Advanced topics in Databases

Hellenic Mediterranean University

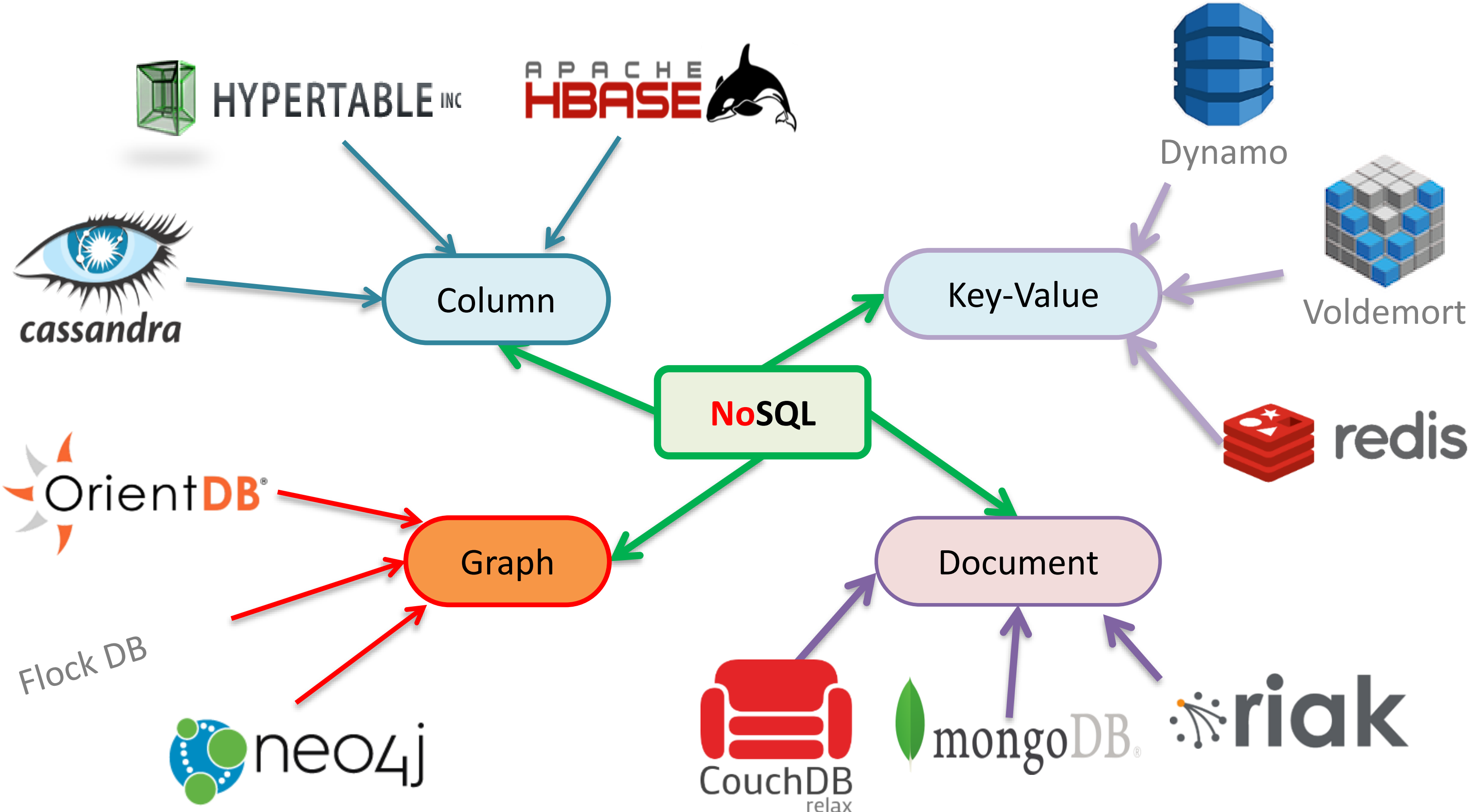
Prof. Demos Akoumianakis (da@hmu.gr)

Progress so far

- ✓ *Object-relational extensions*
  - *New data types*
    - *Enum, ARRAYS, Composite Data Types, XMLTABLE, JSON*
  - *Inheritance in Postgres*
- ✓ *Hierarchical data, Nested sets, Recursion*
- ✓ *Graph data modelling*
  - *Graph data models*
    - *Representing graphs, Property Graph, Path queries*
- Today
  - NoSQL concepts and systems with emphasis on key-value, column-family and document-based systems

