BIG DATA I

NOSQL

What is a data scientist

http://i.stack.imgur.com/eLrhI.png

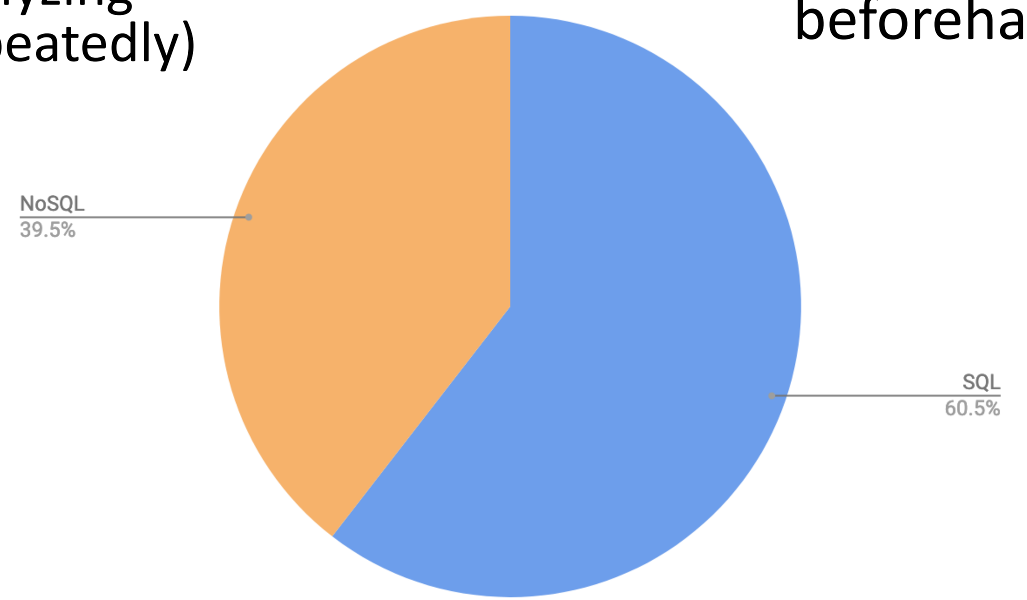
a data scientist should be able to
run a regression, write a sql query, scrape a web site, design an experiment, factor matrices, use a data frame, pretend to understand deep learning, steal from the d3 gallery, argue r versus python, think in mapreduce, update a prior, build a dashboard, clean up messy data, test a hypothesis, talk to a businessperson, script a shell, code on a whiteboard, hack a p-value, machine-learn a model.
specialization is for engineers.

JOEL GRUS

Where do you store your data?

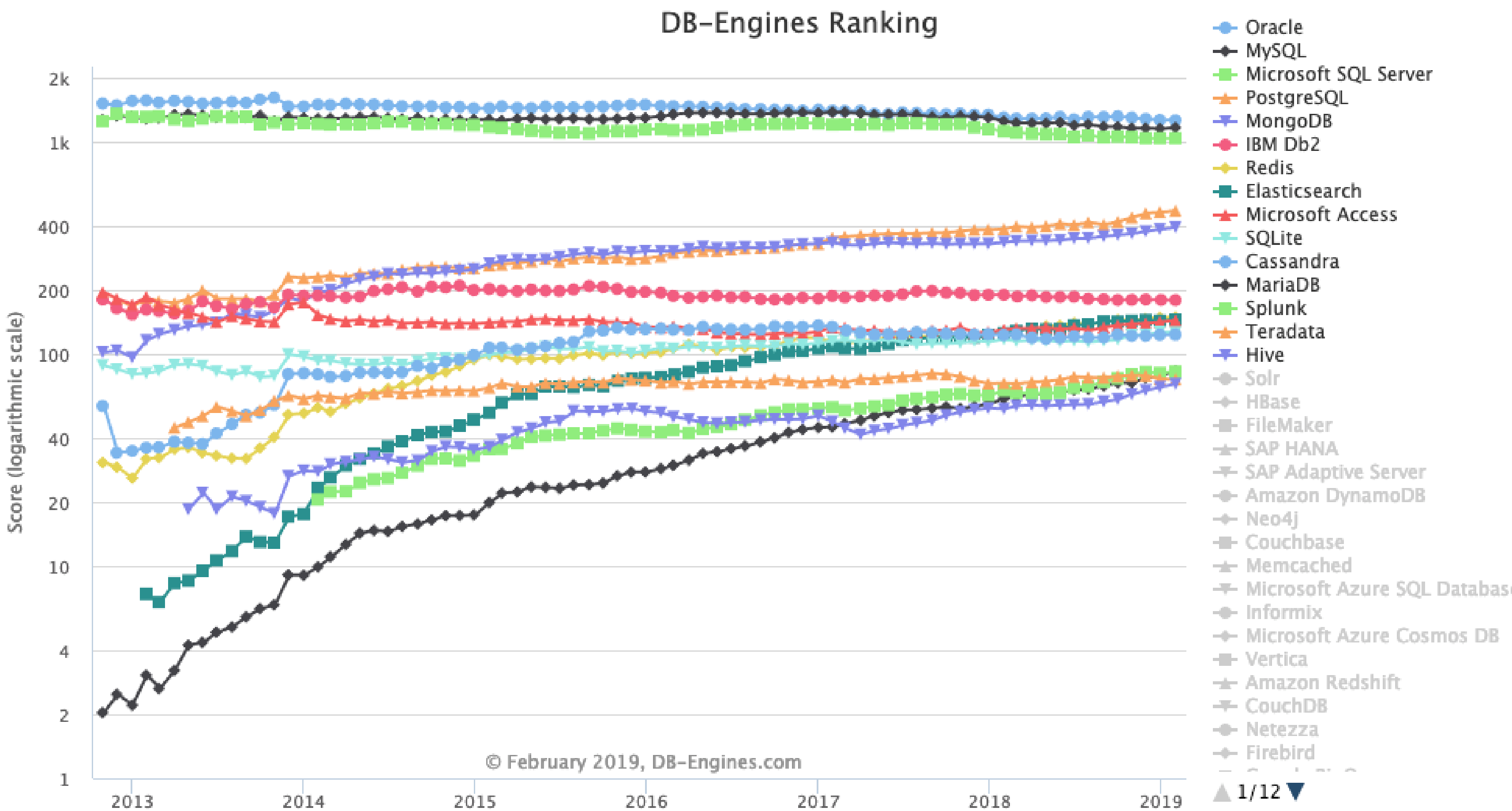
- Files:
  - quick and easy
- But huge cost:
  - For searching
  - Maintaining
  - Analyzing (repeatedly)

- We need structured storage
  - How much structure
  - Do we know beforehand?



https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/

DB Popularity Trends



How can I chose my DB needs

- Do you have a schema?
  - Do you need a strict schema?
  - Is it fixed?
- What are you going to use it for?
  - Live transaction system
    - Reliability?
  - Analytics
    - Multipurpose?
- Scalability?
  - How big are your data?
  - How fast do they grow?

SQL

RELATIONAL DBS

Relational Schemas & SQL

What is SQL good at?

- Structured data
  - Tabular represented by rows
  - When you know the data you are collecting
- Clean management
  - A good schema helps the good management of your data
- Structured queries
  - Create, Read, Update, Delete operations are optimized by sophisticated mechanisms (e.g. indexes)
    - Most NoSQL employ simpler access mechanisms
- BigData/NoSQL and SQL technologies are complementary
  - A lot of NoSQL technologies are implementig SQL-like interfaces

SQL is declarative

- SQL is a **declarative** language:
  - You specify **what** you want (declare) and not **how** you want it (define)
- 4 main types of commands
  A. Database object manipulation (Create Table/View, Alter Table)
  B. Data modification (Insert, Update, Delete)
  C. Data queries (Select…)
  D. Data control (Grant, Revoke, Commit, Rollback)

Creating Tables

```
CREATE TABLE offices(officeCode VARCHAR(10), city VARCHAR(50),
        phone VARCHAR(50) NOT NULL,  addressLine1 VARCHAR(50) UNIQUE,
        addressLine2 VARCHAR(50) DEFAULT NULL,
        CONSTRAINT pk PRIMARY KEY (officeCode) );
```

```
CREATE TABLE  employees (  employeeNumber INT(11),  lastName  VARCHAR(50),
    firstName VARCHAR(50),  email  VARCHAR(100) ,  officeCode VARCHAR(10),
    reportsTo INT(11),  jobTitle VARCHAR(50),
    CONSTRAINT pk PRIMARY KEY (employeeNumber),
    CONSTRAINT fk_employee_department FOREIGN KEY (officeCode) REFERENCES
    offices(officeCode)
);
```

Data insertion

- Insert Values:
  - INSERT INTO *table_name* VALUES (value1, …, valueN);
  - If we do not have a value for a specific field we put NULL
  - Otherwise :

    INSERT INTO *table_name* (fieldnameA, fieldnameB, fieldnameC) VALUES (fieldvalueA, fieldvalueB, fieldvalueC);
- UNIQUE fields. Assume:
  - employeeNumber is unique and mandatory but not a key

```
DROP TABLE IF EXISTS `employees`;
CREATE TABLE employees(
   employeeNumber INTEGER UNIQUE NOT NULL,
   lastName VARCHAR(50) NOT NULL,
.....
);
```

INSERT

```
INSERT INTO employees
(employeeNumber,lastName,firstName,extension,email,officeCode,reportsTo,jobTitle)
VALUES(1056, 'Patterson', 'Mary', 'x4611', 'mpatterso@classicmodelcars.com', '1', 1002,
'VP Sales');

INSERT INTO employees
VALUES(1143, 'Bow', 'Anthony', 'x5428', 'abow@classicmodelcars.com', '1', 1056, 'Sales
Manager (NA)');

INSERT INTO employees
VALUES(1056, Thompson', 'Leslie', 'x4065', 'lthompson@classicmodelcars.com', '1',
1143, 'Sales Rep');

INSERT INTO employees
VALUES(1337, 'Bondur', 'Loui', 'x6493', 'lbondur@classicmodelcars.com', '4', 1102,
'Sales Rep');
```

Data Queries

SELECT * FROM *table_name*;

SELECT *column_name,column_name*
FROM *table_name*
WHERE *column_name operator value*;

- Basic operators:

Operator	Description
=	Equal
<>, !=	Not equal.
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

More SQL commands

- **UNION**
  - **SELECT** Dnumber **FROM** Dept_Locations
    WHERE Dlocation='Houston‘
    UNION
    SELECT Dnum **FROM** Project **WHERE** Plocation='Houston'
- **Wild cards**
  - **SELECT** * **FROM** Employee **WHERE** name **LIKE** 'Ja%'
- **Ordering**
  - **SELECT** Fname,Lname **FROM** employee
    WHERE salary>30000
    ORDER BY Salary **DESC**
- **NULL comparison**
  - **SELECT** * **FROM** employee **WHERE** superssn **IS NULL**
- The **DISTINCT** keyword can be used to return only distinct (different) values.
  - SELECT DISTINCT *column_name*,*column_name* FROM *table_name*;

SQL functions

- Useful aggregate functions:
  - **AVG**() - Returns the average value
  - **COUNT**() - Returns the number of rows
  - **FIRST**()/**LAST**() - Returns the first/last value
  - **MAX**() /**MIN**() - Returns the largest/smallest value
  - **SUM**() - Returns the sum
- Useful scalar functions:
  - **UCASE**() /**LCASE**()- Converts a field to upper/lower case
  - **MID**() - Extract characters from a text field
  - **LEN**() - Returns the length of a text field
  - **ROUND**() - Rounds a numeric field to the number of decimals specified
  - **FORMAT**() - Formats how a field is to be displayed
- Date functions
  - **NOW**() - Returns the current system date and time
  - **DATEDIFF**() - Returns the number of days between two dates
  - **YEAR/MONTH/DAY**() – Returns parts of a date :year/ month / day etc..

Aggregate Functions

- SELECT column_name, **aggregate_function**(column_name) FROM table_name WHERE column_name operator value **GROUP BY column_name;**
- "Where" can not be applied in the values of an aggregate function
  - SELECT column_name, aggregate_function(column_name) FROM table_name WHERE column_name operator value GROUP BY column_name **HAVING aggregate_function(column_name) operator value**;

JOINS

```
SELECT * FROM employees as E
JOIN customers as C
ON C. salesRepEmployeeNumber=
E. employeeNumber
WHERE  customerName='Gift Ideas Corp.'
```

- We rename relations for easier access with "**as**" and so we can reference tables with the same name (self-join)

Different JOINS

- EQUI JOIN
  - INNER JOIN (a.k.a. JOIN)
  - OUTER JOIN:
    - LEFT OUTER JOIN
    - RIGHT OUTER JOIN
    - FULL OUTER JOIN
- SEMI JOIN
  - WHERE fieldname **IN** (
    SELECT table2.fieldname FROM table 2)
  - WHERE fieldname **EXISTS** (
    SELECT table2.fieldname FROM table 2)
  - You can also replace the **IN** or **EXISTS** with logical operators

Indexes

- The joins are the most expensive operations
- Imagine having 2 files with lines that have common fields
  - How would you join them?
  - Not knowing where everything is makes the process slow
- Indexes improve retrieval and joining operations

MongDB

Graph DBs

NO SQL

The traditional SQL model

- Normalized Data
  - Minimize redundancy
- Joins

Traditional DBMSs

- Usually **traditional** means **relational** or **object-relational** (e.g., PostgreSQL, DB2, Oracle, MySQL, SQLServer).

- These DBMSs are the dominant choice for supporting business and in general **OnLine Transaction Processing** applications (OLTP). E.g., **banking** applications are characterized as OLTP.

- They are not designed for **Analytical Processing** (e.g., OLAP, Data Mining).

- Analytical systems contain TBs of data causing queries to exceed what can be done on a single server. Scaling-up the server (adding more resources) does not solve the problem.

The 3 Vs : SQL vs NoSQL

- Volume – large size of data
  - SQL – join pain
    - Create a set of all possible answers and then select the desired one
  - NOSQL
    - Adopt different models
    - Less expressive (- the graph model)

- Velocity – data rate of change
  - high write rates
  - Handle peaks
  - Schema changes over time
  - SQL
    - high write loads translate into a high processing costs
    - High schema volatility has a high operational cost

Variety

- data
  - regularly or irregularly structured,
  - dense or sparse,
  - connected or disconnected

Traditional DBMSs

These types of DBMSs show severe limitations due to challenges posed by big data.

One architectural feature that may not respond promptly is **consistency** (*the second of the ACID properties of transactions*)

Atomicity
Consistency
Isolation
Durability

Traditional DBMSs

Consistency Types

Strict: changes to the data are atomic and appear to take effect instantaneously. This is the highest form of consistency.

Sequential: Every client sees all changes in the same order they were applied.

Causal: All changes that are causally related are observed in the same order by all clients.

Eventual: When no updates occur for a period of time, eventually all updates will propagate through the system and all replicas will be consistent.

Weak: No guarantee is made that all updates will propagate and changes may appear out of order to various clients.

