Ministry of Higher Education and Scientific Research
University of Mohamed Boudiaf, Msila

Faculty of Mathematics and Computer Science
Department of Computer Science

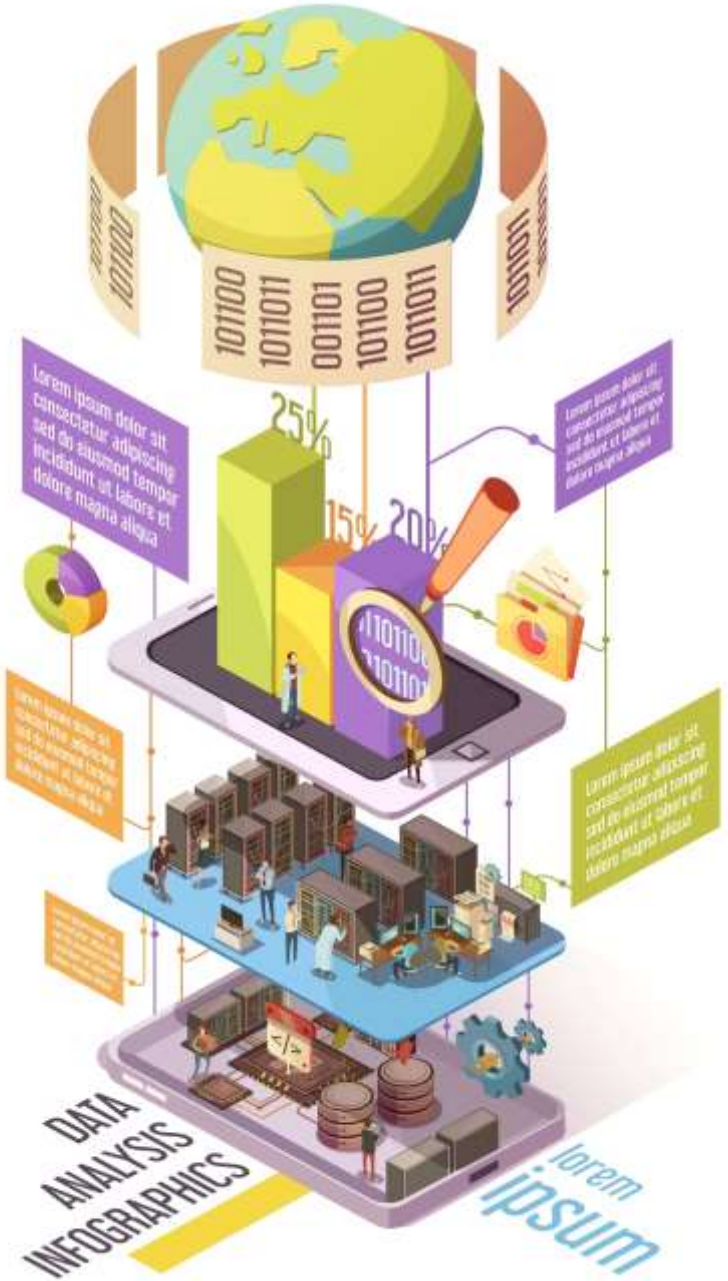
Mater 1

Big Data Analytics

Dr. N.CHALABI

October 2025

Big Data Architecture & Ecosystem



NoSQL Databases

Relational Databases (SQL) – The Foundation

SQL databases have been the backbone of data management since the 1970s.
Strengths:

- Structured data with fixed schemas
- Strong consistency & ACID transactions

Worked well for: banking, payroll, inventory, business apps

The Challenges of the 2000s

The digital world changed with:

- 🌍 Internet growth
- 📱 Social media & mobile apps
- 🔌 IoT devices

New requirements:

- **Unprecedented scale** (petabytes of data daily)
- **New formats** (text, JSON, images, graphs, streams)
- **High speed** (millions of reads/writes per second)

Problem: Traditional SQL struggled to scale and adapt

The Birth of NoSQL

Why SQL struggled:

- Vertical scaling = expensive (bigger servers instead of many cheap ones)
- Rigid schemas slowed down rapid development
- Hard to support unstructured/real-time web data

NoSQL ("Not Only SQL") emerged to:

- Allow **flexible schemas** → store changing data formats (e.g., JSON, logs, sensor data)
- Enable **horizontal scaling** → distribute data across many servers easily
- Provide **specialized models** → choose the right database for the right job

Key innovators:

- Google **Bigtable (2006):** built to index the entire web efficiently
- Amazon **Dynamo (2007):** ensured shopping carts were always available, even with failures
- Facebook **Cassandra (2008):** scaled inbox messages to billions of users

The NoSQL Movement

🌟 Turning Point

The shift from rigid **SQL systems** to **databases designed for the web era**.

Modern applications needed:

- Massive scalability for global users
- Flexibility to store diverse data types
- Real-time performance

Key Focus Areas

Scalability 🔍

- Handle **petabytes of data** distributed over **thousands of servers**.
- Example: Facebook storing billions of user messages.

Agility ⚡

- Support **rapid development** without redesigning schemas.
- Example: Developers can add new fields to JSON documents without downtime.

Performance 🚀

- Real-time **reads/writes** at massive scale for web and mobile apps.
- Example: Amazon Dynamo ensures shopping carts are always available.

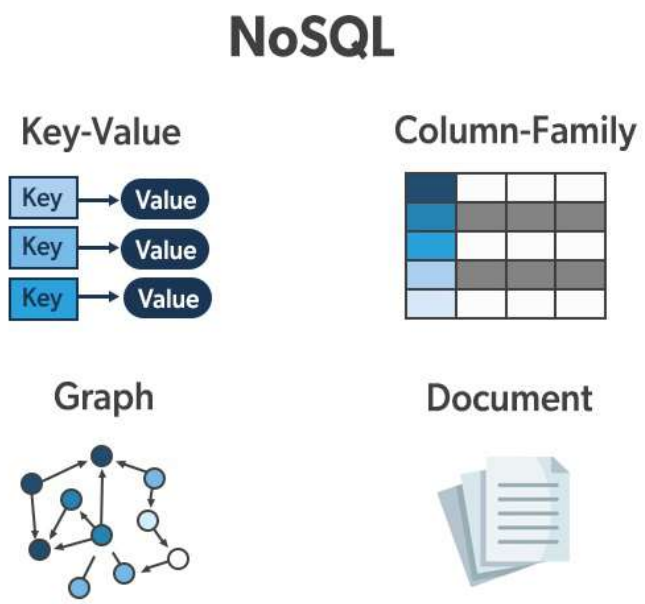
Four Major NoSQL Types

1.Key-Value Stores
   1. Like a giant dictionary: store data as **key → value**.
   2. Use case: Caching, session storage.
   3. Examples: Redis, DynamoDB

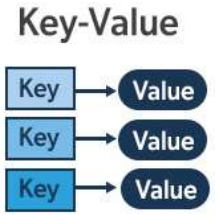
2.Document Databases
   1. Store **semi-structured JSON/XML documents**.
   2. Schema can evolve over time.
   3. Examples: MongoDB, CouchDB

3.Wide-Column Stores
   1. Tables with **many columns** optimized for analytics.
   2. Efficient for **large-scale aggregation** and time-series data.
   3. Examples: Cassandra, HBase

