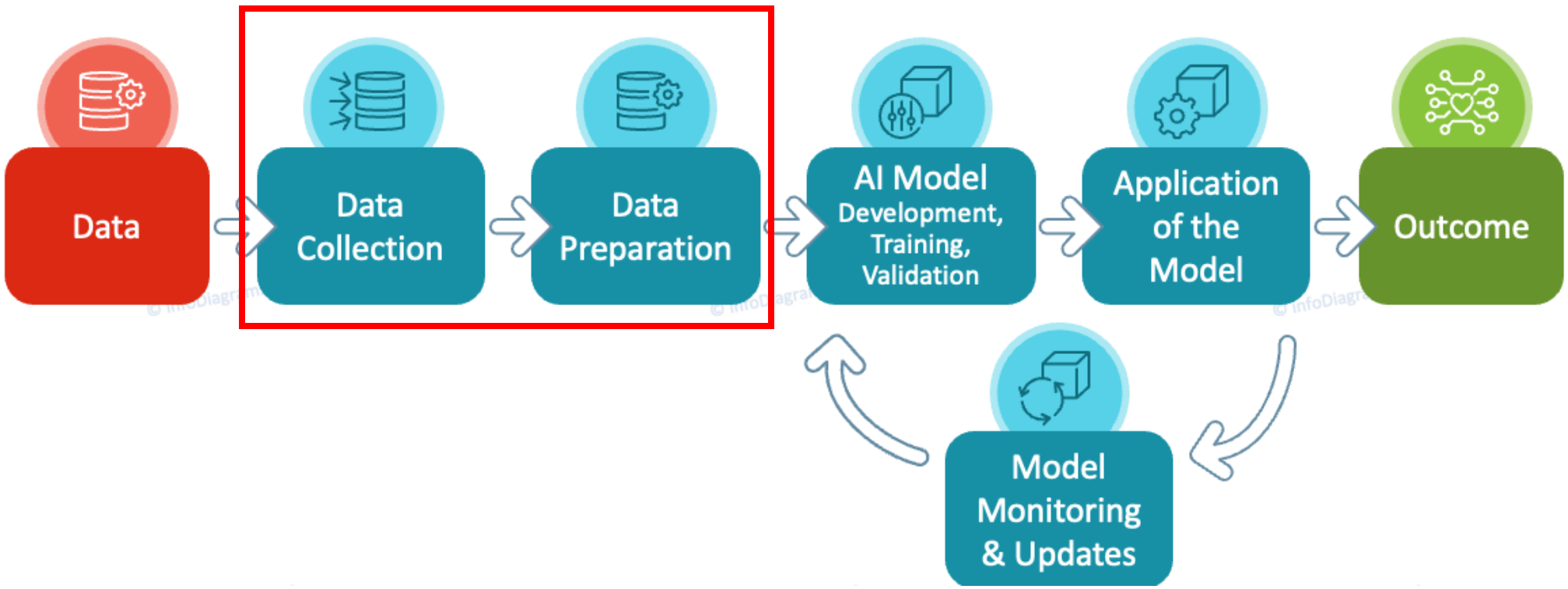
IMEN891K/IDSC501 Data Engineering

10. **NoSQL I**: Landscape

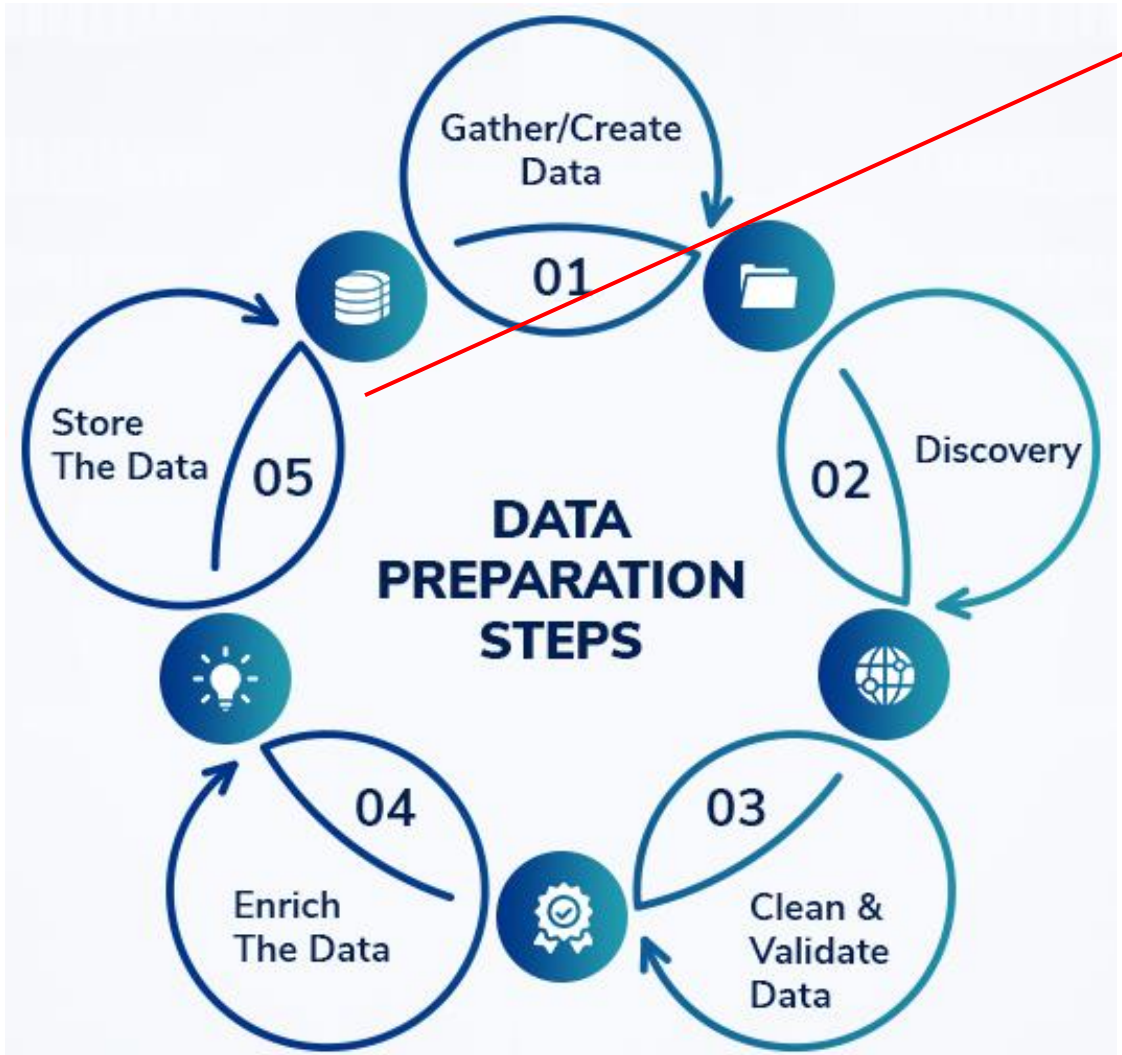
Fall 2025

Instructor: **Sangwoo Mo**

Recap: Course Overview



Course Overview



Lecture 7-15

Disclaimer

The NoSQL part was heavily copy-pasted from Berkely CS186

Introduction to Database Systems

UC Berkeley, Fall 2024



Alvin Cheung
akcheung@cs.berkeley.edu
Pronouns: he/him/his
OH: Tues 11am-noon on zoom

Hi all! I'm an EECS faculty and my group works on data systems research. Looking forward to meeting you all!

Disclaimer

Also heavily copy-pasted from the AlgoMaster blog

SQL vs NoSQL - 7 Key Differences You Must Know

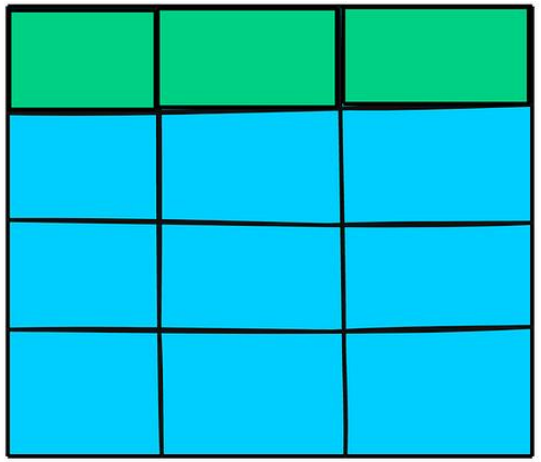


ASHISH PRATAP SINGH

SEP 20, 2024

Why NoSQL?

