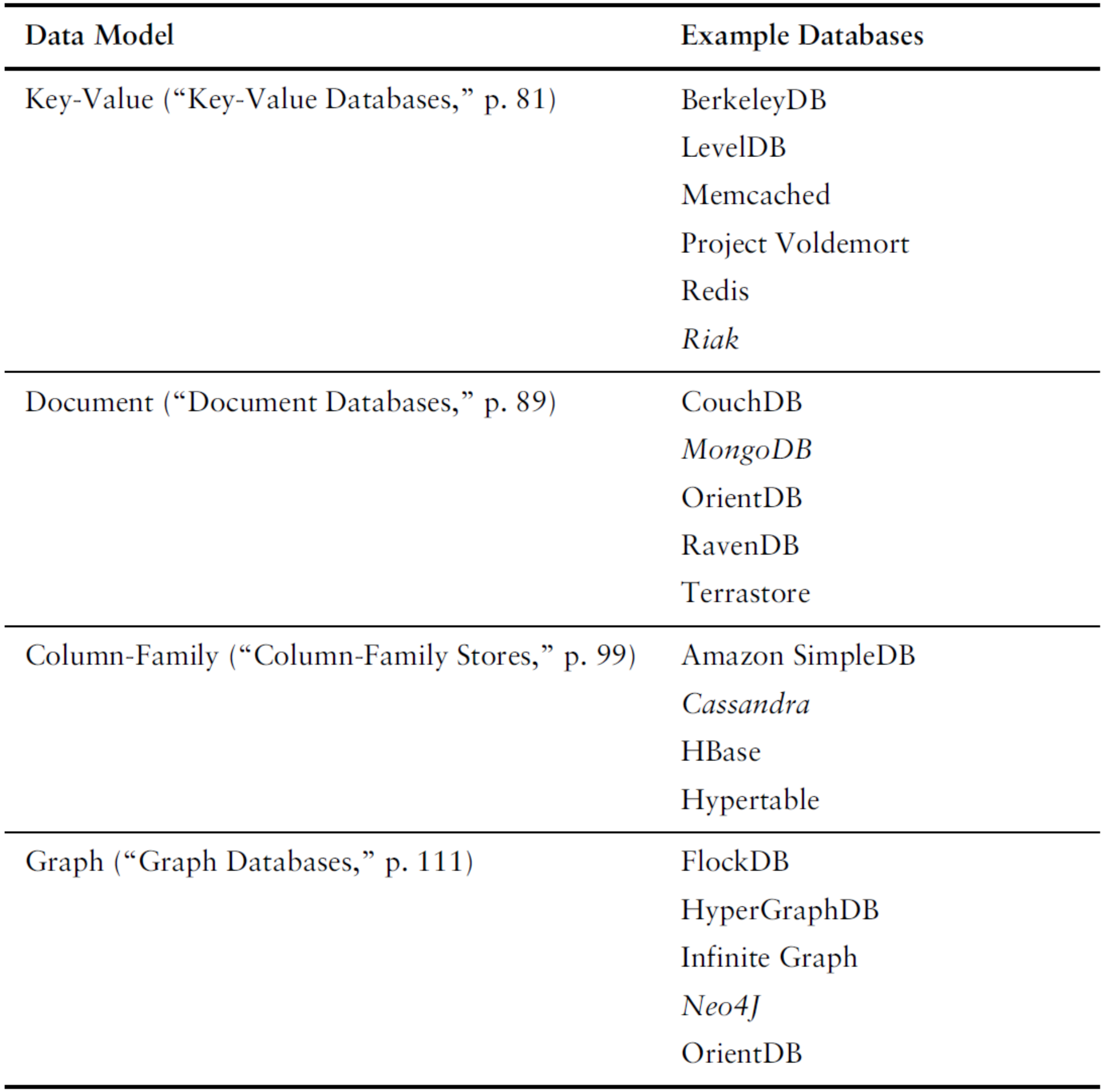
NoSQL Databases

→ NoSQL means **N**ot **O**nly **SQL**

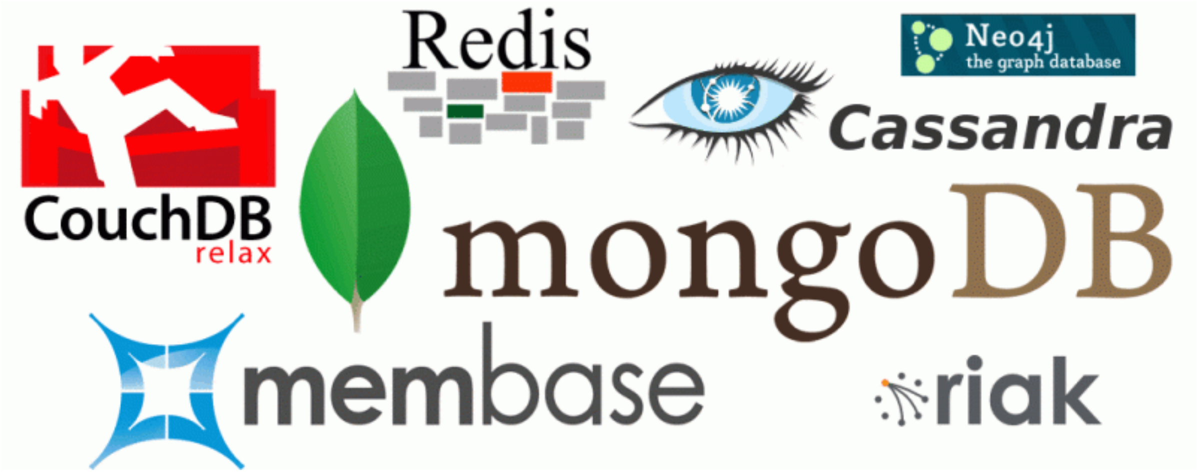
→ NoSQL covers multiple types of databases



Data Model	Example Databases
Key-Value (“Key-Value Databases,” p. 81)	BerkeleyDB LevelDB Memcached Project Voldemort Redis *Riak*
Document (“Document Databases,” p. 89)	CouchDB *MongoDB* OrientDB RavenDB Terrastore
Column-Family (“Column-Family Stores,” p. 99)	Amazon SimpleDB *Cassandra* HBase Hypertable
Graph (“Graph Databases,” p. 111)	FlockDB HyperGraphDB Infinite Graph *Neo4J* OrientDB

Source: NoSQL Distilled, by P. Sadalage and M. Fowler

Why NoSQL?



→ Fits well to many data types and application areas

→ Dynamic Schema! (Bad and Good)

→ Easy to scale

  → Data Size increases (Big Data)

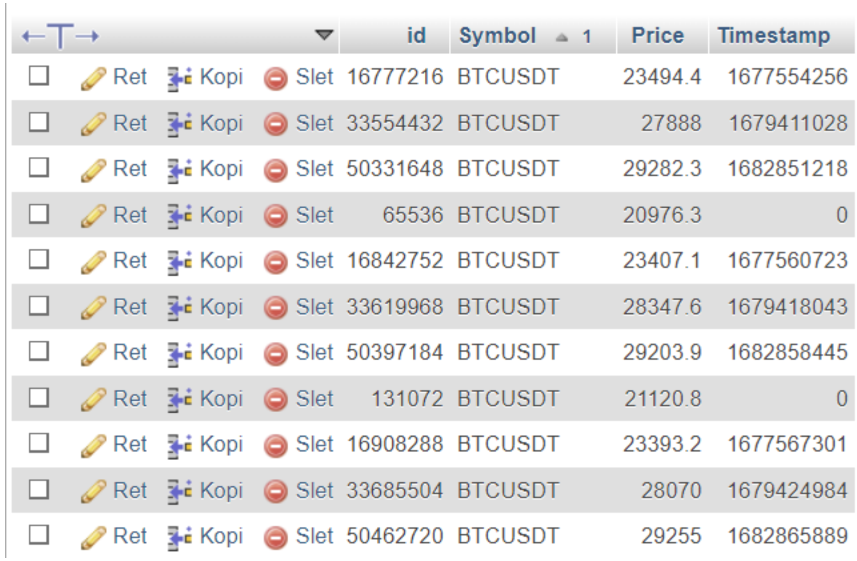
→ Often easy to replicate

→ High Performance

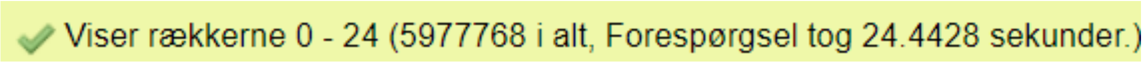
→ Less powerful query languages

→ Versioning

A comment on database sizes (Relational DBs)



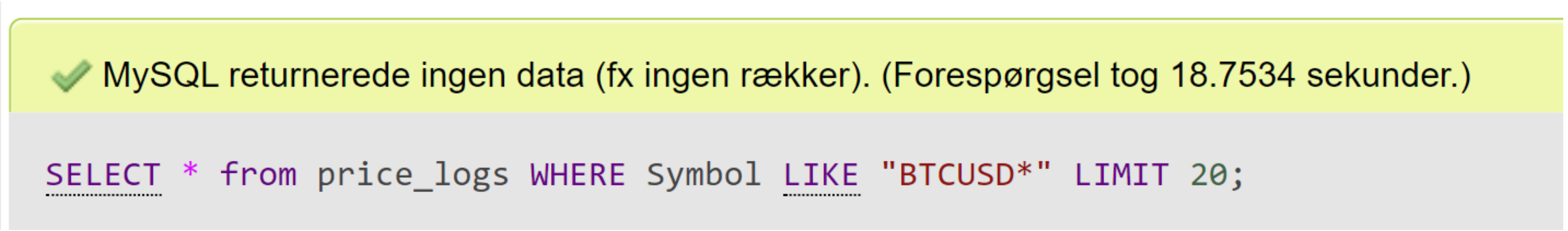
←T→			▼	id	Symbol ▲ 1	Price	Timestamp
☐	Ret	Kopi	Slet	16777216	BTCUSDT	23494.4	1677554256
☐	Ret	Kopi	Slet	33554432	BTCUSDT	27888	1679411028
☐	Ret	Kopi	Slet	50331648	BTCUSDT	29282.3	1682851218
☐	Ret	Kopi	Slet	65536	BTCUSDT	20976.3	0
☐	Ret	Kopi	Slet	16842752	BTCUSDT	23407.1	1677560723
☐	Ret	Kopi	Slet	33619968	BTCUSDT	28347.6	1679418043
☐	Ret	Kopi	Slet	50397184	BTCUSDT	29203.9	1682858445
☐	Ret	Kopi	Slet	131072	BTCUSDT	21120.8	0
☐	Ret	Kopi	Slet	16908288	BTCUSDT	23393.2	1677567301
☐	Ret	Kopi	Slet	33685504	BTCUSDT	28070	1679424984
☐	Ret	Kopi	Slet	50462720	BTCUSDT	29255	1682865889



✓ Viser rækkerne 0 - 24 (5977768 i alt, Forespørgsel tog 24.4428 sekunder.)

