IMEN891K / IDSC501 / AXGS-501 · DATA ENGINEERING · FALL 2026

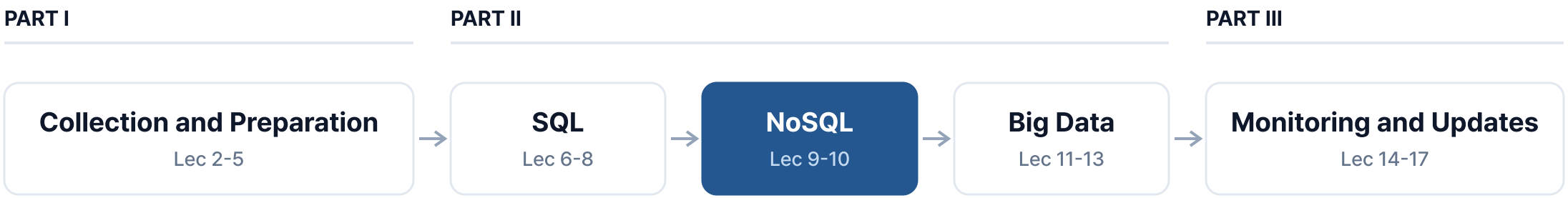
9.

NoSQL I: Landscape

Sangwoo Mo · POSTECH

Industrial & Management Engineering

Where We Are

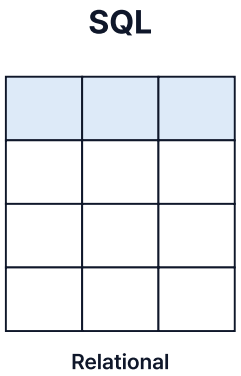

PART I
PART II
PART III
Collection and Preparation
Lec 2-5
SQL
Lec 6-8
NoSQL
Lec 9-10
Big Data
Lec 11-13
Monitoring and Updates
Lec 14-17

Outline: NoSQL I

01	**Four Families of NoSQL**	choose a model by its access pattern
02	**JSON and Semi-Structured Data**	read named fields and ordered arrays
03	**Between Trees and Tables**	convert nesting and repeated values
04	**Why NoSQL**	check flexibility, scale and guarantees

Recap: Relational Tables

Lectures 6 to 8

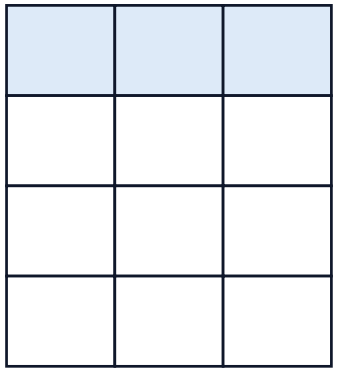


Lectures 6 to 8: one data model, the **relational table**

Four NoSQL Families

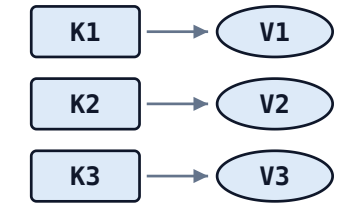
The four NoSQL families

SQL

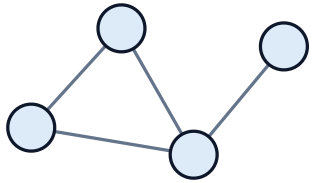


Relational

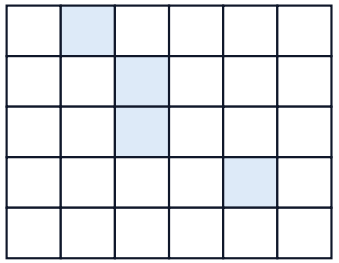
NoSQL



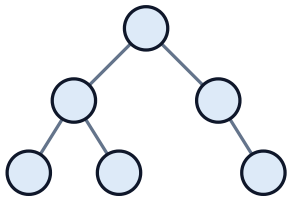
Key-value



Graph



Wide-column



Document

NoSQL is not one thing: four alternative **storage models**

01 · Four Families of NoSQL

01 **Four Families of NoSQL** — choose a model by its access pattern

02 **JSON and Semi-Structured Data** — read named fields and ordered arrays

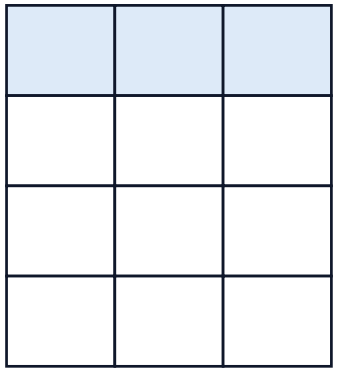
03 **Between Trees and Tables** — convert nesting and repeated values

04 **Why NoSQL** — check flexibility, scale and guarantees

Key-Value Store

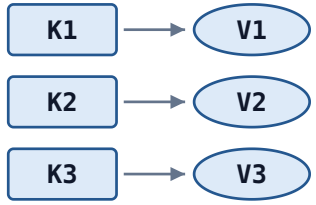
Core idea

SQL

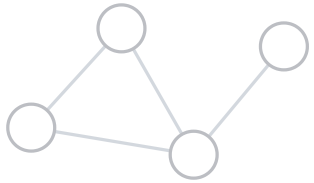


Relational

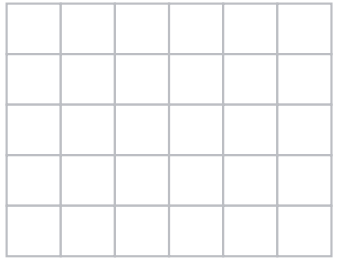
NoSQL



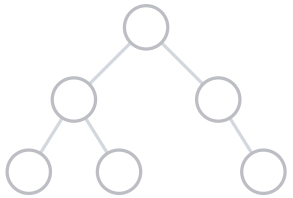
Key-value



Graph



Wide-column



Document

The simplest of the four: a key, and whatever bytes you put under it

Key-Value Records

Request

```
get("user:1")
```



Stored pair

```
key: "user:1"
value: {
  name: "John",
  age: 28
}
```



Answer

```
{ name: "John", age:
28 }
```

A key retrieves its associated value

Key-Value Operations

Data model and operations

- **Data model**: (key, value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)
- **Operations**
  - `get(key)`, `put(key, value)`
  - Operations on value not supported
    - user writes their own code for those

Choosing a Lookup Key

Exercise: the Product table

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

How would you represent the **Product** data (Lecture 6) as key-value pairs?

