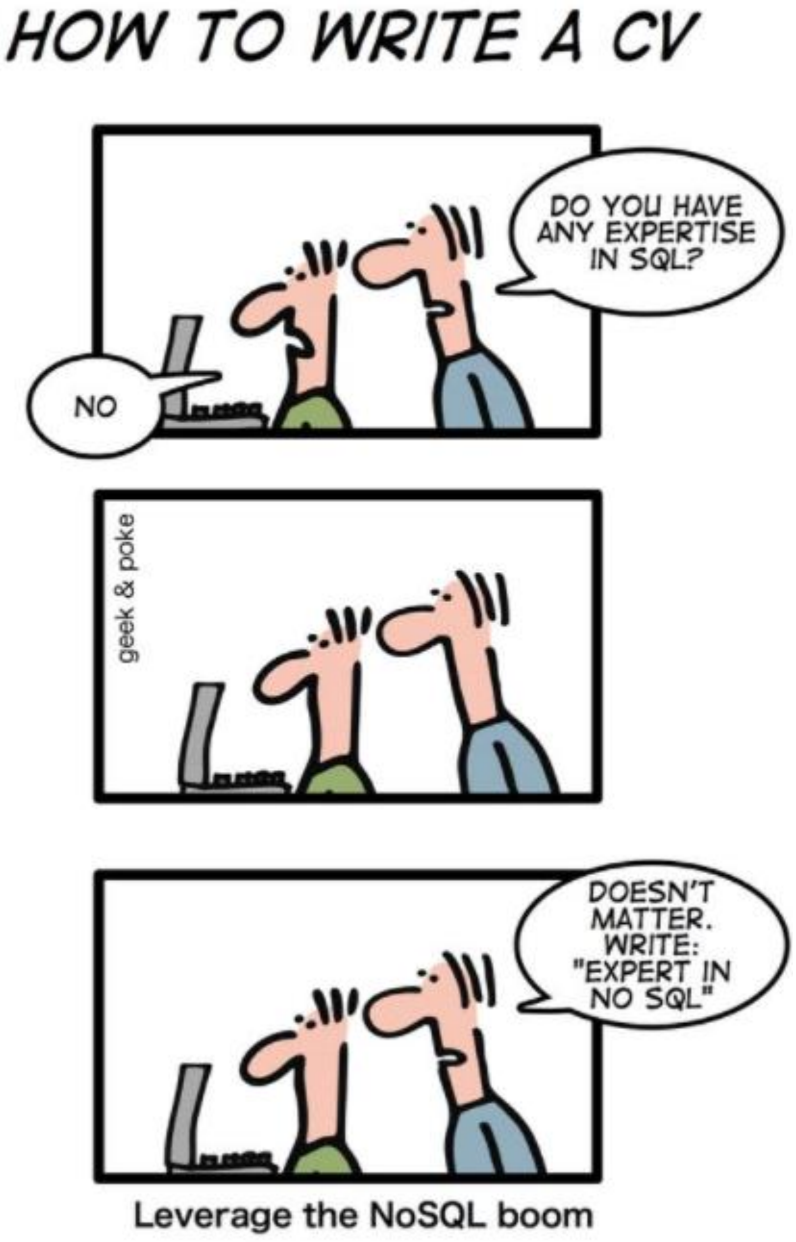
NoSQL Database Management Systems

Amos Azaria, Netanel Chkroun

NoSQL

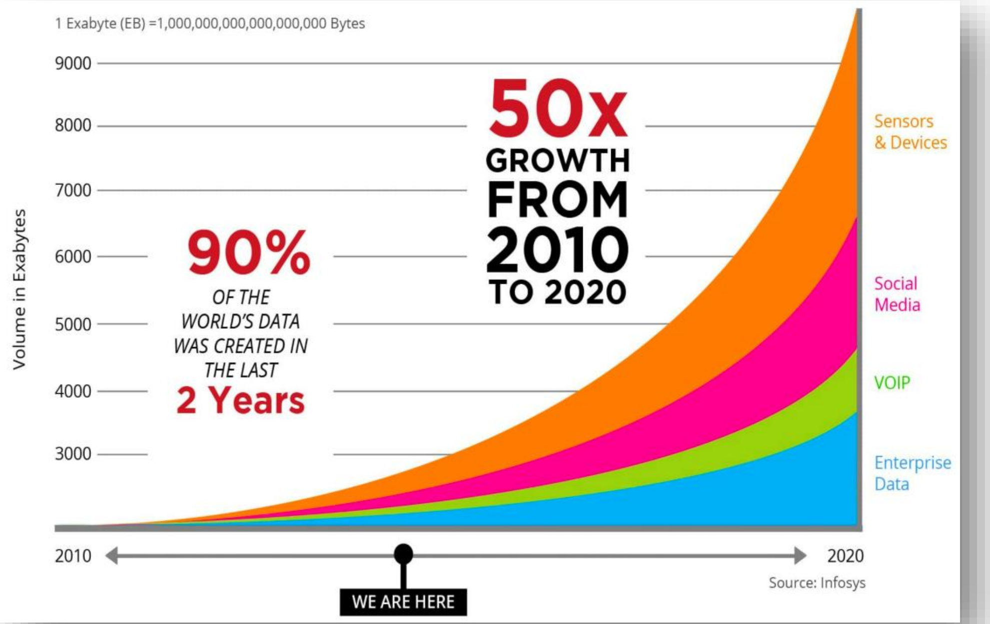

HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom

The world has changed

The world has changed

More connected

More data



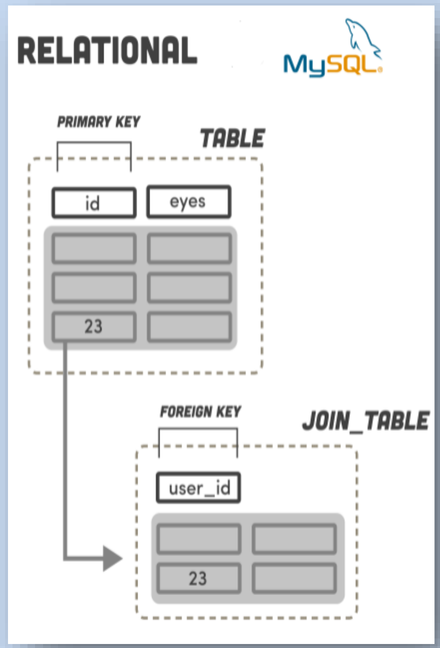
Less tolerant

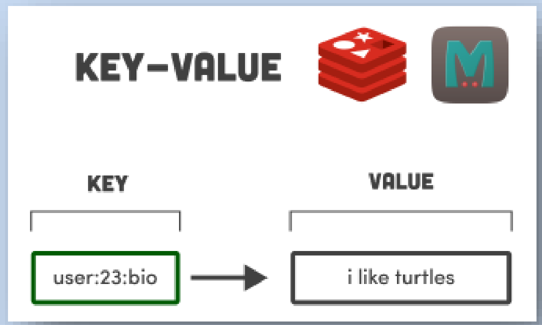
NoSQL

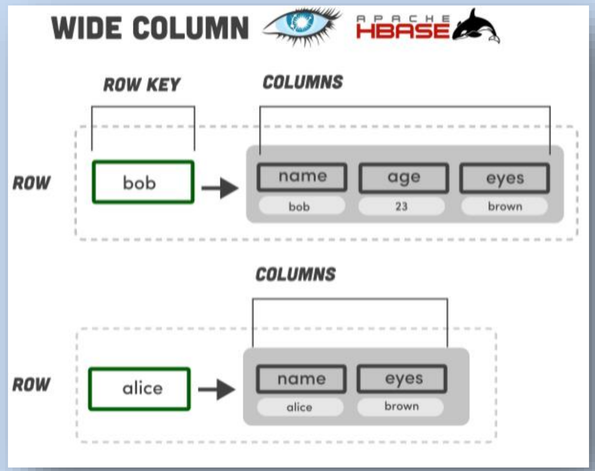
- No SQL, also known as 'not only SQL', refers in general to database management systems that do not rely on the relational (table) data storage.
- Usually avoid joins and have a more relaxed definition of consistency.
- More flexible (usually attributes can be added on the fly).
- Support big data!

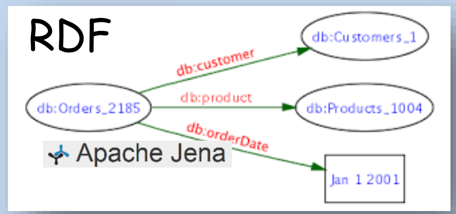
NO SQL

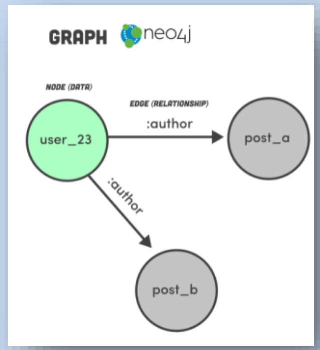
SQL

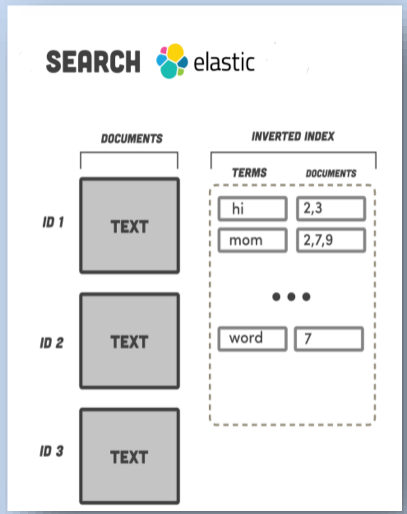


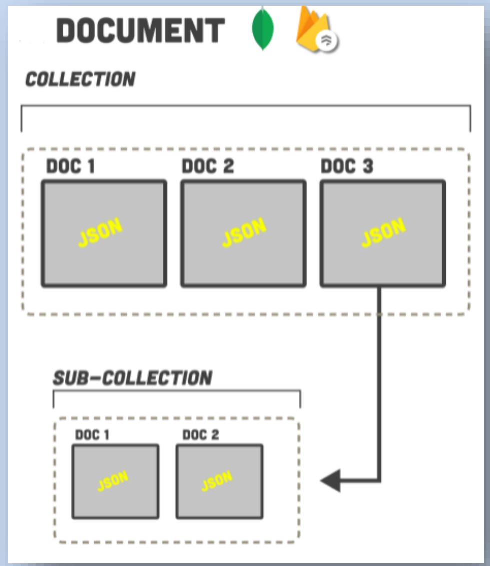








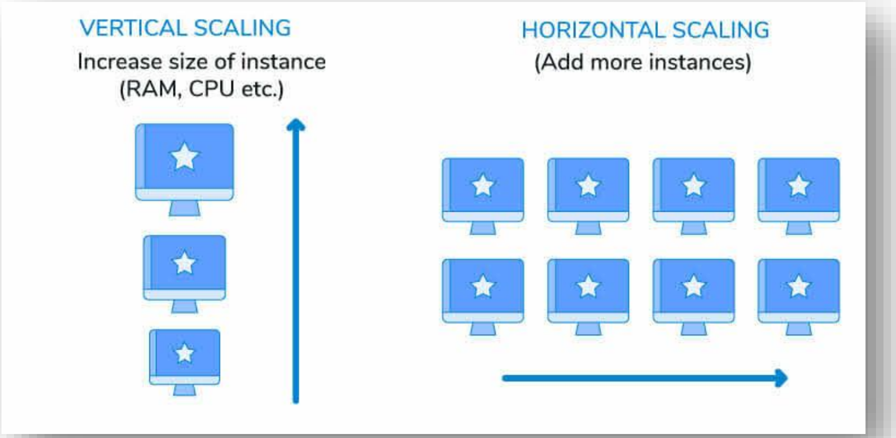




Credit :https://fireship.io/lessons/top-seven-database-paradigms/

Why use NoSQL?

- Scalability!
- Too much data storage for allowing a single controller.
- Structure may change over time.
- Fast perfomance



What is the Price-tag?

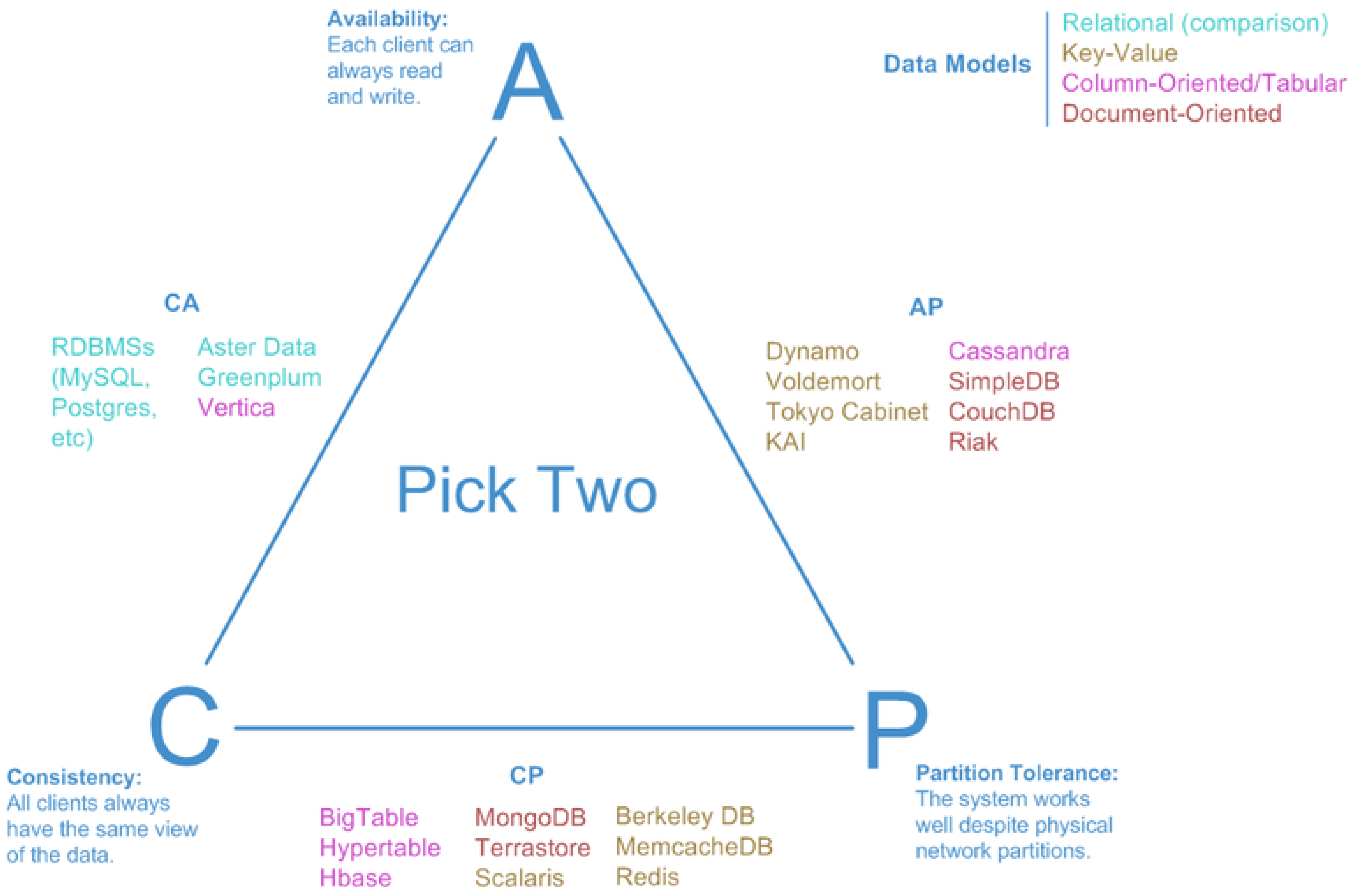
- Limited query capabilities (usually avoiding joins.)
- Usually can't ensure all ACID (Atomicity, Consistency, Isolation, and Durability).
  - Usually support BASE (Basically Available, soft State, eventual consistency).
- Not standardized.

ACID Vs **BASE**

- **B**asically **A**vailable: data is mostly available.
- soft **S**tate: state may change even with no updates (since older updates are still propagating).
- **E**ventual consistency: if we let the data propagate enough time, it will become consistent.

