

# Relational Database Management Systems

- Well-defined formal foundations (*relational data model*)

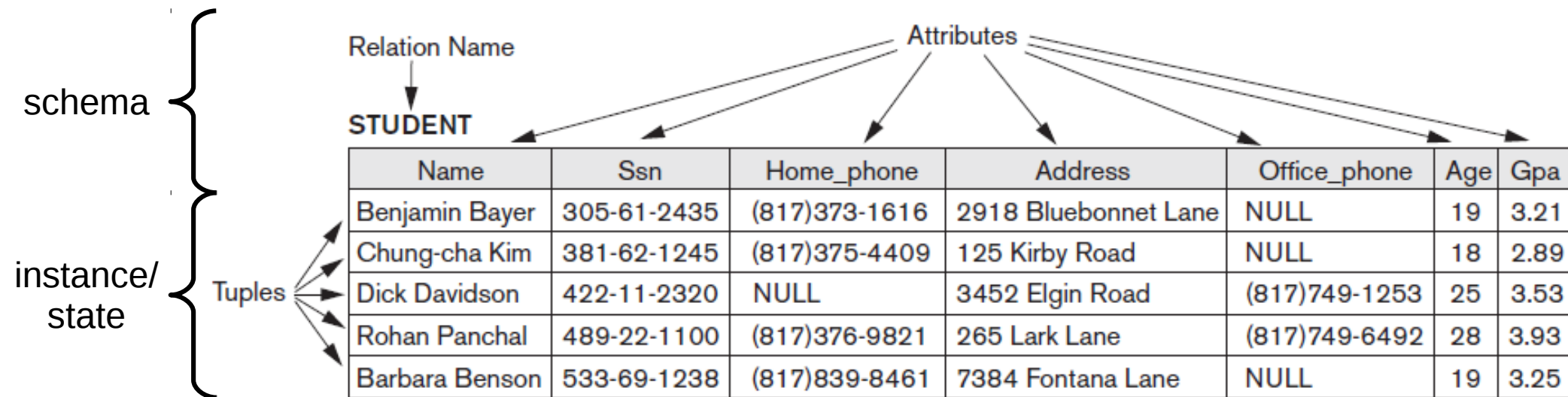


Figure from "Fundamentals of Database Systems" by Elmasri and Navathe, Addison Wesley.

# Relational Database Management Systems

- Well-defined formal foundations (*relational data model*)
- *SQL* – powerful declarative language
  - querying
  - data manipulation
  - database definition
- Support of transactions with *ACID* properties (**A**tomicity, **C**onsistency preservation, **I**solation, **D**urability)
- Established technology (developed since the 1970s)
  - many vendors
  - highly mature systems
  - experienced users and administrators

# Business world has evolved

- Organizations and companies (whole industries) shift to the digital economy powered by the Internet
- Central aspect: new IT applications that allow companies to *run their business* and to *interact with costumers*
  - Web applications
  - Mobile applications
  - Connected devices (“Internet of Things”)



Image source: <https://pixabay.com/en/technology-information-digital-2082642/>

# New Challenges for Database Systems

- Increasing numbers of concurrent users/clients
  - tens of thousands, perhaps millions
  - globally distributed
  - expectations: consistently high performance and 24/7 availability (no downtime)
- Different types of data
  - huge amounts (generated by users and devices)
  - data from different sources together
  - frequent schema changes or no schema at all
  - semi-structured and unstructured data
- Usage may change rapidly and unpredictably



Image source: <https://www.flickr.com/photos/groucho/5523369279/>

# ``NoSQL''

- Some interpretations (without precise definition):
  - “no to SQL”
  - “not only SQL”
  - “not relational”
- 1998: first used for an RDBMS\* that omitted usage of SQL
- 2009: picked up again to name a conference on  
“open-source, distributed, non-relational databases”
- Since then, “NoSQL database” loosely specifies a class of non-relational DBMSs
  - Relax some requirements of RDBMSs to gain efficiency and scalability for use cases in which RDBMSs are a bad fit

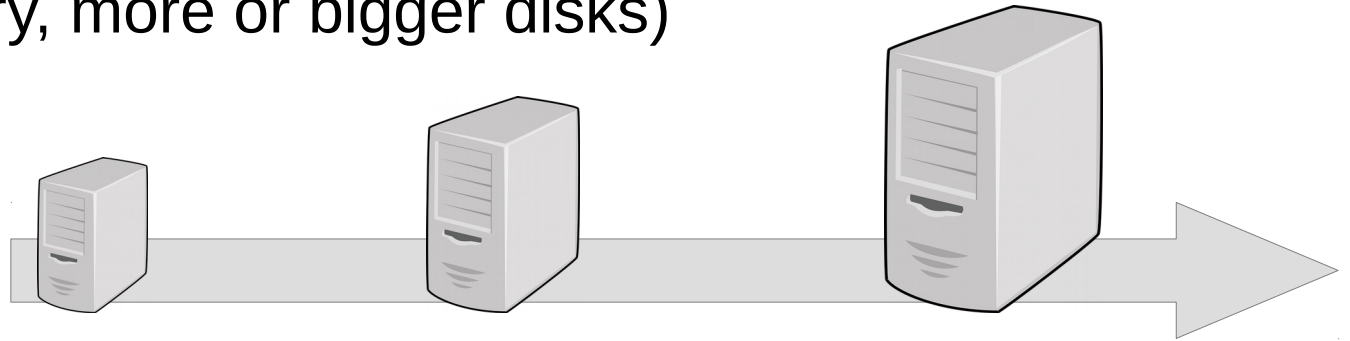
\*RDBMS = relational database management system

