

# TDDD43 Advanced Data Models and Databases

## NoSQL Databases

Huanyu Li  
[huanyu.li@liu.se](mailto:huanyu.li@liu.se)

Based on slides by Olaf Hartig

# 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

# Focuses of this lecture

- What are key characteristics of NoSQL systems?
- How do databases supported by NoSQL systems look?
- How can we use these databases or NoSQL-data models?
  - (in comparison to the databases supported by RDBMS)

# Outline

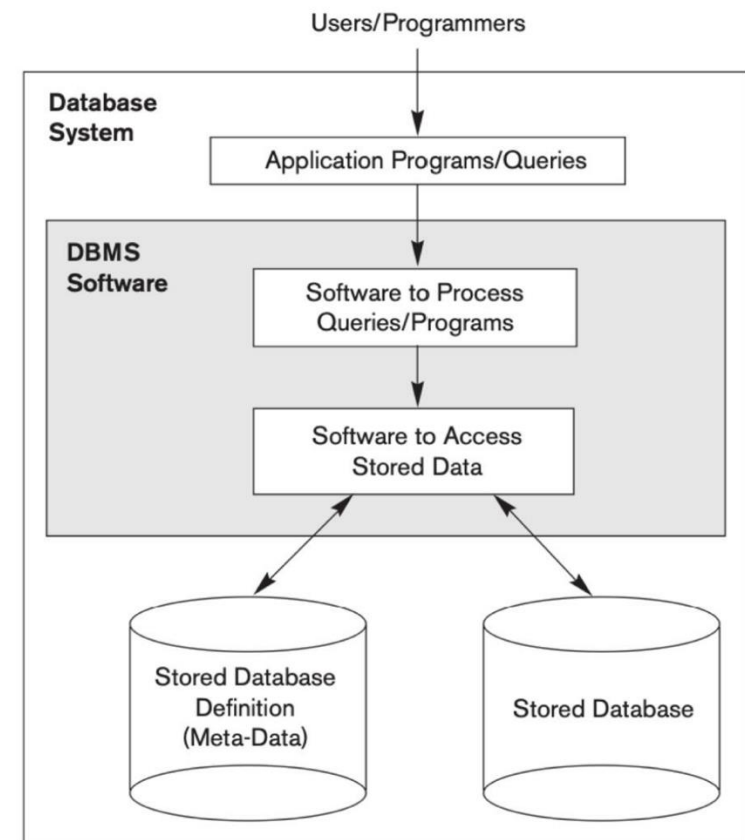
- Why NoSQL came onto the stage
- NoSQL data models
- The performance and scalability of NoSQL
- NoSQL consistency models and basic techniques

# Recap of Relational Database Systems

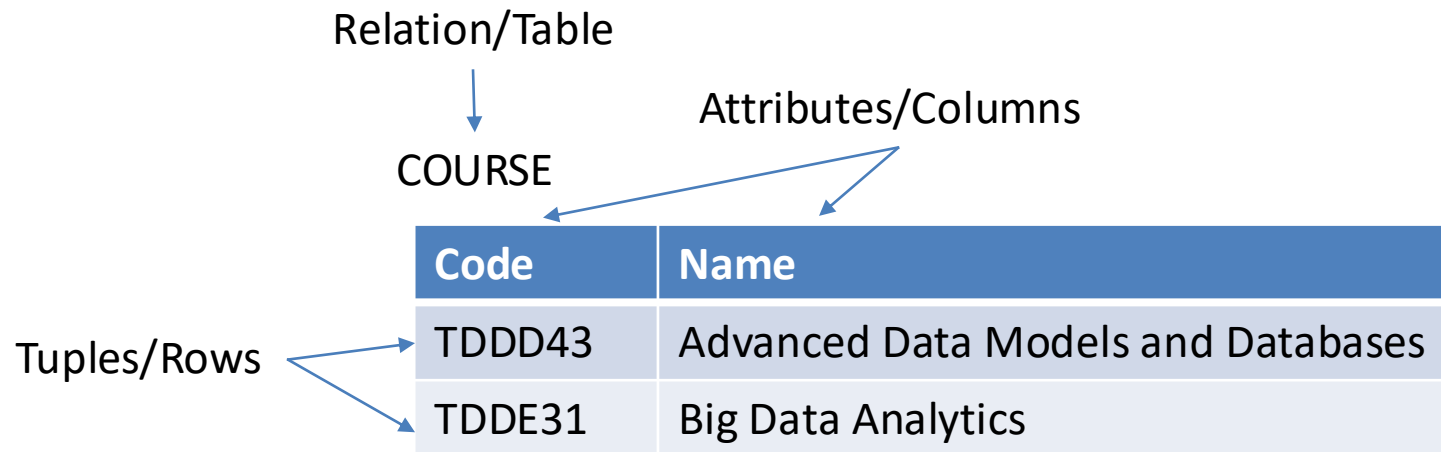
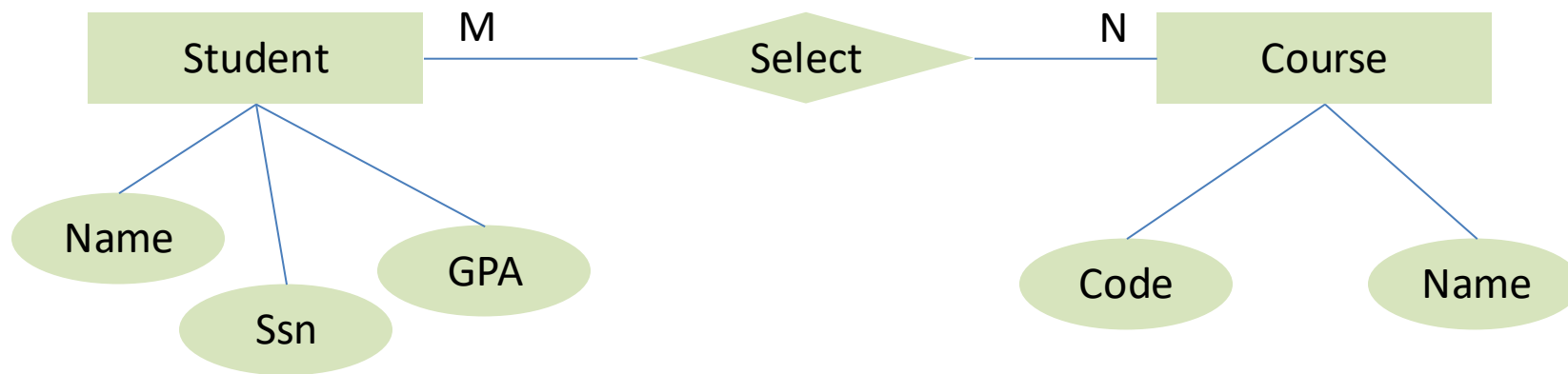
➤ Database System = Database + DBMS

➤ Relational DB

- With well-defined formal foundations
  - Schema level (Entity-Relationship)
  - Relational Model (relation/table, attribute/column, tuple/row)



# Recap of Relational Database Systems



# Recap of Relational Database Systems

- With well-defined formal foundations
- SQL – structured query language
  - query, data manipulation, database definition
- Support of transactions with ACID properties
  - **A**tomicity, **C**onsistency, **I**solation, **D**urability
- 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...