Brewer's CAP Theorem

Brewer's CAP theorem states that a distributed system is not possible to guarantee all three of the following properties simultaneously:

Consistency: all nodes see the same data at the same time

Availability: a guarantee that every request receives a response about whether it succeeded or failed

Partition Tolerance: the system continues to operate despite arbitrary message loss or failure of part of the system

Eric Brewer, “CAP twelve years later: How the "rules" have changed”, IEEE Explore, Volume 45, Issue 2 (2012), pg. 23-29

ACID vs. BASE

SQL databases

- **A**tomic

All operations in a transaction succeed or every operation is rolled back.

- **C**onsistent

On transaction completion, the database is structurally sound.

- **I**solated

Transactions do not interact with one another.

transactions appear to run sequentially.

- **D**urable

The results of applying a transaction are permanent, even in the presence of failures.

NOSQL

- **Ba**sic availability

The store appears to work most of the time.

- **S**oft-state

Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.

- **E**ventual consistency

Stores exhibit consistency at some later point (e.g., lazily at read time).

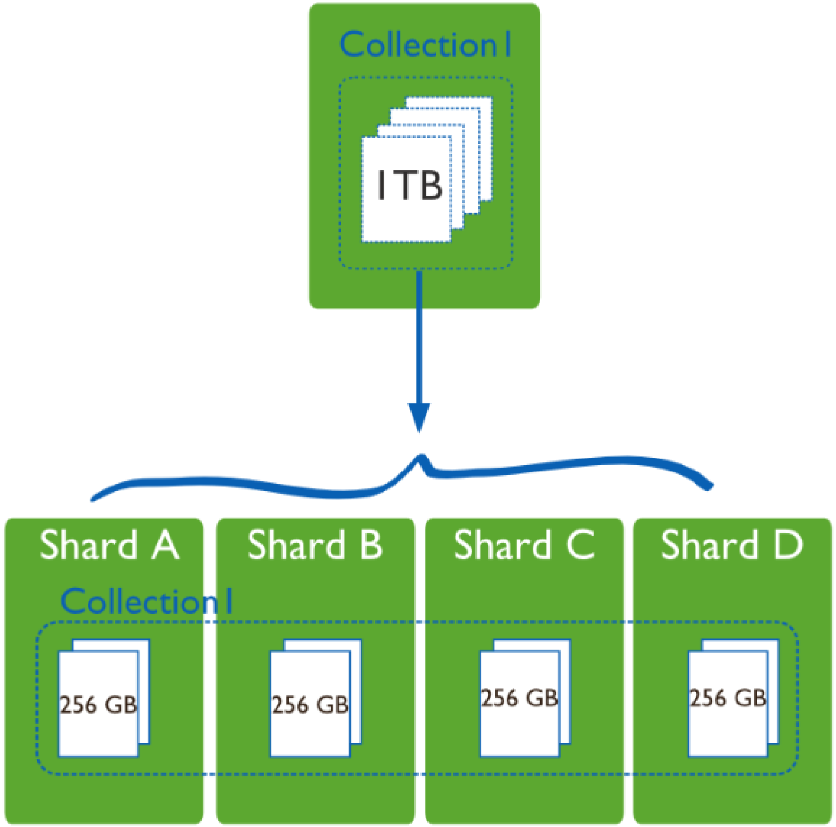
Vertical vs Horizontal scaling

- large data sets and high throughput applications challenge the capacity of a single server.
  - High query rates can exhaust the CPU capacity of the server.
  - Larger data sets exceed the storage capacity of a single machine. Finally, working set sizes larger than the system's RAM stress the I/O capacity of disk drives.
- To address these two basic approaches: **vertical scaling** and **sharding**.
- **Vertical scaling**:
  - add more CPU and storage resources to increase capacity.
  - Limitations: high performance systems with large numbers of CPUs and large amount of RAM disproportionately *more expensive* than smaller systems.
  - Additionally, cloud-based providers may only allow users to provision smaller instances. As a result there is a *practical maximum* capability for vertical scaling.
- **Sharding** (*horizontal scaling)*
  - divides the data set (based on key intervals) and distributes the data over multiple servers, or **shards**.
  - Each shard is an independent database,
  - Shards make up a single logical database.

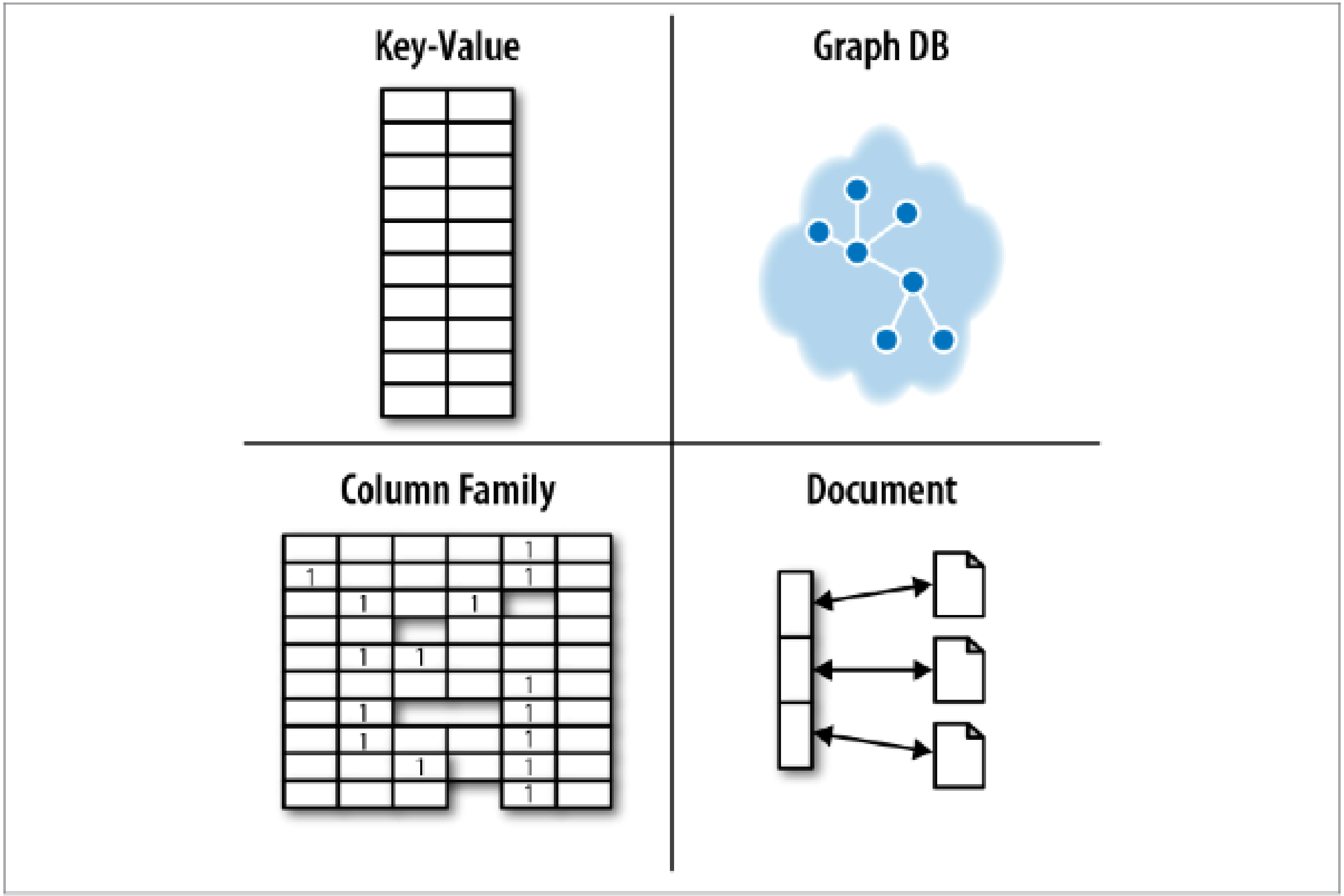
Sharding [4]

Scaling to support high throughput and large data sets:

- reduces the number of operations each shard handles.
    - Each shard processes fewer operations as the cluster grows. As a result, a cluster can increase capacity and throughput *horizontally*.
    - i.e. insert data means access only shard responsible for that record.
- reduces the amount of data that each server needs to store.
    - Each shard stores less data as the cluster grows.
    - For example, a 1TB database can be served by 4 256GB shards, or 40 25GB shards.



NOSQL quadrants

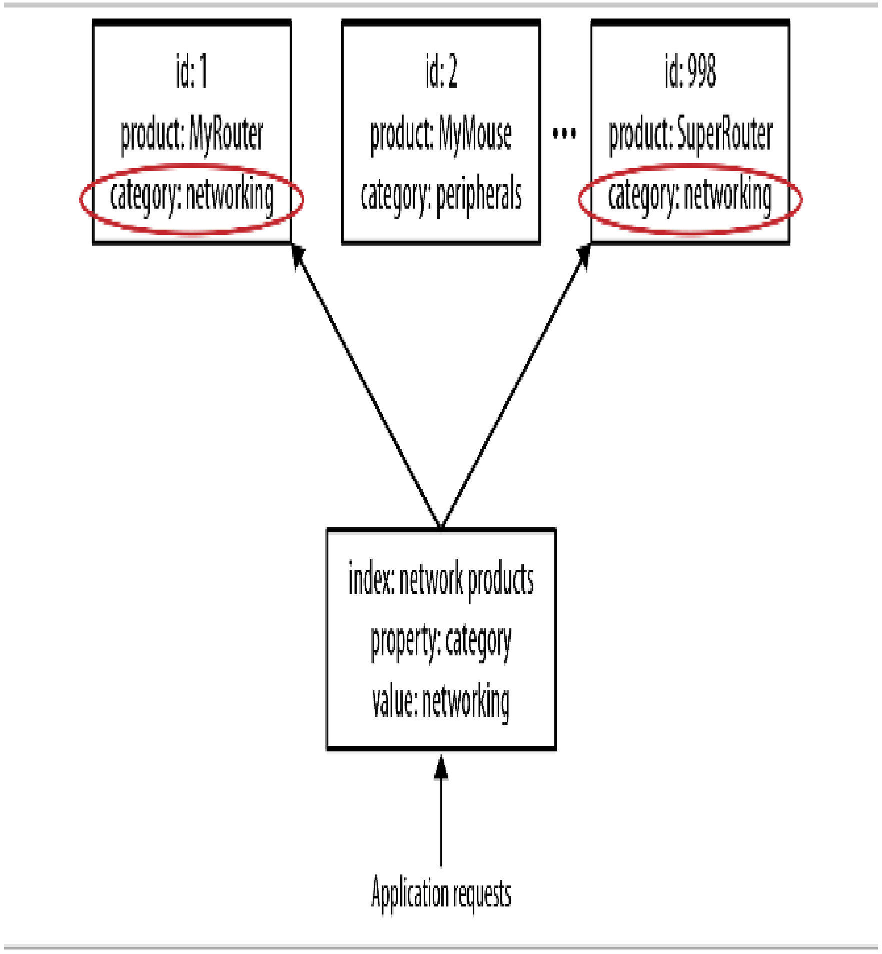


Document Stores

- hierarchically structured documents.
- Document databases store and retrieve documents
- Documents comprise maps and lists, allowing for natural hierarchies (i.e. JSON and XML).
- 2 way access:
  - By a key value
  - By an attribute value

Document Stores

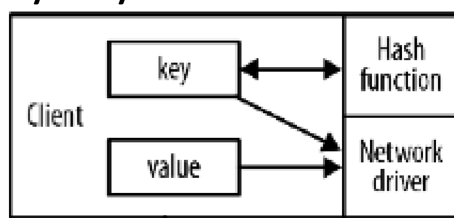
- 2 way access:
  - key value (id)
  - attribute value
- Transaction on single rows (documents)
- No lock mechanism supported
- Indexing on attributes (facilitating reads, complicating writes)
- Horizontal scaling - sharding
- Examples:
  MongoDB, RavenDB, CouchDB

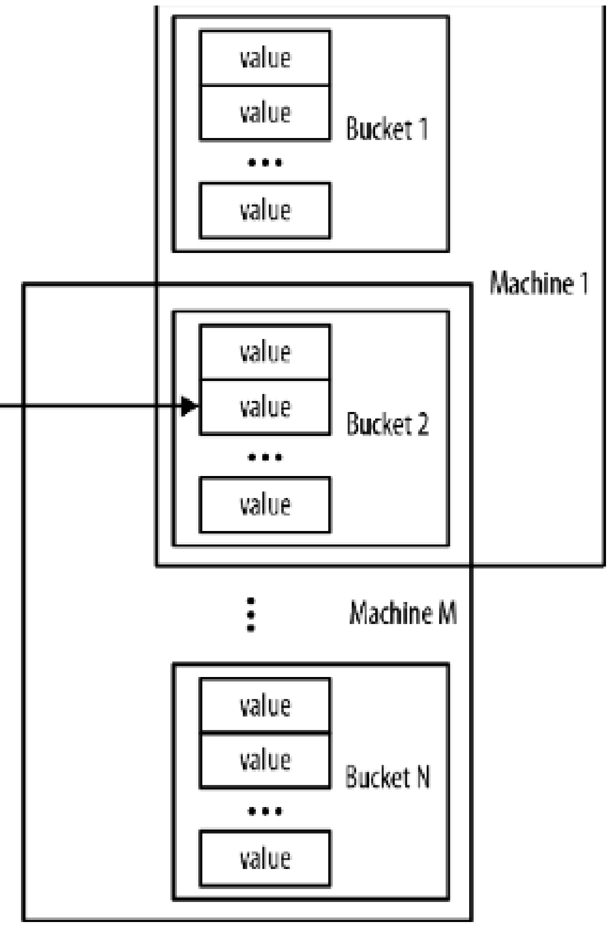


Key value stores

- large, distributed hashmap data structures

- store and retrieve values by key





- key space of the hashmap spread

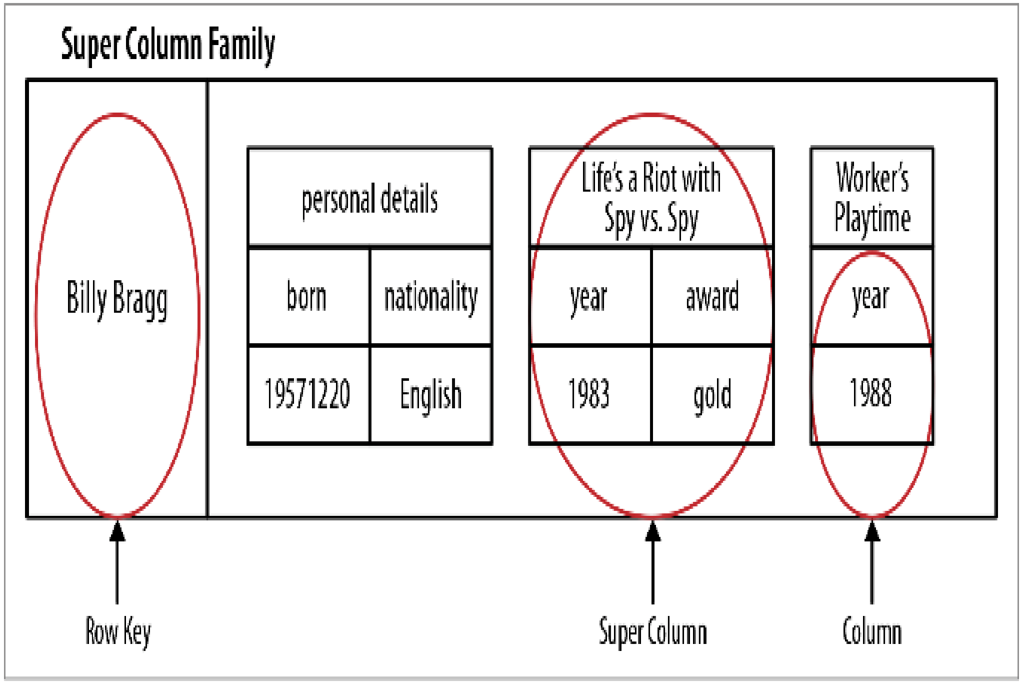
across buckets on the network.

