NoSQL Database: An Introduction

Abu Raihan Mostofa Kamal

Professor, CSE Department
Islamic University of Technology (IUT)

March 18, 2025

Content Outline

[1]The course materials have been prepared based on a number of textbooks including NoSQL for Mere Mortals by Dan Sullivan

Relational Databases: Limitations

- Relational databases have been the dominant type of database used for database applications **for decades**.

Relational Databases: Limitations

- Relational databases have been the dominant type of database used for database applications **for decades**.
- Relational databases **addressed many of the limitations** of flat file–based data stores, hierarchical databases, and network databases.

Relational Databases: Limitations

- Relational databases have been the dominant type of database used for database applications **for decades**.
- Relational databases **addressed many of the limitations** of flat file–based data stores, hierarchical databases, and network databases.
- However, due to **internet** the nature of business applications has changed significantly. And Relational DB losses its suitability in many cases.

Relational Databases: Limitations

- Relational databases have been the dominant type of database used for database applications **for decades**.
- Relational databases **addressed many of the limitations** of flat file–based data stores, hierarchical databases, and network databases.
- However, due to **internet** the nature of business applications has changed significantly. And Relational DB losses its suitability in many cases.
- Companies such as Google, LinkedIn and Amazon found that **supporting large numbers of users** on the Web was **different** from supporting much smaller numbers of business users.

Relations Database: Limitations (Cont.)

Today, Web Application deals with large volumes of data and extremely large numbers of users.

- Large volumes of read and write operations

Relations Database: Limitations (Cont.)

Today, Web Application deals with large volumes of data and extremely large numbers of users.

- Large volumes of read and write operations
- Low latency response times

Relations Database: Limitations (Cont.)

Today, Web Application deals with large volumes of data and extremely large numbers of users.

- Large volumes of read and write operations
- Low latency response times
- High availability

No SQL: Why?

Web applications serving **millions of users** or more are difficult to implement with relational databases.

4 desired characteristics for large-scale data management:

1. Scalability
2. Cost
3. Flexibility
4. Availability

No SQL: Scalability

Scalability is the ability to efficiently meet the needs for varying workloads.
2 options are there:

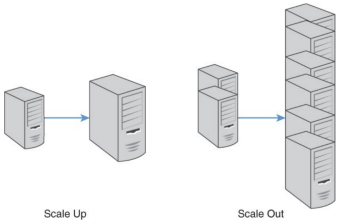
1. **Scale up.** Database administrators could choose to scale up, which is **upgrading an existing database server to add additional** processors, memory, network bandwidth, or other resources that would improve performance on a database management system or replacing an existing server with one with more CPUs, memory, and so on.

No SQL: Scalability

Scalability is the ability to efficiently meet the needs for varying workloads.
2 options are there:

1. **Scale up.** Database administrators could choose to scale up, which is **upgrading an existing database server to add additional** processors, memory, network bandwidth, or other resources that would improve performance on a database management system or replacing an existing server with one with more CPUs, memory, and so on.
2. **Scale out.** If there is a spike in traffic to a website, **additional servers can be brought online** to handle the additional load. When the spike subsides and traffic returns to normal, some of those additional servers can be shut down. Adding servers as needed is called scaling out.

No SQL: Scale up and Scale out



- Scale up in Relational DB is possible, but incurs cost.
- Scale out in Relational DB is hard to ensure, because Transaction (i.e. ACID) management in multiple servers is challenging.
- Scaling out is more flexible than scaling up. Servers can be added or removed as needed when scaling up.
- NoSQL databases are designed to utilize servers available in a cluster with minimal intervention by database administrators.

No SQL: Cost

The cost of database licenses is an obvious consideration for any business or organization.

- **Proprietary Database:** Web applications may have spikes in demand that increase the number of users utilizing a database at any time. Should users of the RDBMS pay for the number of peak users or the number of average users? How should they budget for RDBMS licenses?
- **Open Source Database:** The software is free to use on as many servers of whatever size needed because open source developers do not typically charge fees to run their software. Fortunately for NoSQL database users, the major NoSQL databases are available as open source.

