

# NoSQL Database Management Systems

Amos Azaria

# 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!

# Why use NoSQL?

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

# What is the Price-tag?

## (i.e. What Do We Give-up?)

- 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.)

# Database management system usage

(<http://db-engines.com/en/ranking>)

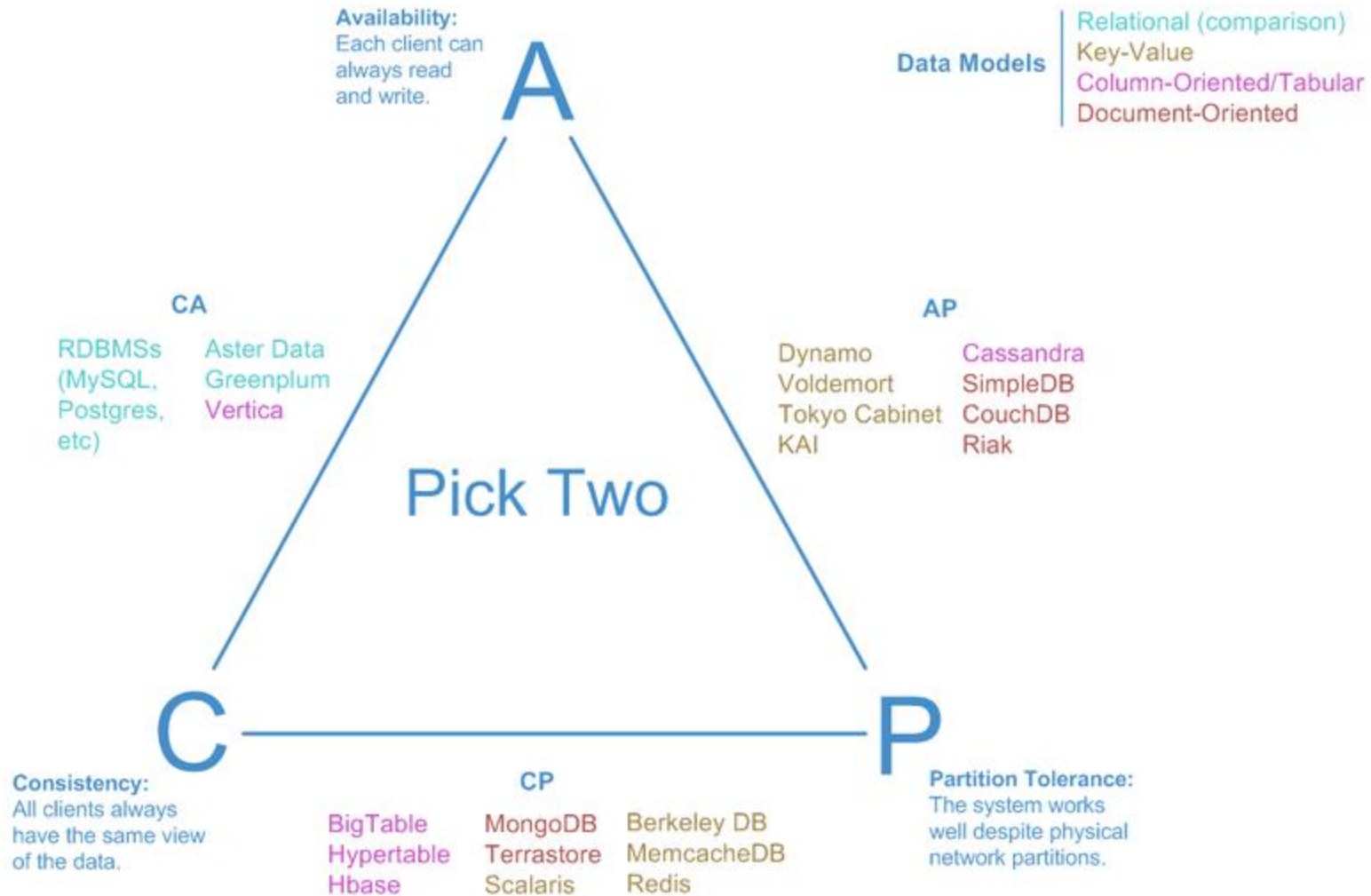
The leading RDBMS have started to add support for additional models