- Buckets replicated for fault-tolerance acroos different machines
- Clients retrieve the whole value, and then filter out the unwanted parent or sibling data elements.
  - Inefficiency compared to document stores, which perform such operations on the server,

- i.e. Amazon Dynamo database - a platform designed for a nonstop shopping cart service with high availability

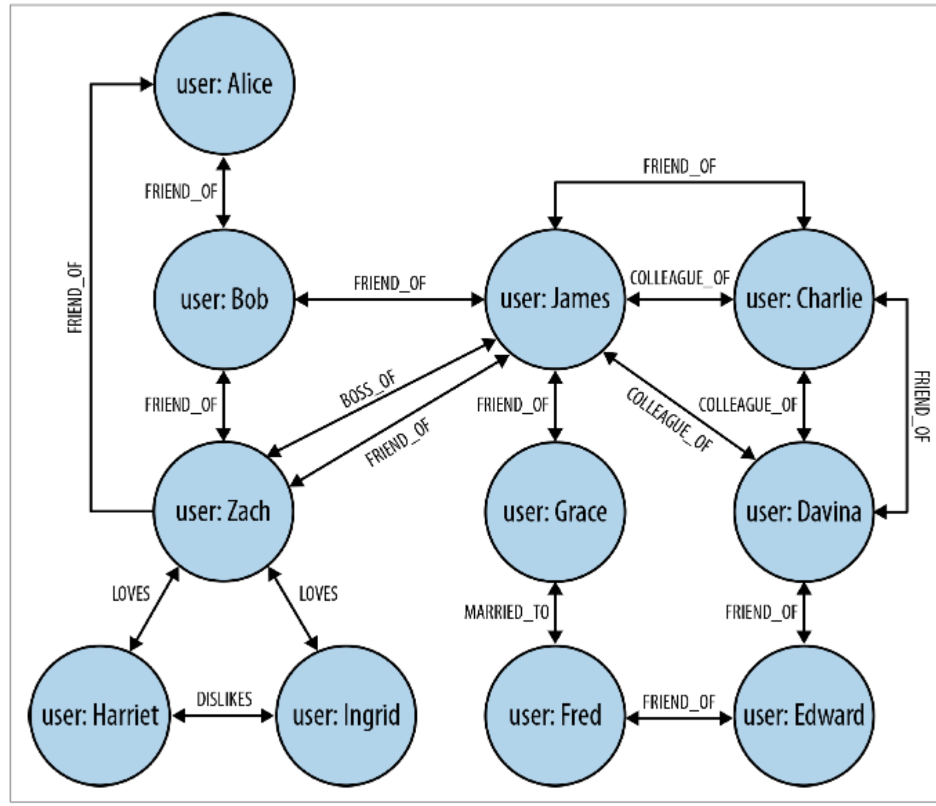
Column based approaches

- Google Bigtable - Hbase
- Model
  - sparsely populated table
  - rows can contain arbitrary columns and grouping them in *column families* (or super columns)
  - *Row keys* for provide natural indexing.
- Column databases are distinguished from document and key-value
  - more expressive data model,
- column family databases
  - reasonably expressive
  - operationally very competent.
  - still *aggregate stores*, just like document and key-value databases,
  - Lack performance for joins …
- More later ….



Graph databases

- Graphs represent rich set of relations
  - Lacking from SQL and keyvalue databases
- Property graph
  - contains nodes and relationships
  - Nodes contain properties (key-value pairs)
  - Relationships are named and directed, and always have a start and end node
  - Relationships can also contain properties



Graph databases

Two properties of graph databases

- *The underlying storage*
  - *native graph storage*

  optimized and designed for storing and managing graphs.

  - *serialize the graph data*

  into a relational database, object-oriented databases, or other types of general-purpose data stores.
- *The processing engine*
  - *index-free adjacency*

  connected nodes physically "point" to each other in the database (*native graph processing)*

  Best example: Neo4j

Graph databases – motivating example

Finding extended friends [3] in a social network

- relational database versus efficient finding in Neo4j
- experiment seeks to find friends-of-friends in a social network, maximum depth of five.
- Given any two persons chosen at random, is there a path that connects them that is at most five relationships long?
- social network
    - containing 1,000,000 people,
    - each with approximately 50 friends,

Depth	RDBMS execution time (s)	Neo4j execution time (s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Outline

- NoSQL
- Mongo DB
- Graph DBs

What is MongoDB

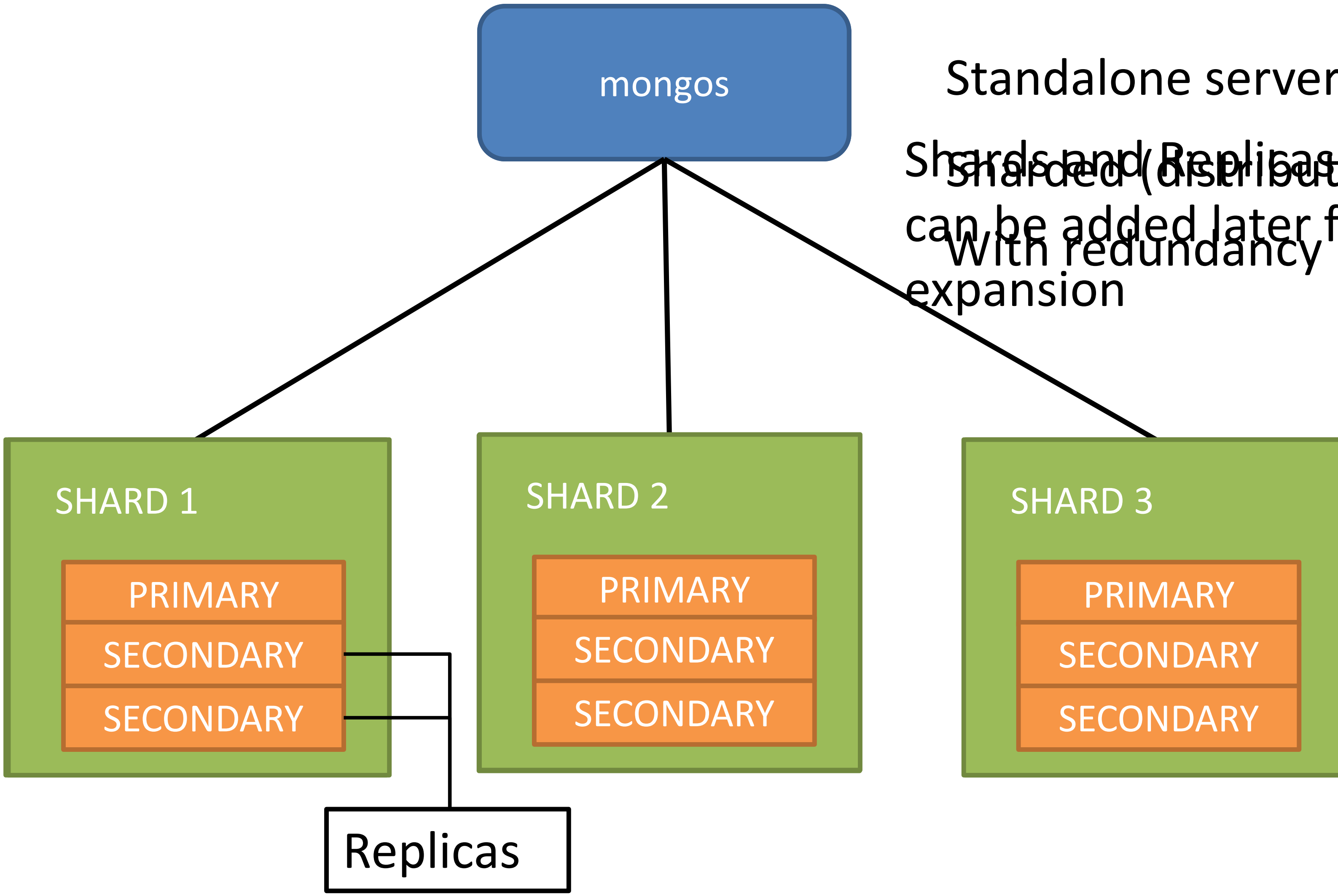
- Document-Oriented storage
  - The concept of a document replaces the row
- Utilizes “SQL” features
  - Index Support
- Easy to scale
  - Auto-Sharding
  - Auto-Balancing
- Querying
  - Native language : Javascript
- Map/Reduce

MongoDB Architectural versatility

Standalone server

Sharded (distributed)
With redundancy

Shards and Replicas can be added later for expansion



Document store

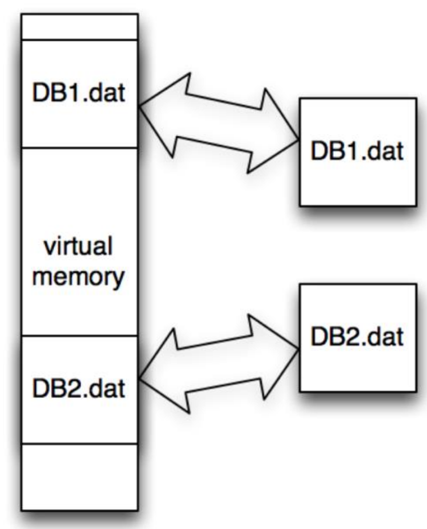
RDBMS	MongoDB
Database	Database
Table, View	Collection
Row	Document (JSON,BSON)
Column	Field
Index	Index
Join	Embedded Document

Document:

```
{
      "_id" : ObjectId("5114e0bd42…"),
      "first" : "John",
      "last" : "Doe",
      "age" : 39,
     "interests" : [
            "Reading",
            "Mountain Biking ]
     "favorites": {
            "color": "Blue",
            "sport": "Soccer"}
}
```

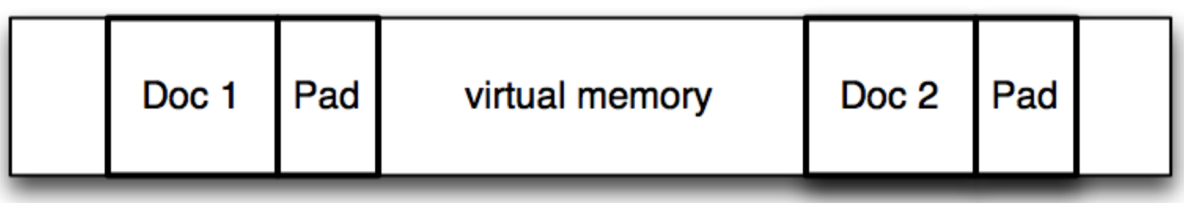
Memory Mapped Files

- Delegate the memory management to the OS with Virtual Memory (VM)
- All files are mapped to the Virtual memory
  - Direct byte-to byte correlation between a file and a piece of VM



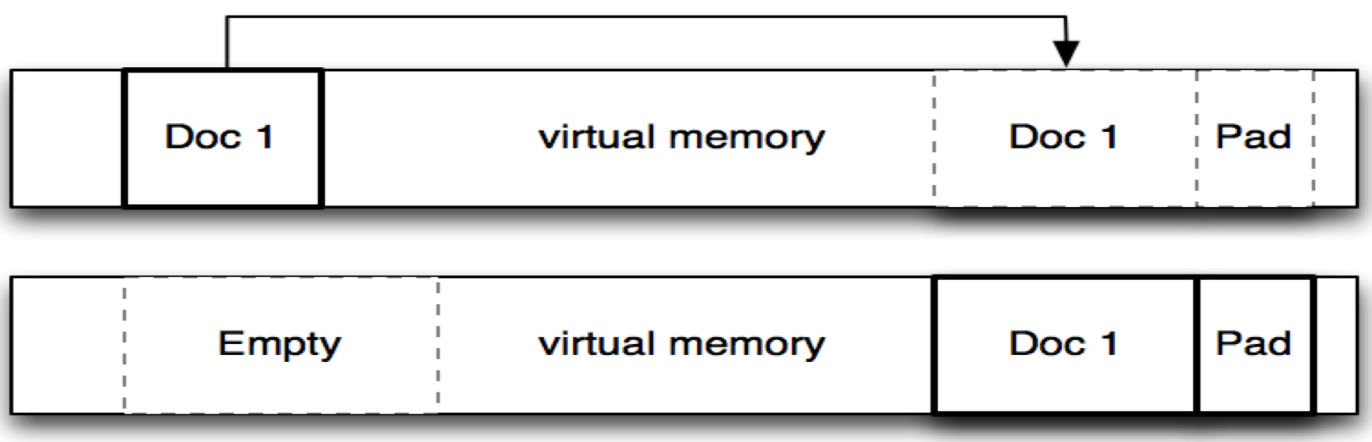
The document-Padding

- Each document is saved in a continuous part of the disk/VM
  - If it grows too large it might be moved
- To minimize document movements, MongoDB uses padding.
  - The padding is using powers of 2 to determine the size of the padded document
  - If moved, the empty space left can be easily filled by another document



Fragmentation

- Moving documents creates holes (Fragmentation)
  - Rearranging the documents to have an efficient use of space is very expensive
  - The “power of 2 sizes” makes it easy to fill one hole with a document



Create Database and Collection

```
>use test

>db.collection_name.insert( <document> )
 //eg db.people.insert({"name":"John", "age":19})
```

- The database instantiates when a table collection is created
- The structure of the documents does not have to be the same
- The field _id is reserved and always set
  - If you don't set it will be set automatically
  - Has to be unique

Update

```
>db.collection_name.update( <query>, <update>, <options> )

db.people.update(
        { name: “John" },
        {rating: 1, score: 1 },
        { upsert: true }
)
```

- Upsert : insert if it does not exist

Query: find

```
>db.collection.find( <query>, <projection> )

db.people.find( {name:"John"},{age:true}  ).limit(1)
```

- Optional
- Keep only the fields you need

- You can limit the number of results you see

Operators (1)

```
>db.people.find( {age:{$gt : 18}})
```

$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$in	Matches any of the values specified in an array.
$nin	Matches none of the values specified in an array.

