid schemas can be limiting in such environments.

NoSQL Databases: Meeting the Demands of Modern Data

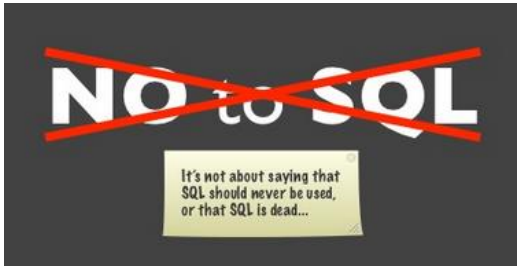
- In situations where massive data volumes, flexible data structures, scalability, and availability take precedence over strict consistency and relational models, traditional database management systems may not suffice.
- What if sacrificing consistency for scalability is acceptable?
- What if flexibility in data modeling is preferred over rigid relational schemas?
- What if cost-effectiveness is a priority?
- The answer to these questions lies in **NoSQL** databases, which provide solutions tailored to these specific needs.

Outline

- The NoSQL Movement
- **What is NoSQL?**
  - Features of NoSQL
  - CAP Theorem, Base Principle
  - NoSQL vs. Relational Databases
- NoSQL Categories

Introduction to NoSQL

- NoSQL stands for "**Not Only SQL**."
  - The movement could have been more accurately termed NoREL (not relational).



- Origin:
  - The term "NoSQL" was initially used by Carl Strozzi in 1998 to name his file-based database.
- Revival:
  - In the late 2000s, the term experienced a resurgence when Eric Evans organized an event to discuss open-source distributed databases.
  - This event sparked renewed interest in alternative database models beyond traditional relational databases.

The Emergence of NoSQL

- Eric Evans emphasizes that the pursuit of alternatives to relational databases arises from the need to address problems for which relational databases are ill-suited.
- Situations where massive data volumes, flexible data structures, scalability, and availability are paramount call for different systems.
- This need led to the rise of NoSQL databases, which offer alternative approaches to data storage and management.
- While relational databases (RDBMSs) remain the preferred choice for storing medium-sized volumes of highly structured data, with a focus on consistency and extensive querying capabilities.

Scalability and Adaptability of NoSQL

- NoSQL databases are engineered to leverage servers within a cluster with minimal administrative intervention.
- With the addition or removal of servers, the NoSQL database system dynamically adapts to utilize the updated set of available servers.
- This inherent scalability and adaptability make NoSQL databases well-suited for environments where workload demands fluctuate or grow rapidly.

High Availability in NoSQL Systems

- In traditional single-server databases, if the server fails, the application becomes unavailable unless a backup server is in place.
- In a NoSQL system, if one server fails or is undergoing maintenance, the remaining servers in the cluster can handle the entire workload.
- While performance might slightly degrade due to the increased load on remaining servers, the application remains available, ensuring continuous service delivery.

Flexibility in NoSQL Databases

- In relational databases, programmers typically need to define all tables and columns at the beginning of a project, assuming that most columns will be needed by most rows.
- NoSQL databases, however, offer more flexibility in terms of schema design.
- Some NoSQL databases do not require a fixed table structure, allowing for dynamic schema changes as the application evolves.
- This flexibility can accommodate diverse data types and evolving requirements more effectively.

Open Source Nature of NoSQL Databases

- Major NoSQL databases are often available as open source software, offering users the freedom to use them on as many servers as needed without licensing fees.
  - Open source developers typically do not charge fees to run their software, making it accessible to businesses of all sizes.
- Third-party companies often provide commercial support services for open source NoSQL databases, ensuring businesses have access to software support similar to commercial relational databases.

Computing Platforms of NoSQL Databases

- NoSQL databases are often designed to operate across multiple servers, forming distributed systems, although this is not always mandatory.
  - Implementing a NoSQL database across multiple servers becomes essential when scalability and availability are primary concerns.
- When designing NoSQL databases and associated applications, it's crucial to balance various factors such as scalability, availability, consistency, partition tolerance, and durability. Consider how each aspect aligns with your specific requirements and priorities.

Summary: Non-Relational Databases (NoSQL)

- Non-relational databases, commonly known as NoSQL databases, store and manage data in formats other than tabular relations.
- NoSQL databases offer flexible schemas, eliminating the need for predefined structures and allowing for dynamic data models.
- They provide simple call interfaces, making them user-friendly and accessible for developers.
- NoSQL databases excel in distribution, efficiently utilizing distributed indexes and RAM to enhance performance.
- They are horizontally scalable, meaning they can handle increased workloads by adding more nodes to the database cluster.
- NoSQL databases are often cost-effective and straightforward to implement, with many open-source options available in the market.

Limitations of NoSQL Databases

- NoSQL databases do not fully support traditional relational features like join, group by, or order by operations, except within partitions or limited contexts.
- They lack referential integrity constraints across partitions, which may lead to data consistency issues in distributed environments.
- NoSQL databases typically do not offer a declarative query language like SQL, requiring developers to rely more on programming for data manipulation.
- Integration with other applications that support SQL may not be as straightforward with NoSQL databases, posing challenges for interoperability.

Outline

- The NoSQL Movement
- What is NoSQL?
  - Features of NoSQL
  - ☞ **CAP Theorem, Base Principle**
  - NoSQL vs. Relational Databases
- NoSQL Categories

Objectives of DBMS and NoSQL

- The main objectives of a Database Management System (DBMS) are to store data persistently, maintain data consistency, and ensure data availability.
- Traditional relational databases enforce ACID (Atomicity, Consistency, Isolation, Durability) properties to support applications like finance and e-commerce, where maintaining consistency is crucial, even if it impacts speed.
- NoSQL databases, however, provide a weaker concurrency model compared to relational databases.
- They prioritize fast database operations over strict consistency, adhering to the BASE (Basically Available, Soft state, Eventually consistent) principle rather than ACID.