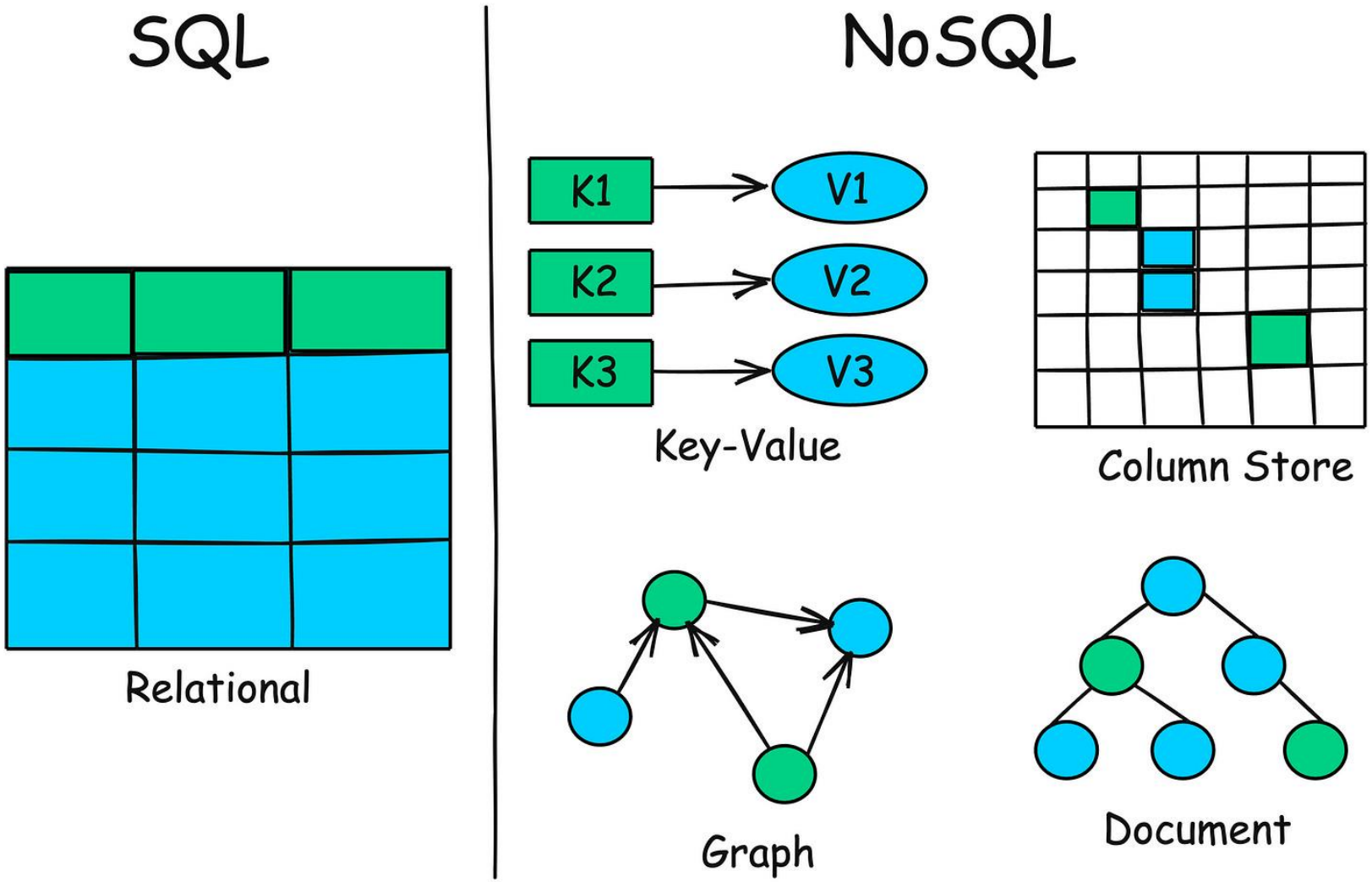
SQL



Relational

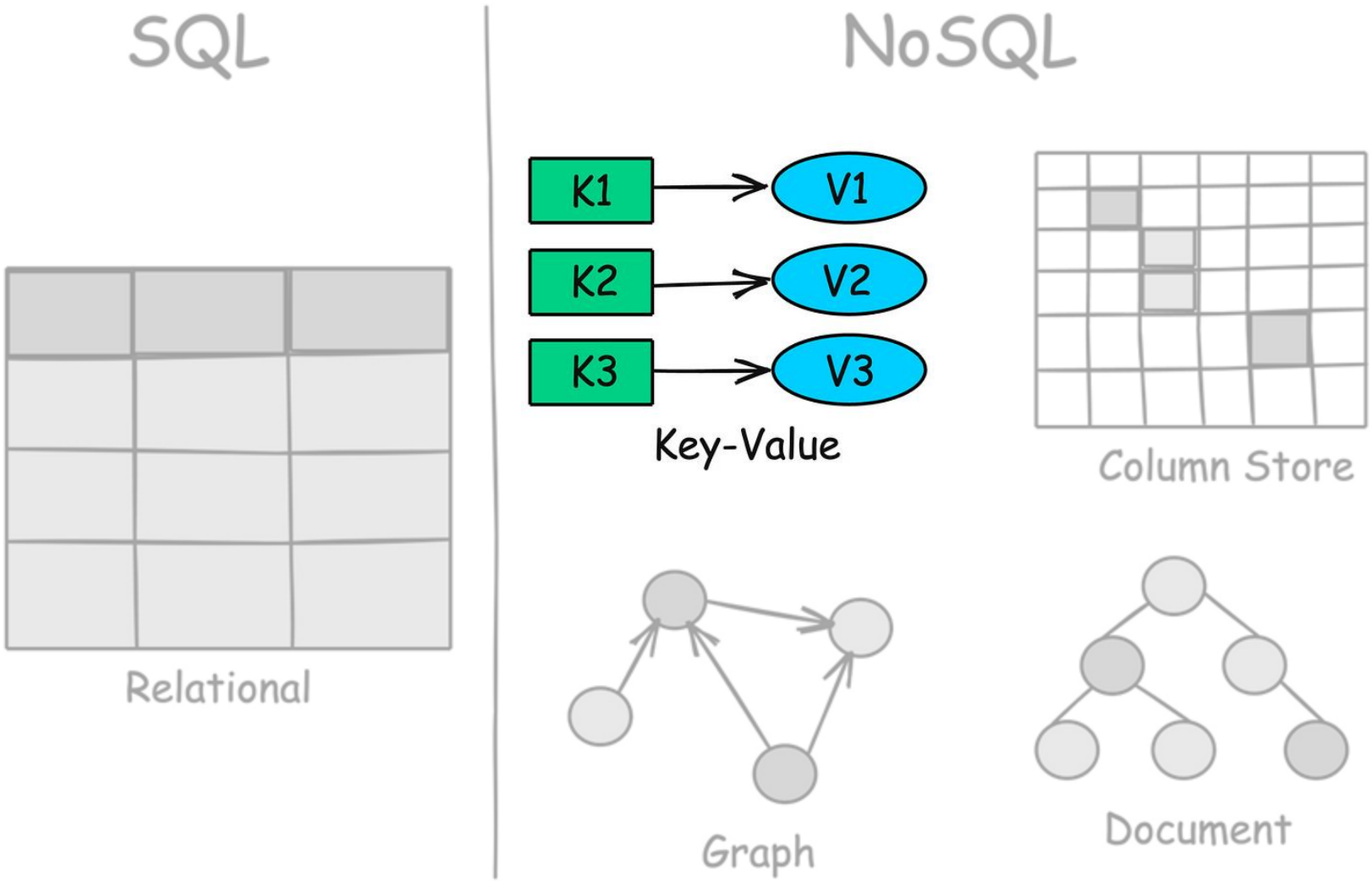
https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

Why NoSQL?



https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

Key-Value Store



https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

Key-Value Store

The key-value model is the simplest NoSQL model, where data is stored as **key-value** pairs. This model works well for applications that need **fast lookups** by a unique key.

For the same **user management system**, user data can be stored as key-value pairs where the key is the `UserID`, and the value is the associated user information.

```
Key: 1
Value: { "name": "John", "email": "john@email.com", "age": 28 }

Key: 2
Value: { "name": "Mike", "email": "mike@email.com", "age": 31 }
```

This model is very efficient for **simple lookups**, but it doesn't support complex querying or relationships between data.

Key-Value Store

- **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

Key-Value Store

- How would you represent the Flights data as key, value pairs?