Operators (2)

$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$not	Inverts the effect of a query expression and returns documents that do not match the query expression.
$nor	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.

```
>db.people.find({ $or: [ { age { $eq: 20 } }, { $eq: 21 } ] })
```

Operators (3)

`$exists`	Matches documents that have the specified field.
`$mod`	Performs a modulo operation on the value of a field and selects documents with a specified result.
`$regex`	Selects documents where values match a specified regular expression.
`$text`	Performs text search.
`$where`	Matches documents that satisfy a JavaScript expression.

```
>db.people.find({rating: { $exists: true }, $where : "this.rating > 4"})
```

Can be any kind of javascript code

Delete records

```
>db.collection_name.remove( <query>, <justOne> )
```

- Optional
- {justOne: true}
- Delete only the first matching record

Indexing (Syntax)

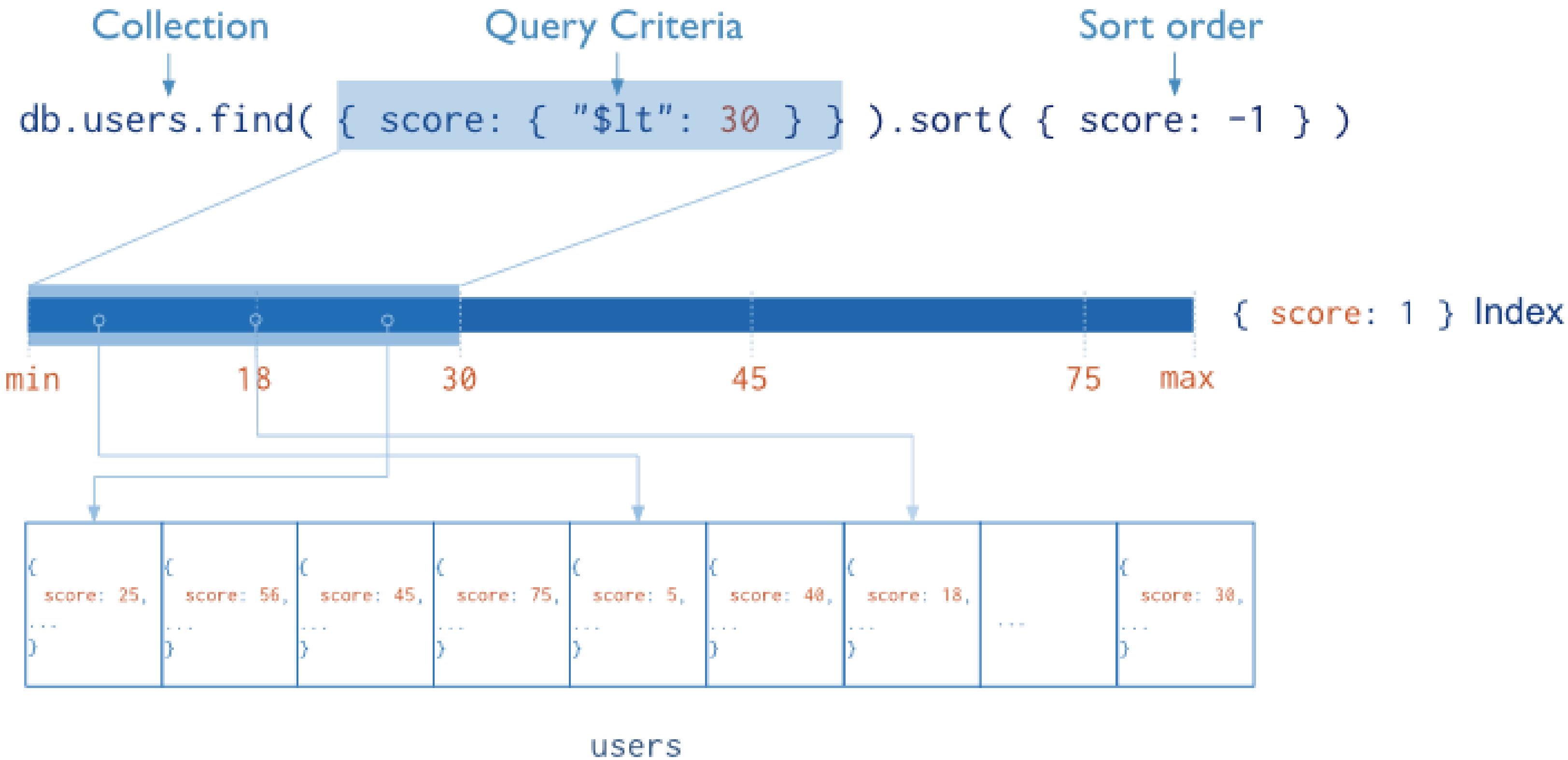
```
>db.people.createIndex( { name: 1 } )
>db.people.createIndex( { name: 1, age: -1 } )
>db.people.createIndex( { “favorites.color ”: 1 } )
```

Index embedded documents

Ascending

Descending

Indexing (Single field)



- Mongo DB uses mainly B-tree index

Image source : http://docs.mongodb.org/

Compound Index (1)

- The order of the keys and their ordering plays a role on whether a compound index can be used

```
db.people.createIndex( { name: 1, age: -1 } )
```

- For sorting the query results:
  - The sorting keys must be specified in the same order as the index
    - ☑ sort({name:1,age:1})
    - ☒ sort({age:1, name:1})

Compound Index (2)

- Index Prefixes: the *beginning* subsets of indexed fields
- The index can support only prefixes for queries (and sorting)
- For index on {name:1,age:1,gender:1}:
  - ☑ name
  - ☑ name, age
  - ☑ name, age, gender
  - ☒ age <or>gender
  - ☒ age, gender

Other Types of indexes

- Geospatial: 2D indexes that use planar or spherical geometry to return values
- Text Indexes: support "special" string search over text
  - Offers stopwords, automatic stemming
- Hash indexes: only for equality matches
- Sparse indexes: only has entries for documents that have the field
- TTL : indexes that specify an expiration time/date
  - The document is removed afterwards

Sharding

```
>sh.enableSharding(“test")

>sh.shardCollection(“test.people", { “_id": 1 } )
```

First enable sharding on the database

- Then Shard the collection
- Pick a field to shard on (can be compound)
- The field must have a “good” range of values
  - The value of the key determines the machine to where it is stored
  - The field becomes indexed

Aggregation (1)

```
db.people.aggregate(
     [{ $group: { _id: “name", avgAge: { $avg: “age" } } },
     { $match: { avgAge: { $gte: 18} } },
      ] )
```

- **SELECT name**, **AVG**(age) **AS** avgAge
  FROM people
  GROUP BY name
  HAVING avgAge >=18

Aggregation(2)

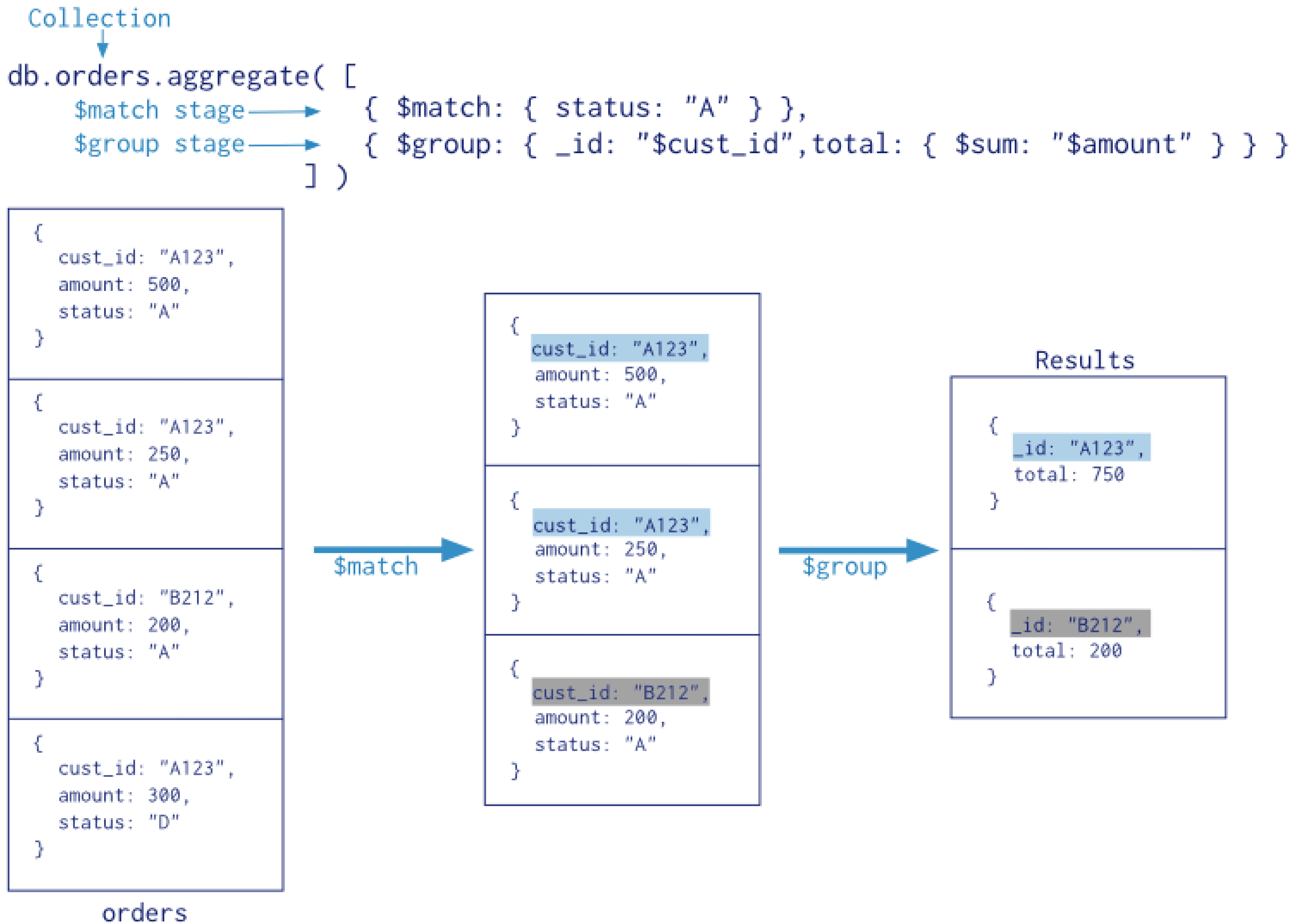


Image source : http://docs.mongodb.org/

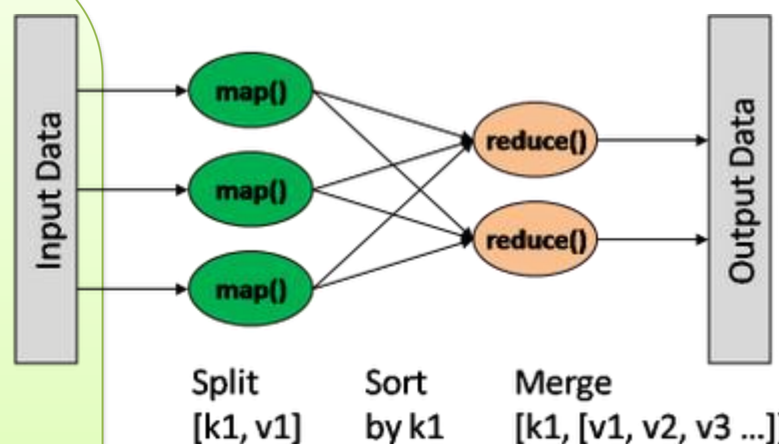
Aggregation(3) –Operator order

Name	Description
$project	Manage the fields you want to use
$match	Apply a query to filter the data
$limit	Use only the first n documents
$skip	Skip n documents
$unwind	Applied in a array which flattens it .
$group	Groups input documents by a specified identifier expression and applies accumulator expression(s),.
$sort	Reorders the document stream by a specified sort key.
$out	Writes the resulting documents of the aggregation pipeline to a collection.

Map/Reduce

```
var mapFunction1 = function() { emit(this.name, this.age); };
var reduceFunction1 = function(keyCustId, values)
{ return avg(values); };
```

```
db.people.mapReduce(
            mapFunction1,
            reduceFunction1,
           {      out: <collection>,
                  sort: <>,
                  limit: <number>,
                  .....
                  })
```



Graph databases

Outline

- Graph relation in databases
- Cypher for neo4j
- Ranking in graphs - Pagerank

Graph relations in relational data bases

Modeling "recommends" relation in a relational database

Person	
ID	**Person**
1	Alice
2	Bob
…	…
99	Zach

Recommends	
ID	**Rec_id**
1	2
2	1
2	99
…	…
99	1

"who is recommended by Bob"

```
SELECT p1.Person
FROM Person p1 JOIN Recommends
    ON Recommends.Rec_id = p1.ID JOIN
        Person p2
    ON Recommends. ID = p2.ID WHERE
            p2.Person = 'Bob'
```

Limited number of rows under consideration using the filter
WHERE p2.Person = 'Bob'

Graph relations in relational data bases

Modeling “recommends” relation in a relational database

Person	
ID	**Person**
1	Alice
2	Bob
…	…
99	Zach

Recommends	
ID	**Rec_id**
1	2
2	1
2	99
…	…
99	1

“who recommends Bob”

```
SELECT p1.Person
FROM Person p1 JOIN Recommends
        ON Recommends.id = p1.ID
        JOIN Person p2
        ON Recommends. Rec_ID = p2.ID
WHERE    p2.Person = 'Bob'
```

- The answer to this query is Alice; sadly, Zach doesn’t recommend Bob.
- query is still easy to implement,
- on the database side it’s more expensive: need to scan the entire Recommends table.

Graph relations in relational data bases

Modeling "recommends" relation in a relational database

Person	
ID	**Person**
1	Alice
2	Bob
…	…
99	Zach

Recommends	
ID	**Rec_id**
1	2
2	1
2	99
…	…
99	1

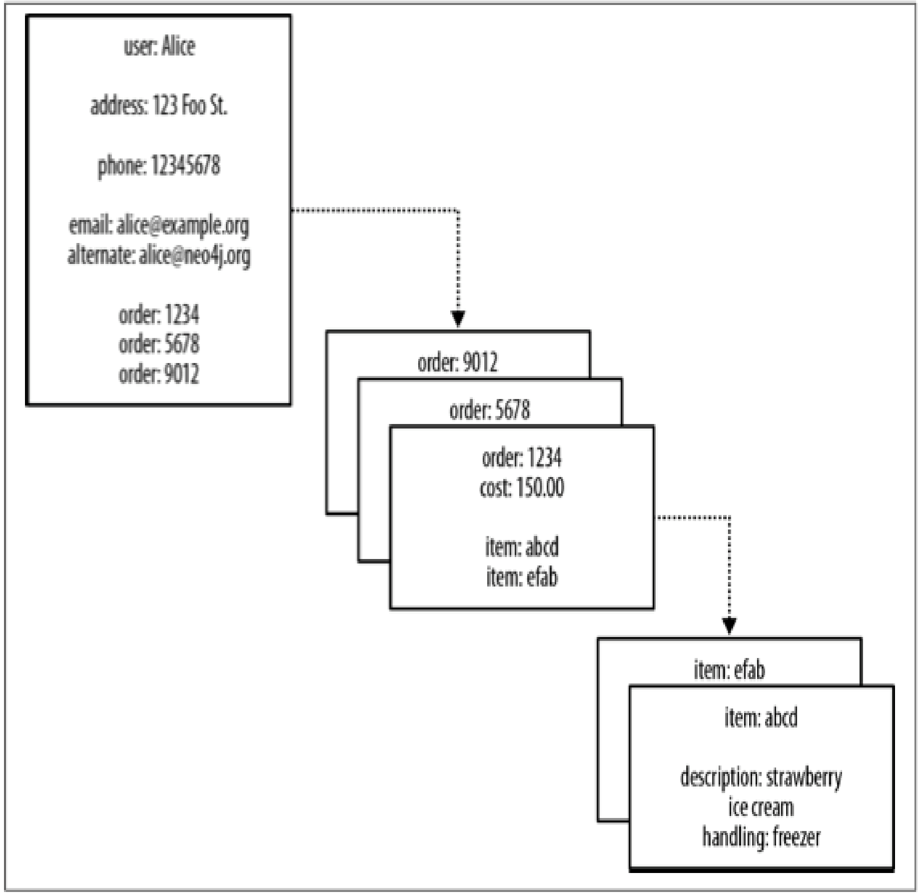
"who is recommended by Alice's recommendations?"