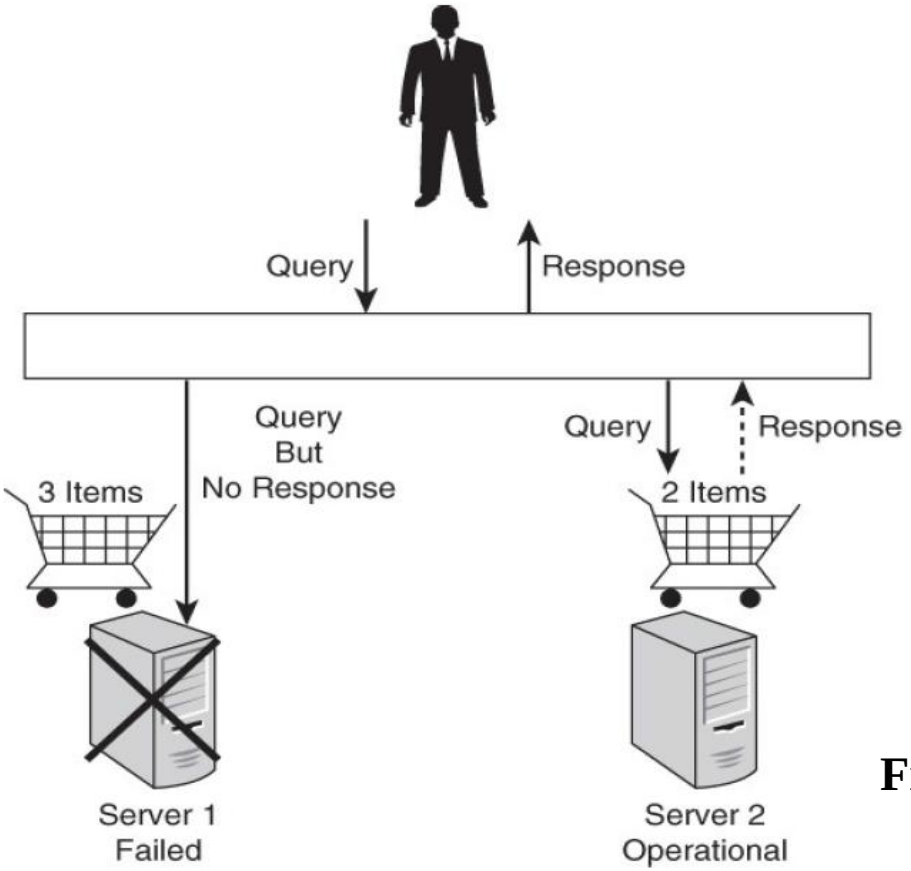
Things to Give Up (Cont.)

- Main objectivities of DBMS are (1) Store data persistently, (2) Maintain data consistently, (3) Ensure data availability
- However, NoSQL provides weaker concurrency model than traditional relational databases which enforce ACID (Atomicity, Consistency, Isolation, Durability) to support applications (e.g., finance, e-commerce), in which the fast database operations are more important than maintaining consistency at all times.
  - Relaxed ACID (CAP theorem) → BASE Principle

CAP Theorem and Distributed Databases

- The CAP theorem states that in a distributed computer system, it's impossible to simultaneously guarantee Consistency, Availability, and Partition tolerance (CAP).
  - **Consistency** refers to all nodes seeing the same data simultaneously,
  - **Availability** ensures every request receives a response indicating success or failure, and
  - **Partition tolerance** means the system continues to function even if some nodes fail.
- Distributed databases must sacrifice one of these properties. They can't guarantee all three simultaneously.
- In practice, designers often face trade-offs between consistency and availability rather than partition tolerance issues.

Trade-off between Availability and Consistency



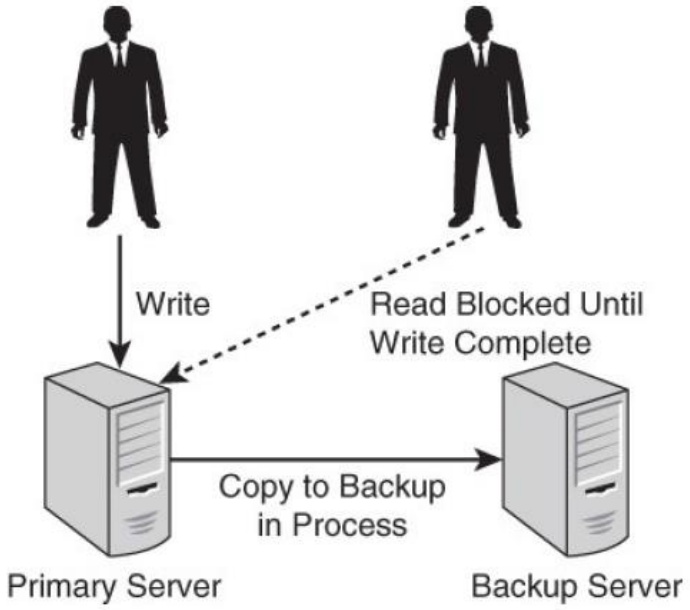


Figure. Data can be consistent but not available

Figure. Data can be available but not consistent.

BASE Principle in NoSQL Databases

- Most NoSQL databases adhere to the BASE principle, sacrificing consistency for availability and partition tolerance.
- **BASE** stands for:
  - **Basically Available**: The system remains operational even in the face of partial failures.
  - **Soft State**: The system can change over time without input, as nodes continue to update each other.
  - **Eventually Consistent**: The system will become consistent over time but may not be consistent at any particular moment.
- These properties are common in NoSQL databases, allowing them to handle large-scale, distributed data with high availability.

Basically Available in NoSQL Databases

Basically Available:

- NoSQL databases adhere to the **availability** guarantee of the CAP theorem.
- In distributed systems, partial failures may occur in certain parts, but the system as a whole continues to function.
- NoSQL databases often maintain multiple copies of data across different servers to ensure redundancy.
- This redundancy allows the database to respond to queries even if one of the servers experiences a failure, ensuring continuous availability of the system.

Soft State in NoSQL Databases

Soft State:

- Soft state refers to the dynamic nature of a system, where changes can occur over time without requiring external input.
- In distributed systems, nodes continue to update each other autonomously, leading to constant changes in the system's state.
- In NoSQL operations, soft state implies that data may be overwritten with more recent updates over time, reflecting the system's evolving state.
  - This concept aligns with the third property of BASE transactions, known as eventual consistency, where data eventually converges to a consistent state across all nodes.

Eventually Consistent in NoSQL Databases

Eventually Consistent:

- Eventually consistent refers to the property of a system where data consistency is achieved over time but might not be guaranteed at any specific moment.
- In distributed databases, there may be intervals where the data is in an inconsistent state across nodes.
  - In NoSQL databases with replication, multiple copies of data are stored on different servers. However, updates to one copy may not immediately propagate to all other copies.
  - During this period of inconsistency, some copies may have the updated data while others retain the old version.
  - Eventually, through the replication mechanism, all copies of the data will synchronize, ensuring eventual consistency across the system.