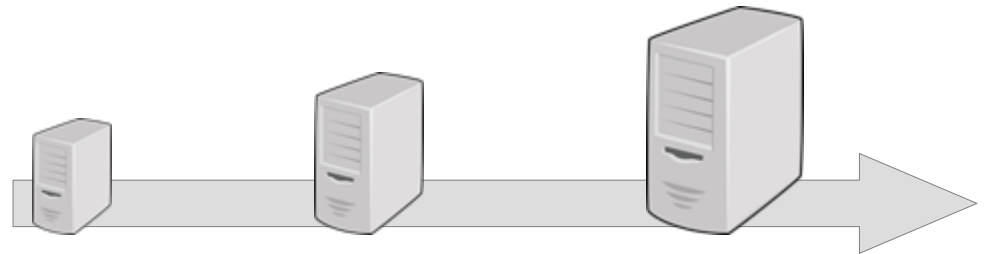
# Therefore, the scalability of a system is important

- 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 vs. Horizontal Scalability

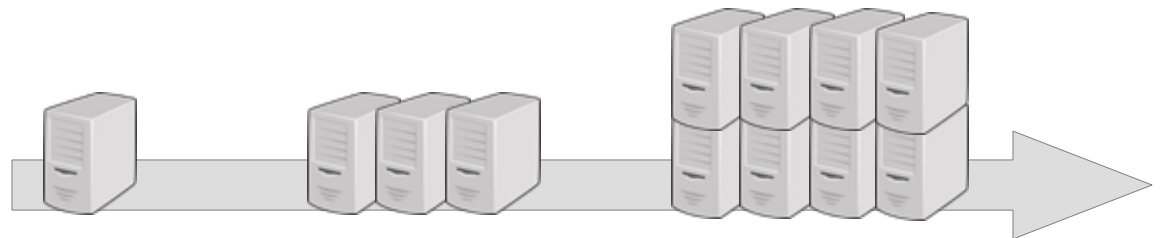
## ➤ Vertical scalability (scale up)

- 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



# To achieve much higher performance and 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

We will get back to transactional access of NoSQL systems later again!

# 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

# Outline

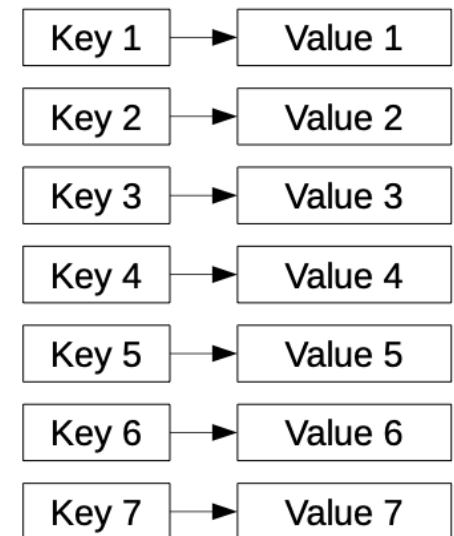
- ✓ Why NoSQL comes to the stage
- NoSQL data models
- BASE, ACID, and CAP
- NoSQL consistency models and basic techniques

# 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



# Example – Key-value stores

- Suppose we have a relational database consisting of a table:

User	<u>login</u>	name	website	twitter
	alice12	Alice	http://alice.name/	NULL
	bob_in_se	Bob	NULL	@TheBob
	charlie	Charlie	NULL	NULL

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



# Example – Key-value stores

- Suppose we have a relational database consisting of a table:

User	<u>login</u>	name	website	twitter
	alice12	Alice	http://alice.name/	NULL
	bob_in_se	Bob	NULL	@TheBob
	charlie	Charlie	NULL	NULL

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



# Example – Key-value stores

- Let's add another table

User	<u>login</u>	<u>name</u>	<u>website</u>	<u>twitter</u>
	alice12	Alice	http://alice.name/	NULL
	bob_in_se	Bob	NULL	@TheBob
	charlie	Charlie	NULL	NULL

Fav	<u>user</u>	<u>favorite</u>
	alice12	bob_in_se
	alice12	charlie

- How to update the key-value model?



# Example – Key-value stores

- Let's add another table

User	<u>login</u>	<u>name</u>	<u>website</u>	<u>twitter</u>
	alice12	Alice	http://alice.name/	NULL
	bob_in_se	Bob	NULL	@TheBob
	charlie	Charlie	NULL	NULL

Fav	<u>user</u>	<u>favorite</u>
	alice12	bob_in_se
	alice12	charlie

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

# Example – Key-value stores

- If we want to find all users for whom Bob is a favorite
- It is possible, but very inefficient (**an efficient way is to use 'Bob' as the key in queries**)

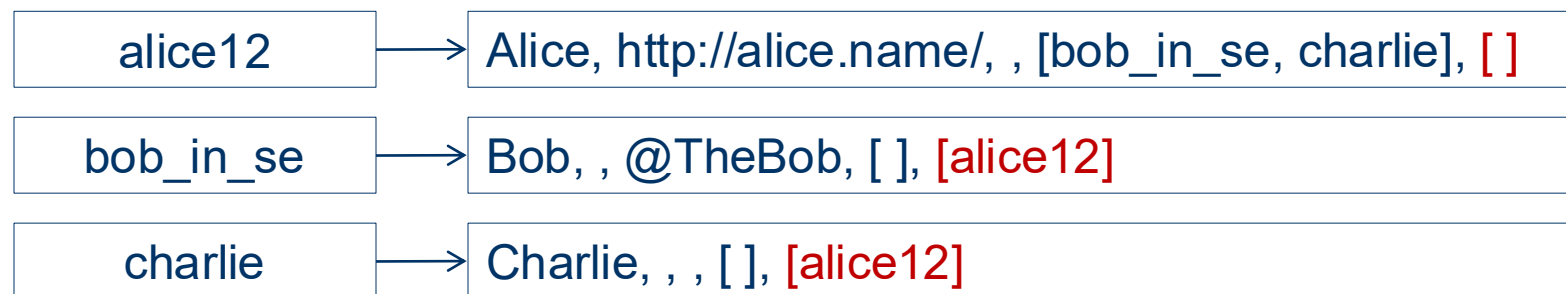
User	<u>login</u>	<u>name</u>	<u>website</u>	<u>twitter</u>
	alice12	Alice	http://alice.name/	NULL
	bob_in_se	Bob	NULL	@TheBob
	charlie	Charlie	NULL	NULL

Fav	<u>user</u>	<u>favorite</u>
	alice12	bob_in_se
	alice12	charlie



# Example – Key-value stores

- If we want to find all users for whom Bob is a favorite
- 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)



# Examples of Key-value stores

- Open-source examples
  - Redis
  - Memcached



# Key-Value Store Examples - 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)
  - Vector database



# Key-Value Store Examples - Memcached

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



# 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

login: "alice12"  
name: "Alice"  
website: "http://alice.name/"  
favorites: ["bob\_in\_se", "charlie"]