SELECT p1.Person **AS** PERSON, p2.Person **AS** recom_of_recom **FROM** Recommends pf1
JOIN Person p1
ON pf1. ID = p1.ID **JOIN** Recommends pf2
ON pf2.ID = pf1.Rec_id **JOIN** Person p2
ON pf2. Rec_id = p2.ID
WHERE p1.Person = 'Alice' **AND** pf2. Rec_id <> p1.ID

- query is computationally complex, even though it only deals with the recommendations of Alice's recommended people, and goes no deeper .
- Things get more complex and more expensive the deeper we go into the network.
- queries that extend to four, five, or six degrees of friendship deteriorate due to the computational and space complexity of recursively joining tables.

NOSQL Databases Lack Relationships

- Most NOSQL databases—key-value, document, or column-oriented—store sets of *disconnected* documents/values/columns. - difficult exploit for connected data and graphs.
- add relationships to such stores: embed an aggregate's identifier pointing to another aggregate—effectively *foreign keys*.
- requires joining aggregates at the application level: may be very expensive.
- Example: in *user: Alice* a reference to *order: 1234*, we infer a connection between user: Alice and order: 1234.

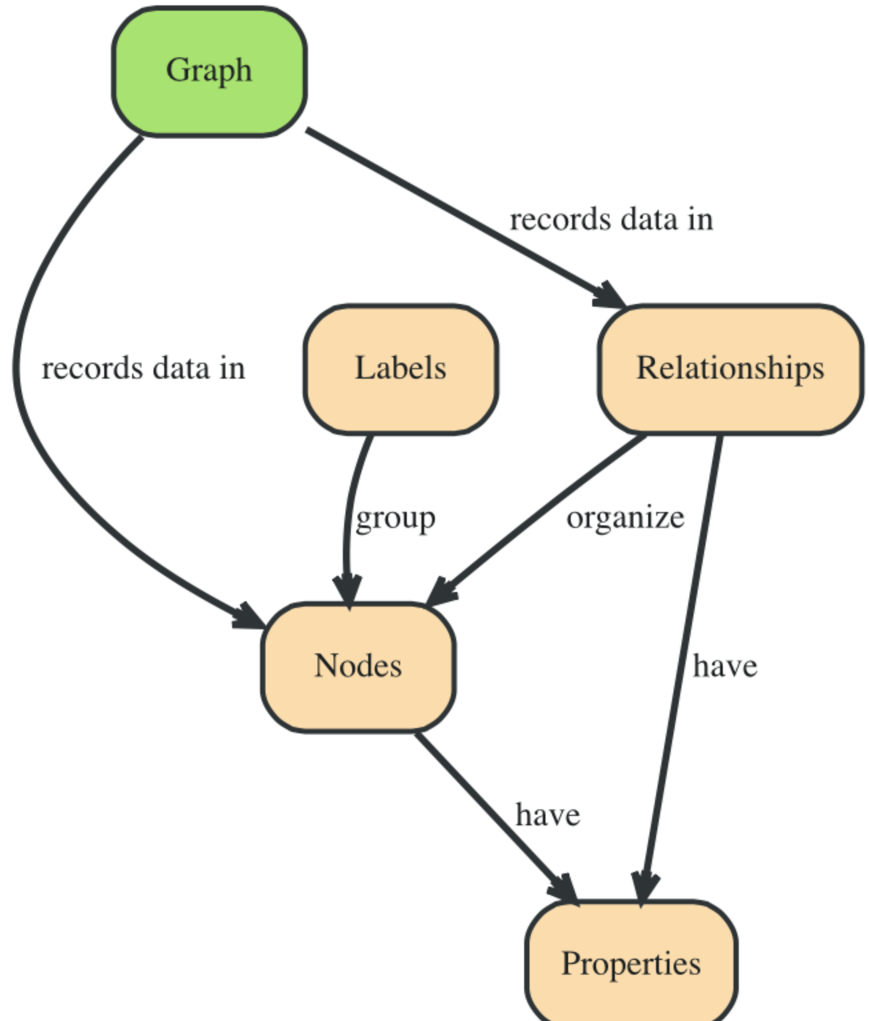


NoSQL Poor performance on connected data

- Path adjacency is problematic in relational/nosql
  - path computations in graph is expensive, i.e. *recommended-of-recommended* example.
  - aggregate stores or relational databases fall short when managing connected data.
  - Only shallow traversals (i.e. immediate “friends”, or possibly “friends- of-friends”) are feasible due to exponential # index lookups .
- Graphs, use index-free adjacency to ensure that traversing connected data is extremely rapid.
  - every element in the database contains a direct link to its adjacent element.
  - No index lookups are required; every element (or node) knows what node or nodes it is connected with <u>(edge)</u>.

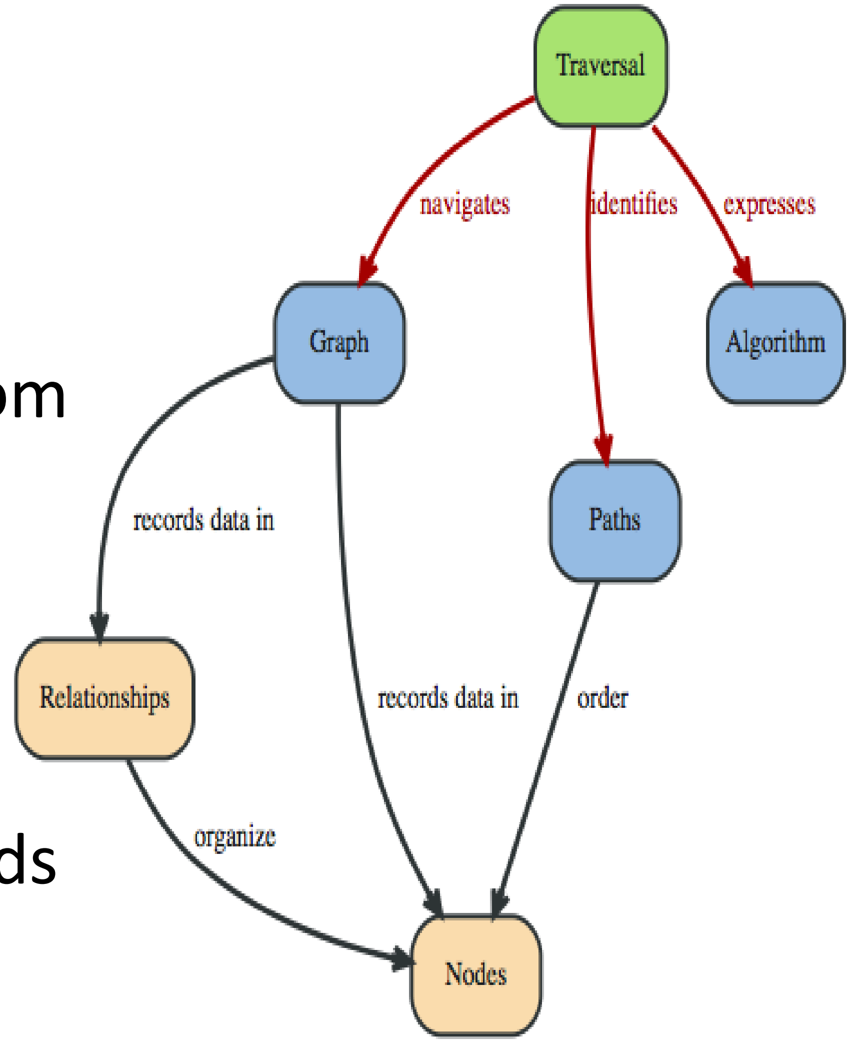
Graph database concepts

- A Graph represents data in *Nodes* which have *Properties*
- Nodes are organized by *Relationships* which also have *Properties*
- Nodes are grouped by→ *Labels* into→ *Sets*



Query a Graph with a Traversal

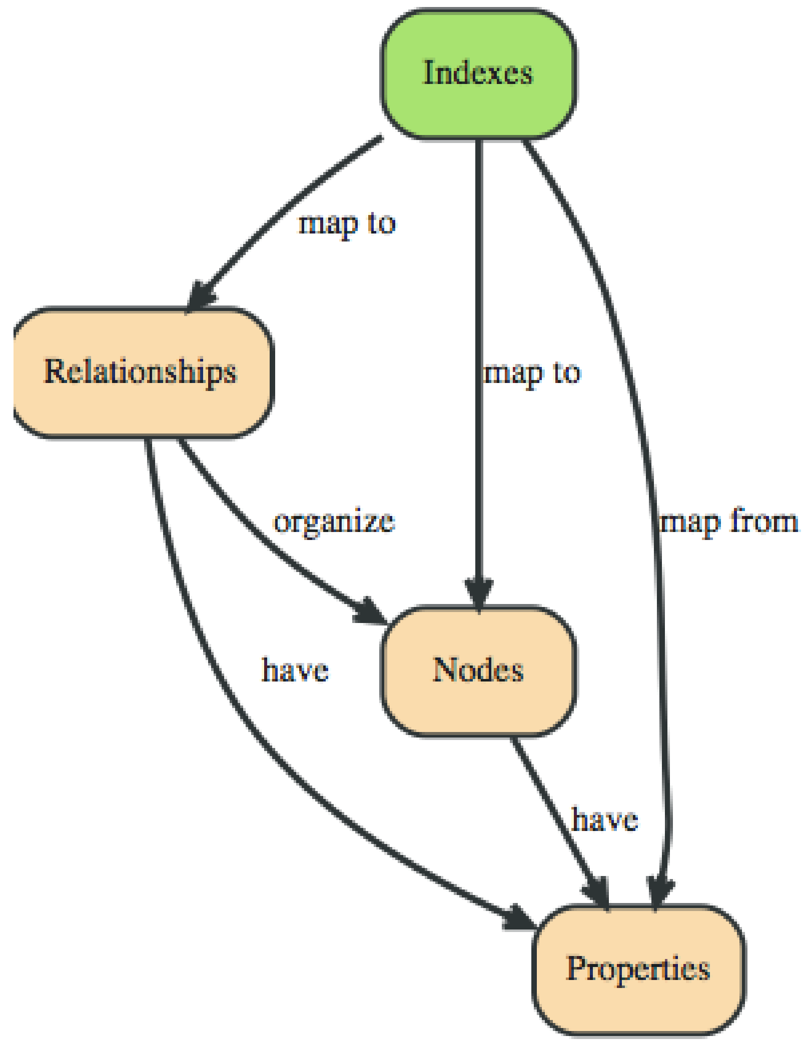
- A Traversal navigates a Graph; it identifies→ Paths which order Nodes
  - query a Graph, navigating from starting Nodes to related Nodes according to an algorithm,
  - finding answers to questions like “what music do my friends like that I don’t yet own”



Indexes look-up Nodes or Relationships

"An Index maps from Properties to either Nodes or Relationships"

- find a specific Node or Relationship according to a Property it has.
- Rather than traversing the entire graph, use an Index to perform a look-up, for questions like "find the Account for username master-of-graphs."



Cypher

- expressive and efficient querying and updating of the graph store.
- relatively simple but still powerful. Complicated database queries can easily be expressed through Cypher.
- declarative graph query language
  - focuses on *what* to retrieve from a graph, not on *how* to retrieve it
- Cypher is inspired by a established practices for expressive querying
  - Most of the keywords like WHERE and ORDER BY are inspired by SQL.
  - Pattern matching borrows expression approaches from SPARQL.
  - Some of the collection semantics have been borrowed from languages such as Haskell and Python.

Cypher structure

- Cypher structure similar to SQL
- Clauses are chained together, feed intermediate result sets between each other.
  - For example, matching identifiers from a MATCH clause will be the context for the next clause

Clauses used to read from the graph:

START

- specifies one or more starting *points*—nodes or relationships—in the graph.
- *points* are obtained via index lookups or accessed directly based on node or relationship IDs.

MATCH: The graph pattern to match - the most common way to get data from the graph.

- *specification by example* part.
- represent nodes and relationships, *draw* the data we're interested in
- parentheses to draw nodes,
- relationships (**-->** and **<--**). < and > indicate relationship direction.
- Relationships: Between the dashes, set off by square brackets and prefixed by a colon -**[:<relation>]->**.

WHERE: part of MATCH, OPTIONAL MATCH and WITH. –

- adds constraints to a pattern,
- filters the intermediate result passing through WITH.

RETURN: What to return.

Cypher: Node Syntax

- (node) to represent a node, eg: (), (foo).

- () (matrix) (matrix:Movie) (matrix:Movie {title: "The Matrix"}) (matrix:Movie {title: "The Matrix", released: 1997})
- (): represents an anonymous, uncharacterized node.
- (matrix): Identifier “matrix”, restricted (ie, scoped) to a single statement
- (:Movie) label declares node’s type.
- (matrix:Movie {title: "The Matrix", released: 1997})
  - node’s properties (e.g. title) represented as a list of key/value pairs, enclosed within a pair of braces
  - Properties can be used to store information and/or restrict patterns..