4.Graph Databases
   1. Represent **nodes (entities) and edges (relationships)**.
   2. Ideal for social networks, recommendation systems.
   3. Example: Neo4j



✓ Key-Value

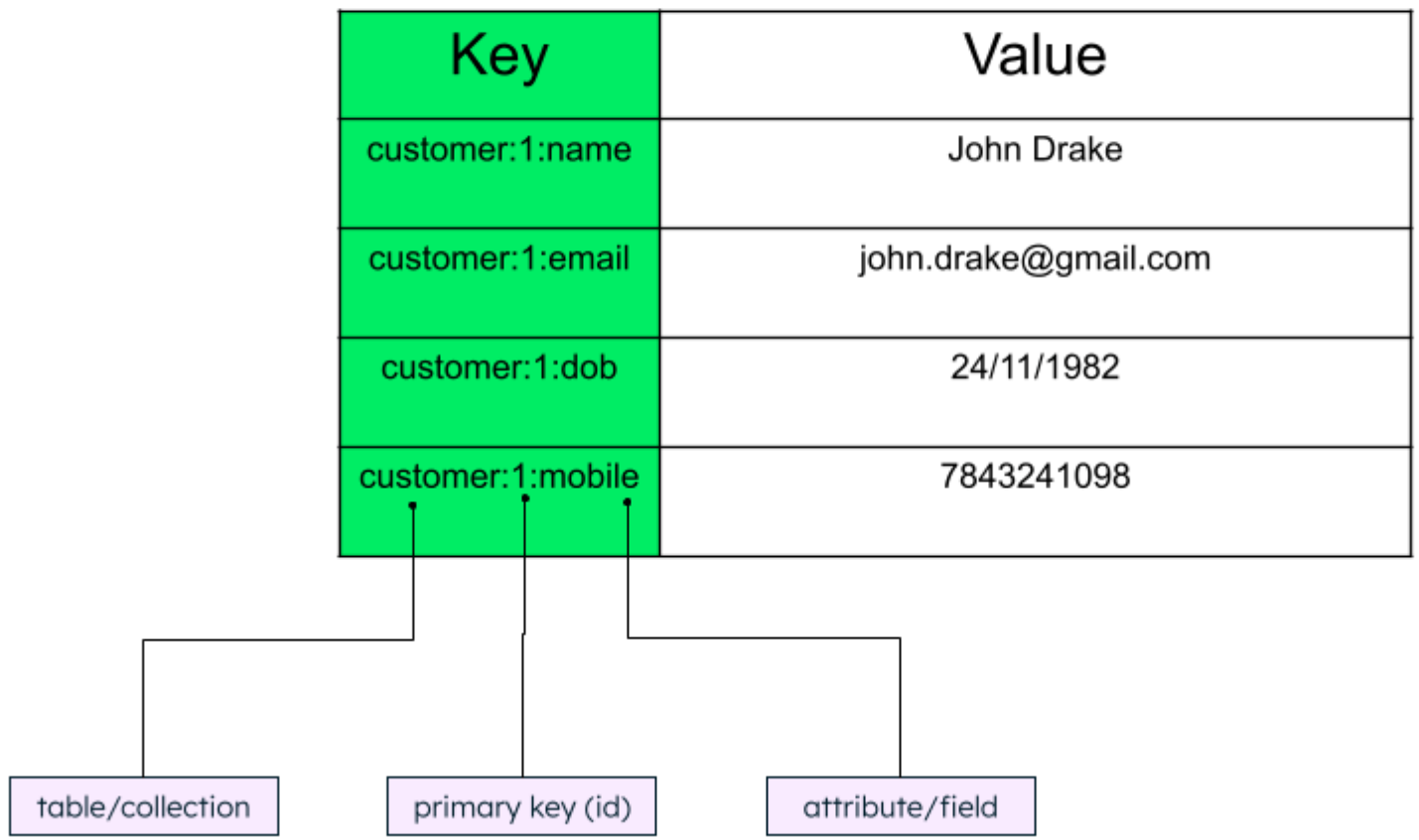


Key value databases, also known as **key value stores**, are NoSQL database types where data is stored as key value pairs and optimized for reading and writing that data. The data is fetched by a **unique key** or a number of unique keys to retrieve the associated value with each key.

The values can be simple data types like **strings** and **numbers** or **complex objects**. The unique key can be anything. Most of the time, it is an **id field**, since that's the unique field in all the documents. To group related items, you can also add a common prefix to the key. The general structure of a key value pair is key: value.

For example, "name": "John Drake."

Key value database storage

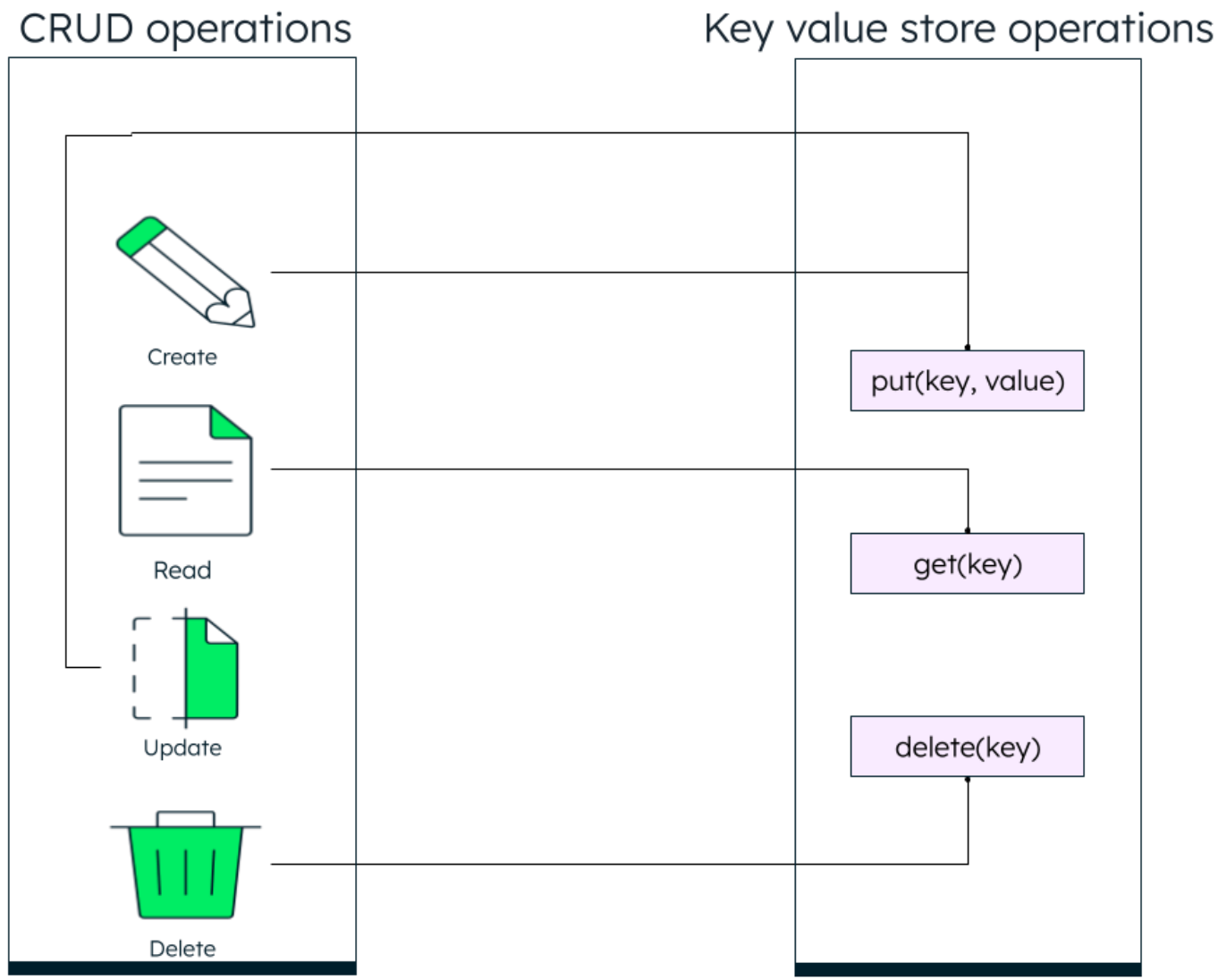


How do key value databases work?