About 6.000.000 rows

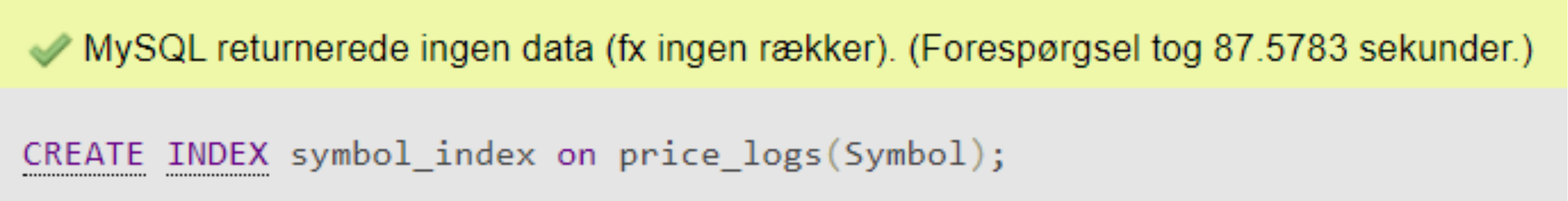
A comment on database sizes (Relational DBs)



✓ MySQL returnerede ingen data (fx ingen rækker). (Forespørgsel tog 18.7534 sekunder.)

```
SELECT * from price_logs WHERE Symbol LIKE "BTCUSD*" LIMIT 20;
```

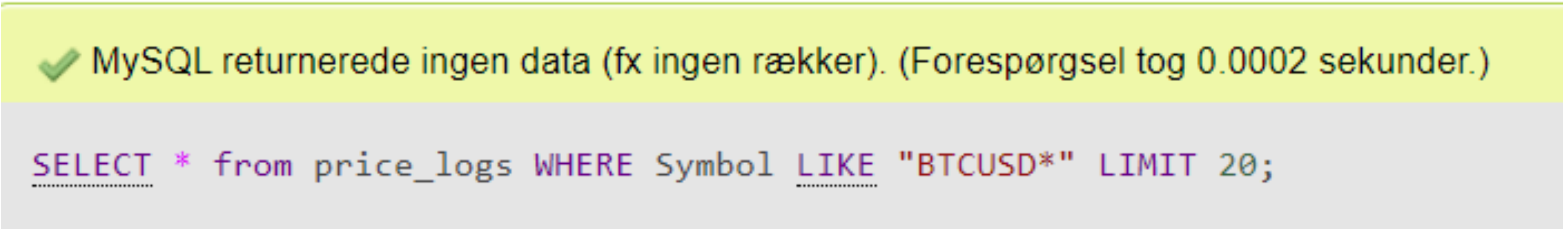
Takes 18,7 seconds to run



✓ MySQL returnerede ingen data (fx ingen rækker). (Forespørgsel tog 87.5783 sekunder.)

```
CREATE INDEX symbol_index on price_logs(Symbol);
```

Create an index to optimize the ‘like’ clause, which takes 87,5783 seconds to create in memory.

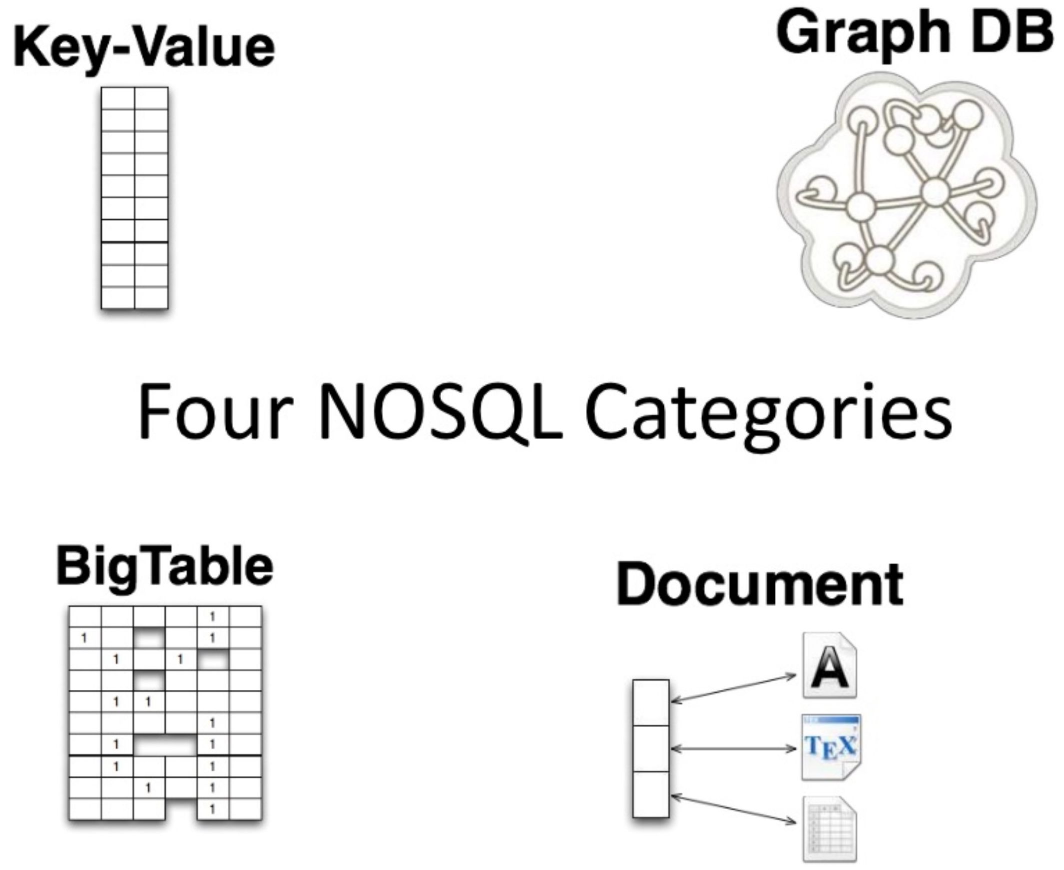


✓ MySQL returnerede ingen data (fx ingen rækker). (Forespørgsel tog 0.0002 sekunder.)

```
SELECT * from price_logs WHERE Symbol LIKE "BTCUSD*" LIMIT 20;
```

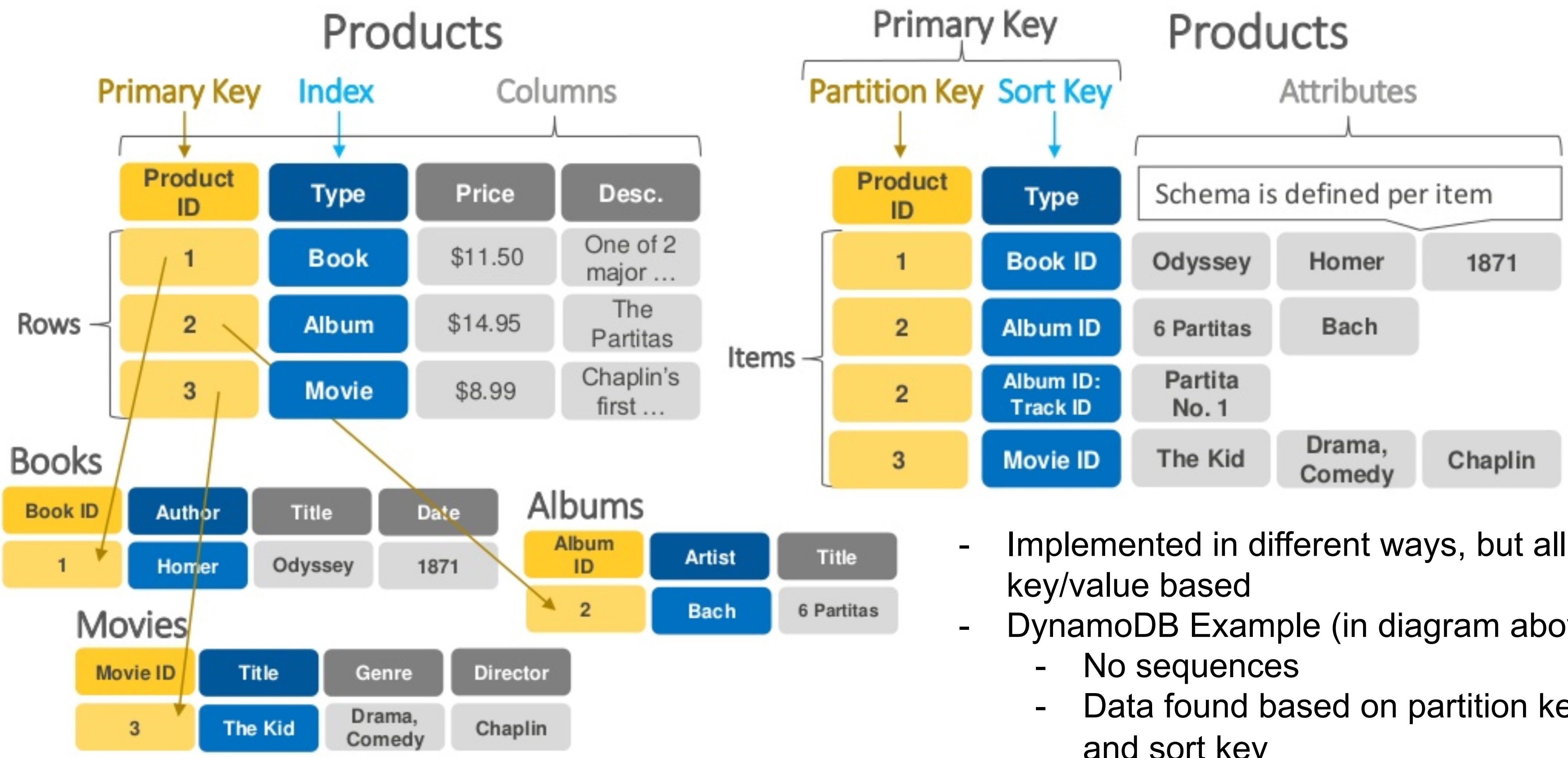
After optimization, the query takes 0,0002 seconds to run (or 0.2 milliseconds)

What types exist?