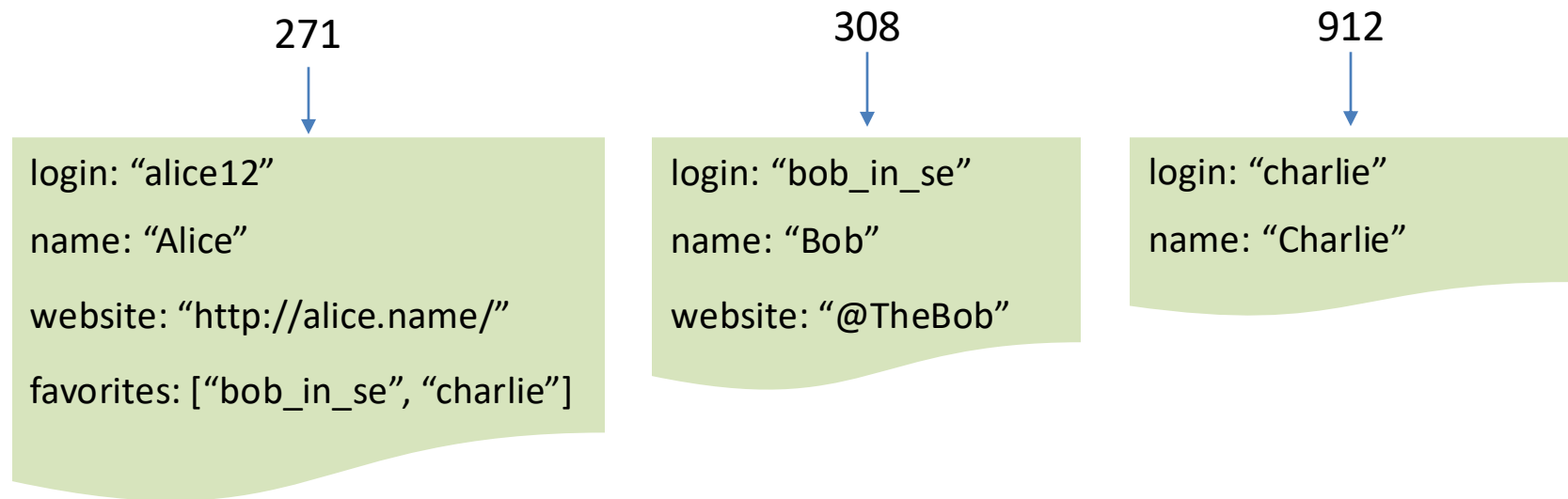
Fav	<u>user</u>	<u>favorite</u>
	alice12	bob_in_se
	alice12	charlie

User	<u>login</u>	<u>name</u>	<u>website</u>	twitter
	alice12	Alice	http://alice.name/	NULL
	bob_in_se	Bob	NULL	@TheBob
	charlie	Charlie	NULL	NULL

# NoSQL Data Models – Document stores

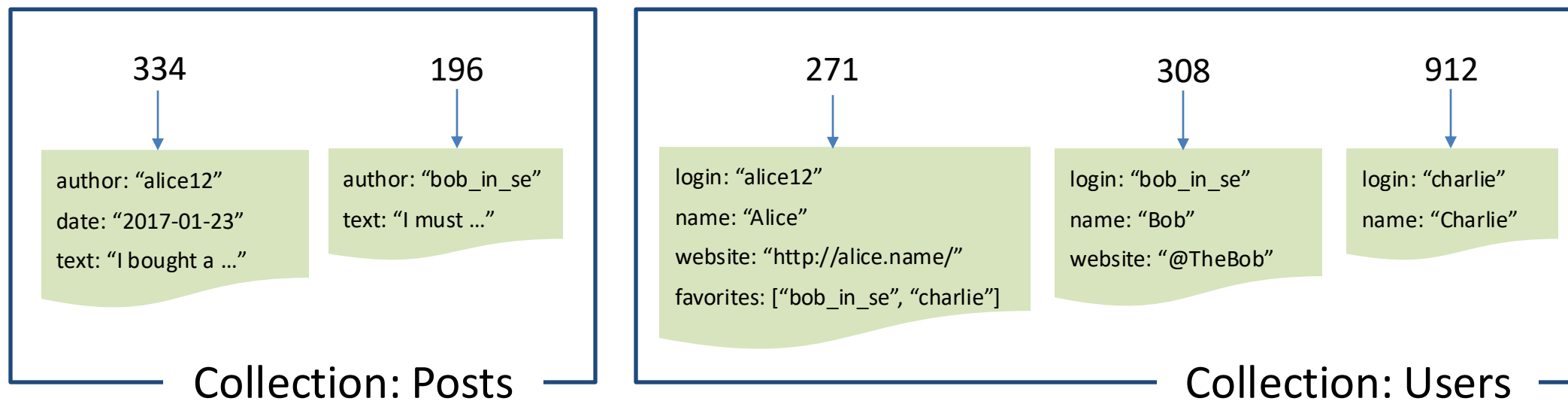
## ➤ Document store-based Databases

- 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

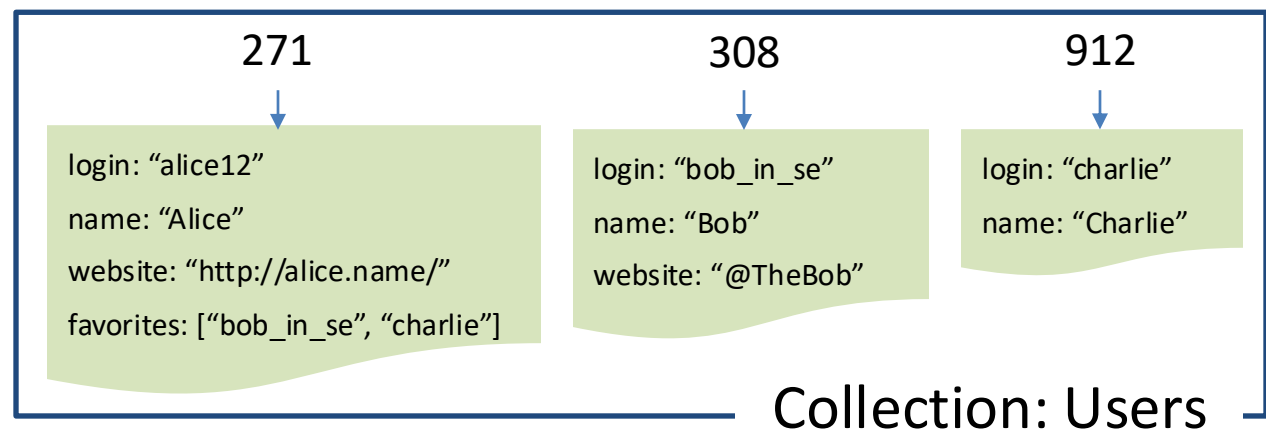
- Document store-based Databases
  - A set of documents (or multiple such sets)
  - Each document additionally associated with a unique identifier
  - Schema free: different documents may have different fields
  - Grouping of documents into separate sets (called “domains” or “collections”)



