732A54/TDDE31

Big Data Analytics

Database Systems for Big Data

Huanyu Li

LINKÖPING
UNIVERSITY

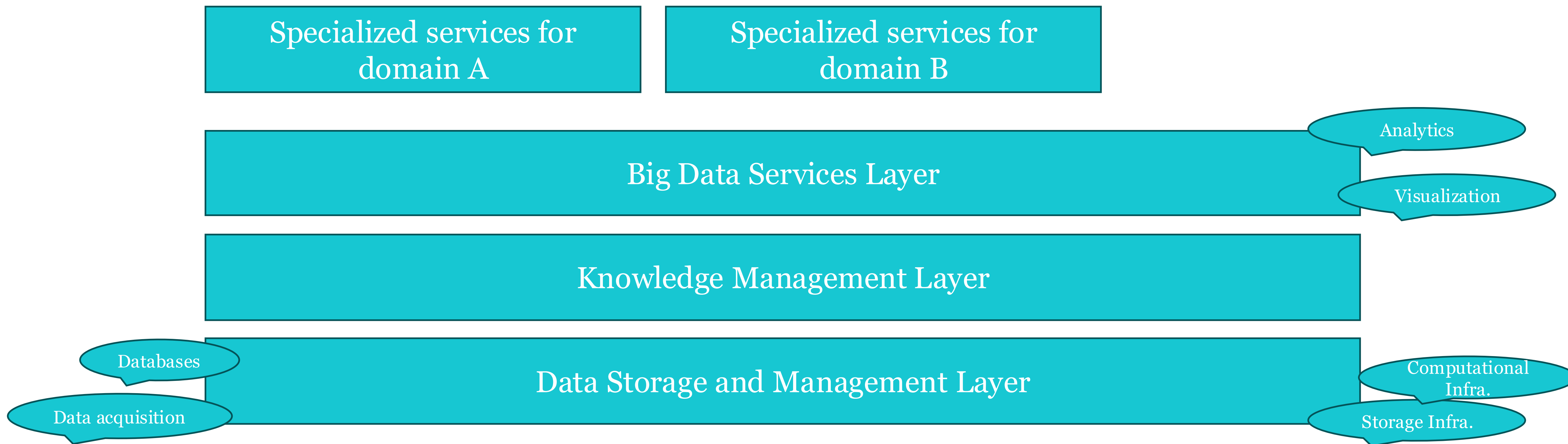
Recap

- What is Big Data and Big Data Analytics?
- Parallel Computing
  - MapReduce, Spark
  - High Performance Computing vs Big Data Computing

Why do we need new databases?

BDA system architecture

- Layered architecture with many different aspects



How to deal with Big Data?

- What we have and why they may/may not work for big data analytics
  - **High Performance Computing architectures**
  - Relational databases

How to deal with Big Data?

- Traditionally, HPCs are suitable for problems requiring more computation on relatively little data
- Big data problems on HPCs will cause lots of input/output (I/O)
  - moving data across the network of the HPC setting
- Big data problems are usually data–intensive jobs

Big Data problems VS HPC problems

Big Data	HPC
Data-intensive jobs	Computation-intensive jobs
Data analytics, graph processing, etc.	Simulation, optimization, etc.
Move jobs to where data is located	Move data to where it will be processed

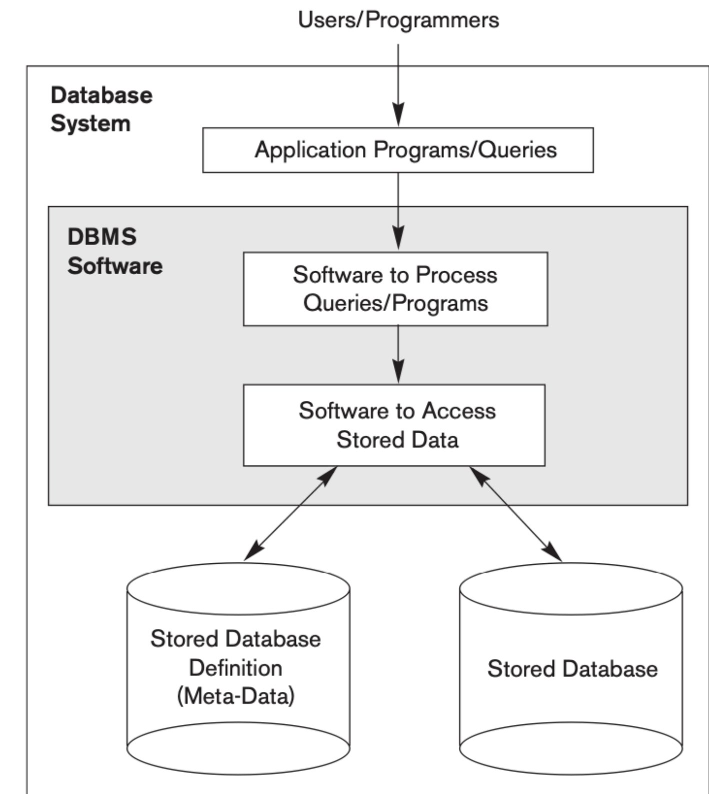
- Data should be easy to be distributed among nodes for BDA
- The compute speed usually is not a goal for BDA, but data volume/variety are challenges

How to deal with Big Data?

- What we have and why they may/may not work for big data analytics
  - High Performance Computing architectures
  - **Relational databases**

Recap of RDB

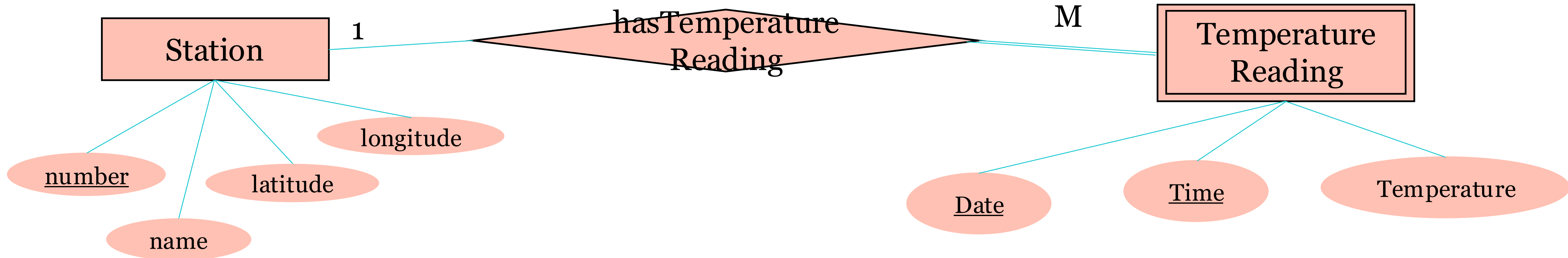
- Database System = Database + DBMS
- Relational DB
  - With well-defined formal foundations
    - Schema level (Entity-Relationship)
    - Relational Model (relation/table, attribute/column, tuple/row)



Elmasri, R. & Navathe, S.B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson. Chapter 1.

Recap of RDB

- ER model



- Relational model
  - Station (<u>number</u>, name, latitude, longitude)

  - TemperatureReading (<u>Date</u>, <u>Time</u>, <u>Station number</u>, Temperature)

Recap of RDB

- With well-defined formal foundations
- SQL – structured query language
  - query, data manipulation, database definition
- Support of transactions with ACID properties
  - Atomicity, Consistency, Isolation, Durability
- Established technology
  - Many vendors
  - Highly mature systems
  - Experienced users and administrators

But the business world has changed...

- More organizations and companies have shifted to the digital economy powered by the Internet

- New IT applications that allow companies to run their business and to interact with customers are required and prioritized
  - Web and Mobile applications
  - Connected devices (“Internet of Things”)

- As a result, new challenges come...

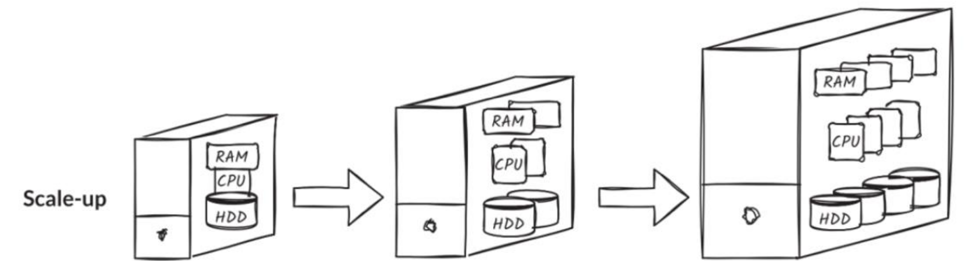
We need to organize data in a flexible way (e.g., schema free), and achieve fault tolerance (e.g., data replication, task-re-execution)

Scalability for big data systems is important

- Data scalability
  - Handle growing amounts of data, **without losing performance**
- Read scalability
  - Handle increasing numbers of read operations, **without losing performance**
- Write scalability
  - Handle increasing numbers of write operations, **without losing performance**

Vertical Scalability and Horizontal Scalability

- Vertical scalability (scale up)
  - Add resources to a server (e.g., more CPUs, more memory, more or bigger disks)



- Horizontal scalability (scale out)
  - Add nodes (more computers) to a distributed system
  - This is an important feature of BDA systems

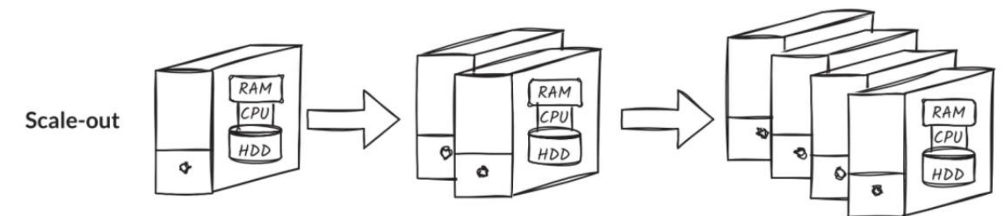


image source: Triguero, I., & Galar, M. (2023). *Large-Scale Data Analytics with Python and Spark: A Hands-on Guide to Implementing Machine Learning Solutions*. Cambridge University Press.