- → Document-based NoSQL systems
- → NoSQL key-value stores
- → Column based or wide column-based NoSQL systems
- → Graph based NoSQL systems
- → Hybrid NoSQL systems
  - → Object databases
  - → XML databases

Relational vs. Key Value



Products

Product ID	Type	Price	Desc.
1	Book	$11.50	One of 2 major …
2	Album	$14.95	The Partitas
3	Movie	$8.99	Chaplin's first …

Books

Book ID	Author	Title	Date
1	Homer	Odyssey	1871

Albums

Album ID	Artist	Title
2	Bach	6 Partitas

Movies

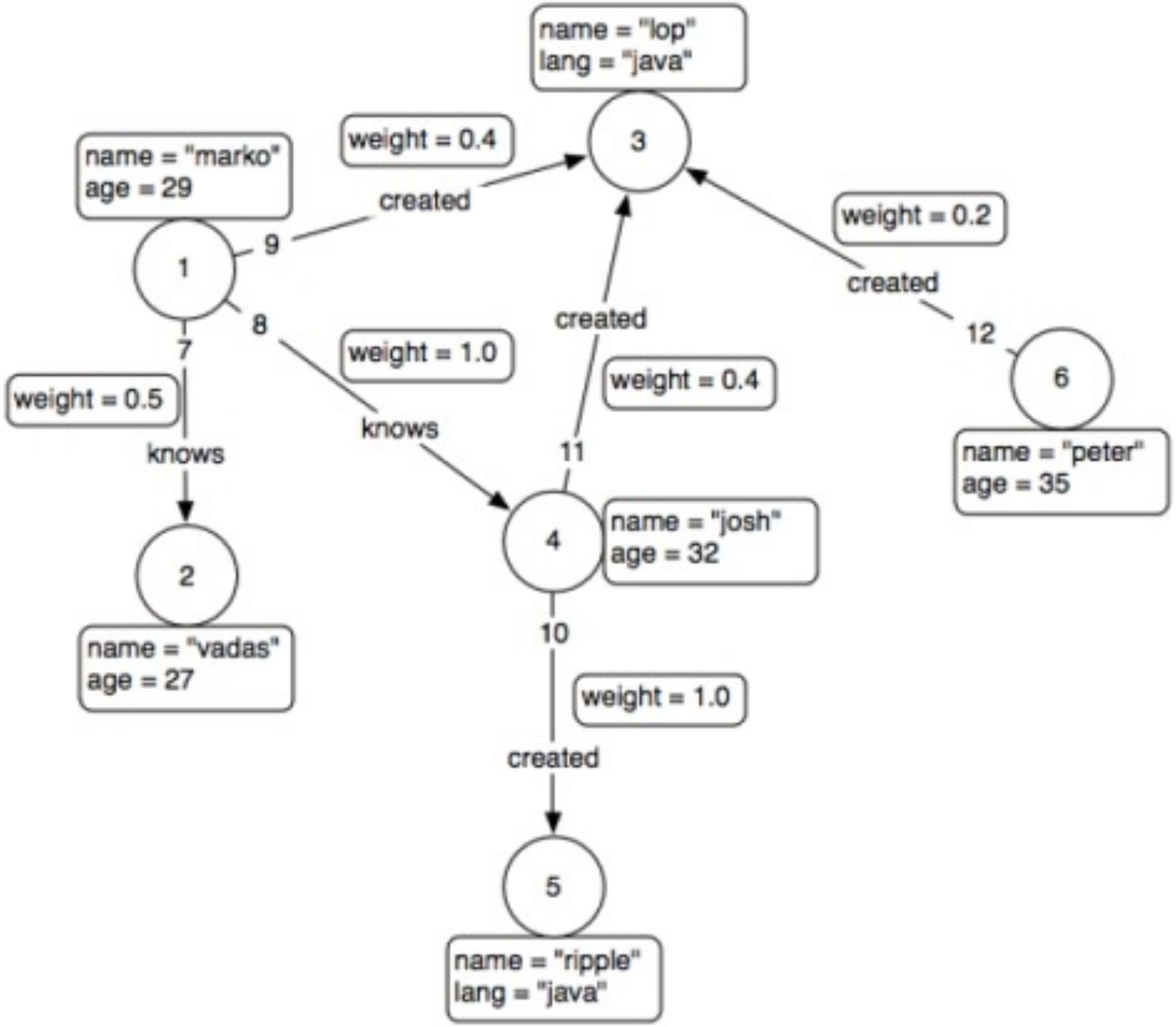
Movie ID	Title	Genre	Director
3	The Kid	Drama, Comedy	Chaplin

Products

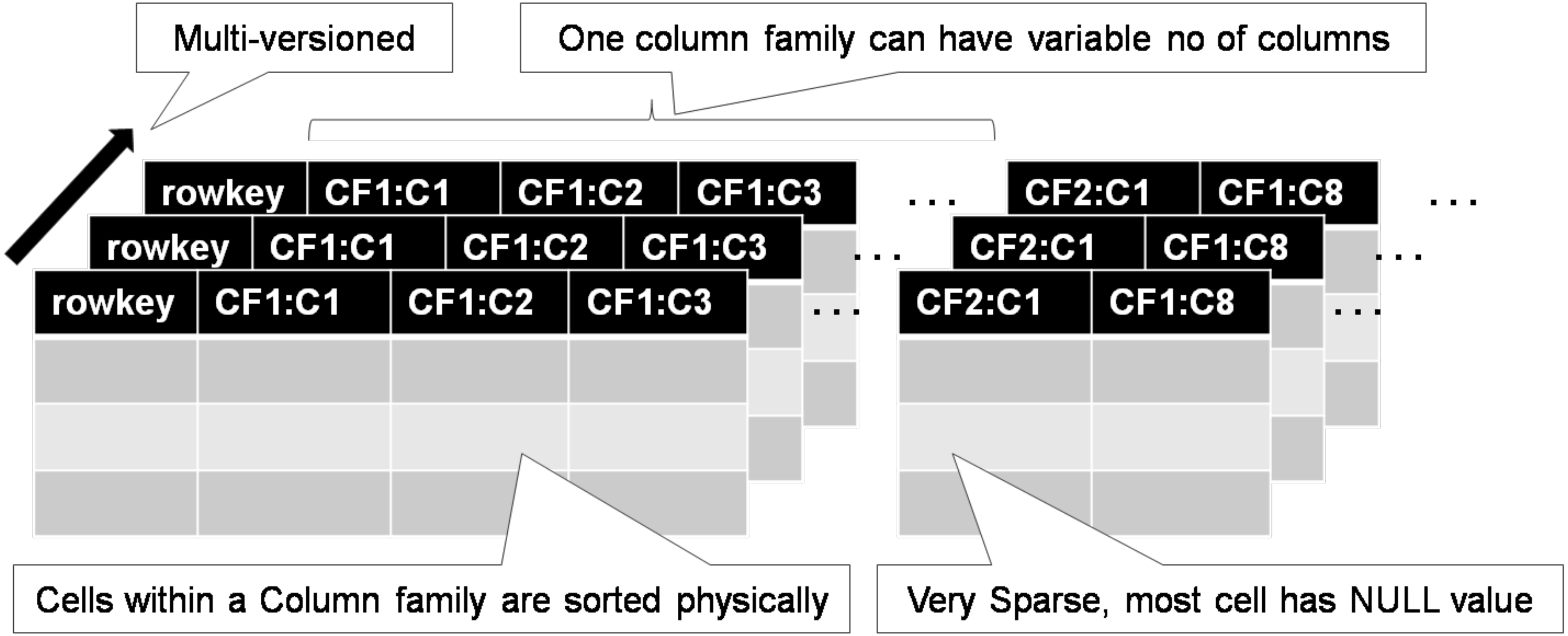
Product ID	Type	Schema is defined per item		
1	Book ID	Odyssey	Homer	1871
2	Album ID	6 Partitas	Bach	
2	Album ID: Track ID	Partita No. 1		
3	Movie ID	The Kid	Drama, Comedy	Chaplin

- Implemented in different ways, but all key/value based
- DynamoDB Example (in diagram above)
  - No sequences
  - Data found based on partition key and sort key

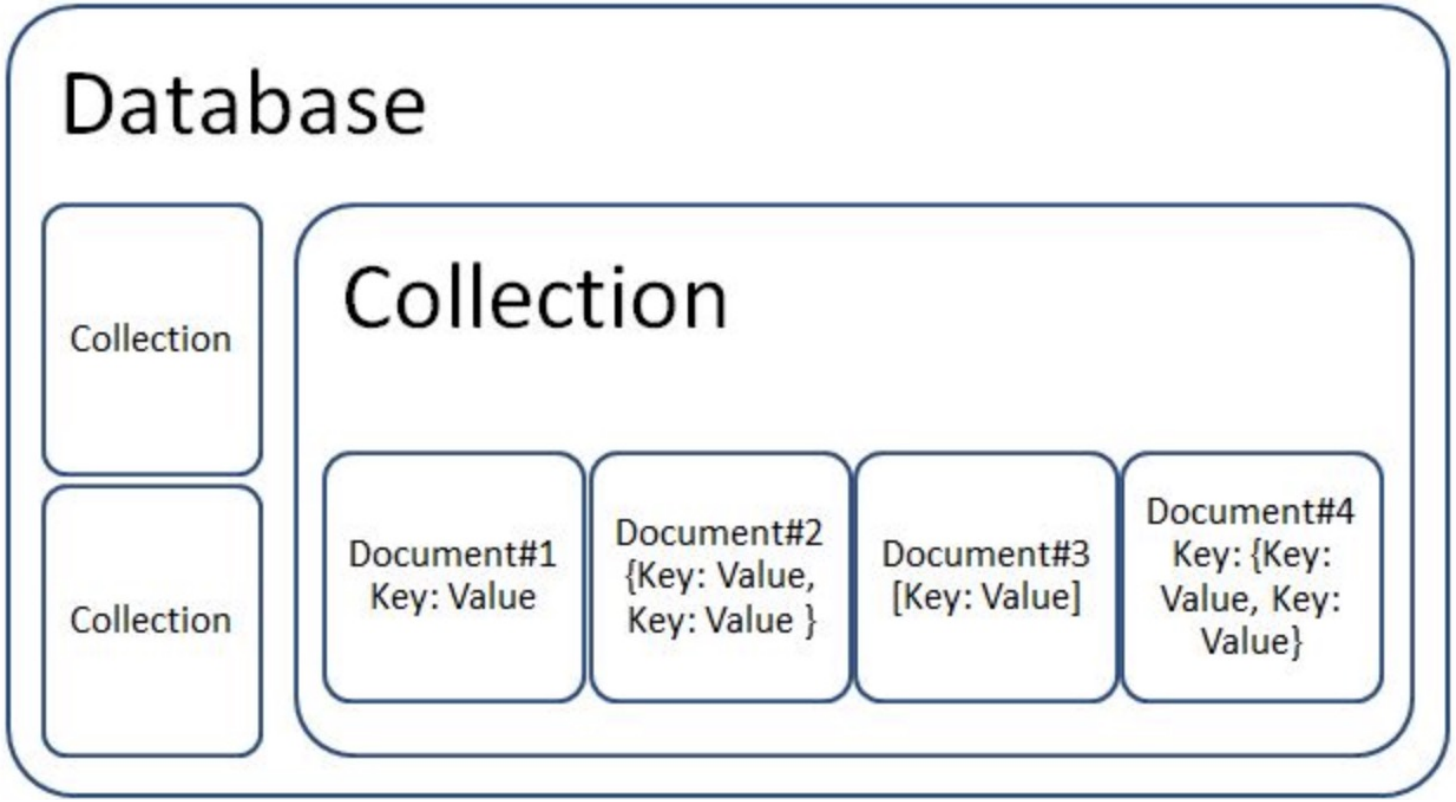
Graph Based


name = "lop"
lang = "java"
3
weight = 0.4
name = "marko"
age = 29
created
9
1
weight = 0.2
created
12
6
8
7
weight = 1.0
created
weight = 0.4
weight = 0.5
knows
knows
11
name = "peter"
age = 35
name = "josh"
age = 32
4
2
name = "vadas"
age = 27
10
weight = 1.0
created
5
name = "ripple"
lang = "java"