CAP theorem

- It is impossible for a distributed computer system to simultaneously provide more than two out of three of the following guarantees:
  - **C**onsistency: Every read receives either the most recent written value or an error.
  - **A**vailability: Every request receives a response (not necessarily the most recent value).
  - **P**artition tolerance: The system continues to operate despite any number of messages being dropped (or delayed) by the network between nodes.



Credit: http://blog.nahurst.com/visual-guide-to-nosql-systems

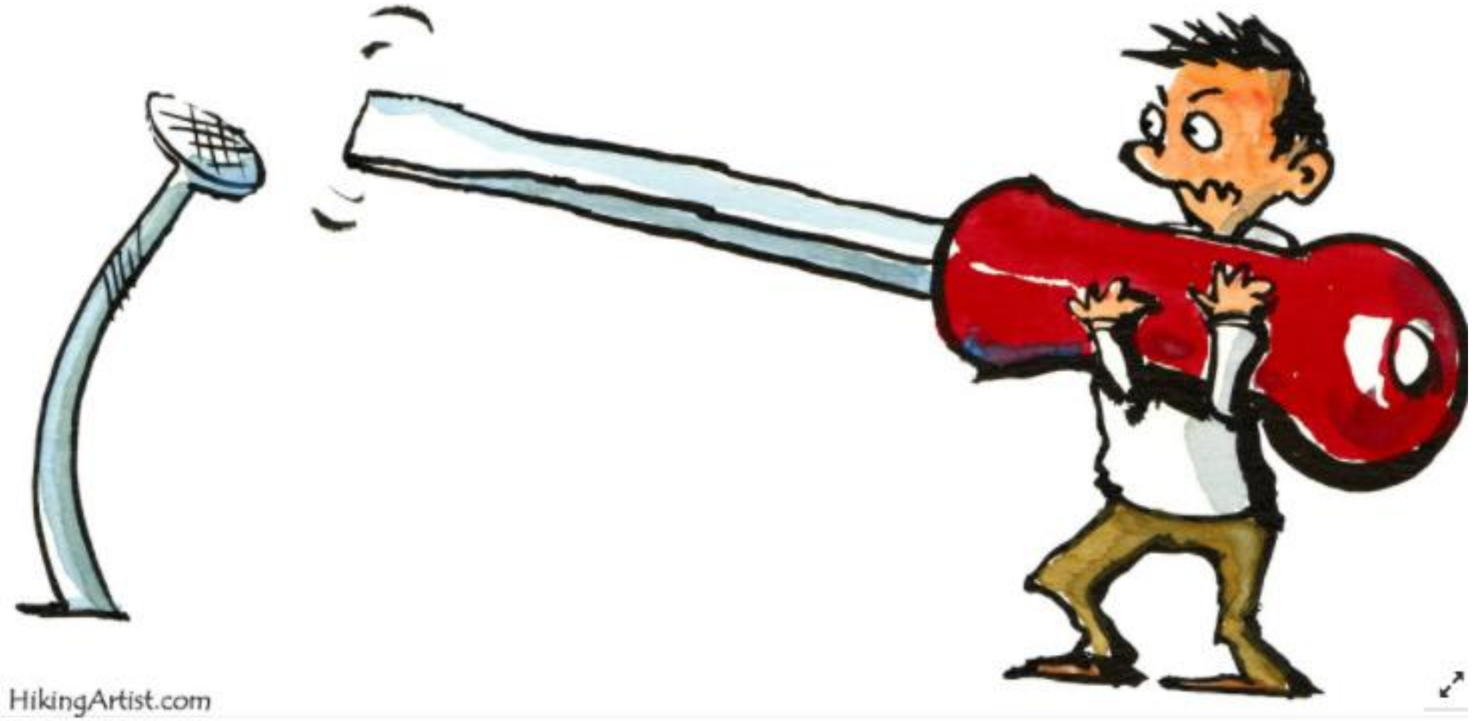
Database management system usage (http://db-engines.com/en/ranking)

The leading RDBMS have started to add support for additional models

Rank			DBMS				
Apr 2020	Mar 2020	Apr 2019					2019
1.	1.	1.	Oracle	Relational, Multi-model			5.48
2.	2.	2.	MySQL	Relational, Multi-model			3.21
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model			3.47
4.	4.	4.	PostgreSQL	Relational, Multi-model			1.14
5.	5.	5.	MongoDB	Document, Multi-model	438.43	+0.82	+36.45
6.	6.	6.	IBM Db2	Relational, Multi-model	165.63	+3.07	-10.42
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	148.91	-0.26	+2.91
8.	8.	↓ 7.	Redis	Key-value, Multi-model	144.81	-2.77	-1.57
9.	↑ 10.	↑ 10.	SQLite	Relational	122.19	+0.24	-2.02
10.	↓ 9.	↓ 9.	Microsoft Access	Relational	121.92	-3.22	-22.73
11.	11.	11.	Cassandra	Wide column	120.07	-0.88	-3.54
12.	↑ 13.	12.	MariaDB	Relational, Multi-model	89.90	+1.55	+4.67
13.	↓ 12.	13.	Splunk	Search engine	88.08	-0.44	+4.99
14.	14.	↑ 15.	Hive	Relational	84.05	-1.32	+9.34
15.	15.	↓ 14.	Teradata	Relational, Multi-model	76.59	-1.25	+1.25
16.	16.	↑ 19.	Amazon DynamoDB	Multi-model	64.27	+1.75	+8.26
17.	17.	↓ 16.	Solr	Search engine	53.59	-1.50	-6.64
18.	18.	↑ 21.	SAP HANA	Relational, Multi-model	53.29	-0.98	-2.05
19.	↑ 20.	↑ 20.	SAP Adaptive Server	Relational	52.63	-0.14	-3.17
20.	↓ 19.	↓ 18.	FileMaker	Relational	52.08	-2.08	-6.34

Relational DBMS,
Document store,
Graph DBMS,
RDF store

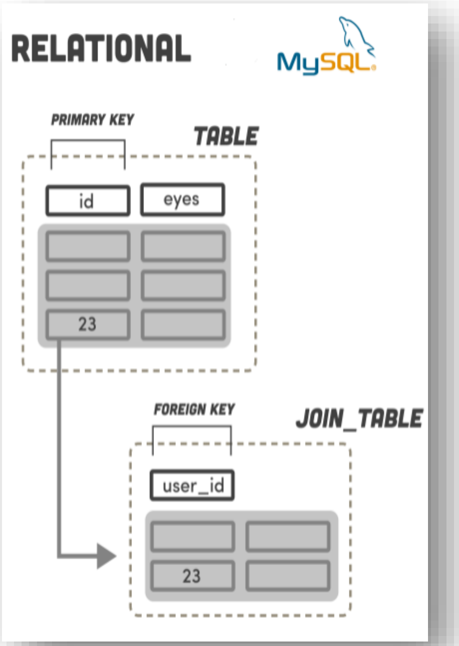
SQL or NoSQL ?

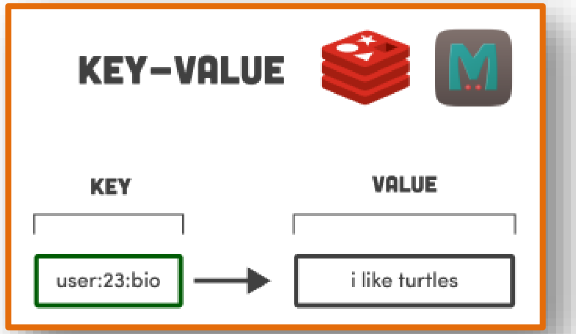


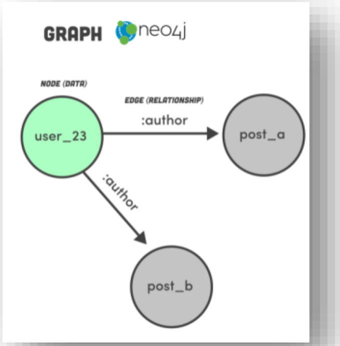
Use the right tool for the job and not vice versa

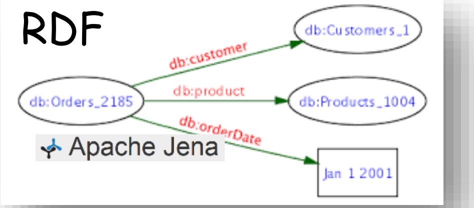
NO SQL

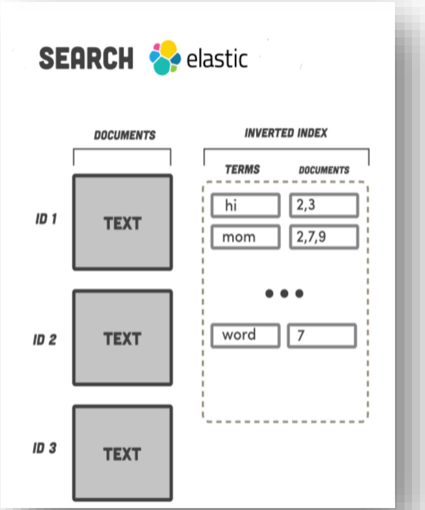
SQL

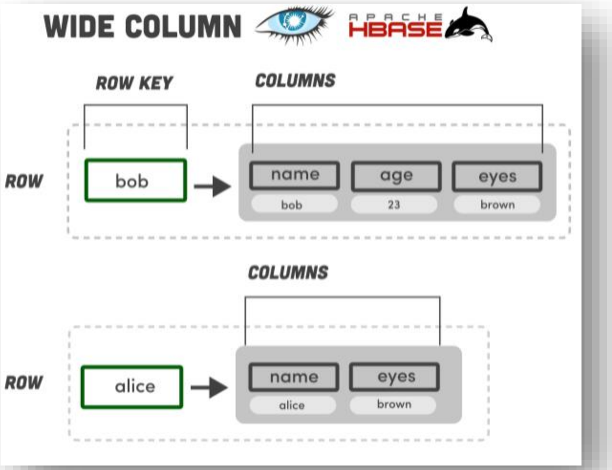


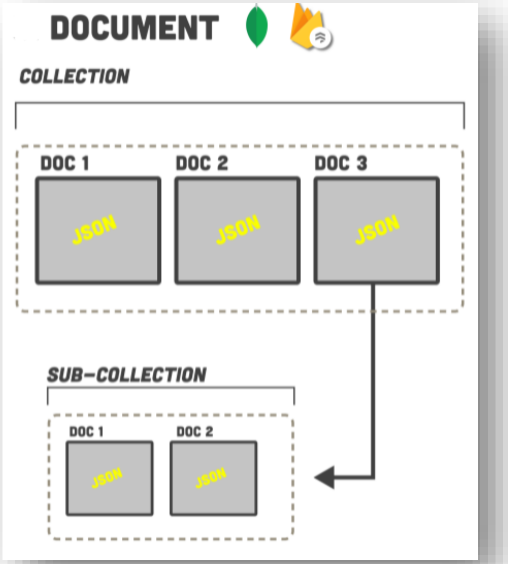






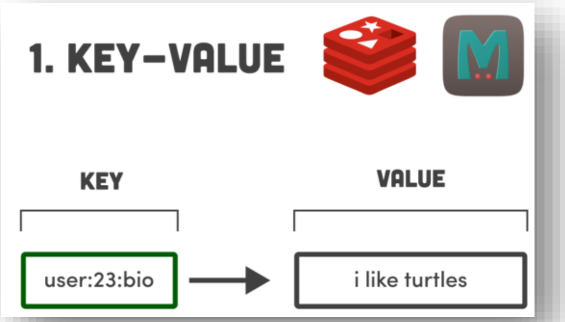






Credit :https://fireship.io/lessons/top-seven-database-paradigms/

Key-Value Store



- Every item in the database is stored as an attribute name (or "key") together with its value.
- No further structure is imposed (values are opaque).
- Usually can only query by key (no features related to any structure are available).
- Our example: Redis (next slides). You can try it online at: https://try.redis.io/. Case insensitive.

SET, GET, DEL

- SET hello "world"

OK

- GET hello

"world"

- GET "world"

(nil)

- SET name1 Yossi
- GET name1

"Yossi"

- SET user:1001"<user><first_name>Joel</first_name>
  <last_name>Cohen</last_name></user>"
- GET user:1001

"<user><first_name>Joel</first_name><last_name>Cohen</last_name></user>"

- DEL user:1001

We usually store the id as part of the key, so we can fetch it easily

The value can be anything (e.g. XML)

INCR and INCRBY

- SET bottles 5
- INCR bottles

(integer) 6

- INCRBY bottles -3

(integer) 3

- INCR name1

(error) ERR value is not an integer or out of range

INCR x is equivalent to x++ (in JAVA), and is guaranteed to be atomic

INCRBY x y is equivalent to x+=y (in JAVA).

Lists (RPUSH, LPUSH, LRANGE)

- RPUSH user:571:items "chair"
- RPUSH user:571:items "table"
- RPUSH user:571:items "water"
- LPUSH user:571:items "cup"
- LRANGE user:571:items 1 2

1) "chair"
2) "table"

- LRANGE user:571:items 0 -1

1) "cup"
2) "chair"
3) "table"
4) "water"

- RPUSH user:573:items "bed" "chair" "table" "can"

RPUSH (Right Push) puts the new item last (on the very right)

LPUSH (Left Push) puts the new item first (on the very left)

LRANGE key x y
Returns items from x to y

-1 means until the end of the list

You can provide the full list in a single command

Hashes(HSET, HGET, HMSET, HGETALL)

- HSET user:302 first_name "Tamar"
- HSET user:302 last_name "Cohen"
- HGET user:302 first_name

"Tamar"

- HMSET user:302 degree 3 gender "female"
- HGET user:302 degree

"3"

HMSET: for setting multiple attributes in a single command

- HGETALL user:302

1) "first_name"
2) "Tamar"
3) "last_name"
4) "Cohen"
5) "degree"
6) "3"
7) "gender"
8) "female"

It looks as if HGETALL returns a list, but it really returns a set

Get KEYS with pattern

- SET user:1000 "Adam"
- SET user:1001 "Tammy"
- SET user:1002 "EVE"
- SET user:1010 "APPLE"
- RPUSH user:1003:items "chair"

- KEYS user:100?

  1) "user:1000"
  2) "user:1001"
  3) "user:1002"

- KEYS user:*

  1) "user:1000"
  2) "user:1010"
  3) "user:1003:items"
  4) "user:1001"
  5) "user:1002"

Sets and Sorted Sets

- Redis also supports sets:
  - SADD x y (add item y to set x)
  - SREM x y (remove item y from set x)
  - SISMEMBER x y (is y a member of x)
  - SUNION x q (returns the union of sets x and q)
- And Sorted sets:
  - ZADD x val y (add item y with score val to sorted set x)
  - ZRANGE x y z (return z items from x, starting at y)

EXPIRE

The EXPIRE commands sets a time in seconds to which a value will expire.