No SQL: Flexibility

- Relational Database has the flexibility in its design phase (i.e. suitable for a wide array of applications). But it **lacks flexibility** in many cases as well.
- **For instance:** An e-commerce application where products are sold. Each product has different set of attributes. **For example**, PC has its CPU, RAM, Display while a Photocopier has its size, capacity, speed of copying etc.
  - A database designer could create separate tables for each type of product or define a table with as many different product attributes as she could imagine at the time she designs the database.
  - Unlike relational databases, some NoSQL databases do not require a fixed table structure.

No SQL: Availability

- Today we need systems/applications live for 24x7. If your favorite social media or e-commerce site were frequently down when you tried to use it, you would likely start looking for a new favorite.
- NoSQL databases are designed to take advantage of multiple, low-cost servers. When one server fails or is taken out of service for maintenance, the other servers in the cluster can take on the entire workload.

Distributed Databases

- Recall the motivations for NoSQL, including the need for **scalability, flexibility, cost control, and availability**.

Distributed Databases

- Recall the motivations for NoSQL, including the need for **scalability, flexibility, cost control, and availability**.
- A common way to meet these needs is by designing data management systems to work across **multiple servers**, that is, as a distributed system.

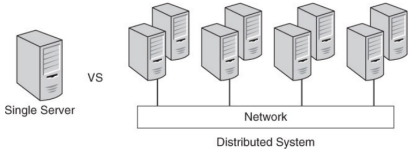
Distributed Databases

- Recall the motivations for NoSQL, including the need for **scalability, flexibility, cost control, and availability**.
- A common way to meet these needs is by designing data management systems to work across **multiple servers**, that is, as a distributed system.
- NoSQL offers some level of **operational simplicity**

> You can add and remove servers as needed rather than adding or removing memory, CPUs, and so on from a single server. Also, some NoSQL databases include features that automatically detect when a server is added or removed from a cluster.

Objectives of Distributed Databases

NoSQL often deploy Distributed Database Architecture.



The database management systems must do three things:

i. Store data persistently
ii. Maintain data consistency
iii. Ensure data availability

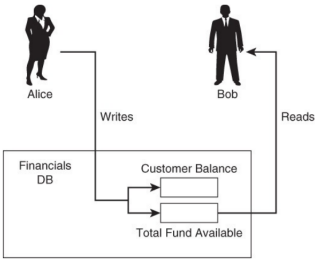
Store Data Persistently

- Data must be stored persistently; that is, it must be stored in a way that data is **not lost** when the database **server is shut down**.
- If data were only stored in memory—that is, RAM — then it would be lost when power to the memory is lost. Only data that is stored on disk, flash, tape, or other **long-term storage** is considered persistently stored.
- Although read/write operation in long-term storage is much slower in comparison to RAM, the response time is minimized using a number of techniques such as **Indexing**.

Maintain Data Consistency

- Data **must not be contradictory** as per the business logic.
- Data may inconsistent due to **hardware failure**, but it is very **rare!!**
- **Common cause** of data inconsistency is data read/write operation by **multiple users at the same time**.

Data Consistency: Example

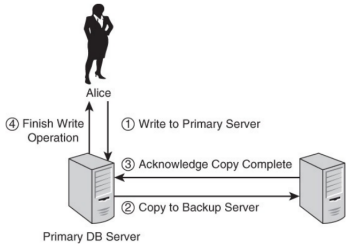


- Consider a small business with two partners, Alice and Bob. Alice is using a database application to update the company's financial records.
- Bob is checking the total fund available **at the same time**. What balance will Bob see?

Ensure Data Availability

- Data should be available whenever it is needed. This is difficult to guarantee.
- A database that runs on a **single server** can be unavailable for a large number of reasons.
- One way to avoid the problem of an unavailable database server is to have **two** database servers: **primary server** and **back-up server**
- 2 server setup is controlled by **two-phase commit** protocol.

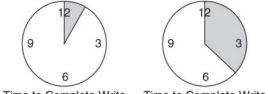
Two-phase Commit



- When the database is in use, any changes to the primary database are reflected in the backup database as well.
- With data consistent on two database servers, you can be sure that if the primary database fails, you can switch to using the backup database and know that you have the same data on both.