# Scalability

- Data scalability: system can handle *growing amounts of data* without losing performance
- Read scalability: system can handle *increasing numbers of read operations* without losing performance
- Write scalability: system can 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

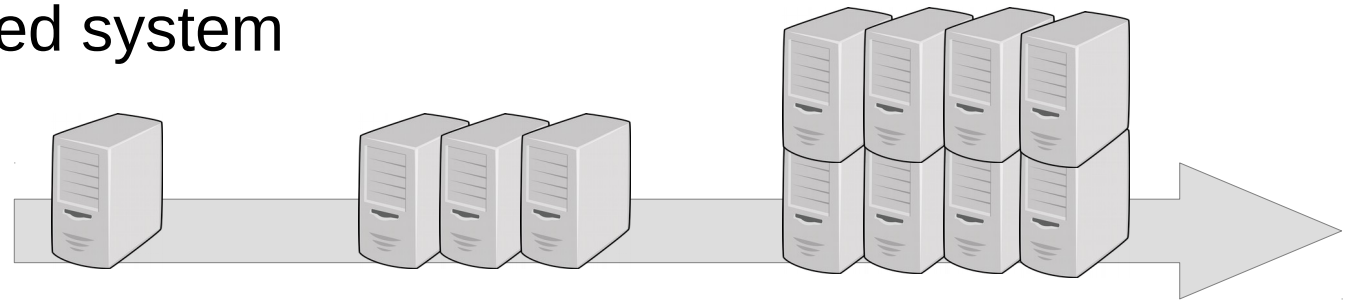


Image source: <https://pixabay.com/en/server-web-network-computer-567943/>

# Typical\* Characteristics of NoSQL Systems

- Ability to scale horizontally over many commodity servers with high performance, availability, and fault tolerance
  - achieved by giving up ACID guarantees
  - and by partitioning and replication of data
- Non-relational data model, no requirements for schemas
  - data model limitations make partitioning effective

\*Attention, there is a *broad variety* of such systems and not all of them have these characteristics to the same degree



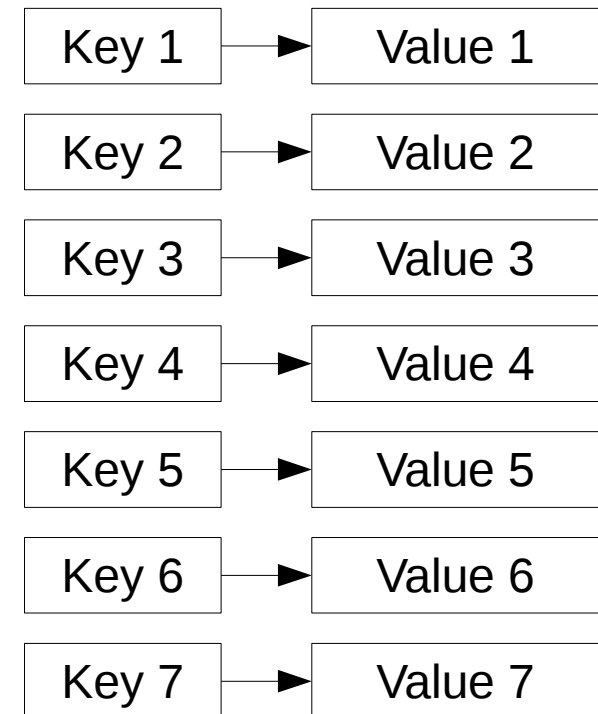
# NoSQL Data Models

# Data Models

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

# Key-Value Stores: Data Model

- Database is simply a set of key-value pairs
  - keys are unique
  - values of arbitrary data types
- Values are opaque to the system



# Example

- Assume a relational database consisting of a single 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 can we capture this data in the key-value model?

# Example

- Assume a relational database consisting of a single 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 can we capture this data in the key-value model?



# Example

- Let's add another table:

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

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

- How can we capture this data in the key-value model?