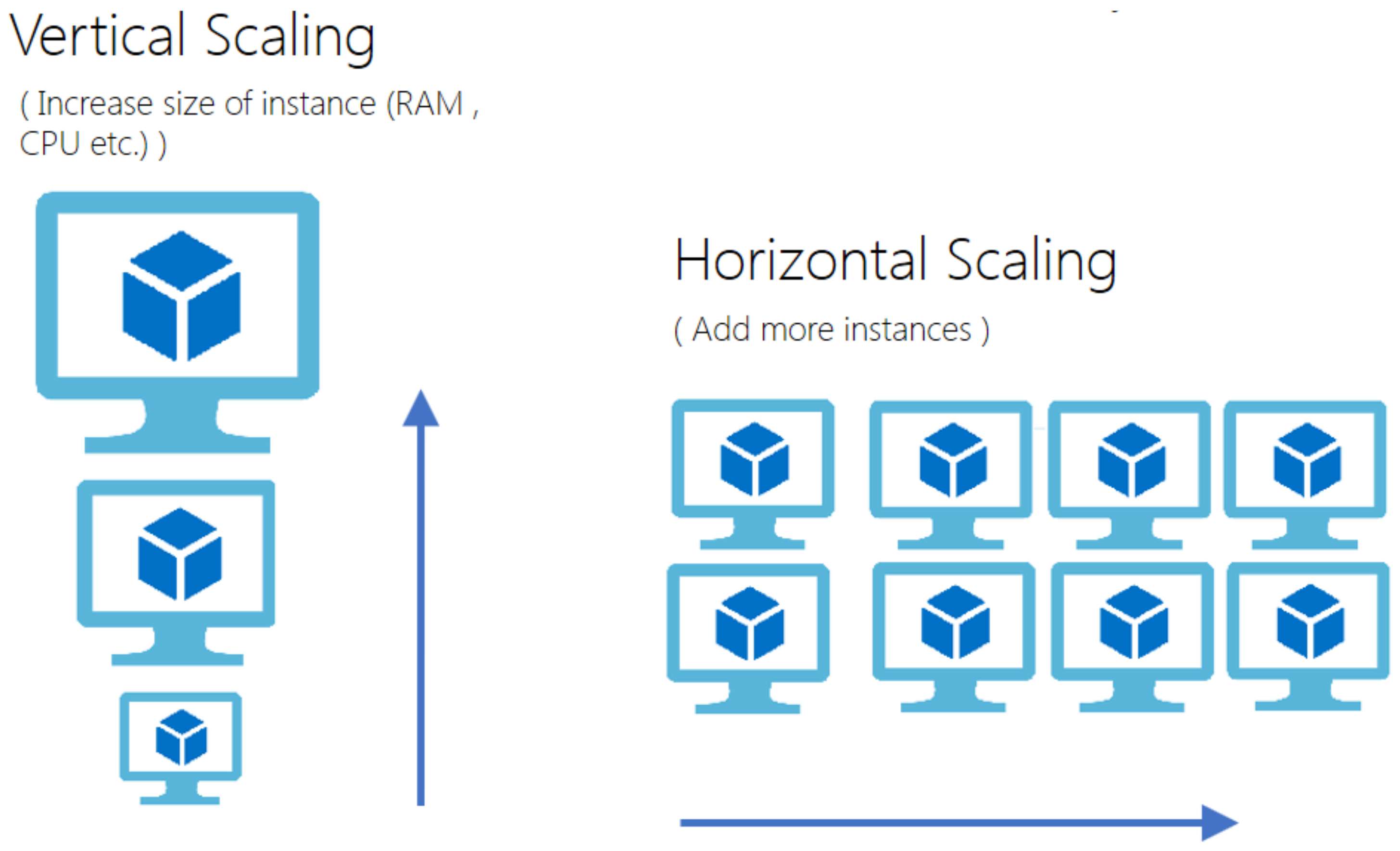
Big Table


Multi-versioned
One column family can have variable no of columns
rowkey
CF1:C1
CF1:C2
CF1:C3
...
CF2:C1
CF1:C8
...
Cells within a Column family are sorted physically
Very Sparse, most cell has NULL value

Document based



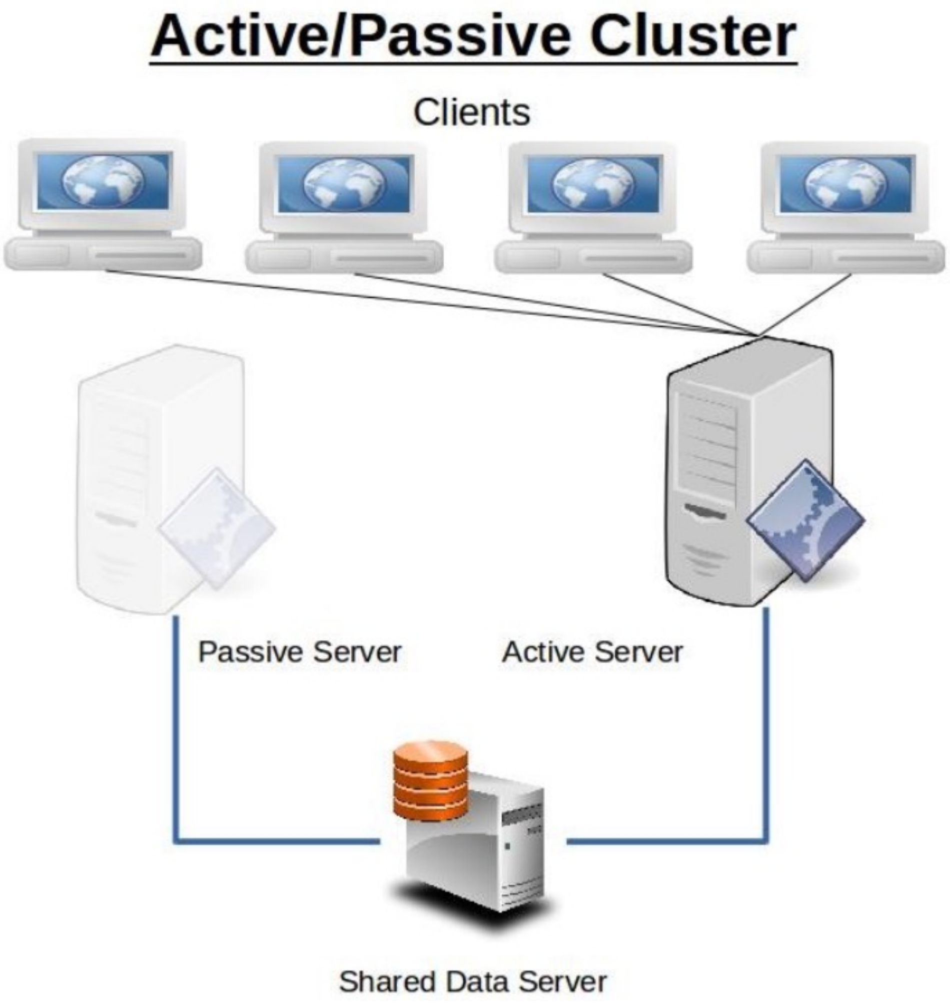
Horizontal vs Vertical Scaling

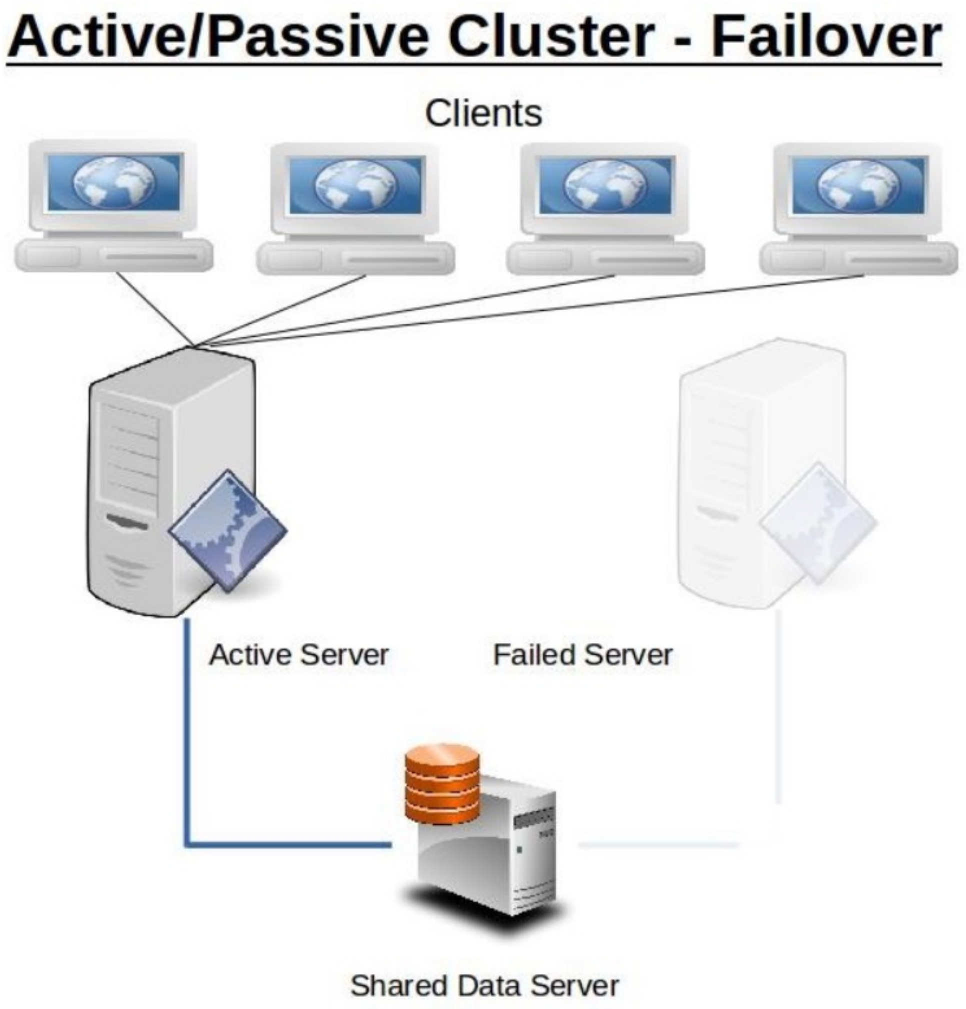


- → Vertical
  - → Increasingly Expensive to get larger hardware
  - → Requires no code changes
- → Horizontal
  - → Less expensive hardware
  - → Allows for large-scale systems
  - → Need to change software for a distributed architecture
    - → Leads to more complex code
  - → Need for Load Balancing