Two-phase Commit (Cont.)

- The **advantage** of using two database servers is that it enables the database to remain available even if one of the servers fails.
- But it **incurs cost**. A write operation is not complete until both databases are updated successfully. And it incurs additional time due to amount of data written, the speed of the disks, the speed of the network between the two servers.



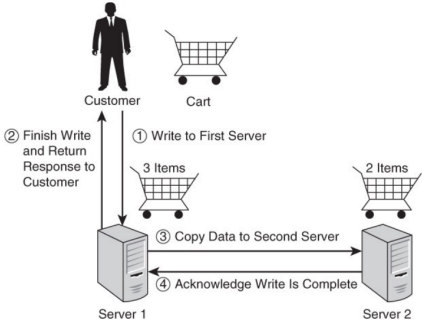
Availability and Consistency: Application Diversity

- When two database servers must keep consistent copies of data, they **incur longer times** to complete a transaction.
- This is acceptable in applications that require both consistency and high availability at all times. **Financial systems at a bank**, for example, fall into this category.
- There are applications, however, in which the **fast database operations are more important than maintaining consistency** at all times.
  For example, an **e-commerce site** might want to maintain copies of your shopping cart on two different database servers. If one of the servers fails, your cart is still available on the other server.
- How long should the customer wait after clicking on an "Add to My Cart" button? Ideally, the interface would respond immediately so the customer could keep shopping. For a slow and sluggish system, the **customer might switch to another site**.

e-commerce Site Challenge

- Let the program **write the updates to one database** and then let the program know the data has been saved. The interface can indicate to the customer that the product has been added to the cart.
- While the customer receives the message that the cart has been updated, the database management system is making a copy of the newly updated data and **writing it to another server**.
- There is a **brief period of time** when the customer's cart on the **two servers is not consistent**, but the customer is able to continue shopping anyway.
- In this case, we are **willing to tolerate inconsistency for a brief period of time** knowing that eventually the two carts will have the same products in it.

e-commerce Site Challenge (Cont.)

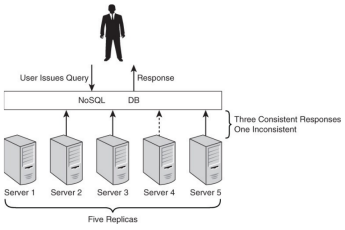


- There is only **a small chance** someone else would read that customer's cart data anyway from the server2.

Eventual Consistency

- NoSQL databases often implement **eventual consistency**; that is, there might be a period of time where copies of data have different values, but eventually all copies will have the same value.
- This raises the possibility of a user **querying the database and getting different results** from different servers in a cluster.
- NoSQL databases often use the concept of **quorums** when working with reads and writes.

Eventual Consistency: Quorums



- A quorum is the number of servers that must respond to a read or write operation for the operation to be considered complete.
- When a read is performed, the NoSQL database **reads data from, potentially, multiple servers**.
- **For example,** assume data in a NoSQL database is replicated to five servers and you have set the read threshold to 3. As soon as **three servers respond with the same response**, the result is returned to the user.
- You can **vary the threshold** to improve response time or consistency.

The CAP Theorem

The CAP Theorem

The CAP theorem, also known as Brewer's theorem after the computer scientist who introduced it, states that distributed databases cannot have consistency (C), availability (A), and partition tolerance (P) all at the same time.

The CAP Theorem (Cont.)

- **Consistency (C):** Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed successful.
- **Availability (A):** Availability means that any client making a request for data gets a response, even if one or more nodes are down.
- **Partition tolerance (P):** A partition is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes. Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.

[1] https://www.ibm.com/

ACID and BASE

- ACID is an acronym derived from four properties implemented in relational database management systems.
- BASE is an acronym for properties common to NoSQL databases.

ACID Properties

- **A** is for atomicity. Atomicity, as the name implies, describes a unit that cannot be further divided.
- **C** is for consistency. In relational databases, this is known as **strict consistency**.
- **I** is for isolation. Isolated transactions are not visible to other users until transactions are complete.
- **D** is for durability. This means that once a transaction or operation is completed, it will remain even in the event of a power loss.