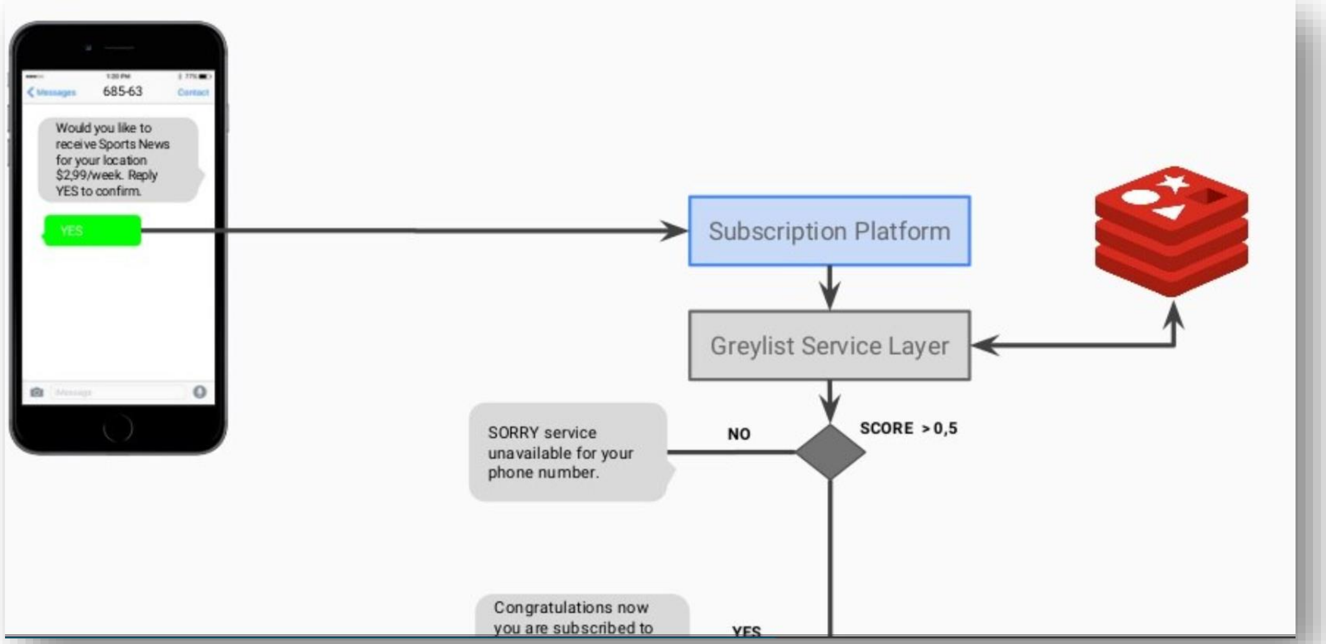
- SET resource:lock "Redis Demo"
- EXPIRE resource:lock 120
- TTL resource:lock
- (integer) 92

Redis Use Case

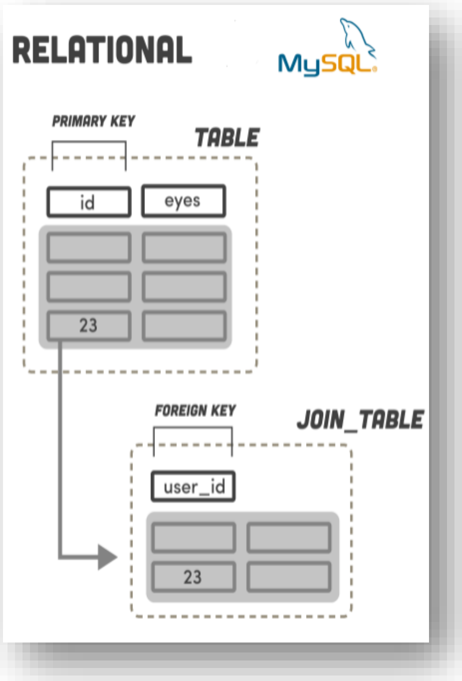
- Managing black lists, grey lists

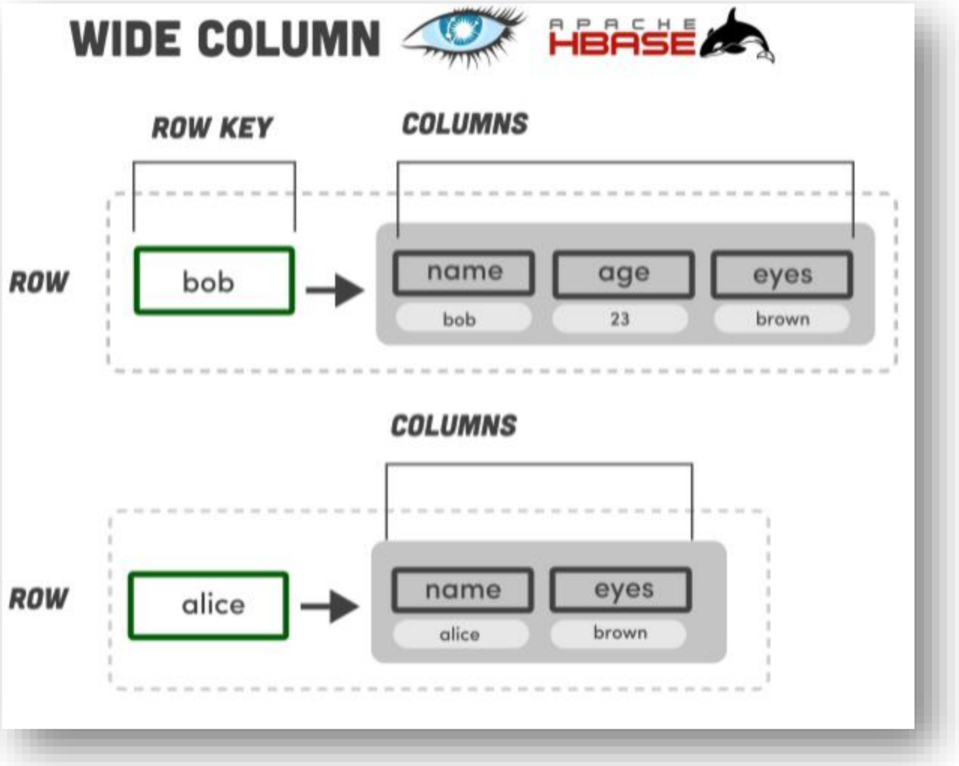
We create a score to our users based on the product they purchased etc.

GREYLIST DATA LAYER	
Phone Number: + 55 19 0000 1111	Score: 0,8
Phone Number: + 55 19 2323 0101	Score: 0,2
Phone Number: + 55 19 0202 0000	Score: 0,4



Wide-Column Store





Wide-Column Store

- Every key identifies a row of a variable number of elements.
- It can be seen as if it stores data together as tables, but the names and format of the columns can vary.
- Our example: Cassandra

Cassandra

- Used by: Facebook, Twitter, Cisco, Rackspace, eBay, Netflix
- Cassandra Query Language (CQL), seems like a simple version of SQL:
  - No Joins.
  - No nested queries (can be done in code).
- Case insensitive.
- Download from: https://academy.datastax.com/planet-cassandra//cassandra and install
- (If the service isn't running you can execute ….\bin\cassandra.bat)
- …\bin\cqlsh.bat opens the client

In Greek mythology, Cassandra was a prophet and the princess of Troy.

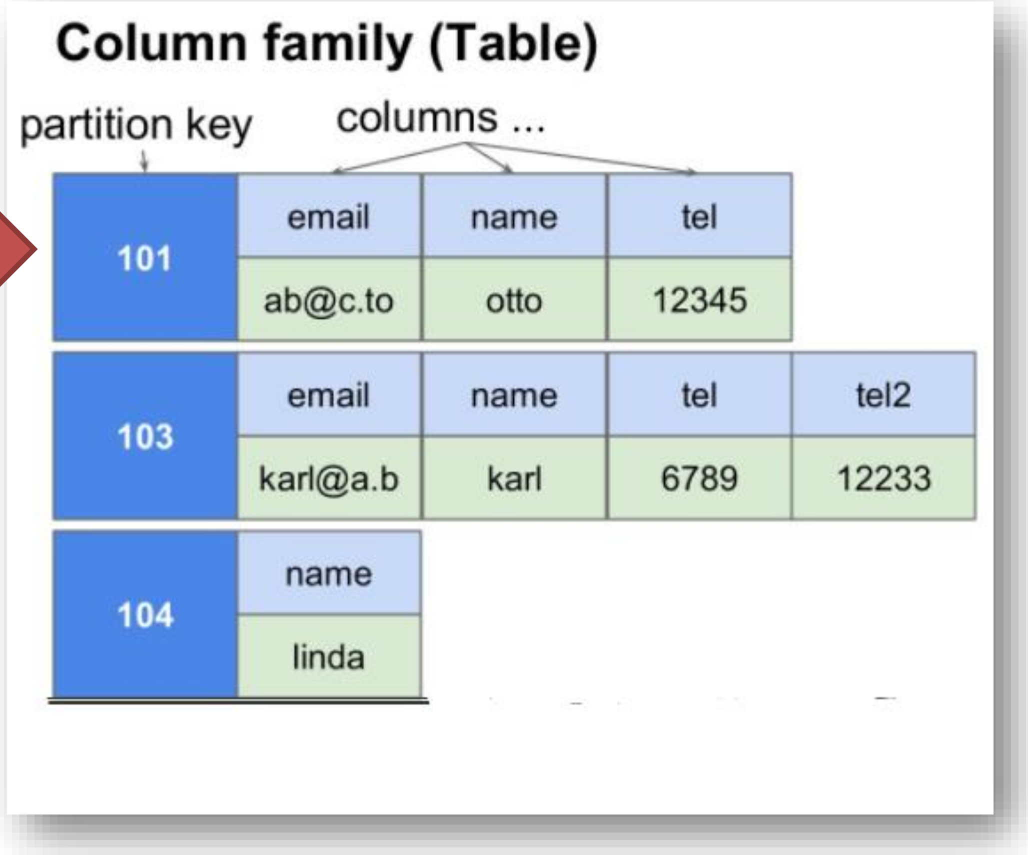
RMDB Habits

- **RMDB:**
  - Minimize the Number of Writes
  - Minimize Data Duplication
- **Cassandra:**
  - Writes in Cassandra are awfully cheap. If you can perform extra writes to improve the efficiency of your read queries, it’s almost always a good tradeoff.
  - Reads tend to be more expensive and are much more difficult to tune.
  - Denormalization and duplication of data is a fact of life with Cassandra. In order to get the most efficient reads, you often need to duplicate data. The queries define the tables, so you might create tables duplicating some data in order to address different queries.

Data Model

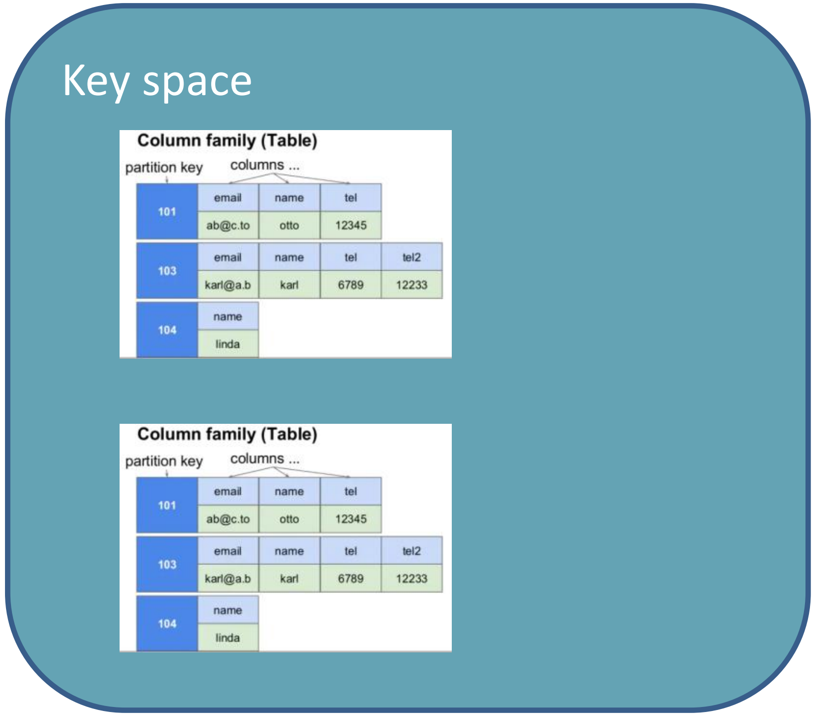
- cluster
- Keyspace (Database)
- Column Family (table)
- Keys and Column

103	email	name	tel	tel2
	karl@a.b	karl	6789	12233



Data Model

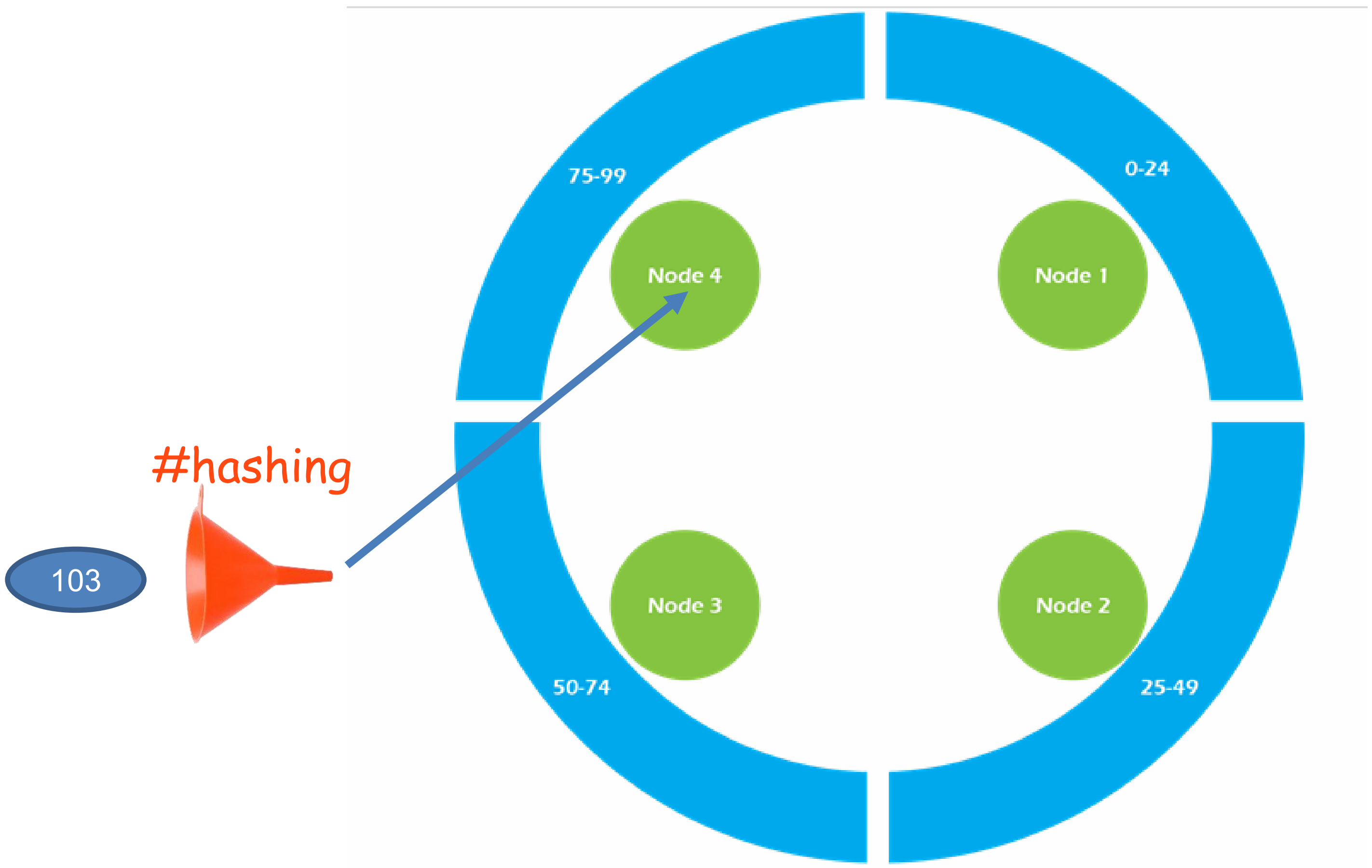
- cluster
- Keyspace (Database)
- Column Family (table)
- Keys and Column



Data Model

- cluster
- Keyspace (Database)
- Column Family (table)
- Keys and Column

103	email	name	tel	tel2
	karl@a.b	karl	6789	12233



Creating a Keyspace (database)

Replication refers to how the data is replicated across different nodes

JSON style

- CREATE KEYSPACE university WITH REPLICATION = {'class':'SimpleStrategy', 'replication_factor':2};

SimpleStrategy: Use for a single data center only. NetworkTopologyStrategy: Can expand to multiple data centers.

Number of replicas of data on multiple nodes

- USE university;
- CREATE TABLE students (id INT PRIMARY KEY, firstName VARCHAR, lastName VARCHAR, age INT);

Like SQL. But there is no need to specify the size for VARCHAR.

Cassandra's storage method

The coordinator node returns the nodes that hold all the replications of the data.