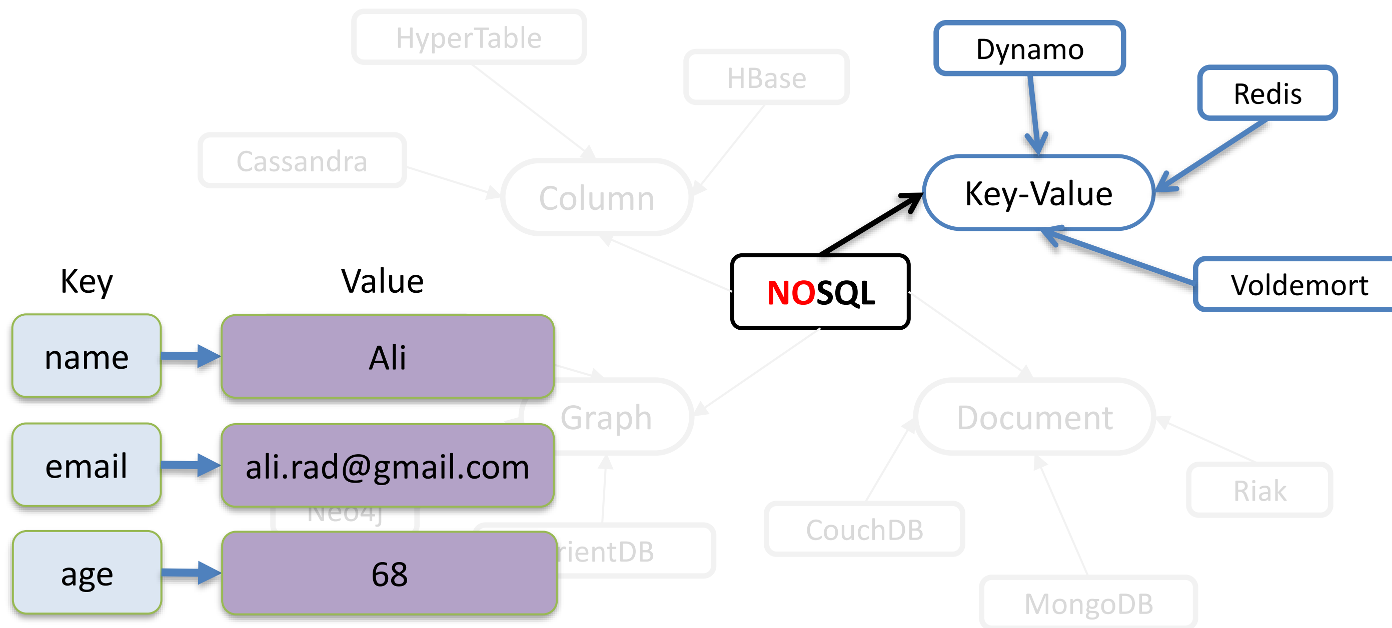
NoSQL systems

- Basic classification



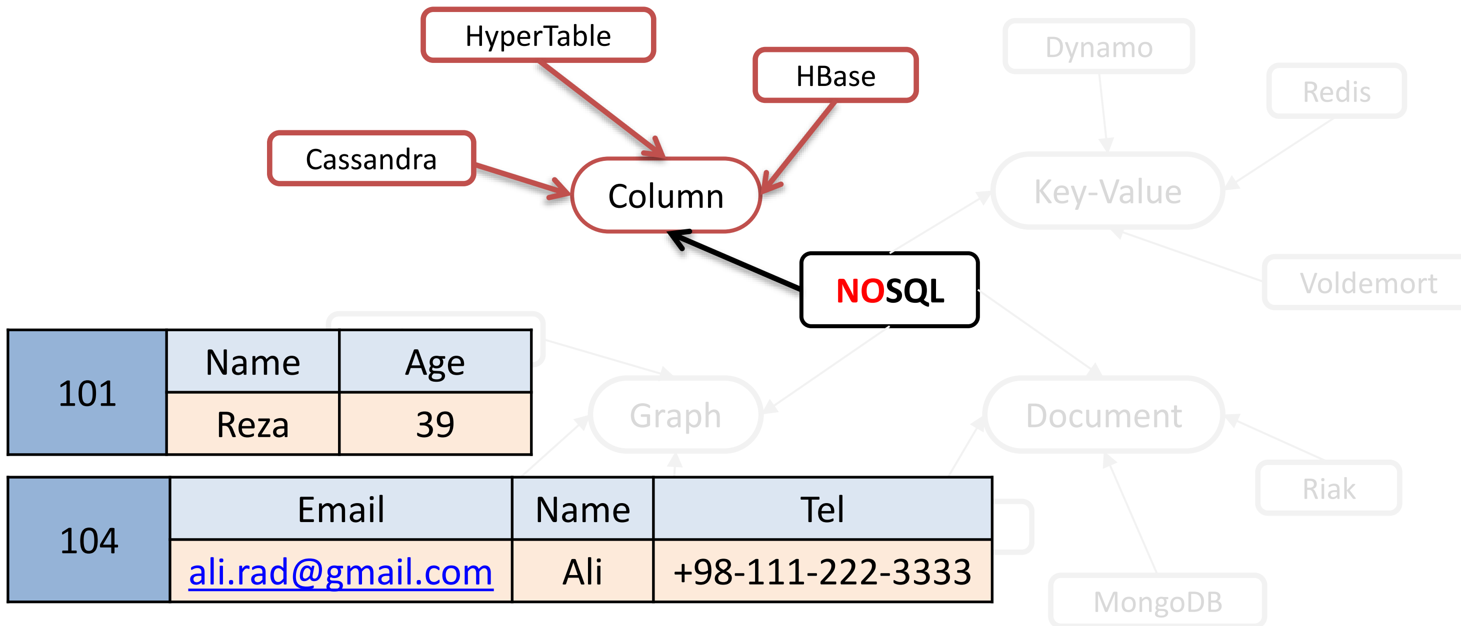
Key-value systems

- Underlying data model



Column

- Underlying data model

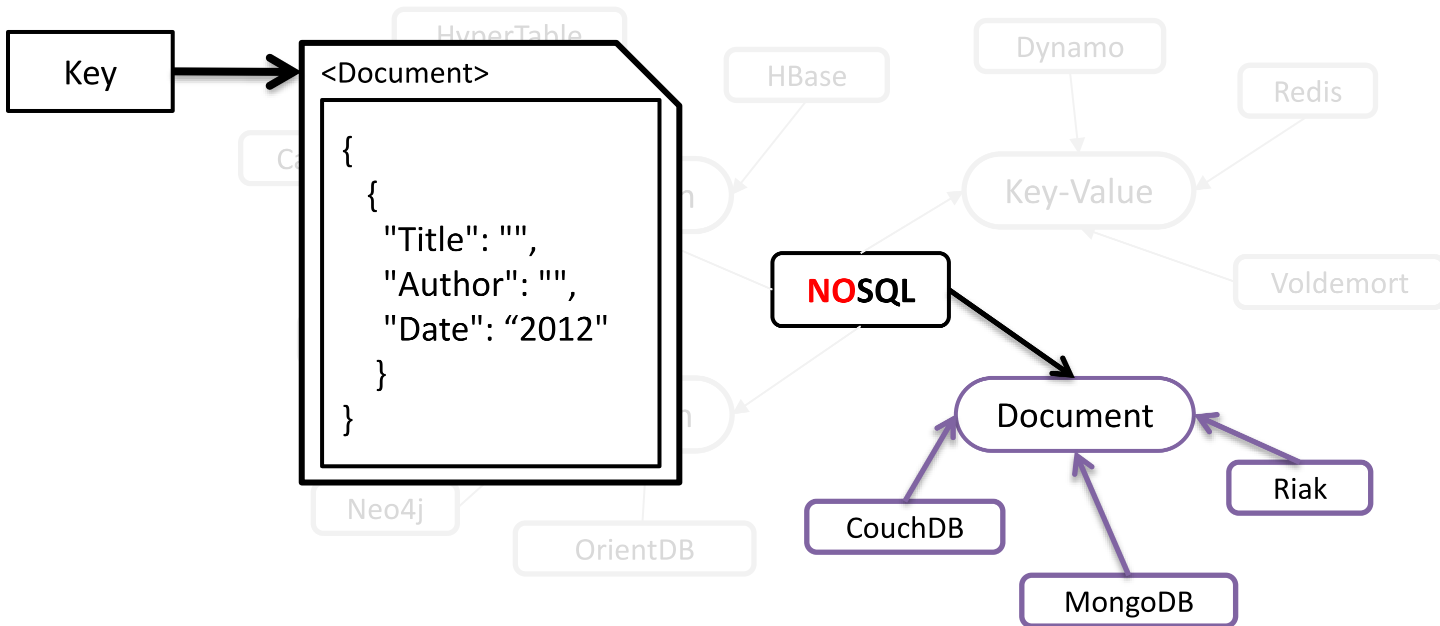


101	Name	Age
	Reza	39

104	Email	Name	Tel
	ali.rad@gmail.com	Ali	+98-111-222-3333

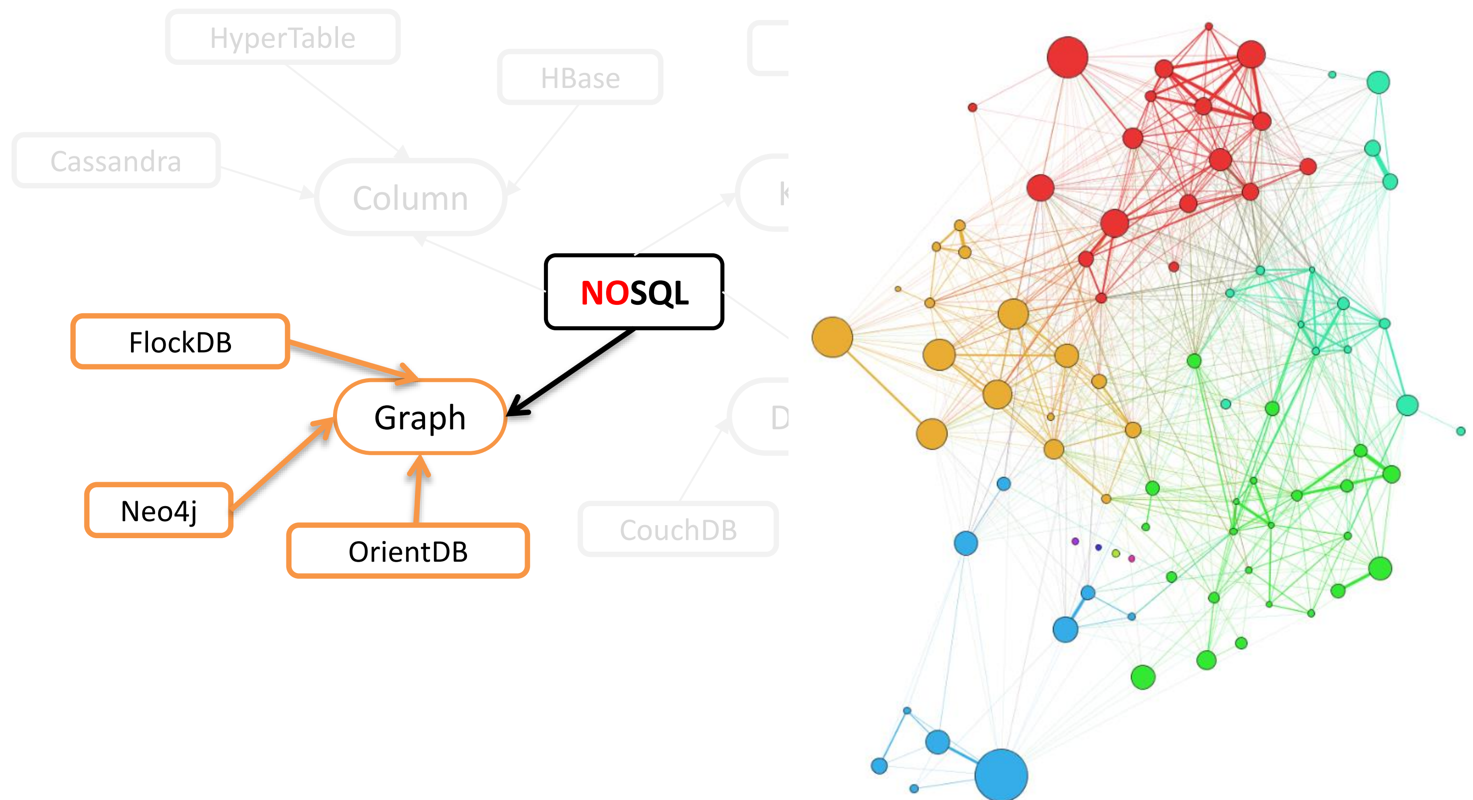
Document-based

- Underlying data model

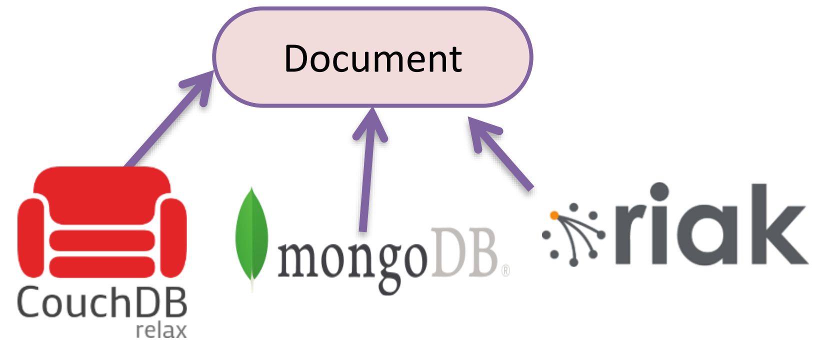


Graph

- Underlying data model



Document-based data stores



Principles

- Documents are stored in some standard format or encoding (e.g., XML, JSON)
  - These are typically referred to as Binary Large Objects (BLOBs)
- Documents can be indexed
  - This allows document stores to outperform traditional file systems
    - e.g., MongoDB and CouchDB (both can be queried using MapReduce)

Structure of a document

- Single identifier
- Key-value pairs for data items
  - First name
  - Surname
- Composites
  - Address
  - Hobbies

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  }
  "hobbies": ["surfing", "coding"]
}
```

Principles (cont.)

- Documents are stored in some standard format or encoding (e.g., XML, JSON)
  - Documents need not follow a single unified structure
- The system relies on internal document structure for *metadata* that allow for optimization

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  }
  "hobbies": ["surfing", "coding"]
}
```

The JSON Standard

JSON

- JSON stands for JavaScript Object Notation
- JSON represents data as
  - *Objects* meaning a collection of key-value pairs
    - Each object is defined as content between brackets { …. }
    - Data items are key-value pairs separated by comma

```
{
    "_id": "tomjohnson",
    "firstName": "Tom",
    "middleName": "William",
    "lastName": "Johnson",
    "email": "tj@digital.com",
}
```

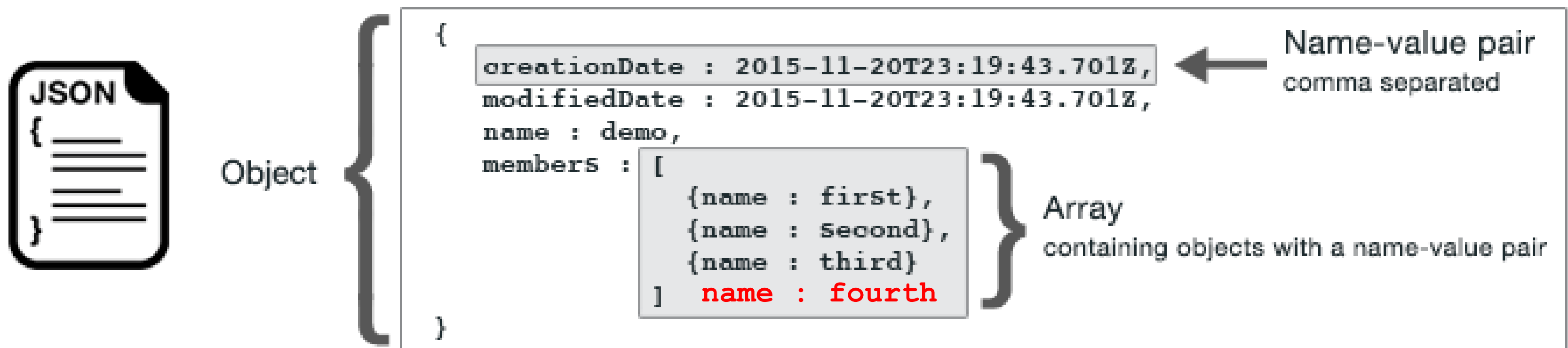
JSON (cont.)

- JSON stands for JavaScript Object Notation
- JSON represents data as
  - ✓ *Objects meaning a collection of key-value pairs*
  - – *Array* meaning an ordered set of data items
    - – Array's content is qualified by square brackets and [….]
    - – Data items separated by comma

```
{
    "_id": "tomjohnson",
    "firstName": "Tom",
    "middleName": "William",
    "lastName": "Johnson",
    "email": "tj@digital.com",
    "department": [
                    "Finance",
                    "Accounting"
                ]
}
```

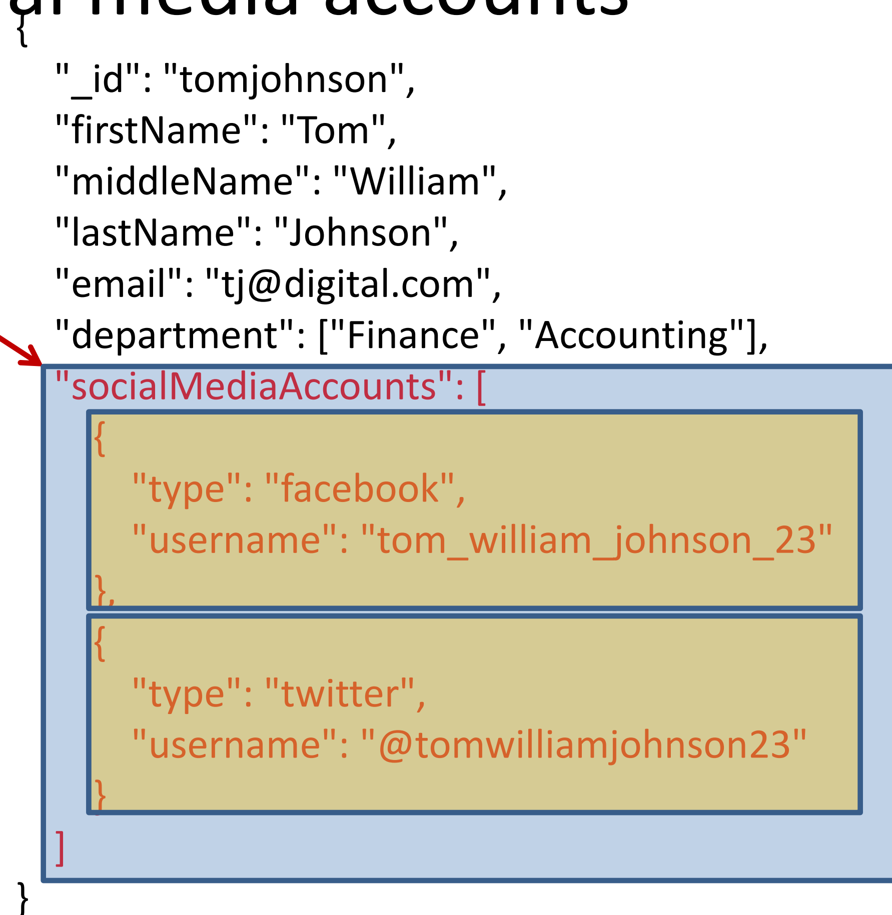
Schema

- Shema is dynamic
  - New objects / arrays may be added / removed following basic rules



Schema flexibility

- How can we update the document to add new objects for different social media accounts
  - socialMediaAccounts
    - ✓New object which instead of a single value it represents a set of profiles



Flexibility in representation

- Equally viable and appropriate alternatives

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  }
  "hobbies": ["surfing", "coding"]
}
```

```
{_id:
ObjectId("5effaa5662679b5af2c58829"),
 email: "email@example.com",
 name: {given:"Jesse", family:"Xiao"},
 age: 31,
 addresses: [{label: "home",
              street: "101 Elm Street",
              city: "Springfield",
              state: "CA",
              zip: "90000",
              country: "US"},
             {label: "mom",
              street: "555 Main
Street",
              city: "Jonestown",
              province: "Ontario",
              country: "CA"}]
}
```

Flexibility in representation (cont.)

- Equally viable and appropriate alternatives

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  }
  "hobbies": ["surfing", "coding"]
}
```

```
{_id:
ObjectId("5effaa5662679b5af2c58829"),
 email: "email@example.com",
 name: {given:"Jesse", family:"Xiao"},
 age: 31,
 addresses: [{label: "home",
              street: "101 Elm Street",
              city: "Springfield",
              state: "CA",
              zip: "90000",
              country: "US"},
             {label: "mom",
              street: "555 Main Street",
              city: "Jonestown",
              province: "Ontario",
              country: "CA"}]
}
```

An example (in class)

- Let us assume two users
  - ‘demos’ who is affiliated with HMU
  - ‘george’ who is affiliated with ‘HMU’ and ‘HOU’

The relational representation

- Let us assume two users
  - ‘demos’ who is affiliated with HMU
  - ‘george’ who is affiliated with ‘HMU’ and ‘HOU’
- Two-table (normalized) schema

_id	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

_id	Department
Demos	HMU
George	HMU
George	HOU

Document-based representation

- Let us assume two users
  - ‘demos’ who is affiliated with HMU
  - ‘george’ who is affiliated with ‘HMU’ and ‘HOU’
- JSON αναπαράσταση

```
{
  "_id": "demos",
  "firstName": "Demosthenes",
  "lastName": "Akoumianakis",
  "email": "da@hmu.gr",
  "department": "HMU"
}
```

```
{
  "_id": "george",
  "firstName": "Georgios",
  "lastName": "Papadakis",
  "email": "gpapadakis@nowhere.com",
  "department": ["HMU", "HOU"]
}
```

Querying models

- To find out the organizations a user is affiliated I need to *join* the tables instead of a issuing a simple *get()*

_id	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

_id	Department
Demos	HMU
George	HMU
George	HOU

```
{
  "_id": "demos",
  "firstName": "Demosthenes",
  "lastName": "Akoumianakis",
  "email": "da@hmu.gr",
  "department": "HMU"
}
{
  "_id": "george",
  "firstName": "Georgios",
  "lastName": "Papadakis",
  "email": "gp@nowhere.com",
  "department": ["HMU", "HOU"]
}
```

Moreover…

- What would be the implications if we needed to record middle names for some users?

_id	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

_id	Department
Demos	HMU
George	HMU
George	HOU

```
{
   "_id": "demos",
   "firstName": "Demosthenes",
   "lastName": "Akoumianakis",
   "email": "da@hmu.gr",
   "department": "HMU"
}
{
   "_id": "george",
   "firstName": "Georgios",
   "lastName": "Papadakis",
   "email": "gp@nowhere.com",
   "department": ["HMU", "HOU"]
}
```

Updates in the tables

- In the relational representation
  - Schema updates / handling NULL values for certain tuples
  - Recompilation of applications

_id	Firstname	Lastname	middleName	email
Demos	Demosthenes	Akoumianakis	John	da@hmu.gr
George	Georgios	Papadakis	NULL	gp@nowhere.com

_id	Department
Demos	HMU
George	HMU
George	HOU

Updates in the document

- Trivial update in the JSON document !

```
{
   "_id": "demos",
   "firstName": "Demosthenes",
   "middleName": "John",
   "lastName": "Akoumianakis",
   "email": "da@hmu.gr",
   "department": "HMU"
}
{
   "_id": "george",
   "firstName": "Georgios",
   "lastName": "Papadakis",
   "email": "gp@nowhere.com",
   "department": ["HMU", "HOU"]
}
```

Furthermore…

- Additional concerns
  - How can we allow our users to have different emails per affiliation?
    - Demos at HMU can be reached at da@hmu.gr
    - George at HMU can be reached at gp@nowhere.com while at HOU he can be reached at gp@hou.gr

Updates in the tables

- In the relational representation we would need an additional table and appropriate referential integrity constraints

_id	firstName	middleName	lastname
Demos	Demosthenes	John	Akoumianakis
George	Georgios		Papadakis

Organization
HMU
HOU

_id	Organization	email
Demos	HMU	da@hmu.gr
George	HOU	gp@hou.gr
George	HMU	gp@nowhere.com

Updates in the document

- In the JSON document we can simply add a separate data item for email per affiliation

```
{
   "_id": "demos",
   "firstName": "Demosthenes",
   "middleName": "John",
   "lastName": "Akoumianakis",
   "contact" :     {
             "email": "da@hmu.gr",
             "department": "HMU"
                 }
}
```

```
{
   "_id": "george",
   "firstName": "Georgios",
   "lastName": "Papadakis",
   "contact" :  {
             "email": "gp@nowhere.com",
             "department": "HMU"
                 }
                 {
             "email": "gp@hou.gr",
             "department": "HOU"
                 }
}
```