Answer: Choosing a Lookup Key

Exercise: the Product table

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

- How would you represent the **Product** data as key-value pairs?
- Option 1: key = `PName`, value = entire product record
- Option 2: key = `Manufacturer`, value = all products of that company
- Option 3: key = `Category`, value = all products in that category

Caveat: A Lookup Key Limits the Question

Stored by product name

```
Gizmo:
  {price: 19.99, category: "Gadgets"}
PowerGizmo:
  {price: 29.99, category: "Gadgets"}
```

Two requests

```
get("Gizmo")
→ {price: 19.99, category: "Gadgets"}

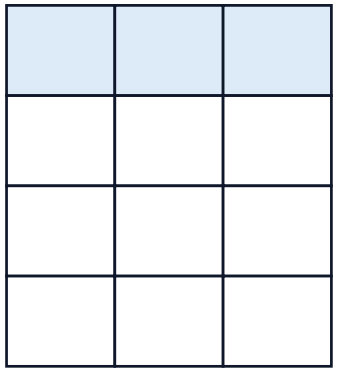
get("Gadgets")
→ no stored key
```

Gizmo is a stored key; Gadgets is inside the values

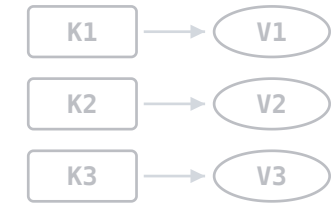
Wide-Column Store

Core idea

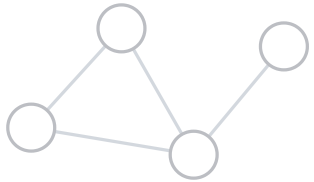
SQL



Relational

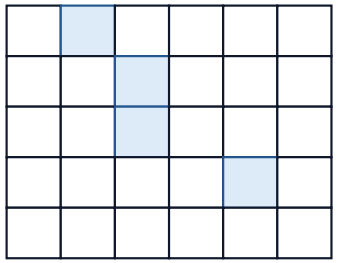


Key-value

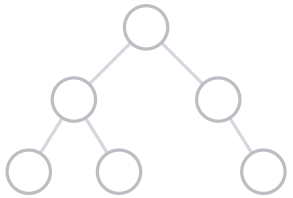


Graph

NoSQL



Wide-column



Document

Looks like a table, but every row may have **its own set of columns**

Sparse Cells Under a Row Key

In a Bigtable-style model, rows can hold different column qualifiers

user:1

Family: activity	Value
page:home	09:01
page:product	09:03
purchase:1	Laptop

user:2

Family: activity	Value
page:home	09:02
purchase:2	TV

Same column family, different populated cells

Caveat: A Busy Partition Key

Incoming readings

device	time
D1	09:00
D1	09:01
D1	09:02
D2	09:00

Partition A: key D1

time	device
09:00	D1
09:01	D1
09:02	D1

Partition B: key D2

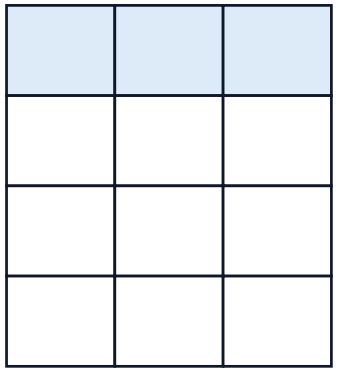
time	device
09:00	D2

The D1 key keeps three readings together and concentrates their requests in one partition