Key-Value Store

- How would you represent the Flights data as key, value pairs?
- Option 1: key=fid, value=entire flight record
- Option 2: key=date, value=all flights that day
- Option 3: key=(origin,dest), value=all flights between

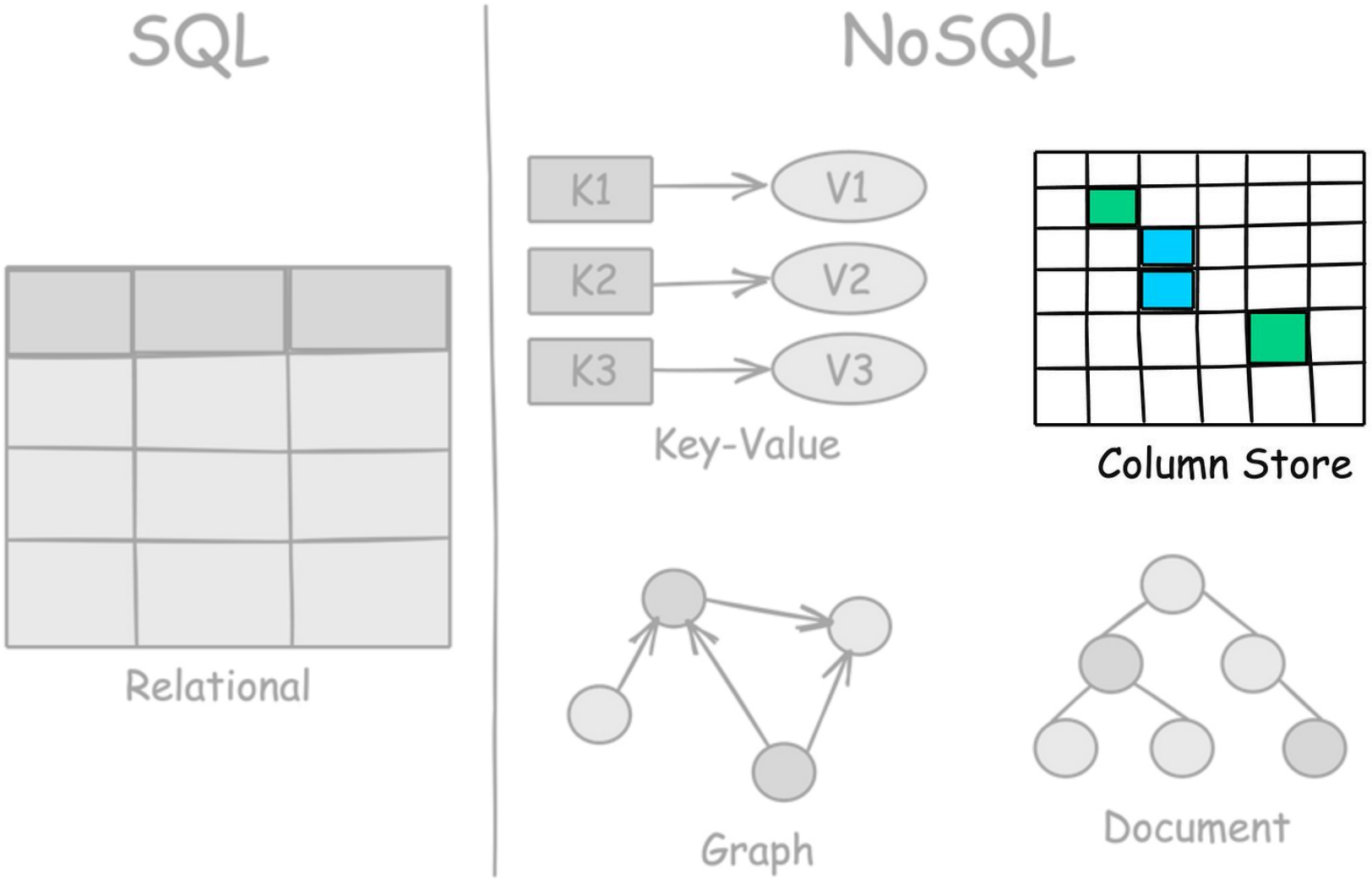
Key-Value Store

☐ include secondary database models

63 systems in ranking, April 2021

Rank Apr 2021	Rank Mar 2021	Rank Apr 2020	DBMS	Database Model	Score Apr 2021	Score Mar 2021	Score Apr 2020
1.	1.	1.	Redis	Key-value, Multi-model	155.89	+1.74	+11.08
2.	2.	2.	Amazon DynamoDB	Multi-model	70.73	+1.84	+6.46
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	33.52	+1.11	+1.47
4.	4.	4.	Memcached	Key-value	25.02	-0.38	-0.31
5.	5.	5.	Hazelcast	Key-value, Multi-model	9.27	+0.18	+0.10
6.	6.	6.	etcd	Key-value	9.22	+0.37	+2.05
7.	7.	↑ 8.	Ehcache	Key-value	7.41	+0.42	+1.14
8.	8.	↓ 7.	Aerospike	Key-value, Multi-model	5.33	-0.29	-1.61
9.	9.	9.	Riak KV	Key-value	5.08	+0.01	-0.16
10.	10.	↑ 11.	ArangoDB	Multi-model	4.77	-0.28	-0.11
11.	↑ 12.	↓ 10.	Ignite	Multi-model	4.37	-0.08	-0.59
12.	↓ 11.	12.	OrientDB	Multi-model	4.23	-0.48	-0.29
13.	13.	13.	Oracle NoSQL	Multi-model	3.89	-0.30	+0.08

Wide-Column Store



https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

Wide-Column Store

In the column-family model, data is organized into rows and columns, but unlike the relational model, each row can have a **variable number of columns.** It is optimized for **fast querying** and **large-scale distributed storage**.

Example:

Let's assume we're building a **user activity tracking system** that stores the actions users take on a website, such as page views and purchases.

Each user has a unique `UserID`, and their activity (page views and purchases) is stored in a column-family.

Column Family: UserActivity

Row Key (UserID)	Page1	Page2	Purchase1	Purchase2
1	Viewed: Home Page	Viewed: Products	Product: Laptop ($1200)	Product: Phone ($800)
2	Viewed: Home Page	Viewed: Contact Us	Product: TV ($1500)	

Wide-Column Store

Row Key (**UserID**): The unique identifier for each user.

Page1, **Page2**, **Purchase1**, **Purchase2**: Columns representing the user's activity, which can vary from row to row.

- In **row 1**, user 1 has viewed two pages and made two purchases.
- In **row 2**, user 2 has viewed two pages and made one purchase.
- In **row 3**, user 3 has viewed one page and made one purchase.

Each user (row) can have a **variable number of columns**, and different users may have different activities stored in each row.

No predefined schema is required, which means new columns (such as additional page views or purchases) can be added dynamically for each user

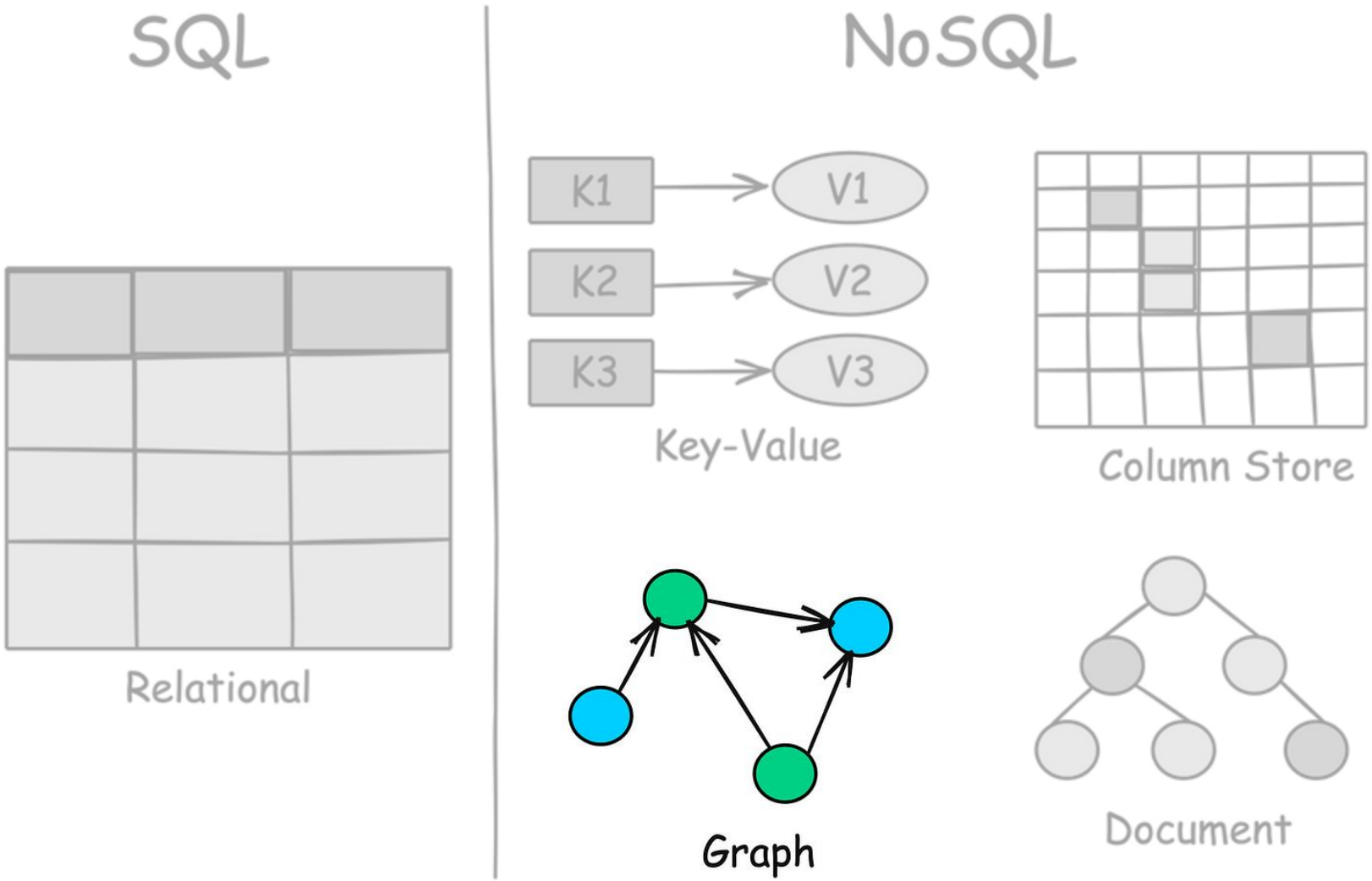
This model allows for high write throughput and distributed storage but doesn't enforce strict relationships like the SQL relational model.