To achieve much higher performance/scalability

- BASE properties:
  - Basically Available: system available whenever accessed, even if parts of it are unavailable
  - Soft state: distributed data does not need to remain consistent at all times
  - Eventually consistent: the state will eventually become consistent after a certain period of time

These properties are usually not guaranteed by relational database systems!

NoSQL in general

- "NoSQL" is interpreted differently (without precise definition)
  - "no to SQL"
  - "not only SQL"
  - "not relational"

- Non-relational databases have been around since the late 1960s
- 1998: first used for an RDBMS without SQL interface
- 2009: picked up again to name the conference "NOSQL 2009" about "open-source, distributed, non-relational databases"

- Since then, "NoSQL database" loosely refers to a class of non-relational DBMSs

Typical Characteristics of NoSQL systems

- Ability to scale horizontally over many commodity servers with high performance, availability and fault tolerance
  - achieved by guaranteeing basically available, soft state, eventually consistent (BASE)
  - and by partitioning and replication of data
- Non-relational data model, no requirements for schemas
- “Typical” means there is a broad variety of such systems, but not all of them have these characteristics to the same degree

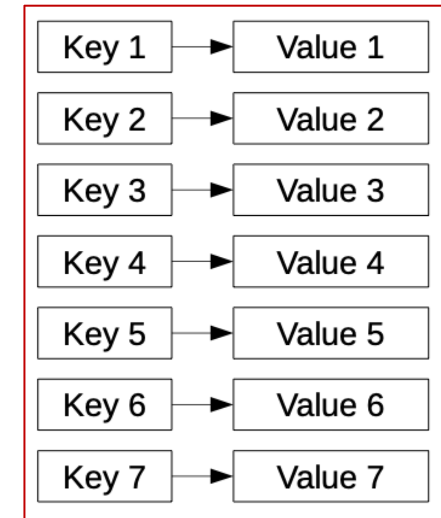
NoSQL models

Data Models for NoSQL

- Key-Value model
- Document model
- Wide-Column models
- Graph database models

NoSQL Data Models – Key-Value Stores

- Simplest form of NoSQL databases
- Schema-free, a dictionary of key-value pairs
  - Keys are unique
  - Values are of arbitrary types
- Efficient in storing distributed data
- Not suitable for
  - Representing structures and relations
  - Accessing multiple items, since the access is by key



Key-Value Stores

- Suppose we have a relational database for the SMHI data used in BDA labs:

Stations

Number	Name	MeasureHeight	Latitude	Longitude
85250	Linköping	2.0	58.4166	15.6333
86340	Norrköping-SMHI	NULL	58.5833	16.15
86360	Norrköping	2.0	58.606	16.2127

- How to represent such data using the key-value model?

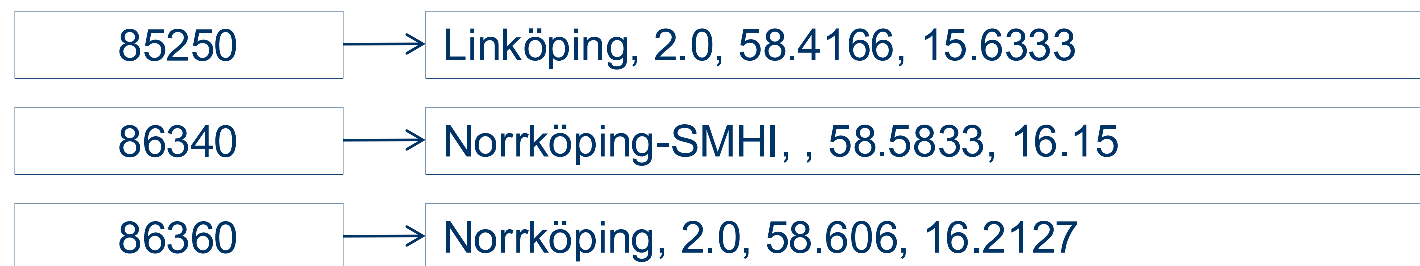
Key-Value Stores

- Suppose we have a relational database for the SMHI data used in BDA labs:

Stations

Number	Name	MeasureHeight	Latitude	Longitude
85250	Linköping	2.0	58.4166	15.6333
86340	Norrköping-SMHI	NULL	58.5833	16.15
86360	Norrköping	2.0	58.606	16.2127

- How to represent such data using the key-value model?



Key-Value Stores

- Let’s add another table

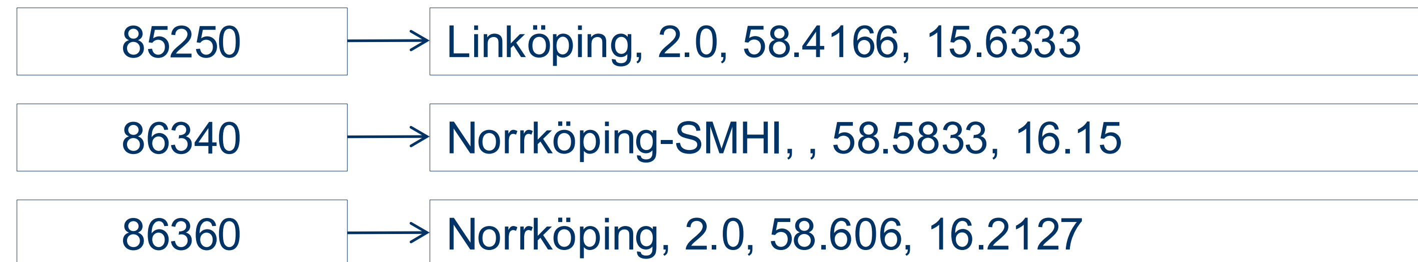
Stations

Number	Name	MeasureHeight	Latitude	Longitude
85250	Linköping	2.0	58.4166	15.6333
86340	Norrköping-SMHI	NULL	58.5833	16.15
86360	Norrköping	2.0	58.606	16.2127

Sensors

Number	Sensor
85250	temperature_s1
85250	temperature_s2
86360	temperature_s1

- How to update the key-value model?



Key-Value Stores

- Let's add another table

Stations

Number	Name	MeasureHeight	Latitude	Longitude
85250	Linköping	2.0	58.4166	15.6333
86340	Norrköping-SMHI	NULL	58.5833	16.15
86360	Norrköping	2.0	58.606	16.2127

Sensors

Number	Sensor
85250	temperature_s1
85250	temperature_s2
86360	temperature_s1

- How to update the key-value model?

Key-Value Stores: Querying

- CRUD operations are only based on keys
  - Create, Read, Update, Delete
  - put(key, value), get(key), delete(key)

- Value-related queries are not supported
  - Recall that values are opaque to the system (i.e., no secondary index over values)

- Accessing multiple items requires separate requests

- However, partitioning the data based on keys (horizontal partitioning or sharding) and distributed processing can be very efficient

Key-Value Stores: Querying

- If we want to find all stations which have a particular sensor type
  - It is possible, but very inefficient
  - An efficient way is to use 'sensor type' as the key in queries

Stations

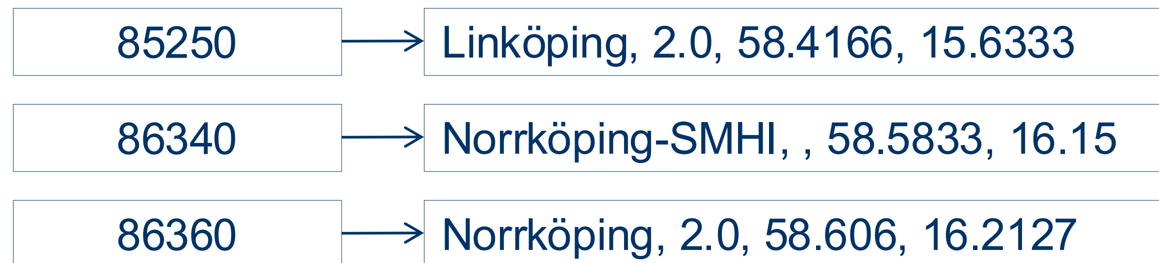
Number	Name	MeasureHeight	Latitude	Longitude
85250	Linköping	2.0	58.4166	15.6333
86340	Norrköping-SMHI	NULL	58.5833	16.15
86360	Norrköping	2.0	58.606	16.2127