# 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 *Users* whose *name* field is "Alice"  
`db.Users.find({name: "Alice"})`
  - Find all docs in collection *Users* whose *age* is greater than 23  
`db.Users.find({age: {$gt: 23}})`
  - Find all docs in collection *Users* where Bob is listed as a favourite  
`db.Users.find({favorites: {$in: ["bob_in_se"]}})`

```
login: "alice12"
name: "Alice"
website: "http://alice.name/"
favorites: ["bob_in_se", "charlie"]
```

# Document stores: Querying

- Examples (based on MongoDB's query language)
  - Find all docs in collection *Users* whose *name* field is "Alice"  
`db.Users.find({name: "Alice"})`
  - Find all docs in collection *Users* whose *age* is greater than 23  
`db.Users.find({age: {$gt: 23}})`
  - Find all docs in collection *Users* where Bob is listed as a favorite  
`db.Users.find({favorites: {$in: ["bob_in_se"]}})`
- However, no cross-document queries (e.g., joins)
  - have to be implemented in the application logic

# NoSQL Data Models – Document store

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



# Data Models for NoSQL

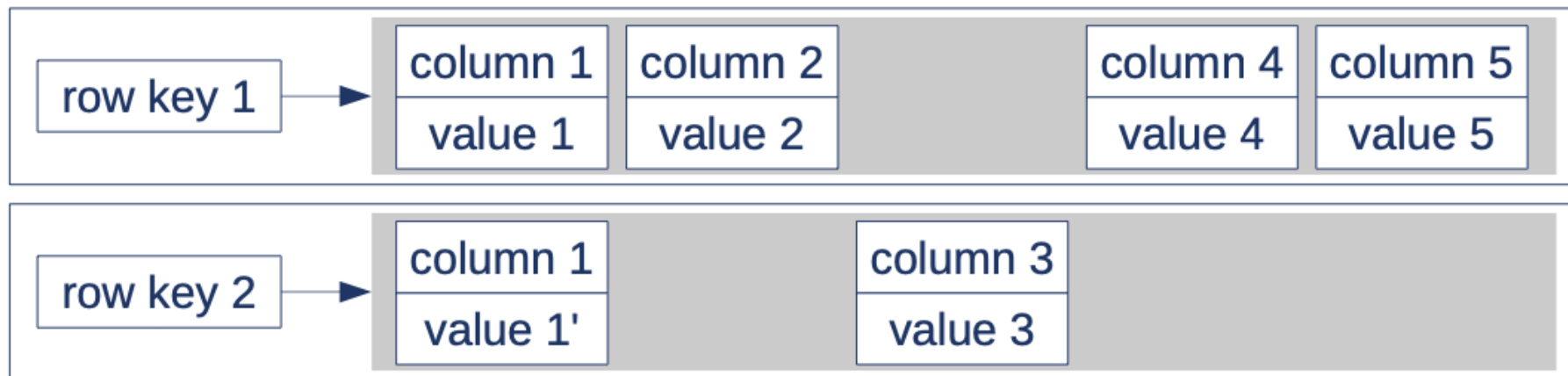
- ✓ Key-value model
- ✓ Document model
- Wide-column models
- Graph database models

# NoSQL Data Models – Wide-column store

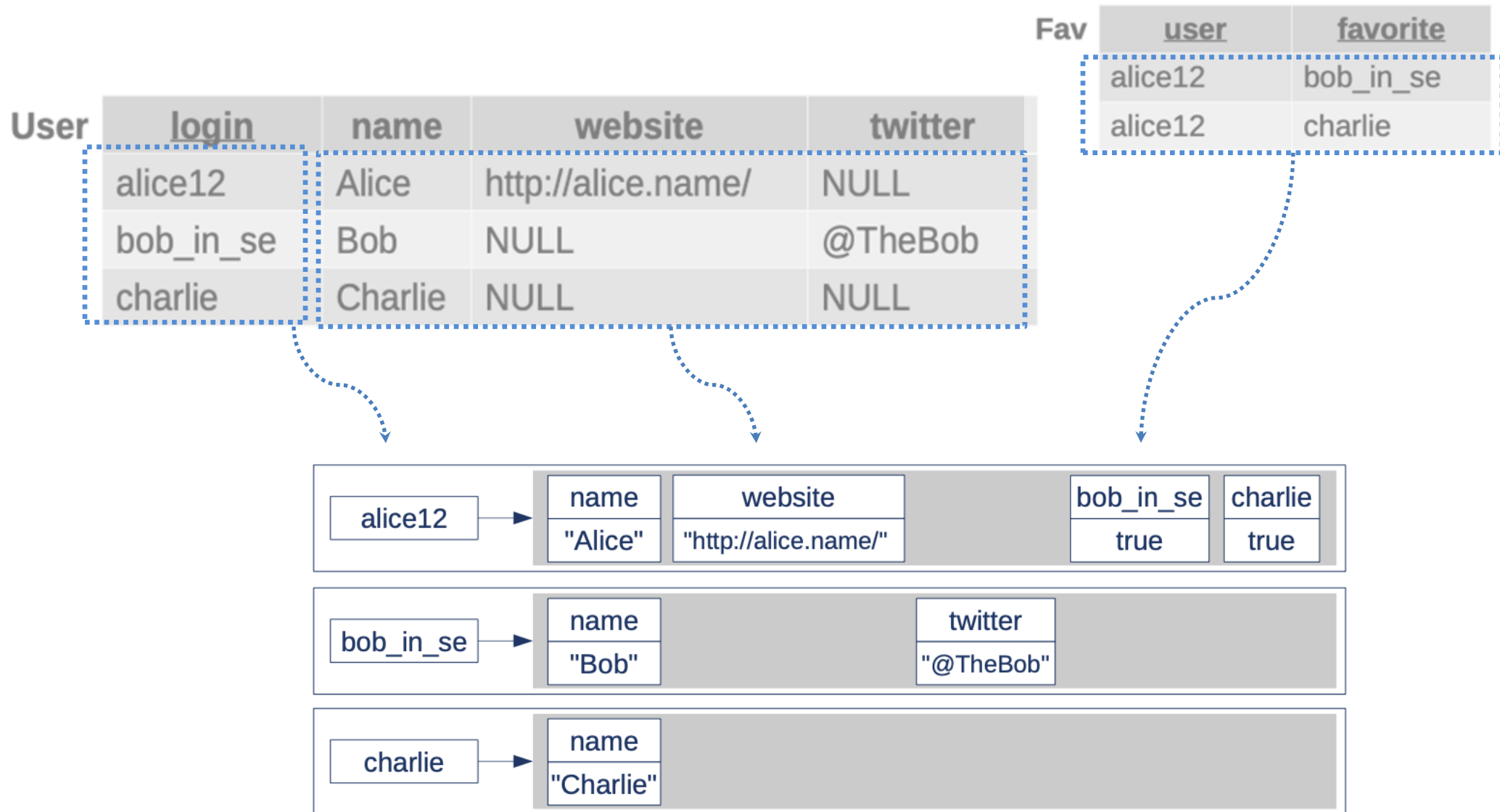
- Also called column-family or extensible-record
- Store data in rows, each row has a unique key and column families
- Schema-free
  - Keys are unique
  - Values are varying column families
  - Columns consist of key value pairs