Cypher: **Relationship Syntax**

- (--) to represent an undirected relationship.
- Directed relationships (eg, <--, -->).
- Bracketed expressions (eg: [...]) used to add details. This may include identifiers, properties, and/or type information, eg:
- **-[role]->**
  - role: variable
- **-[:ACTED_IN]->**
  - :ACTED_IN: type of relationship
- **-[role:ACTED_IN]->**
- **-[role:ACTED_IN {roles: ["Neo"]}]->**
- The syntax and semantics similar to node.
- value of a property may be an array

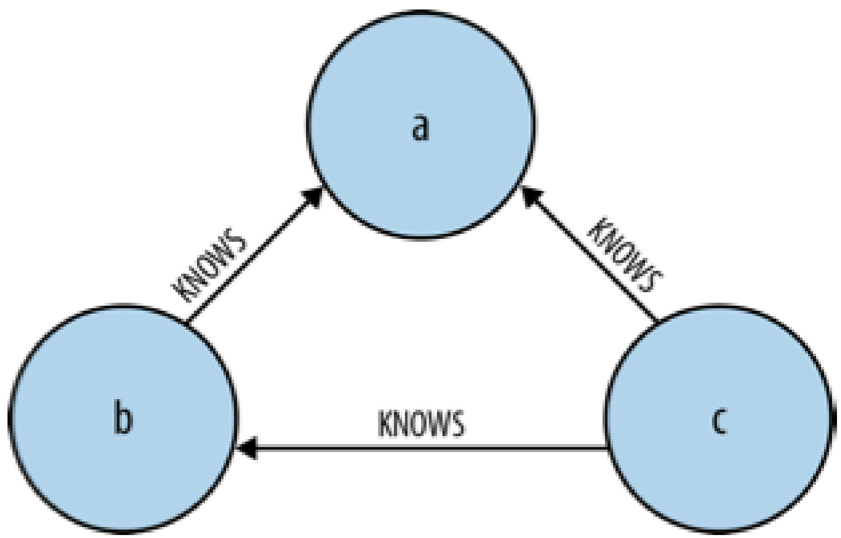
Cypher: pattern syntax

- Combining the syntax for nodes and relationships -> patterns
- (keanu:Person:Actor {name: "Keanu Reeves"})
  -[role:ACTED_IN {roles: ["Neo"] }]->
  (matrix:Movie {title: "The Matrix"})
- relationship type ACTED_IN added as a symbol, prefixed with a colon:
- Identifiers (eg, role) can be used elsewhere in the statement to refer to the relationship.
- Node and relationship properties use the same notation.
- an array property for the roles, allowing multiple roles to be specified.
- acted_in = (:Person)-[:ACTED_IN]->(:Movie)
- Pattern variable acted_in would contain two nodes and the connecting relationship for each path that was found or created.
- functions to access details of a path, including
  - nodes(path), rels(path), length(path).

Data Modeling with graphs

- CYPHER (used in neo4j)

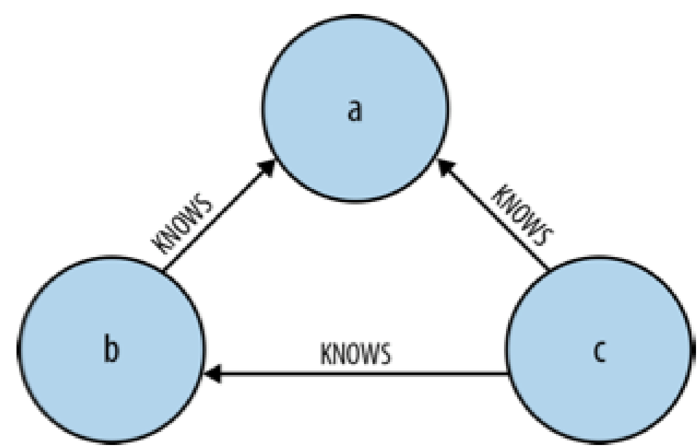
(c)-[:KNOWS]->(b)-[:KNOWS]->(a), (c)-[:KNOWS]->(a)



Cypher

an example: find the mutual friends of user named *Michael*:

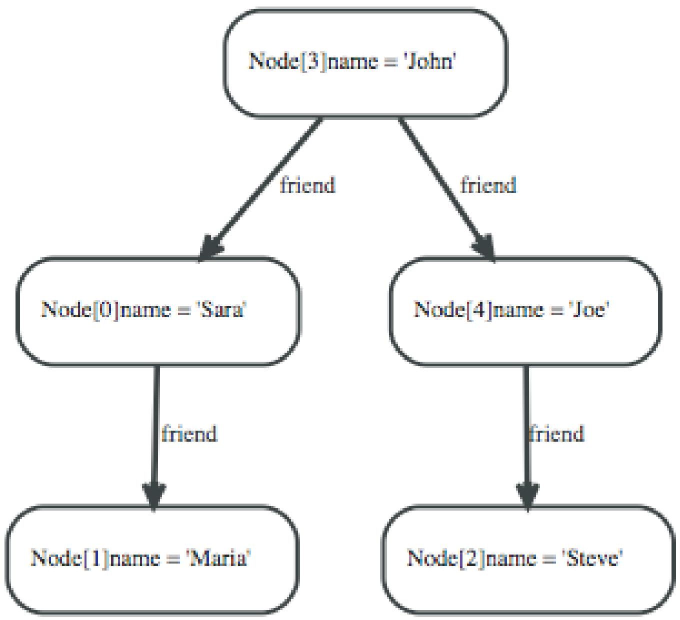
START a=**node**:user(name='Michael')
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
RETURN b, c



Example

Find John’s non immediate friends

MATCH (john {name: 'John'})-[:friend]->()-[:friend]->(fof)
RETURN john, fof



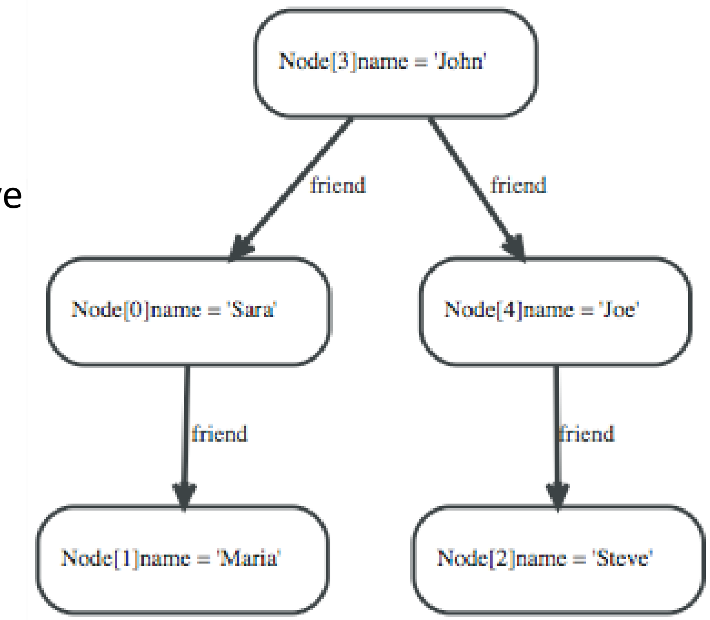
john	fof
Node[3]{name:"John"}	Node[1]{name:"Maria"}
Node[3]{name:"John"}	Node[2]{name:"Steve"}

2 rows

Example

Find users whose friends names start with "S"

MATCH (user)-[:friend]->(follower)
WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve
follower.name =~ 'S.*'
RETURN user, follower.name



user	follower.name
Node[3]{name:"John"}	"Sara"
Node[4]{name:"Joe"}	"Steve"

2 rows

Creating and updating

- CREATE (DELETE): Create (delete) nodes and relationships.
- SET (REMOVE): Set values to properties and add labels on nodes using SET and use REMOVE to remove them.
- MERGE: Match existing or create new nodes and patterns. This is especially useful together with uniqueness constraints.

An example – movie database

CREATE ({ name:"Tom Hanks" });

- To see the node created:

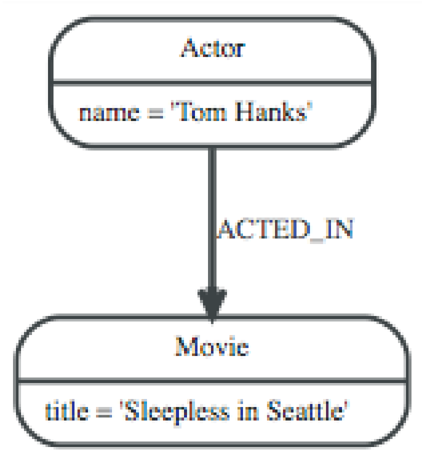
MATCH (actor:Actor { name: "Tom Hanks" }) RETURN actor;

- create a movie and connect it to the Tom Hanks node with an ACTED_IN relationship:

MATCH (actor:Actor)

WHERE actor.name = "Tom Hanks" CREATE (movie:Movie { title:'Sleepless in Seattle' })

CREATE (actor)-[:ACTED_IN]->(movie);



An example – movie database

```
MATCH (actor:Actor { name: "Tom Hanks" })
CREATE UNIQUE (actor)-[r:ACTED_IN]->(movie:Movie { title:"Forrest Gump" })
RETURN r;
```

- CREATE UNIQUE make sure create unique patterns.
- [r:ACTED_IN] lets us return the relationship.

- Set a property on a node:

```
MATCH (actor:Actor { name: "Tom Hanks" })
SET actor.DoB = 1944
RETURN actor.name, actor.DoB;
```

- list all *Movie* nodes:

```
MATCH (movie:Movie)
RETURN movie AS `All Movies`;
```

All Movies
Node[1]{title:"Sleepless in Seattle"}
Node[2]{title:"Forrest Gump"}

2 rows

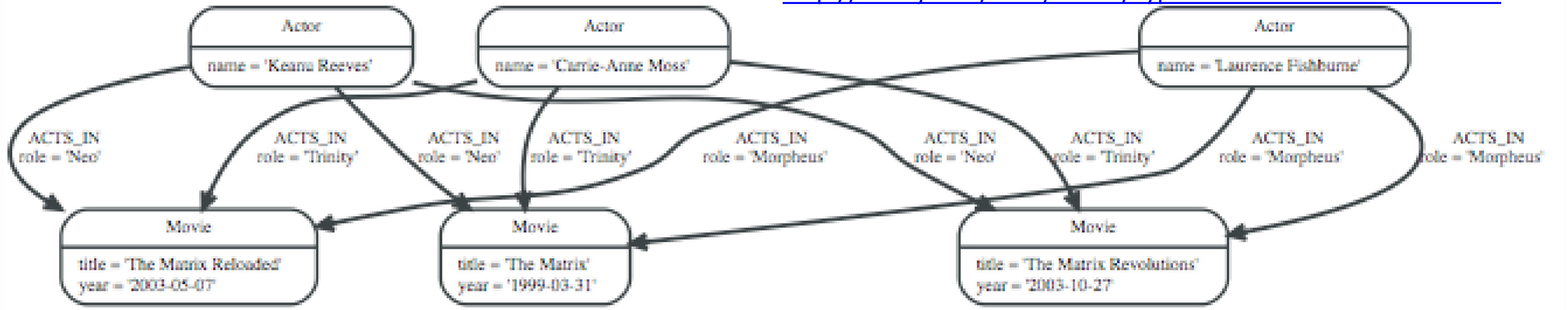
Populate the movie database

```
CREATE (matrix1:Movie { title : 'The Matrix', year : '1999-03-31' })
CREATE (matrix2:Movie { title : 'The Matrix Reloaded', year : '2003-05-07' })
CREATE (matrix3:Movie { title : 'The Matrix Revolutions', year : '2003-10-27' })

CREATE (keanu:Actor { name:'Keanu Reeves' })
CREATE (laurence:Actor { name:'Laurence Fishburne' })
CREATE (carrieanne:Actor { name:'Carrie-Anne Moss' })

CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix1)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix2)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix3)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix1)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix2)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix3)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix1)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix2)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix3)
```

http://neo4j.com/docs/stable/cypherdoc-movie-database.html



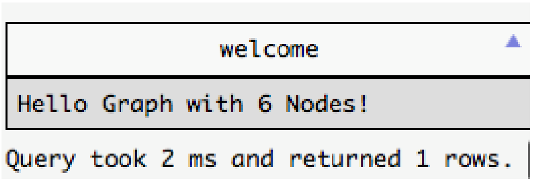
Some queries

- How many nodes do we have:

MATCH (n)

RETURN "Hello Graph with " + count(*)+

“Nodes!" AS welcome;



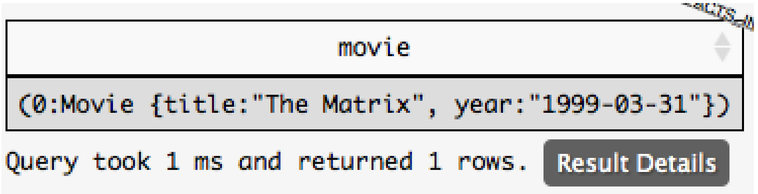
welcome
Hello Graph with 6 Nodes!

Query took 2 ms and returned 1 rows.

- Return a single node, by name:

MATCH (movie:Movie { title: 'The Matrix' })

RETURN movie;



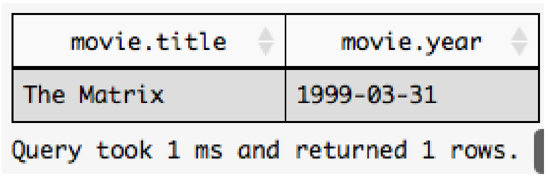
movie
(0:Movie {title:"The Matrix", year:"1999-03-31"})

Query took 1 ms and returned 1 rows. Result Details

- Return the title and date of the matrix node:

MATCH (movie:Movie { title: 'The Matrix' })

RETURN movie.title, movie.year;



movie.title	movie.year
The Matrix	1999-03-31

Query took 1 ms and returned 1 rows.

- List all nodes and their relationships:

  MATCH (n)-[r]->(m)
  RETURN n AS FROM , r AS `->`, m AS to;

Path queries

- Find the movies in which acted the actors of the movie “Matrix”

MATCH (:Movie { title: "The Matrix" })<-[:ACTS_IN]-(actor)-[:ACTS_IN]- >(movie)

RETURN movie.title, count(*)

ORDER BY count(*) DESC ;

movie.title	count(*)
The Matrix Reloaded	7
The Matrix Revolutions	5
New movie	2

Query took 3 ms and returned 3 rows. Result Details

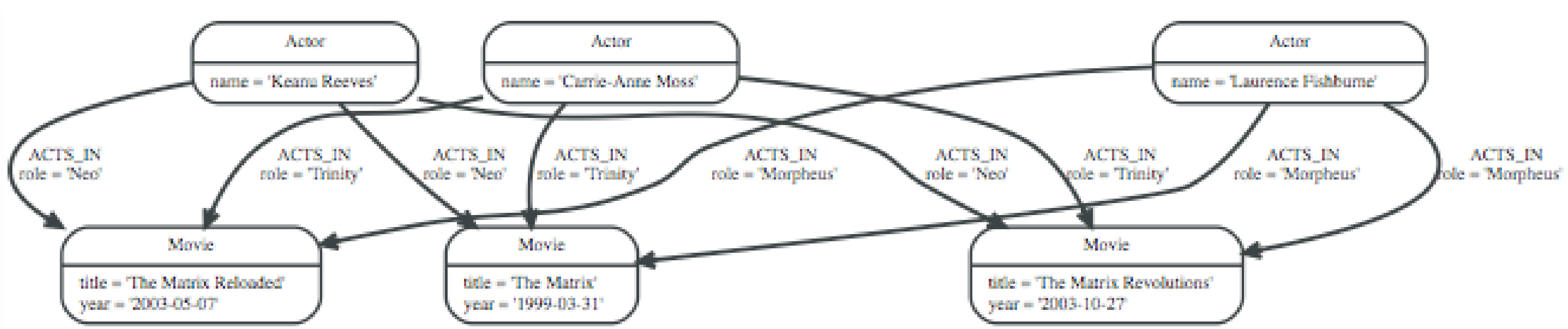
- Who acted in those movies?