A key value database, AKA key value store, associates a value (which can be anything from a number or simple string to a complex object) with a unique identifier (key), which is used to keep track of the object. In its simplest form, a key value store is like a dictionary/array/map object as it exists in most programming paradigms but is managed by a database management system (DBMS).

There are three main operations performed by a key value database:

- put(key, value): insert or update a value into the database.
- get(key): read a value from the database.
- delete(key): delete a value from the database.


CRUD operations
Key value store operations
Create
Read
Update
Delete
put(key, value)
get(key)
delete(key)

Key value database schema

Unlike relational databases, key value data stores do not follow a specific schema, making them flexible. This makes them a good choice for unstructured and semi structured data. Storing and retrieving data becomes much simpler when it is done in a key value store when compared to that of relational database table structure.

In the below image, due to the JSON structure, all the data is grouped together. You can store anything from simple types to complex types as values. While fetching the data, you can retrieve the individual records by iterating through the JSON data.

Relational database storage

customer_id	name	email	dob	mobile
1	John Drake	john.drake@gmail.com	24/11/1982	7843241098
2	Mary Chile	mary.chile@outlook.com	05/06/1981	8903424531
3	Mac Adams	mac_1979@gmail.com	23/04/1979	0920421454
4	Jill Smith	jellyjill@gmail.com	14/02/1987	8795092014

Key value database storage

Key	Value

customer_1	{
	"name": "John Drake",
	"email": "john.drake@gmail.com",
	"dob": "24/11/1982",
	"mobile": 7843241098
	}

customer_2	{
	"name": "Mary Chile",
	"email": "mary.chile@outlook.com",
	"dob": "05/06/1981",
	"mobile": 8903424531
	}

customer_3	{
	"name": "Mac Adams",
	"email": "mac_1979@gmail.com",
	"dob": "23/04/1979",
	"mobile": 0920421454
	}

customer_4	{
	"name": "Jill Smith",
	"email": "jellyjill@gmail.com",
	"dob": "14/02/1987",
	"mobile": 8795092014
	}

✓ Document Databases

What are documents?

A document is a record in a document database. A document typically stores information about one object and any of its related metadata.

Documents store data in field-value pairs. The values can be a variety of types and structures, including strings, numbers, dates, arrays, or objects. Documents can be stored in formats like JSON, BSON, and XML.

a JSON document that stores information about a user named Tom.

Collections

A collection is a group of documents. Collections typically store documents that have similar contents.

Not all documents in a collection are required to have the same fields, because document databases have flexible schemas. Note that some document databases provide schema validation, so the schema can optionally be locked down when needed.

Continuing =>

with the example above, the document with information about Tom could be stored in a collection named users. More documents could be added to the users collection in order to store information about other users.

For example, the document that stores information about *Donna* could be added to the users collection.

```
{
    "_id": 2,
    "first_name": "Donna",
    "email": "donna@example.com",
    "spouse": "Joe",
    "likes": [
       "spas",
       "shopping",
       "live tweeting"
    ],
    "businesses": [
       {
          "name": "Castle Realty",
          "status": "Thriving",
          "date_founded": {
             "$date": "2013-11-21T04:00:00Z"
          }
       }
    ]
 }
```

CRUD operations

Document databases typically have an API or query language that allows developers to execute the CRUD (create, read, update, and delete) operations.

•**Create**: Documents can be created in the database. Each document has a unique identifier.

•**Read**: Documents can be read from the database. The API or query language allows developers to query for documents using their unique identifiers or field values. Indexes can be added to the database in order to increase read performance.

•**Update**: Existing documents can be updated — either in whole or in part.

•**Delete**: Documents can be deleted from the database.

Key features of document databases:

Document model: Store data as documents (like JSON), making it easy to work with programming languages.

Flexible schema: Documents in the same collection can have different fields.

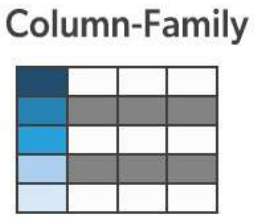
Distributed and resilient: Support horizontal scaling and replication for reliability.

Easy querying: Provide APIs or query languages to perform CRUD operations and search by fields or IDs.

Document databases vs relational databases:

- **Natural to work with:** In document databases, the way you store data matches how your program thinks about it. You don't have to split it into separate tables or join them together.
- **Uses JSON:** Data is stored in JSON, a format that's easy to read, flexible, and works in almost any programming language.
- **Flexible structure:** Each document can have different fields. You don't need to decide the structure ahead of time, and you can change it anytime without breaking anything.

✓ Wide-Column Stores (Column Family):



A **Column Family Database** (CFDB) is a type of **NoSQL database** that organizes data **by column families instead of rows**. The most well-known CFDBs include **Apache Cassandra**, **HBase**, and **ScyllaDB**.

In simple terms, think of a **column family** as a **flexible table** where each row can have a different set of columns. Unlike traditional relational databases where all rows in a table have the same structure, **column families allow each row to store only relevant data, reducing redundancy**.

Example: Online Retail Store

Imagine an **e-commerce system** that stores customer purchase history. In an **RDBMS**, we might have a table like this:

Customer ID	Name	Email	Last Purchase	Order Count
001	Alice	alice@xyz.com	2024-02-01	5
002	Bob	bob@xyz.com	NULL	0

The problem here is **Bob has no purchases**, yet we're storing unnecessary null values.

The problem here is **Bob has no purchases**, yet we're storing unnecessary null values.