Wide-Column Store

- Also called wide-column stores
- Based on Google's BigTable
- HBase is an open source implementation of BigTable

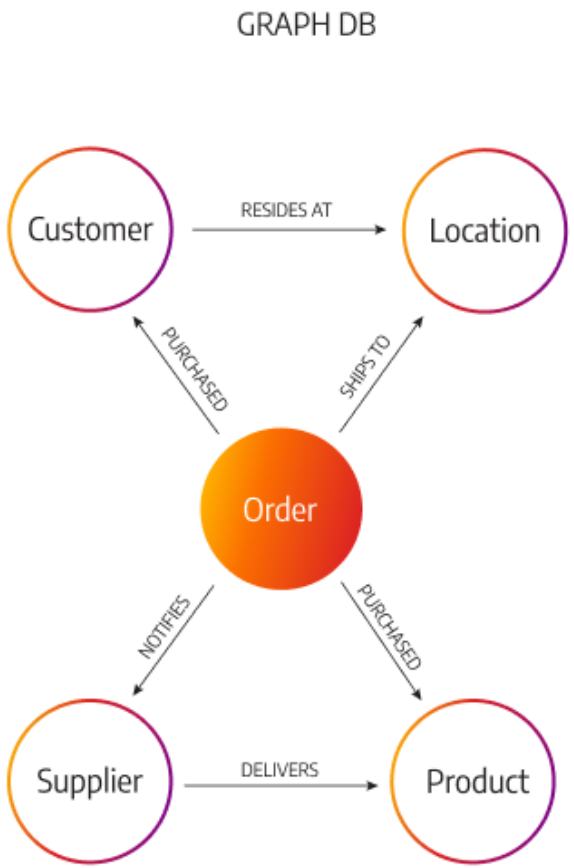
- Data model:
  - Variant 1: key = rowID, value = record
  - Variant 2: key = (rowID, columnID), value = field
    - Or multiple columnIDs in the key