Graph Store

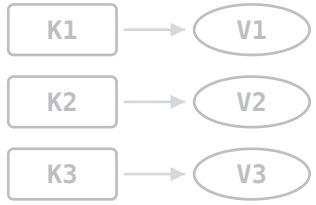
Core idea

SQL

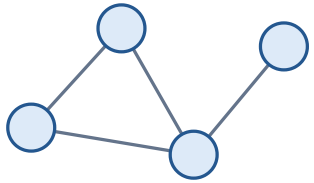


Relational

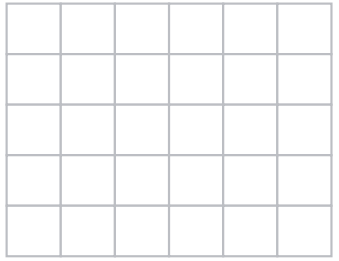
NoSQL



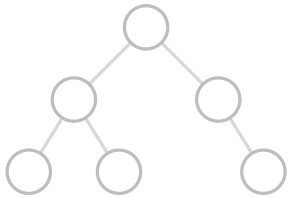
Key-value



Graph



Wide-column

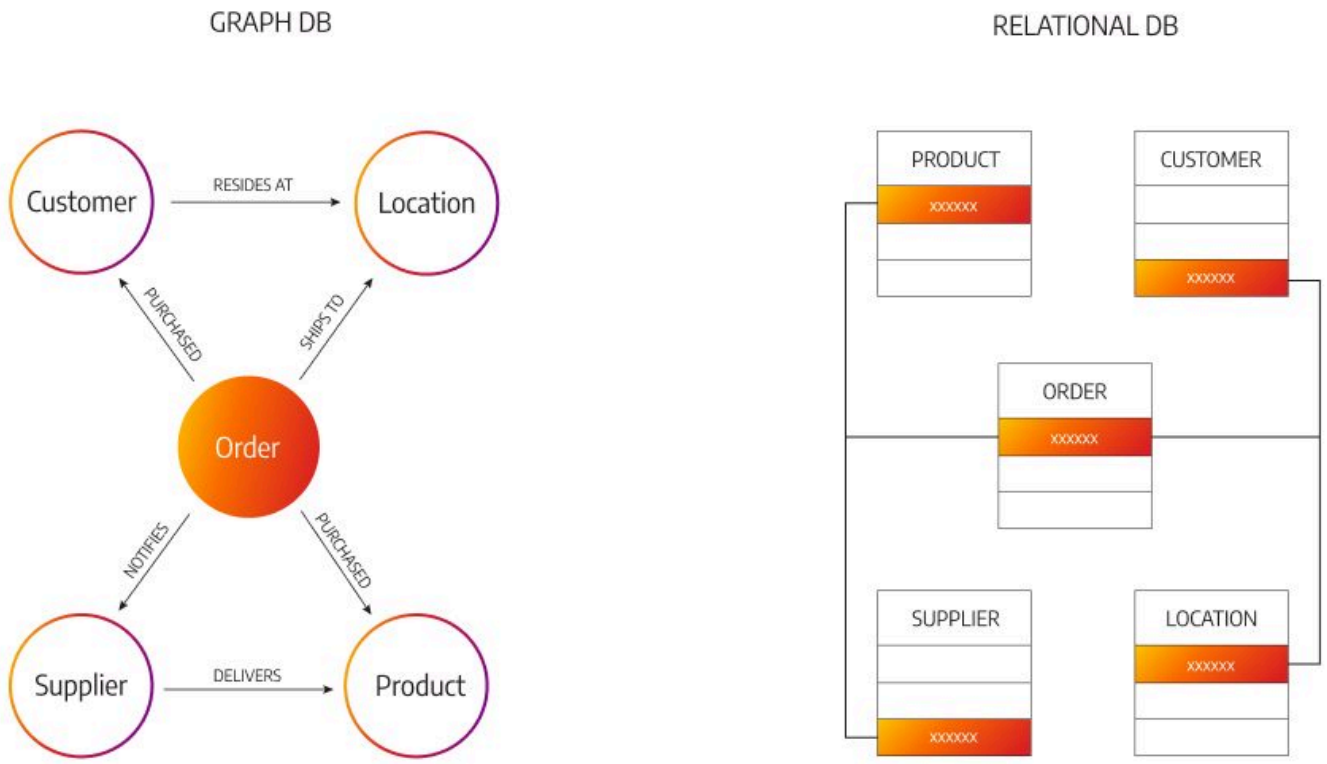


Document

Nodes and edges are the data; **following an edge** replaces a join

Edges and Foreign Keys

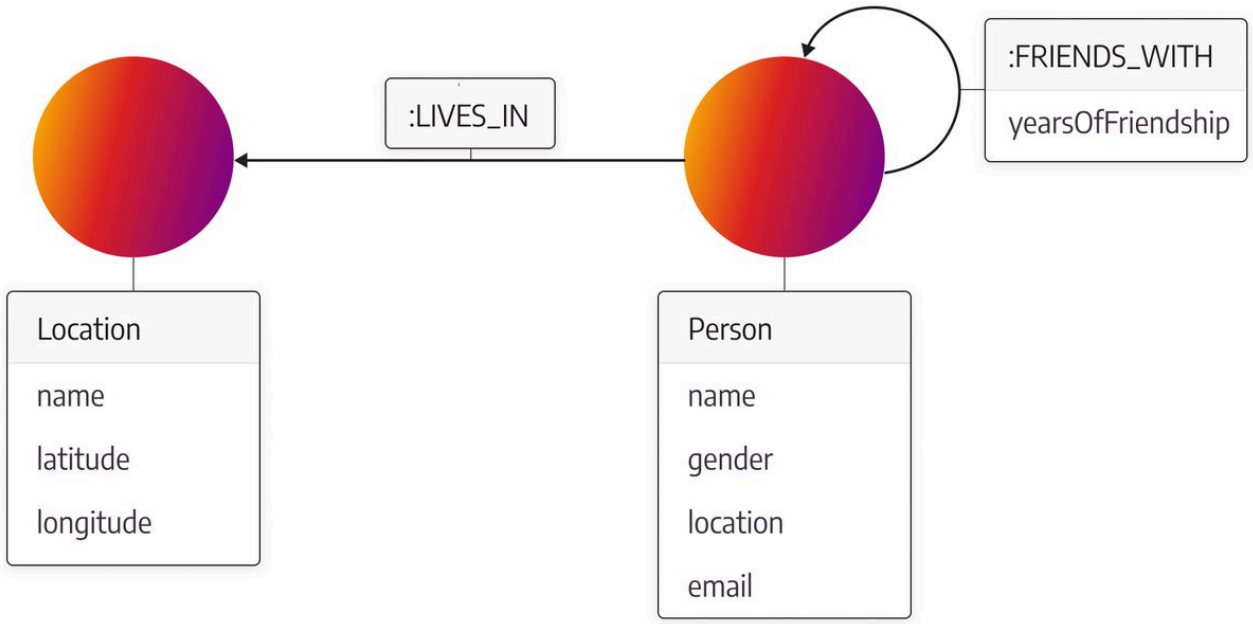
Edges instead of foreign keys



The same five entities: edges on the left, **foreign keys and joins** on the right

Property Graphs

Property graphs

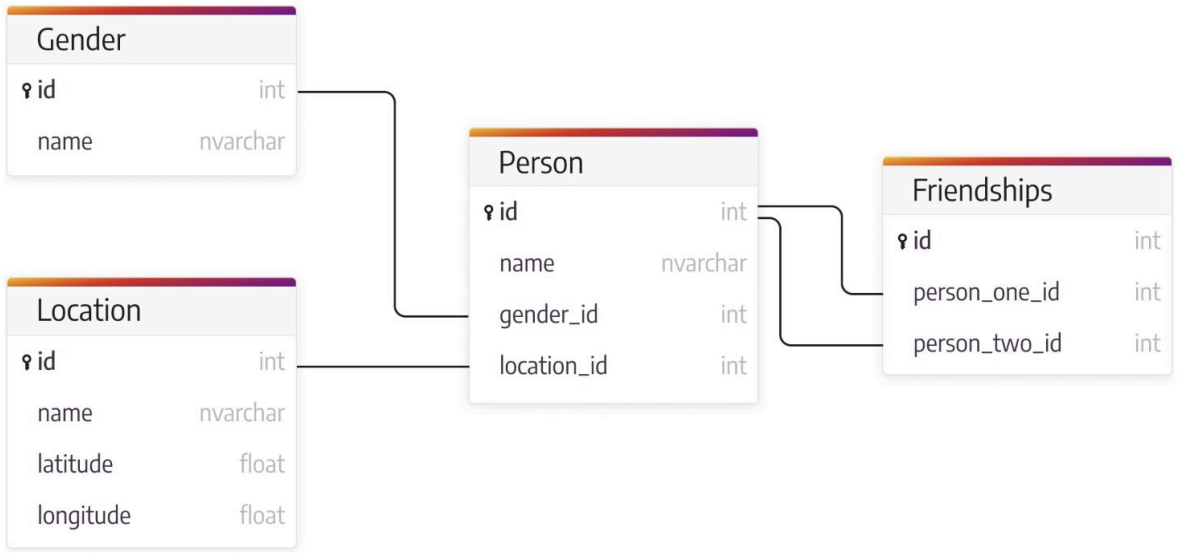


A **property graph**: both nodes and edges carry attributes

Paths and Relational Joins

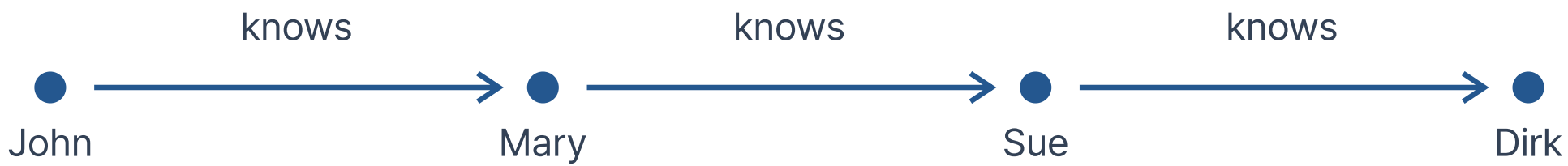
When a graph fits

Graph databases represent **relationships as edges** and offer traversal queries. Relational databases can use an edge table with a relationship-type column. The better fit depends on the traversal pattern and workload.



Friendship as a table: two-hop paths use **two copies** of the edge table

Paths of Unknown Depth



Stored edges

from	to
John	Mary
Mary	Sue
Sue	Dirk

Starting at John

Hops	Reached node
1	Mary
2	Sue
3	Dirk

Each extra hop reads another edge; the destination changes with the requested path depth

Caveat: A Path Can Revisit a Node



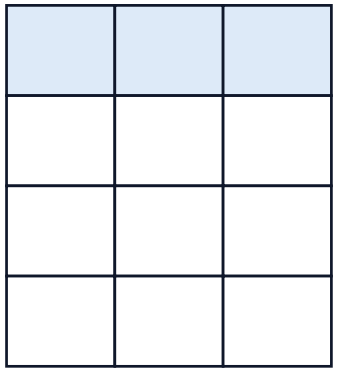

```
John → Mary → Sue → Dirk → Mary → Sue → …
```