# Example

- Let's add another table:

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

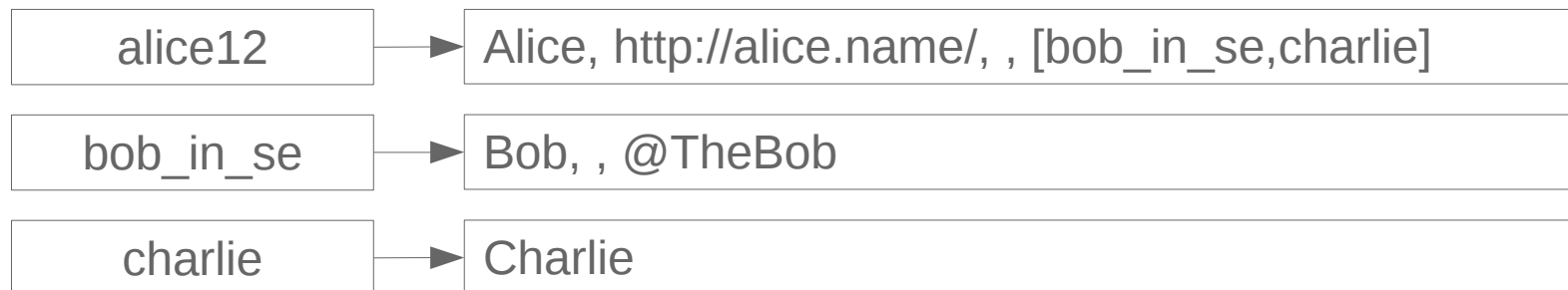
User	<u>login</u>	name	website	twitter
	alice12	Alice	http://alice.name/	NULL
	bob_in_se	Bob	NULL	@TheBob
	charlie	Charlie	NULL	NULL

- How can we capture this data in the key-value model?



# Key-Value Stores: Querying

- Only CRUD operations in terms of keys
  - CRUD: create, read, update, delete
  - `put(key, value)`; `get(key)`; `delete(key)`
- No support for value-related queries
  - Recall that values are opaque to the system (i.e., no secondary index over values)
- Accessing multiple items requires *separate requests*
  - Beware: often no transactional capabilities





# Key-Value Stores: Querying

- Only CRUD operations in terms of keys
  - CRUD: create, read, update, delete
  - `put(key, value); get(key); delete(key)`
- No support for value-related queries
  - Recall that values are opaque to the system (i.e., no secondary index over values)
- Accessing multiple items requires *separate requests*
  - Beware: often no transactional capabilities
- Advantage of these limitations: partition the data based on keys (“*horizontal partitioning*”, also called “*sharding*”) and distributed processing can be very efficient

# Example (cont'd)

- Assume we try to find all users for whom Bob is a favorite
- It is possible (how?), but very inefficient
- What can we do to make it more efficient?



# Example (cont'd)

- Assume we try to find all users for whom Bob is a favorite
- It is possible (how?), 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 Stores: Use Cases

- Whenever values need to be accessed only via keys
- Examples:
  - Storing Web session information
  - User profiles and configuration
  - Shopping cart data
  - Caching layer that stores results of expensive operations (e.g., complex queries over an underlying database, user-tailored Web pages)

# Examples of Key-Value Stores

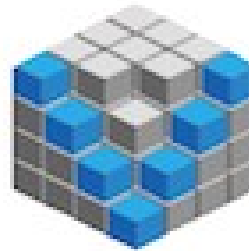
- In-memory key-value stores

- Memcached
- Redis



- Persistent key-value stores

- Berkeley DB
- Voldemort
- RiakDB



# Data Models

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



Image source: <https://pxhere.com/en/photo/1188160>

# Document Stores: Data Model

- Document: a set of fields consisting of a name and a value
  - field names are unique within the document
  - values are scalars (text, numeric, boolean) or lists

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

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

# Document Stores: Data Model

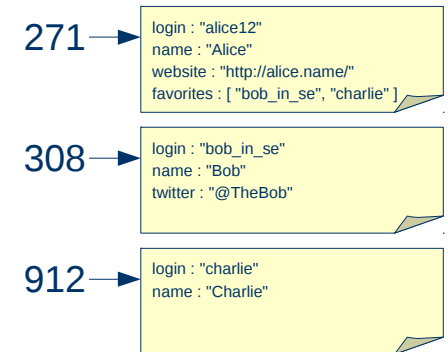
- Document: a set of fields consisting of a name and a value
  - field names are unique within the document
  - values are scalars (text, numeric, boolean) or lists
  - in some systems, values may also be other documents

```
login : "alice12"  
name : "Alice"  
website : "http://alice.name/"  
favorites : [ "bob_in_se", "charlie" ]  
address : {  
    street : "Main St"  
    city : "Springfield"  
}
```