# NoSQL Data Models – Wide column store

- Like a single, very wide relation (table) but extensible, schema-free, potentially sparse
- Like the document model without nesting

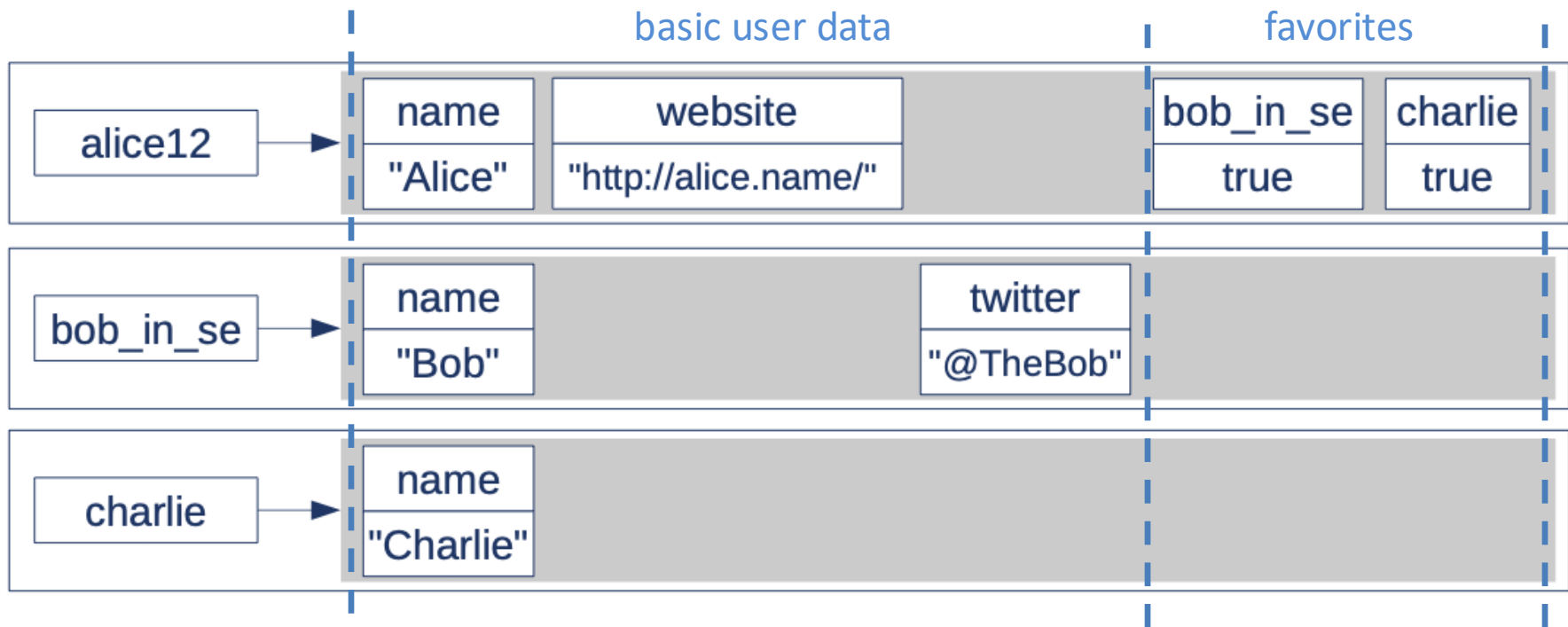


# NoSQL Data Models – Wide column store



# NoSQL Data Models – Wide column store

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





# NoSQL Data Models – Wide column store

- 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, have to be implemented in the application logic
- Better than key value stores for querying and indexing
- Not suitable for
  - Representing structures and relations

# NoSQL Data Models – Wide Column store

- Examples
  - Google BigTable
  - Apache HBase
  - Apache Cassandra






APACHE  
**HBASE**



# Data Models for NoSQL

- ✓ Key-value model
- ✓ Document model
- ✓ Wide-column models
- Graph database models
  - Next lecture on October 7th

# Data Models for NoSQL

- ✓ Key-value model
- ✓ Document model
- ✓ Wide-column models
- Graph database models
- There are NoSQL systems that are based on multi models
  - OrientDB (key-value, documents, graph) 
  - ArangoDB (key-value, documents, graph) 
  - Cosmos DB (key-value, documents, wide-column, graph) 

# Outline

- ✓ Why NoSQL comes to the stage
- ✓ NoSQL data models
- NoSQL consistency models and basic techniques
  - ACID, BASE and CAP
  - Consistent Hashing
  - Vector clock

# 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
    - 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 occur even if the system failure occurs



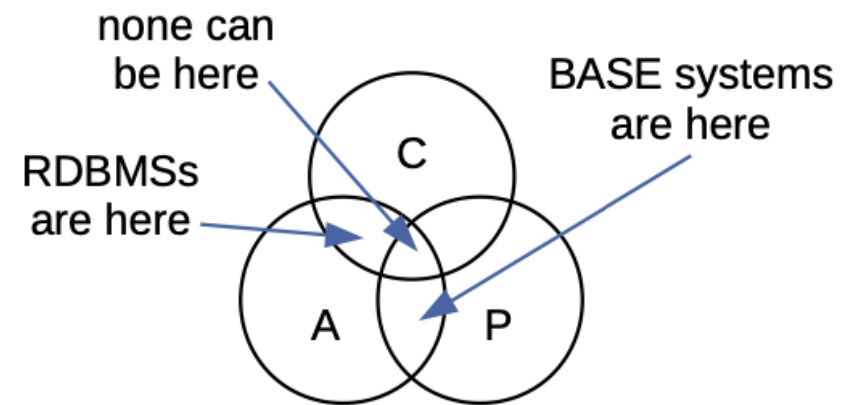
# NoSQL, BASE rather than ACID

- Giving up ACID guarantees, to achieve much higher performance and scalability
  - **B**asically **A**vailable
    - System available whenever accessed, even if parts of it are unavailable
  - **S**oft state
    - distributed data does not need to remain consistent at all times
  - **E**ventually 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



# Outline

- ✓ Why NoSQL comes to the stage
- ✓ NoSQL data models
- NoSQL consistency models and basic techniques