Relational database management systems are designed to support ACID transactions.

BASE Properties

- **BA** is for basically available. This means that there can be a partial failure in some parts of the distributed system and the rest of the system continues to function.

  For example, if a NoSQL database is running on 10 servers without replicating data and one of the servers fails, then 10% of the users' queries would fail, but 90% would succeed.
- **S** is for soft state. Usually in computer science, the term soft state means data will expire if it is not refreshed.
  Here, in NoSQL operations, it refers to the fact that data may eventually be overwritten with more recent data. This property overlaps with the third property of BASE transactions, eventually consistent.
- **E** is for eventually consistent. This means that there may be times when the database is in an inconsistent state.

Types of Eventual Consistency

Eventual consistency is such an important aspect of NoSQL databases. There are several types of eventual consistency:

1. Casual consistency
2. Read-your-writes consistency
3. Session consistency
4. Monotonic read consistency
5. Monotonic write consistency

Casual Consistency

Casual consistency ensures that the database **reflects the order** in which operations were updated.

Example: if Alice changes a customer's outstanding balance to $1,000 and one minute later Bob changes it to $2,000, all copies of the customer's outstanding balance will be updated to $1,000 before they are updated to $2,000.

Read-Your-Writes Consistency

Read-your-writes consistency means that once you have updated a record, all of your reads of that record will **return the updated value**. You would **never retrieve a value inconsistent** with the value you had written.

Example: Alice updates a customer's outstanding balance to $1,500. The update is written to one server and the replication process begins updating other copies. During the replication process, Alice queries the customer's balance. She is guaranteed to see $1,500 when the database supports read-your-writes consistency.

Session Consistency

Session consistency ensures read-your-writes **consistency during a session**.

- ✓ **As long as the conversation (session) continues**, the database "remembers" all writes you have done during the conversation. **If the session ends and you start another session** with the same server, there is **no guarantee it will "remember" the writes you made in the previous session**.
- ✓ A **session may end** if you log off an application using the database or if you do not issue commands to the database for so long that the database assumes you no longer need the session and abandons it.

Monotonic Read Consistency

Monotonic read consistency ensures that if you issue a query and see a result, you will **never see an earlier version of the value**.

Example: Alice is yet again updating a customer's outstanding balance. The outstanding balance is currently $1,500. She updates it to $2,500. Bob queries the database for the customer's balance and sees that it is $2,500. If Bob issues the query again, he will see the balance is $2,500 even if all the servers with copies of that customer's outstanding balance have not updated to the latest value.

Monotonic Write Consistency

Monotonic write consistency ensures that if you were to issue several update commands, they would be **executed in the order you issued them**.

Concept of Aggregates Data Model[1]

- The dominant data model of the last couple of decades is the relational data model, which is best visualized as a set of tables.
- Each NoSQL solution has a different model that it uses, which we put into four categories widely used in the NoSQL ecosystem: key-value, document, column-family, and graph.
- Of these, the first three share a common characteristic of their data models which we will call aggregate orientation.

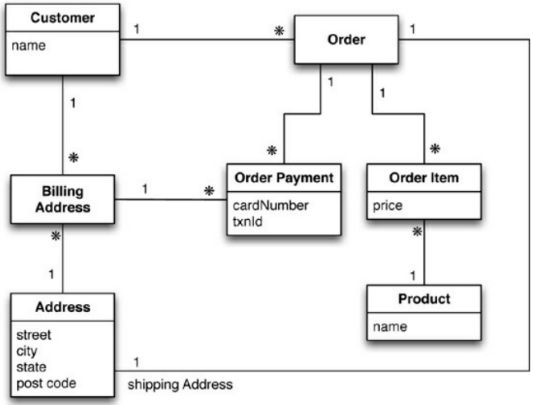
[1] Chapter 2: NoSQL Distilled

Aggregates

- Aggregate is a term that comes from Domain-Driven Design [Evans]. In Domain-Driven Design, an aggregate is a collection of related objects that we wish to treat as a unit.
- Typically, we like to update aggregates with atomic operations and communicate with our data storage in terms of aggregates.
- **What you want** must match with **What you store**.
- Aggregate can work like a Node which is independent of other nodes.
- It **incurs redundancy** but **scales-up more efficiently**.
- **Aggregate boundary** must ba carefully designed.