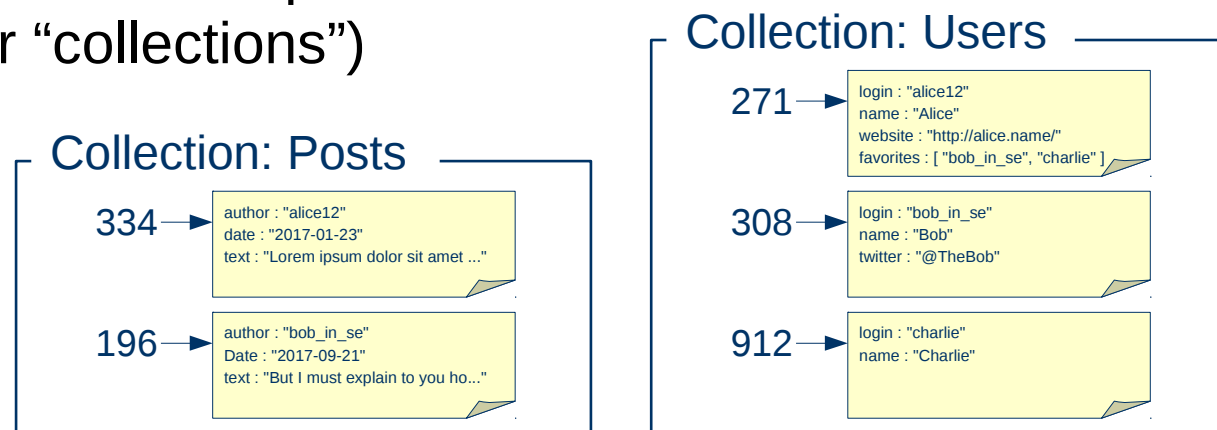
# Document Stores: Data Model

- Document: a set of fields consisting of a name and a value
  - field names are unique within the document
  - values are scalars (text, numeric, boolean) or lists
  - in some systems, values may also be other documents
- Database is a set of documents (or multiple such sets)
  - each document additionally associated with a unique identifier (typically system-generated)
  - *schema free*: different documents may have different fields



# Document Stores: Data Model

- Document: a set of fields consisting of a name and a value
  - field names are unique within the document
  - values are scalars (text, numeric, boolean) or lists
  - in some systems, values may also be other documents
- Database is a set of documents (or multiple such sets)
  - each document additionally associated with a unique identifier (typically system-generated)
  - *schema free*: different documents may have different fields
  - grouping of documents into separate sets (called “domains” or “collections”)



# Document Stores: Data Model

- Document: a set of fields consisting of a name and a value
  - field names are unique within the document
  - values are scalars (text, numeric, boolean) or lists
  - in some systems, values may also be other documents
- Database is a set of documents (or multiple such sets)
  - each document additionally associated with a unique identifier (typically system-generated)
  - *schema free*: different documents may have different fields
  - grouping of documents into separate sets (called “domains” or “collections”)
- Partitioning based on collections and/or on 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
- Queries expressed in terms of program code using an API or in a system-specific query language

# Document Stores: Querying

- Querying in terms of conditions on document content
- Queries expressed in terms of program code using an API or in a system-specific query language
- Examples (using 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: {$gt: 23}} )`
  - Find all docs about *Users* who favorite Bob  
`db.Users.find( {favorites: {$in: ["bob_in_se"]}} )`

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

# Document Stores: Querying

- Querying in terms of conditions on document content
- Queries expressed in terms of program code using an API or in a system-specific query language
- Examples (using 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: {$gt: 23}} )`
  - Find all docs about *Users* who favorite Bob  
`db.Users.find( {favorites: {$in: ["bob_in_se"]}} )`
- However, no cross-document queries (like joins)
  - have to be implemented in the application logic

# Document Stores: Use Cases

- Whenever we have items of similar nature but slightly different structure
- Examples:
  - Blogging platforms
  - Content management systems
  - Event logging
- Fast application development

# Examples of Document Stores

- Amazon's SimpleDB



- CouchDB



- Couchbase



- MongoDB





# Data Models

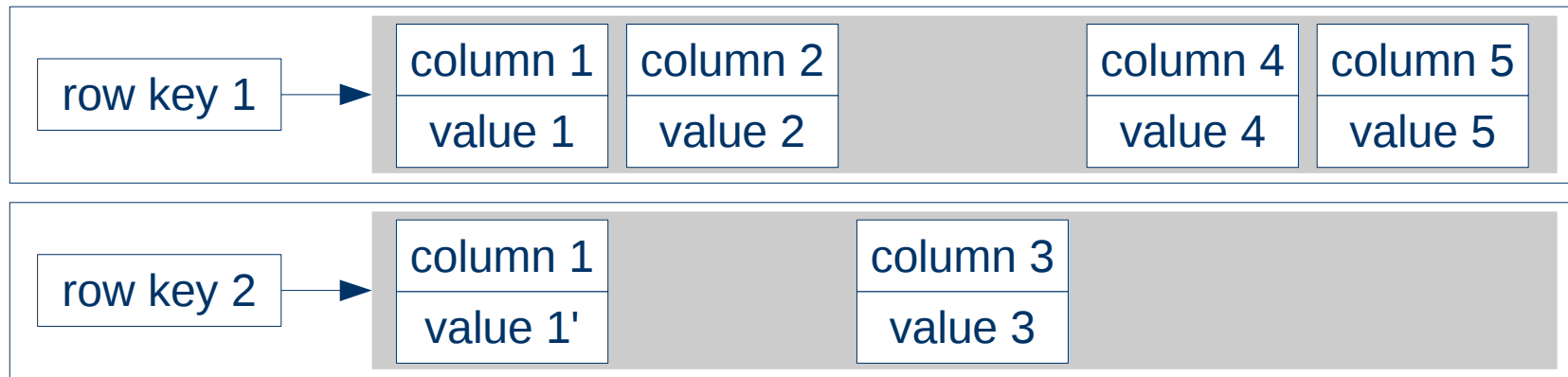
- Key-value model
- Document model
- Wide-column models
- Graph database models