How It Works in a Column Family Database

In a **column family model**, we group related data into **column families** like this:

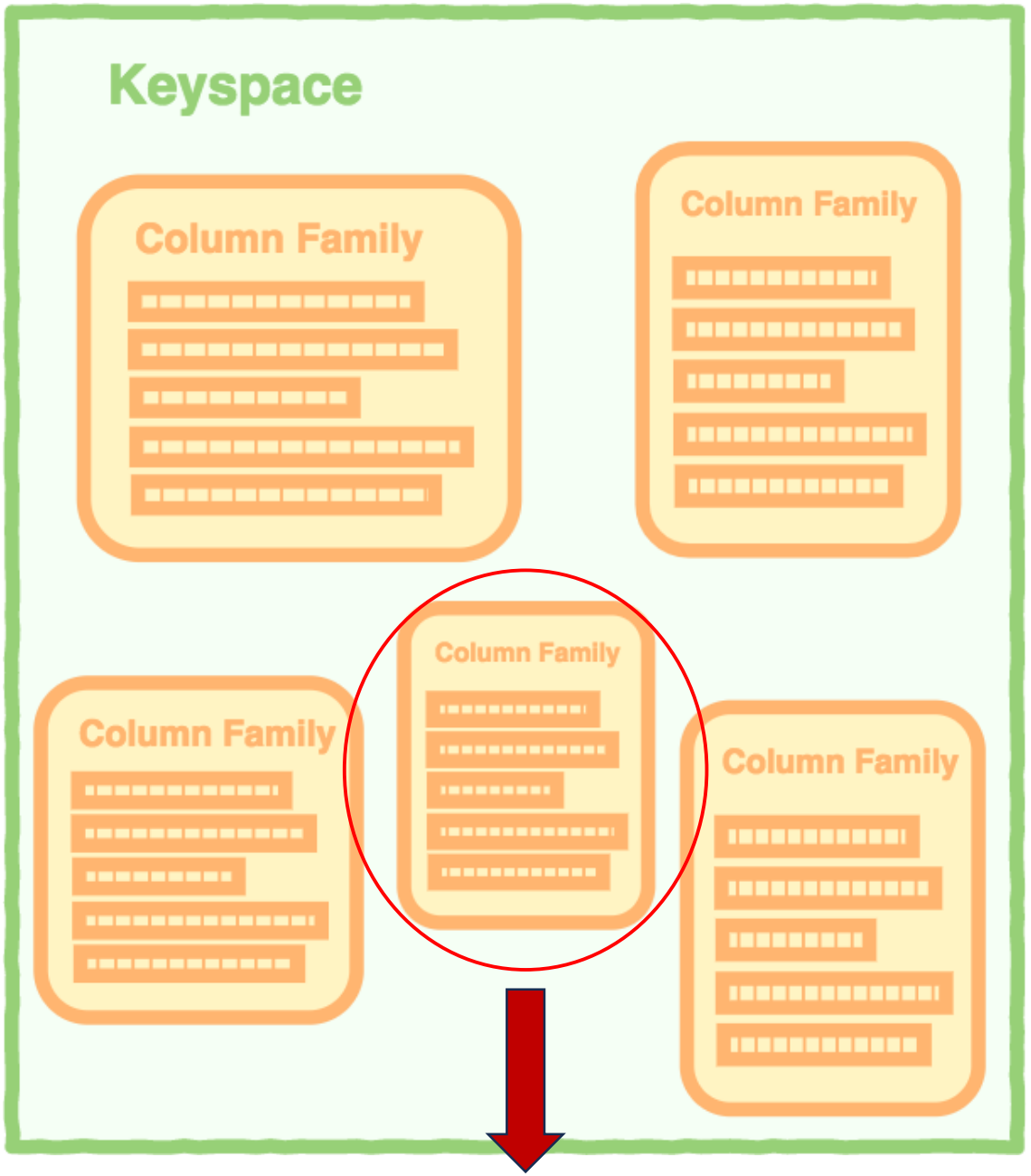
Row Key: 001 (Alice)

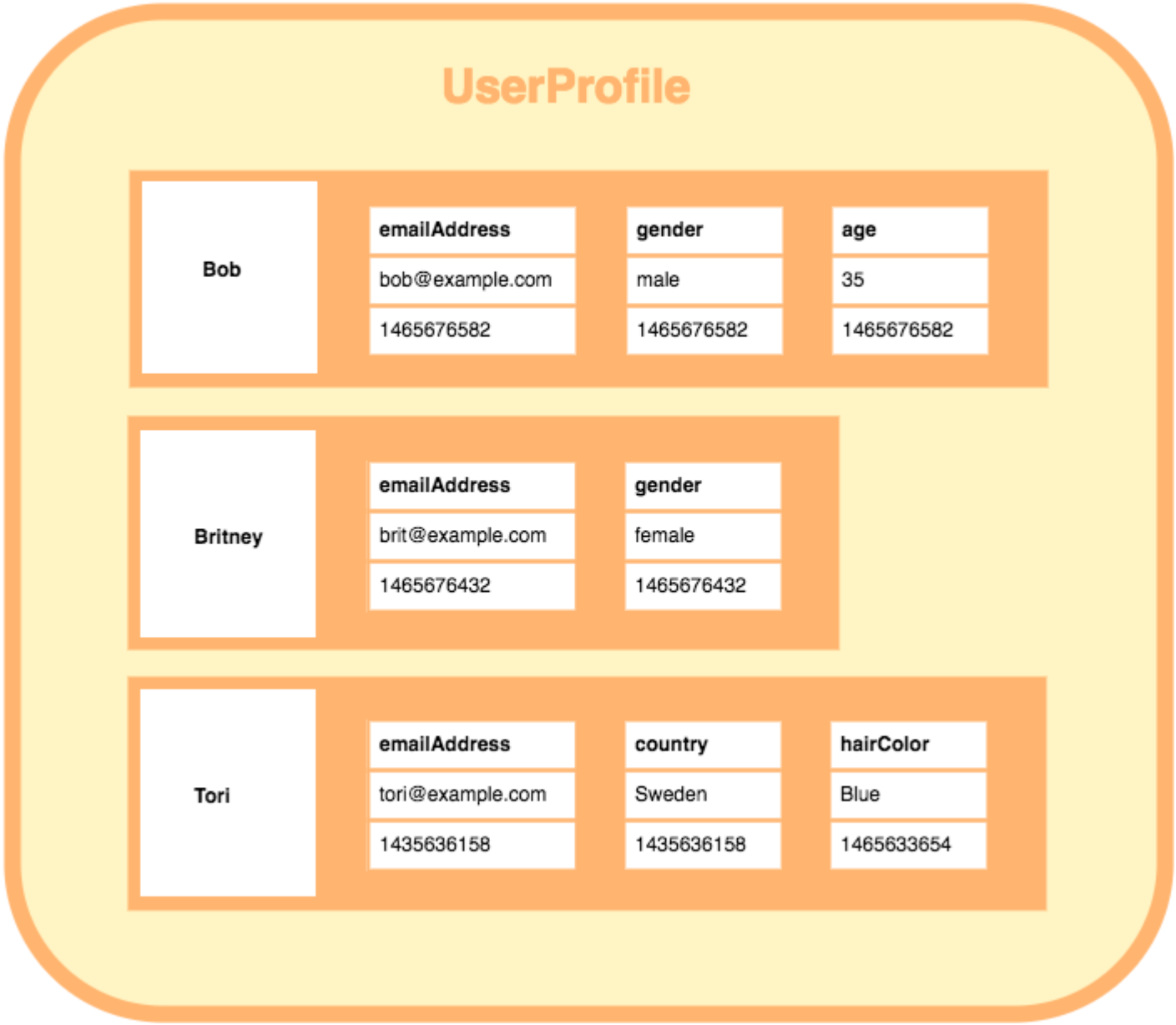
- **Personal Details** → Name: “Alice”, Email: “alice@xyz.com”
- **Purchase History** → Last Purchase: “2024–02–01”, Order Count: “5”

Row Key: 002 (Bob)

- **Personal Details** → Name: “Bob”, Email: “bob@xyz.com”