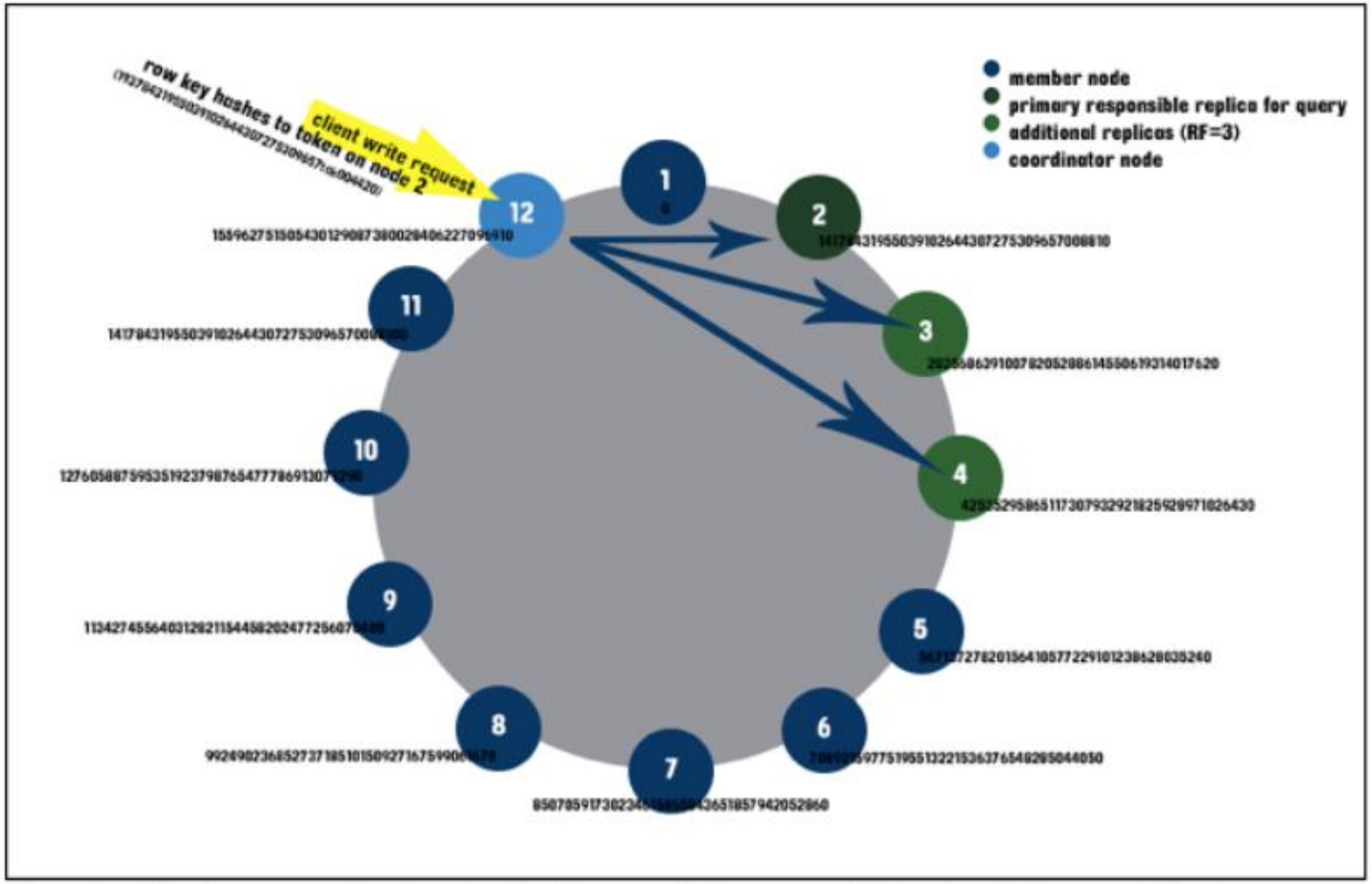


Figure 1 A 12 node cluster using RandomPartitioner and a keyspace with Replication Factor (RF) = 3, demonstrating a client making a write request at a coordinator node and showing the replicas (2, 3, 4) for the query's row key

Data is stored on sequential nodes using a ring model.

https://www.hakkalabs.co/articles/how-cassandra-stores-data cassandra

CQL | INSERT and SELECT

Exactly like SQL…

- INSERT INTO students (id, firstName, lastName, age) VALUES (111, 'Chaya', 'Glass', 21);

Must use single quotes!

- INSERT INTO students (id, firstName) values (222, 'Only First');
- SELECT * from students

```
 id  | age  | firstname  | lastname
-----+------+------------+----------
 111 |   21 |      Chaya |    Glass
 222 | null | Only First |     null
```

CQL|WHERE

WHERE on a key is ok

- SELECT * from students WHERE id=111;

```
 id  | age | firstname | lastname
-----+-----+-----------+----------
 111 |  21 |     Chaya |    Glass
```

WHERE on a non-key attribute…

- SELECT * from students WHERE lastname='Glass';

InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"

- SELECT * from students where lastname = 'Glass' ALLOW FILTERING;

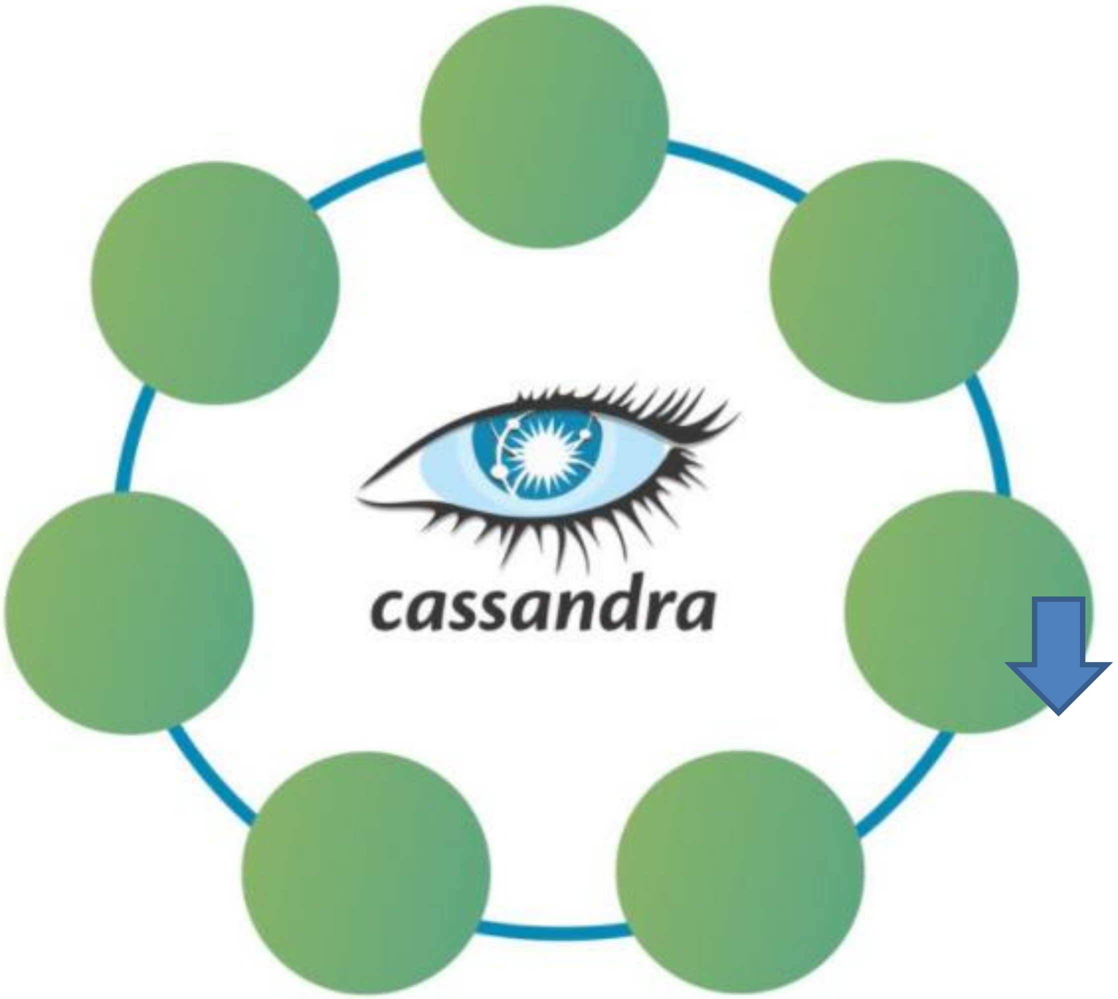
This works…

- SELECT * from students WHERE id > 100;

WHERE on a key but an inequality

InvalidRequest: Error from server: code=2200 [Invalid query] message="Only EQ and IN relation are supported on the partition key (unless you use the token() function)"

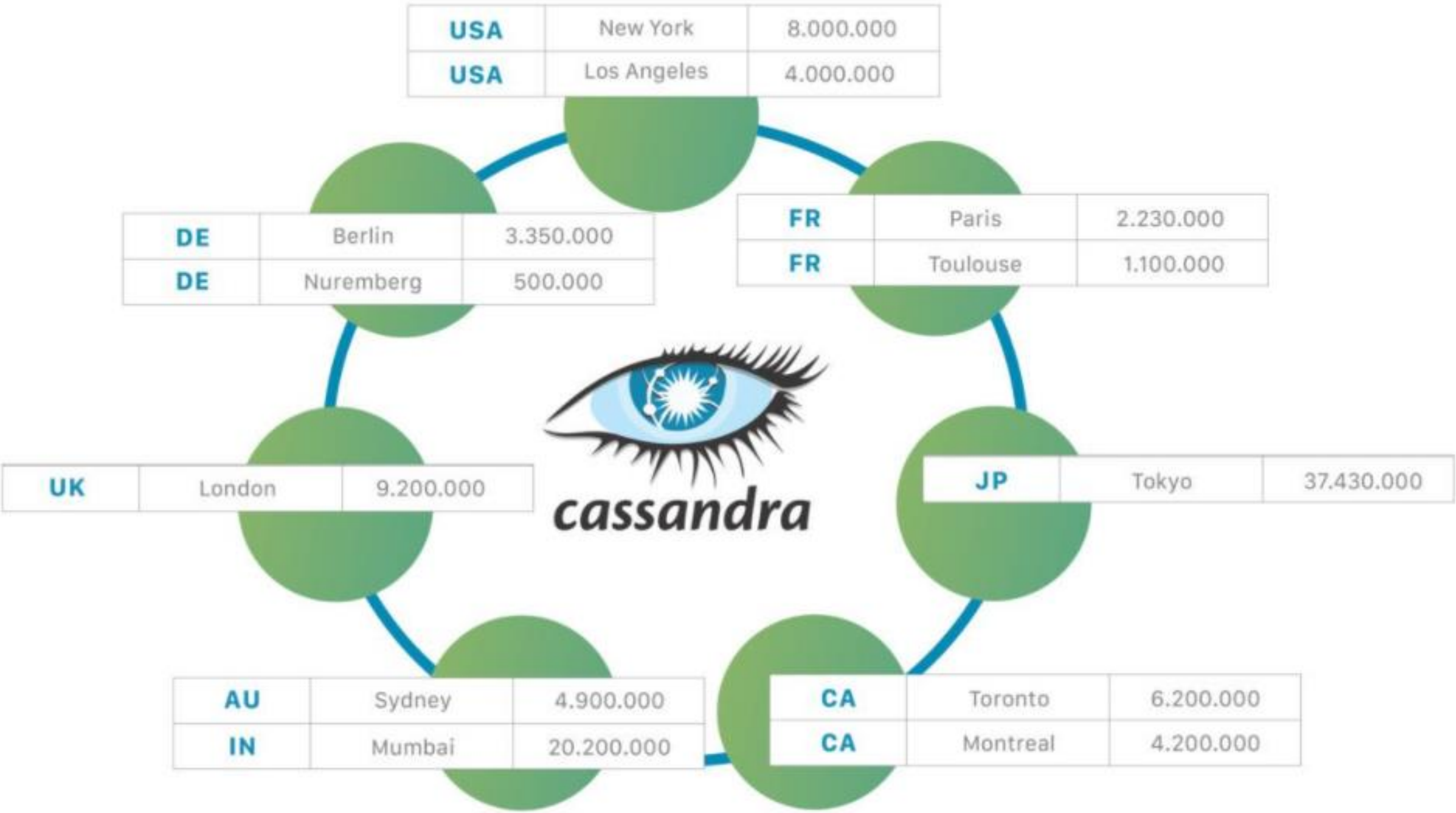
Partit



COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

Partition Key

Credit: https://www.baeldung.com/cassandra-cluster-datacenters-racks-nodes


USA | New York | 8.000.000
USA | Los Angeles | 4.000.000
DE | Berlin | 3.350.000
DE | Nuremberg | 500.000
FR | Paris | 2.230.000
FR | Toulouse | 1.100.000
UK | London | 9.200.000
cassandra
JP | Tokyo | 37.430.000
AU | Sydney | 4.900.000
IN | Mumbai | 20.200.000
CA | Toronto | 6.200.000
CA | Montreal | 4.200.000

Partition key and clustering key

```
CREATE TABLE crossfit_gyms (
 gym_name text,
 city text,
 state_province text,
 country_code text,
 PRIMARY KEY (gym_name)
);
```

```
CREATE TABLE crossfit_gyms_by_location (
 country_code text,
 state_province text,
 city text,
 gym_name text,
 PRIMARY KEY (country_code, state_province, city, gym_name)
);
```

partitioning key

clustering keys

retrieving the node and the partition

Sorting withing the partition

Clustering keys are sorted in ascending order by default.
So, when we query for all gyms in the United States, the result set will be ordered first by state_province in ascending order, followed by city in ascending order, and finally gym_name in ascending order.

Credit :https://shermandigital.com/blog/designing-a-cassandra-data-model/

CQL|Advanced Querying

Multiple primary keys (composite key), note the order

- CREATE TABLE grades (studentId INT, course TEXT, grade FLOAT, PRIMARY KEY(studentId, course));

Partition Key

Clustering key

- INSERT INTO grades(studentId, course, grade) values(111, 'into to intro', 95);
- INSERT INTO grades(studentId, course, grade) values(111, 'calculus', 78);
- INSERT INTO grades(studentId, course, grade) values(111, 'Algebra', 81);
- INSERT INTO grades(studentId, course, grade) values(222, 'Algebra', 51);
- INSERT INTO grades(studentId, course, grade) values(222, 'Algebra', 61);

This row is actually an update! (CQL supports update as well)

- SELECT * from grades;

```
 studentid | course        | grade
-----------+---------------+-------
       111 |       Algebra |    81
       111 |      calculus |    78
       111 | into to intro |    95
       222 |       Algebra |    61
```

- SELECT grade from grades WHERE studentid=111;

No need to provide all primary keys

```
grade
-------
   81
   78
   95
```

- SELECT grade from grades WHERE studentid=111 AND course > 'b';

```
 grade
-------
   78
   95
```

If the partition key (the first primary key) is provided, it is ok to have inequality conditions only on the **last** clustering key provided.

Illegal Queries (Require ALLOW_FILTERING)

- SELECT grade FROM grades WHERE course > 'b';
- SELECT grade FROM grades where course='Algebra';
- SELECT grade FROM grades WHERE grade > 70;
- SELECT grade FROM grades WHERE studentid=111 AND grade > 70;

Keys must be provided by order!

All these limitations/constrains are derived from the way Cassandra stores (and sorts) the data.

Think of the keys as a path in a

- √ partition key () is provided
- √ All the path until the last attribute is provided
- √ Inequality conditions only on the **last** clustering key provided.