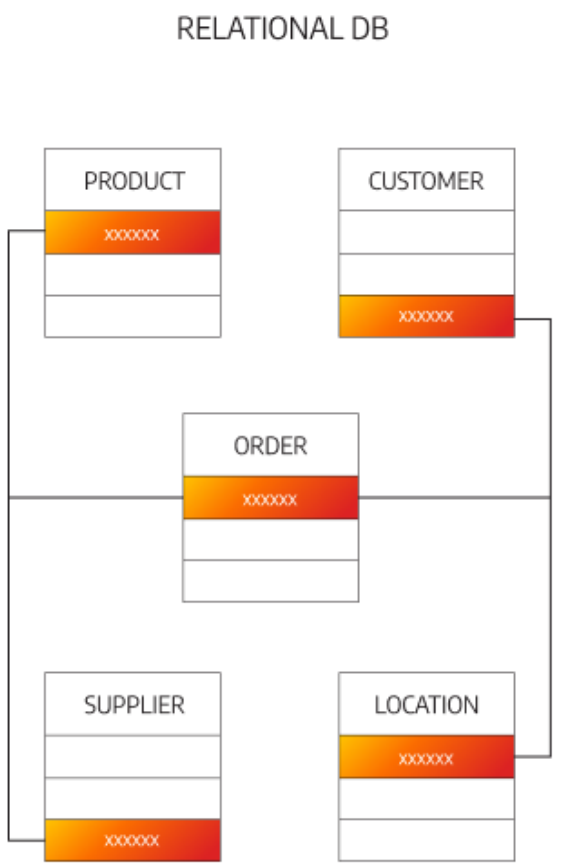
Graph Store



https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

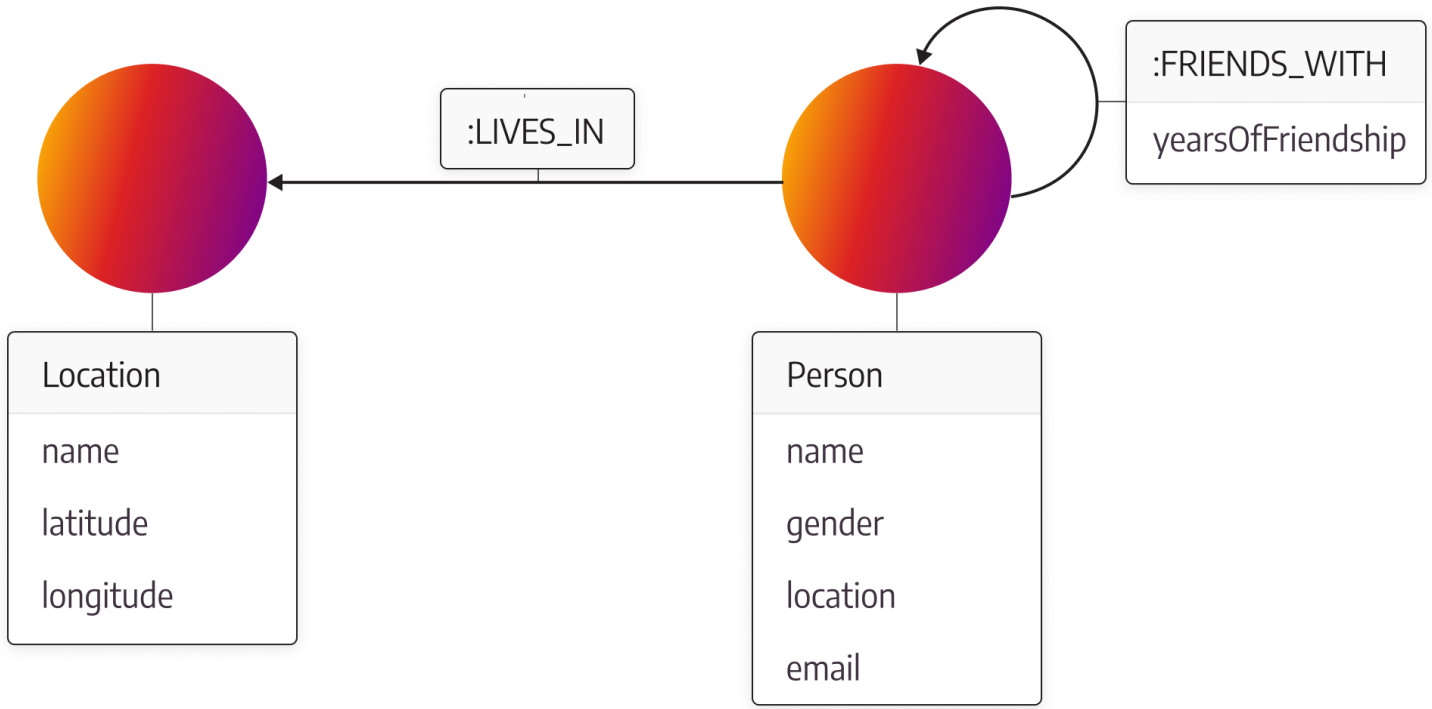
Graph Store





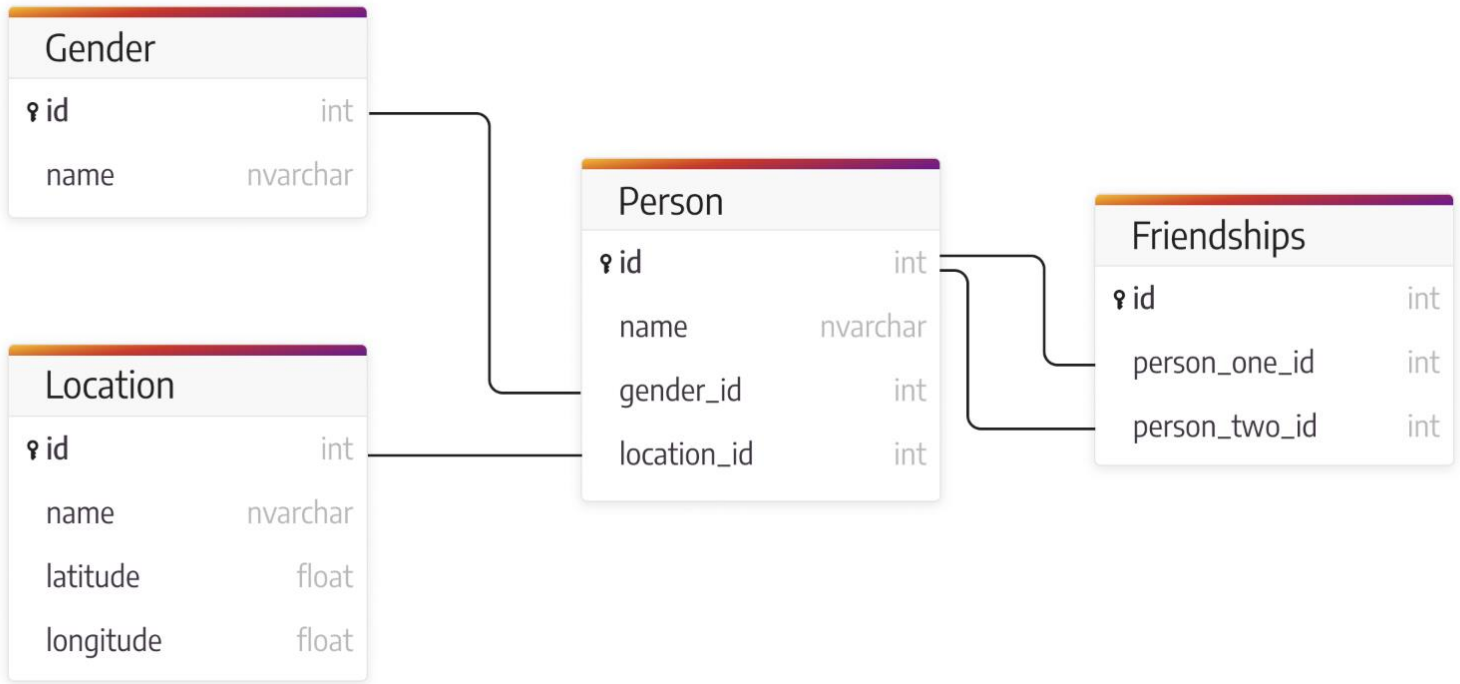
https://memgraph.com/blog/graph-database-vs-relational-database

Graph Store


:LIVES_IN
:FRIENDS_WITH
yearsOfFriendship
Location
name
latitude
longitude
Person
name
gender
location
email

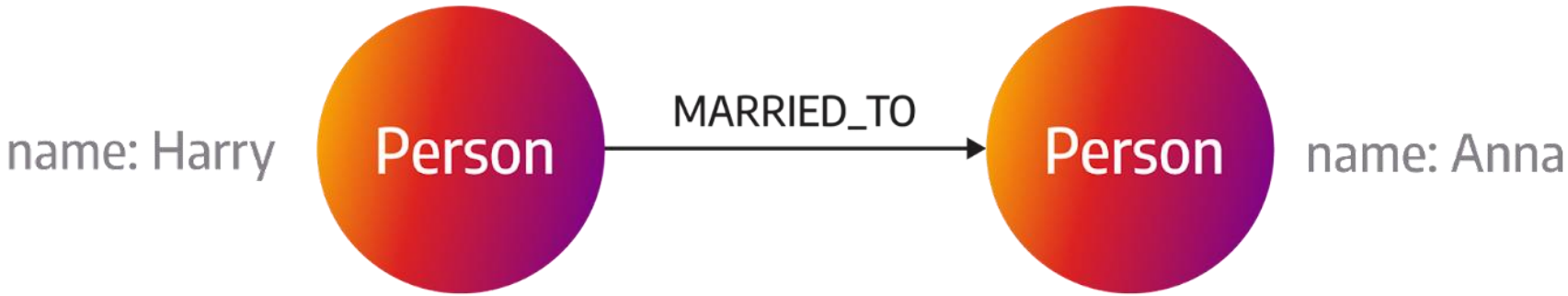
Graph Store

Each person is connected to other people through friendships, and to model this relationship in a relational database, a table needs to be created. If there were different kinds of connections schema would have to be changed accordingly. A relational database isn't suited for this specific use case because the focus isn't on the data but on the relationships within it.



Graph Store

```
CREATE (p1:Person {name: 'Harry'}), (p2:Person  {name: 'Anna'})
CREATE (p1)-[r:MARRIED_TO]->(p2)
RETURN p1, r, p2;
```



Graph Store

With breadth-first search and depth-first search, it is possible to obtain the descendants in a specific order. Here are the SQL and Cypher queries to find all descendants of Thurimbert:

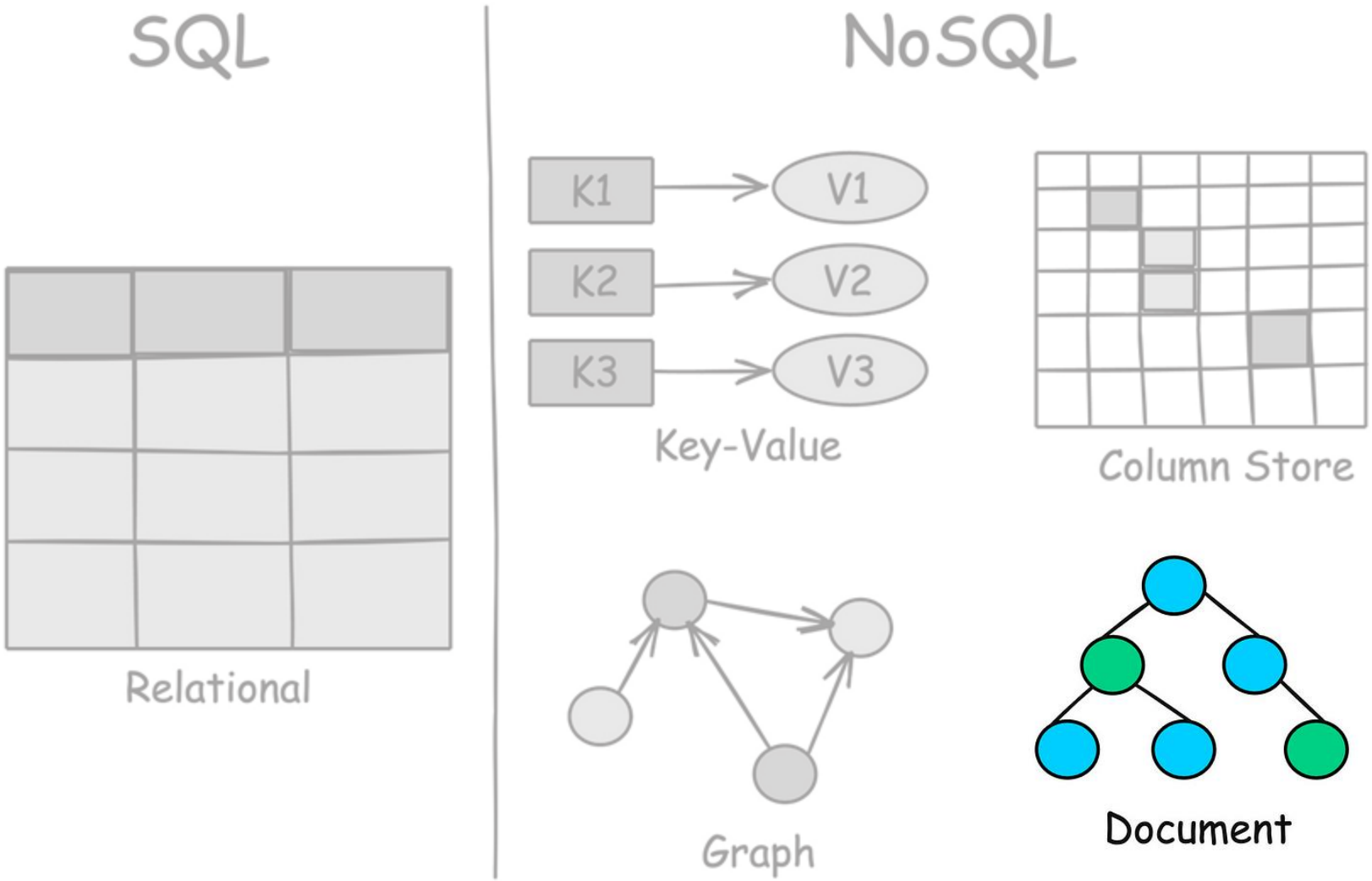
SQL

```sql
WITH RECURSIVE descendants AS
(
SELECT person
FROM tree
WHERE person='Thurimbert'
UNION ALL
SELECT t.person
FROM descendants d, tree t
WHERE t.parent=d.person
)
SELECT * FROM descendants;
```