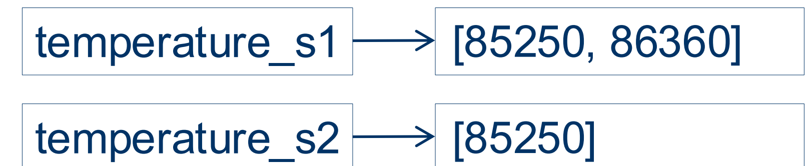
Sensors

Number	Sensor_type
85250	temperature_s1
85250	temperature_s2
86360	temperature_s1

Key-Value Stores: Querying

- If we want to find all stations which have a particular sensor type
- It is possible, but very inefficient
- What can we do to make it more efficient?
  - Add redundancy (downsides: more space needed, updating becomes less trivial and less efficient)





Key-Value Store Examples

- Open-source examples
  - Redis
  - Memcached

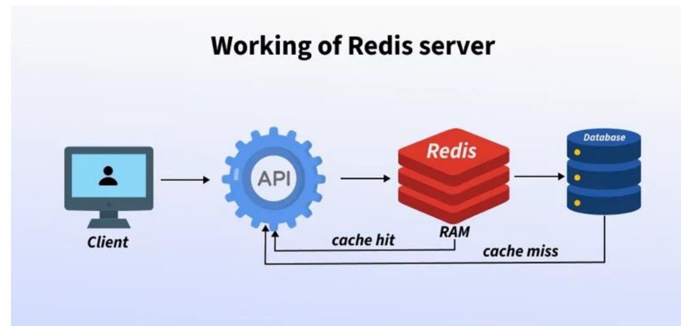


image source: https://www.geeksforgeeks.org/system-design/introduction-to-redis-server/

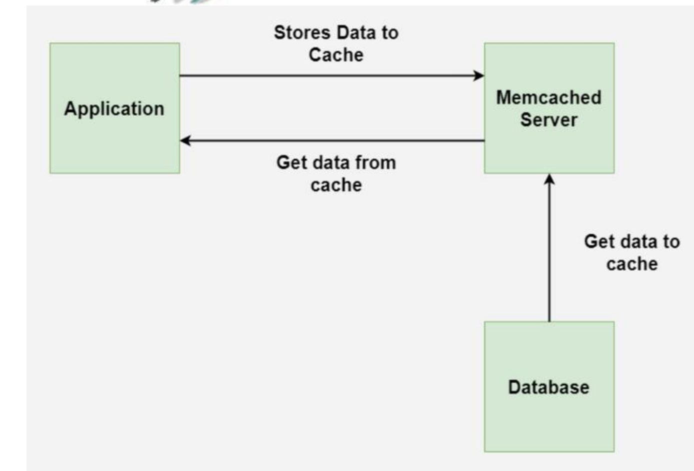


image source: https://www.geeksforgeeks.org/system-design/what-is-memcached/

Redis

- In-memory
- Store data with keys
- Retrieve data using keys
- Data Types
  - String (byte string, integer or floating-point values), List (of strings), Set (of strings), JSON, Hashes, etc.
- Usage Scenarios
  - Caching
  - Session management (web or game applications)

Memcached

- In-memory
- Data Types (Strings only)
- Simple data operation by keys
- Usage scenarios
  - Web applications (small chunks of data e.g., results of database calls, API calls)
  - Session management
  - Database query caching

Data Models for NoSQL

- Key-Value model
- Document model
- Wide-Column models
- Graph database models

NoSQL Data Models – Document Stores

- Store data as documents
- A dictionary of key-value pairs
  - Keys are unique
  - Values are documents, semi-structured data
    - XML, JSON, BSON (binary), etc.
- Efficient in storing distributed data
- Not suitable for
  - Representing structures and relations
  - Accessing multiple items, since the access is by key

NoSQL Data Models – Document Stores

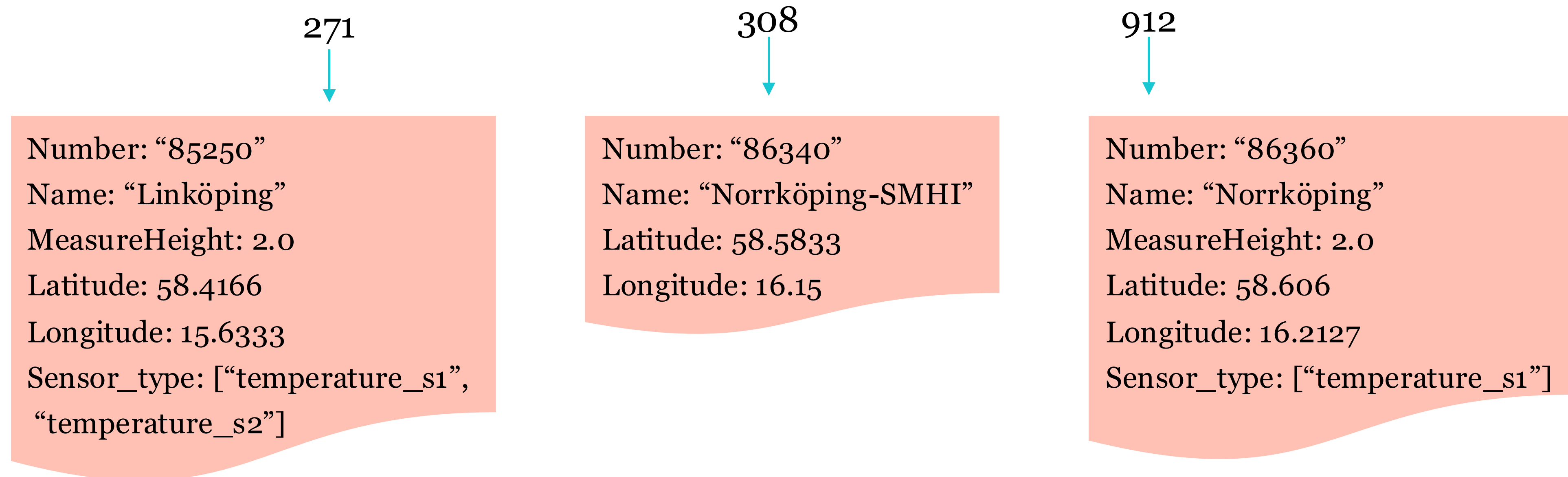
Number: “85250”
Name: “Linköping”
MeasureHeight: 2.0
Latitude: 58.4166
Longitude: 15.6333
Sensor_type: [“temperature_s1”, “temperature_s2”]

Number	Name	MeasureHeight	Latitude	Longitude
85250	Linköping	2.0	58.4166	15.6333
86340	Norrköping-SMHI	NULL	58.5833	16.15
86360	Norrköping	2.0	58.606	16.2127

Number	Sensor_type
85250	temperature_s1
85250	temperature_s2
86360	temperature_s1

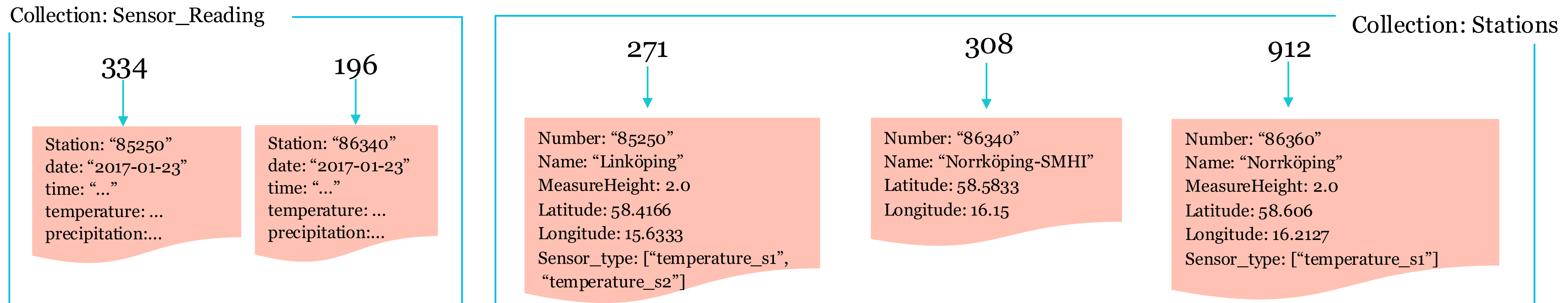
NoSQL Data Models – Document Stores

- Document store-based Databases
  - A set of documents (or multiple such sets)
  - Each document additionally associated with a unique identifier
  - Schema-free: different documents may have different fields



NoSQL Data Models – Document Stores

- Document store-based Databases
  - A set of documents (or multiple such sets)
  - Each document additionally associated with a unique identifier
  - Schema-free: different documents may have different fields
  - Grouping of documents into separate sets (called “domains” or “collections”)