Partition Key- Summery

- When data is inserted into the cluster, the first step is to apply a hash function to the partition key. The output is used to determine what node (and replicas) will get the data.
- The whole partition key must be specified (with equality sign) every query! (unless we request the whole table)
- This is because Cassandra must know where to find the requested data.
- The partition key can include more than a single key.
- E.g. CREATE TABLE grades (studentId INT, course TEXT, grade FLOAT, passed BIT, PRIMARY KEY((studentId, course), grade));
- SELECT * FROM grades WHERE studentId=111 AND course=20 ✔
- SELECT * FROM grades WHERE studentId=11 ✘

Order of Keys
(partition and clustering)

- Table T with Primary keys:

    x as the partition key and y, z as clustering keys

SELECT * FROM T WHERE x = 5; ✓

SELECT * FROM T WHERE y = 5; ✗

SELECT * FROM T WHERE x = 5 AND z = 7; ✗

SELECT * FROM T WHERE x = 5 AND y < 3 AND z > 0; ✗

SELECT * FROM T WHERE x = 5 AND y < 6 AND y > 0; ✓

SELECT * FROM T WHERE x = 5 AND z = 4 AND y > 0; ✗

SELECT * FROM T WHERE x < 10; ✗

SELECT * FROM T WHERE x =2 AND z <4 AND y = 3; ✓

SELECT * FROM T WHERE y = 2 AND z < 8; ✗

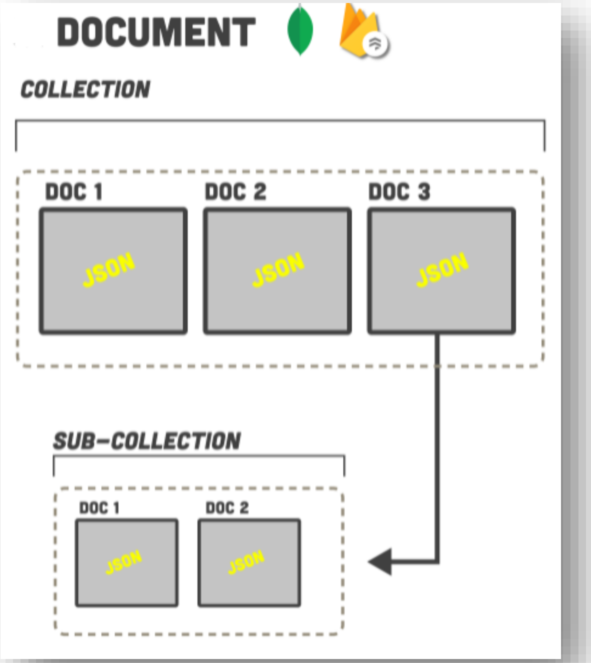
√ partition key is provided
√ All the path till the last attribute is provided
√ Inequality conditions only on the **last** clustering key provided.

Cassandra vs RDBMS

Property	Cassandra	RDBMS
Core Architecture	Masterless (no single point of failure)	Master-slave (single points of failure)
High Availability	Always-on continuous availability	General replication with master-slave
Data Model	Dynamic; structured and unstructured data	Legacy RDBMS; Structured data
Scalability Model	Big data/Linear scale performance	Oracle RAC or Exadata
Multi-Data Center Support	Multi-directional, multi-cloud availability	Nothing specific

Document Store

- Each key is paired with a document (a complex data structure).
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
- These documents are written in JSON.
- Unlike in the key-value store, the database has some level of "understanding" of these documents.
- Our example: MongoDB
  - Comes from the word *Humongous*



MongoDB

- Python, C++, JavaScript, JAVA and C# interfaces.
- Most widely used NoSQL database.
- Case sensitive!
- Syntax uses JavaScript (heavily use of JSON).
- Download from: https://www.mongodb.com/download-center
- Server:
  - ....\bin\mongod.exe --dbpath c:\temp\mongodata
- Client:
  - ....\bin\mongo.exe
  - Or download Robomongo for a GUI.

use and db.dropDatabase()

- use mydb will switch to mydb and create it if it doesn't already exist.

➢ use University

switched to db University

➢ db.dropDatabase()

{ "ok" : 1 }

db.createCollection(name, options)

- Tables are called collections in MongoDB.
- Collections are created automatically, but you can create a collection to set specific parameters.

circular collection (if there is no room, oldest document is deleted)

➢ db.createCollection("students", { capped : true, size : 6142800, max : 10000, autoIndexID : true })

size in bytes

max documents

Automatically create an id. (true is default)

➢ db.students.drop()

Delete collection

db.collection.insert(document)

- db.students.insert({"FirstName": "Chaya",
  "LastName": "Glass",
  "id": "111",
  "age": "21",
  "Address": {
    "Street": "Hatamr 5",
    "City": "Ariel",
    "Zip": "40792"}
  })
- db.students.insert([{"FirstName": "Tom", "LastName": "Glow", "Address": {"Street": "Mishmar 5","City": "Ariel"}}, {"FirstName": "Tal", "LastName": "Negev", "Address": {"Street": "Yarkon 26","City": "Jerusalem"}}])

JSON format

Inserting multiple document using a list

db.collection.insert(document)

- db.students.insert({"FirstName": "Chaya",
  "LastName": "Glass",
  "id": "111",
  "age": "21",
  "Address": {
    "Street": "Hatamr 5",
    "City": "Ariel",
    "Zip": "40792"}
  })
- db.students.insert([{"FirstName": "Tom", "LastName": "Glow", "Address": {"Street": "Mishmar 5","City": "Ariel"}}, {"FirstName": "Tal", "LastName": "Negev", "Address": {"Street": "Yarkon 26","City": "Jerusalem"}}])

db.collection.find()

- db.students.find()

Note the automatic ids

```
{ "_id" : ObjectId("589afa8c44a5653a862dd692"), "FirstName" : "Chaya", "LastName" : "Glass", "id" : "111", "age" : "21", "Address" : { "Street" : "Hatamr 5", "City" : "Ariel", "Zip" : "40792" } }
{ "_id" : ObjectId("589afa9244a5653a862dd693"), "FirstName" : "Tom", "LastName": "Glow", "Address" : { "Street" : "Mishmar 5", "City" : "Ariel" } }
{ "_id" : ObjectId("589afa9244a5653a862dd694"), "FirstName" : "Tal", "LastName": "Negev", "Address" : { "Street" : "Yarkon 26", "City" : "Jerusalem" } }
```

- db.students.find().pretty()

```
{
    "_id" : ObjectId("589af41d44a5653a862dd68d"),
    "Student" : {
        "FirstName" : "Chaya",
        "LastName" : "Glass",
        "id" : "111",
        "age" : "21",
        "Address" : {
            "Street" : "Hatamr 5",
            "City" : "Ariel",
            "Zip" : "40792"
        }
    }
}
{
    "_id" : ObjectId("589af4a844a5653a862dd68e"),
...
```

db.collection.find()

- db.students.find({"FirstName": "Tal"})

{ "_id" : ObjectId("589afa9244a5653a862dd694"), "FirstName" : "Tal", "LastName" : "Negev", "Address" : { "Street" : "Yarkon 26", "City" : "Jerusalem" } }

- db.students.find({"Address.City": "Ariel"})

{ "_id" : ObjectId("589afa8c44a5653a862dd692"), "FirstName" : "Chaya", "LastName" : "Glass", "id" : "111", "age" : "21", "Address" : { "Street" : "Hatamr 5", "City" : "Ariel", "Zip" : "40792" } }

{ "_id" : ObjectId("589afa9244a5653a862dd693"), "FirstName" : "Tom", "LastName": "Glow", "Address" : { "Street" : "Mishmar 5", "City" : "Ariel" } }

Inequalities in MongoDB
(Slide is provided just for completeness)

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50

Credit: https://www.tutorialspoint.com/mongodb/mongodb_query_document.htm

and, or

Note the list.

- db.students.find({$and: [{"FirstName": "Tal"},{"LastName":"Negev"}]})

{ "_id" : ObjectId("589afa9244a5653a862dd694"), "FirstName" : "Tal", "LastName": "Negev", "Address" : { "Street" : "Yarkon 26", "City" : "Jerusalem" } }

Equivalent to above.

- db.students.find({"FirstName": "Tal","LastName":"Negev"})
- db.students.find({"FirstName":"Tom", $or: [{"LastName":"Negev"},{"LastName":"Glow"}]})

{ "_id" : ObjectId("589afa9244a5653a862dd693"), "FirstName" : "Tom", "LastName" : "Glow", "Address" : { "Street" : "Mishmar 5", "City" : "Ariel" } }

Note that in all these examples we always gave find only a single parameter.

Select Specific Fields (Projection in Relational Algebra)

list of the properties you want to get

- db.students.find({"FirstName":"Tim"},{"FirstName":true})

{ "_id" : ObjectId("589afa9244a5653a862dd693"), "FirstName" : "Tim" }

Id is displayed even when not requested explicitly

- db.students.find({"FirstName":"Tim"},{"FirstName":true, _id:false})

{ "FirstName" : "Tim" }

- db.students.find({},{"FirstName":true, _id:false})

{ "FirstName" : "Chaya" }

{ "FirstName" : "Tim" }

{ "FirstName" : "Tal" }

RDBMS: SELECT FirstName FROM students

update, set

```
Syntax: db.collection.update(query, update, options)
```

- db.students.update({"FirstName":"Tom"}, {"FirstName" : "Tim", "LastName": "Glow", "Address" : { "Street" : "Mishmar 5", "City" : "Ariel" }})

MongoDB will search for a FirstName="Tom", and change the whole document to be:
{"FirstName" : "Tim", "LastName": "Glow", "Address" : { "Street" : "Mishmar 5", "City" : "Ariel" }})

- db.students.update({"FirstName":"Tom"}, {$set:{"FirstName":"Tim"}}, {multi:true})

Set will leave all other fields unchanged.

If we don't set multi to true, MongoDB will only set the first item it finds

Map-Reduce Paradigm

- A set of algorithms allowing parallel execution on massive amounts of data.
- **Mapper:** splits data, filters and runs a process.
- **Shuffle and Sort / Grouping:** ensures that all worker nodes have all data required for reduce.
- **Reduce:** worker nodes process each group of output data and build the output.
- We will come back to this paradigm when we learn Spark.

Map-Reduce Example

- Find the minimum and maximum of grades according to student age:
  - Map: every grade is mapped to an age.
  - Grouping: grades are divided into datasets (possibly) sorted by age such that every worker gets an age-group.
  - Reduce: every worker finds the minimum and maximum for every age in its dataset.
    - [Finally, if one group is split among more than single reducer: the minimum amongst the minimums and the maximum amongst the maximums is the final result.]

mapReduce()

- Suppose we have documents with the following structure:

```
{
    _id: ObjectId("50a8240b927d5d8b5891743c"),
    cust_id: "abc123",
    ord_date: new Date("Oct 04, 2012"),
    status: 'A',
    amount: 25,
    items: [ { sku: "chocolates", qty: 5, price: 2.5 },
            { sku: "oranges", qty: 5, price: 2.5 } ]
}
```

SKU = Stock Keeping Unit is an item identifier.

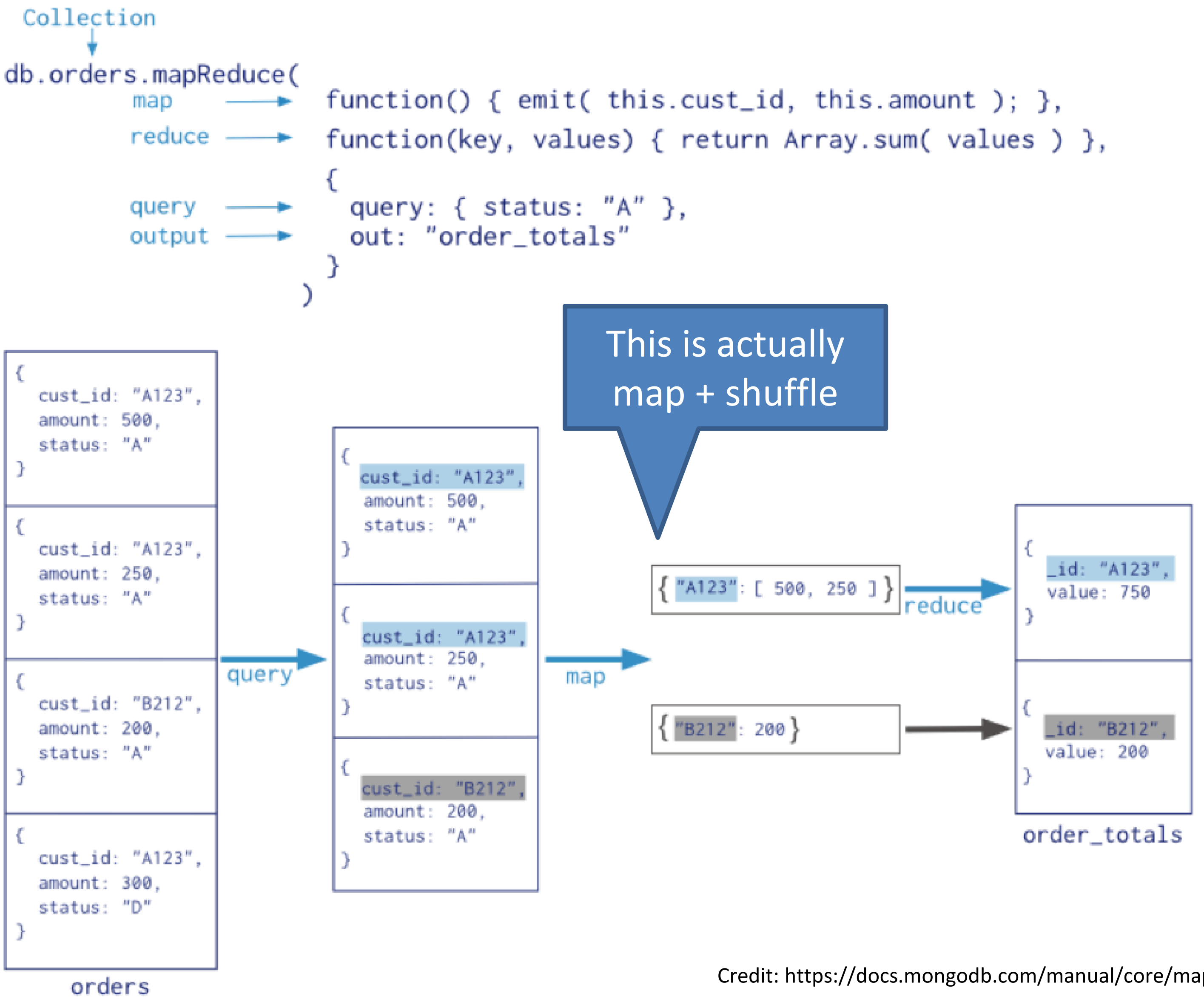
Can we do this in a relational DB, With SQL?

We will need to assume a different structure in a relational data-base.

- We would like to get the total **amount** paid by each cust_id with status 'A'.

Following slides are based on mongodb docs.

mapReduce() (cont.)



Credit: https://docs.mongodb.com/manual/core/map-reduce/

Results

- db.order_totals.find()

```
{ _id: 'Cam Elot', value: 60 }
{ _id: 'Don Quis', value: 155 }
{ _id: 'Busby Bee', value: 125 }
{ _id: 'Ant O. Knee', value: 95 }
```

A more complex example of mapReduce

- We would now like to get for each item identifier (SKU) the following:

  1. how many items were sold,
  2. the average quantity sold for that SKU,

  ignoring all items sold before 1/1/2012.

- E.g.:

```
[{ sku: "chocolates", total: 5543, avg: 5.4 },
{ sku: "apples", total: 1253, avg: 3.6 },
{ sku: "oranges", total: 53, avg: 1.1 }]
```

Answer

```
➢ db.orders.mapReduce(
        mapFunction2,
        reduceFunction2,
        {
            out: { merge: "sku_info" },
            query: { ord_date: { $gt: new Date('01/01/2012') } },
            finalize: finalizeAddAvgField
        }
)
```

Will be defined in next slides

merge means that if "sku_info" exists, we append to it.

Defined later

map Function

- The map function will go over each document that meets the queries filter, and output the SKUs as keys and the quantities as outputs.

```
var mapFunction2 = function() {
for (var idx = 0; idx < this.items.length; idx++) {
        var key = this.items[idx].sku;
        var value = {
                qty: this.items[idx].qty ,
                count: 1
                };
        emit(key, value);
        } };
```

We output count:1, this will help us with calculating the average.

key value

sku	quantity	count
chocolates	5	1
apples	10	1
chocolates	4	1
oranges	8	1

key	values
chocolates :	[(4,1),(5,1)]
apples:	[(10,1)]
oranges:	[(8,1)]

reduce

- The reduce function will go over the SKUs and lis from the map and add up the quantities and counters:

```
var reduceFunction2 =
        function(keySKU, countObjVals) {
        reducedVal = { count: 0, qty: 0 };
        for (var idx = 0; idx < countObjVals.length; idx++) {
                reducedVal.qty += countObjVals[idx].qty;
                reducedVal.count += countObjVals[idx].count;
                        }
        return reducedVal;
        };
```

key value

sku	quantity	count
chocolates	5	1
apples	10	1
chocolates	4	1
oranges	8	1

keySKU countObjVals

key	values
chocolates :	[(4,1),(5,1)]
apples:	[(10,1)]
oranges:	[(8,1)]

keySKU Reduced Values

key	values
chocolates :	[(9,2)]
apples:	[(10,1)]
oranges:	[(8,1)]

?

finalize

- The finalize function will go over the reduce's output and calculate the average:

```
var finalizeAddAvgField =
        function (key, reducedVal) {
                reducedVal.avg = reducedVal.qty/reducedVal.count;
                return reducedVal;
};
```

Results

- db.sku_info.find()

```
{ _id: 'chocolates', value: { count: 3, qty: 15, avg: 5 } }
{ _id: 'apples', value: { count: 4, qty: 35, avg: 8.75 } }
{ _id: 'oranges', value: { count: 7, qty: 63, avg: 9 } }
{ _id: 'pears', value: { count: 1, qty: 10, avg: 10 } }
{ _id: 'carrots', value: { count: 2, qty: 15, avg: 7.5 } }
```

Search Engine Databases

- Search engine databases are a sub-type of document stores.
- Scoring or relevance of documents plays a deep role, something that we hadn't seen before in relational DB or other NoSQL DBs.
- Our example: Elastic Search

Elastic Search

Elastic Search

- Used in: Wikipedia, StackOverflow, GitHub, The Guardian and more.
- RESTful and Java API.
- Case sensitive (even in url).
- Builds upon Lucene.
- Near Real Time
- Download from: https://www.elastic.co/downloads/elasticsearch and unzip (or use apt in Linux).
- Run as administrator …\bin\elasticsearch.bat
- We will use curl for HTTP commands: basically GET, POST, PUT, HEAD and DELETE. [Better use other user interfaces (e.g. fiddler2).]