Aggregates: Example

An application of e-commerce website.



Aggregates: Example (Cont.)

Typical data using RDBMS data model:

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

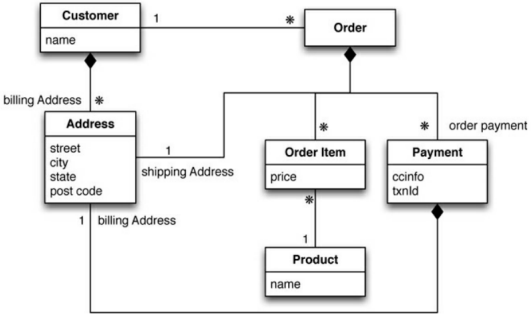
Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Aggregates: Example (Cont.)

In this model, we have two main aggregates: **customer** and **order**.



Aggregates: Example (Cont.)

Again, we have some sample data, which we'll show in JSON format as that's a common representation for data in NoSQL.

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Aggregates: Example (Cont.)

✓ The important thing to notice here isn't the particular way we've drawn the aggregate boundary so much as the fact that you have to think about accessing that data (i.e.UI)

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
```

✓ Like most things in modeling, there's **no universal answer** for how to draw your aggregate boundaries. It depends entirely on **how you tend to manipulate your data**.

Types of NoSQL Databases

The most widely used types of NoSQL databases are:

- Key-value pair databases e.g. Riak KV (Key-Value)
- Document databases e.g. MongoDB, Apache CouchDB
- Column family store databases Apache Cassandra
- Graph databases e.g. Neo4j ✓

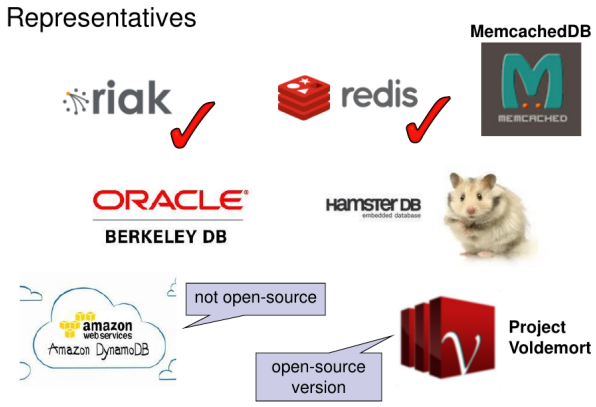
Note: NoSQL databases do not have to be implemented as distributed systems. Many can **run on a single server**. Some of the most interesting and appealing features of NoSQL databases, however, **require a distributed implementation**.

Key-value Databases: Basic characteristics

- The simplest NoSQL data store:
  - ✓ A hash table (map)
  - ✓ When all access to the database is via primary key
- Like a table in RDBMS with two columns:
  - ✓ ID = key
  - ✓ NAME = value (any type)
- Basic operations:
  - ✓ **get** the value for the key
  - ✓ **put** a value for a key (existing value will be overwritten)
  - ✓ **delete** a key from the data store

[1] https://www.ksi.mff.cuni.cz/~svoboda/courses/2015-2-MIE-PDB/

Key-value Databases: Examples



[1] https://www.ksi.mff.cuni.cz/~svoboda/courses/2015-2-MIE-PDB/

Key-value Databases: Suitable Applications

- **Storing Session Information**
  - ✓ Every web session is assigned a unique session_id value
  - ✓ Everything about the session can be stored by a single **PUT** request or retrieved using a single **GET**
- **User Profiles, Preferences**
  Every user has a unique user_id, user_name + preferences (e.g., language, colour, time zone, which products the user has access to, ...)
- **Shopping Cart Data**

Key-value Databases: When NOT To Use

- **Relationships among Data**
  - ✓ Relationships between different sets of data