also called  
*column-family models*  
or  
*extensible-record models*

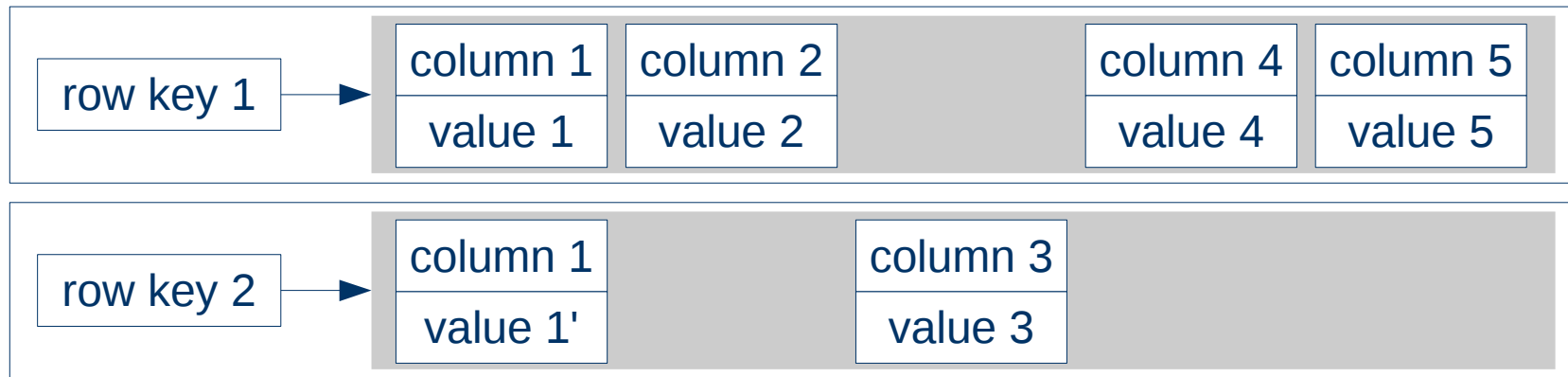
# Wide-Column Stores: Data Model (Basic)

- Database is a set of “rows” each of which ...  
... has a unique key, and  
... a set of key-value pairs (called “columns”)
- Schema free: different rows may contain different columns



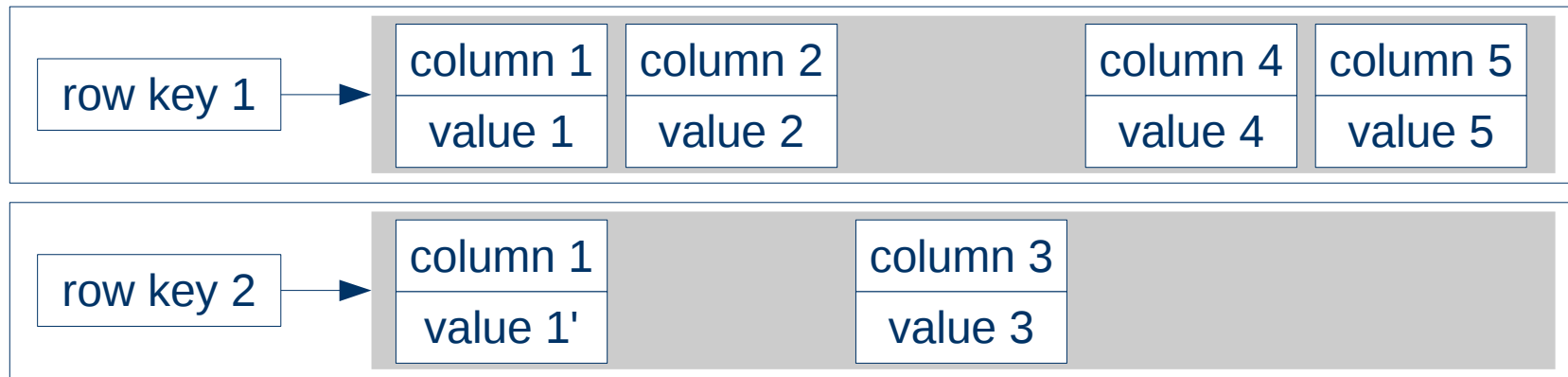
# Wide-Column Stores: Data Model

- Database is a set of “rows” each of which ...  
... has a unique key, and  
... a set of key-value pairs (called “columns”)
- Schema free: different rows may contain different columns
- Like a single, very wide relation (SQL table) that is  
a) extensible, b) schema-free, and c) potentially sparse