It takes approximately 1 second from the time a document is added or edited until it can be searched (compare this with the time it takes Google to index new pages).

Elastic Search Data Model terms

Relational Databases	Elastic Search
Database	Index
Table	type
Row	Document
column	field
schema	mapping

Elastic Search

Adding Documents (items)

- There is no need to create an Index (Database) or a Type (a table), we can right away insert a document using:

dbServer address index type docID

➢ curl -XPUT "http://localhost:9200/university/students/111" -H "Content-Type: application/json" -d "{\"FirstName\": \"Chaya\", \"LastName\": \"Glass\", \"age\": \"21\", \"Address\": { \"Street\": \"Hatamr 5\", \"City\": \"Ariel\",\"Zip\": \"40792\"}}“

➢ curl -XPUT http://localhost:9200/university/students/333 -H "Content-Type: application/json" -d "{\"FirstName\": \"Gadi\", \"LastName\": \"Golan\", \"age\": \"24\"}"

Adding Documents (items)

POST without id, will generate id automatically

- curl -XPOST http://localhost:9200/university/students -H "Content-Type: application/json" -d "{\"FirstName\": \"Tal\", \"LastName\": \"Negev\", \"age\": \"28\"}"

Generally, in REST API, PUT is idempotent (n{msg} = {msg}), and POST isn't. (What will happen if we send each of the above messages twice?)

GET, HEAD, DELETE

- curl -XGET "http://localhost:9200/university/students/333"

{"_index":"university","_type":"students","_id":"333","_version":1,"_seq_no":1,"_primary_term":1,"found":true,"_source":{"FirstName": "Gadi", "LastName": "Golan", "age": "24"}}

Note all the metadata.

- HEAD only tests if document exists.
  - curl -I –XHEAD http://localhost:9200/university/students/333
    - will return: OK
  - curl -I –XHEAD http://localhost:9200/university/students/555
    - will return: NOT FOUND
- DELETE deletes the document
  - curl -XDELETE "http://localhost:9200/university/students/333

UPDATE

- Let's first add some descriptions to the students' documents, using _update:

- curl -XPOST http://localhost:9200/university/students/111/_update -H "Content-Type: application/json" -d "{ \"script\" : \"ctx._source.description = \\\"Likes learning but gets board very quickly. Doesn't enjoy trips that much.\\\"\" }"
- curl -XPOST http://localhost:9200/university/students/333/_update -H "Content-Type: application/json" -d "{ \"script\" : \"ctx._source.description = \\\"Doesn't show-up to lessons, but is very smart and learns a lot.\\\"\" }"
- curl –XPOST http://localhost:9200/university/students/lA9AInsBL04IeaKD9LL9/_update -H "Content-Type: application/json" -d "{ \"script\" : \"ctx._source.description = \\\"Doesn't know anything. Goes on trips all day, never showed-up to a single lesson.\\\"\" }"

Search

- The search API allows you to execute a search query and get back search hits that match the query.
- The query can either be provided using a simple query string as a parameter or using a request body.

Search (query string as a parameter)

- curl –XGET "http://localhost:9200/university/students/_search"

SELECT * FROM students

{"took":2,"timed_out":false,"_shards":{"total":1,"successful":1,"skipped":0,"failed":0},"hits":{"total":{"value":3,"relation":"eq"},"max_score":1.0,"hits":[{"_index":"university","_type":"students","_id":"333","_score":1.0,"_source":{"FirstName":"Gadi","LastName":"Golan","age":"24","description":"Doesn't show-up to lessons, but is very smart and learns a lot."}},{"_index":"university","_type":"students","_id":"111","_score":1.0,"_source":{"FirstName":"Chaya","LastName":"Glass","age":"21","Address":{"Street":"Hatamr 5","City":"Ariel","Zip":"40792"},"description":"Likes learning but gets board very quickly. Doesn't enjoy trips that much."}},{"_index":"university","_type":"students","_id":"lA9AInsBL04IeaKD9LL9","_score":1.0,"_source":{"FirstName":"Tal","LastName":"Negev","age":"28","description":"Doesn't know anything. Goes on trips all day, never showed-up to a single lesson."}}]}}

~ SELECT * FROM students WHERE LastName *like* 'Negev'

- curl -XGET http://localhost:9200/university/students/_search?q=LastName:Negev"

{"took":3,"timed_out":false,"_shards":{"total":1,"successful":1,"skipped":0,"failed":0},"hits":{"total":{"value":1,"relation":"eq"},"max_score":0.6931471,"hits":[{"_index":"university","_type":"students","_id":"lA9AInsBL04IeaKD9LL9","_score":0.6931471,"_source":{"FirstName":"Tal","LastName":"Negev","age":"28","description":"Doesn't know anything. Goes on trips all day, never showed-up to a single lesson."}}]}}

Search (with request body)|match

Match - Returns documents that match a provided text, number, date or boolean value . The *match* query is the standard query for performing a full-text search, including options for fuzzy matching.

- curl –XGET "http://localhost:9200/university/students/_search" -H "Content-Type: application/json" -d"{\"query\": {\"match\": {\"description\": { \"query\": \"very smart quickly \" }}}}

It is interpreted as "very" OR "smart" OR "quickly"

- curl –XGET "http://localhost:9200/university/students/_search" -H "Content-Type: application/json" –d"{\"query\": {\"match\": {\"description\": { \"very smart quickly \" }}}}

{"took":2,"timed_out":false,"_shards":{"total":1,"successful":1,"skipped":0,"failed":0},"hits":{"total":{"value":2,"relation":"eq
"},"max_score":1.5127167,"hits":[{"_index":"university","_type":"students","_id":"111","_score":1.5127167,"_source":{"First
Name":"Chaya","LastName":"Glass","age":"21","Address":{"Street":"Hatamr
5","City":"Ariel","Zip":"40792"},"description":"Likes learning but gets board very quickly. Doesn't enjoy trips that
much."}},{"_index":"university","_type":"students","_id":"333","_score":1.4658242,"_source":{"FirstName":"Gadi","LastNam
e":"Golan","age":"24","description":"Doesn't show-up to lessons, but is very smart and learns a lot."}}]}}

Search (with request body)|match_phrase

- curl –XGET "http://localhost:9200/university/students/_search" -H "Content-Type: application/json" -d"{\"query\": {\"match_phrase\": {\"description\": { \"query\": \"very smart \" }}}}

{"took":3,"timed_out":false,"_shards":{"total":1,"successful":1,"skipped":0,"failed":0},"hits":{"total":{"value":1,"relation":"eq"},"max_score":1.4658242,"hits":[{"_index":"university","_type":"students","_id":"333","_score":1.4658242,"_source":{"FirstName":"Gadi","LastName":"Golan","age":"24","description":"Doesn't show-up to lessons, but is very smart and learns a lot."}}]}}

Phrase matching is necessary when the ordering of the words is important. Only the documents that contain the words in the same order as the search input are matched.

Advanced Search (bool query)

```
POST _search
{  "query":
        {    "bool" :
{      "must" : [],
"filter": [],
"must_not" : [],
"should" : []    }  }}
```

If the results include this query,

keyword	meaning	Scoring the results
Should	Finding the text will increase the score	Yes
Must	The results **must** contain the string	Yes
filter	The results **must** contain the string	No
Must not	The results **must not** contain the string	no

Advanced Search (bool query)

```
POST employees/_search
{
 "query": {
  "bool": {
   "must": [
     {
      "match": {
       "position": "manager"
      }
     },
     {
      "range": {
       "experience": {
        "gte": 12
       }
      }
     }
    ],
  "should": [
    {
     "match": {
      "phrase": "versatile"
     }
    }
   ]
  }
 }
}
```

```
{
  "took" : 0,
  "timed_out" : false,
  "_shards" : { },
  "hits" : {
    "total" : { },
    "max_score" : 2.8970814,
    "hits" : [
      {
        "_index" : "employees",
        "_type" : "_doc",
        "_id" : "3",
        "_score" : 2.8970814,
        "_source" : {
          "id" : 3,
          "name" : "Winston Waren",
          "email" : "wwaren2@4shared.com",
          "gender" : "Male",
          "ip_address" : "202.37.210.94",
          "date_of_birth" : "10/11/1985",
          "company" : "Yozio",
          "position" : "Human Resources Manager",
          "experiance" : 12,
          "country" : "China",
          "phrase" : "Versatile object-oriented emulation",
          "salary" : 50616
        }
      },
      {
        "_index" : "employees",
        "_type" : "_doc",
        "_id" : "4",
        "_score" : 1.7261542,
        "_source" : {
          "id" : 4,
          "name" : "Alan Thomas",
          "email" : "athomas2@gmail.com",
          "gender" : "Male",
          "ip_address" : "200.47.210.95",
          "date_of_birth" : "11/12/1985",
          "company" : "Yamaha",
          "position" : "Resources Manager",
          "experiance" : 12,
          "country" : "China",
          "phrase" : "Emulation of roots heuristic coherent systems",
          "salary" : 300000
        }
      }
    ]
  }
}
```

3 matches (2 from must and 1 from should). Hence the score is high and the document is listed above the other document (document with id=4)

only 2 matches (2 from must) compared to the document with id=3

Scoring: Relevance

- Elasticsearch returns the documents with the highest score.
- As seen, when querying with "filter" no score is aggregated. ("must_not" doesn't have a score either).
- Elasticsearch supports also "must" and "should". Both influence the score.
- "must" and "should" can also appear inside a "filter", in which case "must" is a logical "AND", while "should" is a logical "OR".
- Scoring becomes interesting when we start matching strings.
- Elasticsearch uses a bag of words TF-IDF model (will discuss later on).

Advanced Search (bool query)

➢ curl -XGET "http://localhost:9200/university/students/_search" -d"

```
{ \"query\" :
  { \"bool\" :
    { \"filter\" :
      { \"match\" :
        { \"Address.City\" : \"Ariel\" } },
    \"filter\" :
      { \"range\" :
        { \"age\" :
          { \"lt\" : 30 } } } } } } "
```

Boolean combination of several queries.

All students living in Ariel

Students under 30

{"took":6,"timed_out":false,"_shards":{
"total":5,"successful":5,"failed":0},"hit
s":{"total":1,"max_score":0.0,"hits":[{"_
index":"university","_type":"students",
"_id":"111","_score":0.0,"_source":{"Fi
rstName": "Chaya", "LastName":
"Glass", "
age": "21", "Address": { "Street":
"Hatamr 5", "City": "Ariel","Zip":
"40792"}}}
]}}

String Queries (cont.)

- curl -XGET http://localhost:9200/university/students/_search -d "{ \"query\" : { \"match\" : { \"description\" : \"doesn't show-up learns\" } }}

,\"highlight\": { \"fields\" : { \"description\" : {} } }

{"took":8,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{"total":3
,"max_score":1.0514642,"hits":[
{"_index":"university","_type":"students","_id":"222","_score":1.0514642,"_source":{"FirstName":"Tal","LastName":"Negev","age":"28","description":"Doesn't show-up to lessons, but is very smart and learns a lot."}},
{"_index":"university","_type":"students","_id":"AVohwsYTVPDjS737L8vs","_score":0.56008905,"_source":{"FirstName":"Gadi","LastName":"Golan","age":"24","description":
"Doesn't know anything. Goes on trips all day, never showed-up to a single lesson."}},
{"_index":"university","_type":"students","_id":"111","_score":0.25316024,"_source":{"FirstName":"Chaya","LastName":"Glass","age":"21","Address":{"Street":"Hatamr
5","City":"Ariel","Zip":"40792"},"description":"Likes learning but gets board very quickly.
Doesn't enjoy trips that much."}}]}}

The underlined words, can be highlighted automatically, using the "highlight" option!

Information Retrieval (TF-IDF)
Document Ranking

- Suppose we have a query (Q) and many possible documents (D={d1,d2…}). How can we rank these documents based on the query?

Document Ranking Example

- Q: "Who is the president of the united states?"

========== ? What is the most relevant match ? =========

- D1: Donald Trump is United States' president.
- D2: We are the most united out of all the people and of all the places.
- D3: The United States of America is united again, who is more united than it?
- D4: Who would like to take the box out of the kitchen?

Tf-Idf

- TF: Term Frequency. A simple count of how many times the given keyword appears in the document normalized by the number of words in the document.
- IDF: Inverse Document Frequency: The *log* of: the total number of *documents* divided by the number of documents the term appears in.

Number of instances of k in d

Total number of documents

- $tfidf(d) = \sum_{k=0}^{|Q|} \frac{\#k\ in\ d}{|d|} \log(\frac{|\#D|}{\#D\ with\ k})$

number of documents with the word “k”

TF

IDF

Total Number of words in d

k: word in document/query
Q: set of words in query

Document Ranking

	Who	Is	The	President	Of	United	States	#words
D1	0	1	0	1	0	1	1	6
D2	0	0	3	0	2	1	0	15
D3	1	1	1	0	1	3	1	14
D4	1	0	2	0	1	0	0	11
#D with k	2	2	3	1	3	3	2	

number of documents with the word “k”

- Q: Who is the president of the united states?
- D1: Donald Trump is United States’ president.
- D2: We are the most united out of all the people and of all the places.
- D3: The United States of America is united again, who is more united than it?
- D4: Who would like to take the box out of the kitchen?

Document Ranking

	Who	Is	The	President	Of	United	States	#words
D1	0	1	0	1	0	1	1	6
D2	0	0	3	0	2	1	0	15
D3	1	1	1	0	1	3	1	14
D4	1	0	2	0	1	0	0	11
#D with k	2	2	3	1	3	3	2	

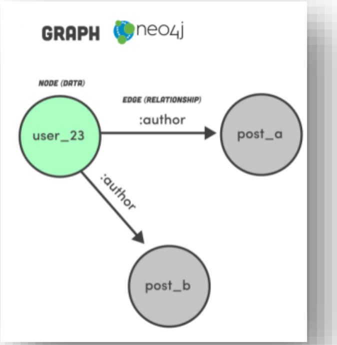
number of documents with the word “k”

$$tfidf(d) = \sum_{k=0}^{|Q|} \frac{\#k\ in\ d}{|d|} \log\left(\frac{|D|}{\#D\ with\ k}\right)$$

Doc	Tf-Idf score
D1	(1/6)*log(4/2)+(1/6)*log(4/1)+(1/6)*log(4/3)+(1/6)*log(4/2)=0.736
D2	(3/15)*log(4/3)+(2/15)*log(4/3)+(1/15)*log(4/3)=0.166
D3	(1/14)*log(4/2)+(1/14)*log(4/2)+(1/14)*log(4/3)+(1/14)*log(4/3)+(3/14)*log(4/3)+(1/14)*log(4/2)=0.363
D4	(log(4/2)+2*log(4/3)+log(4/3))/11=0.204

Graph Databases

- Store information as graphs, with nodes and relations between the nodes.
- Allow interesting queries such as, finding all the friends of a person, all their friends, and so on until 10 levels.