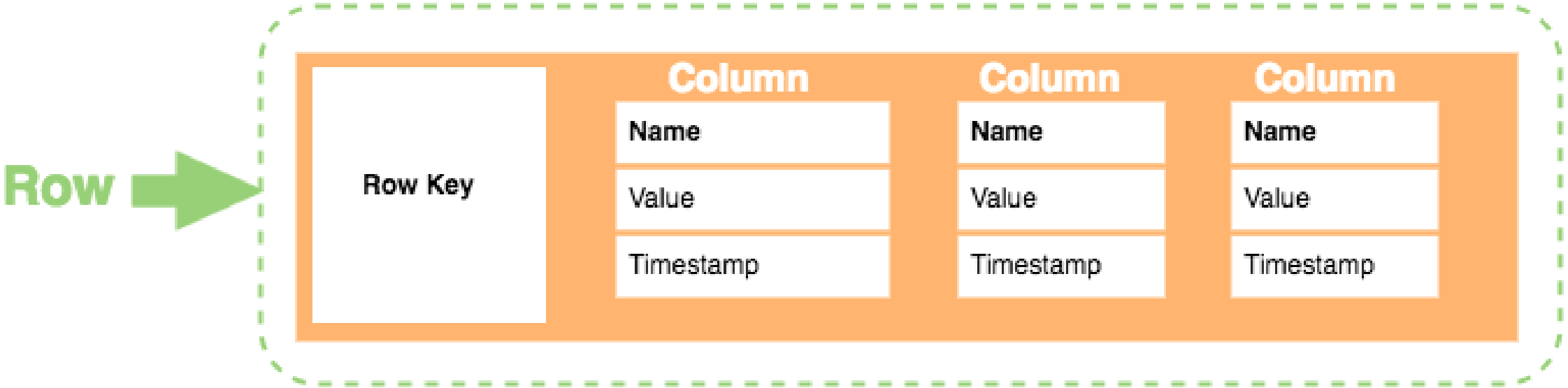
Since **Bob has no purchase history**, that column family simply doesn’t exist for him, making storage more **efficient**.

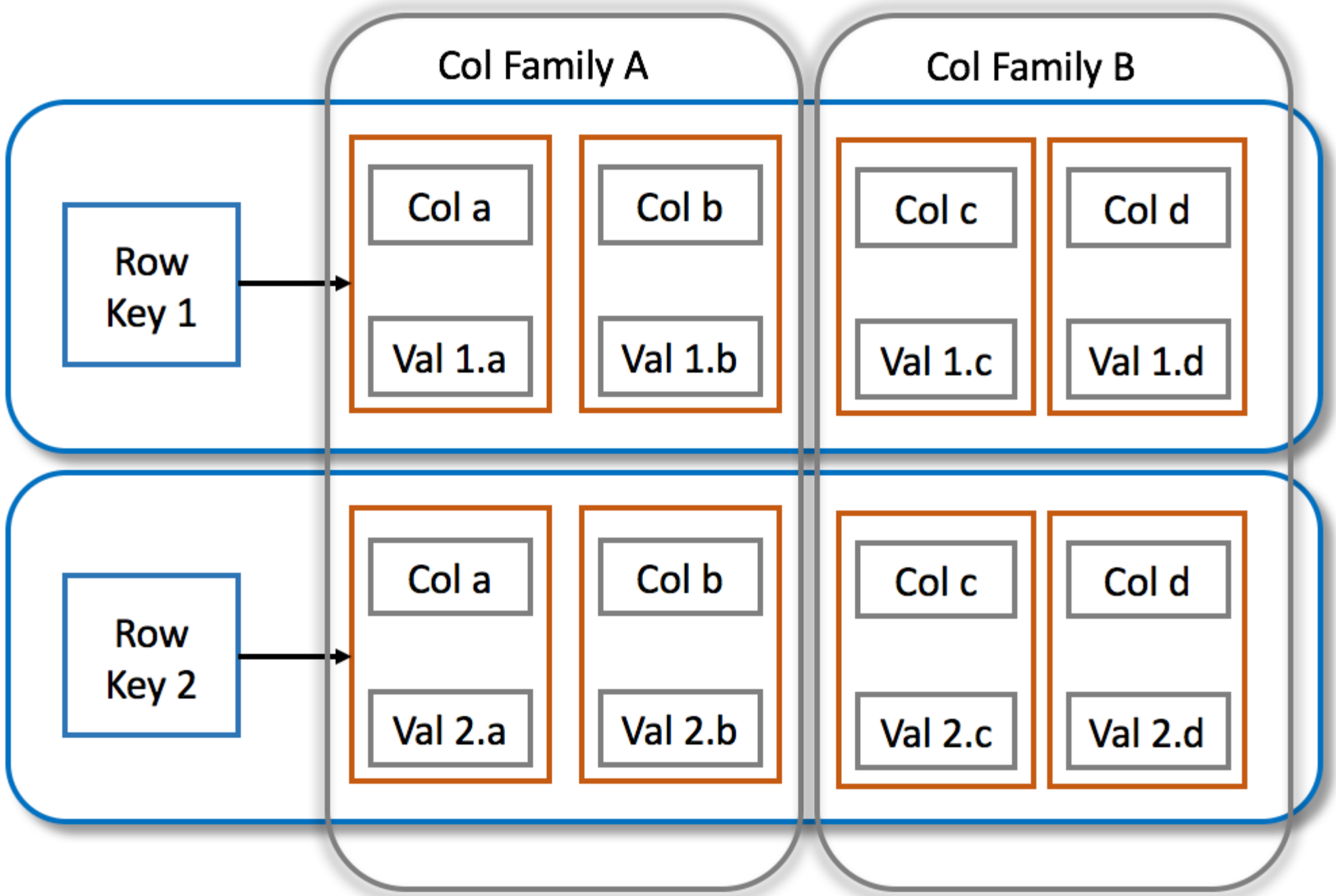

Keyspace
Column Family
Column Family
Column Family
Column Family
Column Family



Here's a closer look at a column family:

Here’s how each row is constructed:




Col Family A
Col Family B
Row Key 1
Col a
Col b
Col c
Col d
Val 1.a
Val 1.b
Val 1.c
Val 1.d
Row Key 2
Col a
Col b
Col c
Col d
Val 2.a
Val 2.b
Val 2.c
Val 2.d

Example: Imagine you're building a "User Preferences / Purchases" system, where each user may have different attributes (columns) and purchase history entries. You want flexibility: some users have "favorite_color", others have "recent_purchase_date", etc.

Row Key (UserID)	name	age	favorite_color	last_login	Purchase_2025-10-01	Purchase_2025-10-03	….
user_001	Alice	30	Blue	2025-10-10 12:00	"Book"	"Shoes"	….
user_002	Bob	25	—	2025-10-09 08:45	"Laptop"	**(no other purchases)**	…..
user_003	Carol	—	Green	2025-10-10 16:20	"Watch"	"Bag"	….