# Wide-Column Stores: Data Model

- Database is a set of “rows” each of which ...  
... has a unique key, and  
... a set of key-value pairs (called “columns”)
- Schema free: different rows may contain different columns
- Like a single, very wide relation (SQL table) that is  
a) extensible, b) schema-free, and c) potentially sparse
- Like the document model without nesting



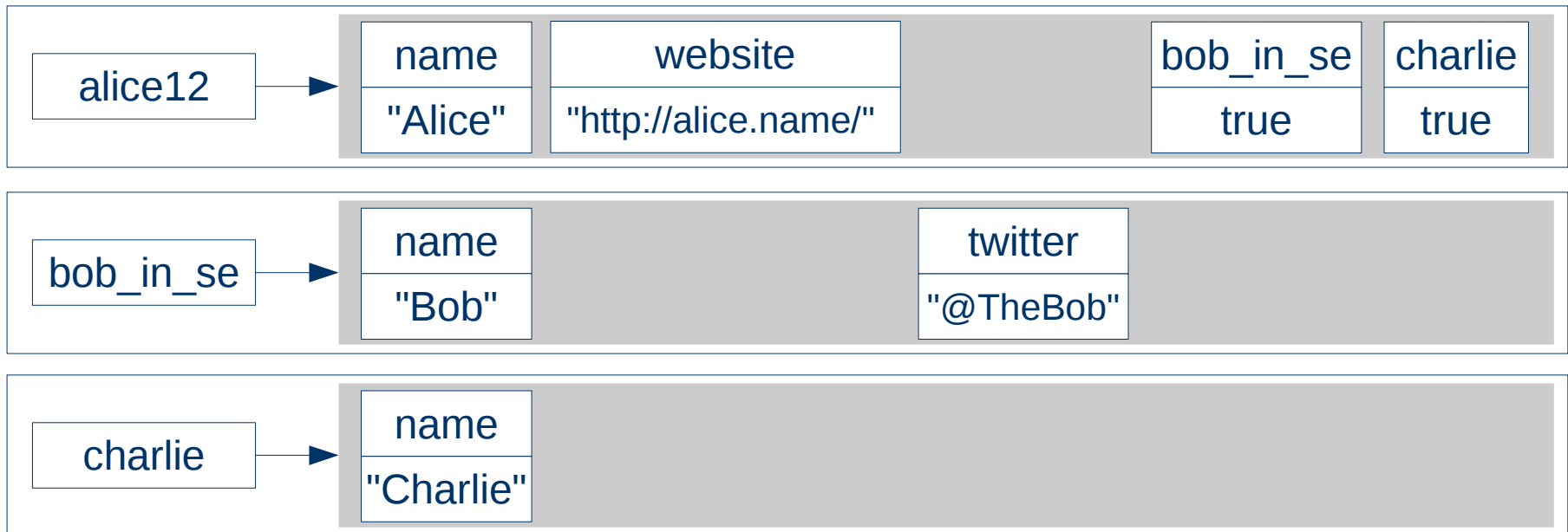
# Example (cont'd)