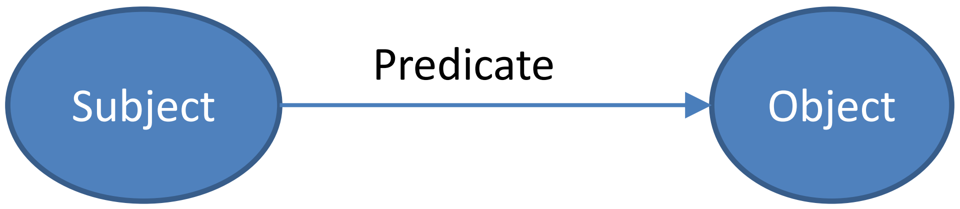


Resource Description Framework (RDF) graph databases

- RDF is a standard model for data interchange on the Web.
- RDF has features that facilitate data merging even if the underlying schemas differ.
- RDF specifically supports the evolution of schemas over time without requiring all the data consumers to be changed.

RDF Triples

- The dataset includes a list of triples of the form:
  - subject, predicate, object.



- All data is presented only by these triples.
- There is only a single "table", containing all these triples (a triple store).

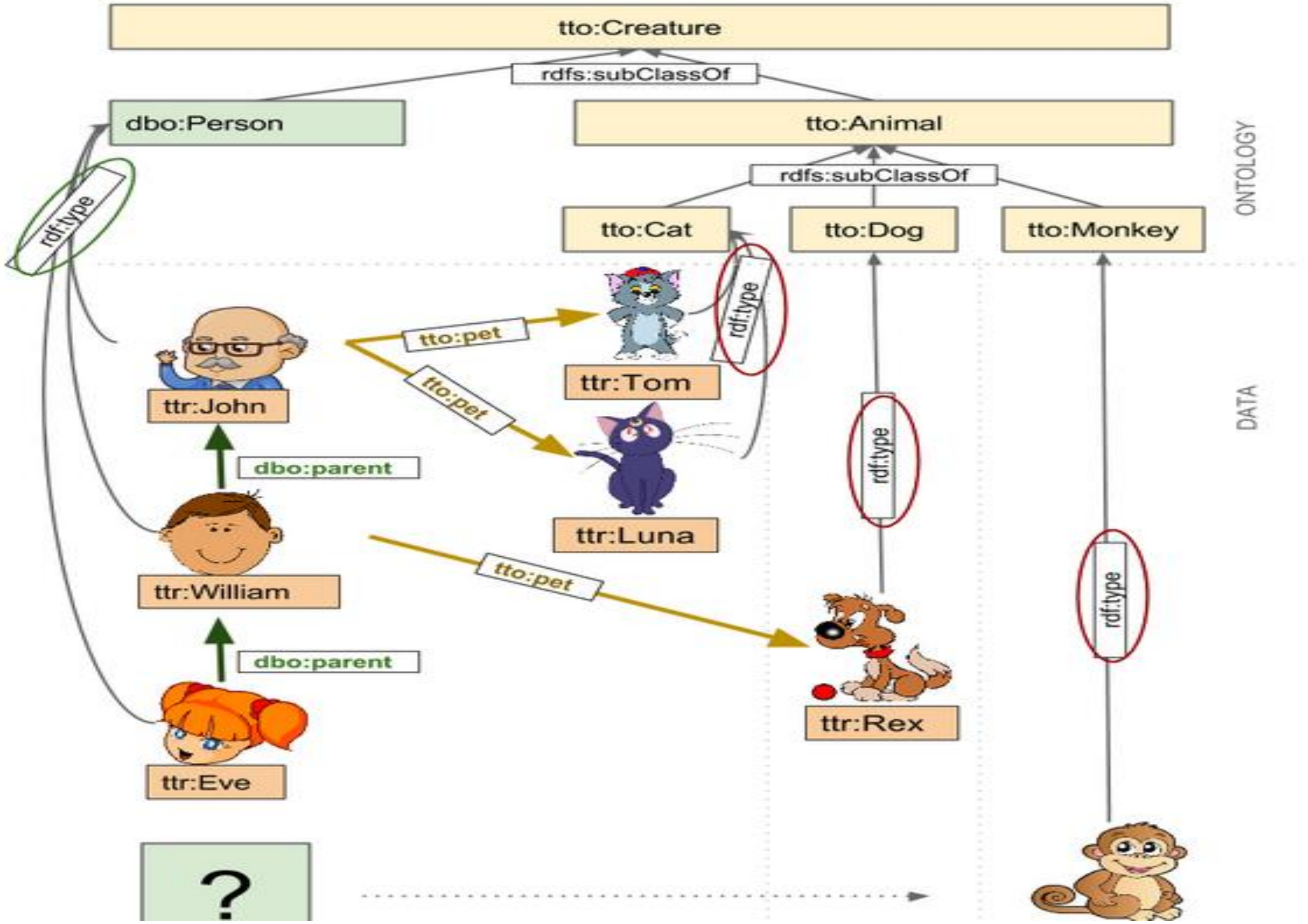
Apache Jena

- Download from: https://jena.apache.org/download/index.cgi
- There is no need to actually install Jena, unless you intend to use it.
- Jena uses URIs to identify all entities and relations.
- We will focus on SPARQL which is the RDF query language.

ARQ / SPARQL

- RDF query language is called SPARQL.
- ARQ is an implementation of a SPARQL Processor for Jena.
- All queries include a set of triples: subject, predicate, object.
- We will only learn selection.
- You can try SPARQL at: http://sparql-playground.sib.swiss/
- [The queries and images in the following slides are taken from there.]

A Partial View of Our Database (Ontology + Data)



SELECT *

- SELECT * WHERE {?s ?p ?o}

Why is there no "FROM" clause?

s	p	o
ttr:Eve	dbo:parent	ttr:William
ttr:Eve	dbp:birthDate	"2006-11-03"
ttr:Eve	dbp:name	"Eve"
ttr:Eve	tto:sex	"female"
ttr:Eve	rdf:type	dbo:Person
ttr:John	dbp:birthDate	"1942-02-02"
ttr:John	dbp:name	"John"
ttr:John	tto:pet	ttr:LunaCat
ttr:John	tto:pet	ttr:TomCat
ttr:John	tto:sex	"male"
ttr:John	rdf:type	dbo:Person
ttr:LunaCat	dbp:name	"Luna"
ttr:LunaCat	tto:color	"violet"
ttr:LunaCat	tto:sex	"female"
ttr:LunaCat	tto:weight	"4.2"
ttr:LunaCat	rdf:type	tto:Cat
ttr:RexDog	dbp:name	"Rex"
ttr:RexDog	tto:color	"brown"
ttr:RexDog	tto:sex	"male"
ttr:RexDog	tto:weight	"8.8"
ttr:RexDog	rdf:type	tto:Dog
ttr:SnuffMonkey	dbp:name	"Snuff"
ttr:SnuffMonkey	tto:color	"golden"
ttr:SnuffMonkey	tto:sex	"male"

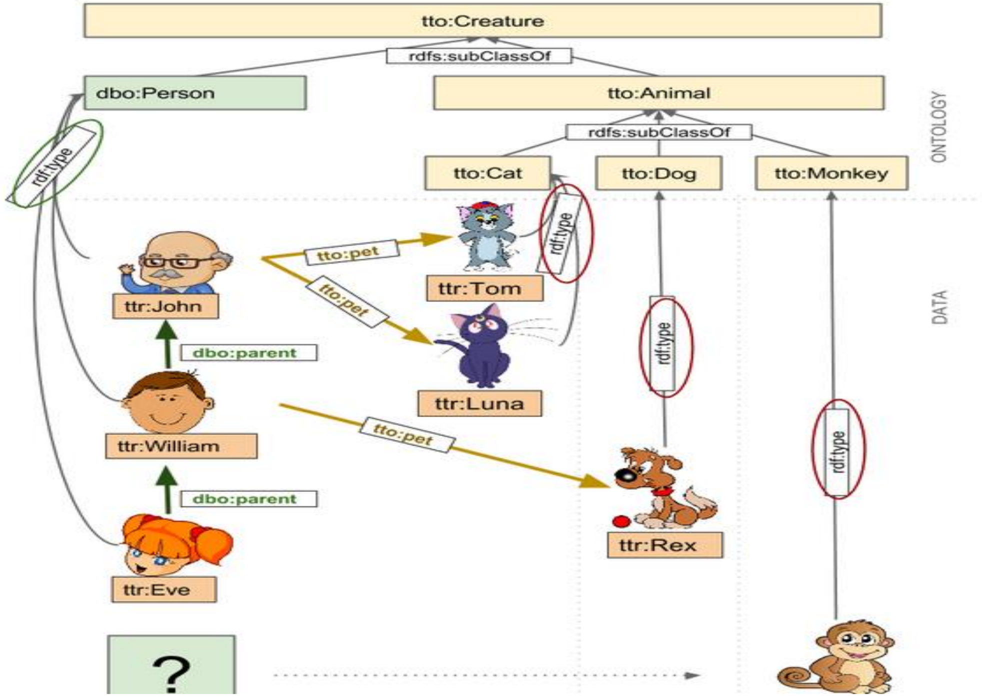
Select all Persons

- SELECT ?something WHERE

{?something rdf:type dbo:Person .

}

something
ttr:Eve
ttr:John
ttr:William



Select all Women

- SELECT ?thing WHERE {

    ?thing rdf:type dbo:Person .

    ?thing tto:sex "female" .

    }

"rdf:type" can be replaced by "a"

thing
ttr:Eve

Select Persons That Have Pets

- SELECT ?person WHERE {

    ?person rdf:type dbo:Person .

    ?person tto:pet ?pet .

}

person
ttr:John
ttr:John
ttr:William

Why do we have two ttr:John?

We can add the "DISTINCT" keyword above to avoid this.

Select Persons That Have **Cats**

- SELECT DISTINCT ?person WHERE {

```
        ?person rdf:type dbo:Person .
        ?person tto:pet ?type .
        ?type rdf:type tto:Cat .
}
```

person
ttr:John

Select Persons That do **not** Have any Pets

- SELECT ?person WHERE {

  ?person rdf:type dbo:Person .

  FILTER NOT EXISTS {?person tto:pet ?pet } .

}

person
ttr:Eve

Select Persons That do **not** Have any **Cats**

- SELECT ?person WHERE {

```
        ?person rdf:type dbo:Person .
        FILTER NOT EXISTS {
                ?person tto:pet ?petType
                ?petType rdf:type tto:Cat .}
}
```

person
ttr:Eve
ttr:William

Select Persons That do **not** Have any **Cats**

- SELECT ?person WHERE {

    ?person rdf:type dbo:Person .

    FILTER NOT EXISTS {

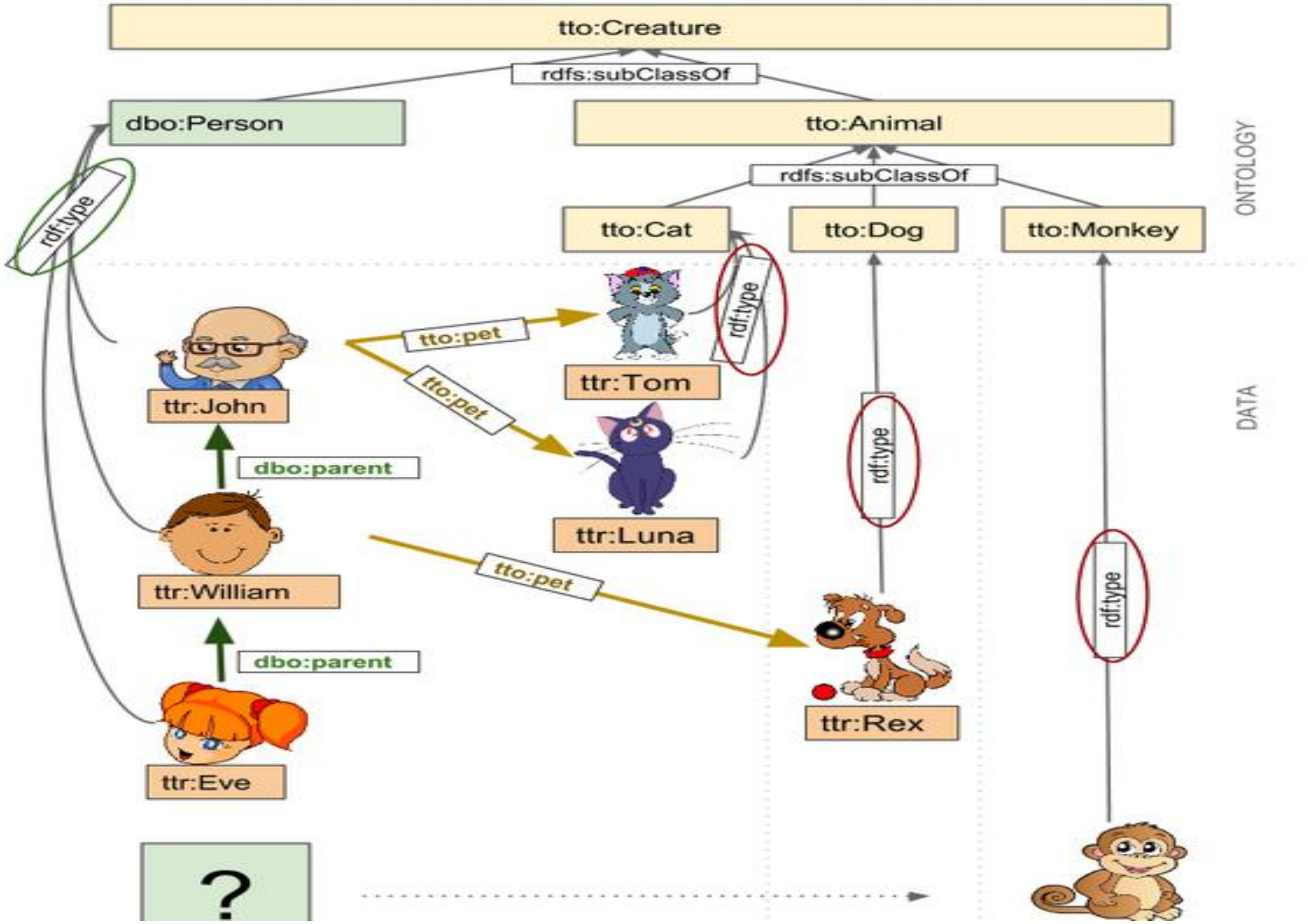
        ?person tto:pet / rdf:type tto:Cat .}

}

/ Concatenates relations

person
ttr:Eve
ttr:William

A Partial View of Our Database (Ontology + Data)



Select all Creatures (using UNION)

```
SELECT ?thing WHERE
 {
          ?thing rdf:type ?type .
          {
                ?type rdfs:subClassOf tto:Creature .
           }
          UNION
           {
                ?type rdfs:subClassOf ?subcreature .
                ?subcreature rdfs:subClassOf tto:Creature .
            }
}
```

thing
ttr:Eve
ttr:John
ttr:William
ttr:LunaCat
ttr:TomCat
ttr:RexDog
ttr:SnuffMonkey

Select all Creatures 2 (simpler)

```
SELECT ?thing WHERE
 {
        {
            ?thing rdf:type / rdfs:subClassOf tto:Creature .
        }
        UNION
        {
            ?thing rdf:type / rdfs:subClassOf / rdfs:subClassOf tto:Creature .
        }
}
```

thing
ttr:Eve
ttr:John
ttr:William
ttr:LunaCat
ttr:TomCat
ttr:RexDog
ttr:SnuffMonkey

Select all Creatures 3 (simpler and more correct)

- SELECT ?thing WHERE {

  ?thing rdf:type / rdfs:subClassOf+ tto:Creature .