Cypher

```cypher
MATCH path=(n:Person {name: 'Thurimbert'})-[*]->(m)
RETURN m;
```

Document Store



https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

Document Store

- In Key, Value stores, the Value is often a very complex object
  - Key = '2010/7/1', Value = [all flights that date]

- Better: *value* to be structured data
  - JSON or Protobuf or XML
  - Called a "document" but it's just data

We will discuss JSON

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
  - Rigid flat structure (tables)
  - Schema must be fixed in advanced
  - Binary representation: good for performance, bad for exchange
  - Query language based on Relational Algebra

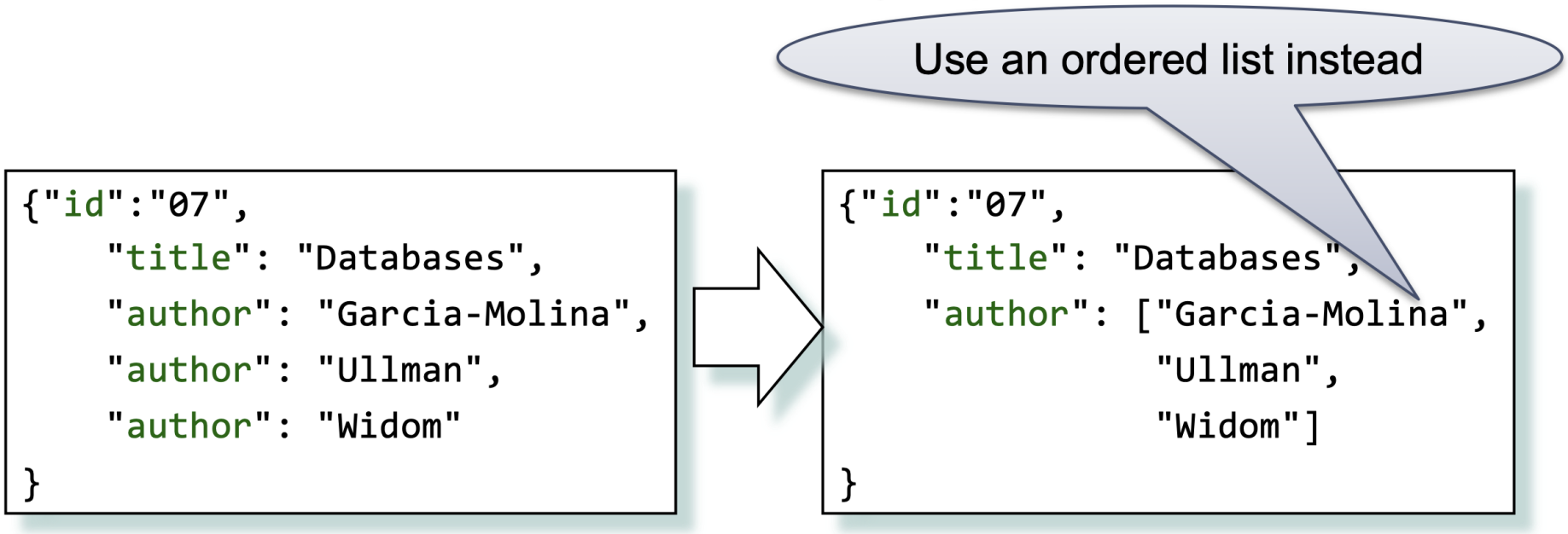
- Semistructured data model / JSON
  - Flexible, nested structure (trees)
  - Does not require predefined schema (“self-describing”)
  - Text representation: good for moving data between apps, bad for performance
  - Query languages emerging

JSON Types

- Primitive: number, string, Boolean, null
- Object: collection of name-value pairs:
  - `{“name1”: value1, “name2”: value2, …}`
  - “name” is also called a “key”
- Array: *ordered* list of values:
  - `[obj1, obj2, obj3, ...]`