Schema (conceptual)
Let's define a **column family** called UserData (in a wide-column store like Cassandra):
•**Row key**: UserID (this is your primary key / partition key)
•**Columns in that row**: dynamic, e.g.:
•And column family called PurchaseData

UserData

User_001	Name	Age	City
	Alice	30	Paris

User_002	Name	Last Login
	Bob	2025-10-16

User_003	Favourite colour	City
	Green	Berlin

Purchases

User_001	Purchase-01	Purchase-07
	Book	Shoes

User_002	Purchase-05
	Lap Top

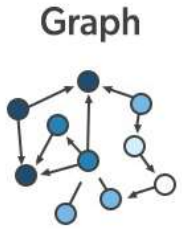
User_003	Purchase-02
	Watch

Query Example:

- First key = user_001 (row identifier)
- Second key = e.g. UserDetails:city or Purchases:purchase_07
- Value = the actual field (“Paris”, “Shoes”, etc.)

get(UserID = user_001, columns = [“city”, “purchase_07”])

✓ Graph Databases

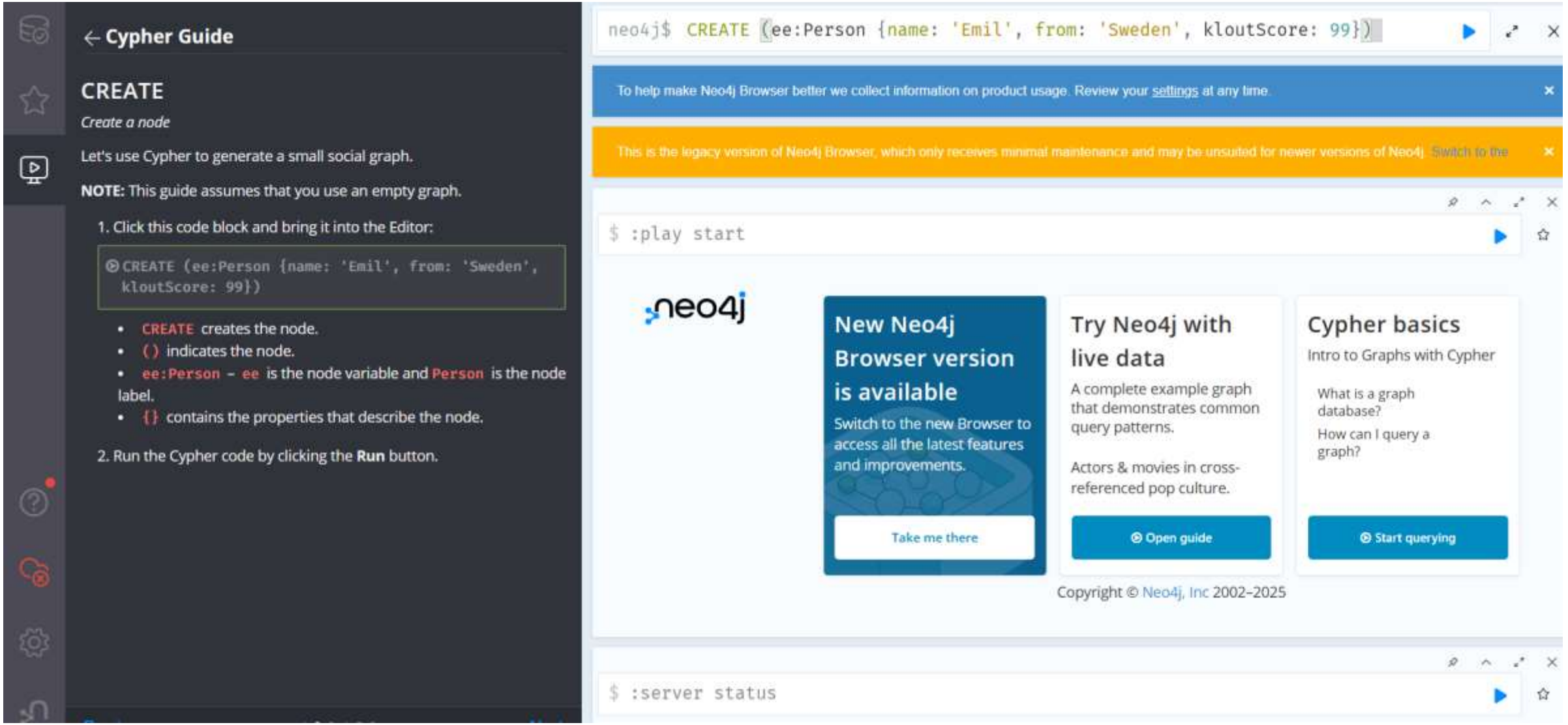


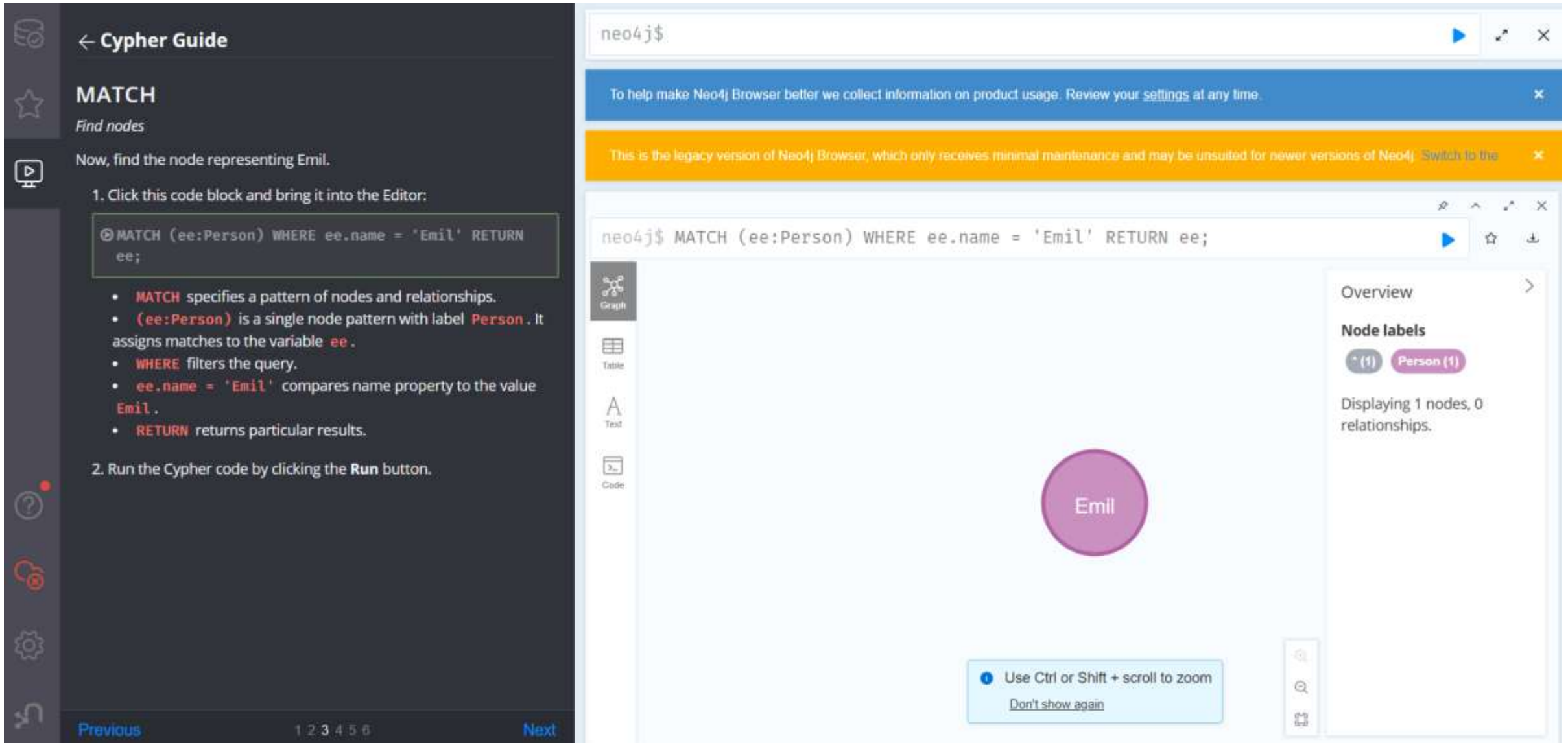
A NoSQL data store that persists data in a network of nodes and edges.