Summary: Relational Database vs. NoSQL Database

	Relational Databases	NoSQL Databases
Data paradigm	Relational tables	Key-value based, Document based, Column based Graph based, XML, object based Others: time series, probabilistic, etc.
Distribution	Single-node and distributed	Mainly distributed
Scalability	Vertical scaling, harder to scale horizontally	Easy to scale horizontally, easy data replication
Openness	Closed and open source	Mainly open source
Schema role	Schema-driven	Mainly schema-free or flexible schema
Query language	SQL as query language	No or simple querying facilities, or special-purpose languages
Transaction mechanism	ACID: Atomicity, Consistency, Isolation, Durability	BASE: Basically available, Soft state, Eventual consistency
Feature set	Many features (triggers, views, stored procedures, etc.)	Simple API
Data volume	Capable of handling normal-sized data sets	Capable of handling huge amounts of data and/or very high frequencies of read/write requests

Outline

- The NoSQL Movement
- What is NoSQL?
- **NoSQL Database Categories**
  - Key-Value Stores
  - Document-oriented Stores
  - Column-oriented Databases
  - Graph based Databases

NoSQL Database Implementations (Products)

Categories of NoSQL Solutions

- NoSQL solutions fall into two major categories:
  - **Key/Value** or 'the big hash table'.
    - Redis, Amazon Dynamo, Memcached (in-memory key/value store), Riak, Voldemort, Scalaris

  - **Schema-less** which comes in multiple flavors:
    - Column-based (Cassandra, HBase)
    - Document-based (CouchDB, MongoDB)
    - Graph-based (Neo4J)

Major Categories of NoSQL Solutions

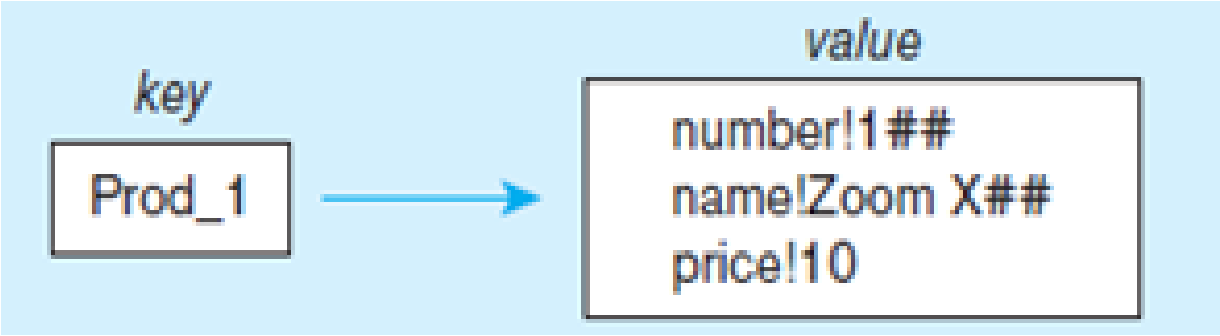
- NoSQL solutions can be also divided into four major categories:
  - **Key-value stores**
  - **Document-oriented stores (Document databases)**
  - **Column-oriented databases (Column family databases)**
  - **Graph-based databases (Graph databases)**

Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - **Key-Value Stores**
    - Basic Concepts of Key-Value Stores
    - Key-Value Database Model
    - Scalability
  - Document-oriented Stores
  - Column-oriented Databases
  - Graph based Databases

Key-Value Stores

- **Key-value stores** are the simplest of the NoSQL databases
- Data is stored as **(key, value) pairs**.
  - **Keys** are unique identifiers associated with **values**.
  - Values can be of any data type and length.
- Data records are stored and retrieved **using keys**



Example of Key-Value Pairs

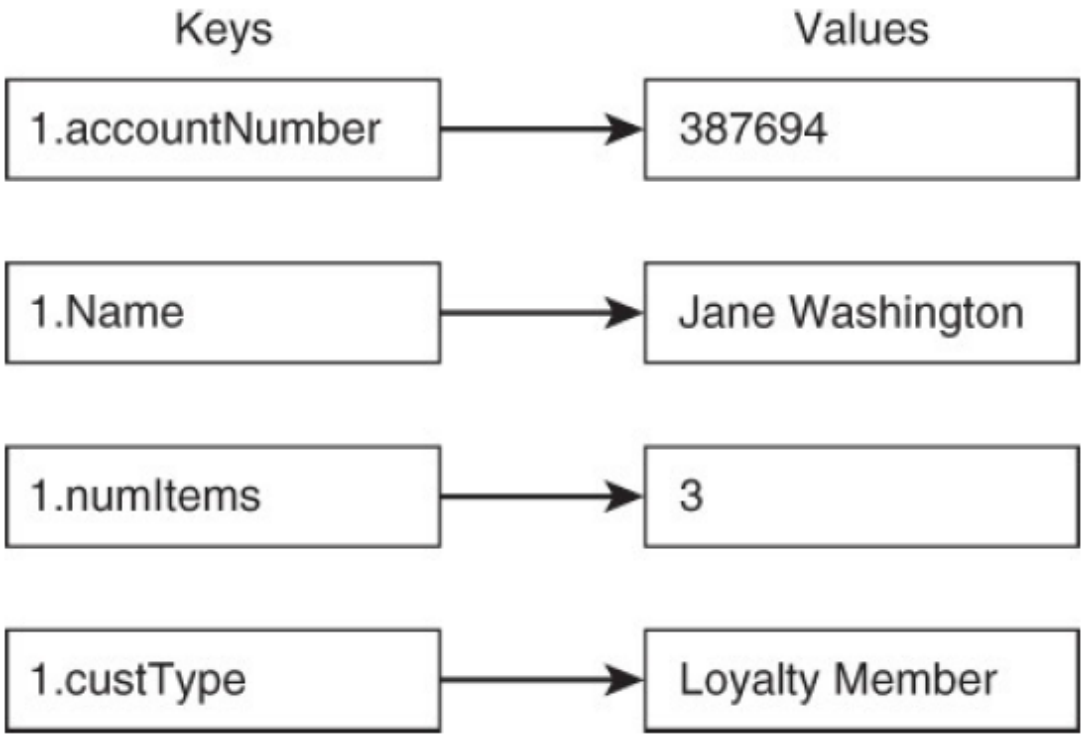


Figure. Key-value stores are modeled on a simple, two-part data structure consisting of an identifier and a data value.