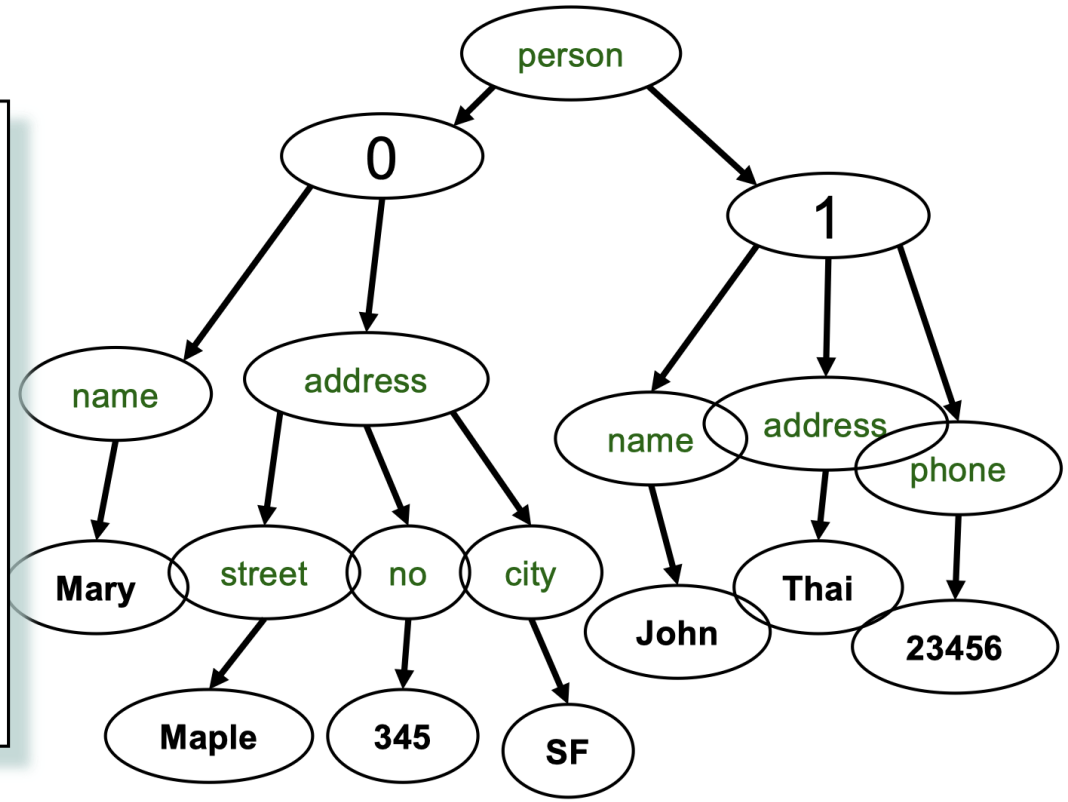
Avoid Using Duplicate Keys

The standard allows them, but many implementations don't



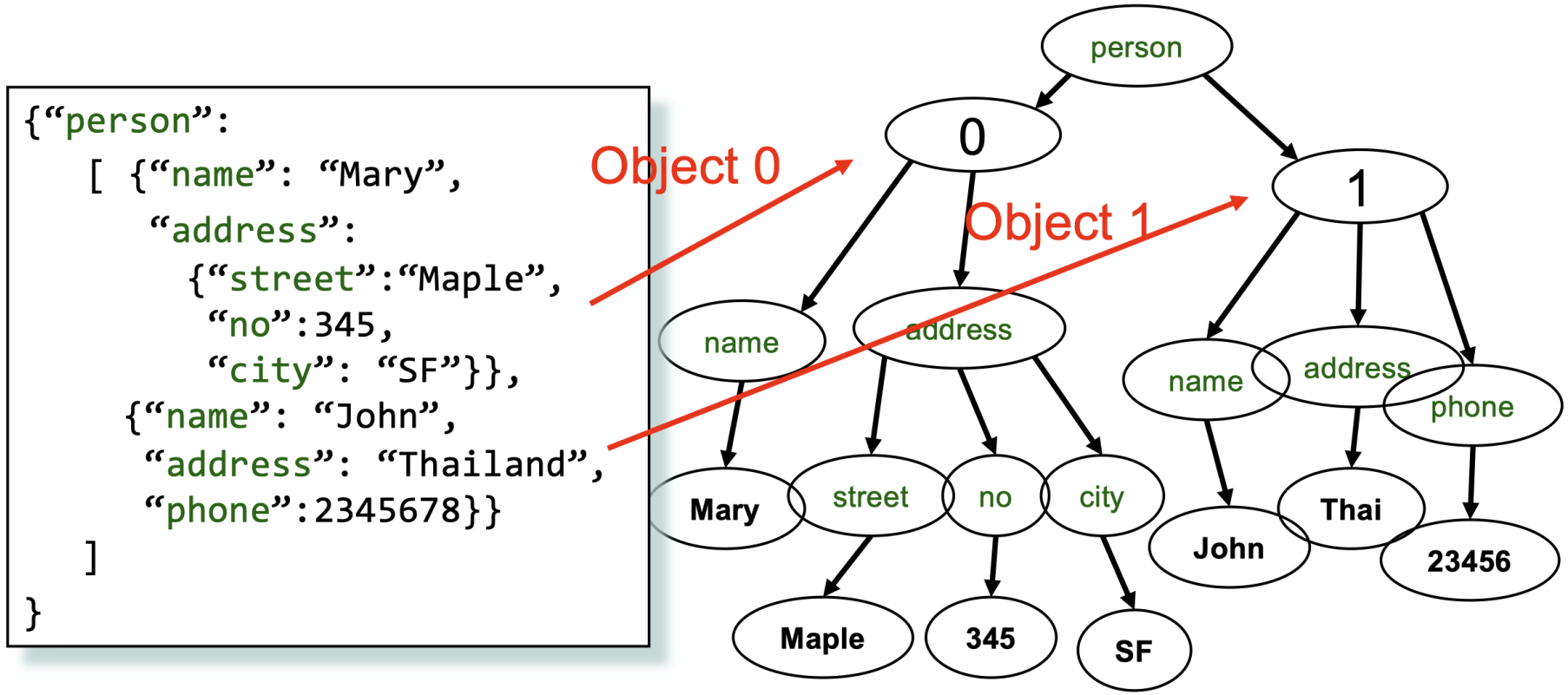
JSON Semantics: a Tree!

```
{“person”:
   [ {“name”: “Mary”,
      “address”:
        {“street”:“Maple”,
         “no”:345,
         “city”: “SF”}},
     {“name”: “John”,
      “address”: “Thailand”,
      “phone”:2345678}}
   ]
}
```



JSON Semantics: a Tree!

```
{“person”:
   [ {“name”: “Mary”,
      “address”:
        {“street”:“Maple”,
         “no”:345,
         “city”: “SF”}},
     {“name”: “John”,
      “address”: “Thailand”,
      “phone”:2345678}}
   ]
}
```



Recall: arrays are *ordered* in JSON!

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

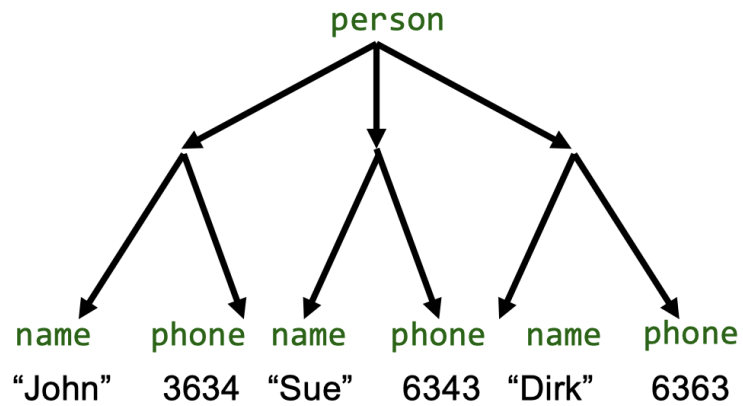
Storing JSON in RDBMS

- Using JSON as a data type provided by RDBMSs
  - Declare a column that contains either json or jsonb (binary)
    - `CREATE TABLE people (person json)` [or jsonb for binary]
  - In our previous example, we will have one row per person
    - i.e., a row corresponding to that person's attributes
  - Queries now mix relational and semi-structured syntax
    - ```
      SELECT * FROM people
      WHERE person @> `{"name": "Mary"}';
      ```
- Translate JSON documents into relations
      ```

Mapping Relational Data to JSON

Person

name	phone
John	3634
Sue	6343
Dirk	6363




```
{“person”: [
   {“name”: “John”, “phone”:3634},
   {“name”: “Sue”,  “phone”:6343},
    {“name”: “Dirk”, “phone”:6383}
   ]
}
```

Mapping Relational Data to JSON

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":3646,
   "Orders":[
    {"date":2002,"product":"Gizmo"},
    {“date”:2004,"product":"Gadget"}
   ]
  },
  {"name": "Sue",
   "phone":6343,
   "Orders":[
   {"date":2002,"product":"Gadget"}
   ]
  }
 ]
}
```

Mapping Relational Data to JSON

Many-many relationships are more difficult to represent

Person

name	phone
John	3634
Sue	6343

Product

prodName	price
Gizmo	19.99
Phone	29.99
Gadget	9.99

Orders

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

Options for the JSON file:

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

Mapping Semi-structured Data to Relations

- Missing attributes:

```
{“person”:
     [{“name”:“John”, “phone”:1234},
      {“name”:“Joe”}]
}
```

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	NULL

Mapping Semi-structured Data to Relations

- Repeated attributes

```
{“person”:
     [{“name”:”John”, “phone”:1234},
      {“name”:”Mary”, “phone”:[1234,5678]}]
}
```

Two phones !

- Impossible in one table:

name	phone	
Mary	2345	3456

???



Mapping Semi-structured Data to Relations

- Attributes with different types in different objects

```
{“person”:
     [{“name”:“Sue”, “phone”:3456},
      {“name”:{“first”:“John”, “last”:“Smith”},“phone”:2345}
     ]
}
```

Structured name !

- Nested collections
- Heterogeneous collections

- These are difficult to represent in the relational model

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)

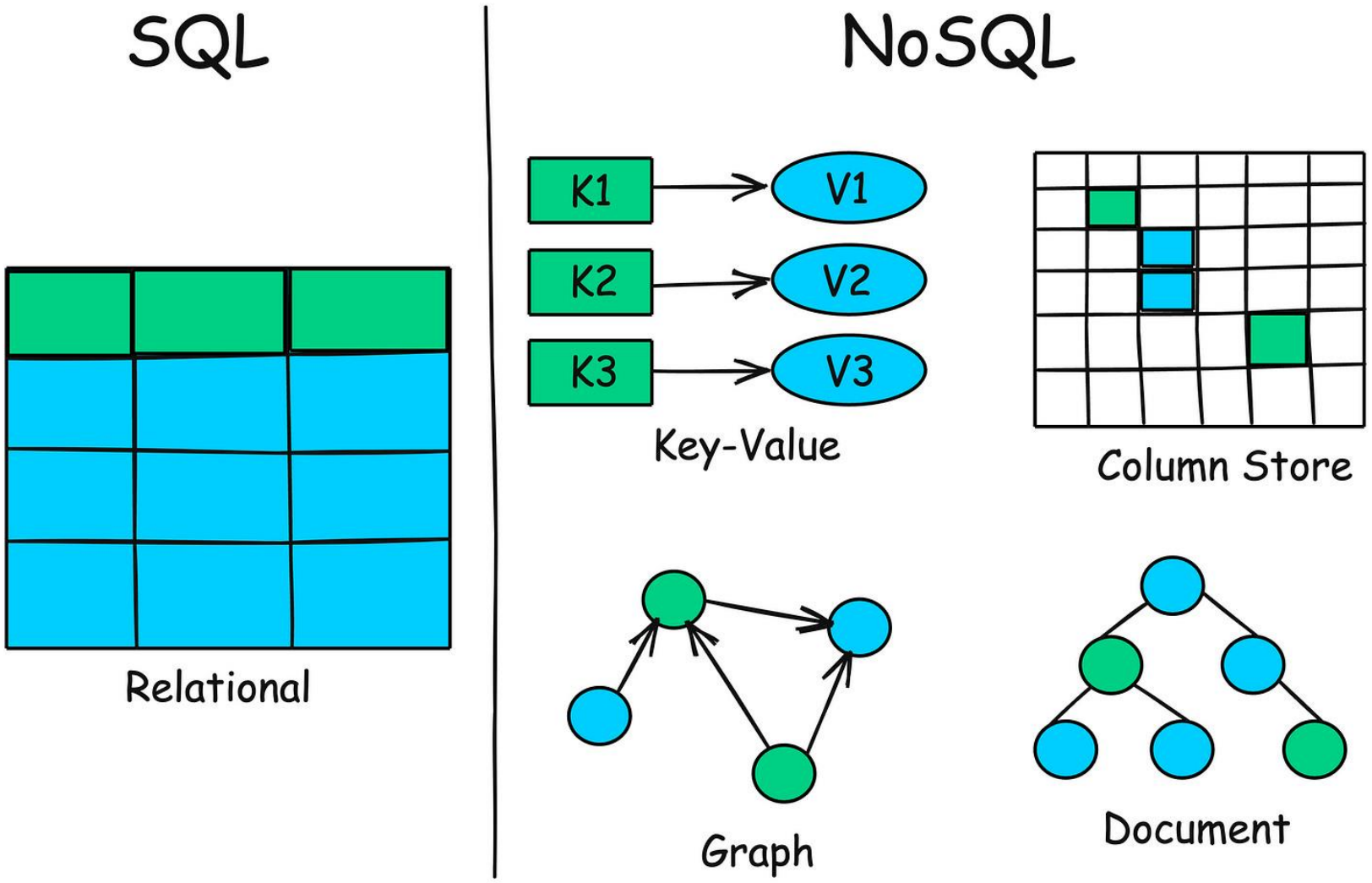
- Increasingly, systems use them as a data model for DBs:
  - SQL Server supports for XML-valued relations
  - CouchBase, MongoDB, Snowflake: JSON
  - Dremel (BigQuery): Protobuf

Semi-Structured Data Model

- Several file formats: JSON, protobuf, XML
- Data model = Tree

- Query language take non first normal form into account as we will see
  - Various “extra” constructs introduced as a result
    - Nesting & Unnesting, strict aggregates, splitting

Why NoSQL? – Data Model



https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

Why NoSQL? – Schema-less

Let's take the example of a **user management system**. In SQL, we first define the structure (schema) of the `Users` and `Orders` tables before adding data.