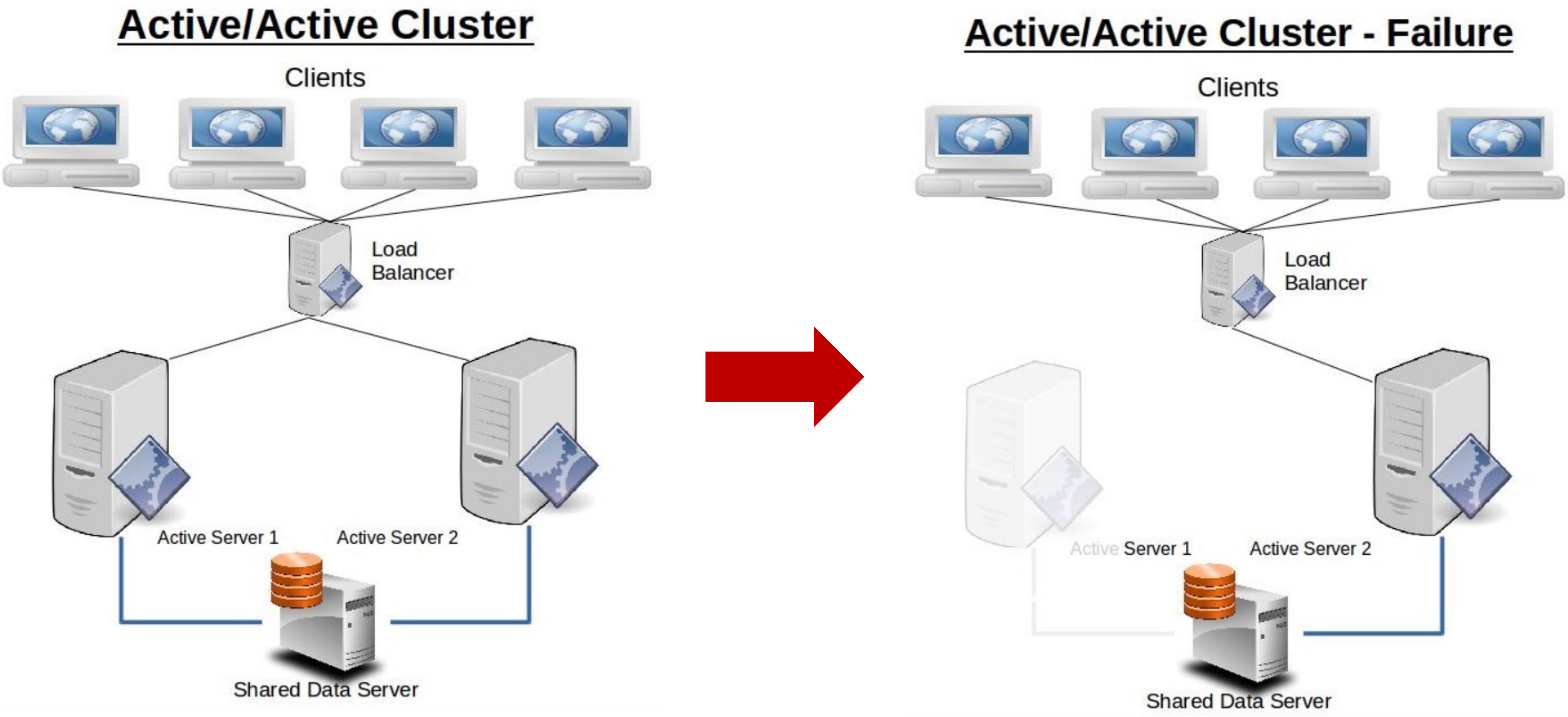
→ Do you know examples of these?