NoSQL Data Models – Document Stores

- Document store-based Databases
  - A set of documents (or multiple such sets)
  - Each document additionally associated with a unique identifier
  - Schema-free: different documents may have different fields
  - Grouping of documents into separate sets (called “domains” or “collections”)
  - Partitioning based on collections and/or document IDs
  - Secondary indexes over fields in the documents possible
    - Different indexes per domain/collection of documents

Document Stores: Querying

- Querying in terms of conditions on document content
- Depending on specific systems, queries may be expressed:
  - program code using an API
  - in a system-specific query language

Document Stores: Querying

- Querying in terms of conditions on document content
- Depending on specific systems, queries may be expressed:
  - program code using an API
  - in a system-specific query language
- Examples (based on MongoDB's query language)
  - Find all docs in collection Stations whose name field include "Norrköping"
    - `db.Stations.find({Name: {$regex: "Norrköping"}})`
  - Find all docs in collection Stations whose latitude is greater than 58
    - `db.Stations.find({Latitude: {$gt: 58}})`
  - Find all docs in collection Stations which contain specific sensor types
    - `db.Stations.find({Sensor_type: {$in: ["temperature_s1", "temperature_s2"]}})`

Number: "86360"
Name: "Norrköping"
MeasureHeight: 2.0
Latitude: 58.606
Longitude: 16.2127
Sensor_type: ["temperature_s1"]

Document Store Example

- MongoDB
  - Stores data records as BSON documents (Binary representation of JSON documents, but more data types than JSON)

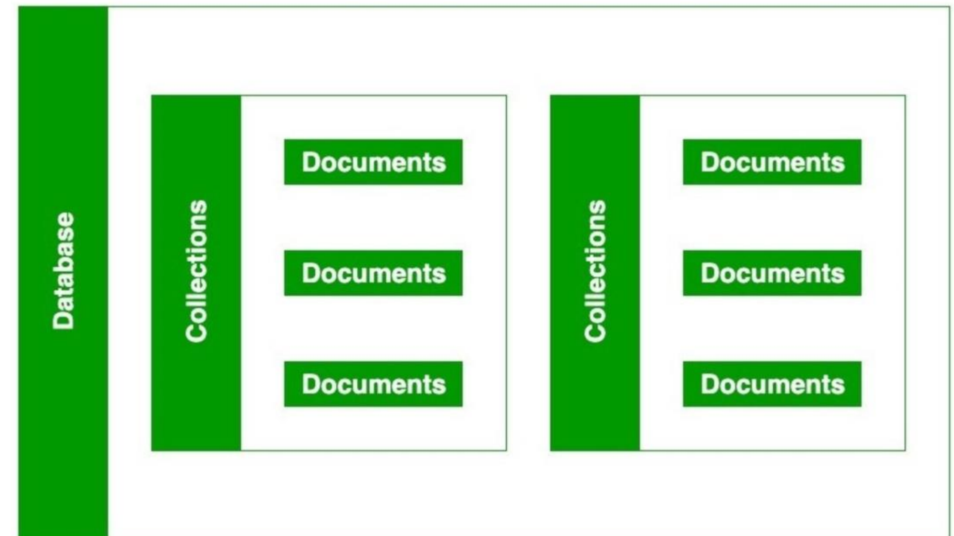


image source: https://www.geeksforgeeks.org/mongodb/what-is-mongodb-working-and-features/

Data Models for NoSQL

- Key-Value model
- Document model
- Wide-Column models
- Graph database models

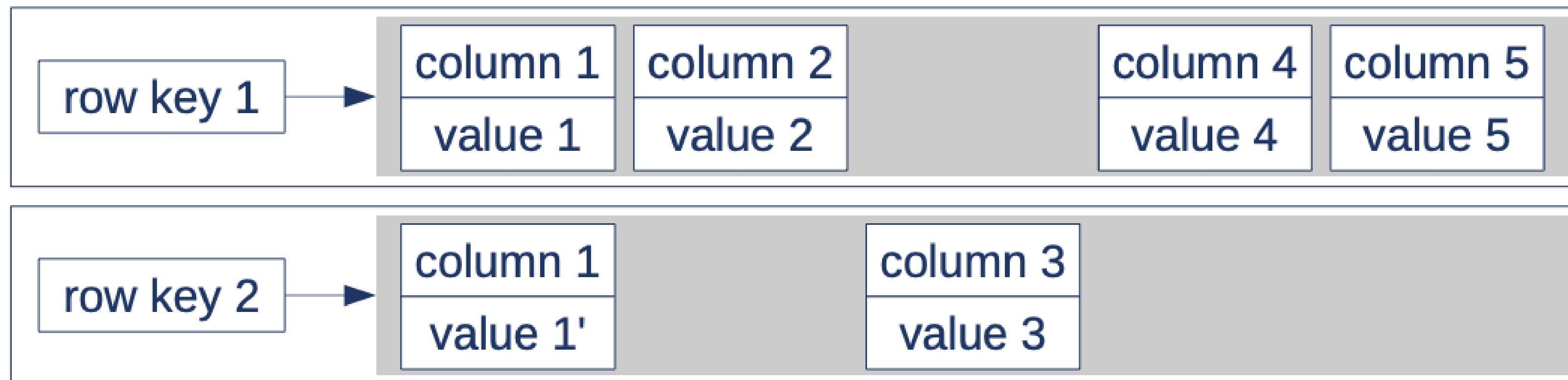
NoSQL Data Models – Wide-Column Stores

- Also called column-family or extensible-record stores
- Store data in rows, each row has a unique key and column families
- Schema-free
  - Keys are unique
  - Values are varying column families
  - Columns consist of key-value pairs

NoSQL Data Models – Wide-Column Stores

- Like a single, very wide relation (table)
- but extensible, schema-free, potentially sparse

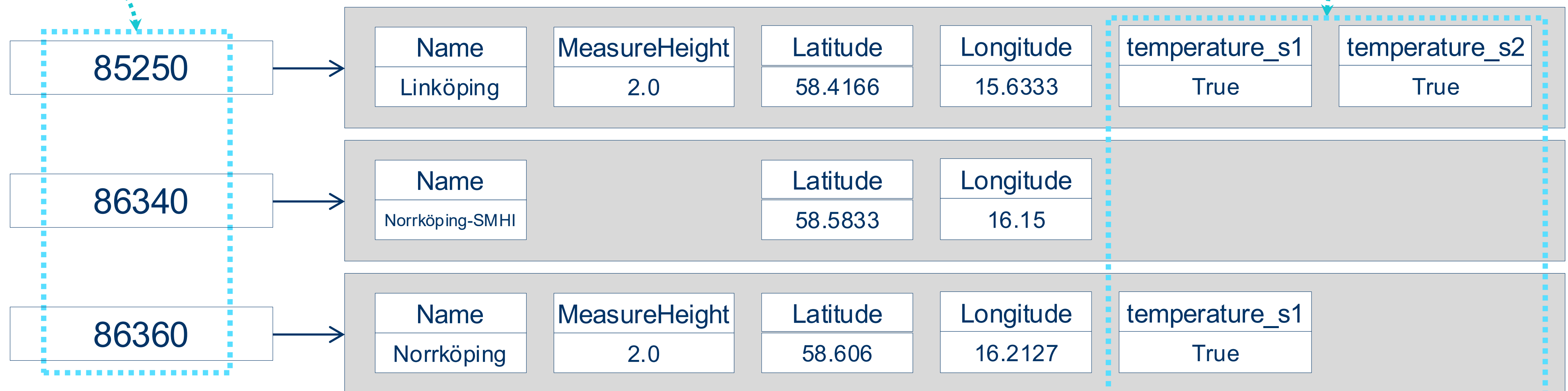
- Like the document model without nesting