# Consistency models

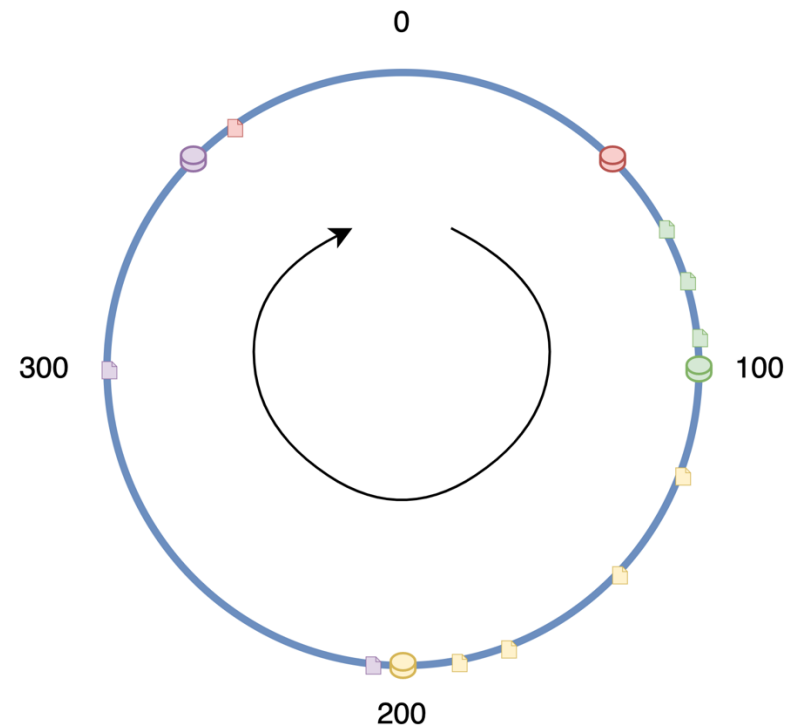
- Strong consistency
  - After the update completes, any subsequent access will return the updated value
- Weak consistency
  - The system does not guarantee that subsequent accesses to the system will return the updated value
  - *Inconsistency window*: period until all replicas have been updated in a lazy manner
  - *Eventual consistency*: if no new updates are made, eventually all accesses will return the last updated value
    - Employed by many NoSQL databases

# 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)

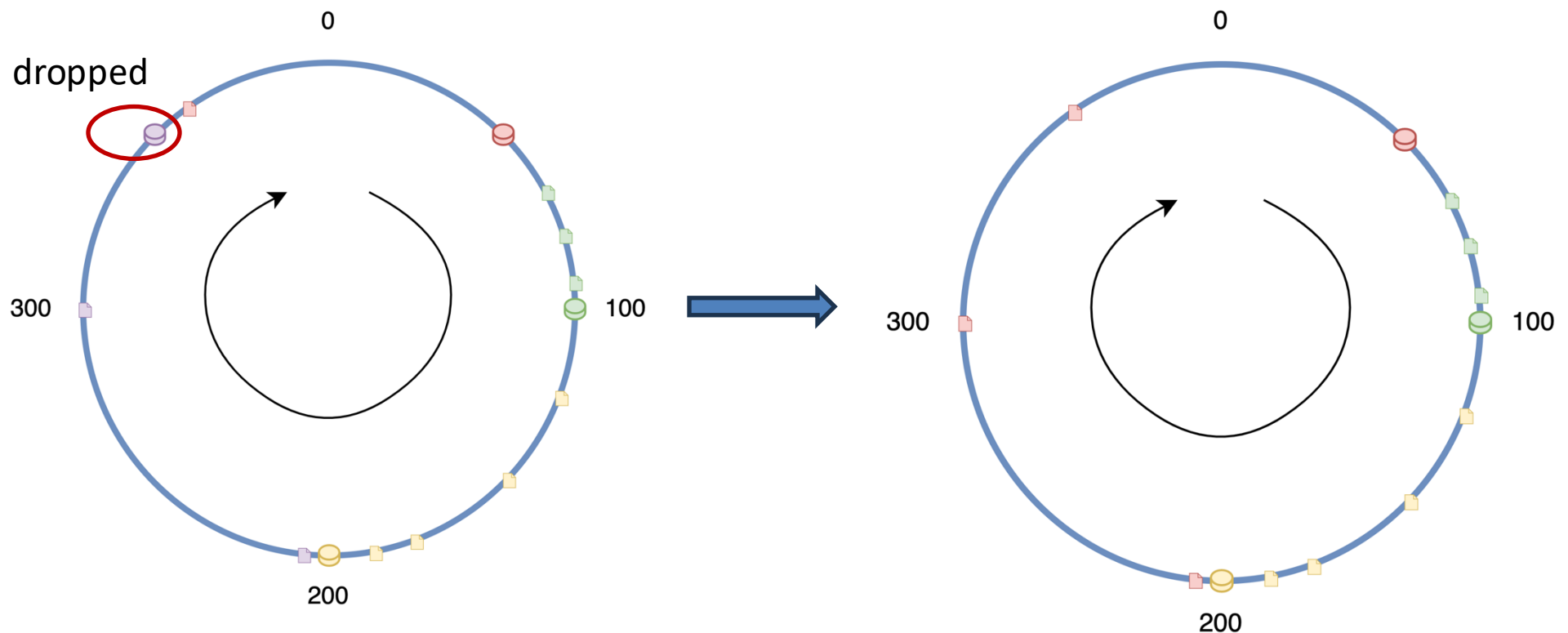
# NoSQL Consistent hashing

- 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



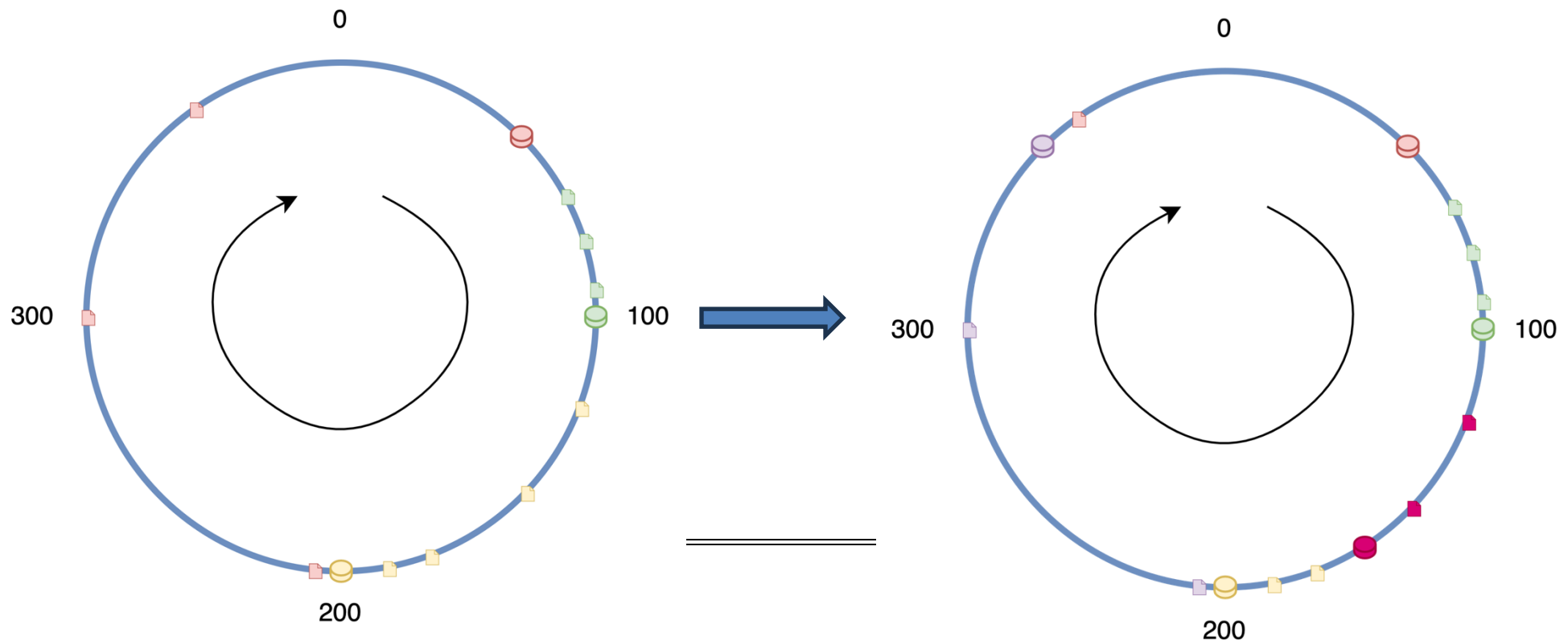
# Consistent hashing – Node Removal

- 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 table
  - the hash realm is repartitioned, and hash data will be transferred to new neighbor
  - No need to update remaining nodes