Linked documents

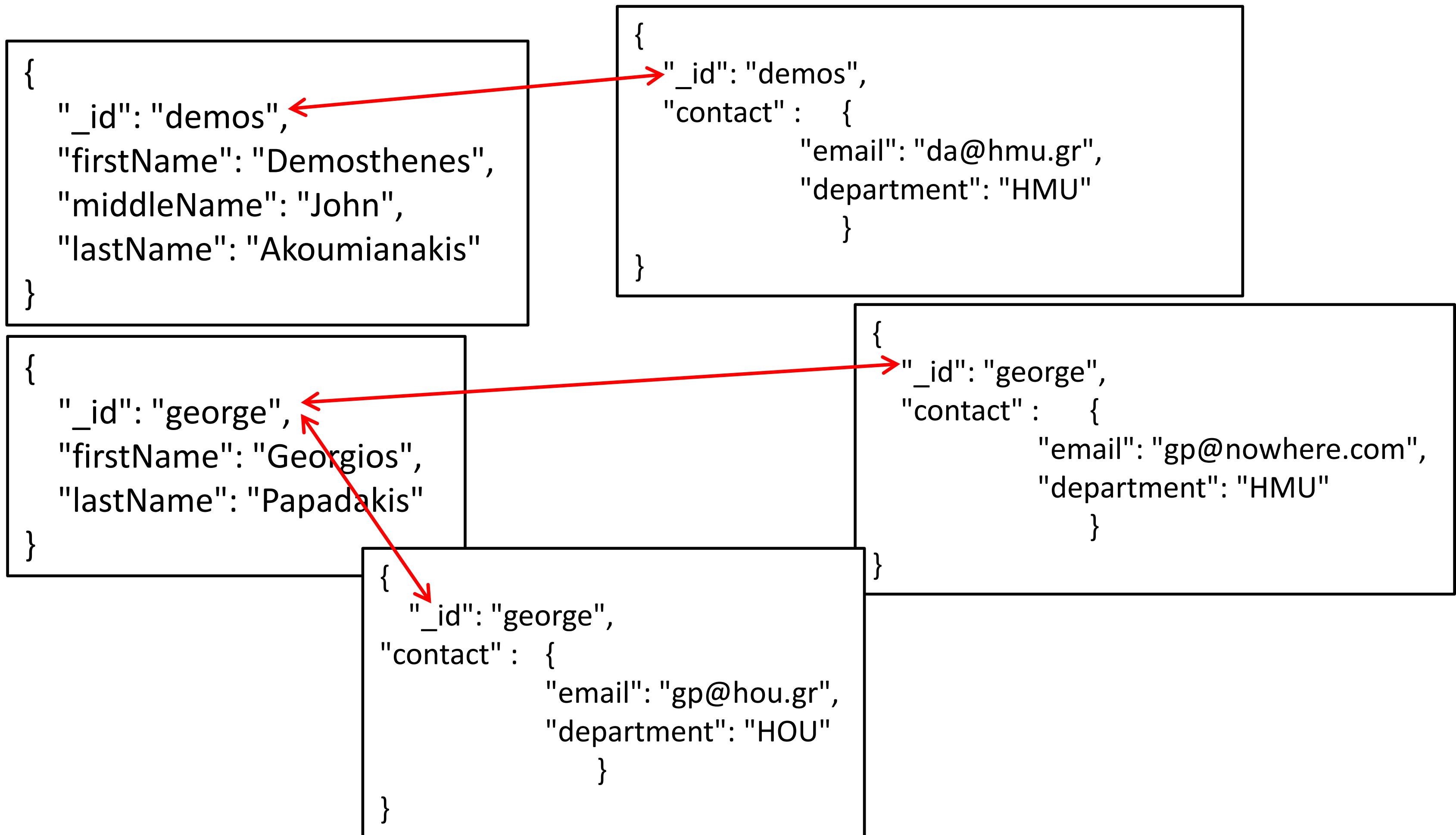
- An alternative would be to *link* documents

```
{
   "_id": "demos",
   "firstName": "Demosthenes",
   "middleName": "John",
   "lastName": "Akoumianakis"
}
```

```
{
  "_id": "demos",
  "contact" :    {
                   "email": "da@hmu.gr",
                   "department": "HMU"
                 }
}
```

Linked documents (cont.)

- An alternative would be to *link* documents



Data definition, manipulation and querying

mongoDB

Create or insert operations

- MongoDB provides the following methods to insert documents into a collection

```
db.collection.insertOne()
db.collection.insertMany()
```

Create or insert operations

- MongoDB provides the following methods to insert documents into a collection

```
db.collection.insertOne()
db.collection.insertMany()
```

```
db.users.insertOne(          <- collection
   {
     name: "sue",            <- field: value
     age: 26,                <- field: value   } document
     status: "pending"       <- field: value
   }
)
```

Create or insert operations

- MongoDB provides the following methods to insert documents into a collection

```
db.collection.insertOne()
db.collection.insertMany()
```

```
db.collection.insertMany(
[ <document 1> , <document 2>, ... ],
{
writeConcern: <document>,
ordered: <boolean>
}
)
```

Read Operations

- Read operations retrieve documents from a collection

```
db.collection.find(query, projection, options)
```

```
db.users.find(                            ← collection
   { age: { $gt: 18 } },                  ← query criteria
   { name: 1, address: 1 }                ← projection
).limit(5)                                ← cursor modifier
```

The operator $eq

- $eq
  - Specifies equality condition and matches documents where the value of a field equals the specified value
    { <field>: { $eq: <value> } }

- Select all documents in a collection where the value of the qty field equals 20

```
db.inventory.find( { qty: { $eq: 20 } } )
```

The operator $gt

- $gt
  - Selects those documents where the value of the field is greater than (i.e. >) the specified value

```
{ field: { $gt: <value> } }
```

- Select all documents in a collection where quantity is greater than 20

```
db.inventory.find( { quantity: { $gt: 20 } } )
```

The API online

- https://www.mongodb.com/docs/manual/crud/#std-label-crud

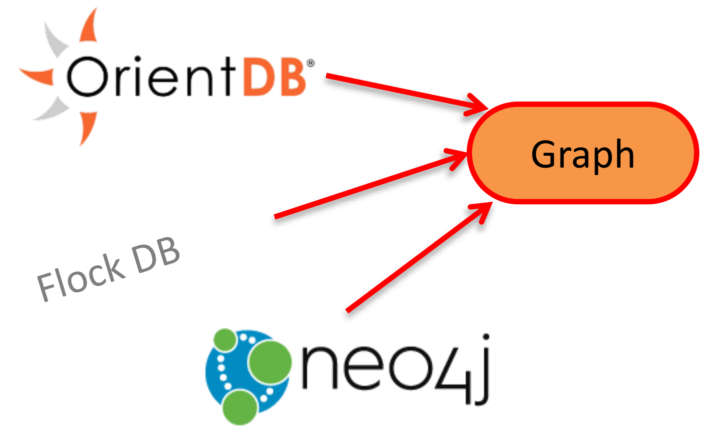
Document-based databases

- Positive aspects
  - Simple model with powerful representational power
  - Easy scalability
- Negative aspects
  - Problems with interconnected data
  - Limited capacity of querying language

Resources

- How to create a Mongo database
  - https://www.mongodb.com/basics/create-database

Neo4j - https://neo4j.com/

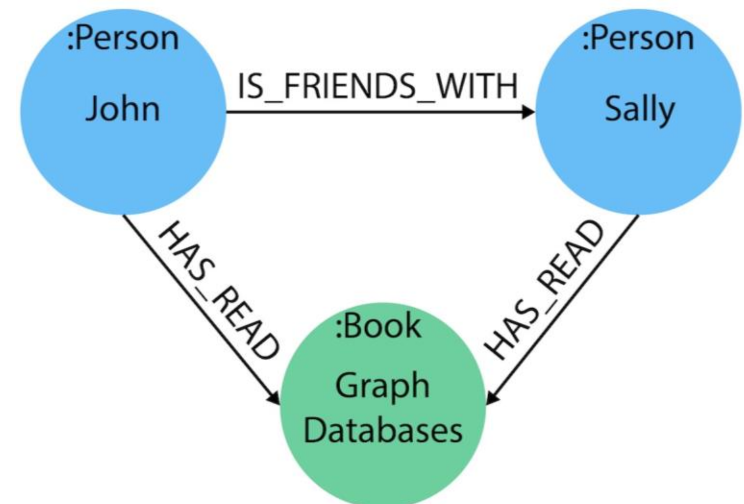


Outline

- Neo4j
  - Concepts
  - Usage
- Cypher Querying Language
  - Path querying
  - Basic patterns

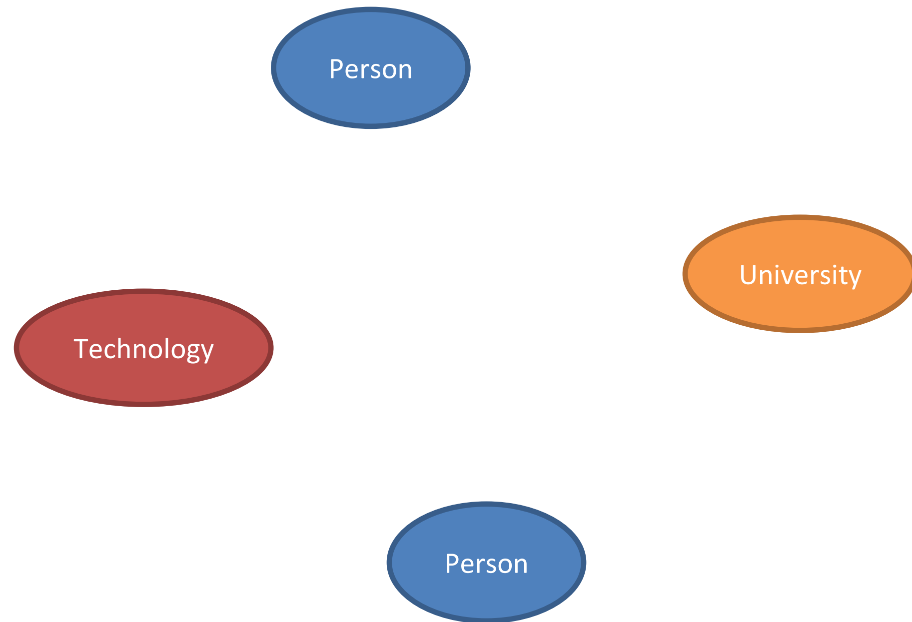
Graph Databases

- A graph is simply a collection of nodes joined by edges
  - Each *node* represent information, and
  - Each *edge* represent some connection between two nodes
- A graph database will typically be a *directed* graph



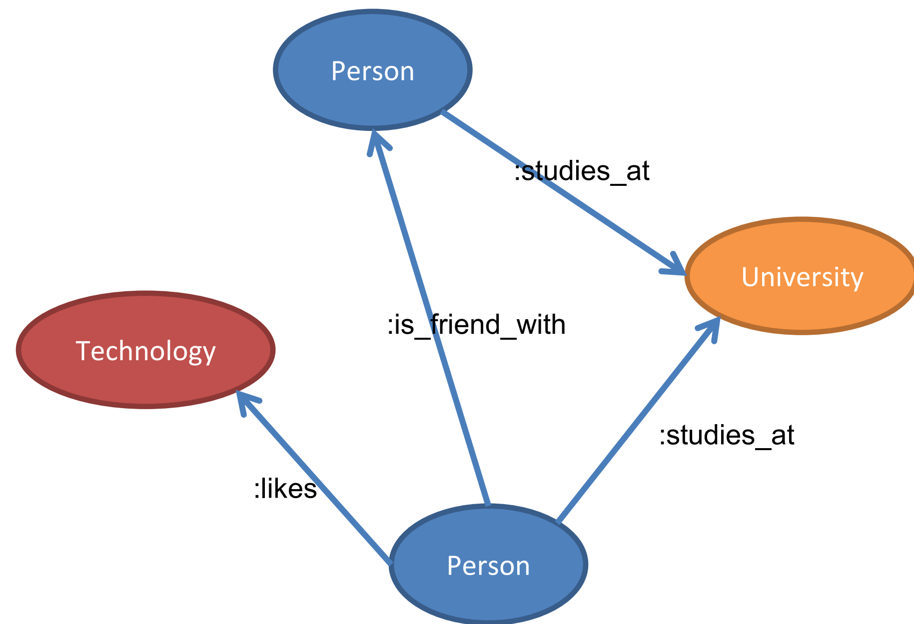
Neo4j – Native graph database

- Graph structure
  - **Nodes (Vertex)** represents the objects of our graph and they can be labeled



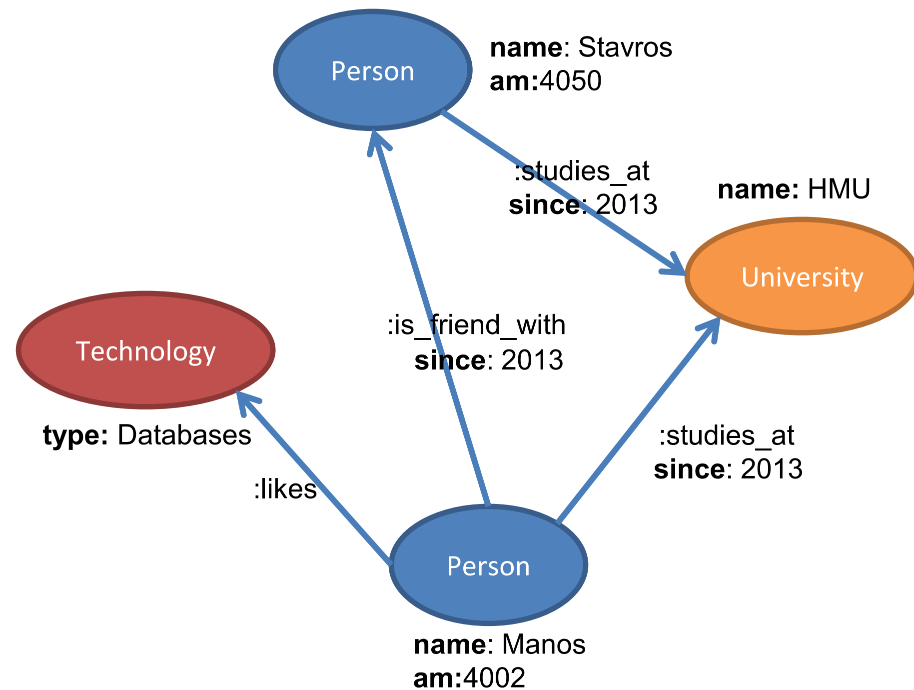
Neo4j – Native graph database

- Graph structure
  - **Nodes (Vertex)** represents the objects of our graph and they can be labeled
  - **Relationships (Edges)** relate nodes by type and direction