NoSQL Data Models – Wide-Column Stores

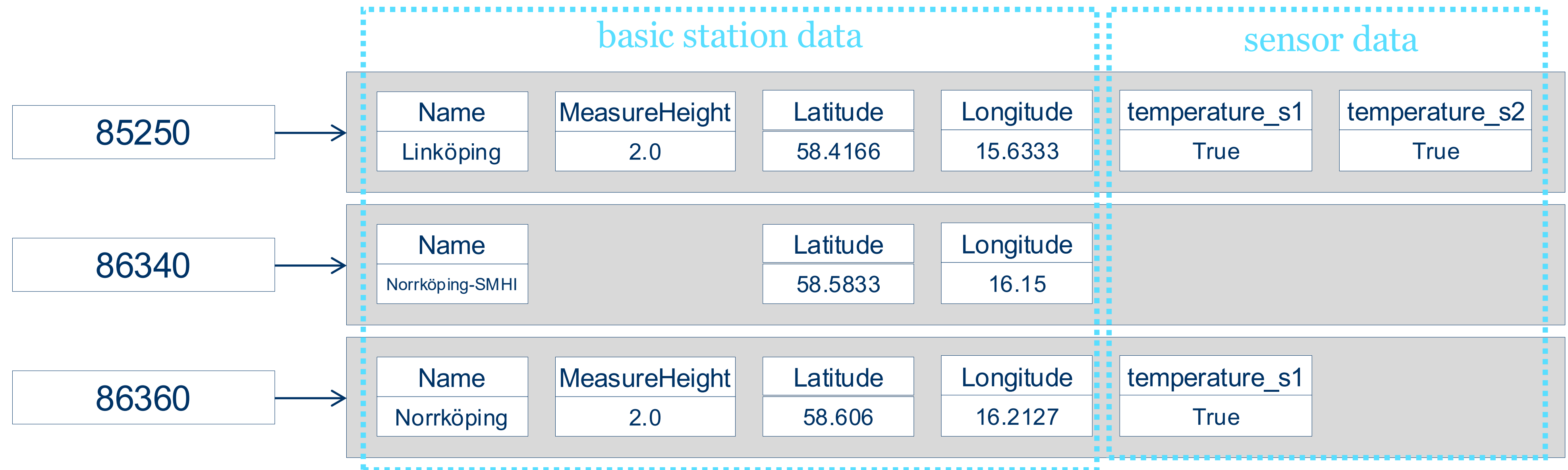
Number	Name	MeasureHeight	Latitude	Longitude
85250	Linköping	2.0	58.4166	15.6333
86340	Norrköping-SMHI	NULL	58.5833	16.15
86360	Norrköping	2.0	58.606	16.2127

Number	Sensor_type
85250	temperature_s1
85250	temperature_s2
86360	temperature_s1



NoSQL Data Models – Wide-Column Stores

- Columns may be grouped into “column families”
  - Therefore, values are addressed by row key, column family, and column key



NoSQL Data Models – Wide-Column Stores

- Columns may be grouped into “column families”
  - Therefore, values are addressed by row key, column family, and column key
- Data may be partitioned …
  - based on row keys (horizontal partitioning),
  - but also based on column families (vertical partitioning),
  - or even on both
- Secondary indexes can be created over arbitrary columns

Wide-Column Stores: Querying

- Querying in terms of keys or conditions on column values
- Conceptually similar to queries in document stores
  - program code using an API
  - in a system-specific query language
  - Again, no joins, these have to be implemented in the application logic
- Better than key value stores for querying and indexing
- Not suitable for
  - Representing structures and relations

Wide-Column Store Examples

- Google Cloud Bigtable (introduced in 2006)
  - master-slave architecture
  - Tables, Rows
  - Columns (Column Family and Column Qualifier)
  - Cells with timestamps
- Use cases
  - Time-series information
  - IoT data

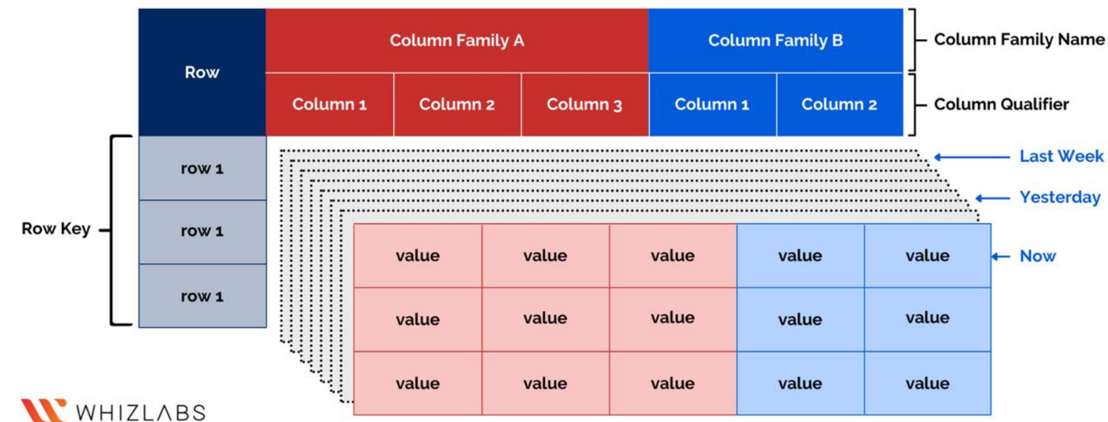


image source: https://www.whizlabs.com/blog/what-is-a-bigtable/

Wide-Column Store Examples

- Apache HBase
  - master-slave architecture
  - built on top of Hadoop and HDFS
  - Compared with pure HDFS
    - HBase supports row key lookup (faster key-based queries)

- Apache Cassandra (originally from Facebook, its source code was made open-source in 2008)
  - masterless ring of nodes
  - logical grouping of nodes as a data center

Schema-free for NoSQL

- For data modeling in relational databases
  - From conceptual models (ER) to relational models (tables)
    - Joining tables when you need to get data from different entities or data based on relationships
- For data modeling in schema-free stores
  - In general,
    - Schema can change over time if the needs of an application change
    - Data can be stored in different collections

Data Models for NoSQL

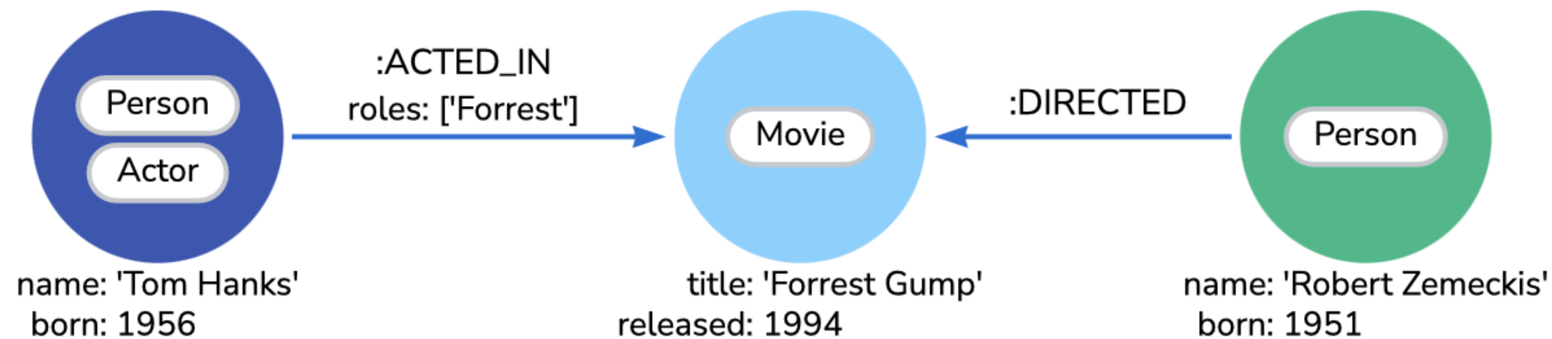
- Key-Value model
- Document model
- Wide-Column models
- Graph database models

Graph Database Models

- Graph databases
  - Prevalent data model: property graphs
  - Typically, navigational queries
- Graph processing systems
  - Prevalent data model: generic graphs
  - Typically, complex graph analysis tasks

Graph Database Models

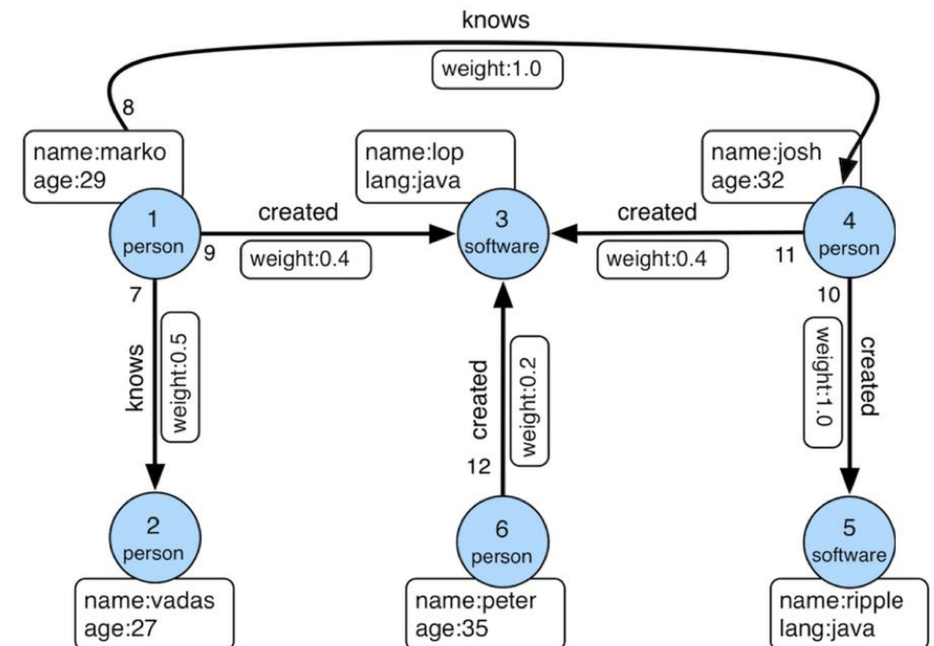
- Nodes represent entities of a domain
- Edges represent relationships/connections among nodes
- Labels represent the kind of nodes/edges
- Properties are associated with nodes and edges