One added edge creates a cycle; traversal needs an explicit stopping rule

Document Store

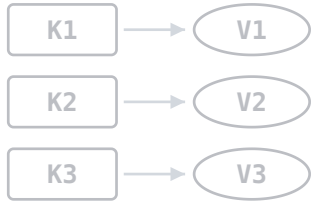
Core idea

SQL

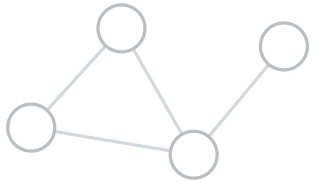


Relational

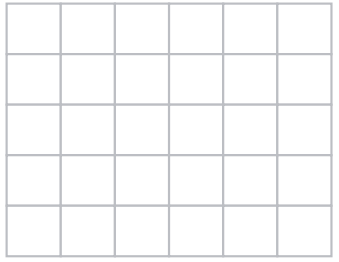
NoSQL



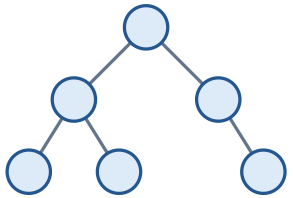
Key-value



Graph



Wide-column



Document

A key-value store whose values have **structure the store can read**

Structure Inside a Value

Values with structure

- In key-value stores, the value is often a very complex object
  - Key = `'GizmoWorks'`, Value = [all products of that company]
- Better: *value* to be structured data
  - JSON or Protobuf or XML
  - Called a “document” but it’s just data

We will discuss JSON

Documents Read Together

Person

name	phone
John	3634

Orders

person	date	product
John	2002	Gizmo
John	2004	Gadget

One person document

```
{name: "John",
 phone: 3634,
 orders: [
  {date:2002,
product:"Gizmo"},
  {date:2004,
product:"Gadget"}
 ]}
```

John and his two orders become one document that the application can read together

02 · JSON and Semi-Structured Data

01 Four Families of NoSQL — choose a model by its access pattern

02 **JSON and Semi-Structured Data** — read named fields and ordered arrays

03 Between Trees and Tables — convert nesting and repeated values

04 Why NoSQL — check flexibility, scale and guarantees

JSON Overview

- JavaScript Object Notation = lightweight text-based open standard designed for human-readable data interchange. Interfaces in C, C++, Java, Python, Perl, etc.
- The filename extension is `.json`.

We will emphasize JSON as **semi-structured** data

JSON Syntax

```
{ "book": [
    { "id": "01",
      "language": "Java",
      "author": "H. Javeson",
      "year": 2015
    },
    { "id": "07",
      "language": "C++",
      "edition": "second",
      "author": "E. Sepp",
      "price": 22.25
    }
  ]
}
```

JSON vs Relational

- Relational data model
  - Rows with declared columns and types
  - Declare columns before inserting rows; evolve them explicitly
  - Storage encoding is a separate design choice
  - Query language based on Relational Algebra
- Semistructured data model / JSON
  - Flexible, nested structure (trees)
  - Does not require predefined schema (“self-describing”)
  - JSON text is an interchange format; stores may use binary encodings
  - Queries via MQL, SQL/JSON, and other languages

JSON Values in One Record

One JSON object

```
{
  "name": "Mary",
  "age": 35,
  "address": {"city": "SF"},
  "phones": [1234, 5678],
  "active": true,
  "note": null
}
```

Read its structure

Path	Kind
name	string
age	number
address	object
phones	ordered array
active	Boolean
note	null

A field value can contain another object or an ordered array

Avoid Using Duplicate Keys

Object names should be unique; repeated names are handled differently

```
{ "id": "07",
  "title": "Databases",
  "author": "Garcia-Molina",
  "author": "Ullman",
  "author": "Widom"
}
```

same key again

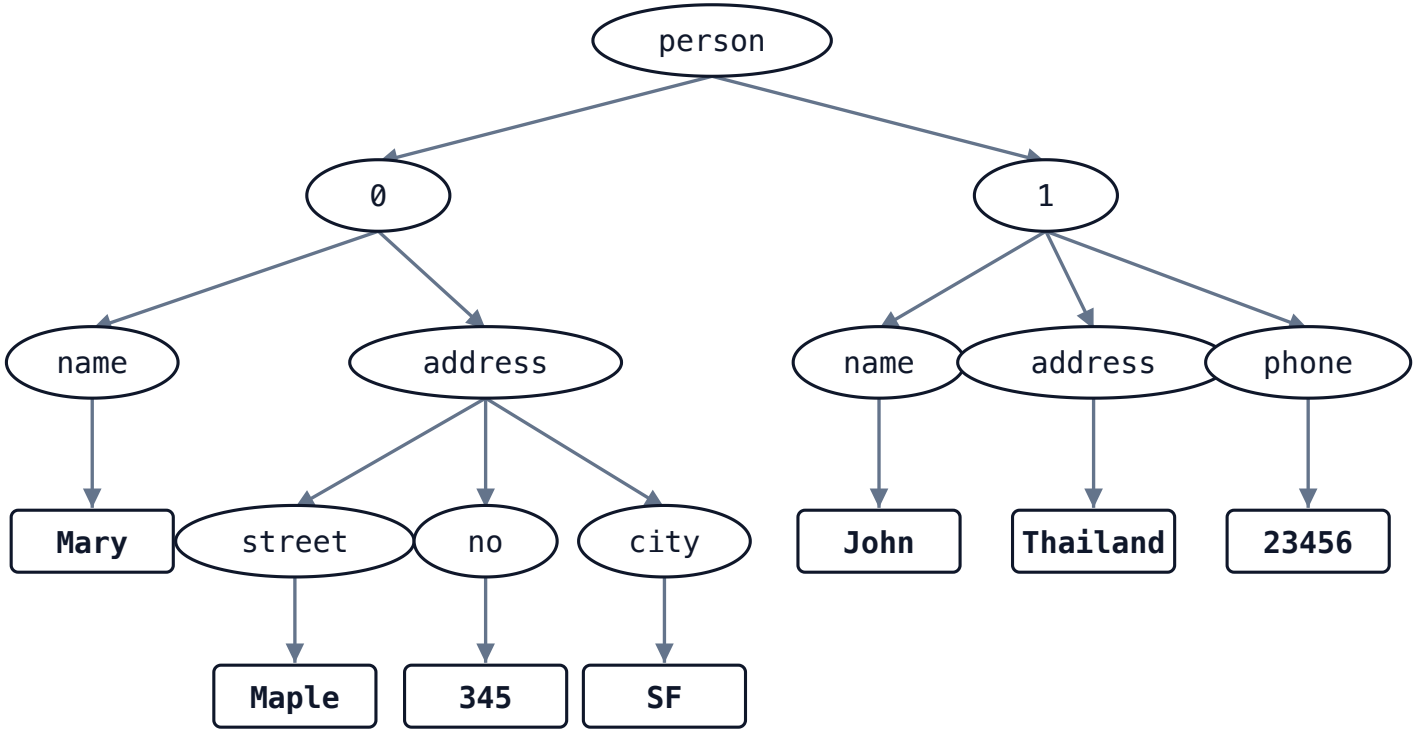
Use an ordered list instead

```
{ "id": "07",
  "title": "Databases",
  "author": ["Garcia-
Molina",
             "Ullman",
             "Widom"]
}
```