High Availability 1/3

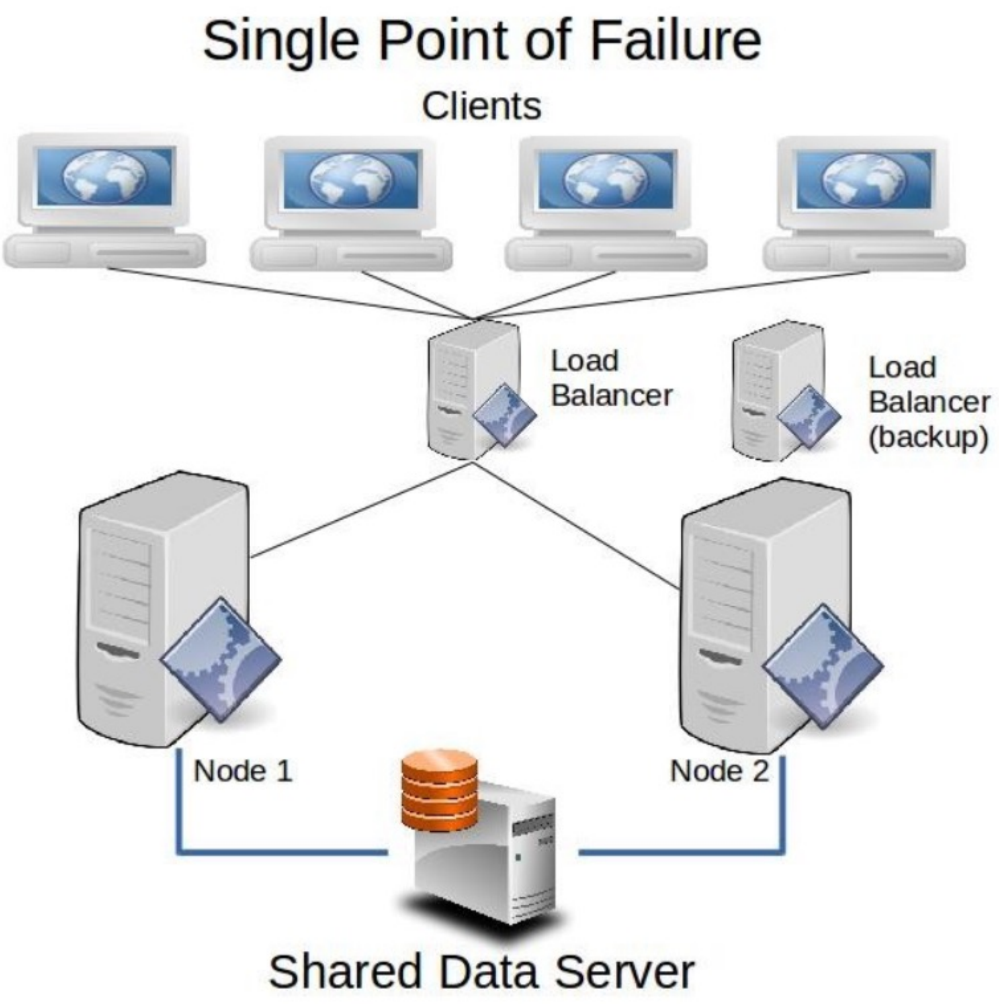


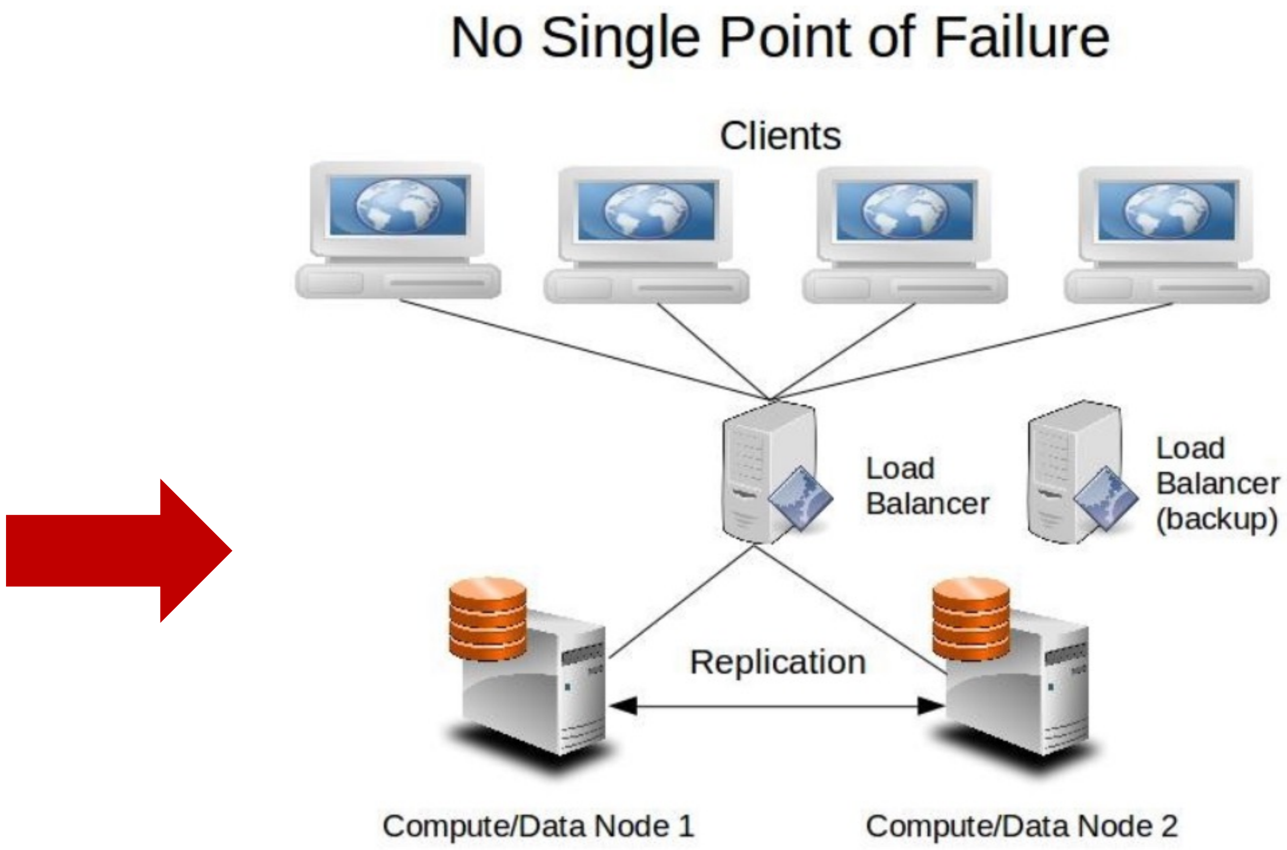


High Availability 2/3

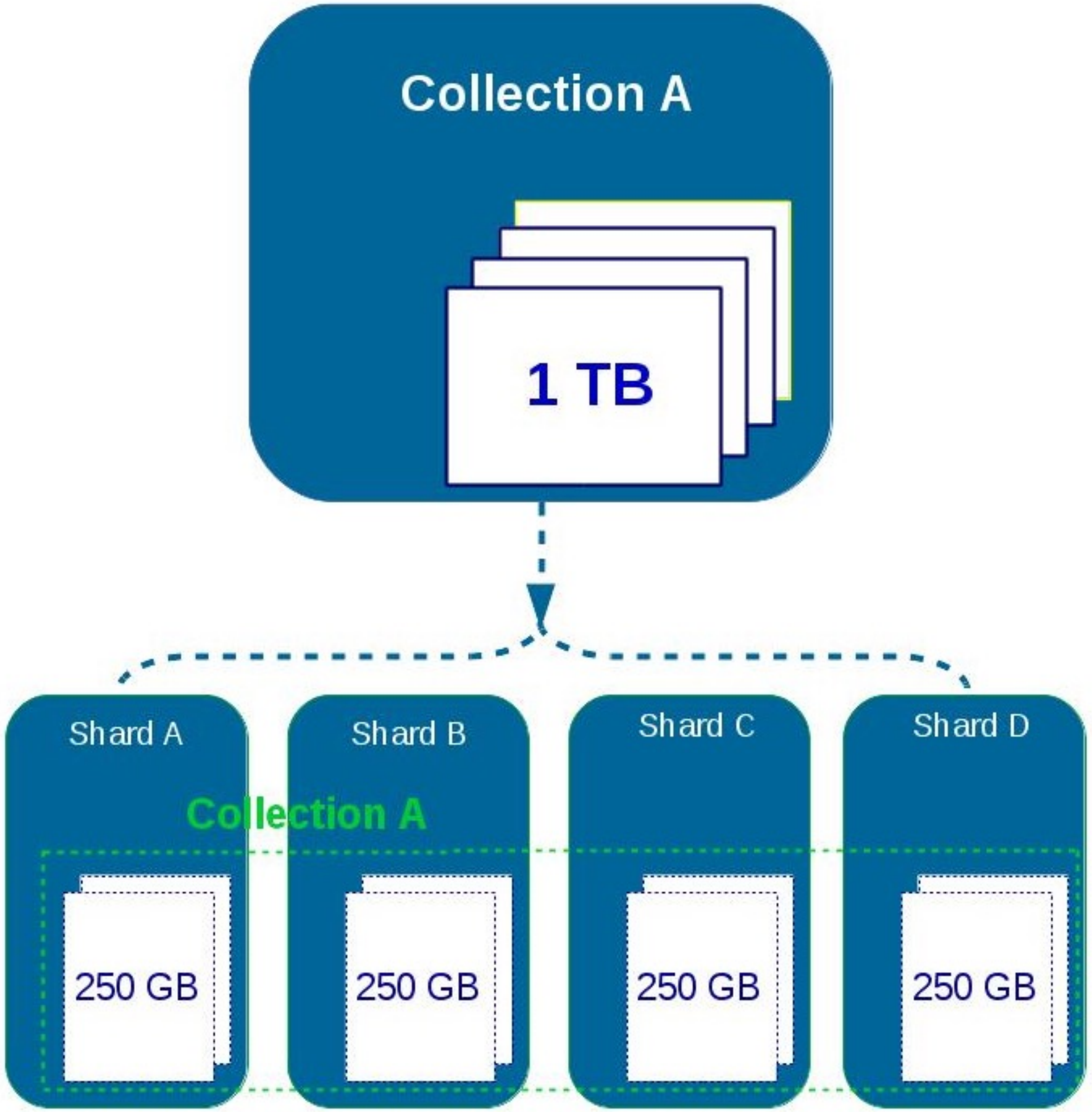


High Availability 3/3





Partitioning (Sharding)



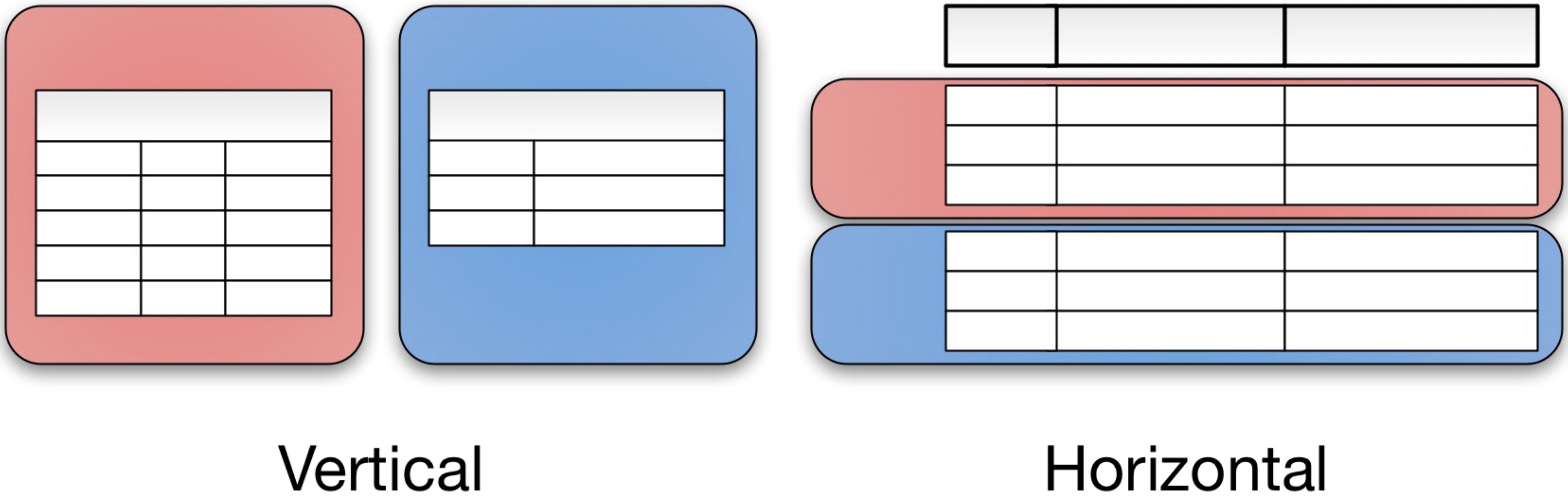
→ Splitting up the data between multiple servers

→ Together they represent the entire collection

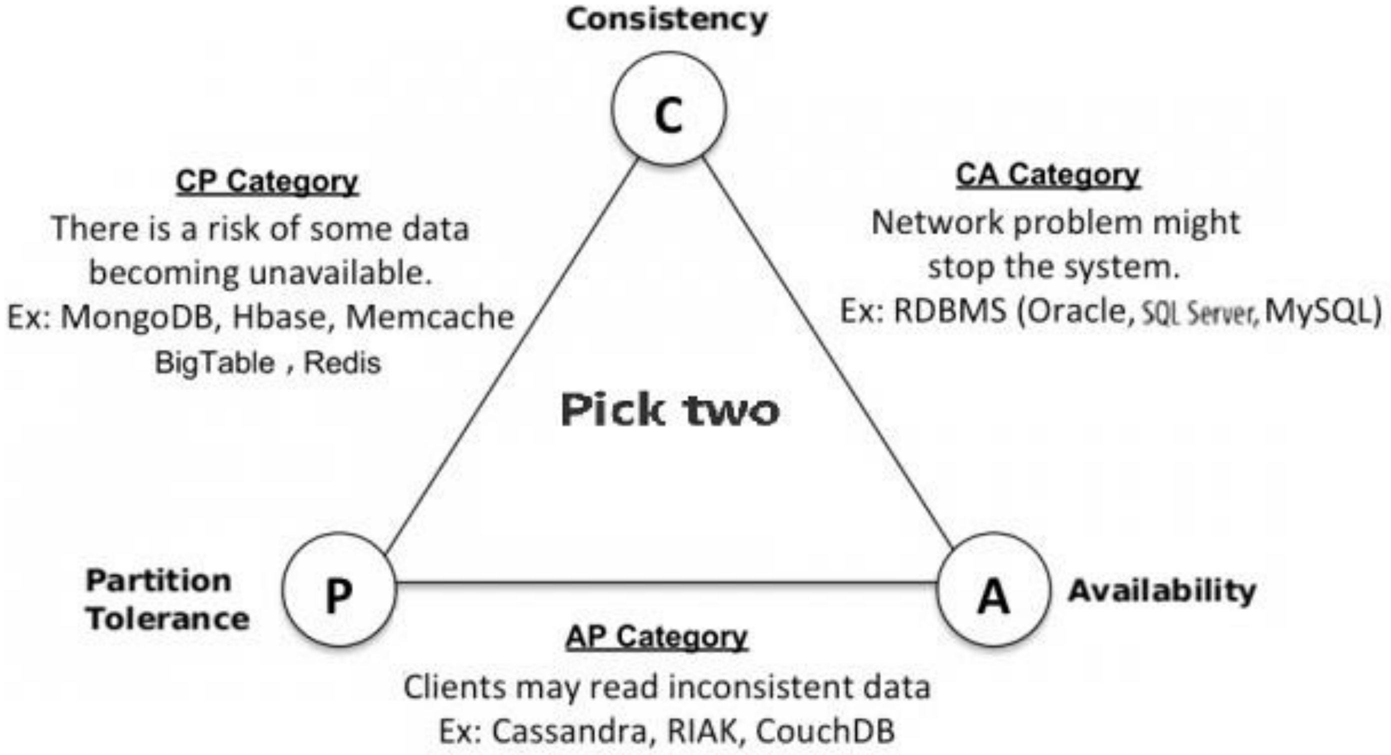
→ Allows for better throughput as calculations and search are shared

→ Makes sense for Big Data Applications

Partitioning (Sharding)