- Nodes represent entities (**Nodes typically store information about people, places, and things (like nouns)**)
- Edges represent relationships between entities(**edges store information about the relationships between the nodes**)

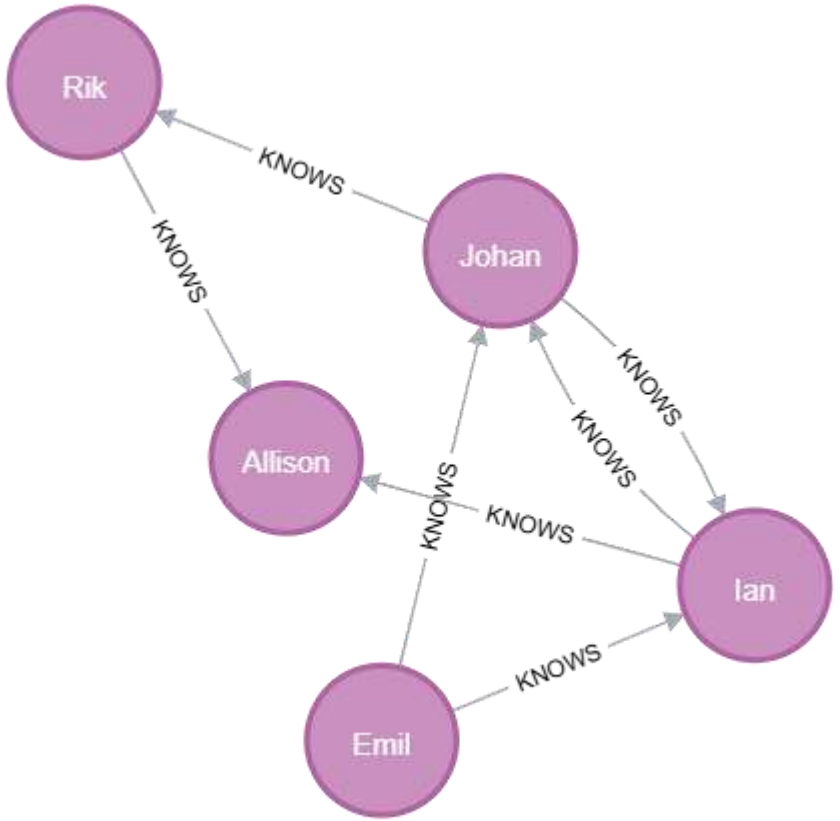
They work well for highly connected data, where the relationships or patterns may not be very obvious initially

Use cases: social networks

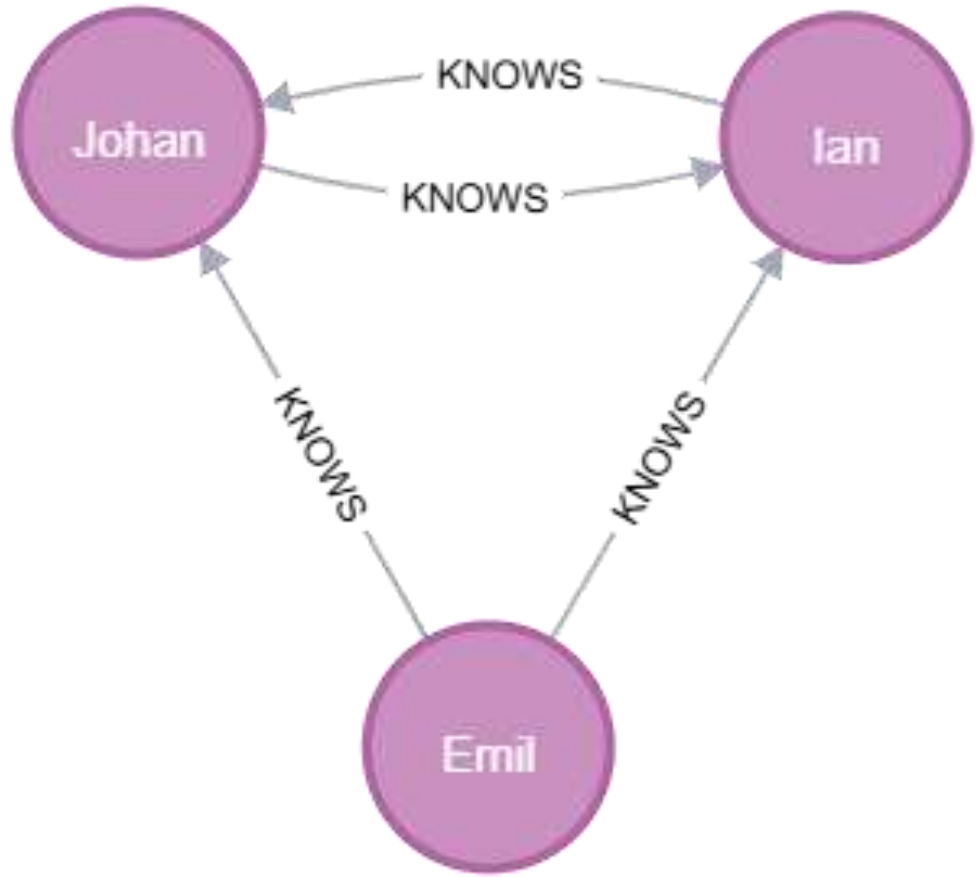

← Cypher Guide
CREATE
Create a node
Let's use Cypher to generate a small social graph.
NOTE: This guide assumes that you use an empty graph.
1. Click this code block and bring it into the Editor:
CREATE (ee:Person {name: 'Emil', from: 'Sweden', kloutScore: 99})
CREATE creates the node.
() indicates the node.
ee:Person – ee is the node variable and Person is the node label.
{} contains the properties that describe the node.
2. Run the Cypher code by clicking the Run button.
neo4j$ CREATE (ee:Person {name: 'Emil', from: 'Sweden', kloutScore: 99})
To help make Neo4j Browser better we collect information on product usage. Review your settings at any time.
This is the legacy version of Neo4j Browser, which only receives minimal maintenance and may be unsuited for newer versions of Neo4j. Switch to the
$:play start
neo4j
New Neo4j Browser version is available
Switch to the new Browser to access all the latest features and improvements.
Take me there
Try Neo4j with live data
A complete example graph that demonstrates common query patterns.
Actors & movies in cross-referenced pop culture.
Open guide
Cypher basics
Intro to Graphs with Cypher
What is a graph database?
How can I query a graph?
Start querying
Copyright © Neo4j, Inc 2002–2025
$:server status


← Cypher Guide
MATCH
Find nodes
Now, find the node representing Emil.
1. Click this code block and bring it into the Editor:
MATCH (ee:Person) WHERE ee.name = 'Emil' RETURN ee;
MATCH specifies a pattern of nodes and relationships.
(ee:Person) is a single node pattern with label Person. It assigns matches to the variable ee.
WHERE filters the query.
ee.name = 'Emil' compares name property to the value Emil.
RETURN returns particular results.
2. Run the Cypher code by clicking the Run button.
Previous
1 2 3 4 5 6
Next
neo4j$
To help make Neo4j Browser better we collect information on product usage. Review your settings at any time.
This is the legacy version of Neo4j Browser, which only receives minimal maintenance and may be unsuited for newer versions of Neo4j. Switch to the
neo4j$ MATCH (ee:Person) WHERE ee.name = 'Emil' RETURN ee;
Graph
Table
Text
Code
Emil
Use Ctrl or Shift + scroll to zoom
Don't show again
Overview
Node labels
* (1)
Person (1)
Displaying 1 nodes, 0 relationships.