one key, an array

Paths in a JSON Tree

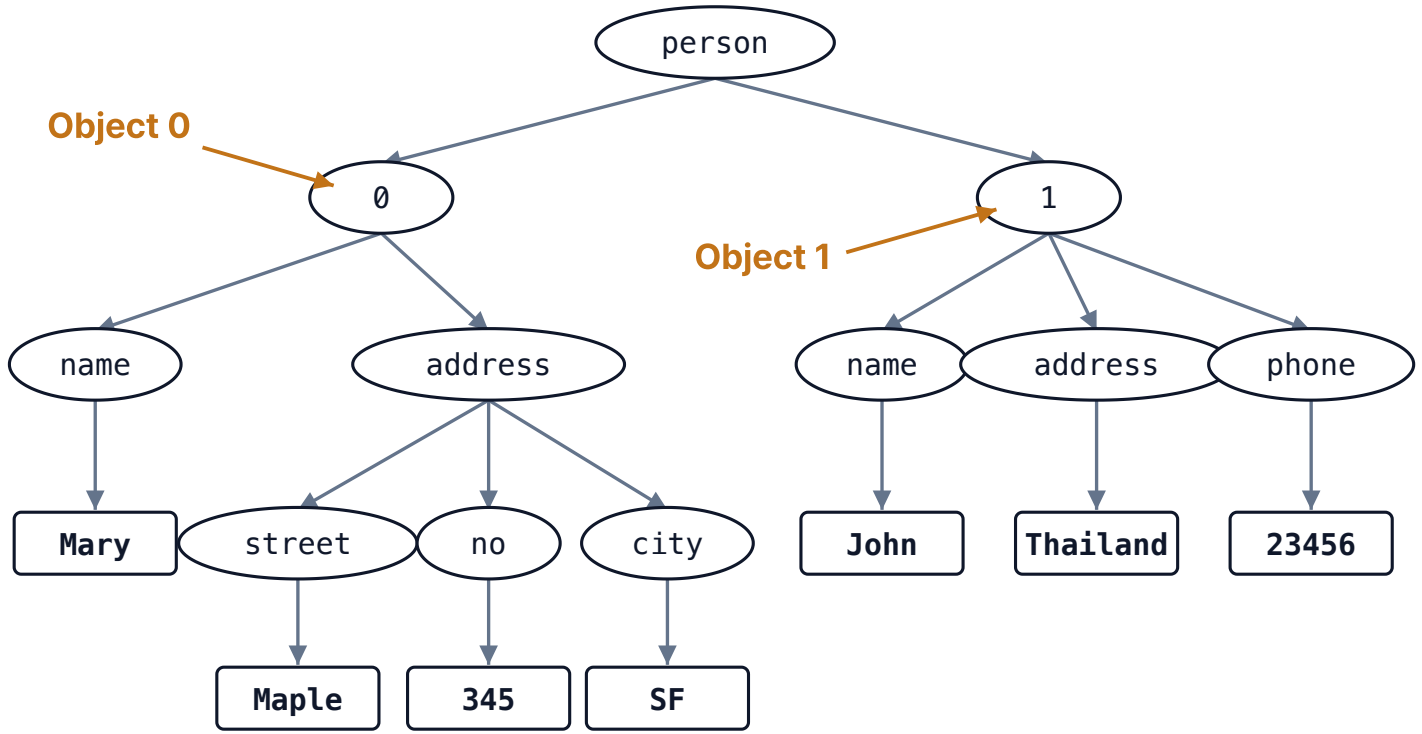
```
{"person":
  [ {"name": "Mary",
     "address":
       {"street": "Maple",
        "no": 345,
        "city": "SF"}},
    {"name": "John",
     "address": "Thailand",
     "phone": 23456}
  ]
}
```



Every value sits at the end of **one path** from the root

Array Order in a JSON Tree

```
{"person":
  [ {"name": "Mary",
     "address":
       {"street": "Maple",
        "no": 345,
        "city": "SF"}},
    {"name": "John",
     "address": "Thailand",
     "phone": 23456}
  ]
}
```



Recall: arrays are *ordered* in JSON!

Array positions label the edges; arrays preserve order while relational rows have no inherent order

Semi-Structured Data

- JSON is **self-describing**
- Schema elements become part of the data
  - Relational schema: `person(name, phone)`
  - In JSON `"person"`, `"name"`, `"phone"` are part of the data, and are repeated many times
- ⇒ JSON is more flexible
  - Schema can change per tuple

Storing JSON in RDBMS

The same person can be represented as a document value or as related rows

A JSON value in one row

```
people
  id: 2
  person: {name: "Mary",
           phones: [1234, 5678]}
```

Separate related rows

```
Person: (2, "Mary")
Phone:  (2, 1234)
        (2, 5678)
```

A document representation and a database engine are separate choices

03 · Between Trees and Tables

01 Four Families of NoSQL — choose a model by its access pattern

02 JSON and Semi-Structured Data — read named fields and ordered arrays

03 **Between Trees and Tables** — convert nesting and repeated values

04 Why NoSQL — check flexibility, scale and guarantees

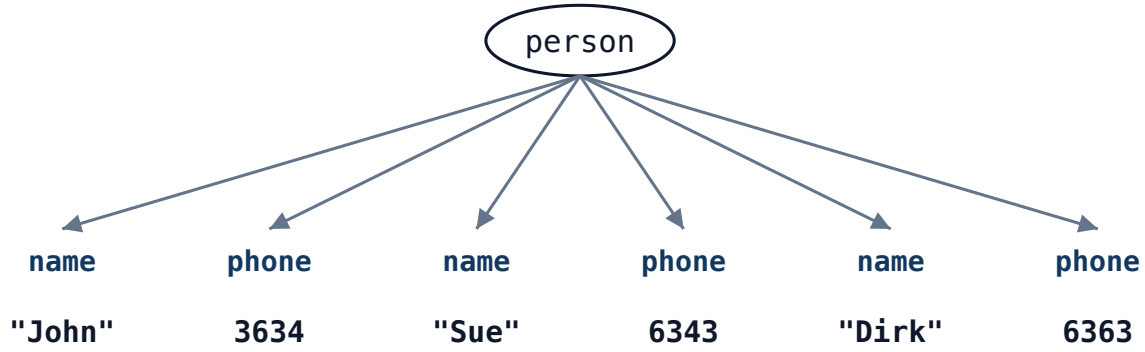
Rows as JSON Objects

One table

Person

name	phone
John	3634
Sue	6343
Dirk	6363

```
{"person": [
  {"name": "John", "phone": 3634},
  {"name": "Sue",  "phone": 6343},
  {"name": "Dirk", "phone": 6363}
]}
```



Represent each row as one object, with named fields for its columns

Nesting Related Orders

Inlining a foreign key

May inline multiple relations based on foreign keys

Person

name	phone
John	3634
Sue	6343

Orders

personName	date	product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

```
{"Person": [
  {"name": "John", "phone": 3634,
   "Orders": [
     {"date": 2002, "product": "Gizmo"},
     {"date": 2004, "product": "Gadget"}
   ]},
  {"name": "Sue", "phone": 6343,
   "Orders": [
     {"date": 2002, "product": "Gadget"}
   ]}
]}
```

orders nested inside the person

One-to-many nests cleanly: the join is **done once, at write time**

Many-to-Many Nesting

Many-to-many relationships

Many-many relationships are more difficult to represent

Person

name	phone
John	3634
Sue	6343

Orders

personName	date	product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

Product

prodName	price
Gizmo	19.99
Phone	29.99
Gadget	9.99

Options for the JSON file:

- 3 flat relations: Person, Orders, Product
- Person → Orders → Products: products are duplicated
- Product → Orders → Person: persons are duplicated

Embedding a many-to-many relationship can **repeat shared data**

Caveat: Stale Copies

Two documents cache the current catalog price, but only one copy receives the update

John's Document

```
{"name": "John",
 "Orders": [
   {"date": 2004,
    "product": {"prodName": "Gadget",
                "currentCatalogPrice": 12.99}}
 ]}
```

Sue's Document

```
{"name": "Sue",
 "Orders": [
   {"date": 2002,
    "product": {"prodName": "Gadget",
                "currentCatalogPrice": 9.99}}
 ]}
```

Remedy: reference the current price or maintain every cached copy

One product, **two prices**: copies need an owner who updates them all

Missing Fields as NULL

Missing attributes

- Missing attributes:

```
{"person":
  [{"name": "John", "phone": 1234},
   {"name": "Joe"}]          // no phone!
}
```

- Could represent in a table with nulls

name	phone
John	1234
Joe	NULL

Repeated Values Need Rows

One person document

```
{
  "name": "Mary",
  "phone": [1234,
5678]
}
```

Person

person_id	name
2	Mary

Phone

person_id	phone
2	1234
2	5678

One person remains one person; each phone gets a related row

Mixed Field Types

Attributes with mixed types

- Attributes with different types in different objects

```
{"person":
  [{"name": "Sue", "phone": 3456},
   {"name": {"first": "John", "last": "Smith"},   // structured name!
    "phone": 2345}]
}
```

- Nested collections
- Heterogeneous collections
- **These are difficult to represent in the relational model**

Recap: The People Table

The four people of the last three slides, as rows of one SQLite table. The whole document sits in a text column; SQLite reads it with its JSON functions.

id	person
1	{"name": "John", "phone": 1234}
2	{"name": "Mary", "phone": [1234, 5678]}
3	{"name": "Joe"}
4	{"name": {"first": "John", "last": "Smith"}, "phone": 2345}

One row per person; the **document is a column value**

Demo: JSON Inside a SQL Table

Storing JSON in RDBMS, made runnable (SQLite JSON functions)

1. Each person's name: `person ->> '$.name'`. What comes back for row 4?
2. One row per phone number: `FROM people p, json_each(p.person, '$.phone') j`. Who disappears, who doubles?
3. People with no phone at all: `json_type(person, '$.phone') IS NULL`.

```
-- write your query here, then Run
```

QUERY RESULT

Write a query on the left, then Run.

Answer: JSON Inside a SQL Table

```
SELECT p.id, p.person ->> '$.name' AS name, j.value
AS phone
FROM   people p, json_each(p.person, '$.phone') j;
```

id	name	phone
1	John	1234
2	Mary	1234
2	Mary	5678
4	{"first":"John","last":"Smith"}	2345

`json_each` turns the phone array into rows: Mary twice, Joe not at all. The relational answer to a repeated attribute is **a second table**, built here on the fly.

A repeated attribute becomes **one row per value**: the join is back

Discussion: Why Semi-Structured Data?

- Semi-structured data works well as *data exchange formats*
  - i.e., exchanging data between different apps
  - Examples: XML, JSON, Protobuf (protocol buffers)
- Increasingly, systems use them as a data model for DBs:
  - SQL Server supports XML-valued relations
  - CouchBase, MongoDB, Snowflake: JSON
  - Dremel (BigQuery): Protobuf

Nesting and Unnesting

Nesting groups related values; unnesting makes repeated values separate rows

Nested document

```
{ name: "Mary",
  phones: [1234, 5678] }
```

One row per phone

name	phone
Mary	1234
Mary	5678

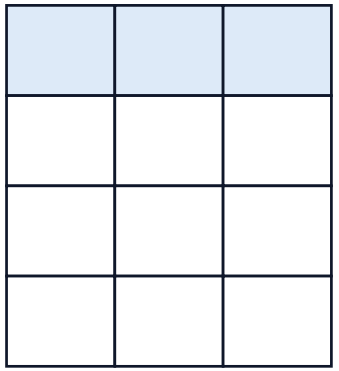
The representation changes how a query reaches each phone

04 · Why NoSQL

01	Four Families of NoSQL	choose a model by its access pattern
02	JSON and Semi-Structured Data	read named fields and ordered arrays
03	Between Trees and Tables	convert nesting and repeated values
04	**Why NoSQL**	check flexibility, scale and guarantees

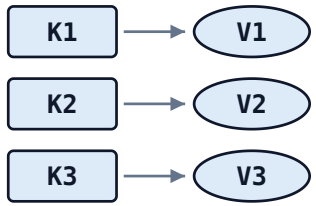
Why NoSQL? – Data Model

SQL

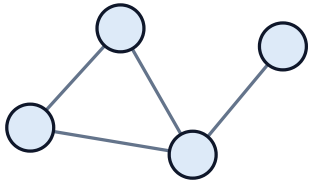


Relational

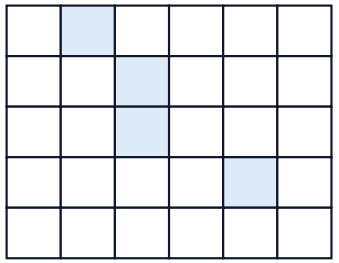
NoSQL



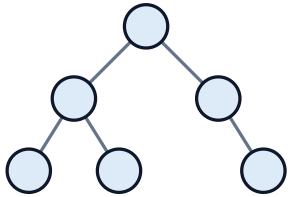
Key-value



Graph



Wide-column



Document

Choose a model for the records and access patterns

Declaring a Relational Schema

Shared rule

```
User
  user_id: unique identifier
  age: numeric value
```

Incoming record

```
{user_id: 2, age: "unknown"}
```

The age value **fails the numeric rule**; reject it or resolve its meaning

A schema makes the expected type explicit before readers try to use the value

Evolving Document Shapes

Documents: mixed shapes

Document databases can allow **mixed record shapes**. Adding a field need not rewrite every document, but readers, validation rules, and old records may still need migration.

Example: User and Orders in a Document Database (e.g., MongoDB)

In a document-based NoSQL database, you store user and order data in a single document.