Rank			DBMS	Database model	Score	Feb 2019	Jan 2019	Feb 2018
Feb 2019	Jan 2019	Feb 2018						
1.	1.	1.	Oracle +	Relational, Multi-model i	Relational DBMS, Document store, Graph DBMS, RDF store	473.56	+7.45	+85.18
2.	2.	2.	MySQL +	Relational, Multi-model i		395.09	+7.91	+58.67
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model i		179.42	-0.43	-10.55
4.	4.	4.	PostgreSQL +	Relational, Multi-model i		149.45	+0.43	+22.43
5.	5.	5.	MongoDB +	Document	395.09	+7.91	+58.67	
6.	6.	6.	IBM Db2 +	Relational, Multi-model i	179.42	-0.43	-10.55	
7.	7.	↑ 8.	Redis +	Key-value, Multi-model i	149.45	+0.43	+22.43	
8.	8.	↑ 9.	Elasticsearch +	Search engine, Multi-model i	145.25	+1.81	+19.93	
9.	9.	↓ 7.	Microsoft Access	Relational	144.02	+2.41	+13.95	
10.	10.	↑ 11.	SQLite +	Relational	126.17	-0.63	+8.89	
11.	11.	↓ 10.	Cassandra +	Wide column	123.37	+0.39	+0.59	
12.	↑ 13.	↑ 17.	MariaDB +	Relational, Multi-model i	83.42	+4.60	+21.77	
13.	↓ 12.	13.	Splunk	Search engine	82.81	+1.39	+15.55	
14.	14.	↓ 12.	Teradata +	Relational	75.97	-0.22	+2.98	
15.	15.	↑ 18.	Hive +	Relational	72.29	+2.38	+17.23	
16.	16.	↓ 14.	Solr	Search engine	60.96	-0.52	-2.91	
17.	17.	↓ 16.	HBase +	Wide column	60.28	-0.12	-1.43	
18.	18.	↑ 19.	FileMaker	Relational	57.79	+0.64	+3.43	
19.	19.	↑ 20.	SAP HANA +	Relational, Multi-model i	56.55	-0.09	+9.19	
20.	↑ 21.	↓ 15.	SAP Adaptive Server	Relational	55.75	+0.71	-7.74	
21.	↓ 20.	21.	Amazon DynamoDB +	Multi-model i	54.95	-0.15	+15.07	
22.	22.	22.	Neo4j +	Graph	47.86	+1.06	+8.04	

# 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 an arbitrary number of messages being dropped (or delayed) by the network between nodes.

# Visual Guide to NoSQL Systems



# From ACID to **BASE**

- **Basically Available**: data is mostly available.
- **soft State**: state may change even with no updates (since older updates are still propagating).
- **Eventual consistency**: if we let the data propagate enough time, it will become consistent.



# NoSQL database types:

Type	Example	
Key-Value Store	 redis	 riak
Wide Column Store	 H-BASE	 cassandra
Document Store	 mongoDB	 CouchDB relax
Graph Store	 Neo4j	 InfiniteGraph The Distributed Graph Database
RDB	 Apache Jena	
Search-Engine DB	 Elastic Search	 Apache Solr

For each of the types we will go into details and learn how to operate the leading database of that type!

<http://www.algoworks.com/blog/nosql-database/>

# 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"

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

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

➤ 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



# INCR and INCRBY

➤ SET bottles 5

➤ INCR bottles

(integer) 6

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

➤ INCRBY bottles -3

(integer) 3

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

➤ INCR name1

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



# Lists (RPUSH, LPUSH, LRANGE)

➤ RPUSH user:571:items "chair"

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

➤ RPUSH user:571:items "table"

➤ RPUSH user:571:items "water"

➤ LPUSH user:571:items "cup"

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

➤ LRANGE user:571:items 1 2

LRANGE key x y  
Returns items from x to y

1) "chair"

2) "table"

➤ LRANGE user:571:items 0 -1

-1 means until the end of the list

1) "cup"

2) "chair"

3) "table"

4) "water"

You can provide the full list in a single command

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



# Hashes(HSET, HGET, HMSET, HGETALL)

➤ HSET user:302 first\_name "Tamar"

➤ HSET user:302 last\_name "Cohen"

➤ HGET user:302 first\_name  
"Tamar"

HMSET: for setting multiple  
attributes in a single command

➤ HMSET user:302 degree 3 gender "female"

➤ HGET user:302 degree  
"3"

➤ 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"
- R PUSH 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)
  - SUNIOUN 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 command sets a time in seconds to which a value will expire.
  - `EXPIRE user:302 100`
  - `TTL user:302`  
(integer) 92

# Wide-Column Store

- Every key identifies a row of a variable number of elements.
- Have a strict format: Each key may be one level (or maybe two level) dictionaries. (Not any XML.)
- Therefore, 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**
- **Cassandra:** Writes in Cassandra are awfully cheap. Cassandra is optimized for high write throughput, and almost all writes are equally efficient. 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.
- **RMDB: Minimize Data Duplication**
- **Cassandra:** 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.