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

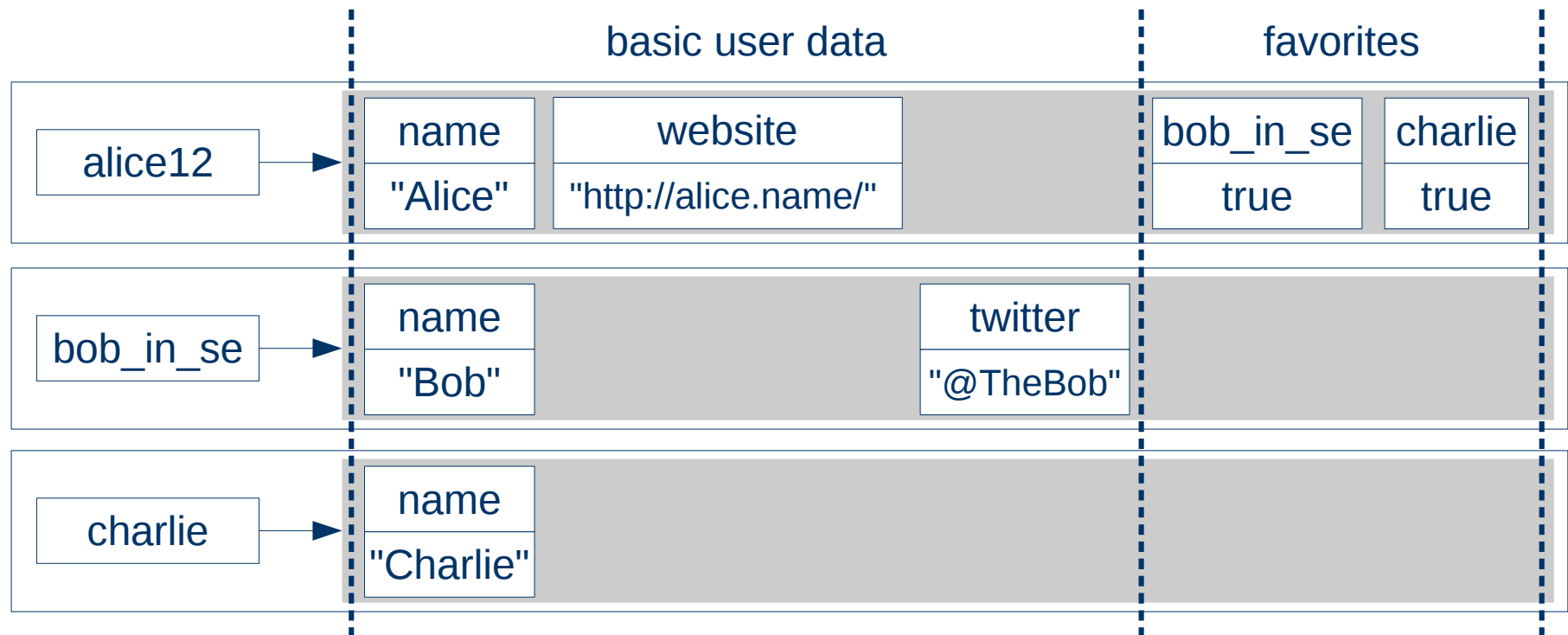
Fav

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



# Wide-Column Stores: Data Model (cont'd)

- Columns may be grouped into so called “column families”
  - Hence, values are addressed by  
(*row key*, *column family*, *column key*)



# Wide-Column Stores: Data Model (cont'd)

- Columns may be grouped into so called “column families”
  - Hence, values are addressed by  
*(row key, column family, column key)*
- Data may be partitioned ...
  - ... based on row keys (*horizontal partitioning*),
  - ... but also based on column families (*vertical partitioning*),
  - ... or even on both

# Wide-Column Stores: Data Model (cont'd)

- Columns may be grouped into so called “column families”
  - Hence, values are addressed by  
*(row key, column family, 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
- Queries expressed in a system-specific query language or in terms of program code using an API
  - Conceptually similar to queries in document stores
- No joins
  - Again, must be implemented in the application logic

# Wide-Column Stores: Use Cases

- Similar to use cases for document store
- Analytics scenarios
  - Web analytics
  - Personalized search
  - Inbox search

# Examples of Wide-Column Stores

- Basic form (no column families):

- Amazon SimpleDB
- Amazon DynamoDB




- With column families:

- Google's BigTable
- Hadoop HBase
- Apache Cassandra

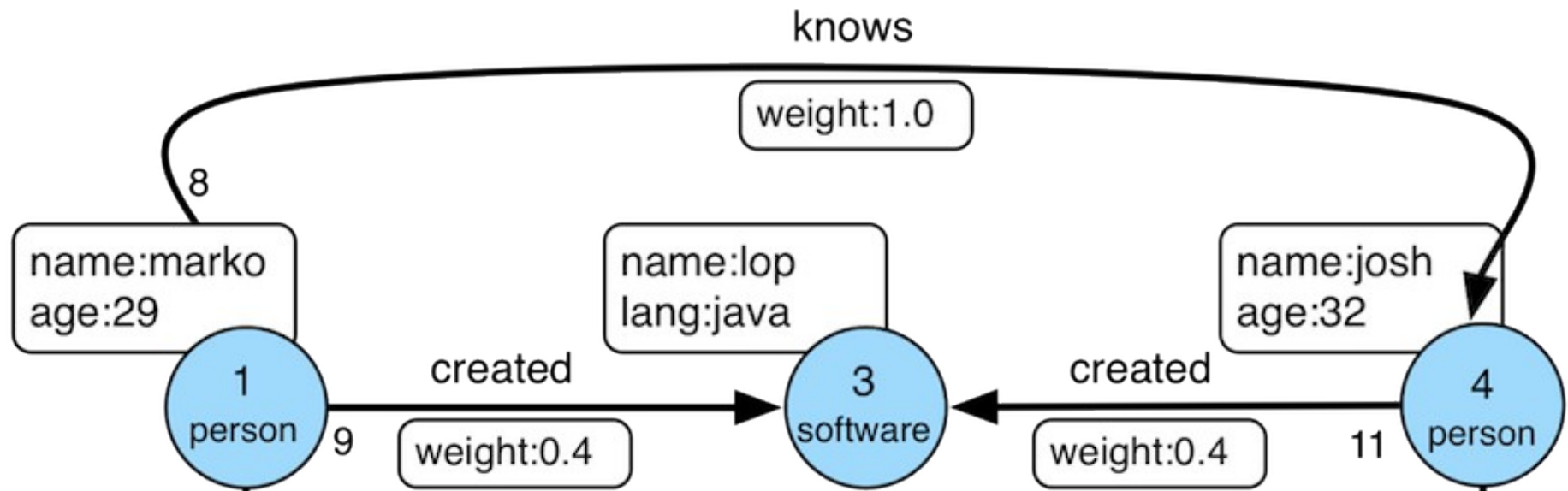


# Data Models

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

# Graph Database Systems: Data Model

- Database is some form of a graph (nodes and edges)
  - plus some extra features
- Prominent example: *Property Graphs* in which any node and any edge may additionally have a label as well as key-value pairs (called “properties”)