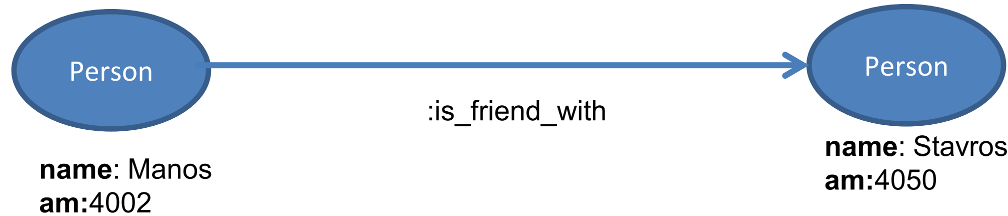
Neo4j – Native graph database

- Graph structure
  - **Nodes (Vertex)** represents the objects of our graph and they can be labeled
  - **Relationships (Edges)** relate nodes by type and direction
  - **Properties** are name-value pairs that can define nodes and relationships



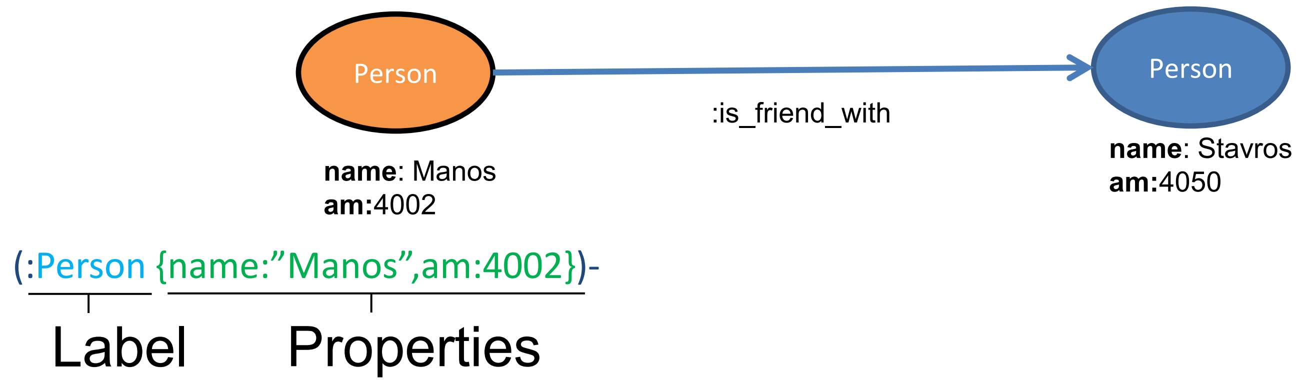
Neo4j Querying

- Cypher is a declarative pattern matching query language made for graphs that relies on graph patterns or path queries



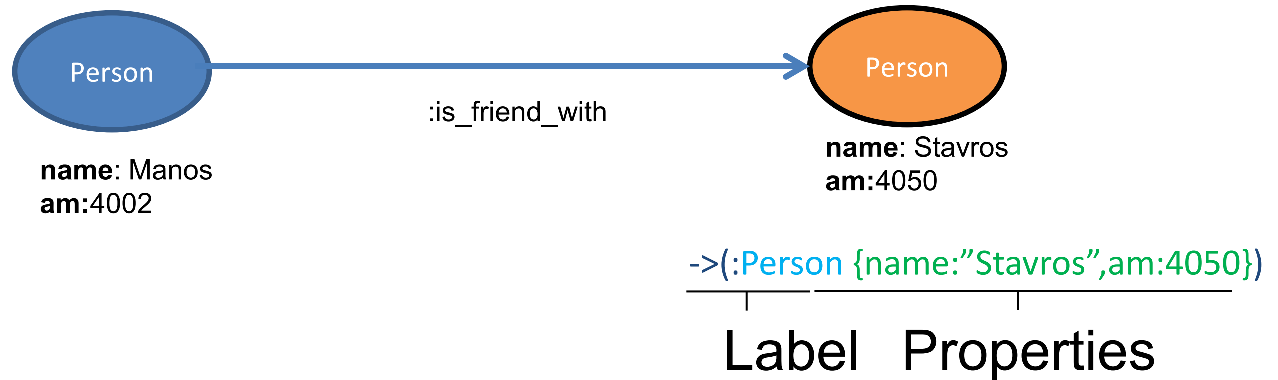
Patterns

- Specifying a starting node



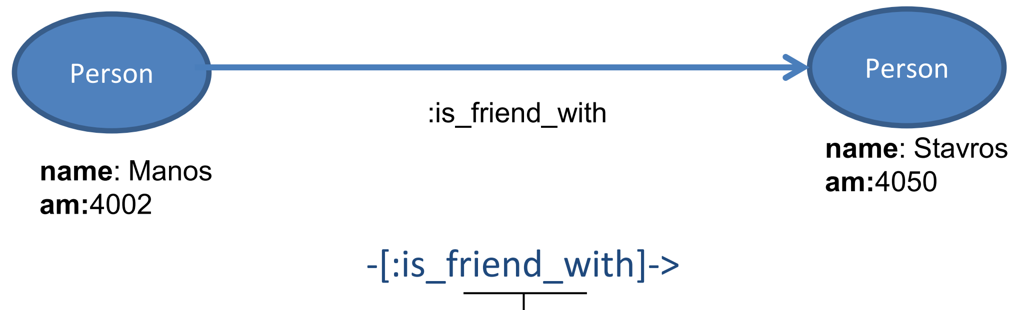
Patterns

- Specifying a destination node



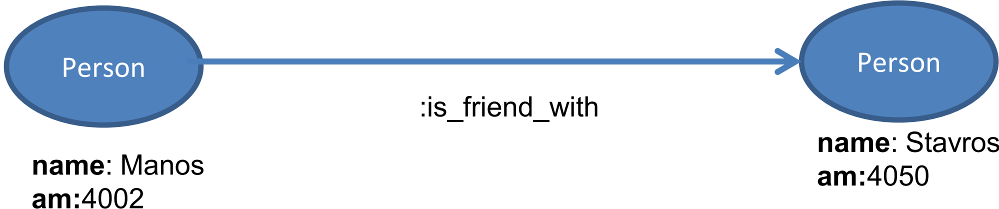
Patterns

- … and the connection between them



Patterns

- A typical path query or graph pattern

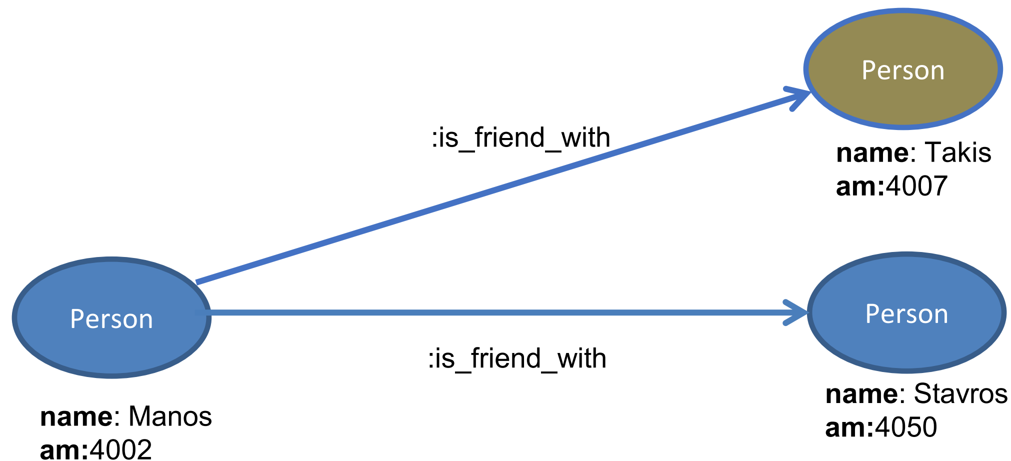


(:Person {name:”Manos”,am:4002}) -[:is_friend_with] ->(:Person {name:”Stavros”,am:4050})

Label Properties Label Properties

Neo4j Querying - Create

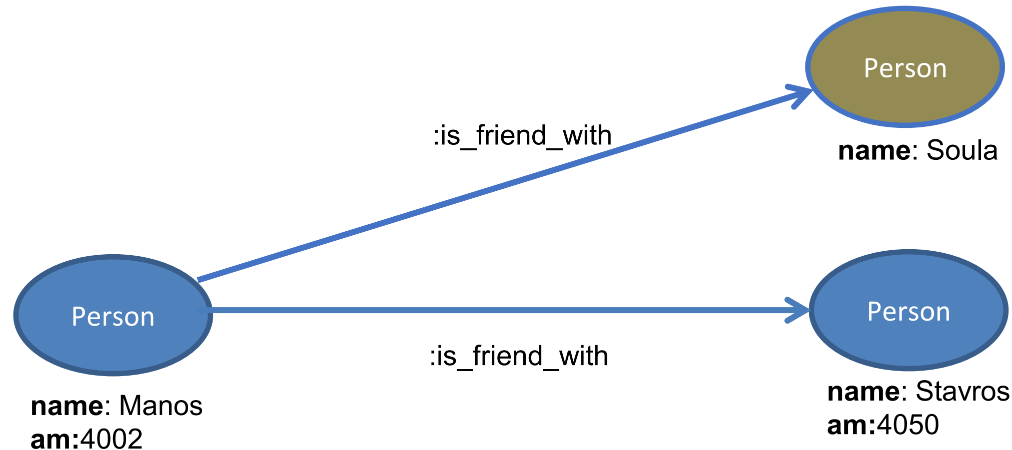
- Creating a graph pattern




```
CREATE(:Person {name:"Manos",am:4002})-[:is_friend_with]->(:Person {name:"Takis",am:4007})
```

Neo4j querying – Match

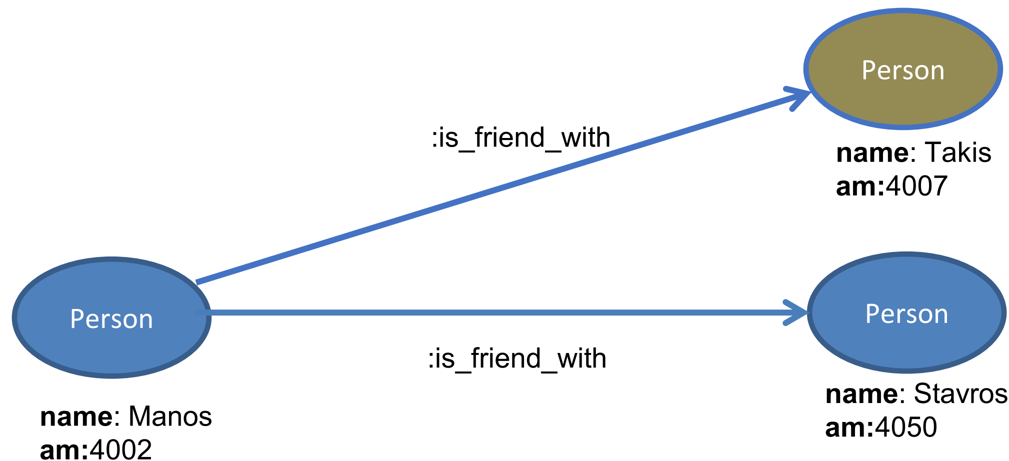
- Let's assume that I would like to add my new friend into the graph without adding yet another node of mine!




```
MATCH (p)
WHERE p.name="Manos"
CREATE (:Person {name:"Soula"})-[:is_friend_with]->(p)
```

Neo4j querying – Add

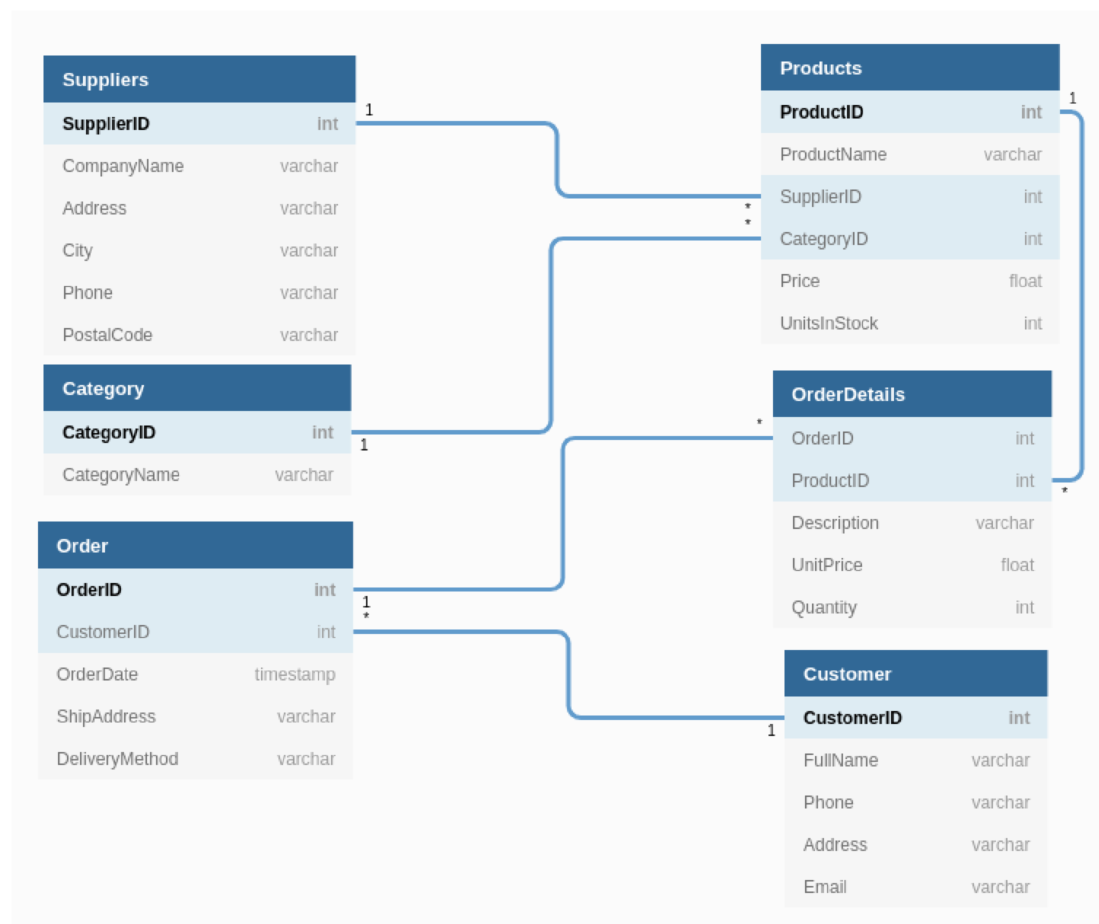
- Let’s find the friend of mine using the **MATCH** method