Constructing Keys in Key-Value Stores

- The formula for constructing a key can vary depending on the specific system and the preferences of the developer or the requirements of the application.
  - You can use meaningful names that include information about entity types, entity identifiers, and entity attributes.

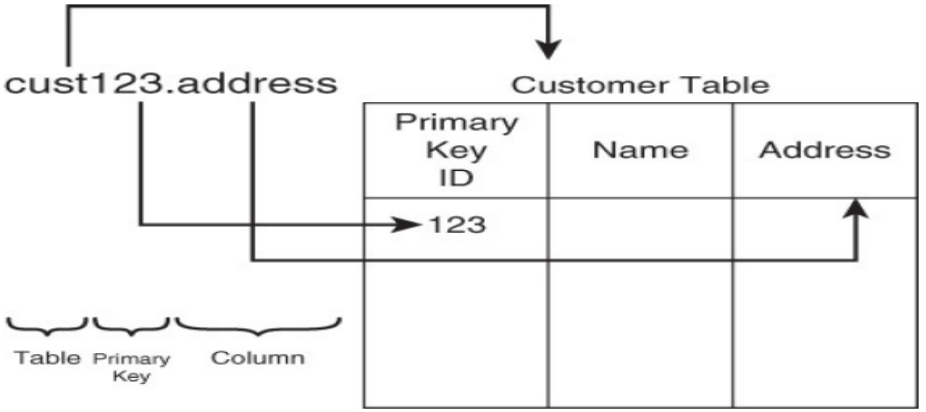


Figure. The key-naming convention maps to patterns seen in relational database tables.

- Examples:
  - **Entity Name + Entity Identifier + '.' + Entity Attribute**
    - E.g., cust123.address
  - **Entity Name + ':' + Entity Identifier + ':' + Entity Attribute**
    - E.g., cust:123:address

Examples of Keys

- For the customer data
  - cust1.accountNumber
  - cust1.name
  - cust1.address
  - cust1.numItems
  - cust1.custType
  - cust2.accountNumber
  - cust2.name
  - cust2.address
  - cust2.numItems
  - cust2.custType
- For the warehouse data
  - wrhs1.number
  - wrhs1.address
  - wrhs2.number
  - wrhs2.address

Values in Key-Value Stores

- **Values** in key-value stores can store various types of data, offering flexibility in data representation:
  - **String** format: e.g., ‘1232 NE River Ave, St. Louis, MO’
  - **List** format : e.g., (‘1232 NE River Ave’, ‘St. Louis’, ‘MO’)
  - **Structured** format using JavaScript Object Notation (JSON): e.g., { ‘Street:’ : ‘1232 NE River Ave’, ‘City’ : ‘St. Louis’,: ‘State’ : ‘MO’ }
- Key-value databases do not require you to specify data types for the values you store, allowing for versatile data storage.

Namespaces in Key-Value Stores

- A **namespace** in a key-value store is a collection of key-values pairs that ensures uniqueness of keys within it.

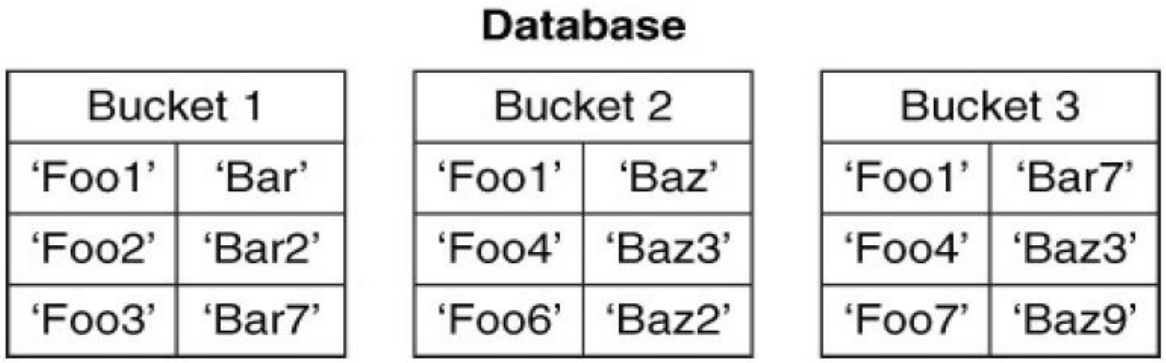


Figure. Keys of a key-value database must be unique within a namespace

Namespaces in Key-Value Stores (cont.)

- A key-value store may have many different namespaces.
- Each namespace maintains its own set of key-value pairs, ensuring that keys are unique within that namespace.