- **Multioperation Transactions**
  - ✓ Saving multiple keys. Failure to save any one of them → **revert or roll back** the rest of the operations. (Since it does not ensure ACID)
- **Query by Data**
  - ✓ **Search** the keys based on something found in the **value part**
- **Operations by Sets**
  - ✓ **No way to operate** upon **multiple keys** at the **same time**

Key-value Pair Databases

These databases are modeled on two components: **keys** and **values**.

Keys:

- Keys are identifiers associated with values. They are analogous to tags you get when you check luggage at the airport.
- **Example:** a key-generating program for an **e-commerce website**. Need to track **five pieces of information** as follows:
  i. customer's account number
  ii. name
  iii. address
  iv. number of items in the shopping cart
  v. customer type indicator.

Key-value Pair Databases: Key (Cont.1)

- All of these values are associated with a customer, so you can generate a sequential number for each customer.

  For example, data about the first customer in the system would use keys **1.accountNumber, 1.name, 1.address, 1.numItems, and 1.custType**

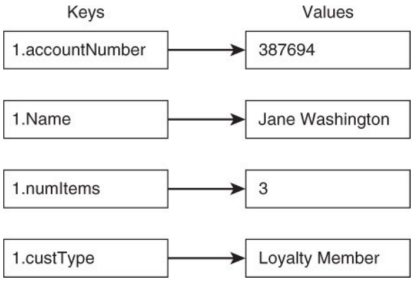


Figure 1: Two-part data structure: Key and its value

Key-value Pair Databases: Key (Cont.2)

- Multiple entities involving similar attribute will **cause a problem**, since in this case one key will point to two different values at two different times. But at the end, only one value will persist (which has been written most recently)

  For instance: 1.Name in another entity means Name of the first branch of the company. But if you save 1.Name value as the brach name and query next time to find the name of the 1st customer by the key 1.Name, it will generate wrong answer (i.e. branch name)

- **Solution:** Use a key-naming convention that includes the entity type.
  For Instance: the prefix `cust` and `brn` could be used to differentiate them:
  `cust1.name and brn1.name` are now two different keys

Key-value Pair Databases: Key (Cont.3)

- To differentiate among entities, data structures for separate collections of identifiers within a database are used, commonly called **buckets**

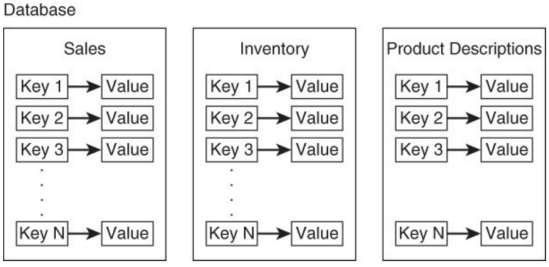


Figure 2: Separate namespaces within a single database

Key-value Pair Databases: Key (Cont.3)

Buckets Example Code: Let's store an object containing information about a dog named Rufus. We'll store that object in the **key=rufus** in the bucket dogs, which bears the animals bucket type.
Notice that we specified both a **value for the object, i.e. WOOF!**, and a content type, text/plain

Python Code for **Riak KV**:

```
bucket = client.bucket_type('animals').bucket('dogs')
obj = RiakObject(client, bucket, 'rufus')
obj.content_type = 'text/plain'
obj.data = 'WOOF!'
obj.store()
```

Key-value Pair Databases: Value

- Values can be as **simple** as a string or **complex** values, such as images or binary objects, too.
- It offers **great flexibility** when storing values.
  - ✓ For example, its string length may vary.
  - ✓ Another example: An employee database might include photos of employees using keys such as Emp328.photo. That key could have a picture stored as a binary large object (BLOB) type or a string value such as "Not available."
- Key-value databases typically **do not enforce checks on data types of values**. Hence, software developers must implement checks in their programs.

Key-Value and Relational Databases: Difference

- Key-value databases are modeled on minimal principles for storing and retrieving data.
- Unlike in relational databases, there are no tables, so there are no features associated with tables, such as columns and constraints on columns.
- There is no need for joins in key-value databases, so there are no foreign keys. Key-value databases do not support a rich query language such as SQL.

Key-Value and Relational Databases: Difference (Cont.)

- But, there are parallels between the key-naming convention and tables, primary keys, and columns :

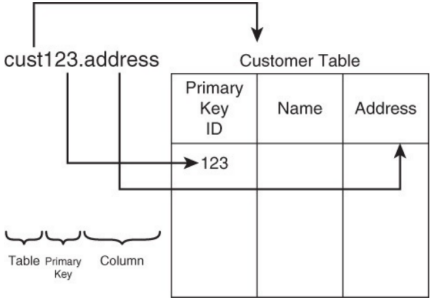


Figure 3: Mapping from KV to Relational DB

Graph Databases

Graph databases are the most specialized of the four NoSQL databases discussed in this book. Instead of modeling data using columns and rows, a graph database uses structures called **nodes and relationships (edges)**. It is based on **Principles of Graph Theory.**

Relational Databases: Major Weakness

- Since the 80s, they have been the power-horse of most software applications and continue to be so today. Relational databases (RDBMSs) were initially designed to codify paper forms and tabular structures, and they do that exceedingly well.
- RDBMSs **don't** robustly store **relationships between data elements**. They are not designed to capture this rich relationship information.
- Relational database heavily depends on **JOIN operations**. These operations are **compute and memory-intensive and have an exponential cost** as queries grow.

Relational Databases: SQL Strain

Any attempt to solve a connected data problem with a relational database may cause the following performance problems commonly known as **SQL Strain**:

- **A Large Number of JOINs.** When you utilize queries that JOIN many different tables, there's an explosion of complexity and computing resource consumption. This results in a corresponding increase in query response times.
- **Numerous Self-JOINs (or Recursive JOINs).** Self-JOIN statements are common for hierarchy and tree representations of data. It also incurs higher respoonse time.
- **Frequent Schema Changes.** Relational databases isn't designed for frequent modifications of its structure. But today's agile nature of applications demand such frequent changes. A more flexible data model is needed.

Graph Database an Alternative of Relational Databases

- For highly structured, predetermined schemas, an RDBMS is the perfect tool. But for ever-changing nature of applications we need an efficient alternative - Graph is a good option in this regard.
- Graph database offers better performance and flexibility (and agility).
- In fact, the flexibility of a graph model **allows you to add new nodes and relationships without compromising your existing network** or expensively migrating your data. All of your original data (and its original relationships) remain intact.

Graph Database an Alternative of Relational Databases (Cont.)

- With data relationships at their center, graphs are **incredibly efficient** when it comes to query speeds, even for deep and complex queries. In the book Neo4j in Action, the authors performed an experiment between a relational database and a Neo4j graph database.

✓ Their experiment used a basic social network to find friends-of-friends connections to a depth of five degrees. Their dataset included 1,000,000 people each with approximately 50 friends. Result is given as follows:

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2,500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Graph Database an Alternative of Relational Databases: Example

Consider the following sample relational database:

User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

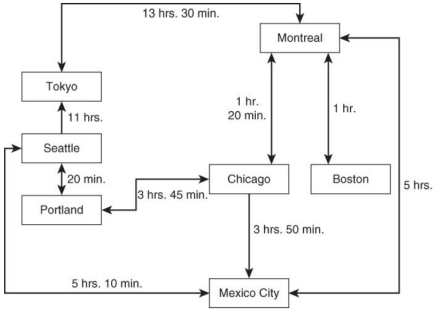
ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

- Q1: What items did a customer buy?
- Q2: Which customers bought this product?

Both queries will be very slow because of several JOINs.

Graph Databases: Motivating Example

- **Example:** A node could be a city, and a relationship between cities could be used to store information about the distance and travel time between cities.
  Objective: What is the best route from Seattle to Boston?
- **Solution:** Solution to the Shortest Path Problem



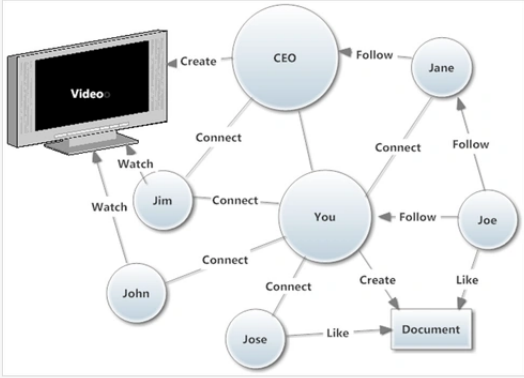
Graph Databases: Motivation (Cont.)

City 1	City 2	Flying Time
Montreal	Boston	1 hr.
Montreal	Chicago	1 hr. 20 min.
Montreal	Tokyo	13 hr. 30 min.
Montreal	Mexico City	5 hr.
Chicago	Mexico City	3 hr. 50 min.
Chicago	Portland	3 hr. 45 min.
Portland	Seattle	20 min.
Seattle	Tokyo	11 hr.
Seattle	Mexico City	5 hr. 10 min.

- In RDBMS it is **hard to achieve**. Performing this same kind of analysis in a relational database would be more challenging.
- Querying is more difficult. You would have to write multiple SQL statements or use **specialized recursive statements** if they are provided **to find paths** using the table representation of the data.

Graph Databases: Motivation (Cont.)

- Social Network with different relationships.



Graph Databases: Motivation Summary

- Graph Database offers flexible and efficient (i.e. scalable) data operation specially with high degree of data dependency (of large volume)
- Graph Database applications include: social network modeling, e-mail fraud detection, Recommendations in Real Time and so on.
- Basic operations of Graph (such as search, find the degree, find sub-graph) can be used to satisfy user Query in these applications.

Examples of Graph Databases[1]

[1]Image Source: https://www.datacamp.com/blog/what-is-a-graph-database

Example of Graph Databases: neo4j

Neo4j is one of the world's leading graph databases. It is a **highly scalable NoSQL open-source database** developed using **Java**.

Main Features[1]:

- **Property graph data model.** Enables intuitive and flexible data modeling, easy to capture complex data relationships.
- **Native graph processing and storage.** Optimizes data retrieval and graph traversals, ensuring swift and efficient handling of large datasets and complex queries.
- **ACID compliant transactions.** Guarantees reliable data processing, maintaining data accuracy and trustworthiness across all transactions.

[1] https://www.datacamp.com/blog/what-is-a-graph-database

Example of Graph Databases: neo4j

Main Features[1] (Cont.):

- **Cypher graph query language.** Provides a powerful yet user-friendly method for querying graph data, simplifying the extraction of meaningful insights from interconnected data.
- **High-performance native API.** Ensures efficient interaction with the database, crucial for applications requiring low-latency and high-throughput database interactions.
- **Cypher client.** Facilitates seamless execution of Cypher queries from applications, enhancing dynamic and interactive user experiences.
- **Driver support for multiple programming languages.** Offers flexibility in development by providing drivers for various programming languages, including **C#, Go, Java, JavaScript, and Python**, ensuring easy integration into diverse technology stacks.

[1] https://www.datacamp.com/blog/what-is-a-graph-database

Neo4j : Basics of Cypher

- Cypher statements to create vertices[1]:

```
1  CREATE (robert:Developer { name: 'Robert Smith' })
2  CREATE (andrea:Developer { name: 'Andrea Wilson' })
3  CREATE (charles:Developer { name: 'Charles Vita'  })
```

✓ These three statements create three vertices.

✓ The text robert: Developer creates a developer vertex with a label of robert.

✓ The text { name: 'Robert Smith' } adds a property to the node to store the name of the developer.

[1] NoSQL for Mere Mortals:Chapter14

Neo4j : Basics of Cypher (Cont.)

- Edges are added with create statements as well[1]. For example:

```
1  CREATE (robert)-[FOLLOWS]->(andrea)
2  CREATE (andrea)-[FOLLOWS]->(charles)
```

[1] NoSQL for Mere Mortals:Chapter14

Neo4j : Gremlin

- Gremlin: It is a query language used to retrieve data from and modify data in the graph. (part of Apache TinkerPop Framework founded and maintained by Apache)
- It provides both Depth-First and Breadth-First Searches for a given graph.
- Interested readers may consult the following urls:

  i. https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html#tpintro
  ii. https://tinkerpop.apache.org/docs/current/reference/#_tinkerpop_documentation

Thank You!!!