  }

thing
ttr:Eve
ttr:John
ttr:William
ttr:LunaCat
ttr:TomCat
ttr:RexDog
ttr:SnuffMonkey

Direct and indirect sub classes

- select ?subSpecies where {
  ?subSpecies rdfs:subClassOf? tto:Creature .

subSpecies
tto:Creature
dbo:Person
tto:Animal

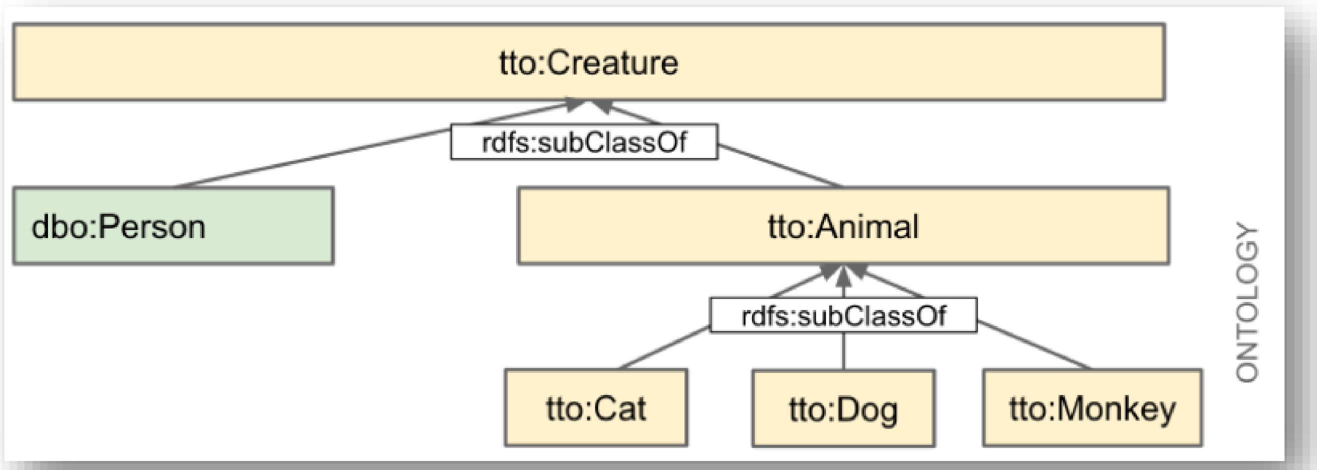
? Is 0 or 1
* is 0 or more
+ is 1 or more of same predicate

- select ?subSpecies where {
  ?subSpecies rdfs:subClassOf* tto:Creature .

subSpecies
tto:Creature
dbo:Person
tto:Animal
tto:Cat
tto:Dog
tto:Monkey

- select ?subSpecies where {
  ?subSpecies rdfs:subClassOf+ tto:Creature .

subSpecies
dbo:Person
tto:Animal
tto:Cat
tto:Dog
tto:Monkey

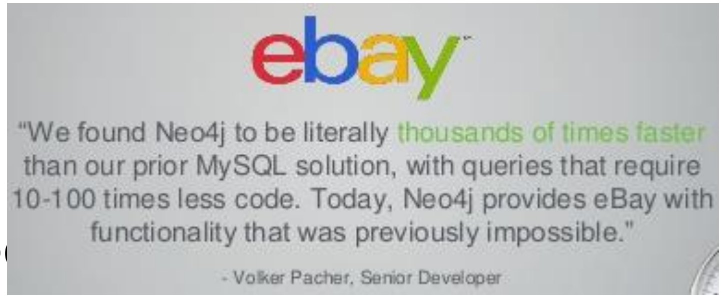


General Graph Databases

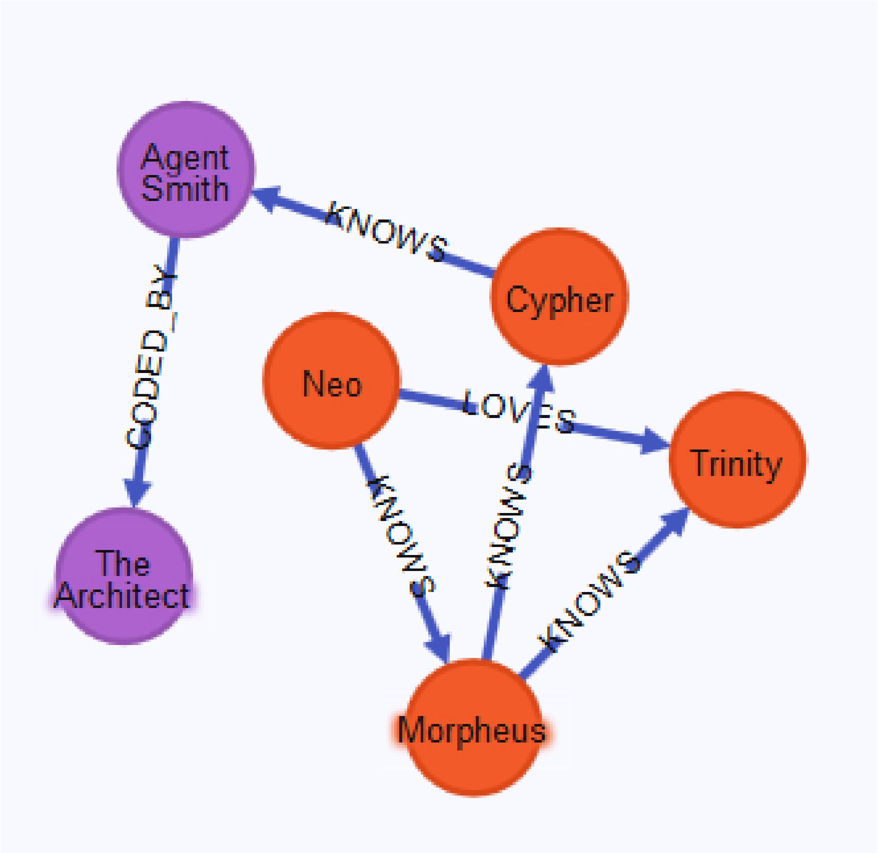
- Graph databases are not limited to RDF and SPARQL.
- The more general graph databases allow more complex queries (not only triples).

Neo4J

- Name comes from "The Matrix".
- Query Language called Cypher.
- ACID (almost…)
- Labeled property graph
- Case insensitive.
- Used by: ebay, Walmart, Cisco and many more…
- Initially was only with Java interface (that's why it is 4J), but also has a REST API.
- Can try without installing at:
  - http://console.neo4j.org/
- Open source:
  - Community edition: GPL (for any use, but any mo source)
  - Enterprise edition: AGPL (they claim that it is only for open source – but this seems like a false claim to me.)



Graph Example



(Neo)-[:LOVES]->(Trinity)

(Neo)-[]->(Trinity)

(Neo)-->(Trinity)

(Trinity)--(Neo)

(Architect)<-[:CODED_BY]-(Smith)

(Morpheus)-[:KNOWS]-(Trinity)

CREATE

- CREATE (n)

The node reference ("glass") can only be used during the same query

Here student is a label. Labels act like categories or types.

- CREATE (glass:student {name: 'Chaya Glass', id:111, age:21, degree:'1'})

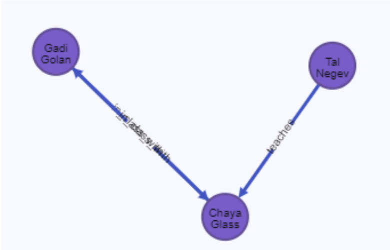
Properties

Can use an empty reference

- CREATE (:student {name: 'Tal Negev', id:222, age:28, degree:'3'}), (:student {name: 'Gadi Golan', id:333, age:24, degree:'1'})

Can create many nodes at once

Adding Relations



- When creating the graph and adding the nodes we can easily add relationships between the nodes, in the same CREATE statement using the node name.

➢ CREATE (glass:student {name: 'Chaya Glass', id:111, age:21, degree:'1'}), (negev:student {name: 'Tal Negev', id:222, age:28, degree:'3'}), (golan:student {name: 'Gadi Golan', id:333, age:24, degree:'1'}), (negev)-[r1:teaches]->(glass), (golan)-[:in_class_with]->(glass), (glass)-[:in_class_with]->(golan)

All edges are directional. It is redundant to create the inverse relationship e.g.: (glass)-[:taught_by]->(negev).

Furthermore, also the 'in_class_with' relationship, could be defined only in one direction, and later ignore the direction when traversing the graph.

Adding Relations (cont.)

- To add relations to nodes already present in the graph, we first need to find them:
  - ➢ MATCH (a:student),(b:student) WHERE a.name = 'Tal Negev' AND b.name = 'Chaya Glass' CREATE (a)-[r1:teaches]->(b)

Queries: MATCH

The *MATCH* clause is used to search for the pattern described in it.

pattern

- MATCH (a)-->(b{name:'Chaya Glass'}) RETURN a
  - Find all nodes who have any relation with Chaya Glass
- MATCH (a)-[:teaches]->(b:student) RETURN a.name
  - Find all names of those who teach students
- MATCH (a)-[:teaches]->(b:student {name:'Chaya Glass'}) RETURN a.name
  - Find all names of those who teach the student Chaya Glass

a.name
Tal Negev

Variable-length pattern matching

- (a)-[*2]->(b)   equivalent to: (a)-->()-->(b)

This describes a graph of three nodes and two relationships, all in one path (a path of length 2).

A range of lengths can also be specified: such relationship patterns are called 'variable length relationships'. For example:

(a)-[*3..5]->(b)

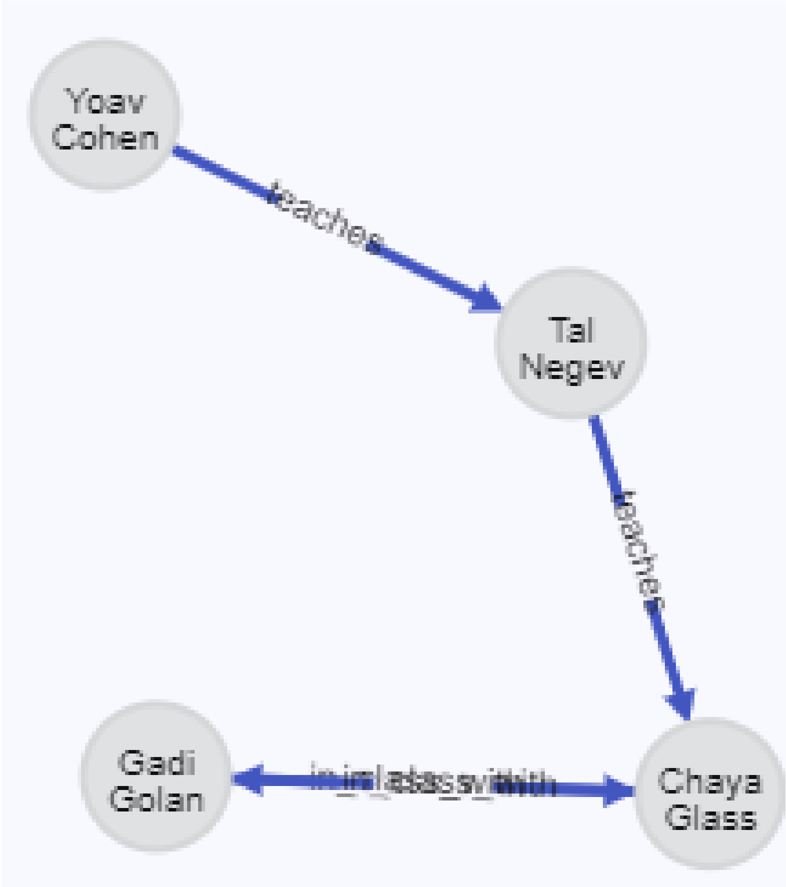
This is a minimum length of 3, and a maximum of 5.

It describes a graph of either 4 nodes and 3 relationships, 5 nodes and 4 relationships or 6 nodes and 5 relationships, all connected together in a single path

More Queries

Add some data:

- CREATE (cohen:student {name: 'Yoav Cohen', id:666, age:21, degree: '1'})
- MATCH (a:student),(b:student) WHERE a.name = 'Yoav Cohen' AND b.name = 'Tal Negev' CREATE (a)-[r1:teaches]->(b)

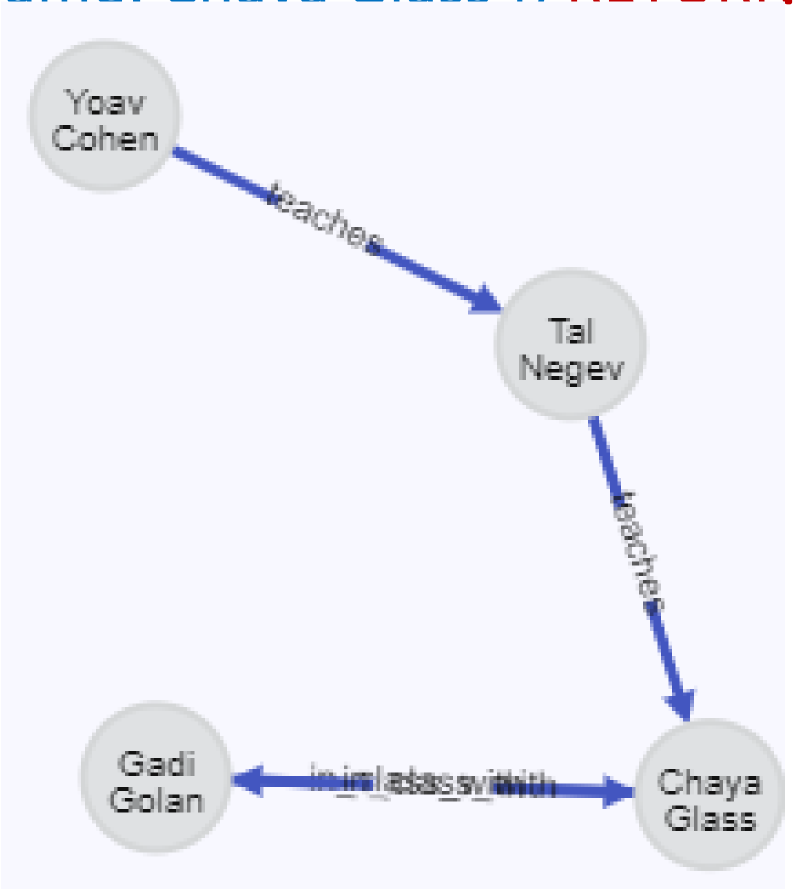


More Queries

- Find all names of those who teach the student Chaya Glass, or teach anyone who teaches the student Chaya Glass:

➢ MATCH (a)-[:teaches*1..2]->(b:student {name:'Chaya Glass'}) RETURN a.name

a.name
Tal Negev
Yoav Cohen



Paths

- MATCH p=(a {name:'Gadi Golan'})-[:KNOWS*2..4]->(b) RETURN p
  - Will return all paths of length 2 to 4 of type KNOWS, between Gadi Golan and others.
- MATCH p=shortestPath((s1:student {name:'Gadi Golan'})-[*]-(s2:student {name:'Tal Negev'})) RETURN p
  - Will return the shortest path (using any type of relation, and in any direction) between Gadi Golan and Tal Negev (of type students).
- ... RETURN LENGTH(p)

p
[(8:student {age:24, degree:"1", id:333, name:"Gadi Golan"}), (6)-[8:in_class_with]->(8), (6:student {age:21, degree:"1", id:111, name:"Chaya Glass"}), (7)-[6:teaches]->(6), (7:student {age:28, degree:"3", id:222, name:"Tal Negev"})]

WITH, ALL / ANY, IN, COLLECT, COUNT, AND, OR

s must appear in WITH part, so we group by it and can return s.name

- MATCH (c:course) WITH COLLECT(c) AS courses
  MATCH (s:student) WHERE ALL (x IN courses WHERE (s)-[:studies]->(x))
  RETURN s.name
  – Returns all students that study all courses.
- MATCH (s:student)-[:studies]->(c:course)WITH s, COUNT(c) as num_courses
  WHERE num_courses <= 4
  RETURN s.name
  – Returns all students that study at most 4 courses.
- MATCH (negev { name:"Tal Negev" })-[:friend]->(frOfNegev:student)-[:knows]->(st:student)
  WITH frOfNegev, COUNT(st) AS frCount WHERE frCount > 3
  RETURN frOfNegev
  – Returns all students that are Tal Negev's friend and know more than 3 students.

count() is an **aggregate function**. When using any aggregate function, result rows will be grouped by whatever is included in the WITH clause (**not** in the aggregate function).

Additional query examples

- Write a query that returns all the nodes that have any connectivity (of any length) with 'Tal Negev'

```
MATCH (a {name:'Tal Negev'})-[*]-(b)
RETURN DISTINCT b
```

- Write a query that returns all students that study all courses that 'Tal Negev' learns, but are under the age of 30.

```
MATCH (a {name:'Tal Negev'})-[:studies]->(c:course)
WITH COLLECT(c) AS negev_courses MATCH (s:student) WHERE
s.age < 30 AND ALL (x IN negev_courses WHERE (s)-[:studies]->(x))
 RETURN s
```