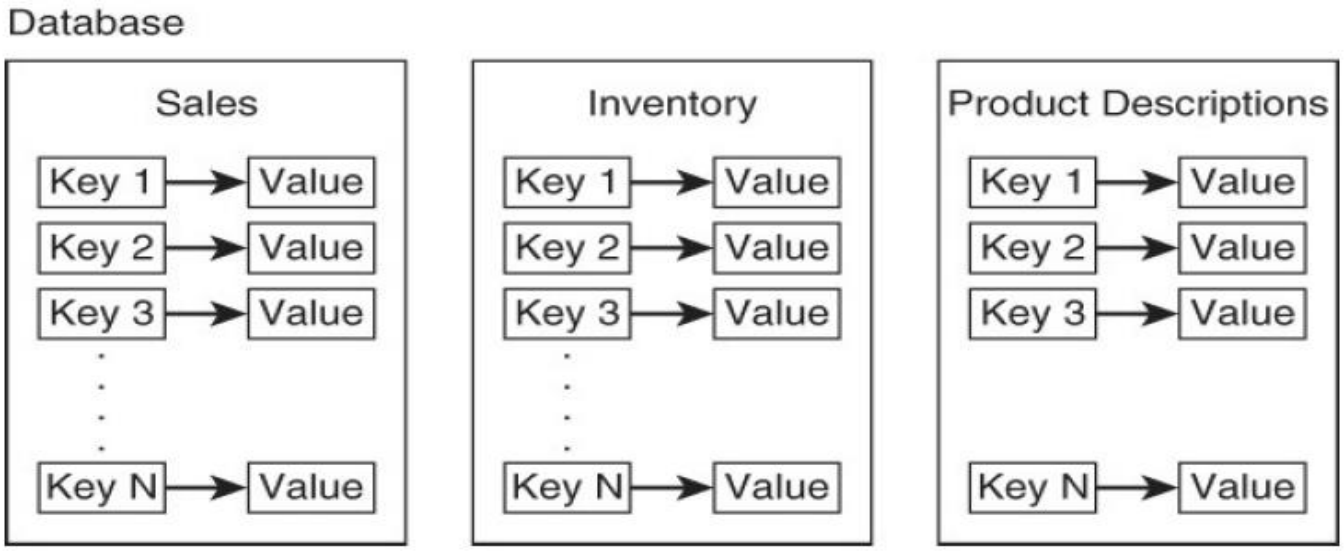
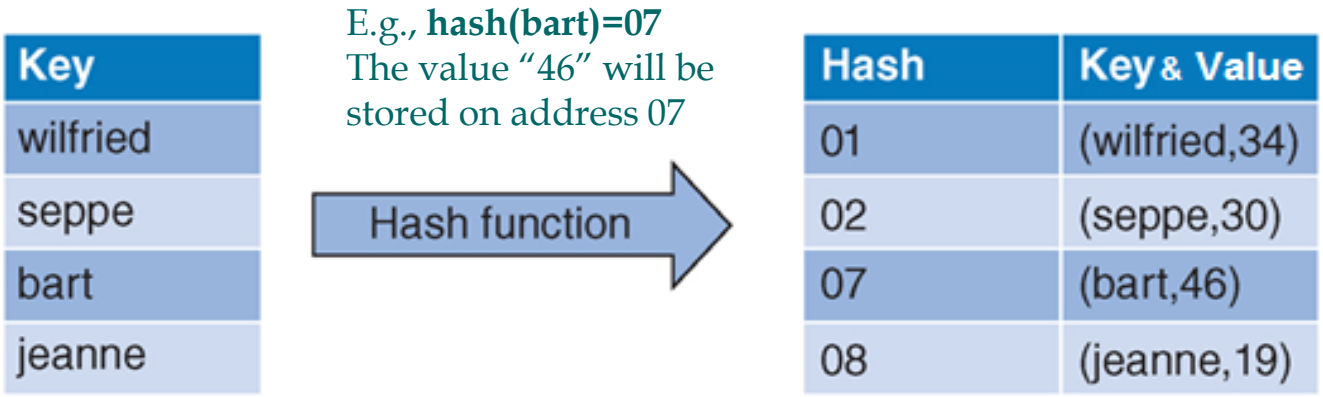


Figure. Keys of a key-value database must be unique within a namespace

Hashing in Key-Value Stores

- In key-value stores, efficient key search is facilitated by **hashing**, which involves the use of a hash function.
- A **hash function** takes an arbitrary value of arbitrary length and maps it to a key with a fixed length, known as the **hash value** or **hash code**.



Hashing Function

- **Properties of a good hash function**
  - **Deterministic**: Hashing the same input value must always provide the same hash value
  - **Uniform**: A good hash function should evenly distribute the inputs across its output range
  - **Defined size**: It is desirable that the output of a hash function has a fixed size to ensure consistency and efficiency in key storage and retrieval.

Example: Key to Hash Value Mappings

- Application code in an in-memory cash for customer shipping information:

```
customerCache['1982737:firstname'] = firstName
customerCache['1982737:lastname'] = lastName
customerCache['1982737:shippingAddress'] = shippingAddress
customerCache['1982737:shippingCity'] = shippingCity
customerCache['1982737:shippingState'] = shippingState
customerCache['1982737:shippingZip'] = shippingZip
```

- In a key-value store, customer shipping information may be mapped to hash values for keys, such as:

Key	Hash Value
customer:1982737: firstName	e135e850b892348a4e516cfcb385eba3bfb6d209
customer:1982737: lastName	f584667c5938571996379f256b8c82d2f5e0f62f
customer:1982737: shippingAddress	d891f26dcdb3136ea76092b1a70bc324c424ae1e
customer:1982737: shippingCity	33522192da50ea66bfc05b74d1315778b6369ec5
customer:1982737: shippingState	239ba0b4c437368ef2b16ecf58c62b5e6409722f
customer:1982737: shippingZip	814f3b2281e49941e1e7a03b223da28a8e0762ff

Operations in Key-Value Stores

- In key-value stores, all operations on values are based on keys:
  - Retrieve a value by key
  - Set a value by key
  - Delete values by key
- Key-value stores do not inherently support querying based on the values within the data.
  - E.g., If you want to search for specific data items, such as locating an address where the city is “Fort Wayne,” you will typically need to handle this logic within your application’s code.

Searching for Values

- Some key-value stores integrate search functionality directly into the database.
- **Example**: The search system maintains a list of words along with the keys of each key-value pair in which that word appears.

Word	Keys
'IL'	'cust:2149:state', 'cust:4111:state'
'OR'	'cust:9134:state'
'MA'	'cust:7714:state', 'cust:3412:state'
'Boston'	'cust:1839:address'
'St. Louis'	'cust:9877:address', 'cust:1171:address'
	. . . .
'Portland'	'cust:9134:city'
'Chicago'	'cust:2149:city', 'cust:4111:city'

Background: Array vs. Associative Array

- **Array**
  - An array is a fundamental data structure consisting of an ordered list of elements, where each element is of the same data type.
  - Elements within an array are accessed and manipulated based on their numerical index.

1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	True
9	False
10	True

- **Associate Array**
  - An associative array, also known as a **dictionary** or **map**, is a more flexible data structure.

'Pi'	3.14
'CapitalFrance'	'Paris'
17234	34468
'Foo'	'Bar'
'Start_Value'	1

  - Unlike traditional arrays, associative arrays are not limited to using integers as indexes, nor are they restricted to storing values of the same type. Instead, elements in an associative array are accessed and manipulated using keys, which can be of any data type.

Associative Arrays for Key-Value Databases

- Key-value databases extend the concept of associative arrays, but they incorporate several important distinctions.
- **Storage mechanism** - While associative arrays typically reside in memory and are transient, key-value databases often maintain persistent copies of data on long-term storage mediums, such as hard drives or flash devices.