The CAP Theorem – Distributed Databases

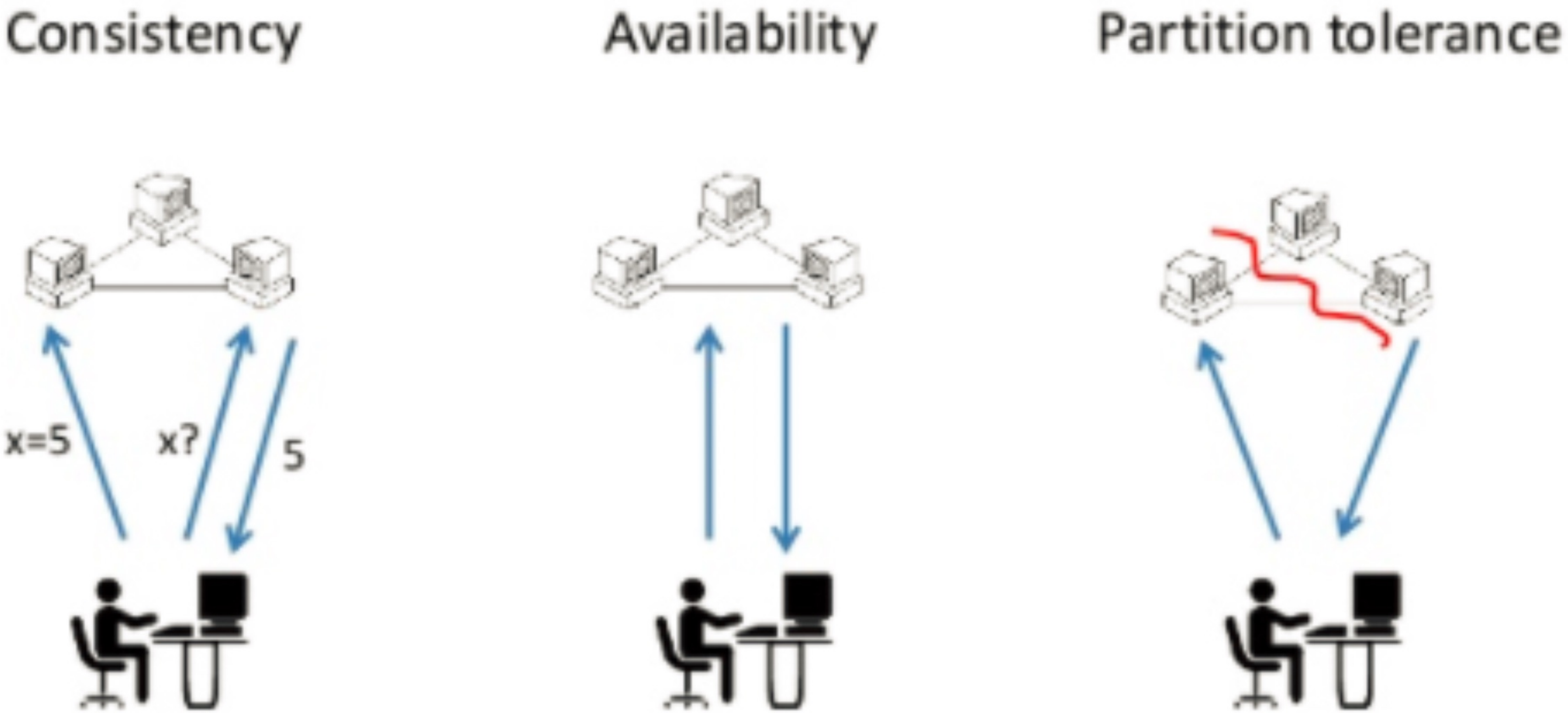

Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

Consistency: Every read recieves the most recent write or an error. All nodes see the same data at all times.

Availability: System is always operational dispite invividual server status. Every Request recives a (non-error) repsonse, without the gurantee that it contains the most recent write

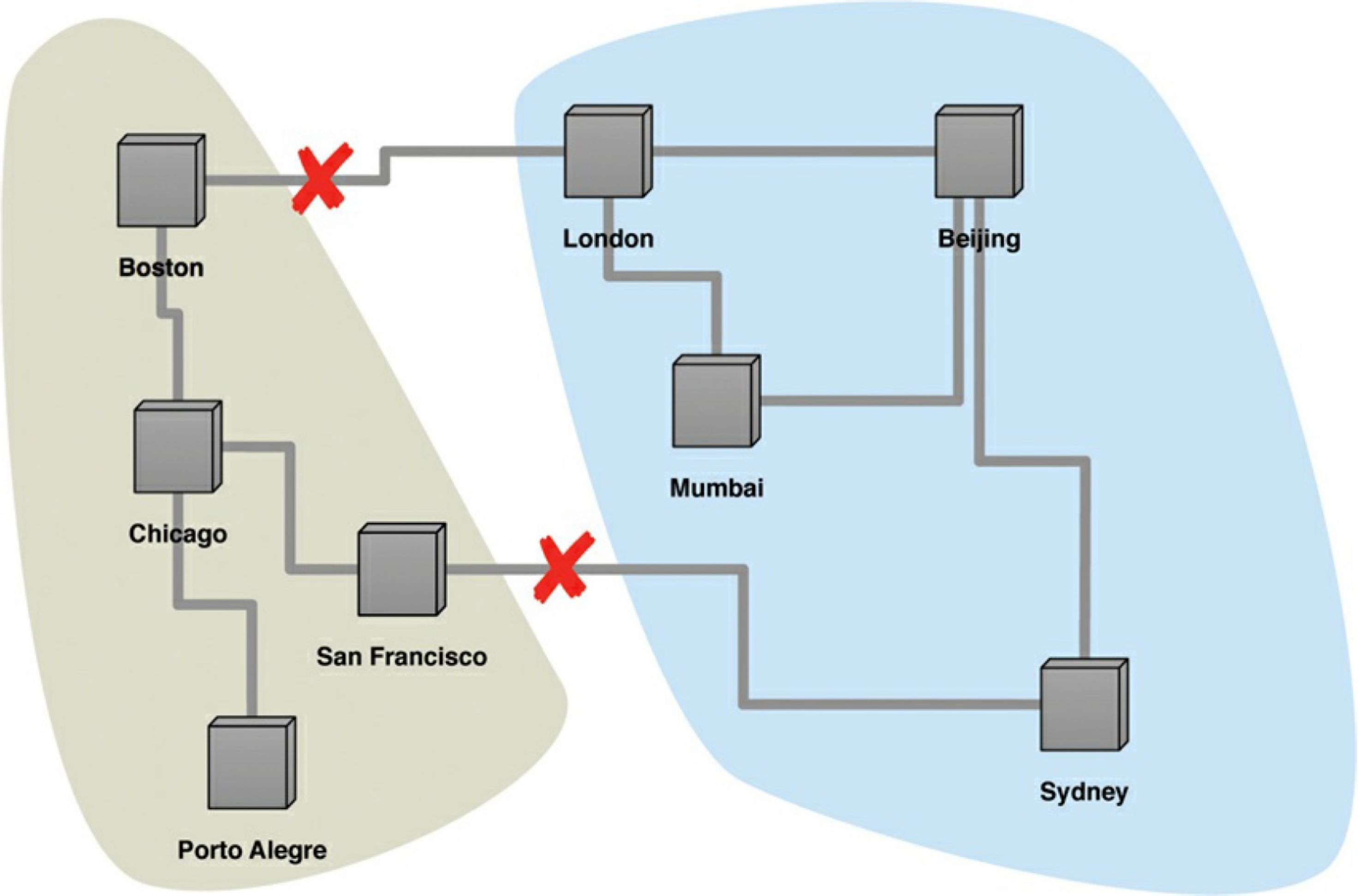
Partition Tolerance: The system continues to operate dispte of partition failure (a node dies with part of the data)

Confusing? Indeed - Read the FAQ: https://github.com/henryr/cap-faq



Source: https://toppertips-bx67a.ondigitalocean.app/cap-theorem/

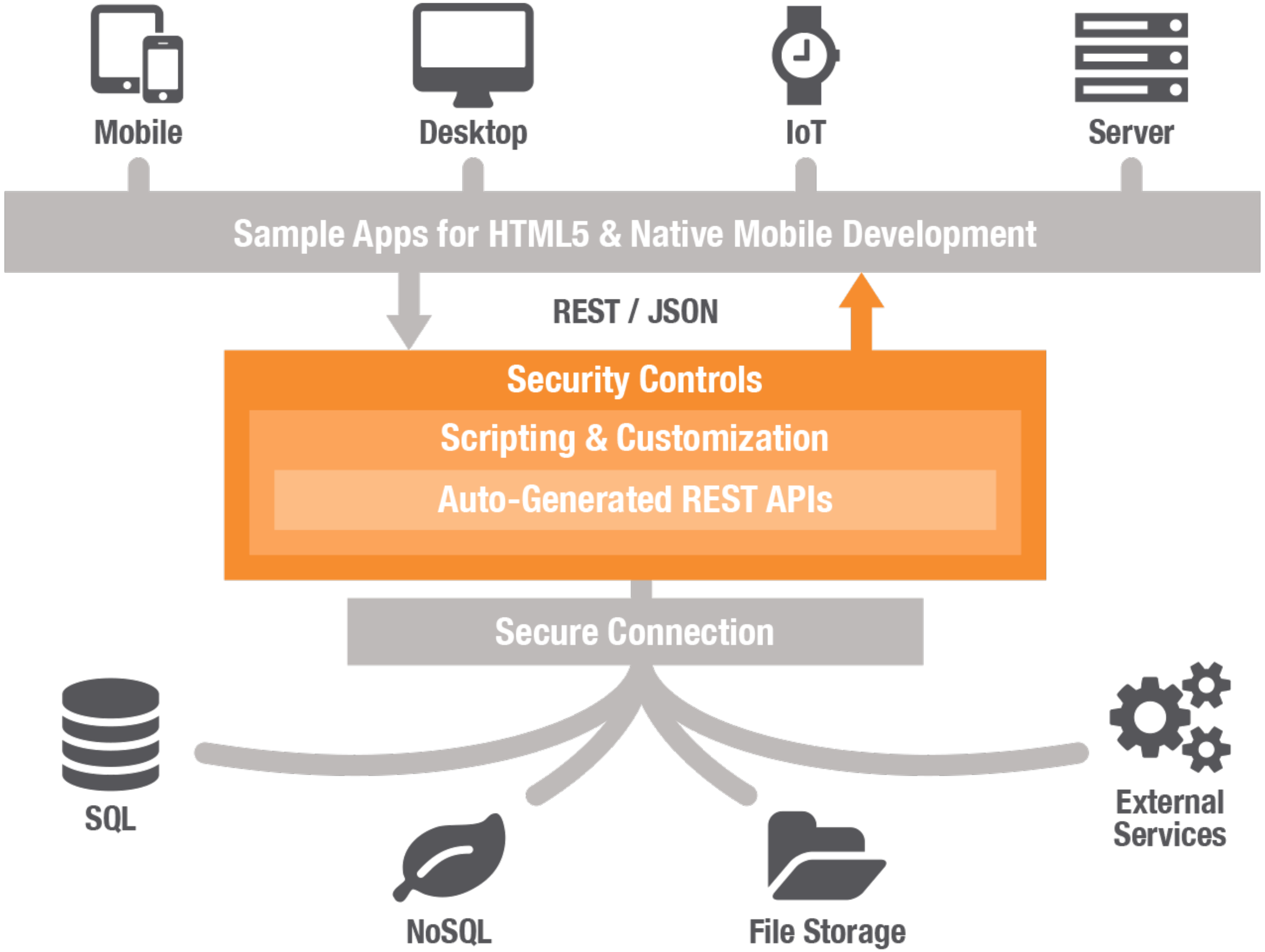
A CAP Theorem example



What is JSON?