MATCH(:Person {name:”Manos”,am:4002})-[:is_friend_with]->(whom) **RETURN** whom

From relations to graphs

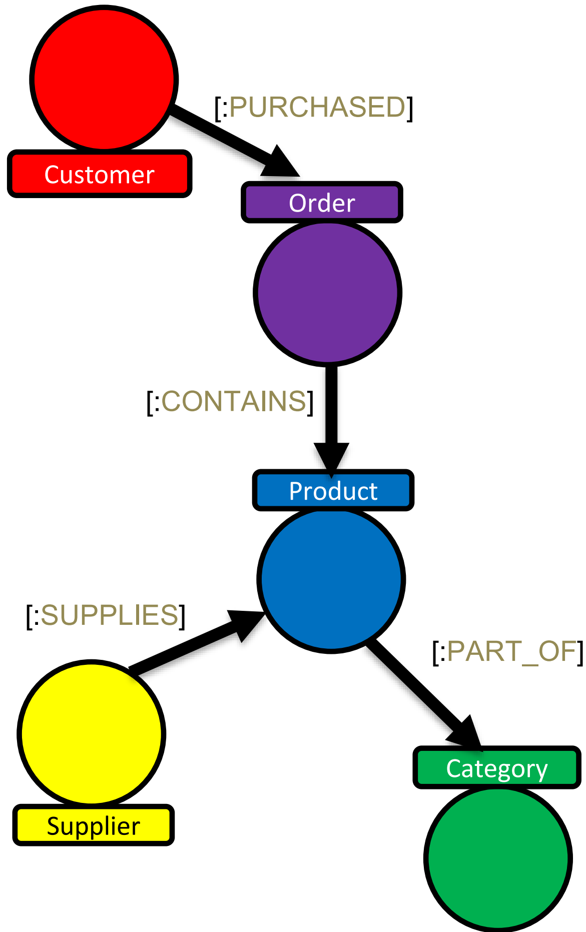
- Let assume that we have a "mini" database containing Products, Categories, Suppliers, and Customers

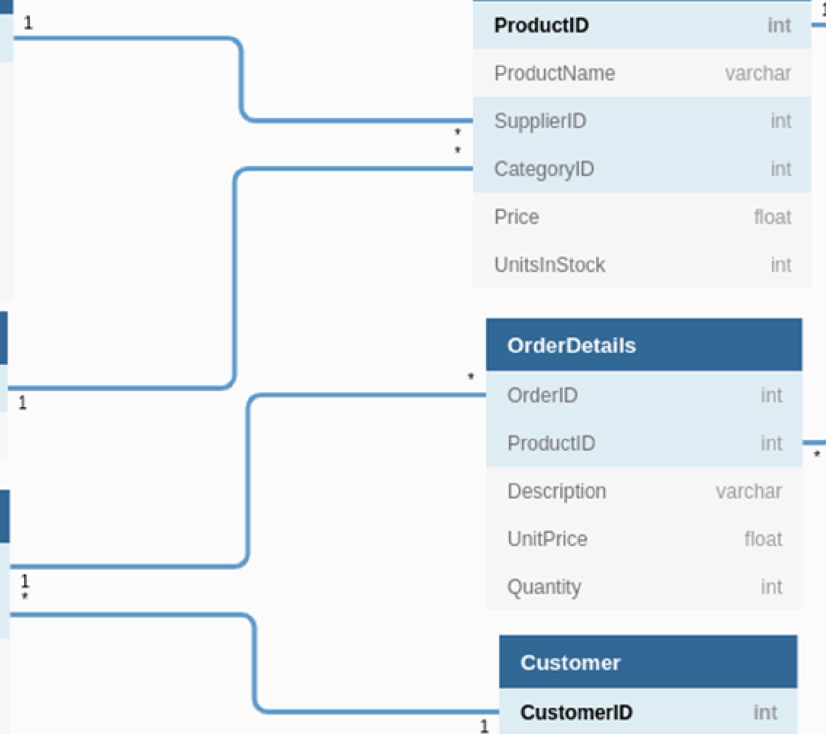


Developing a graph model

- Simple guidelines (not the only ones)
  - A ***table name*** is a ***label name***
  - Each ***row*** is a ***node***
  - Example
    - Each row on our Orders table becomes a node in our graph with Order as a label
  - …. and so on
  - A ***join*** or ***foreign key*** is a ***relationship***
  - Example
    - Join between Suppliers and Products becomes a relationship name SUPPLIES
    - Join between Customer and Order becomes a relationship named PURCHASED
  - …. and so on

Whiteboard Translation



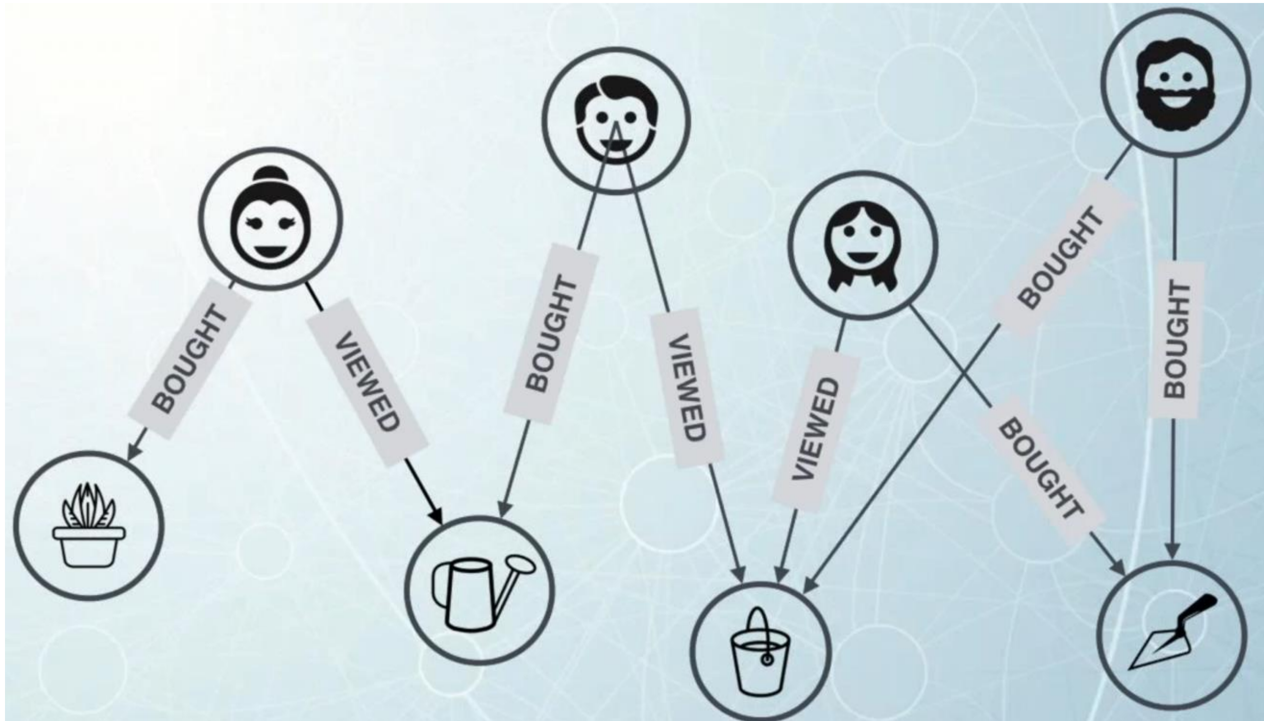


Graph database use cases

- Keyword-based search tools for enterprises
- Facebook and Google offered a basic "keyword" search, where users would type in a word or phrase and get back a list of all results that included those keywords

Graph database use cases (cont.)

- Real-time RECOMMENDATIONS

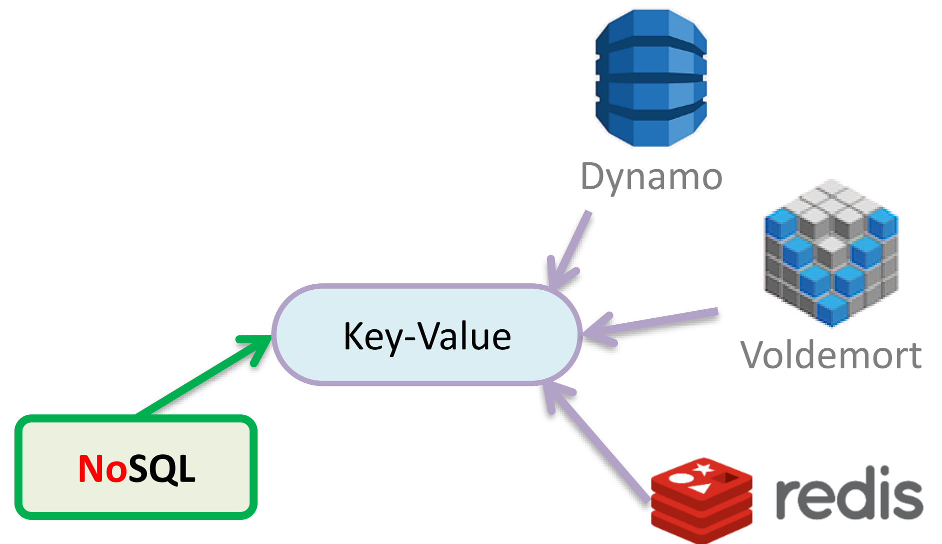


Free Library from Neo4j

- O'Reilly Graph Databases - Neo4j Graph Database Platform
- Fullstack GraphQL Applications with GRANDstack – Essential Excerpts (neo4j.com)
- Graph Databases for Dummies - Neo4j Graph Database Platform
- O'Reilly Graph Algorithms Book - Neo4j Graph Database Platform
- Graph-Powered Machine Learning - Neo4j Graph Database Platform

- **Cypher Docs :**
- https://neo4j.com/docs/cypher-manual/current/

Key-value data stores



Key-value data stores

- Amazon
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)

- Facebook, Twitter
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)

- iCloud/iTunes
  - Key: Movie/song name
  - Value: Movie, Song

- Distributed file systems
  - Key: Block ID
  - Value: Block

Key-value store

- Principles
  - Schema-less
  - Data stored as *key-value* pairs

K_1	V_1
K_2	V_2
K_3	V_3
K_v	V_v

Key-value store

- Principles
  - Schema-less
  - Data stored as *key-value* pairs

K_1	V_1
K_2	V_2
K_3	V_3
K_v	V_v

- A *key-value pair*
  - ✓The key
    - *unique* id of a data item
    - Simple or composit
    - Assigned by an application or the *key-value data store*

Key-value store

- Principles
  - Schema-less
  - Data stored as *key-value* pairs

K_1	V_1
K_2	V_2
K_3	V_3
K_v	V_v

- A *key-value pair*

✓The key
  - *unique* id of a data item
  - Simple or composit
  - Assigned by an application or the *key-value data store*

✓The value
  - the *data item* itself or its *location*

Key-value store

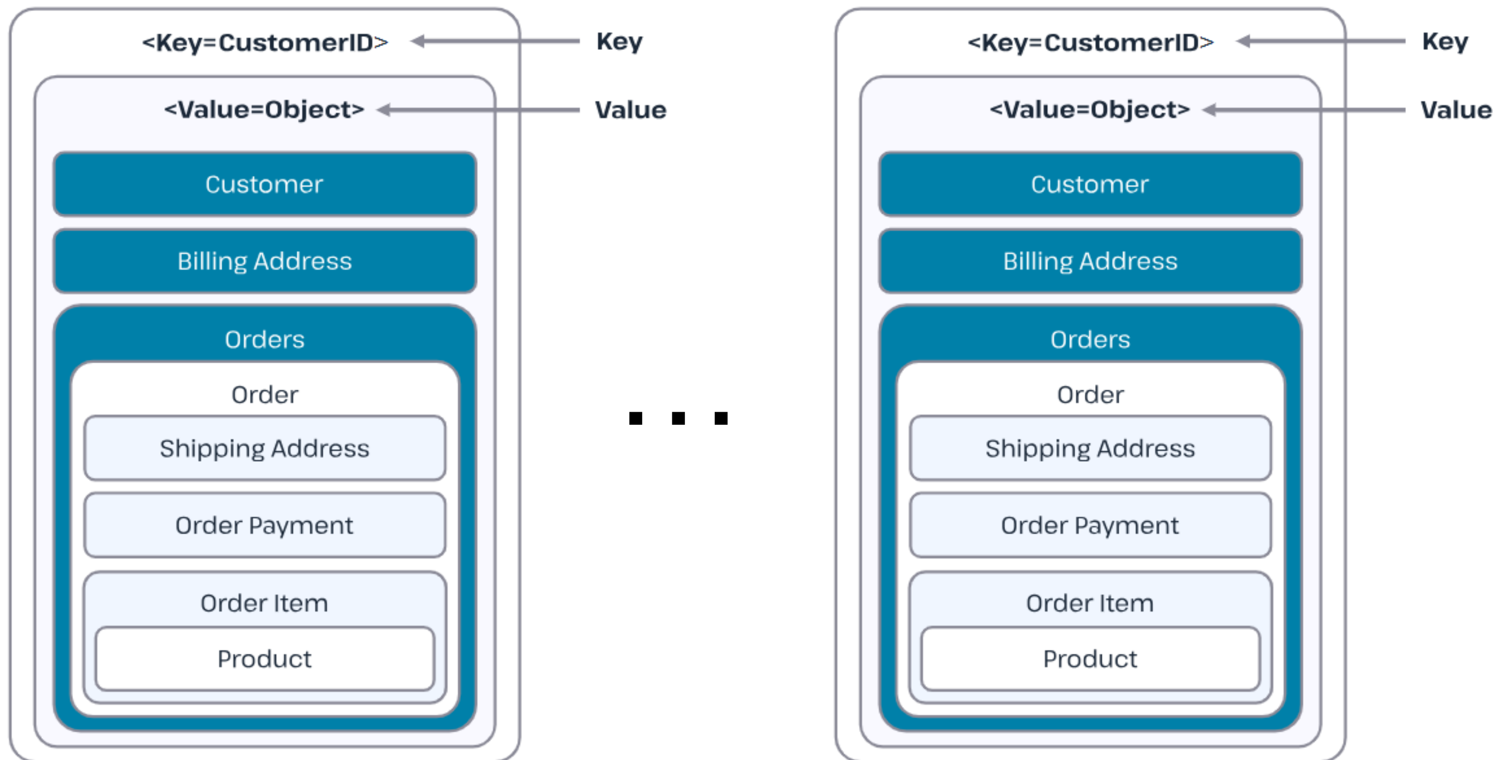
- Principles
  - Schema-less
  - Data stored as *key-value* pairs

K_1	V_1
K_2	V_2
K_3	V_3
K_v	V_v

- A *key-value pair*
  - ✓The key
    - *unique* id of a data item
    - Simple or composit
    - Assigned by an application or the *key-value data store*
  - ✓The value
    - the *data item* itself or its *location*
- No *querying language*

Key-value store (cont.)