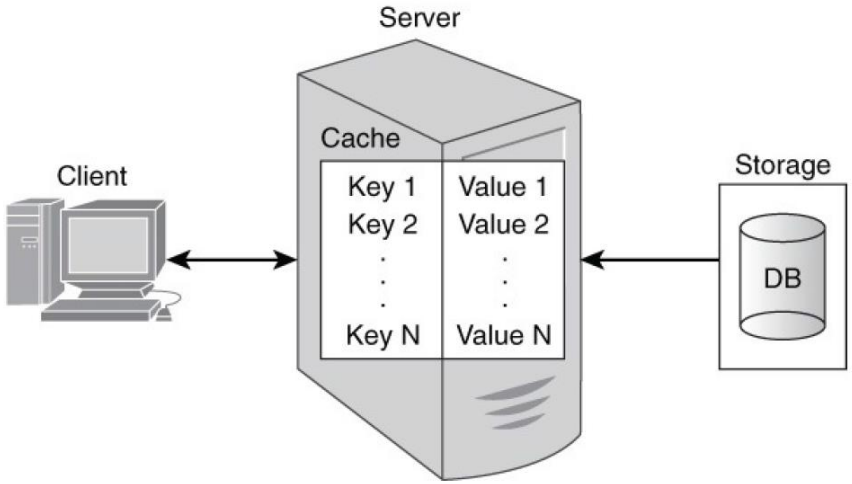


Figure. Caches are associative arrays used by application programs to improve data access performance.

Key-Value Approach in Programming Language

- The key-value approach in programming language maps directly to a data structure such as **hash map**, **hash table** or **dictionary**.
- **Example:**

  The **HashMap** class in Java allows storing a arbitrary objects (in internal memory) based on a single “key”.

```
import java.util.HashMap;
import java.util.Map;
 public class KeyValueStoreExample {
  public static void main(String...  args) {
      // Keep track of age based on name
      Map<String, Integer> age_by_name = new HashMap<>();

      // Store some entries
      age_by_name.put("wilfried", 34);
      age_by_name.put("seppe", 30);
      age_by_name.put("bart", 46);
      age_by_name.put("jeanne", 19);

      // Get an entry
      int age_of_wilfried = age_by_name.get("wilfried");
      System.out.println("Wilfried's age: " + age_of_wilfried);

      // Keys are unique
      age_by_name.put("seppe", 50); }
 }
```

Memcached: A Pioneering NoSQL System

- **Memcached** is a famous, early NoSQL system.
- Memcached is essentially a distributed, in-memory key-value store. It operates as a cache, storing frequently accessed data objects in memory to accelerate read operations.
- While Memcached primarily serves as a caching solution rather than a persistent database, many subsequent NoSQL databases have drawn inspiration from Memcached's API and caching principles to develop persistent key-value stores with enhanced capabilities.

Example of Memcached Program Codes

```
import java.util.ArrayList;
import java.util.List;
import net.spy.memcached.AddrUtil;
import net.spy.memcached.MemcachedClient;

public class MemCachedExample {
  public static void main(String[] args) throws Exception {
    List<String> serverList = new ArrayList<String>() {
      {
        this.add("memcachedserver1.servers:11211");
        this.add("memcachedserver2.servers:11211");
        this.add("memcachedserver3.servers:11211");
      }
    };
```

information of multiple servers

Example: Java program codes with SpyMemcached library

Example of Memcached Program Codes (cont.)

```
MemcachedClient memcachedClient = new MemcachedClient(
        AddrUtil.getAddresses(serverList));
```

Pass the information of multiple server to client

```
// ADD adds an entry and does nothing if the key already exists
// Think of it as an INSERT
// The second parameter (0) indicates the expiration - 0 means no expiry
memcachedClient.add("marc", 0, 34);
memcachedClient.add("seppe", 0, 32);
memcachedClient.add("bart", 0, 66);
memcachedClient.add("jeanne", 0, 19);
memcachedClient.add("marc", 0, 1111); // ADD will have no effect

// SET sets an entry regardless of whether it exists
// Think of it as an UPDATE-OR-INSERT
memcachedClient.set("jeanne", 0, 12); //The value associated with
                                      //the key is changed
```

key

value

Example of Memcached Program Codes (cont.)

```
        // REPLACE replaces an entry and does nothing if the key does not exist
        // Think of it as an UPDATE
        memcachedClient.replace("not_existing_name", 0, 12); //<- no effect
        memcachedClient.replace("jeanne", 0, 10);

        // DELETE deletes an entry, similar to an SQL DELETE statement
        memcachedClient.delete("seppe");

        // GET retrieves an entry
        Integer age_of_marc = (Integer) memcachedClient.get("marc");
        Integer age_of_seppe = (Integer) memcachedClient.get("seppe");

        System.out.println("Age of Marc: " + age_of_marc);
        System.out.println("Age of Seppe (deleted): " + age_of_seppe);
        memcachedClient.shutdown();
    }
}
```

Examples of Key-value Databases

- There is a wide array of key-value databases available, each offering unique features and capabilities to meet diverse needs in the data management landscape.
  - SimpleDB
  - Redis
  - Riak
  - Dynamo
  - Memcached
  - Voldemort
  - Aerospike
  - Oracle Berkely DB
  - IBM Informix C-ISAM
  - HyperDex

Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - **Key-Value Stores**
    - Basic Concepts of Key-Value Stores
    - ☞ **Key-Value Database Model**
    - Scalability
  - Document-oriented Stores
  - Column-oriented Databases
  - Graph based databases

Key-Value Database Model

- **Simple data model** : Key-value Stores offer a straightforward data model, characterized by its minimalistic structure, often referred to as *schemaless*.
- You are not required to define all keys and value types beforehand; You can add them as needed.
- Flexible and forgiving nature.
  - There's no requirement to alter database code to accommodate new attributes. Key-value databases seamlessly handle the addition of new attributes on-the-fly.
  - This adaptability is particularly beneficial when dealing with changing data types or supporting multiple types for the same attribute.
- The syntax for manipulating data is simple.

Example: Simple Model

Key-Value Database	
Keys	Values
cust : 8983 : firstName	'Jane'
cust : 8983 : lastName	'Anderson'
cust : 8983 : fullName	'Jane Anderson'

Figure. Schemaless data models enable various representations of the same data to coexist without the need for a fixed database schema.