# Vector clock

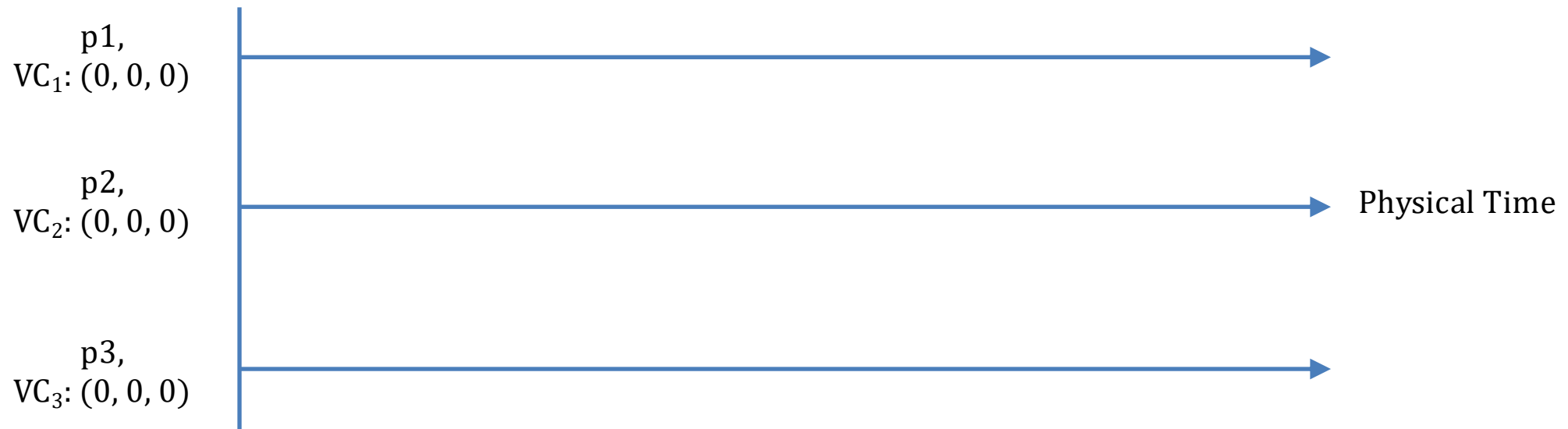
- MVCC (Multi-version concurrency control)
- Commonly used in DBMS
- Vector clock is an extension of MVCC
  - A vector clock is an array/vector of  $N$  logical clocks ( $N$  is the number of processes)
  - Each process has a vector clock
  - When processes communicate to each other, vector timestamps 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  $i, j = 1, 2, \dots N$
- 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
  - $VC_i[i] = VC_i[i] + 1$
- The new vector of  $p_i$  is piggybacked when  $p_i$  sends messages to other processes
- When a process ( $p_j$ ) receives a message from another process ( $p_i$ ), it first increments its own logical clock by one, then compare 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  $k = 1, 2, \dots N$