# 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};
```

Number of replicas of data on multiple nodes

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

```
➤ 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.



# 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



# 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)"



# Advanced Querying

Multiple primary keys (composite key), note the order

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

grade

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

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

grade

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

Partition Key

Clustering key

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

No need to provide all primary keys

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/constraints are derived from the way Cassandra stores (and sorts) the data.

Think of the keys as a path in a file system. You must specify the whole path until the directory you are querying.



# Cassandra's storage method

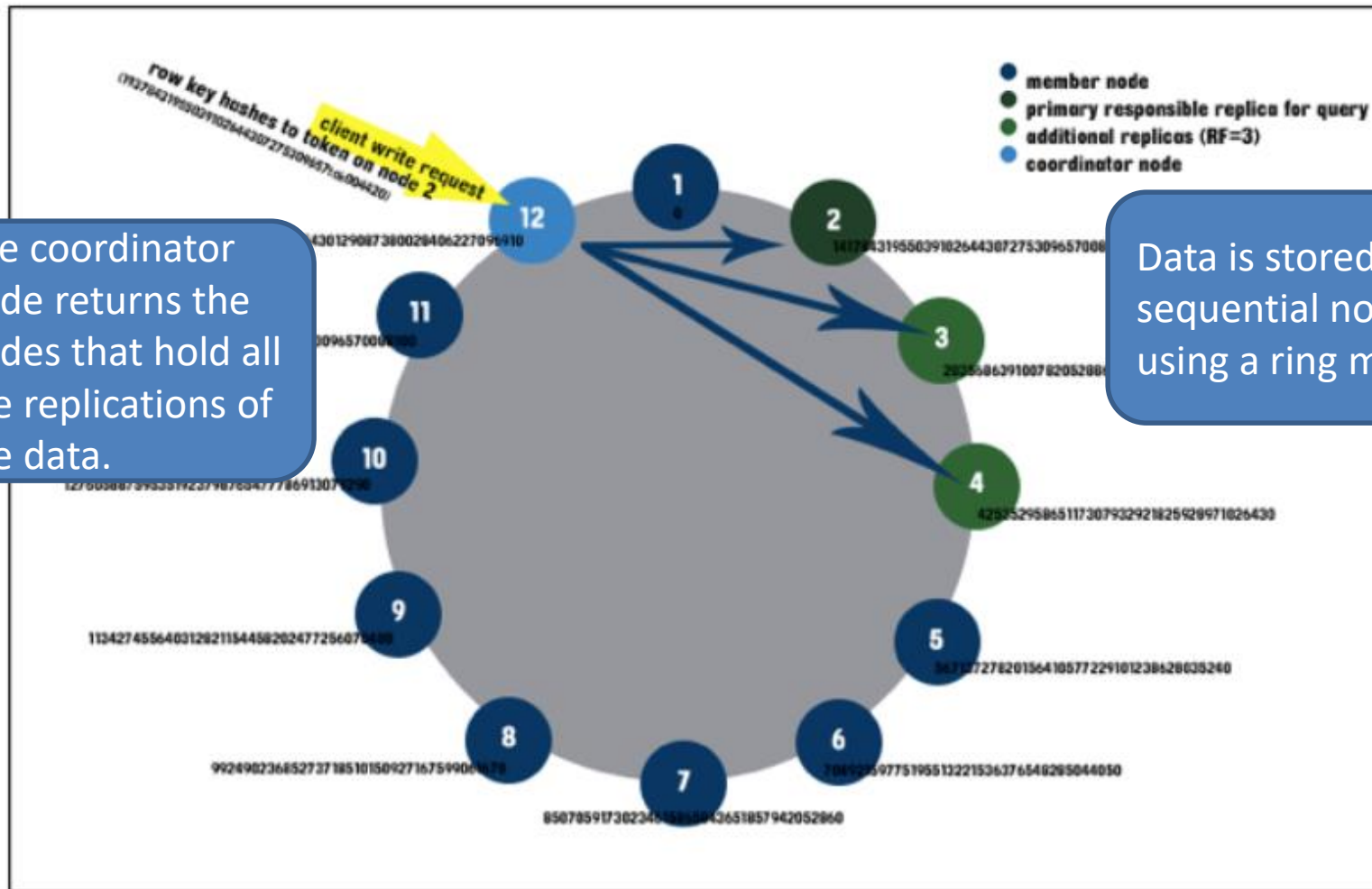











Figure 1 A 12 node cluster using RandomPartitioner and a keyspace with Replication Factor (RF) = 3, demonstrating a client making a write request at a coordinator node and showing the replicas (2, 3, 4) for the query's row key