Key-Value Database Model

- Key-value databases operate on minimalist principles when it comes to storing and retrieving data.
  - No tables, no columns, no column constraints, no join
  - No relationships – Each data item is stored as an independent key-value pair, with no inherent relationships or dependencies between them.
  - Any constraints or relationships must be managed and enforced entirely at application level.
- Key-value databases does not provide a rich query language like SQL.
  - Instead, operations on data are primarily based on keys, allowing for simple retrieval and manipulation of values associated with those keys.

Data Models in Key-value Databases

Name	Producer	Data model	Querying
SimpleDB	Amazon	set of couples (key, {attribute}), where attribute is a couple (name, value)	restricted SQL; select, delete, GetAttributes, and PutAttributes operations
Redis	Salvatore Sanfilippo	set of couples (key, value), where value is simple typed value, list, ordered (according to ranking) or unordered set, hash value	primitive operations for each value type
DynamoDB	Amazon	like SimpleDB	simple get operation and put in a context
Voldemort	LinkedIn	like SimpleDB	similar to Dynamo
…	…	…	…

Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - **Key-Value Stores**
    - Basic Concepts of Key-Value Stores
    - Key-Value Database Model
    - ☞ **Scalability**
  - Document-oriented Stores
  - Column-oriented Databases
  - Graph based databases

Horizontal Scaling in Key-Value Stores

- Key-value stores excel in horizontal scaling, enabling the distribution of key-value pairs across multiple nodes for enhanced performance and scalability.
- By leveraging horizontal scaling, key-value stores can accommodate growing datasets and increasing workloads, making them well-suited for applications with demanding performance and scalability requirements.

Sharding in Distributed Databases

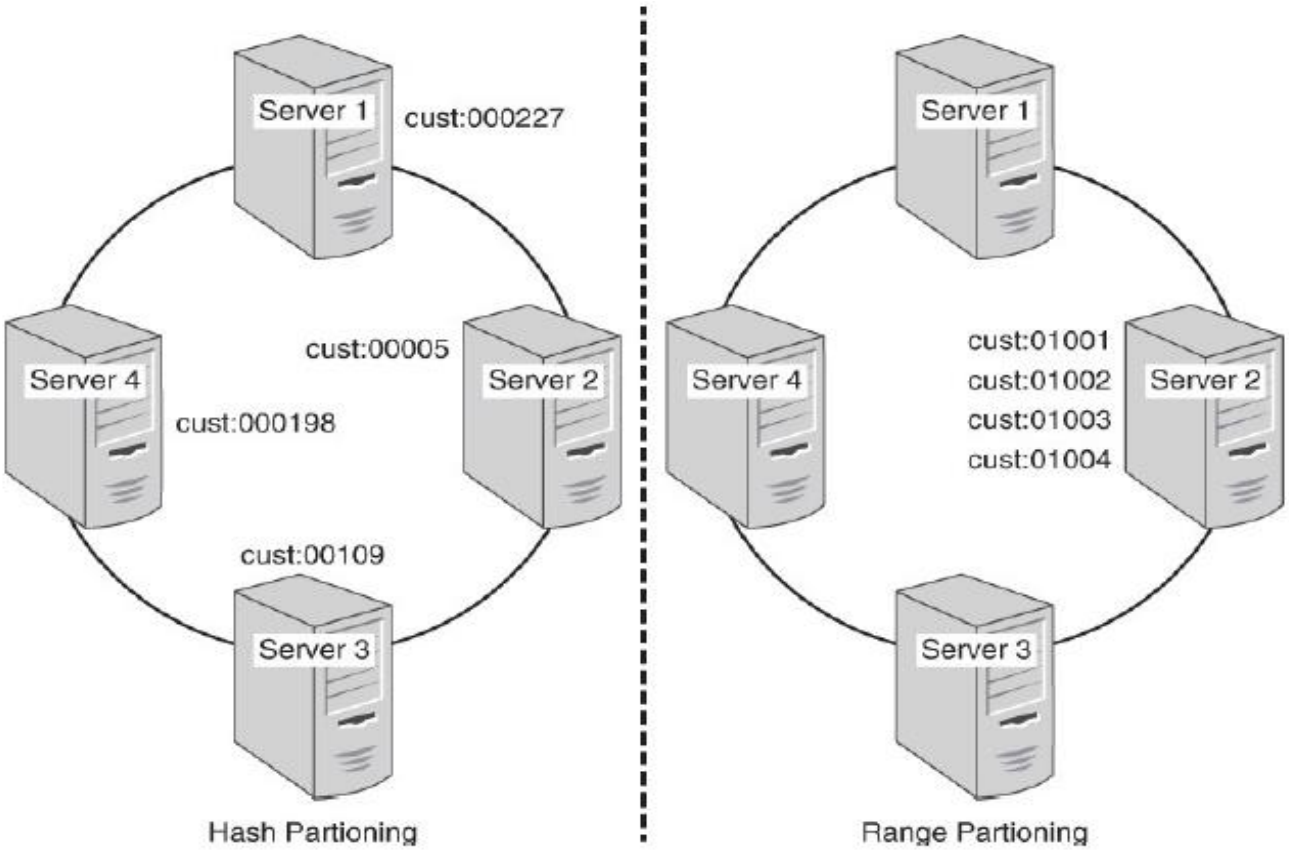
- **Sharding**, or distributing data across multiple nodes, is a fundamental concept in distributed databases.
- Sharding involves partitioning the data into separate sets, with each partition known as a **shard**.
- Each shard is stored on a different node within the distributed database cluster.

Partitioning in Distributed Databases

- **Partitioned Cluster**: In a partitioned cluster, servers or instances of the key-value database software are assigned to manage specific subsets of the database.
- **Partition Key**: A partition key is a crucial component of sharding. It's a key used to determine which partition, or shard, should hold a particular data value.
- **Choosing Partition Keys**: Selecting effective partition keys is essential for balanced workload distribution.
  Good partition keys evenly distribute data across shards, preventing hotspots and ensuring optimal performance across the distributed system.