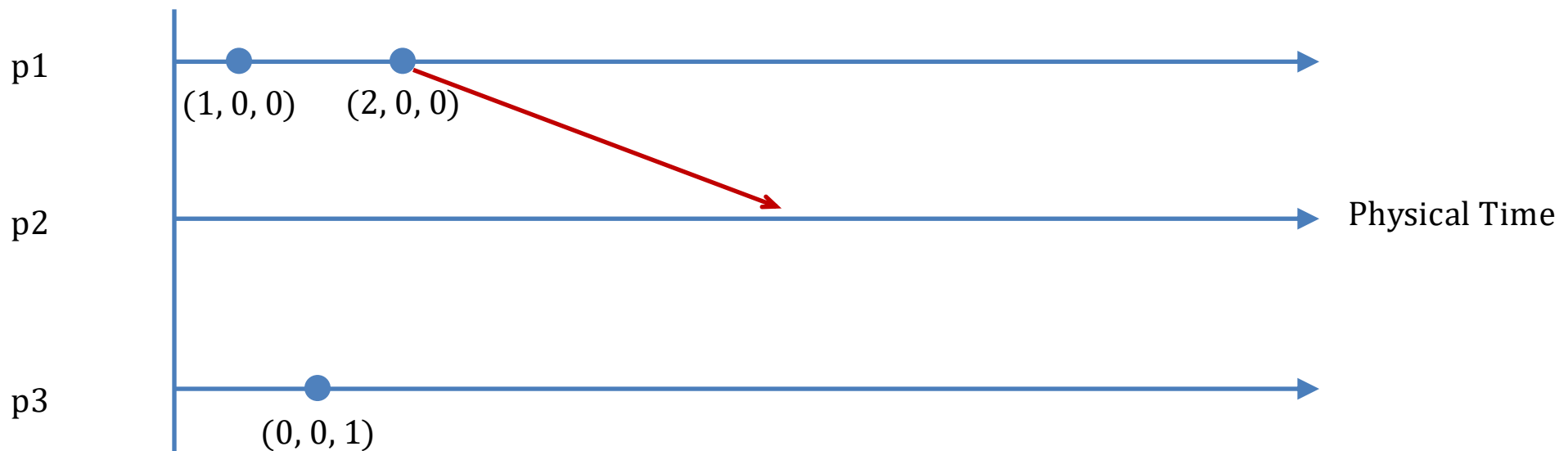
# Vector clock maintenance

- 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  $i, j = 1, 2, \dots, N$



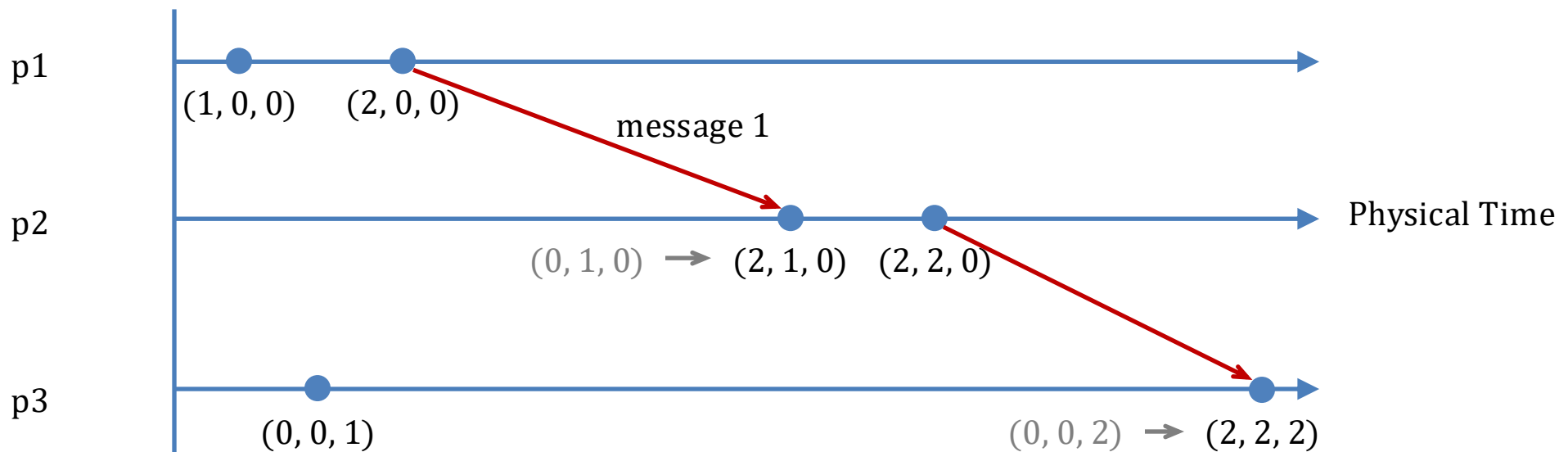
# Vector clock Maintenance

- 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  $p_i$  sends messages to other processes
- When a process ( $p_j$ ) receives a message from another process ( $p_i$ ), it first increments its own logical clock by one, then compare its vector with the received one and takes the maximum values for each logical clock
  - $VC_j[i] = VC_j[i] + 1$
  - $VC_j[k] = \max(VC_j[k], VC_i[k]); \text{ for } k = 1, 2, \dots, N$

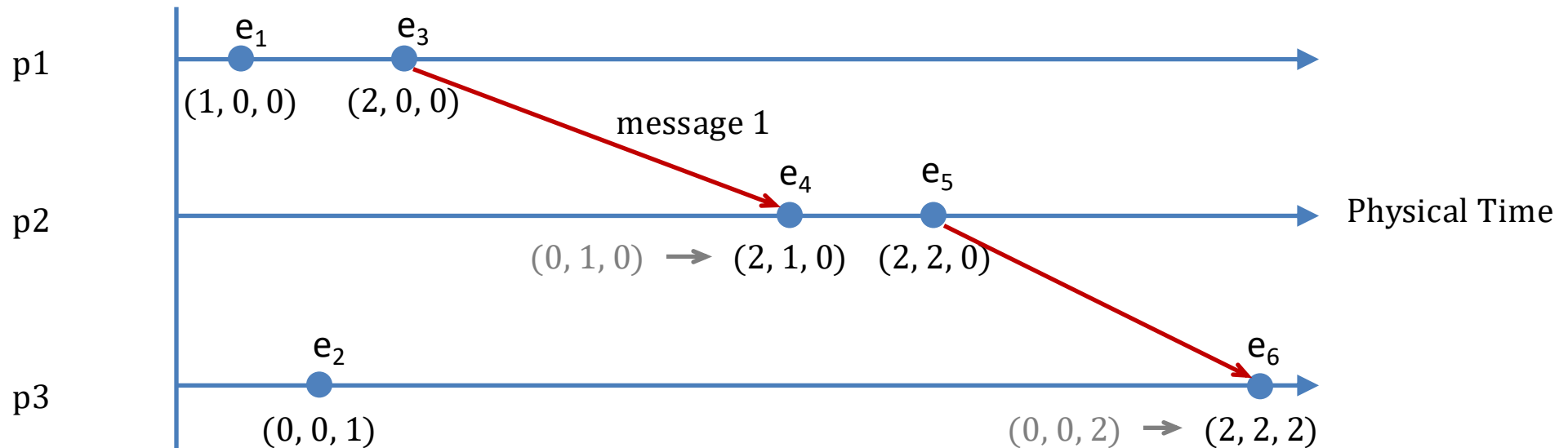


# Vector clock Maintenance

- 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  $e'$ ,  $e \rightarrow e'$  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

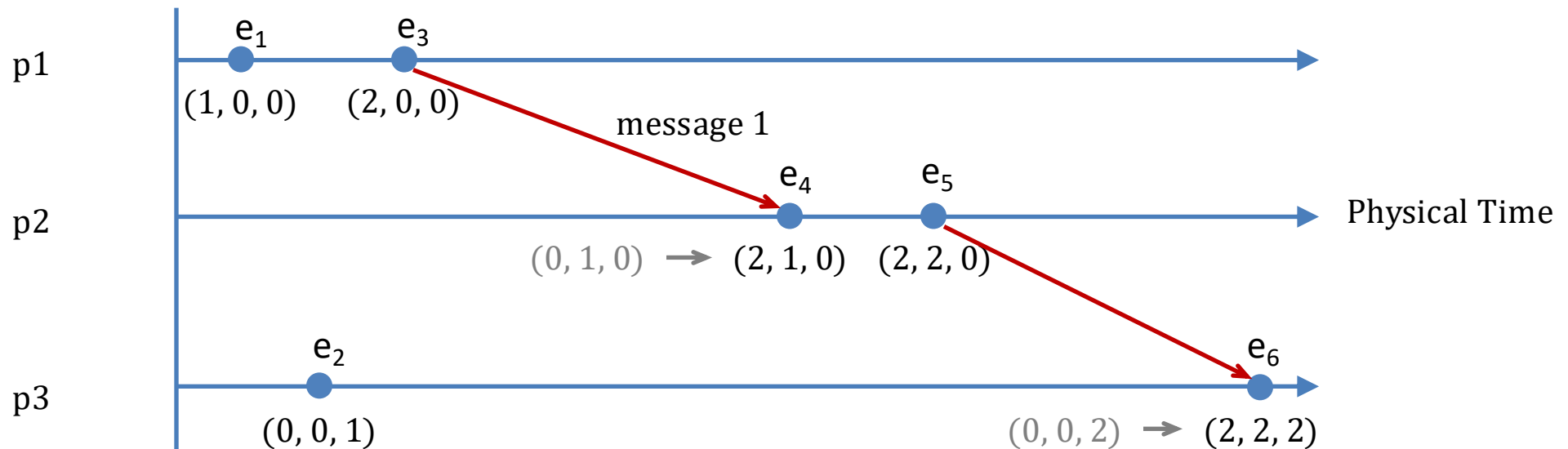
- How to detect if two events are in conflict with each other?
  - For two events  $e$  and  $e'$ , if neither  $VC(e) \leq VC(e')$  nor  $VC(e) \geq VC(e')$  satisfies, then the two events are concurrent



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

[www.liu.se](http://www.liu.se)