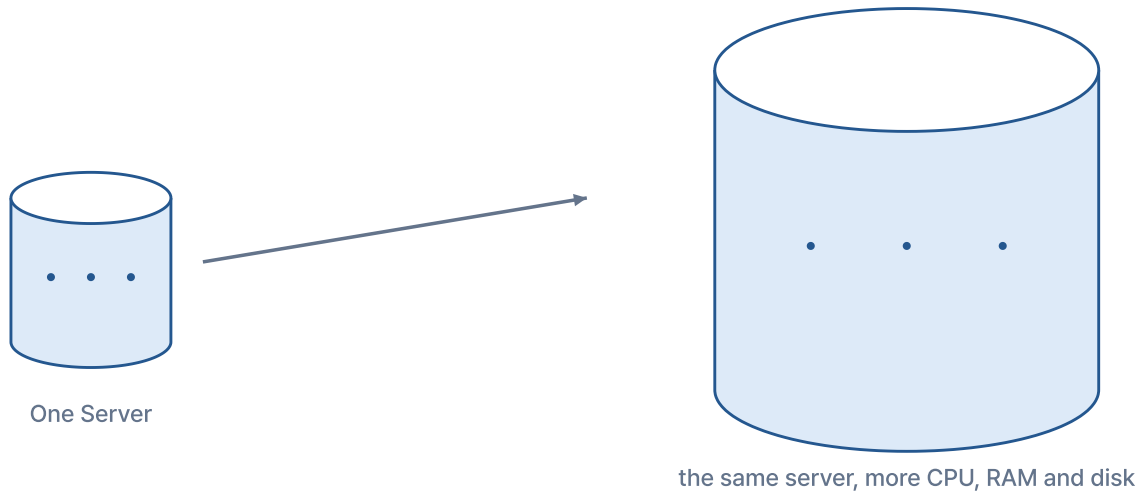
```
{
  "_id": 1,
  "name": "John",
  "email": "john@email.com",
  "age": 28,
  "orders": [
    { "orderId": 101,
      "product": "Laptop",
      "price": 1200 },
    { "orderId": 104,
      "product": "Smartphone",
      "price": 800 }
  ]
}
```

orders live inside the user

Scaling Up

Scale up

A single-server system can scale **vertically** (also known as **scale-up**).



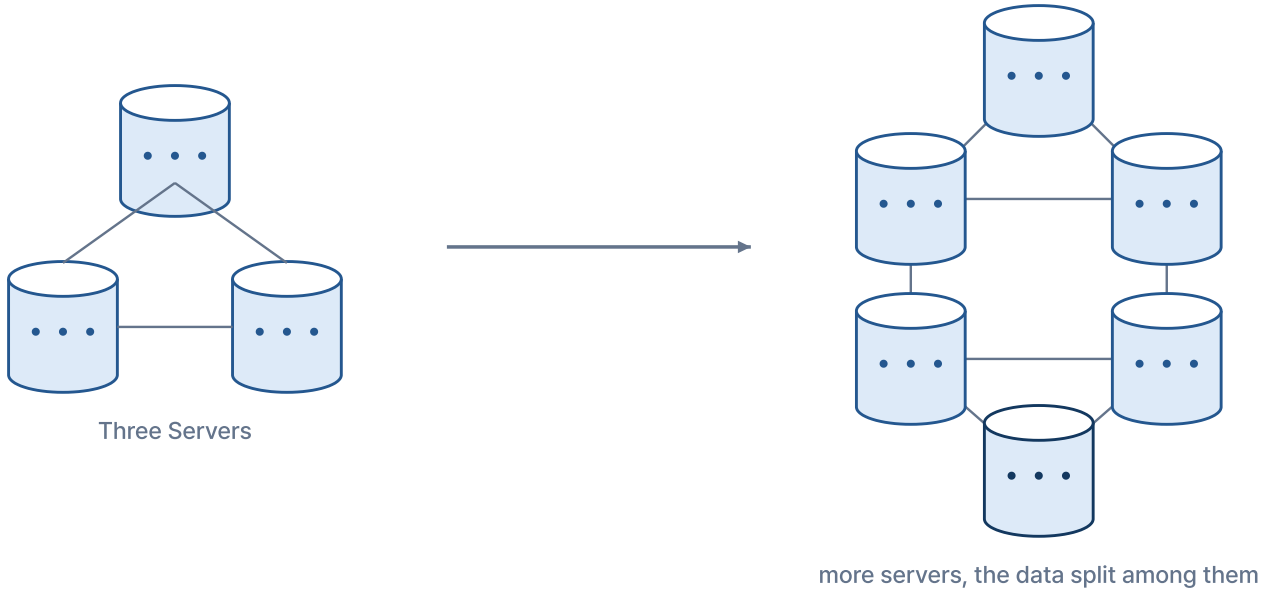
This means improving performance and capacity by adding **more power** (e.g., CPU, RAM, or storage) to a single server.

Scale up: the same server, made bigger

Scaling Out

Scale out

Distributed SQL and many NoSQL systems scale **horizontally** (also known as **scale-out**).



This means increasing capacity by adding **more servers** or nodes to a distributed system.

Scale out: more servers, and the data must be split among them

Partitioning and Replication

Splitting records and copying records solve different problems

Partition: different records

```
Server A: users 1–100
Server B: users 101–200
```

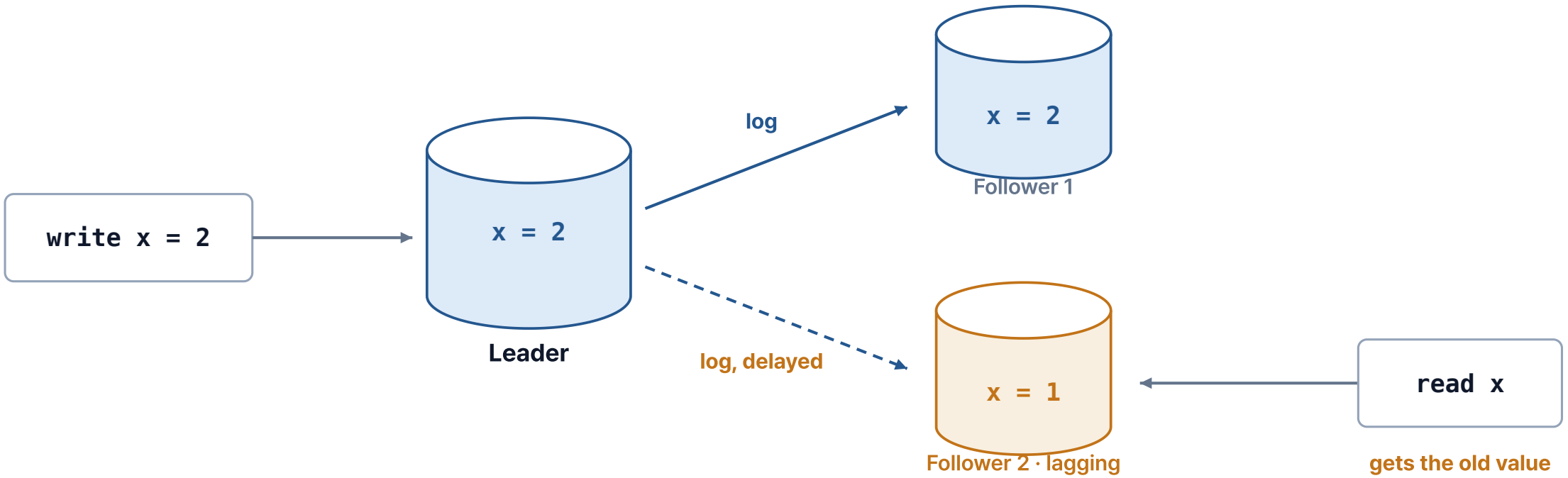
Replicate: copies of records

```
Leader: user 1, value 2
Follower: user 1, value 1
          update pending
```