A property graph example: https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/

Graph Database Models – Property Graph

- Directed multigraph
  - Multiple edges between the same pair of nodes
- Any node and any edge may have a label
- Any node and any edge may have an arbitrary set of key-value pairs ("properties")



Examples of Graph DB Systems

- Systems that focus on graph databases
  - Neo4j
  - Sparksee
  - Titan
  - Infinite Graph
- Multi-model NoSQL databases with support for graphs
  - OrientDB
  - ArangoDB

*Sparksee

NoSQL Techniques

NoSQL system features

- NoSQL consistency models and basic techniques
  - ACID, BASE and CAP
  - Consistent Hashing, Master/Slave (HDFS)
  - Vector clock

Typical features of NoSQL systems

- Ability to scale horizontally over many commodity servers with high performance, availability and fault tolerance
  - achieved by guaranteeing basically available, soft state, eventually consistent
    - in other words, by giving up ACID guarantees
  - and by partitioning and replication of data
- Non-relational data model, no requirements for schemas
- "Typical" means there is a broad variety of such systems, but not all of them have these characteristics to the same degree

ACID in database systems at transaction level

- Transaction: An application-specified, atomic and durable unit of work (a process) that comprises one or more database access operations

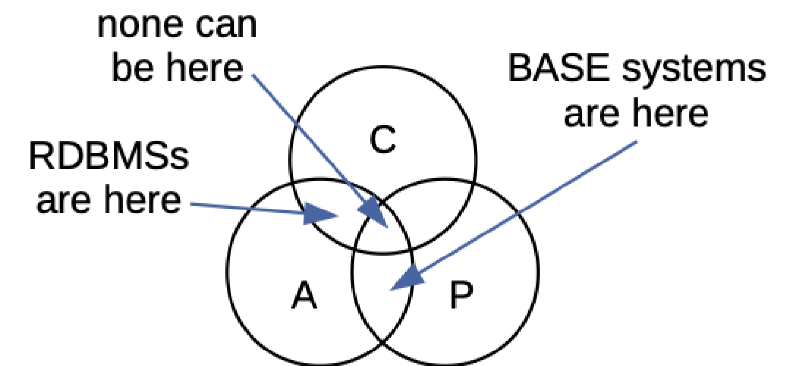
- Atomicity
  - The entire transaction takes place at once or doesn’t happen at all
- Consistency
  - The database must be consistent before and after the transaction
- Isolation
  - Multiple transactions occur independently without interference
- Durability
  - The changes of a successful transaction persist even if the system failure occurs

NoSQL, BASE rather than ACID

- Giving up ACID guarantees, to achieve much higher performance and scalability
  - Basically Available
    - System available whenever accessed, even if parts of it are unavailable
  - Soft state
    - distributed data does not need to remain consistent at all times
  - Eventually consistent
    - the state will eventually become consistent after a certain period of time

CAP Theorem for distributed data store

- Consistency
  - After an update, all readers in a distributed system see the same data
  - All nodes are supposed to contain the same data at all times
- Availability
  - All requests will be answered, regardless of crashes or downtimes
- Partition Tolerance
  - System continues to operate, even if two sets of servers get isolated



Only 2 of 3 properties can be guaranteed at the same time in a distributed system with data replication

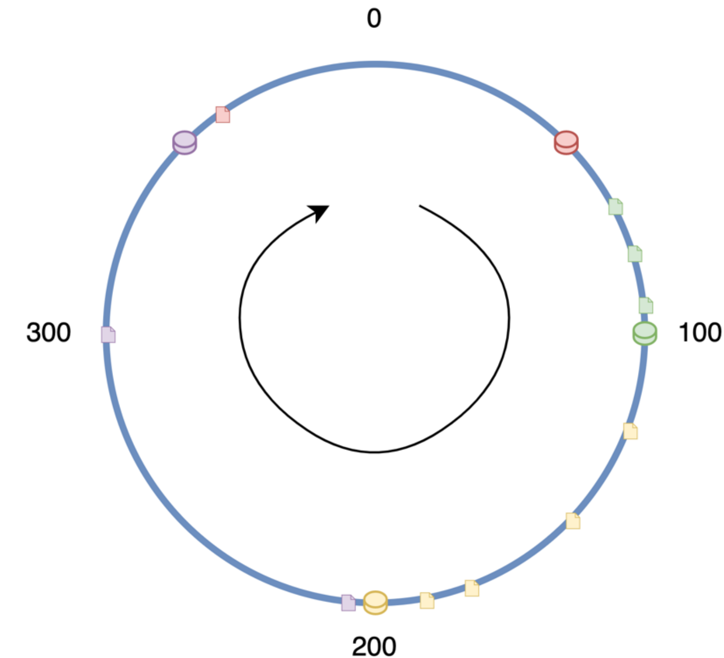
NoSQL Techniques

- Basic techniques (widely applied in NoSQL systems)
  - Distributed data storage, replication (Consistent hashing)
  - Recognize order of distributed events and potential conflicts (Vector clock)
  - Distributed query strategy (MapReduce)

Consistent Hashing

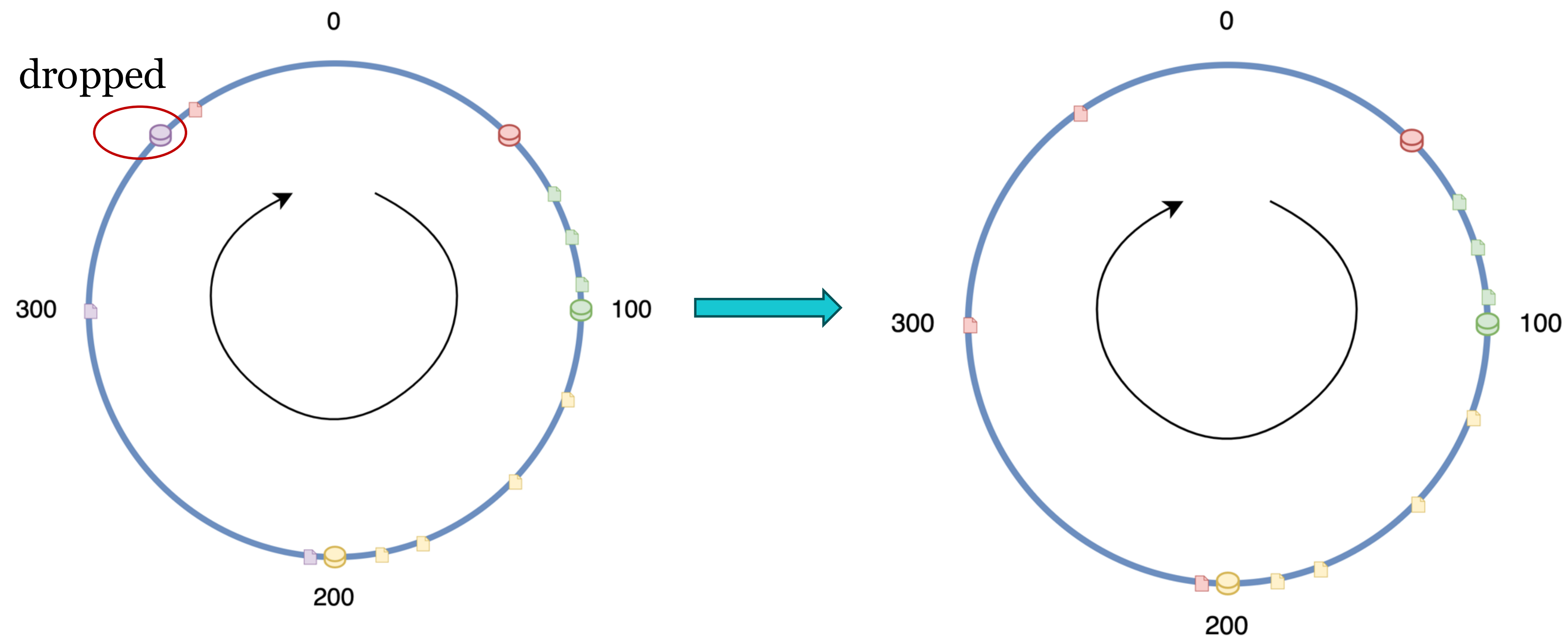
- A virtual ring structure (hash ring)
- Use the same hashing function to hash both the node (server) identifiers (IP addresses) and data keys

- The ring is traversed in the clockwise direction
- Each node is responsible for the region of the ring between the node and its predecessor on the ring