# Graph Database Systems: Querying

- Graph pattern matching
- Traversal queries
  - e.g., shortest paths, navigational expressions
- Graph algorithms
  - e.g., PageRank, connected components, clustering

# Graph Database Systems: Use Cases

- Complex networks
  - e.g., social, information, technological, biological
- Concrete example use cases:
  - Location-based services
  - Recommendation
  - Fraud detection

# Examples of (Property) Graph Systems

- Neo4j 
- TigerGraph 
- InfiniteGraph 
- JanusGraph 
- Cambridge Semantics' AnzoGraph 
- Amazon Neptune 



# Data Models

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

There are also *multi-model NoSQL stores*

Examples:

- OrientDB (key-value, documents, graph)
- ArangoDB (key-value, documents, graph)
- Cosmos DB (key-value, documents, wide-column, graph)



# Typical\* Characteristics of NoSQL Systems

- Ability to scale horizontally over many commodity servers with high performance, availability, and fault tolerance
  - achieved by giving up ACID guarantees
  - and by partitioning and replication of data
- Non-relational data model, no requirements for schemas
  - data model limitations make partitioning effective



\*Attention, there is a *broad variety* of such systems and not all of them have these characteristics to the same degree

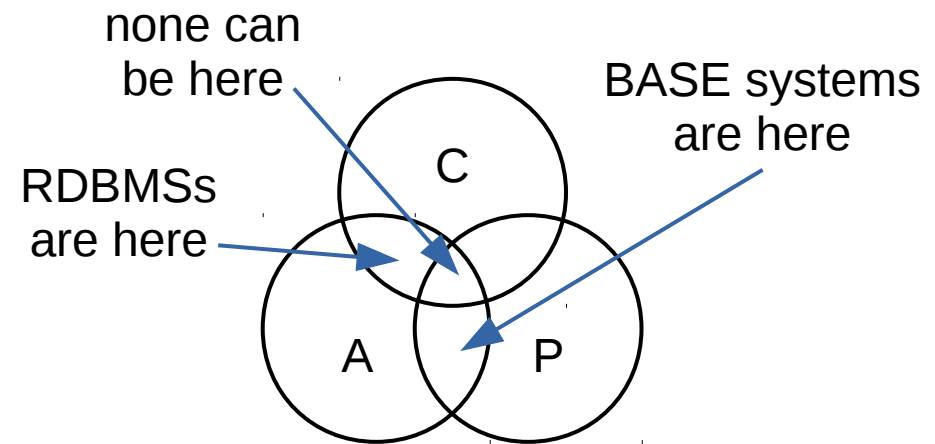
**BASE rather than ACID**

# What is BASE?

- Idea: by giving up ACID guarantees, one can achieve much higher performance and scalability
- **Basically Available**
  - system available whenever accessed, even if parts of it unavailable
- **Soft state**
  - the distributed data does not need to be in a consistent state at all times
- **Eventually consistent**
  - state will become consistent after a certain period of time
- BASE properties suitable for applications for which some inconsistency may be acceptable

# CAP Theorem

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



- **C**onsistency: the same copy of a replicated data item is visible from all nodes that have this item
  - Note that this is something else than consistency in ACID
- **A**vailability: all requests for a data item will be answered
  - Answer may be that operation cannot be completed
- **P**artition Tolerance: system continues to operate even if it gets partitioned into isolated sets of nodes

# Consistency Models

- *Strong consistency*: after an update completes, every subsequent access will return the updated value
  - may be achieved without consistency in the CAP theorem
- *Weak consistency*: no guarantee that all subsequent accesses will return the updated value
  - *eventual consistency*: if no new updates are made, eventually all accesses will return the last updated value
  - *inconsistency window*: the period until all replicas have been updated in a lazy manner

# Consistency Models (cont'd)

- Let:
  - $N$  be the number of nodes that store replicas
  - $R$  be the number of nodes required for a successful read
  - $W$  be the number of nodes required for a successful write
- Then:
  - Consistency as per CAP requires  $W = N$
  - Strong consistency requires  $R + W > N$
  - Eventual consistency if  $R + W \leq N$
  - High read performance means a great  $N$  and  $R = 1$
  - Fault tolerance/availability (and relaxed consistency)  $W = 1$

# Summary



# Summary

- NoSQL systems support non-relational data models (key-value, document, wide-column, graph)
  - 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 (effective due to data model limitations)
    - data replication
    - giving up consistency requirements