# 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

- The partition key can include more than a single key.
- 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.
- 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` ✗

# 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 usually written in XML or JSON.
- Unlike in the key-value store, the database has some level of "understanding" of these documents.
- Our example: MongoDB





# 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"}  
})

JSON format

Inserting multiple  
document using a list

➤ 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"
    }
  }
}
```

## ➤ 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" } }
{
  "_id" : ObjectId("589af4a844a5653a862dd68e"),
  ...
}
```



# Inequalities in MongoDB

(Slide is provided just for completeness)

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

# 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)

➤ `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

➤ `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: "mmm", qty: 5, price: 2.5 },  
           { sku: "nnn", qty: 5, price: 2.5 } ]  
}
```

SKU = Stock Keeping Unit  
is an item identifier.

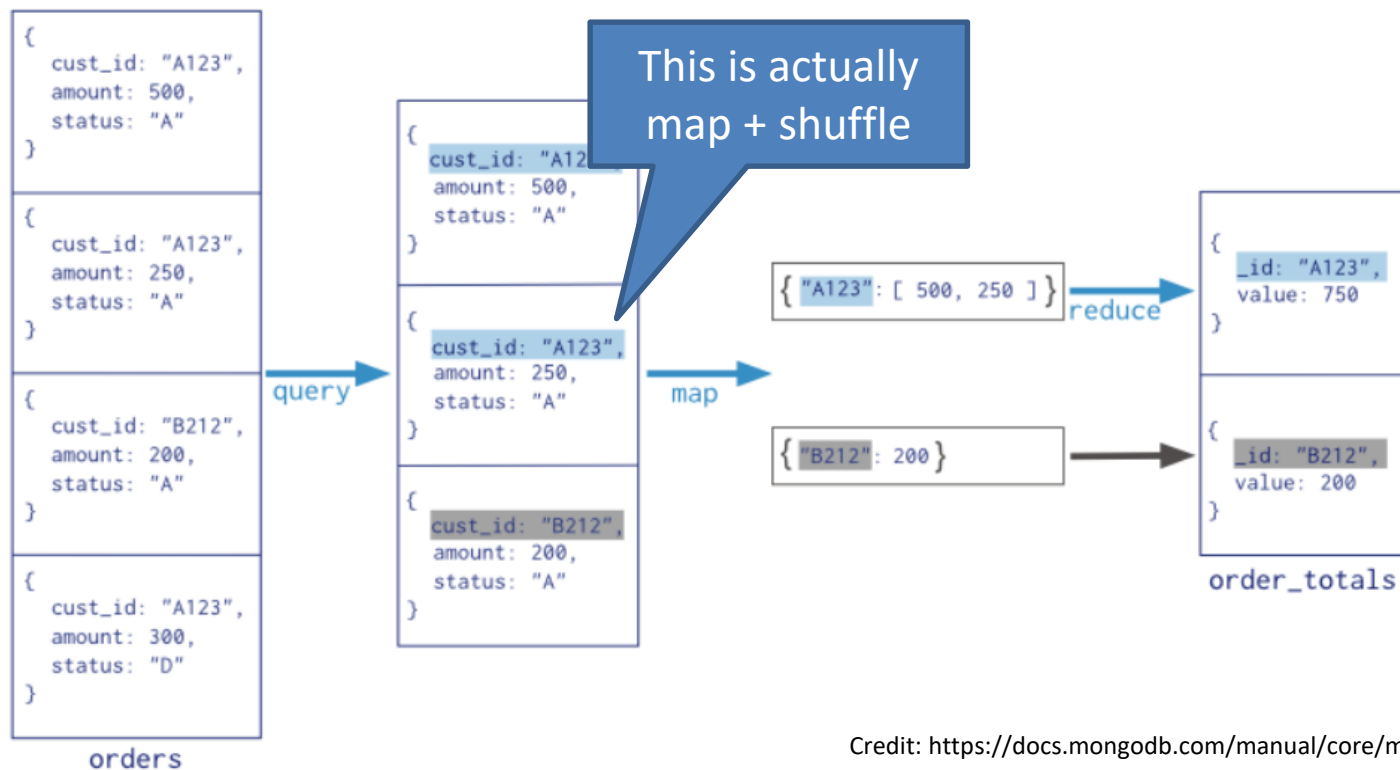
Can we do this in a  
relational DB, With SQL?

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'.

# mapReduce() (cont.)

Collection  
↓  
db.orders.mapReduce(  
  map     → function() { emit( this.cust\_id, this.amount ); },  
  reduce  → function(key, values) { return Array.sum( values ) },  
  {  
    query: { status: "A" },  
    out: "order\_totals"  
  }  
)





# 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: "mmm", total: 5543, avg: 5.4 },  
 { sku: "nnn", total: 1253, avg: 3.6 },  
 { sku: "a453", 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 = {  
      count: 1,  
      qty: this.items[idx].qty };  
    emit(key, value);  
  }  
};
```