Partitions divide the data; replicas copy it and must coordinate updates

Asynchronous Leader–Follower Replication

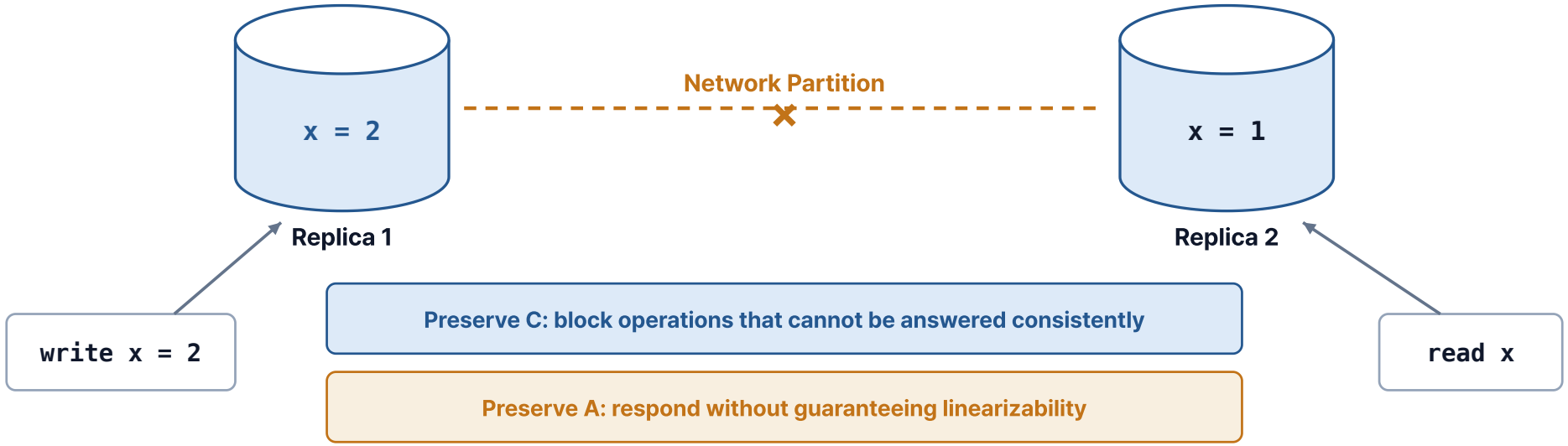
Followers replay the leader's log asynchronously, so a follower read can return an older value



A read from the **lagging follower** returns the old value

CAP Theorem

A partition forces a choice: block some operations, or answer without a consistency guarantee



Without a partition, replication still trades **latency** against **consistency**

During a partition, a system cannot guarantee both **linearizable operations** and **availability**

ACID Transaction Properties

ACID

SQL databases are known for their robust support of **ACID transactions** (Lecture 6).

- **Atomicity**: a transaction is a single, indivisible unit. All of its operations succeed, or none are applied.
- **Consistency**: a transaction takes the database from one valid state to another, keeping constraints, triggers and types intact.
- **Isolation**: the isolation level defines which interactions concurrent transactions may observe.
- **Durability**: once committed, the result survives a crash.

This makes SQL databases ideal where **data integrity** and **consistency** are critical, such as financial systems.

Availability and Eventual Consistency

BASE

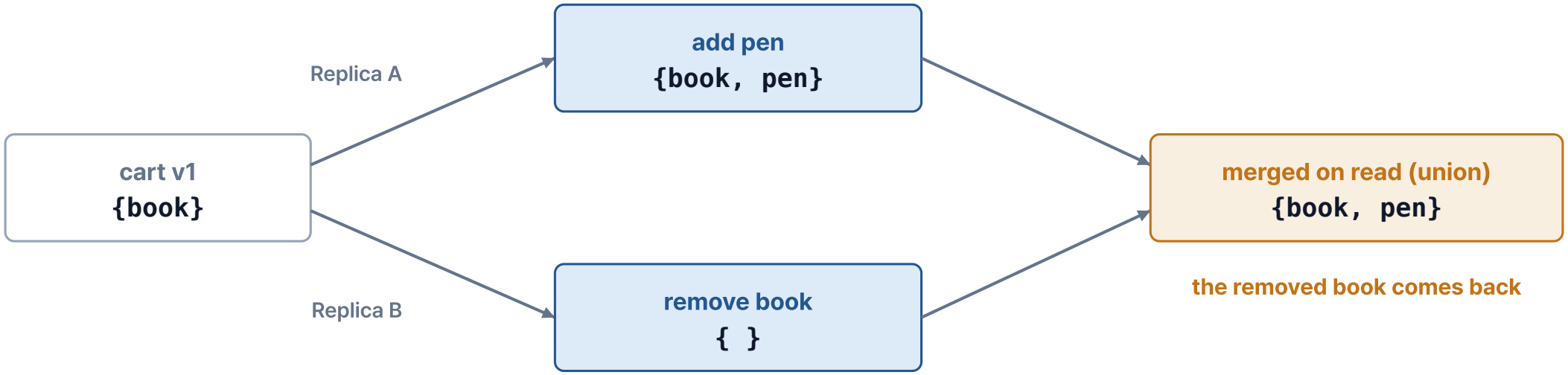
Some distributed systems favor **availability** with eventual consistency, often described as **BASE**:

- **Basically Available**: aim to keep serving requests during failures; success is not unconditional.
- **Soft state**: replicas may differ while updates propagate.
- **Eventually consistent**: replicas converge once writes stop and communication and reconciliation resume.

NoSQL does not imply BASE. MongoDB supports multi-document ACID transactions; guarantees depend on the product and configuration.

Case study: Amazon Dynamo

Dynamo prioritized cart writes during failures and let the application reconcile conflicting versions



Two versions merged by union: a **deleted item can come back**

Check Guarantees per Operation

Write a new price

```
Gizmo
price = 24.99
```

→

Two copies

Copy	Gizmo price
Leader	24.99
Follower	**19.99**

→

Read the follower

```
get("Gizmo")
→ price: 19.99
```

Can this screen accept the older price?

The same product has two observed prices while the update is still propagating

Discussion: Which Store Would You Try?

Start with the question, then choose a candidate

Need	Main question
Login sessions	Retrieve a user by session ID
Device readings	Read one device over a time range
Suspicious account links	Follow shared devices and payment links
Product catalog	Query products with different fields

For each row: **one candidate family and one caveat** to check before choosing

Answer: Which Store Would You Try?

Name the question first; the product comes last

Need	Candidate	Check before choosing
Sessions	Key-value	Expiry and persistence
Device readings	Wide-column	Key distribution and group size
Account links	Graph	Path depth and candidate count
Catalog	Document	Indexes and shared updates

These are starting candidates, not exclusive answers: SQL and **SQL with JSON** remain valid for all four

Takeaway: NoSQL I

1. Choose a model for **the records and access patterns** your application needs
2. Nesting trades read convenience against **shared updates and changing result grain**
3. Treat **model, partitioning, replication and transaction guarantees** as separate decisions