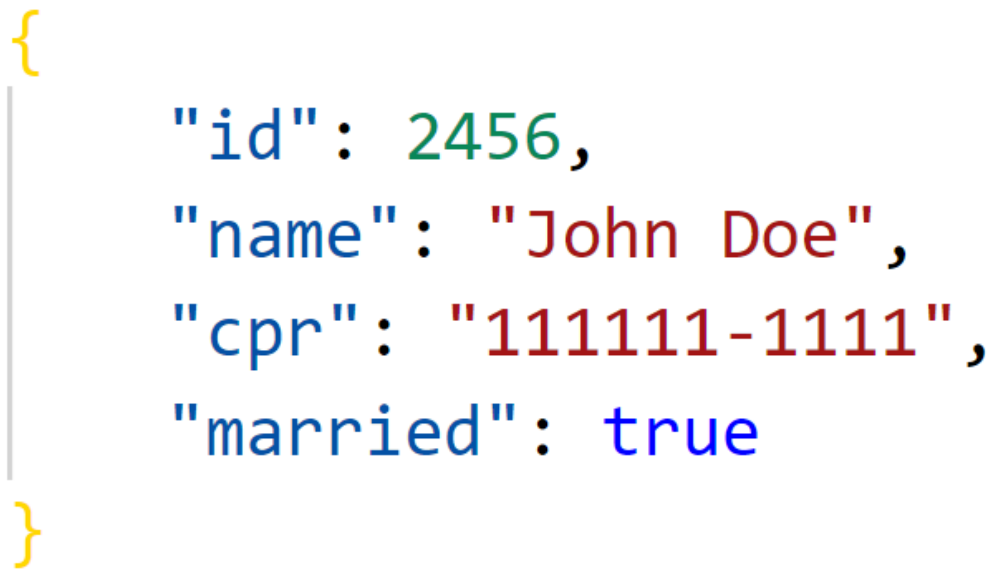
- → JSON is stands for “JavaScript Object Notation”,
- → It’s used to store and exchange data as an alternative solution for XML.
- → Easy to parse, and easy to read and write for a human.
- → Based on JavaScript Object Literals, but it’s a text format.
- → JSON is language independent; means you can use parse and generate JSON data in other programming languages.

How is JSON used?



- → Websites communicating with the backend
- → Mobile applications communicating with the backend
- → Temporary storage format of objects
- → …

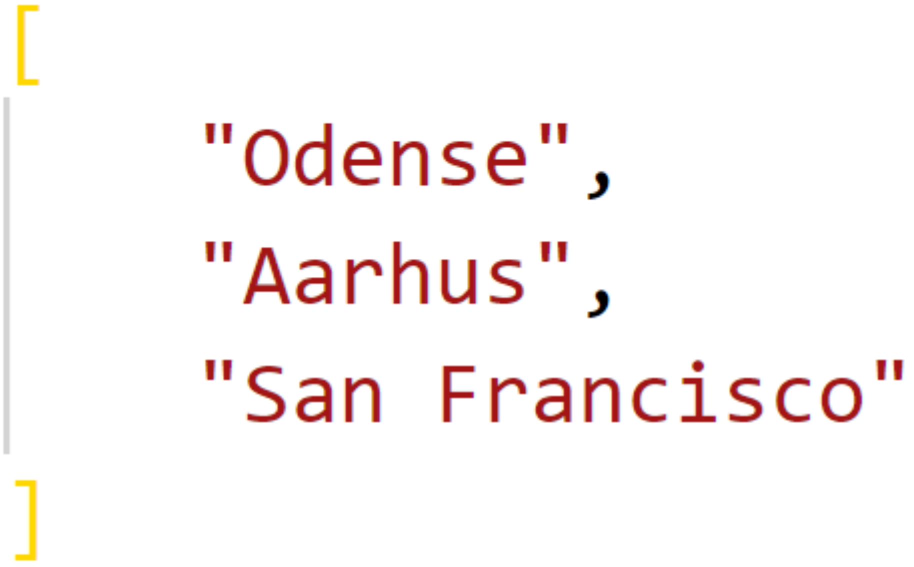
Object Declaration



```
{
    "id": 2456,
    "name": "John Doe",
    "cpr": "111111-1111",
    "married": true
}
```

- → An object starts with { and ends with }.
- → Inside is a collection of Key/Values separated by comma
  - → Be aware it is a syntax error to have a comma after the last key/value
- → Each key is a string, so it uses the double quotes - “key”
- → A : mark separates a key and a value
- → Each value can be of several types. In the example we have number, string, and boolean.

Array Declaration



```
[
    "Odense",
    "Aarhus",
    "San Francisco"
]
```

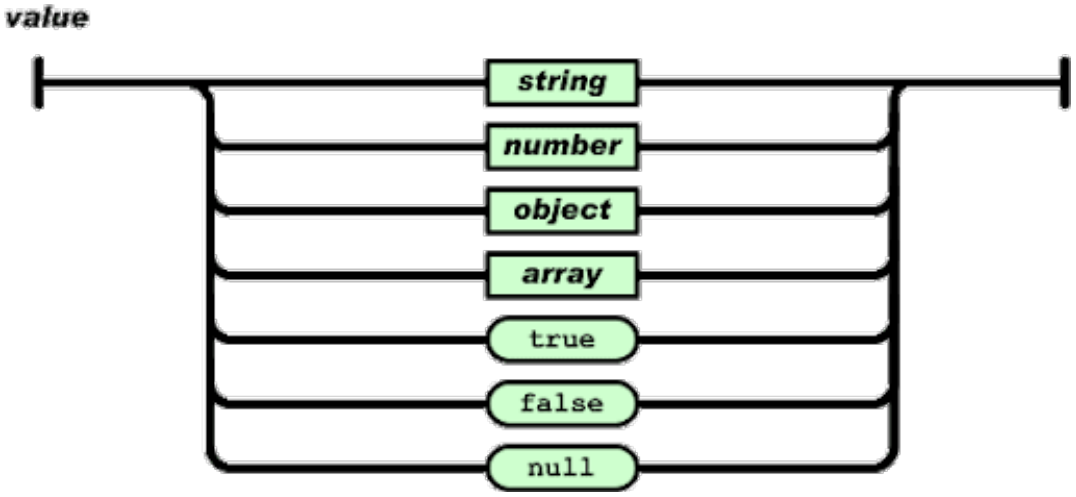
→ Arrays are ordered lists, and starts from 0

→ Initiated with [and ends with]

→ Values are separated by comma

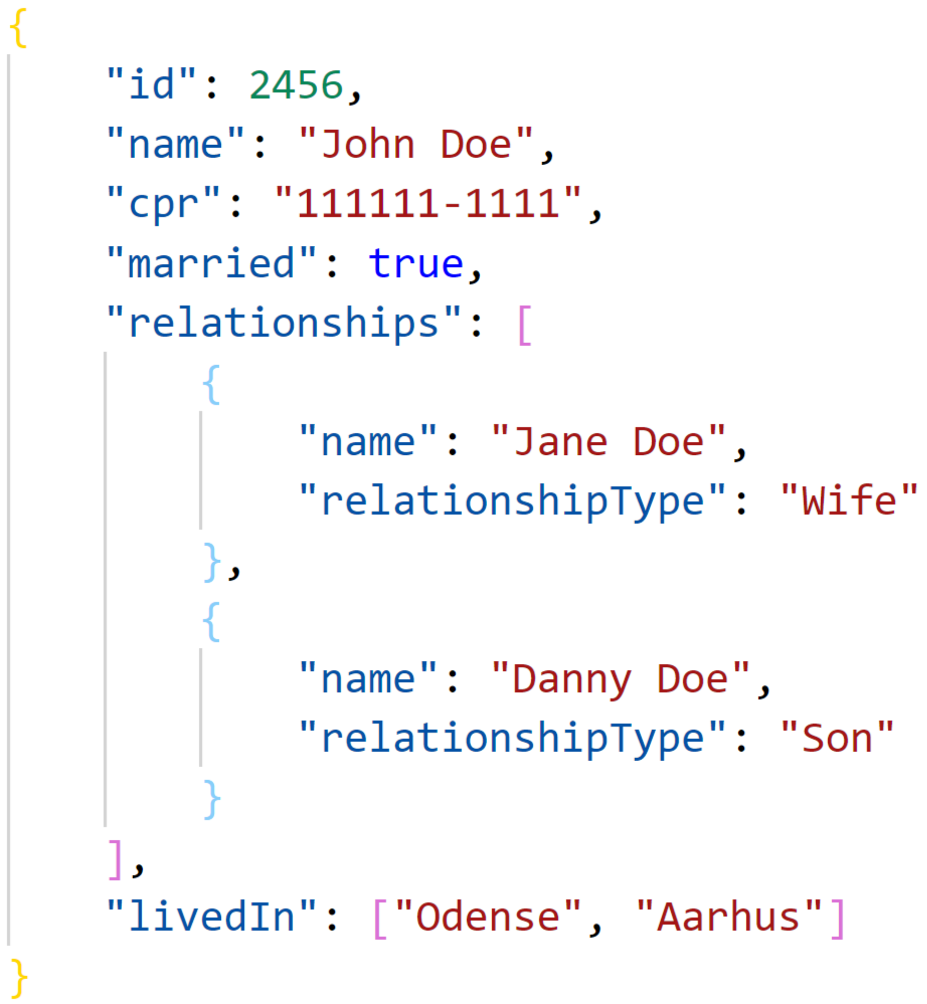
→ An array can also hold objects

JSON Values



→ string; “string”
→ number; 1
→ boolean; true or false
→ null; null/left out
→ an object; {“key”: “value”}
→ an array; [1, “2”, null, true]

Putting it together



```
{
    "id": 2456,
    "name": "John Doe",
    "cpr": "111111-1111",
    "married": true,
    "relationships": [
        {
            "name": "Jane Doe",
            "relationshipType": "Wife"
        },
        {
            "name": "Danny Doe",
            "relationshipType": "Son"
        }
    ],
    "livedIn": ["Odense", "Aarhus"]
}
```