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



# reduce

- The reduce function will go over the SKUs and lists 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.count += countObjVals[idx].count;  
            reducedVal.qty += countObjVals[idx].qty;  
        }  
        return reducedVal;  
    };
```



# 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;  
    };
```

# 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: elasticsearch







# Elasticsearch

- 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 ...\\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).

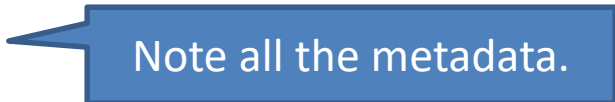
# Adding Documents (items)

- There is no need to create an Index (Database) or a Type (a table), we can right away insert a document using:
  - `curl -XPUT "http://localhost:9200/university/students/111" -d '{"FirstName": "Chaya", "LastName": "Glass", "age": "21", "Address": { "Street": "Hatamr 5", "City": "Ariel", "Zip": "40792" }}'`
  - `curl -XPUT "http://localhost:9200/university/students/333" -d '{"FirstName": "Gadi", "LastName": "Golan", "age": "24"}'`
  - `curl -XPOST "http://localhost:9200/university/students" -d '{"FirstName": "Tal", "LastName": "Negev", "age": "28"}'`

POST without id, will  
generate id automatically

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,"found":true,`  
`"_source":{"FirstName": "Gadi", "LastName": "Golan", "age": "24"}}`  
Note all the metadata.
- HEAD only tests if document exists. (replace XGET with XHEAD in the command above).
- DELETE deletes the document

# Search (SELECT)

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

```
{"took":5,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{"total":3,"max_score":1.0,"hits":[{"_index":"university","_type":"students","_id":"222","_score":1.0,"_source":{"FirstName": "Tal", "LastName":"Negev", "age":"28"}}, {"_index":"university","_type":"students","_id":"333","_score":1.0,"_source":{"FirstName": "Gadi", "LastName": "Golan", "age": "24"}}, {"_index":"university","_type":"students","_id":"111","_score":1.0,"_source":{"FirstName": "Chaya", "LastName": "Glass", "age": "21", "Address": { "Street": "Hatamr 5", "City": "Ariel", "Zip": "40792"}}}]}}
```

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

```
SELECT * FROM students WHERE LastName='Negev'
```

```
{"took":21,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{"total":1,"max_score":0.2876821,"hits":[{"_index":"university","_type":"students","_id":"222","_score":0.2876821,"_source":{"FirstName": "Tal", "LastName": "Negev", "age": "28"}}]}}
```

# Advanced Search (bool query)

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

```
{ \"query\" : 
```

Boolean combination of other queries.

```
{ \"bool\" : 
```

```
{ \"filter\" : 
```

All students living in Ariel

```
{ \"match\" :  
  { \"Address.City\" : \"Ariel\" } },
```

```
\"filter\" : 
```

```
{ \"range\" : 
```

Students under 30

```
{ \"age\" :  
  { \"lt\" : 30 } } } } } }
```

```
{\"took\":6,\"timed_out\":false,\"_shards\":{\"total\":5,\"successful\":5,\"failed\":0},\"hits\":{\"total\":1,\"max_score\":0.0,\"hits\":[{\"_index\":\"university\",\"_type\":\"students\",\"_id\":\"111\", \"score\":0.0, \"_source\":{\"FirstName\":\"Chaya\", \"LastName\":\"Glass\", \"age\": \"21\", \"Address\": { \"Street\": \"Hatamr 5\", \"City\": \"Ariel\", \"Zip\": \"40792\"}}}]}}
```

# 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).

# String Queries

- Let's first add some descriptions to the students' documents, using `_update`:
  - `curl -XPOST http://localhost:9200/university/students/111/_update -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/222/_update -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/AVohwsYTVPDjS737L8vs/_update -d '{"script": {"ctx._source.description = "\\\"Doesn't know anything. Goes on trips all day, never showed-up to a single lesson.\\\""}}'`