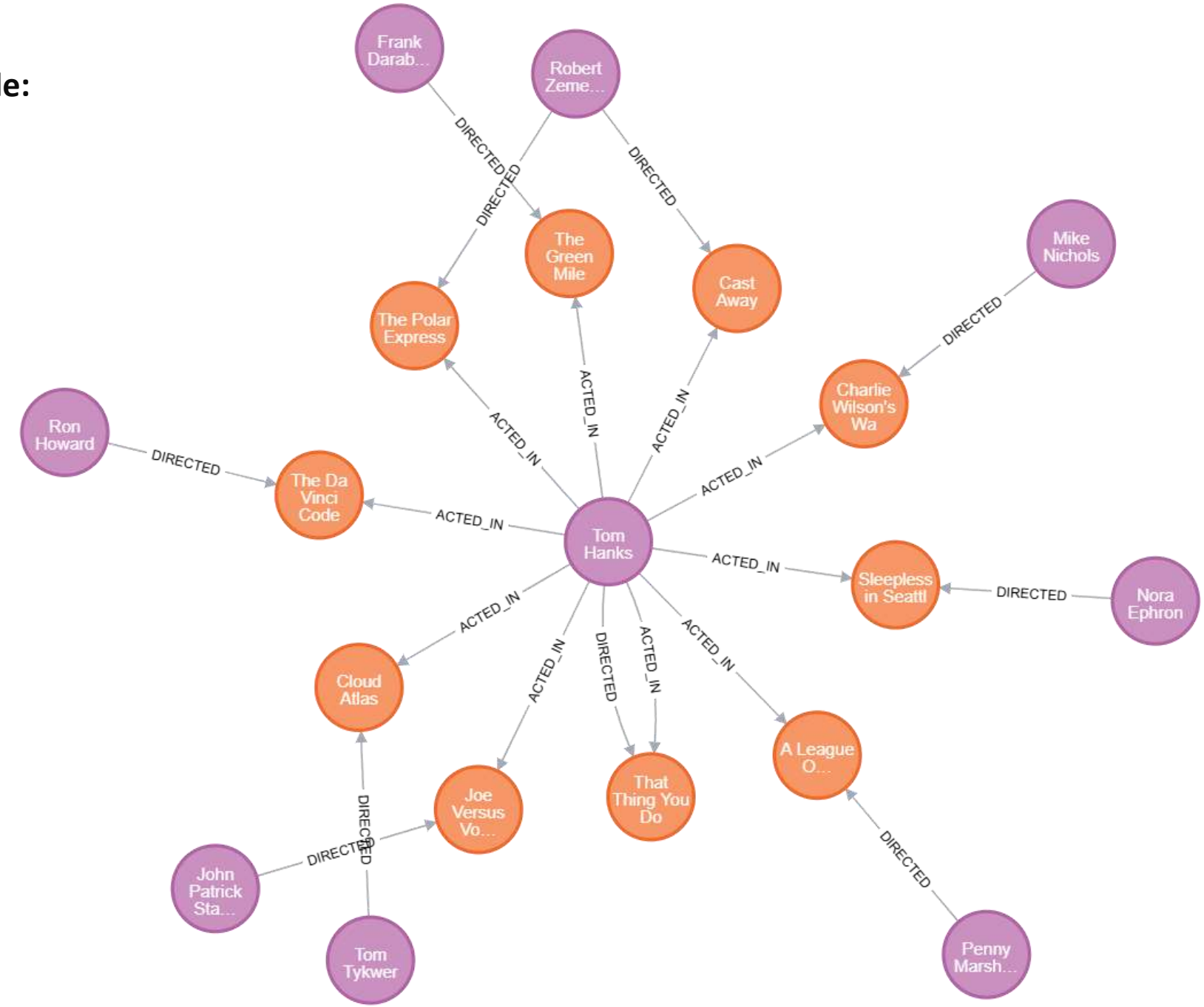
```
MATCH (ee:Person) WHERE ee.name = 'Emil'

CREATE (js:Person { name: 'Johan', from: 'Sweden', learn: 'surfing' }),

(ir:Person { name: 'Ian', from: 'England', title: 'author' }),

(rvb:Person { name: 'Rik', from: 'Belgium', pet: 'Orval' }),

(ally:Person { name: 'Allison', from: 'California', hobby: 'surfing' }),

(ee)-[:KNOWS {since: 2001}]->(js),(ee)-[:KNOWS {rating: 5}]->(ir),

(js)-[:KNOWS]->(ir),(js)-[:KNOWS]->(rvb),

(ir)-[:KNOWS]->(js),(ir)-[:KNOWS]->(ally),

(rvb)-[:KNOWS]->(ally)
```



```
MATCH (ee:Person)-[:KNOWS]-(friends)
WHERE ee.name = 'Emil' RETURN ee, friends
```



Movies example:



Key Features of NoSQL

NoSQL databases share a few common high-level qualities

- **Flexible schema**: NoSQL often allows schema-on-read. You do not need to define a rigid table structure before storing data. This makes it easier to store semi-structured and unstructured data.
- **Horizontal scaling**: They can distribute data across many machines (shards, partitions), rather than relying on scaling up a single server. This supports growth in both data size and access traffic.
- **Optimized for specific workloads**: Depending on the data model (key-value, document, wide-column, graph), NoSQL systems are tuned for different access patterns. For example, wide-column stores are good for analytical queries over many similar columns; key-value stores are good for simple, repeated lookups.
- **Relaxed consistency in many cases**: Many NoSQL systems use models like *eventual consistency* rather than strict ACID properties, which allows more flexibility, availability, and partition tolerance.

Advantages

- **Flexible schema:** NoSQL databases allow changes to data structure without redesigning the whole system, supporting fast development.
- **Scalable and reliable:** Data is distributed and often replicated across multiple nodes, ensuring availability even if some nodes fail.
- **Efficient for large and diverse data:** Performs well with massive or constantly changing datasets, such as sensor data, logs, or social media content.

Drawbacks

- **Limited tool support:** Development tools and management systems are not as mature as those for relational databases.
- **Lack of standard query language:** Each NoSQL system has its own query methods, making learning and migration more difficult.
- **Reduced consistency and complex operations:** Features like strict consistency, joins, and multi-record transactions are less efficient or sometimes unsupported.

When to Use SQL vs NoSQL

Relational SQL databases are still the right choice when you need strict consistency, well-defined schema, complex transactional workflows, and relational queries. NoSQL shines where you need flexibility, scale, and speed with evolving data types.

Some example use cases: real-time web or mobile apps, IoT data ingestion, personalization / recommendation engines, catalogs, fraud detection.