```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100),
    Age INT
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    UserID INT,
    Product VARCHAR(100),
    Price DECIMAL(10, 2),
    FOREIGN KEY (UserID) REFERENCES Users(UserID)
);
```

Why NoSQL? – Schema-less

In NoSQL databases, schema changes are much simpler because the schema is **dynamic**. You can add new fields to individual records without affecting other records or requiring a schema migration.

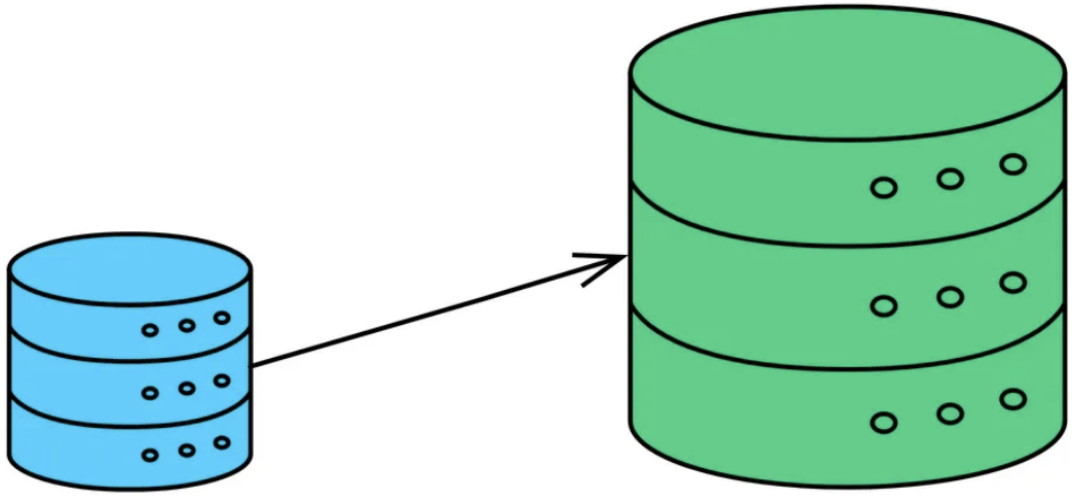
Example: User and Orders in a Document Database (e.g., MongoDB)

In a document-based NoSQL database, such as MongoDB, 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
        }
    ]
}
```

Why NoSQL? – Scalability

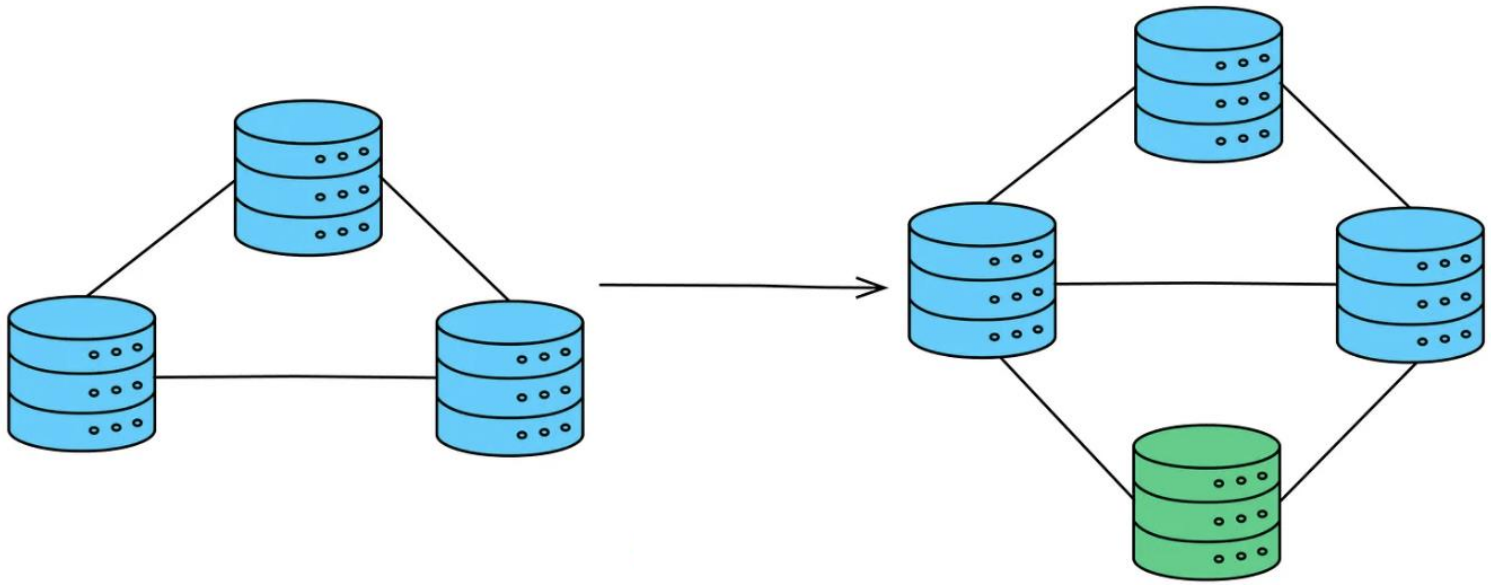
SQL databases are typically designed to scale **vertically** (also known as **scale-up**).



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

Why NoSQL? – Scalability

NoSQL databases are designed to scale **horizontally** (also known as **scale-out**).



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

SQL vs NoSQL

SQL databases are known for their robust support of **ACID transactions**.

- **Atomicity**: A transaction is treated as a single, indivisible unit. Either all operations within the transaction succeed, or none of them are applied (all-or-nothing).
- **Consistency**: Transactions take the database from one valid state to another, maintaining all predefined rules such as constraints, triggers, and data types.
- **Isolation**: Multiple transactions can occur concurrently without interfering with each other, ensuring that one transaction's intermediate state is not visible to other transactions.
- **Durability**: Once a transaction is committed, its results are guaranteed to persist, even in the event of a system crash.

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

SQL vs NoSQL

NoSQL databases typically do not prioritize full ACID transactions due to the need for high **availability** and **scalability** in distributed environments.

Instead, many NoSQL databases follow the **BASE** model:

- **Basically Available**: The system guarantees availability, meaning that data can always be read or written, even if some nodes in the distributed system are unavailable.
- **Soft state**: The system may be in a temporarily inconsistent state, but eventual consistency will be reached over time.
- **Eventually consistent**: Over time, the system will become consistent, though it may not happen immediately. This trades immediate consistency for higher availability.

The BASE model is designed for scenarios where strict consistency is not required, and **performance** and **availability** are more important, such as real-time data analytics, social media platforms, or large-scale distributed applications.