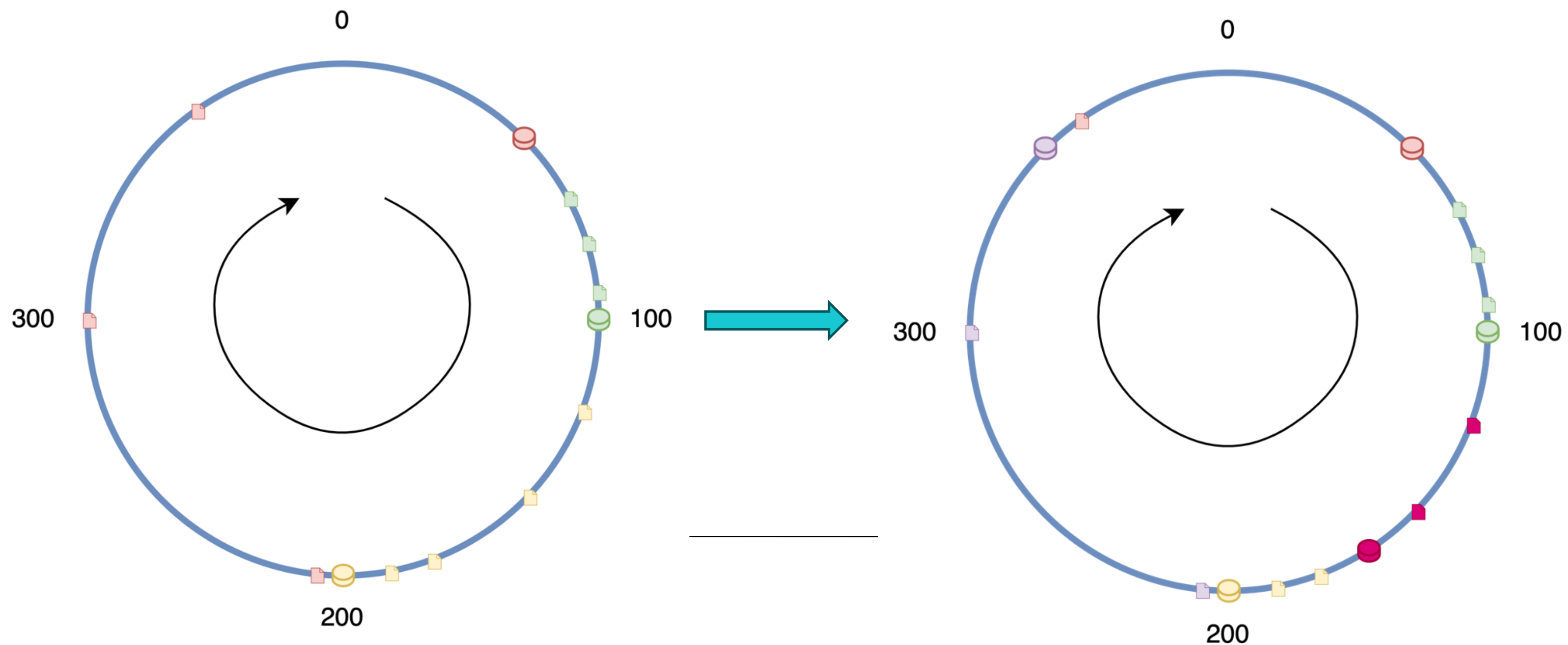
Consistent Hashing – Node Removal

- If a node is dropped out or gets lost
  - Its responsible data will be redistributed to an adjacent node



Consistent Hashing – Node Addition

- If a node is added
  - Its hash value is added to the hash space/ring
  - The hash realm is repartitioned, and hash data will be transferred to a new neighbor
  - No need to update remaining nodes



NoSQL – Vector clock

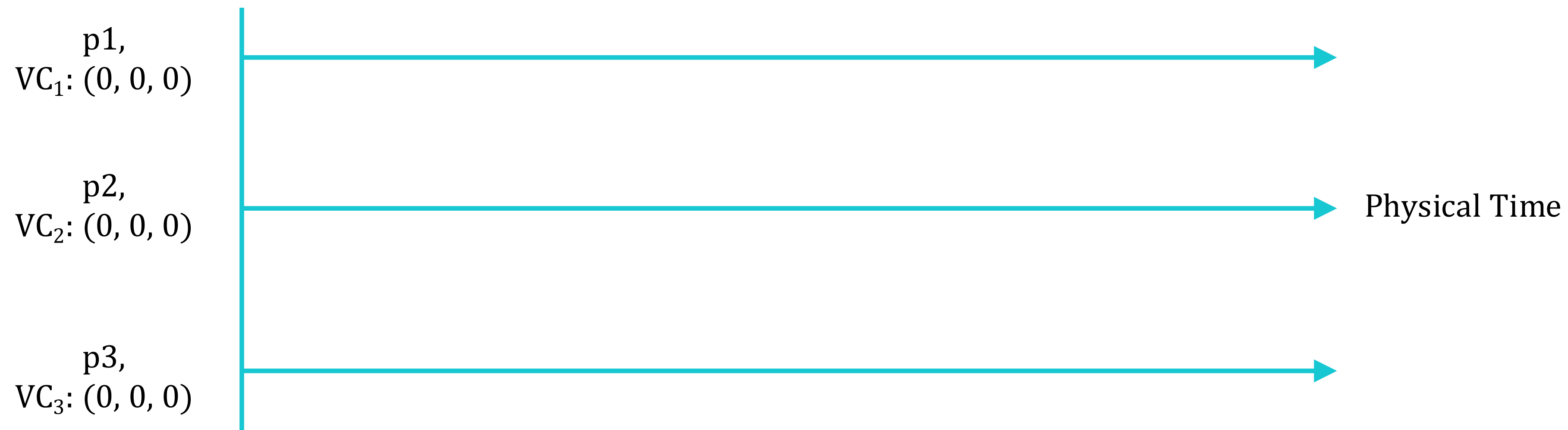
- MVCC (Multi-version concurrency control)
- Commonly used in DBMS

- Vector clock is an extension of MVCC
  - A vector clock is an array/vector of N logical clocks (N is the number of processes)
  - Each process has a vector clock
  - When processes communicate with each other, vector timestamps are piggybacked

- How are vector clocks maintained?

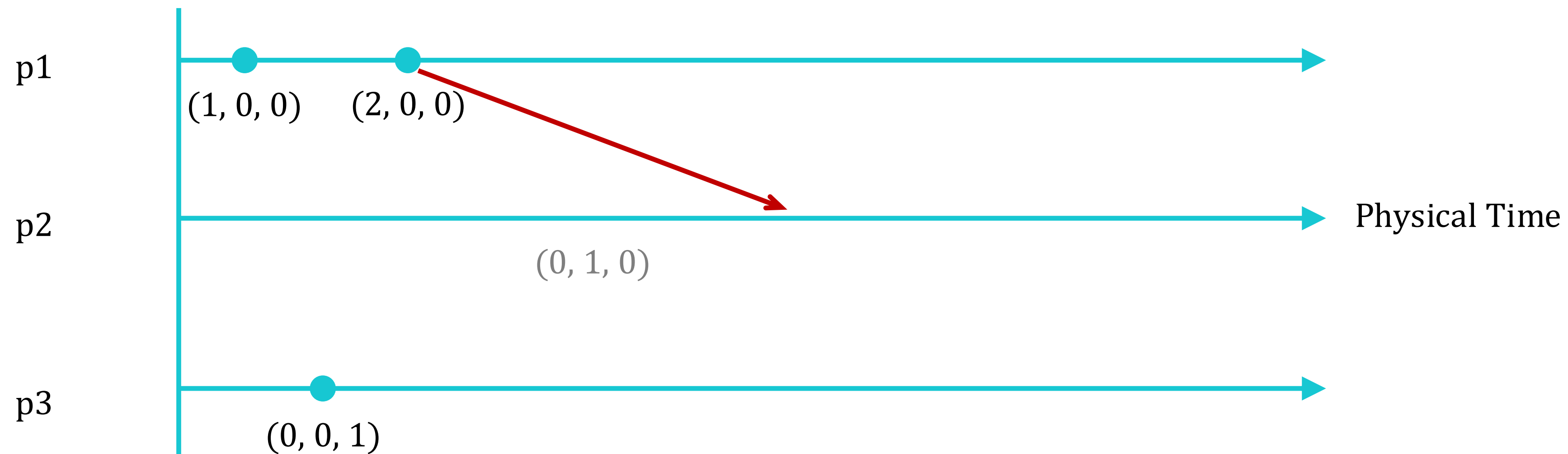
Vector clock maintenance

- Let VC_i denote the vector clock for process i (p_i):
- Initialize all clocks for all processes as zero
  - $VC_i[j] = 0$; for i, j = 1, 2, … N