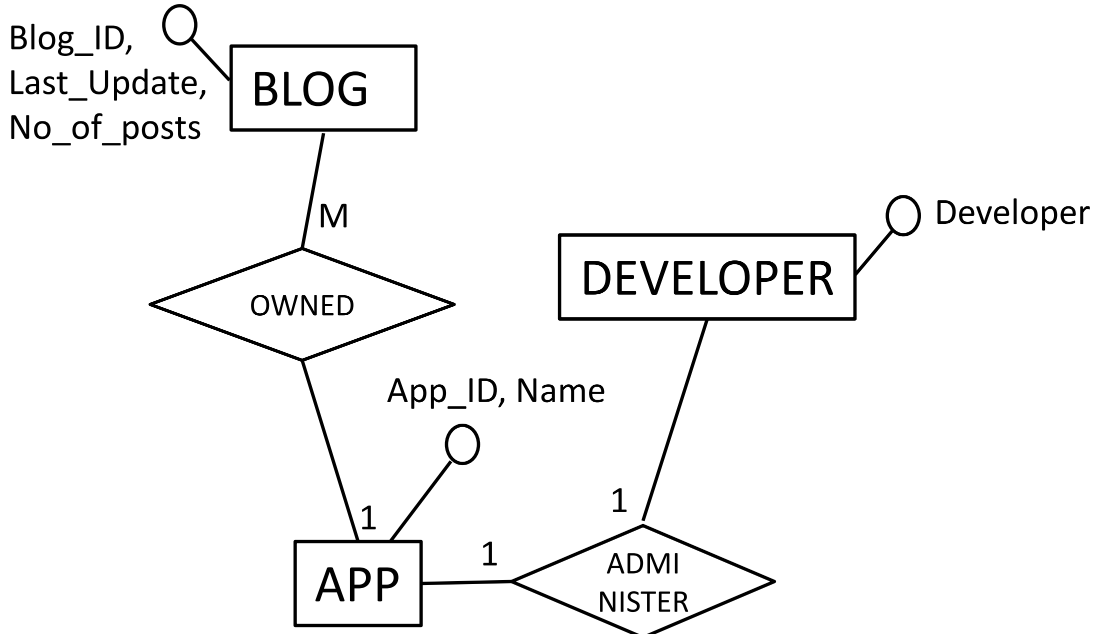
- The value of a data item can be of any type – numeric, text or another key-value pair



From relations to key-value store

Example

- AppStore



- Resulting in three-table schema

Example (cont.)

- Assuming that for the two tables we have

APP

App_id	Name	Developer
'com.facebook.orca'	'Messenger'	4
'com.viber.voip'	'Viber Messenger'	6
'com.whatsapp'	'WhatsApp Messenger'	5

Primary key

Foreign key

BLOG

Blog_ID	*Blog_Owner*	Last_Update	No_of_posts
1	'com.facebook.orca'	13/03/2022	332
2	'com.viber.voip'	15/03/2022	10003
3	'com.whatsapp'	17/03/2022	7

Example (cont.)

- Table APP as key-value pairs

APP

App_id	Name	Developer
'com.facebook.orca'	'Messenger'	4
'com.viber.voip'	'Viber Messenger'	6
'com.whatsapp'	'WhatsApp Messenger'	5

Key	Value
'com.facebook.orca'	<'Messenger’, 4>
'com.viber.voip'	<'Viber Messenger’, 6>
'com.whatsapp'	<'WhatsApp Messenger’, 5>

Example (cont.)

- Table BLOG as key-value pairs

BLOG

Blog_ID	Blog_Owner	Last_Update	No_of_posts
1	'com.facebook.orca'	13/03/2022	332
2	'com.viber.voip'	15/03/2022	10003
3	'com.whatsapp'	17/03/2022	7

Key	Value
1	<'com.facebook.orca’, 13/03/2022, 332>
2	<'com.viber.voip', 15/03/2022, 10003>
3	<'com.whatsapp', 17/03/2022, 7>

Example (cont.)

- One (unified) representation

Key	Value
1	<'com.facebook.orca', 13/03/2022, 332>
2	<'com.viber.voip', 15/03/2022, 10003>
3	<'com.whatsapp', 17/03/2022, 7>

Key	Value
'com.facebook.orca'	<'Messenger', 4>
'com.viber.voip'	<'Viber Messenger', 6>
'com.whatsapp'	<'WhatsApp Messenger', 5>

App_ID
Name
Developer
Last_Update
No_of_posts

Key	Value
1	<'com.facebook.orca', 'Messenger', 4, 13/03/2022, 332>
2	<'com.viber.voip', 'Viber Messenger', 6, 15/03/2022, 10003>
3	<'com.whatsapp', 'WhatsApp Messenger', 5, 17/03/2022, 7>

Example (cont.)

- What is the benefit?
  - ‘What is the number of posts in the APP blog administered by developer with code 4’?

Relational store

BLOG

Blog_ID	Blog_Owner	Last_Update	No_of_posts
1	'com.facebook.orca'	13/03/2022	332
2	'com.viber.voip'	15/03/2022	10003
3	'com.whatsapp'	17/03/2022	7

APP

App_id	Name	Developer
'com.facebook.orca'	'Messenger'	4
'com.viber.voip'	'Viber Messenger'	6
'com.whatsapp'	'WhatsApp Messenger'	5

Key-value

Key	Value
1	<'com.facebook.orca’, 'Messenger’, 4, 13/03/2022, 332>
2	<'com.viber.voip’, 'Viber Messenger’, 6, 15/03/2022, 10003>
3	<'com.whatsapp', 'WhatsApp Messenger’, 5, 17/03/2022, 7>

Example (cont.)

- Using a relational language (SQL)
  - *Selection* (σ) from table App where developer is 4

BLOG

Blog _ID	Blog_Owner	Last_Update	No_of_ posts
1	'com.facebook.orca'	13/03/2022	332
2	'com.viber.voip'	15/03/2022	10003
3	'com.whatsapp'	17/03/2022	7

APP

App_id	Name	Developer
'com.facebook.orca'	'Messenger'	4
'com.viber.voip'	'Viber Messenger'	6
'com.whatsapp'	'WhatsApp Messenger'	5

σ DEVELOPER = 4

APP

Example (cont.)

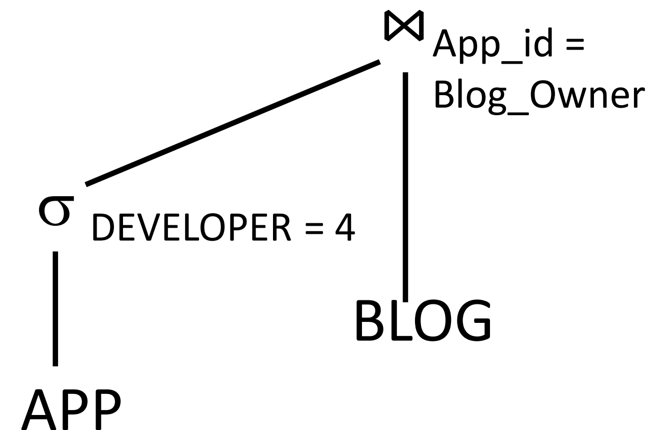
- Using a relational language (SQL)
  - *Selection* (σ) from table App where developer is 4
  - *Natural join* (⋈) with table BLOG

BLOG

Blog _ID	Blog_Owner	Last_Update	No_of_ posts
1	'com.facebook.orca'	13/03/2022	332
2	'com.viber.voip'	15/03/2022	10003
3	'com.whatsapp'	17/03/2022	7

APP

App_id	Name	Developer
'com.facebook.orca'	'Messenger'	4
'com.viber.voip'	'Viber Messenger'	6
'com.whatsapp'	'WhatsApp Messenger'	5



Example (cont.)

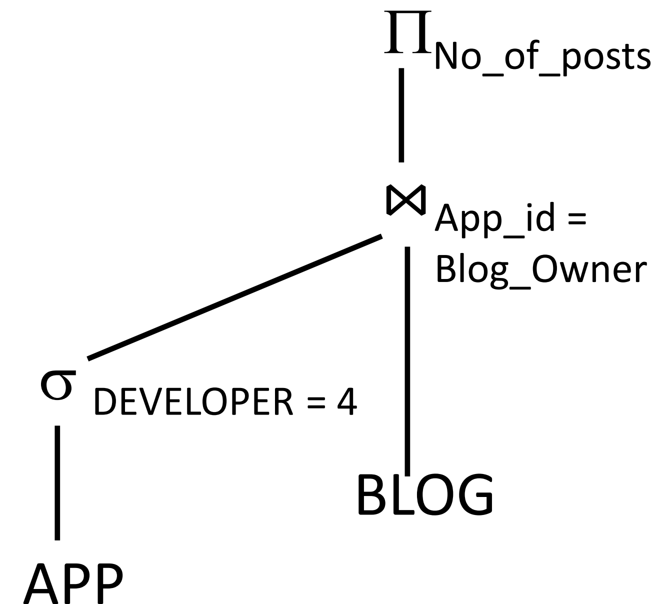
- Using a relational language (SQL)
  - *Selection* (σ) from table App where developer is 4
  - *Natural join* (⋈) with table BLOG
  - *Projection* (Π) on result

BLOG

Blog_ID	Blog_Owner	Last_Update	No_of_posts
1	'com.facebook.orca'	13/03/2022	332
2	'com.viber.voip'	15/03/2022	10003
3	'com.whatsapp'	17/03/2022	7

APP

App_id	Name	Developer
'com.facebook.orca'	'Messenger'	4
'com.viber.voip'	'Viber Messenger'	6
'com.whatsapp'	'WhatsApp Messenger'	5



Example (cont.)

- Using key-value store
  - ‘What is the number of posts in the APP blog administered by developer with code 4’?
    - Direct access to the required field (programmatically)

Key	Value
1	<'com.facebook.orca’, 'Messenger’, 4, 13/03/2022, 332>
2	<'com.viber.voip’, 'Viber Messenger’, 6, 15/03/2022, 10003>
3	<'com.whatsapp', 'WhatsApp Messenger’, 5, 17/03/2022, 7>

Another example (cont.)

- Let us assume the table

User_ID	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

- Key-Value pairs

Key-value

User:Demos:Firstname:Demosthenes
User:Demos:Lastname:Akoumianakis
User:Demos:email:da@hmu.gr
User:George:Firstname:Georgios
User:George:Lastname:Papadakis
User:George :email:gp@nowhere.com

Another example (cont.)

- The key

User_ID	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

User:Demos:Firstname:Demosthenes
User:Demos:Lastname:Akoumianakis
User:Demos:email:da@hmu.gr
User:George:Firstname:Georgios
User:George:Lastname:Papadakis
User:George :email:gp@nowhere.com

Another example (cont.)

- The value

User_ID	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

User:Demos:Firstname:**Demosthenes**
User:Demos:Lastname:Akoumianakis
User:Demos:email:da@hmu.gr
User:George:Firstname:Georgios
User:George:Lastname:Papadakis
User:George :email:gp@nowhere.com

Another example (cont.)

- (Relational) tuple vs. key-values

User_ID	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

Key-value

User:Demos:Firstname:Demosthenes
User:Demos:Lastname:Akoumianakis
User:Demos:email:da@hmu.gr
User:George:Firstname:Georgios
User:George:Lastname:Papadakis
User:George :email:gp@nowhere.com

Another example (cont.)

- In other words

User_ID	Firstname	Lastname	email
Demos	Demosthenes	Akoumianakis	da@hmu.gr
George	Georgios	Papadakis	gp@nowhere.com

Key-value

User:Demos:Firstname:Demosthenes
User:Demos:Lastname:Akoumianakis
User:Demos:email:da@hmu.gr
User:George:Firstname:Georgios
User:George:Lastname:Papadakis
User:George :email:gp@nowhere.com

Exercise in class

- Μετατροπή σε K-V

EmpID	EmpName	EmpAddress	EmpBdate
100	Laika Al-Mamari	Al Batinah, Sohar	1/21/1980
101	Khalid Al-Ameri	Al Aqur, Shinas	2/3/1990
102	Ranya Al-Balushi	Al Mutaqa, Sohar	5/25/1985

Employee:100:EmpName = “Laika Al-Mamari”
Employee:100:EmpAddress = “Al Batinah, Sohar”
Employee:100:EmpBdate= 1/21/1980

Employee:101:EmpName = “Khalid Al-Ameri”
Employee:101:EmpAddress = “Al Aqur, Shinas”
Employee:101:EmpBdate= 2/3/1990

Employee:102:EmpName = “Ranya Al-Balushi”
Employee:102:EmpAddress = “Al Mutaqa, Sohar”
Employee:102:EmpBdate= 5/25/1985

Exercise in class

- Το σχεσιακό σχήμα (users – orders)

ID	FName	LName	DOB	city	state	country	pin
111	ABC	XYZ	01/01/2000	BANGALORE	KA	India	001
112	EFG	HIJ	21/01/2000	BANGALORE	KA	India	001

ID	UserID	Product	Amount
200	111	Mobile	1000
201	112	Laptop	5000
203	112	Keyboard	500

Exercise in class

- Key-value store (users – orders)

ID	111
FName	ABC
LName	XYZ
city	BANGALORE
state	KA
country	India

ID	111
FName	EFG
city	BANGALORE
state	KA
country	India

ID	200
UserId	111
Product	Mobile
Amount	1000

ID	201
UserId	112
Product	Laptop
Amount	5000

ID	203
UserId	112
Product	keyborad
Amount	500

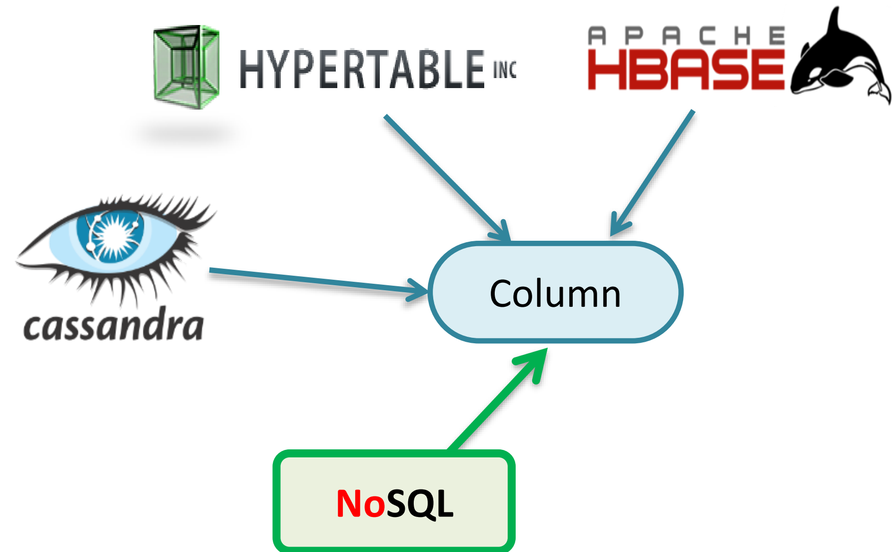
API σε Key-Value stores

- Operations σε key-value stores
  - put (key, value)
    - Προσδιορίζει το κλειδί σε μια τιμή
  - get(key)
    - Ανακτά την τιμή ενός κλειδιού
  - multiGet(key1, keyN)
    - Ανακτά τιμές των κλειδιών 1 μέχρι N
  - delete(key)
    - Διαγράφει την τιμή ενός κλειδού