MATCH (:Movie { title: "The Matrix" })<-[:ACTS_IN]-(actor)

-[:ACTS_IN]-> (movie)

RETURN movie.title, collect(actor.name), count(*) AS count

ORDER BY count DESC ;



Path queries

- co-acting: find actors that acted together with those of “The Matrix”

  MATCH (:Movie { title: "The Matrix" })<-[:ACTS_IN]-(actor)-[:ACTS_IN]->(movie)<-[:ACTS_IN]-(colleague)

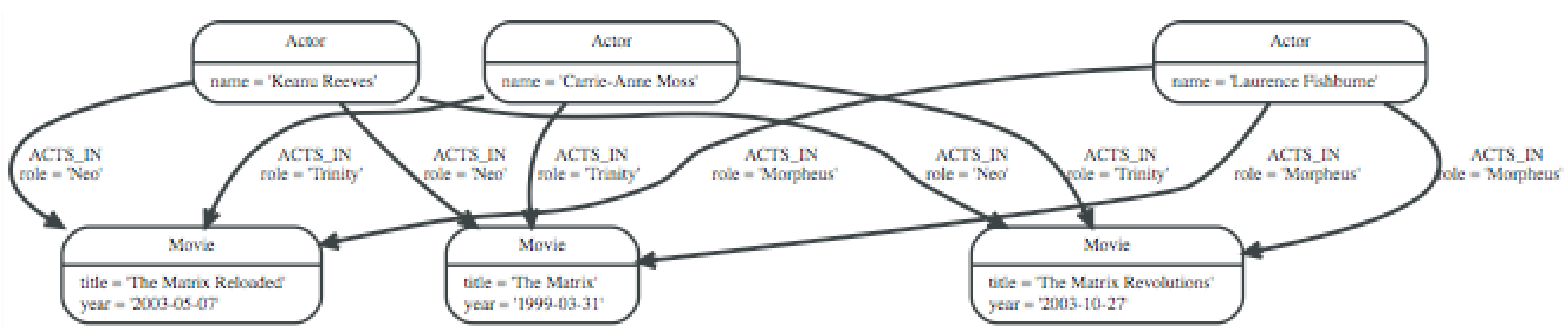
  RETURN actor.name, collect(DISTINCT colleague.name);

- How many paths exist in the graph among the actors “Keanu Reeves” and "Carrie-Anne Moss”?

  MATCH p =(:Actor { name: "Keanu Reeves" })-[:ACTS_IN*0..5]-(:Actor { name: "Carrie-Anne Moss" })

  RETURN extract(n IN nodes(p)| coalesce(n.title, n.name)) AS `names AND titles`, length(p)

  ORDER BY length(p) LIMIT 10;



Is a Graph DB Useful?

- Fraud detection
- Ontologies
- Monitoring Complex systems
- https://neo4j.com/blog/analyzing-panama-papers-neo4j/
- https://neo4j.com/blog/how-boston-scientific-improves-manufacturing-quality-using-graph-analytics/
- https://neo4j.com/blog/anti-money-laundering-infographic/

References

- "Graph databases", Ian Robinson Jim Weber and Emily Eifren, O'Reilly
- http://neo4j.com/
- Amy Nicole Langville, Carl Dean Meyer: Survey: Deeper Inside PageRank. Internet Mathematics 1(3): 335-380 (2003)
- "PageRank Computation and the Structure of the Web: Experiments and Algorithms", Arvind Arasu, Jasmine Novak, Andrew Tomkins & John Tomlin
- http://backtobazics.com/big-data/spark/

HBASE

HBase model primitives

- **Table:** HBase organizes data into tables. Table names are Strings are safe for use in a file system path (i.e. < > : " / \ | ? *)
- **Row:** Within a table, data is stored according to its row and identified uniquely by their *row key*. Row keys are type-less , treated as byte arrays .
- **Column Family:**
  - Data grouped by column family within a row .
  - Column families impact the physical arrangement of data stored in HBase. Must be defined at design time, not easily modified.
  - Every row in a table has the same column families, although a row need not store data in all its families .
  - Column families are Strings and composed of characters that are safe for use in a file system path .
- **Column Qualifier or column**
  - Data within a column family is addressed via its column qualifier, or simply, column .
  - Column qualifiers need **not** be specified in advance.
  - Column qualifiers need **not** be consistent between rows.
  - Column qualifiers have nodata type and are always treated as a *byte[ ]* .

HBase model primitives

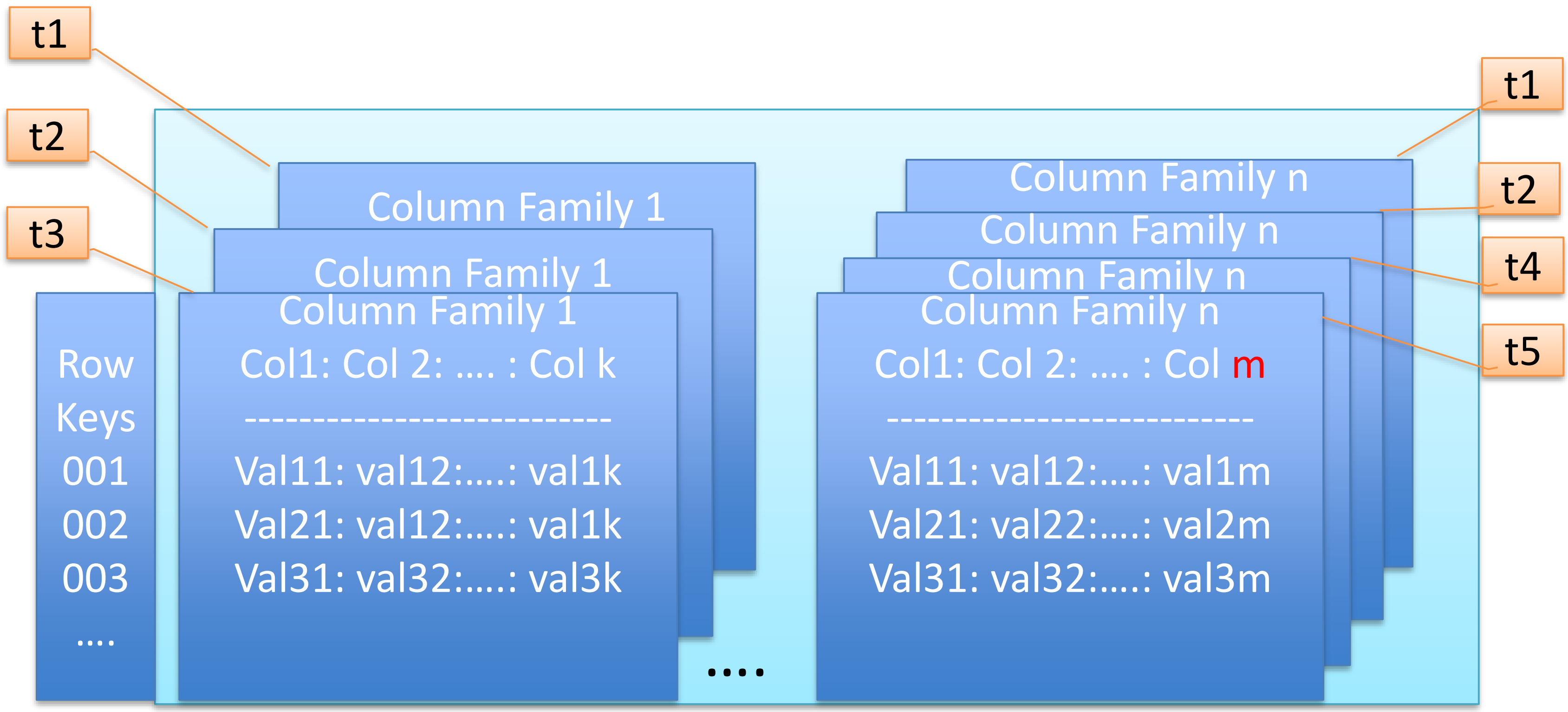
- **Cell:**
  - data identified by <row key, column family, column qualifier> combination uniquely identifies a cell .
  - The data stored in a cell is referred to as that cell's value .
  - Values also do not have a data type and are always treated as a *byte[ ]* .
- **Timestamp:**
  - Values within a cell are versioned.
  - Versions are identified the **timestamp** of when the cell was written
  - If a timestamp is not specified during a write, the current timestamp is used
  - If the timestamp is not specified for a read, the latest one is returned
  - The number of cell value versions retained by HBase is configured for each column family - default number of cell versions is three .

Column Families

Logical View of Customer Contact Information in HBase

Row Key	**Column Family: {Column Qualifier:Version:Value}**
00001	CustomerName: {‘FN’: 1383859182496:‘John’, ‘LN’: 1383859182858:‘Smith’, ‘MN’: 1383859183001:’Timothy’, ‘MN’: 1383859182915:’T’} ContactInfo: {‘EA’: 1383859183030:‘John.Smith@xyz.com’, ’SA’: 1383859183073:’1 Hadoop Lane, NY 11111’}
00002	CustomerName: {‘FN’: 1383859183103:‘Jane’, ‘LN’: 1383859183163:‘Doe’, ContactInfo: { ’SA’: 1383859185577:’7 HBase Ave, CA 22222’}

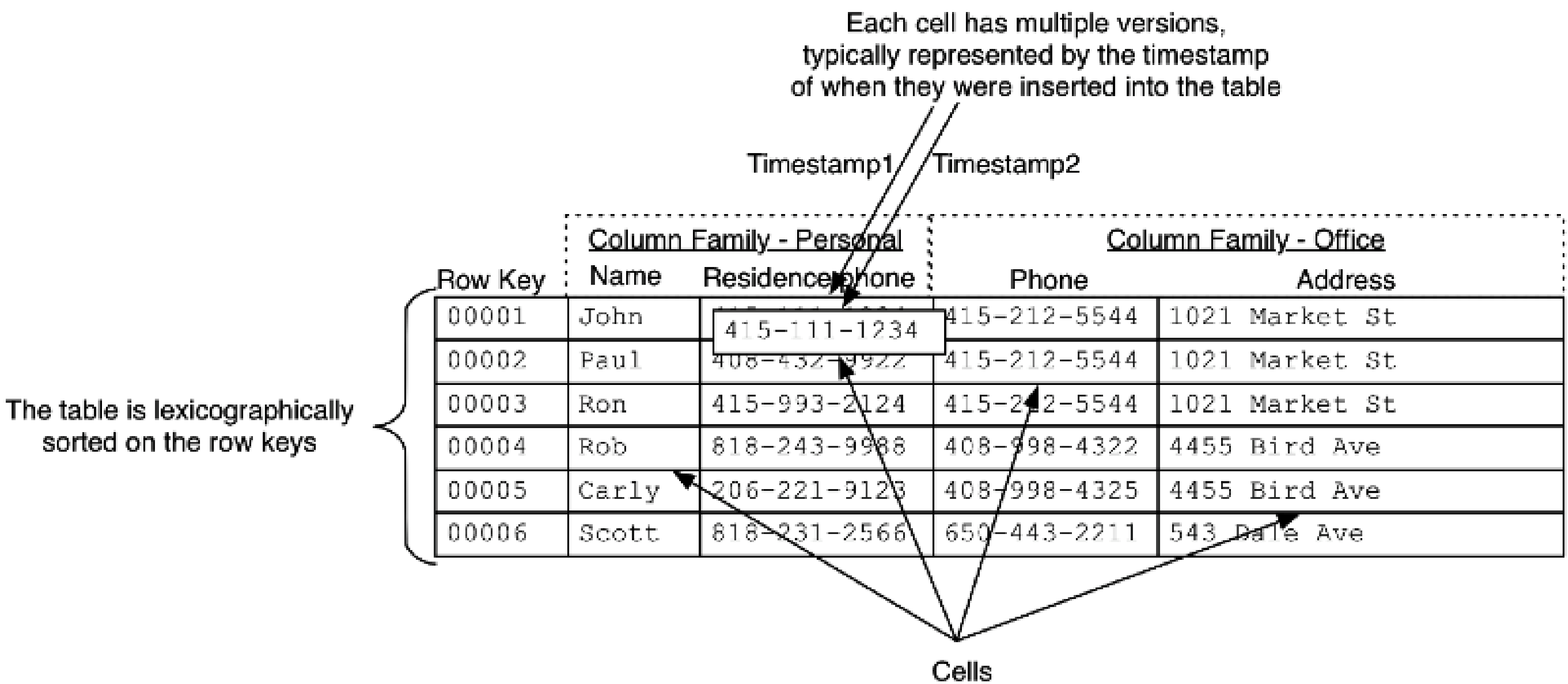
Hbase table – multidimensional map



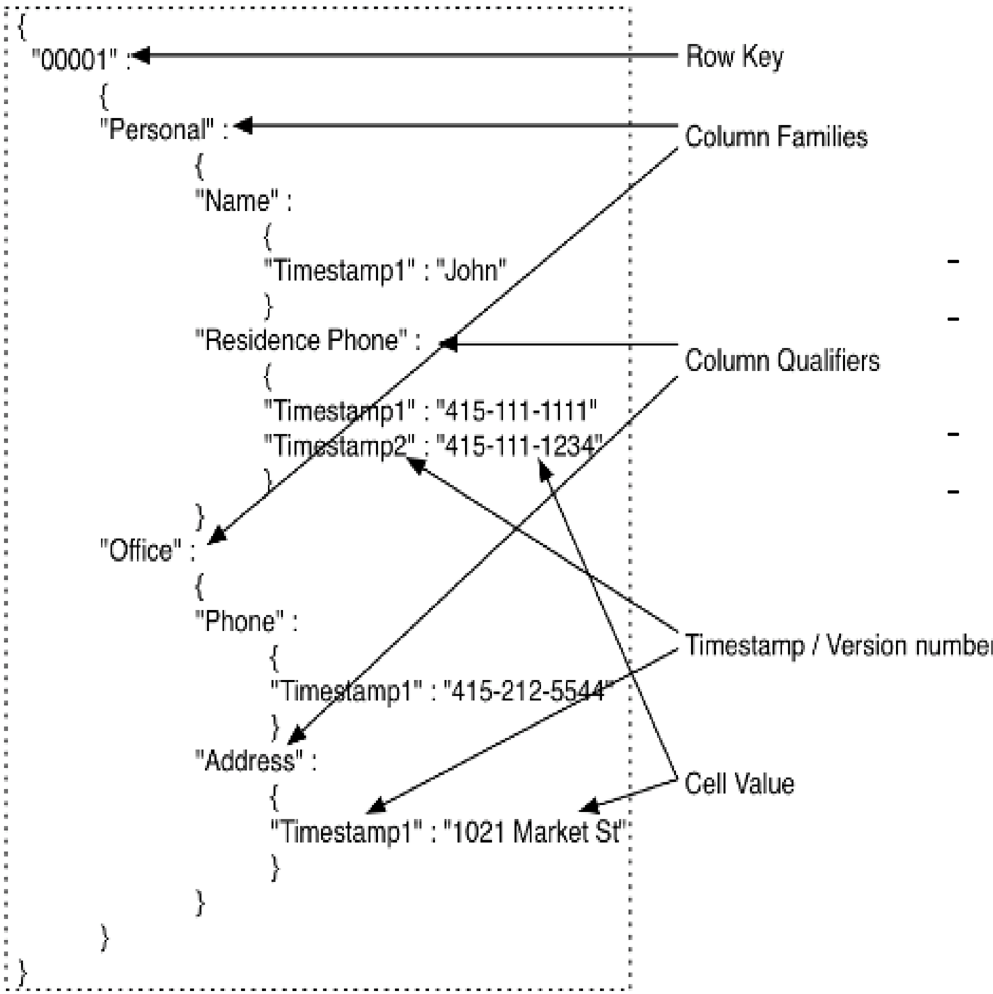
Basic data access

- HBase's API for data manipulation
- *Get*, *Put* are specific to particular rows and need the row key to be provided .
- *Scan* are done over a range of rows . The range could be defined by a start and stop row key or could be the entire table if no start and stop row keys are defined .