# String Queries (cont.)

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

```
{"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."}]}
```

```
curl -XGET http://localhost:9200/university/students/_search -d '{"query":{"match":{"description":"doesn't show-up learns"}}, "highlight":{"fields":{"description":{"_source":{"FirstName":"Tal","LastName":"Negev","age":"28","description":"Doesn't show-up to lessons, but is very smart and learns a lot."}}}}}'
```

```
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!



# Exact Phrase (Like " " in Google)

- If we want the exact phrase, we can use "match\_phrase" instead of "match".
- We can also mix "match" and "match\_phrase".
- If we use a query URI, we can just add " ", e.g.:
  - curl -XGET  
"http://localhost:9200/university/students/\_search?  
q=description:\"works hard\""

# Information Retrieval (TF-IDF)

## Document Ranking

- Suppose we have a query (Q) and many possible documents ( $D=\{d_1, d_2, \dots\}$ ). How can we rank these documents based on the query?

# Document Ranking Example

- 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?

# 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.

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

d: current document  
D: full corpus (of documents)  
k: word in document/query  
Q: set of words in query

TF

IDF

# 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
#Doc	2	2	3	1	3	3	2	

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

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

Doc	Tf-Idf score
D1: Donald Trump is United States' president.	$(1/6)*\log(4/2)+(1/6)*\log(4/1)+(1/6)*\log(4/3)+(1/6)*\log(4/2)=0.736$
D2: We are the most united out of all the people and of all the places.	$(3/15)*\log(4/3)+(2/15)*\log(4/3)+(1/15)*\log(4/3)=0.166$
D3: The United States of America is united again, who is more united than it?	$(1/14)*\log(4/2)+(1/14)*\log(4/2)+(1/14)*\log(4/3)+(1/14)*\log(4/3)+(3/14)*\log(4/3)+(1/14)*\log(4/2)=0.363$
D4: Who would like to take the box out of the kitchen?	$(\log(4/2)+2*\log(4/3)+\log(4/3))/11=0.204$

# Graph Databases

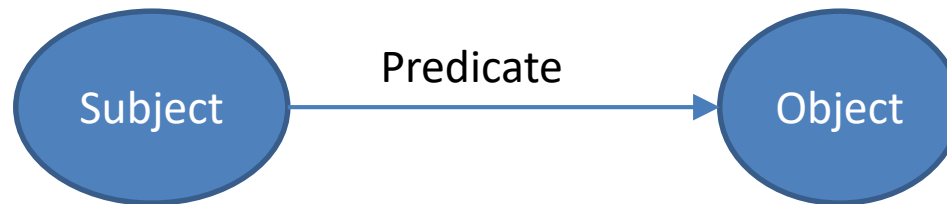
- Store information as graphs, with nodes and relations between the nodes.
- Allow interesting queries such as, finding all the friends of a person, all their friends, and so on until 10 levels.

# 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.

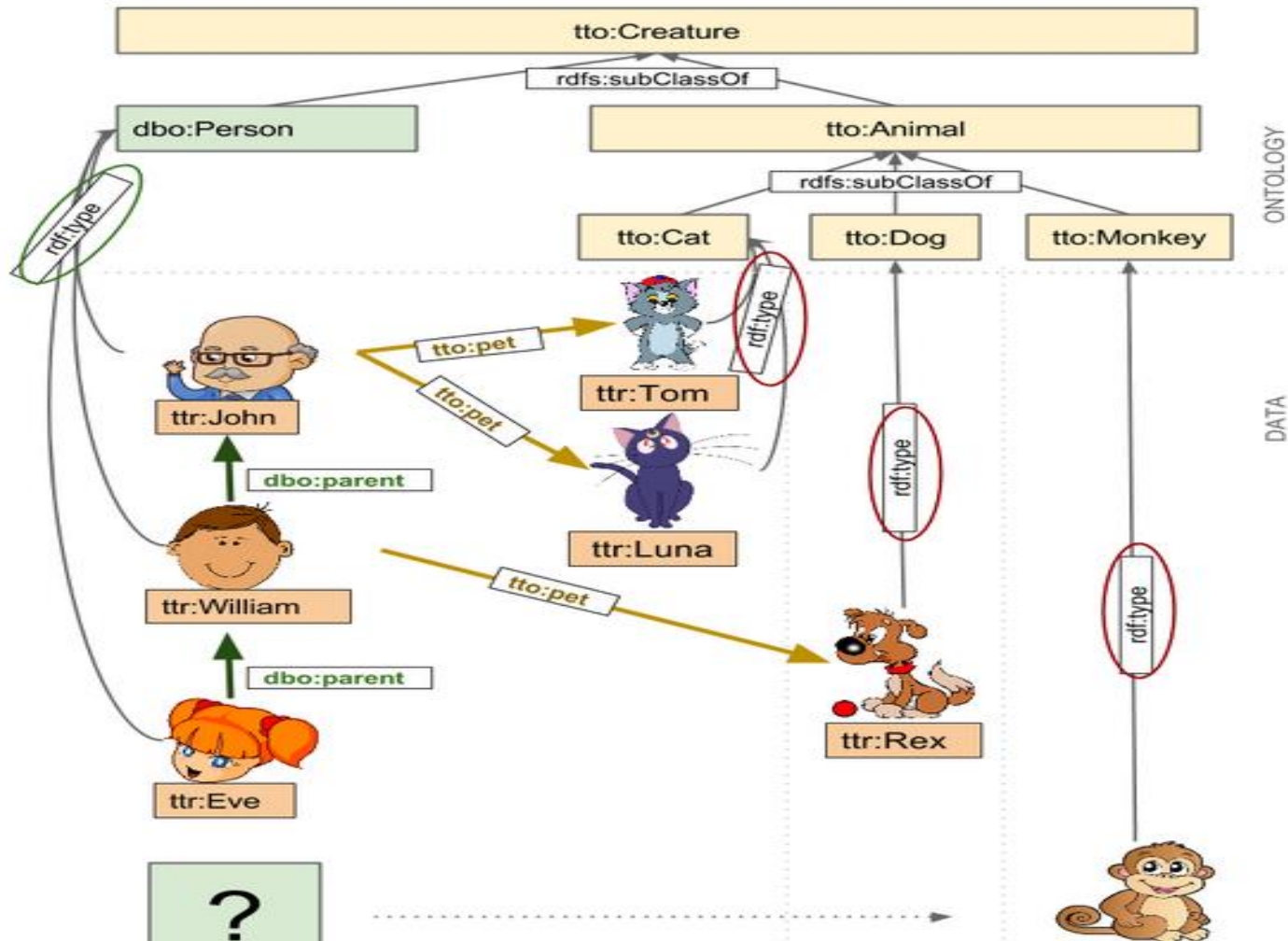


# ARQ / SPARQL

- 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"



# Select all Persons

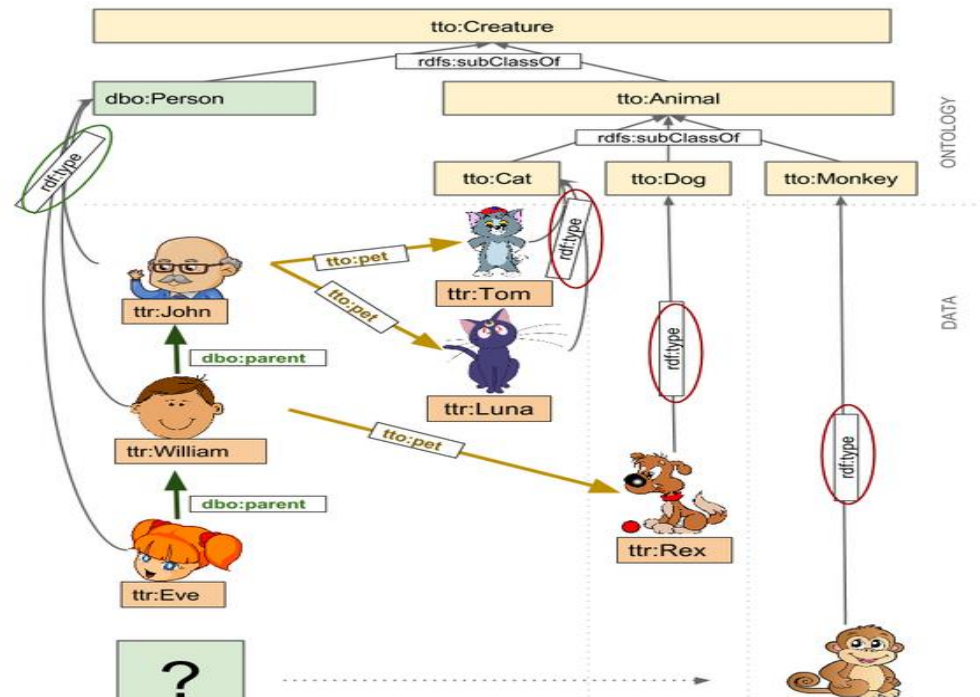
- SELECT ?something WHERE {?something rdf:type dbo:Person .}

**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 / rdf:type tto:Cat .}  
    }  
}

/ Concatenates  
relations

person
ttr:Eve
ttr:William



# Select all Creatures (using UNION)