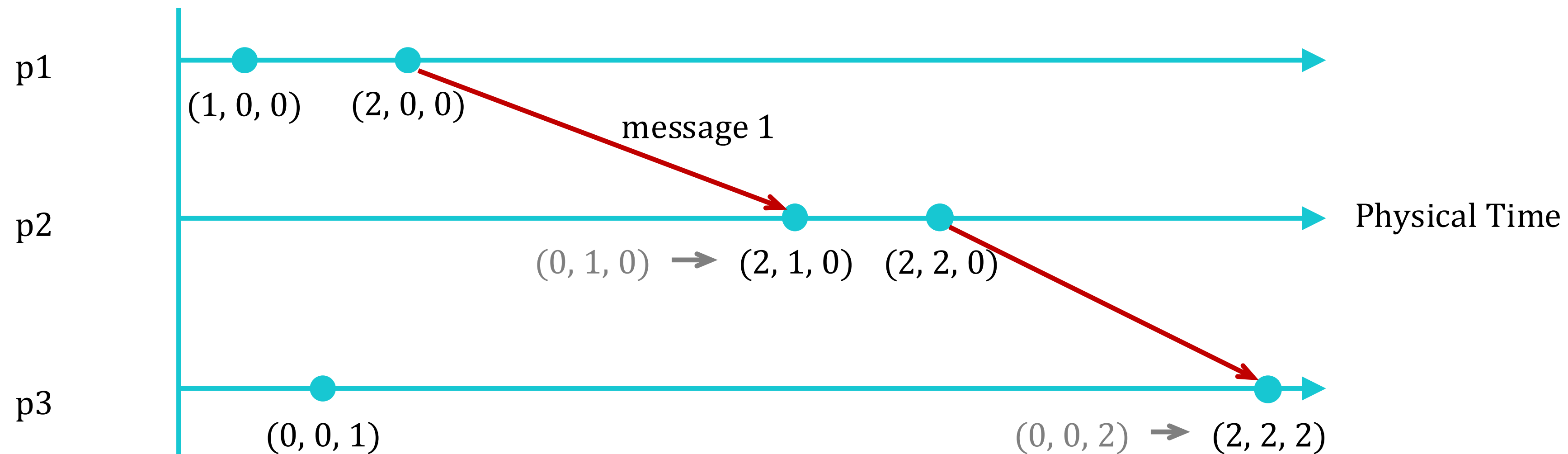
Vector clock maintenance

- Just before a process (p_i) timestamps an internal event (e.g., sending messages), its own logical clock in its vector will be incremented by one
  - $VC_i[i] = VC_i[i] + 1$



Vector clock maintenance

- The new vector of p_i is piggybacked when pi sends messages to other processes
- When a process (p_j) receives a message from another process (pi), it first increments its own logical clock by one, then compares its vector with the received one and takes the maximum value for each logical clock
  - $VC_j[i] = VC_j[i] + 1$
  - $VC_j[k] = \max (VC_j[k], VC_i[k])$; for $k = 1,2, \ldots N$



Vector clock maintenance

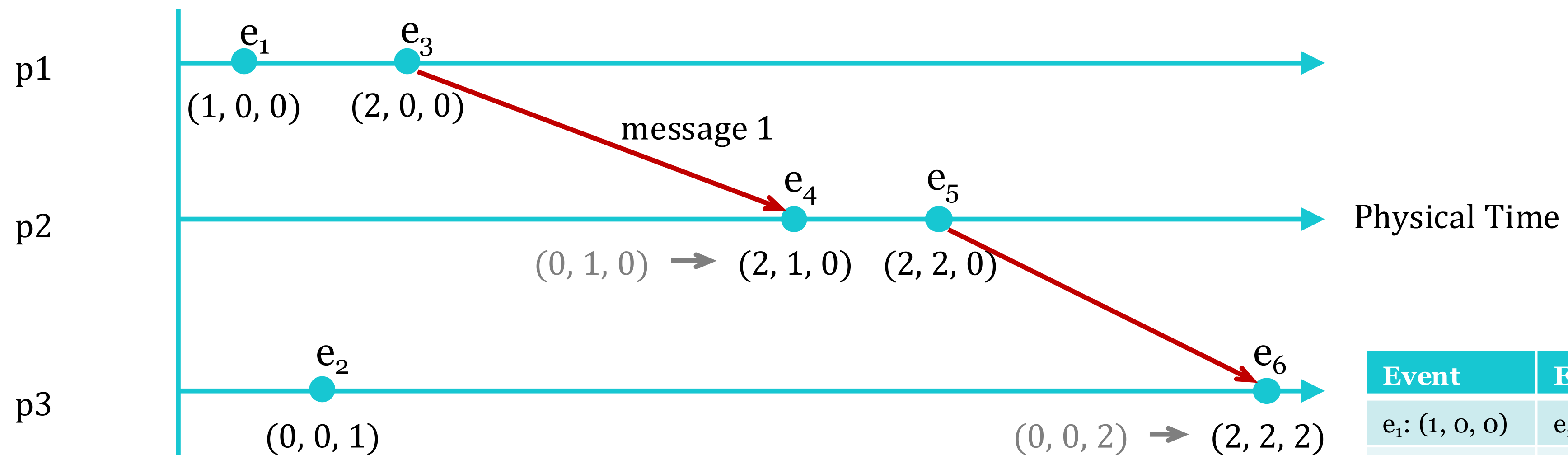
- Properties (comparing vector clocks for two events):
  - $VC = VC'$ iff. $VC[i] = VC'[i]$; for $i = 1, 2, \dots N$
  - $VC \leq VC'$ iff. $VC[i] \leq VC'[i]$; for $i = 1, 2, \dots N$
  - $VC < VC'$ iff. $VC[i] \leq VC'[i]$; for $i = 1, 2, \dots N$, meanwhile there exists a process p_j that, $VC[j] < VC'[j]$

- For two events e and e', e→e' iff. $VC(e) < VC(e')$
  - event e happens before event e'

- How to detect if two events are in conflict with each other?
  - For two events e and e', if neither $VC(e) \leq VC(e')$ nor $VC(e') \leq VC(e)$ satisfies, then the two events are concurrent

Vector clock maintenance

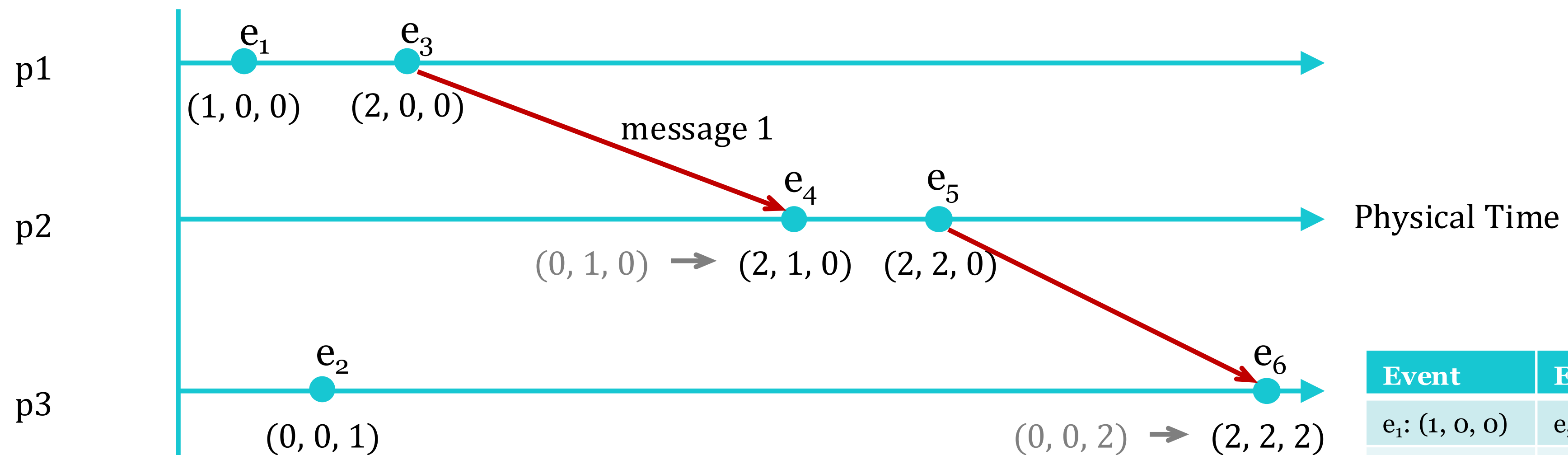
- How to detect if two events are in conflict with each other?
  - For two events e and e’, if neither VC(e) ≤ VC(e’) nor VC(e) ≥ VC(e’) satisfies, then the two events are concurrent



Event	Event	Conflict?
e_1: (1, 0, 0)	e_2: (0, 0, 1)	concurrent
e_2: (0, 0, 1)	e_4: (2, 1, 0)	concurrent
e_1: (1, 0, 0)	e_4: (2, 1, 0)	
e_4: (2, 1, 0)	e_6: (2, 2, 2)	

Vector clock maintenance

- How to detect if two events are in conflict with each other?
  - For two events e and e', if neither $VC(e) \leq VC(e')$ nor $VC(e) \geq VC(e')$ satisfies, then the two events are concurrent



Event	Event	Conflict?
e_1: (1, 0, 0)	e_2: (0, 0, 1)	concurrent
e_2: (0, 0, 1)	e_4: (2, 1, 0)	concurrent
e_1: (1, 0, 0)	e_4: (2, 1, 0)	
e_4: (2, 1, 0)	e_6: (2, 2, 2)	

Summary

- NoSQL systems support non-relational data models
  - Schema free
  - Support for semi-structured and unstructured data
  - Limited query capabilities (no joins!)

- NoSQL systems provide high (horizontal) scalability with high performance, availability, and fault tolerance
  - Achieved by:
    - Data partitioning
    - Data replication
    - Giving up consistency requirements

LINKÖPING
UNIVERSITY