Key Value Stores – Consolidation

- Positive aspects
  - Simple model
  - Scalability (εύκολη κλιμάκωση)
- Negative aspects
  - Problems with complex data

Column family data stores

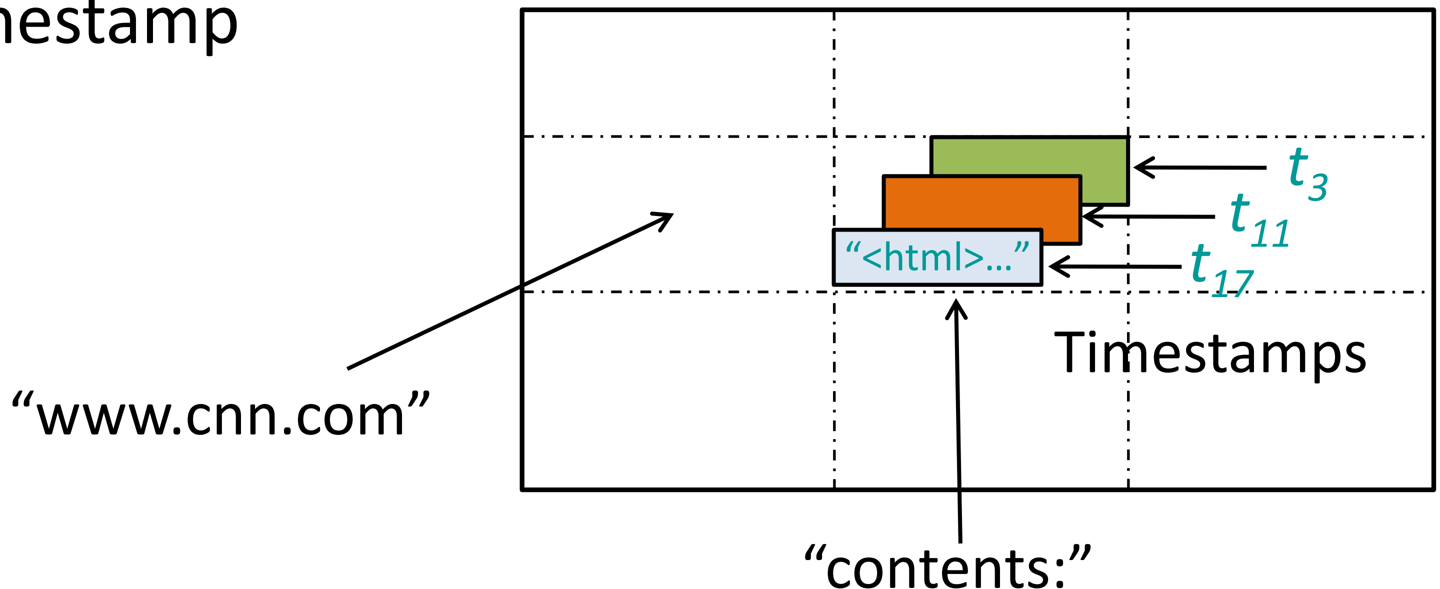


Indicative systems in this cluster

- Παραδείγματα
  - Bigtable [Google]
  - Cassandra [Cassandra]
  - HBase [Hbase]
  - Hypertable[Hypertable],
  - Amazon SimpleDB [Amazon SimpleDB]

Data model

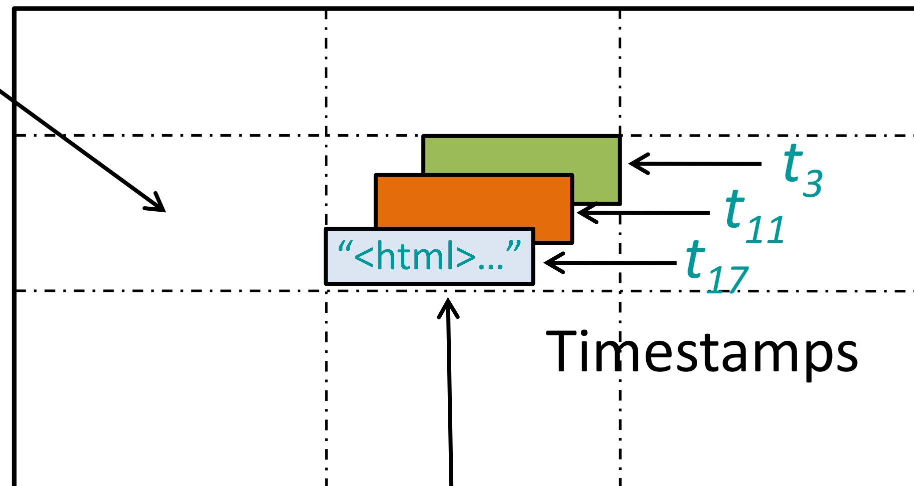
- Πρόκειται για ζεύγος κλειδιού-τιμής (key-value pair) που απαρτίζεται από τρία συστατικά
  - Μοναδικό όνομα για αναφορά στη στήλη
  - Τιμή της στήλης
  - timestamp



Data model

- Κάθε cell αναγνωρίζεται από ένα *key*, μια *τιμή* και ένα *timestampt*

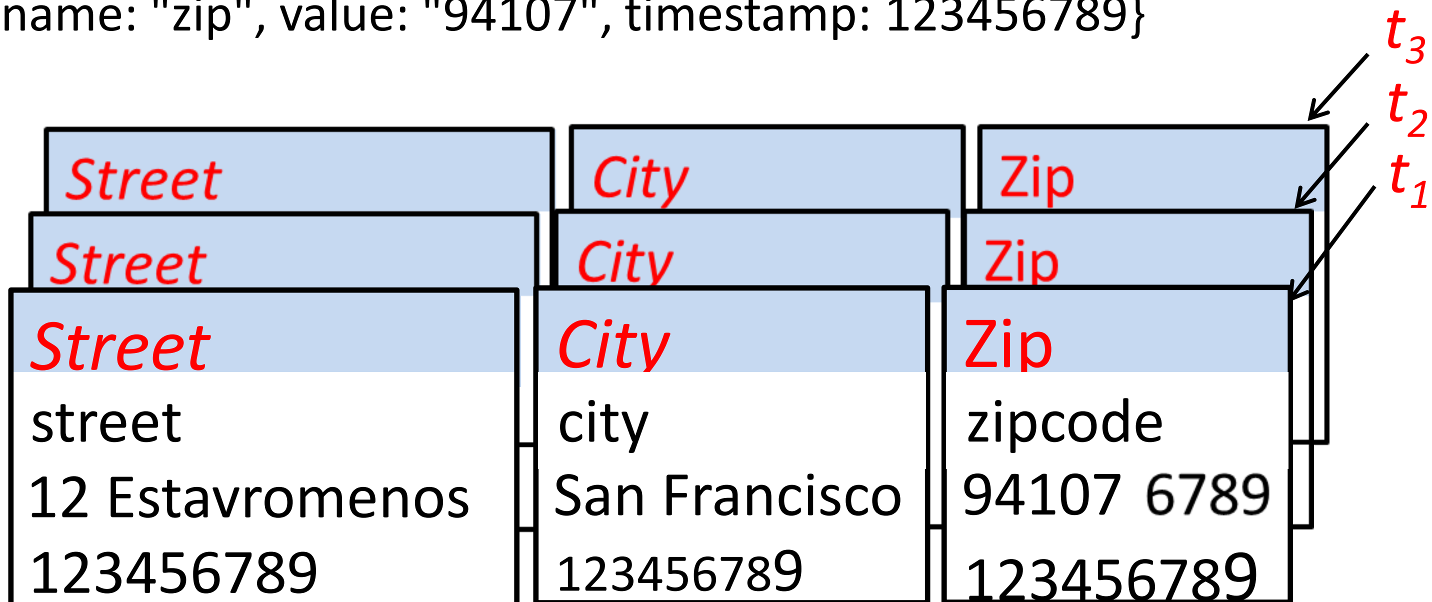
(k, V, t)



Example

- Κάθε cell αναγνωρίζεται από ένα key, μια τιμή και ένα timestampt

```
{
street : {name: "street", value: "1234 x street", timestamp: 123456789},
   city : {name: "city", value: "san francisco", timestamp: 123456789},
   zip  : {name: "zip", value: "94107", timestamp: 123456789}
}
```



Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage
  - Οι στήλες μετατρέπονται σε γραμμές τους πίνακα

Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage

1X2B	2X2B	3X3Y

Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage

1X2B	2X2B	3X3Y
1234	3456	

Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage

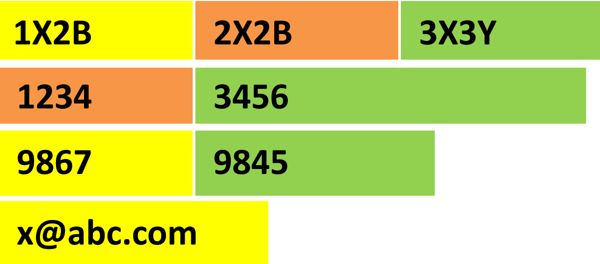
1X2B	**2X2B**	**3X3Y**
1234	**3456**	
9867	**9845**	

Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage

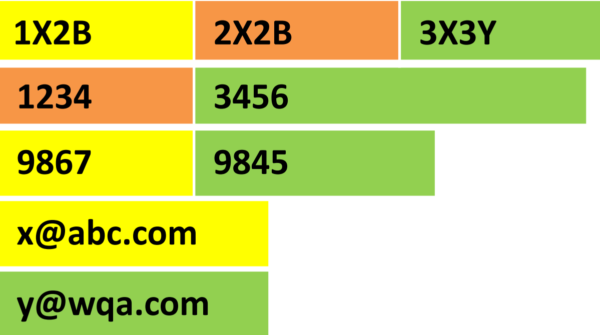


Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage



Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage

1X2B	**2X2B**	**3X3Y**
1234	**3456**	
9867	**9845**	
x@abc.com		
y@wqa.com		
a@fb.com		

Example – A user profile

- Row-oriented storage

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Column-oriented storage

1X2B	**2X2B**	**3X3Y**
1234	**3456**	
9867	**9845**	
x@abc.com		
y@wqa.com		
a@fb.com		
#bigtable	**#handoop**	

Example (BigTable)

- Για το χρήστη 1X2B

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Τα δεδομένα που αποθηκεύονται για το 1X2B
  - NULL τιμές αγνοούνται

RowKey	Column Values	
1234	ph:cell=19867	email:1=x@abc.com

Example (BigTable)

- Παρόμοια για τον χρήστη 2X2B

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

- Τα δεδομένα που αποθηκεύονται για 2X2B
  - NULL τιμές αγνοούνται

RowKey	Column Values	
1234	ph:cell=19867	email:1=x@abc.com
3678	ph:home=1234	social:twitter=#bigtable

Example (BigTable)

- Τέλος για το χρήστη 3X3Y

ContactID	Home-Phone	Cell-Phone	Email1	Email2	Facebook	Twitter
1X2B	NULL	9867	x@abc.com	NULL	NULL	NULL
2X2B	1234	NULL	NULL	NULL	NULL	#bigtable
3X3Y	3456	9845	NULL	y@wqa.com	a@fb.com	#hadoop

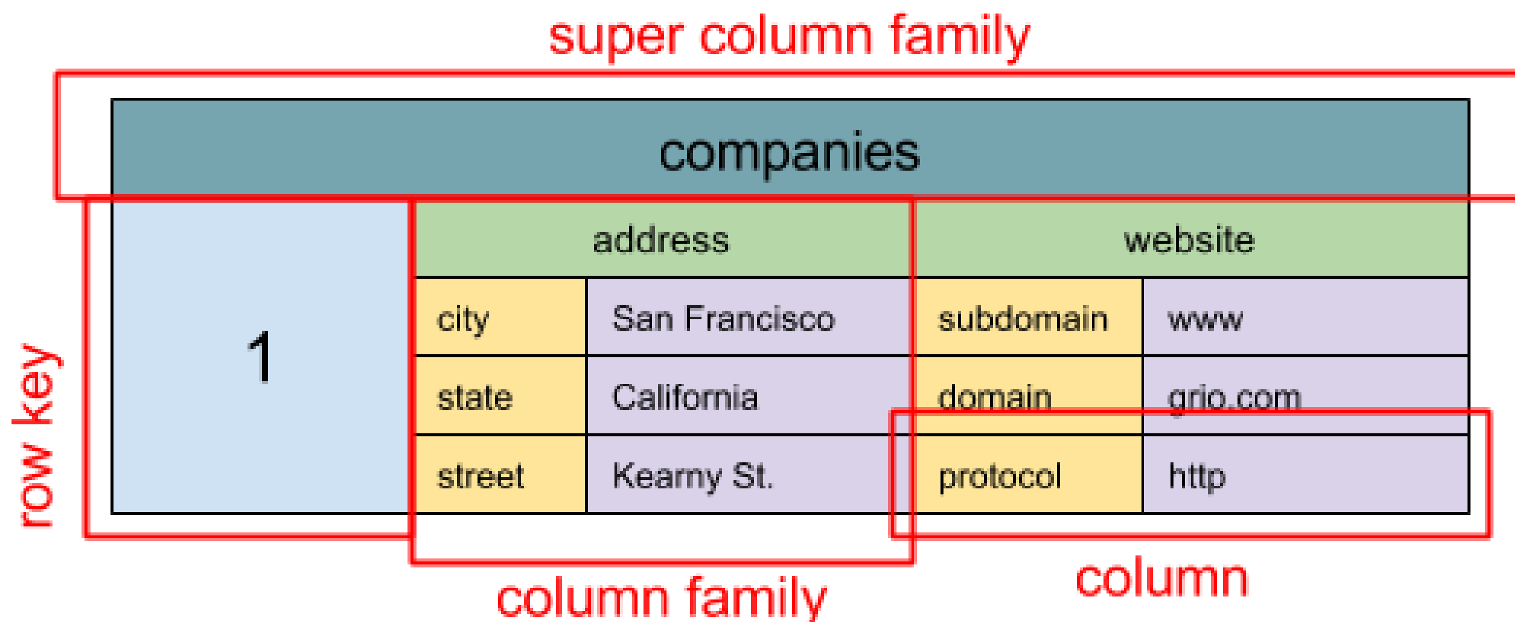
- Τα δεδομένα που αποθηκεύονται για 3X3Y
  - NULL τιμές αγνοούνται