- ```
SELECT ?thing WHERE {  
  {  
    ?thing a ?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 (simpler and more correct)

- `SELECT ?thing WHERE {  
 ?thing a / rdfs:subClassOf+ tto:Creature .  
}`

+ is 1 or more of same predicate  
(\* is 0 or more)  
(? is 0 or 1)

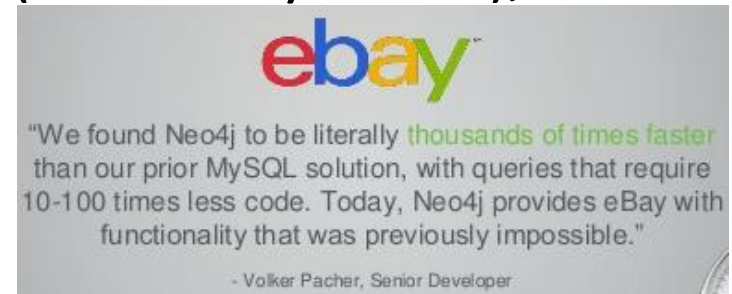
| thing           |
|-----------------|
| ttr:Eve         |
| ttr:John        |
| ttr:William     |
| ttr:LunaCat     |
| ttr:TomCat      |
| ttr:RexDog      |
| ttr:SnuffMonkey |

# General Graph Databases

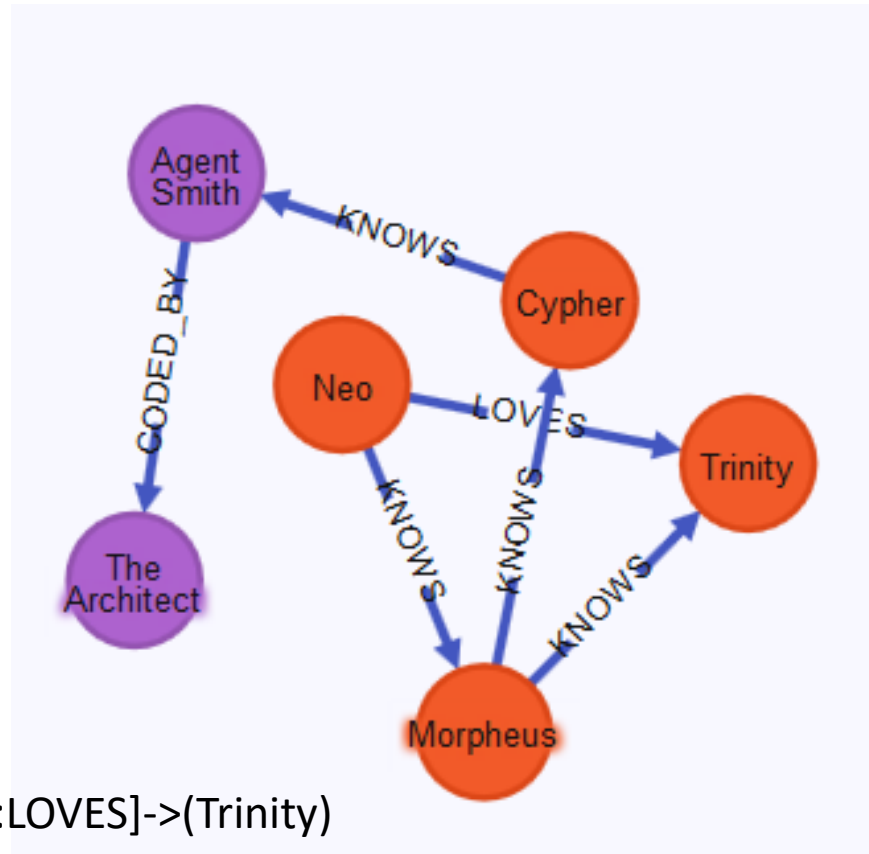
- Graph databases are not limited to RDF and SPARQL.
- The more general graph databases allow more complex queries (not only triples).

# Neo4J

- Name comes from "The Matrix".
- Query Language called Cypher.
- ACID (almost...)
- 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 modifications remain open-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 (SELECT a node)

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

# 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).

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

- `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" })-[:friends]->(frOfNegev:student)-[:knows]->(n:student) WITH frOfNegev, COUNT(frOfNegev) AS frCount WHERE frCount > 3 RETURN frOfNegev`
  - Returns all students that are Tal Negev's friend and know more 3 students.

s must appear again in  
WITH part, so we can  
return s.name

# 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`