Key-Based Partitioning Strategies

- How keys are used in partitioning
  - **Hash partitioning** vs **Range partitioning**



Key-Based Partitioning Strategies

- **Hash Partitioning**
  - In hash partitioning, a hash function is applied to the partition key to determine the shard where the data will be stored.
  - The hash function generates a unique hash value for each key, ensuring an even distribution of data across shards.
  - This approach is efficient for distributing data randomly and preventing hotspots.
- **Range Partitioning**
  - Range partitioning involves dividing data based on specific ranges of values in the partition key.
    - For example, you might partition data based on timestamps, customer IDs, or geographical regions. Each shard is responsible for storing data within a specific range of values.
  - Range partitioning offers more control over data placement and can be beneficial for certain types of queries.

Hashing for Data Distribution

- Hashing plays a crucial role in distributing key-value pairs across nodes in a distributed database.
- **Partitioning Key-Value Pairs**: Nodes in the network can partition hashed key-value pairs among them using a hash function. This function takes the key as input and outputs the index of the node where the corresponding value will be stored.
- **Choosing the Server**: Similarly, a hash function can determine which server in a cluster should store the value associated with a given key.
  - For example, a simple hash function might be **h(key) = key % *n***, where *n* is the number of servers in the cluster.
- The primary goal of hashing in this context is to balance the distribution of keys across the available servers.

Example: Distribution using Modulo Hash Function

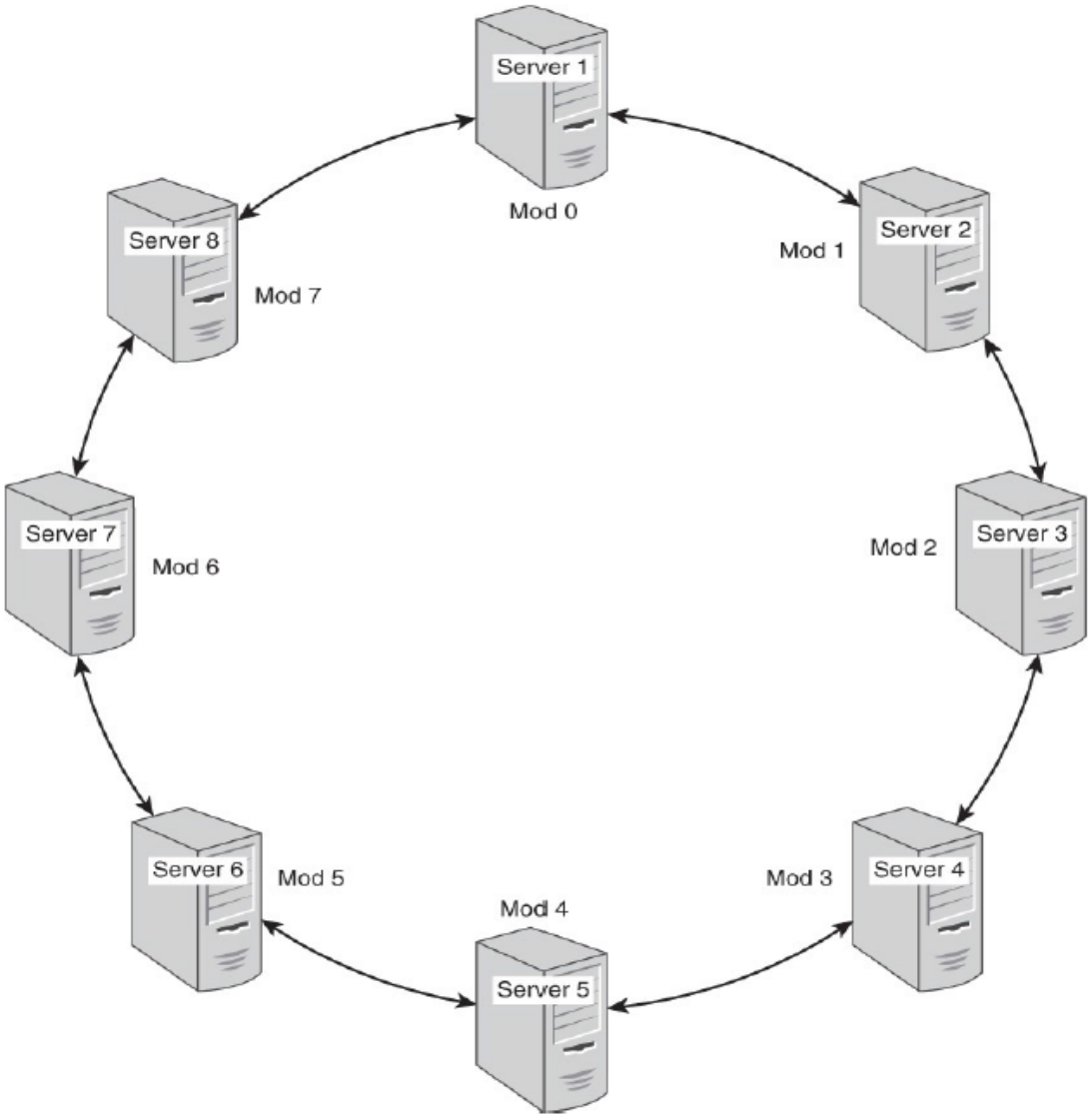


Figure An eight-server cluster in a ring configuration with modulo number assigned.

Summary of Key-Value Stores

- Key Features:
  - **Efficiency and Scalability**: Key-value stores excel in providing an efficient and scalable environment for storing and retrieving disparate data elements. Their simple model allows for the addition of new key-value pairs with ease.
  - **Speed**: Leveraging hash functions for data access enables key-value stores to achieve exceptional speed, particularly when retrieving data by key. However, performance may degrade when hash functions are not utilized optimally.
  - **Scalability**: Key-value stores are highly scalable and support horizontal distribution, making them suitable for handling large volumes of data across multiple nodes in a cluster.
  - **Foundation for Complex Systems**: They often serve as foundational layers for systems with more sophisticated functionalities, providing a robust infrastructure for building complex applications.

Summary of Key-Value Stores (cont.)

- Limitations:
  - **Limited Search Capabilities**: The primary mode of data retrieval in key-value stores is by key. This limitation restricts the ability to search for values based on criteria other than the key.
  - **Absence of Range Queries**: Some key-value databases lack support for range queries, which can be essential for querying data within specific ranges or intervals.
  - **Lack of Standard Query Language**: Unlike relational databases that utilize SQL for querying data, key-value stores typically lack a standardized query language. Developers must implement custom logic for data retrieval and manipulation.