RowKey	Column Values				
1234	ph:cell=19867	email:1=x@abc.com			
3678	ph:home=1234	social:twitter=#bigtable			
5987	ph:home=3456	ph.cell=9845	email:2=y@wqa.com	social:facebook=a@fb.com	social:twitter=#hadoop

Column-family data stores

Παράδειγμα

- Families of similar columns



Παράδειγμα

- Table with single-row partitions

columns

partition key

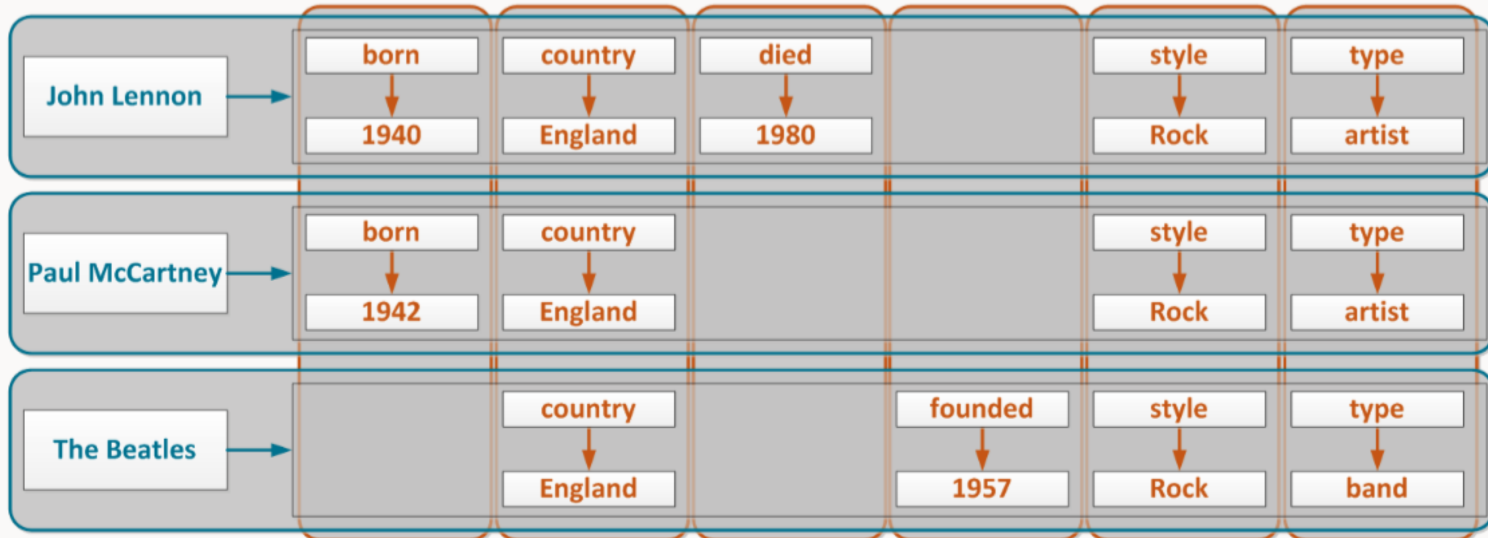
performer	born	country	died	founded	style	type
John Lennon	1940	England	1980		Rock	artist
Paul McCartney	1942	England			Rock	artist
The Beatles		England		1957	Rock	band

partitions

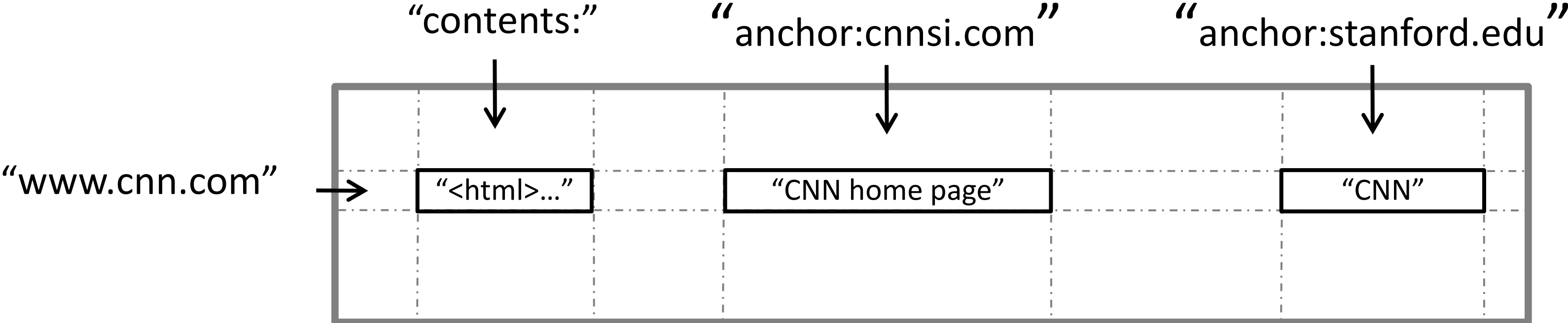
rows

cells

- Column family view



Παράδειγμα



Column Family: contents

Row Key	Timestamp	Qualifier	Value
com.cnn.www	t3	html	"<html>..."
com.cnn.www	t5	html	"<html>..."
com.cnn.www	t6	html	"<html>..."
com.example.www	t5	html	"<html>..."

Column Family: anchor

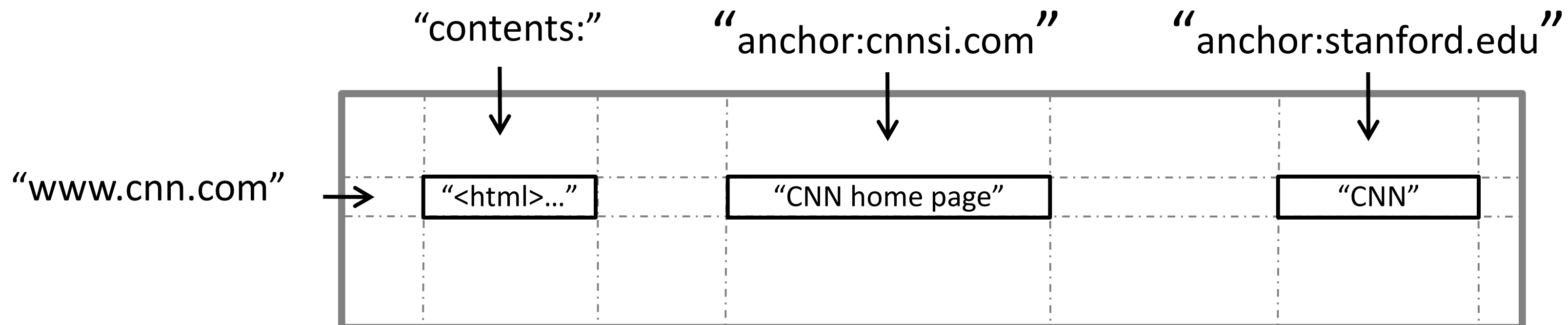
Row Key	Timestamp	Qualifier	Value
com.cnn.www	t8	cnnsi.com	"CNN"
com.cnn.www	t5	my.look.ca	"CNN.com"

Column Families

- Οικογένειες από συναφής στήλες
  - Πρέπει αν οριστούν πριν χρησιμοποιηθούν για καταχώρηση δεδομένων
  - Καλή πρακτική να υπάρχει μικρός αριθμός column families
  - Columns έχουν ονοματολογία δύο επιπέδων
    - family:optional_qualifier
- Column family
  - Μονάδα ελέγχου πρόσβασης
  - Συνηθίζεται να έχει τύπο δεδομένων

Rows

- Όνομα στήλης είναι τύπου string
  - Πρόσβαση σε data της γραμμής είναι ατομική (atomic)
  - Η δημιουργία μιας γραμμής επιτυγχάνεται έμμεσα με την αποθήκευση δεδομένων
- Rows ordered lexicographically



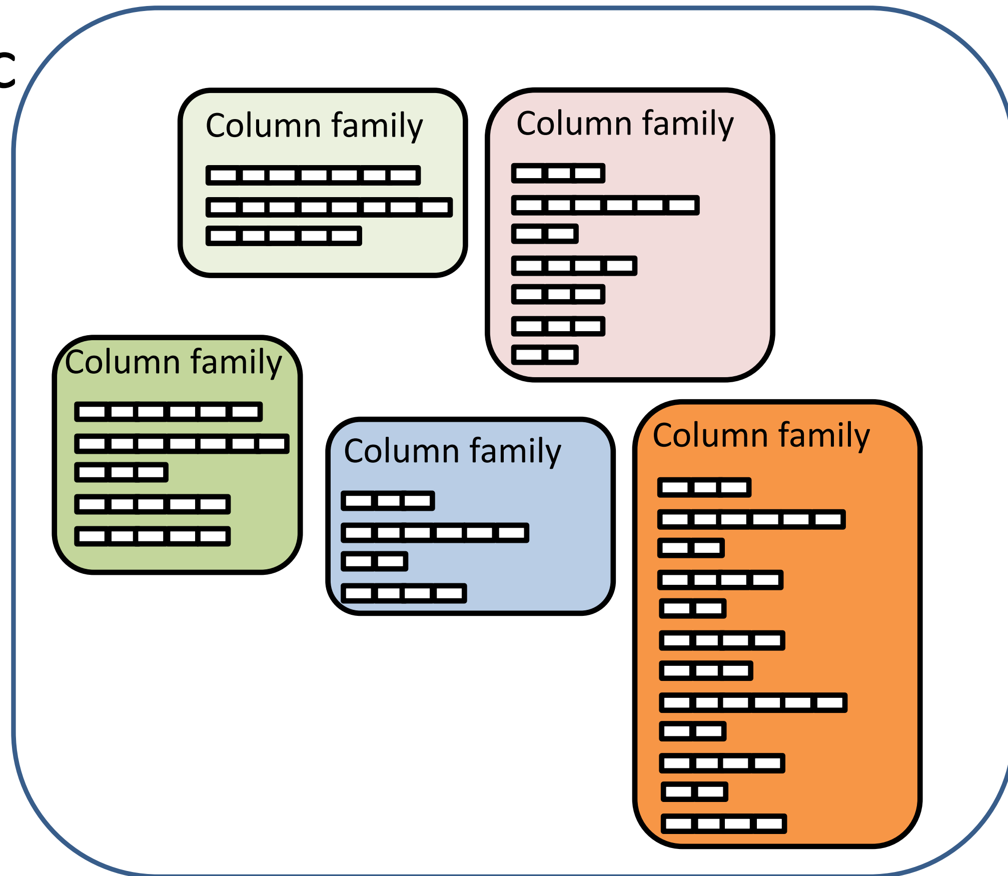
Timestamps

- Χρησιμοποιούνται για την καταχώρηση διαφορετικών εκδόσεων δεδομένων σε ένα cell
  - Νέες εγγραφές έχουν ως default την τρέχουσα χρονική στιγμή
  - Timestamps μπορούν να προσδιοριστούν από εφαρμογές
- Τεχνικές διαχείρισης ‘σκουπιδιών’
  - Καθορίζονται per-column-family settings
    - *“Only retain most recent K values in a cell”*
    - *“Keep values until they are older than K seconds”*

Keyspace

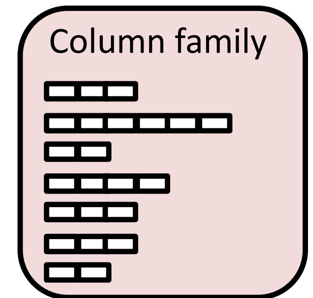
- Ένα keyspace περιλαμβάνει όλες τις οικογένειες στηλών

Keyspac
e



Column family

- ✓ *Ένα keyspace περιλαμβάνει όλες τις οικογένειες στηλών*
- Μια οικογένεια στηλών περιλαμβάνει πολλαπλές γραμμές

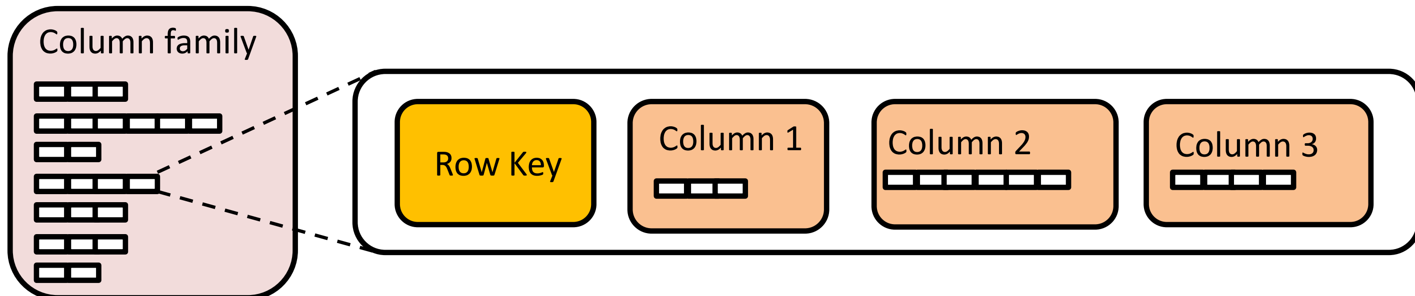


Rows

✓ *Ένα keyspace περιλαμβάνει όλες τις οικογένειες στηλών*

✓ *Μια οικογένεια στηλών περιλαμβάνει πολλαπλές γραμμές*

- Κάθε γραμμή περιλαμβάνει διαφορετικό αριθμό στηλών ενώ στήλες σε μια γραμμή μπορούν να διαφέρουν από αυτές μιας άλλης



Columns

- ✓ *Ένα keyspace περιλαμβάνει όλες τις οικογένειες στηλών*
- ✓ *Μια οικογένεια στηλών περιλαμβάνει πολλαπλές γραμμές*
- ✓ *Κάθε γραμμή περιλαμβάνει διαφορετικό αριθμό στηλών ενώ στήλες σε μια γραμμή μπορούν να διαφέρουν από αυτές μιας άλλης*
- Κάθε στήλη περιορίζεται στη γραμμή της και περιλαμβάνει ένα ζεύγος name/valueκαι ένα timestamp

Column
Name
Value
Timestamp

Column Family: Pros and Cons

- Pros:
  - Supports Simi-Structured Data
  - Naturally Indexed (columns)
  - Scalable
- Cons
  - Poor for interconnected data