Sample HBase table

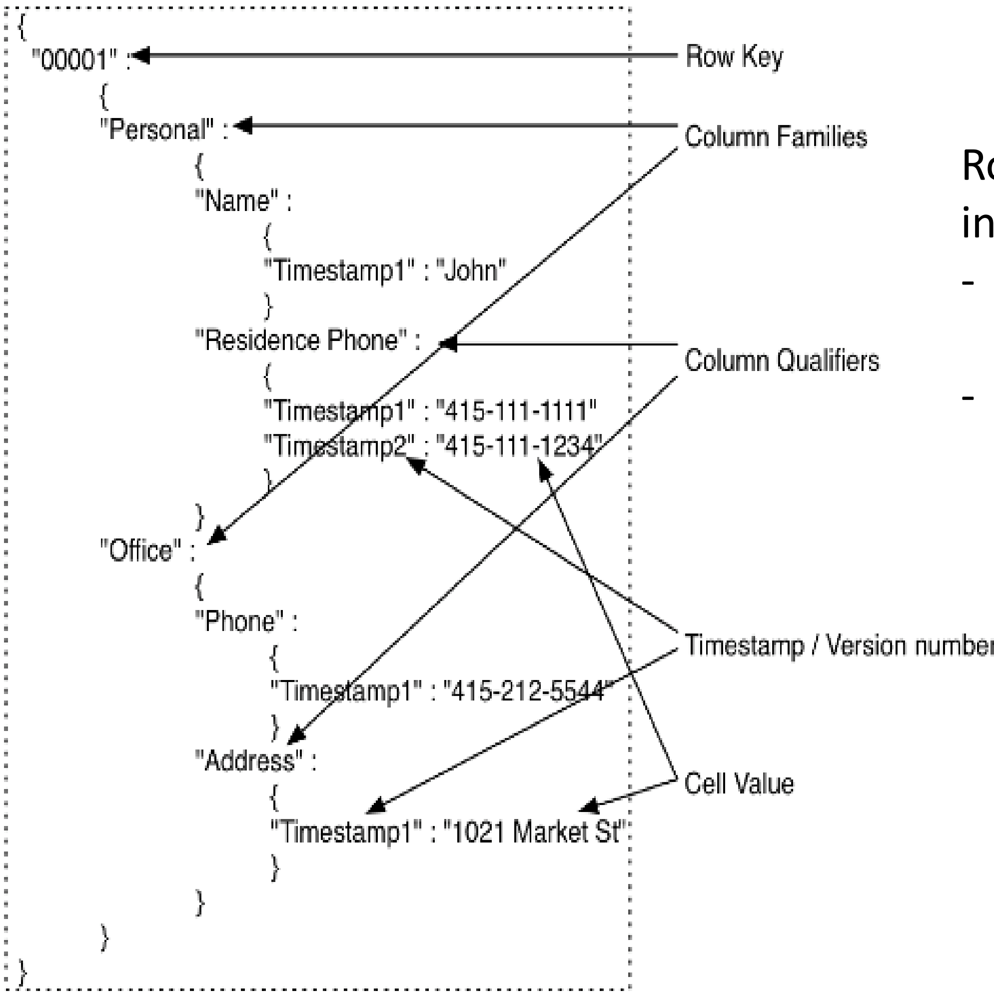


HBase rows as multidimensional maps



- row key => list of column families,
- column family => list of column qualifiers,
- column qualifier => list of timestamps,
- Timestamp => one value (cell).

HBase rows as multidimensional maps



Row keys are the equivalent of primary keys in relational database tables .

- row key is defined at creation time and can't change aftermath
- In other words, the column *Name* in the *Personal* column family cannot be chosen to become the row key after the data has been put into the table .

The notion of "key" in HBase

- GET (rowkey): retrieve data from all the columns

00001 → { Personal : { Name : { Timestamp1 : John }, Residence Phone : { Timestamp1 : 415-111-1234 } },
{ Office : { Phone : { Timestamp1 : 415-212-5544 }, Address : { Timestamp1 : 1021 Market St } } }

- GET (rowkey, column family): retrieve the item that a particular column family maps to, you'd get back all the column qualifiers and the associated maps.

00001 , Personal → { Name : { Timestamp1 : John }, Residence Phone : { Timestamp1 : 415-111-1234 } }

- GET (rowkey, column family, column qualifier):retrieve the timestamps and the associated values of particular column qualifier maps to.

00001, Personal:Residence Phone → { {Timestamp1 : 415-111-1111 } , { Timestamp2 : 415-111-1234))

- GET (rowkey, column family, column qualifier, timestamp): retrieve the associated values of particular column qualifier and timestamp

00001, Personal:Residence Phone , Timestamp2 → { 415-111-1234)

HBase vs Hadoop/HDFS?

HDFS

- distributed file system that is well suited for the storage of large files.
- not a general purpose file system
- does not provide fast individual record lookups in files.

HBase

- built on top of HDFS and provides fast record lookups (and updates) for large tables.
- HBase internally puts your data in indexed "StoreFiles" that exist on HDFS for high-speed lookups..

HBase Design principles

1 . row key structure

2 . # column families

3 . which data goes into what column family?

4 . How many columns are in each column family?

5 . What should the column names be? Column names don’t need to be defined on table creation.

6 . What information should go into the cells?

7 . How many versions should be stored for each cell?

Hbase – rowkey design

- The most important issue in HBase tables design is the row-key structure
- Need to define access patterns (read, write) up front.

Assumptions

1 . Indexing is only done based on the *Key* .

2 . Tables are stored sorted based on the row key. Each contiguous storage *region* in the table contains part of the row key space and is identified by the start and end row key. The *region* contains a sorted list of rows from the start key to the end key .

3 . Everything in HBase tables is stored as a *byte[ ]*. There are no types .

4 . Atomicity is guaranteed only at a row level. There is no atomicity guarantee across rows, which means that there are no multi-row transactions .

5 . Column families have to be defined up front at table creation time .

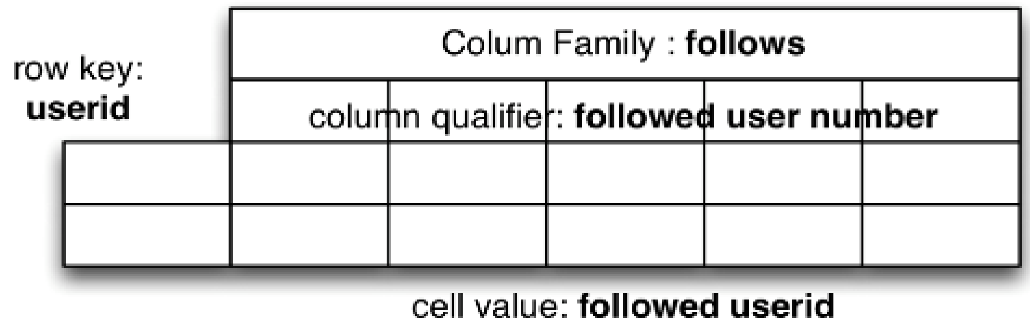
6 . Column qualifiers are dynamic and can be defined at write time. They are stored as *byte[ ]* so you can even put data in them .

HBase table design – Twitter example

- model the Twitter relationships (users following other users) in HBase tables.
- Follower-followed relationships are essentially graphs
- define the access pattern of the application:
  - *Read access pattern:*
    - Who does a user follow?
    - Does a particular user A follow user B?
    - Who follows a particular user A?
  - *Write access pattern:*
    - User follows a new user
    - User un-follows someone they were following .

Hbase table – Twitter example

- Hbase table for users followed by another user



- A table with sample data

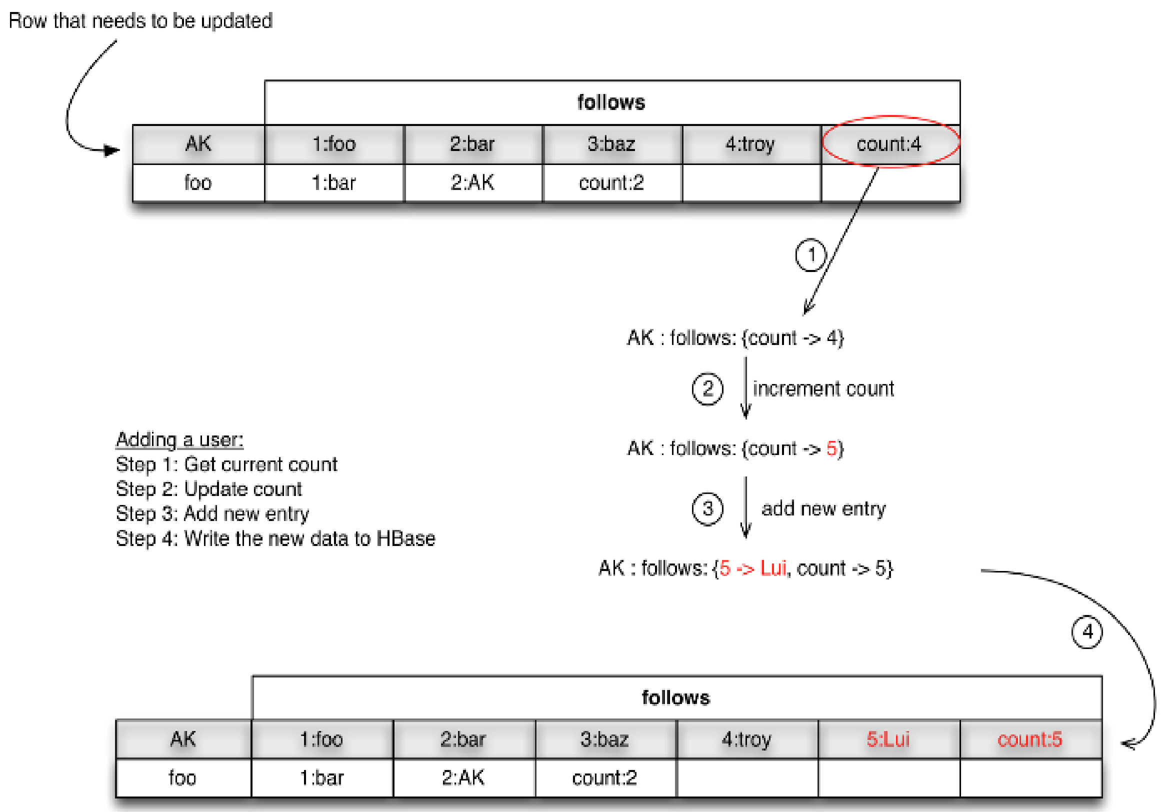
Col Qualifier
Cell value

	follows			
AK	1:foo	2:bar	3:baz	4:troy
foo	1:bar	2:AK		

- Extend the design with a counter to facilitate user addition

	follows				
AK	1:foo	2:bar	3:baz	4:troy	count:4
foo	1:bar	2:AK	count:2		

To add a new follower



- Long and tedious process resembling transaction - HBase does not handle it.
- Handling “un follow” – delete – is also problematic – why?
  - have to read the entire row to find out the column you need to delete .

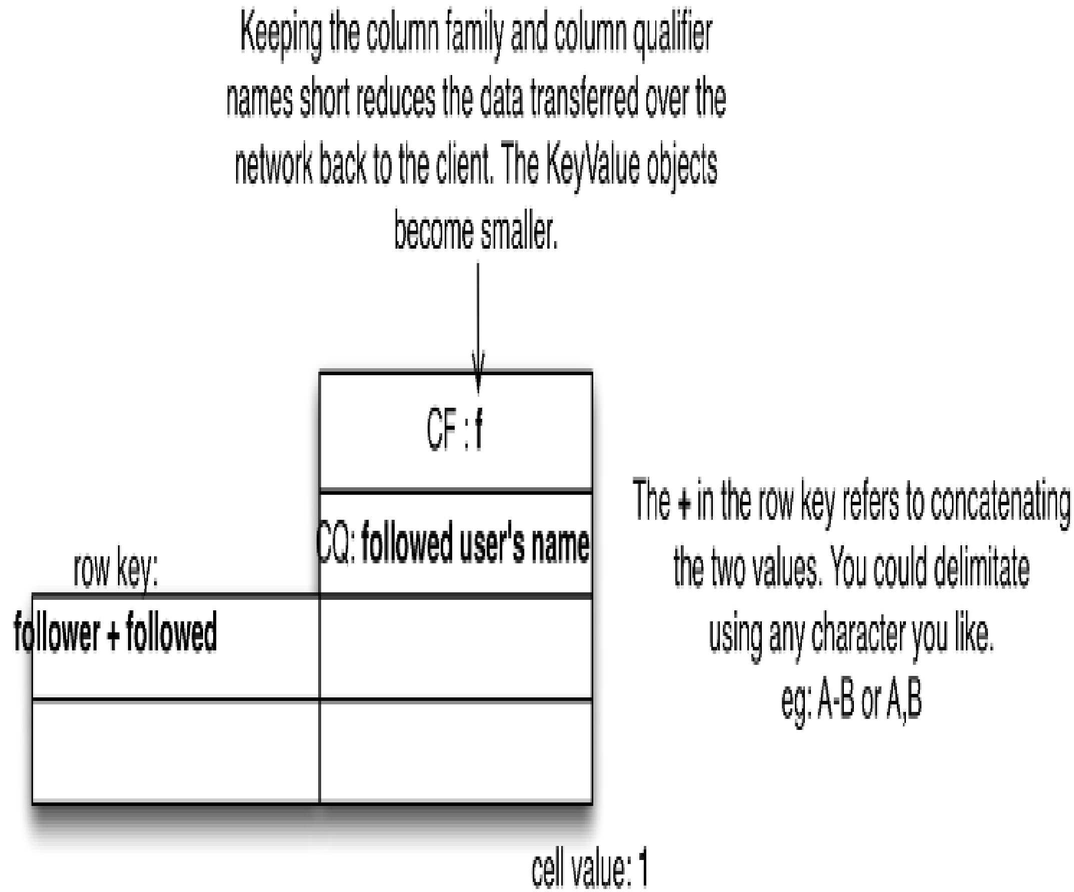
Who follows a particular user A?

In the current design,

- indexing is only done on the row key,
- need to do a full table scan to answer this question
- should figure in the index somehow
- Brute force solution:
  - maintain another table which contains the reverse list (*userid* and a list of all who *follow userid*) .

who follows a particular user A?

- Store information in the same table with different row keys
- HBase store byte arrays
- need to materialize both follows and followed pairs
- access it quickly, without doing large scans .



Compare designs

	follows				
AK	1:foo	2:bar	3:baz	4:troy	count:4
foo	1:bar	2:AK	count:2		

	f
AK+foo	James Foo:1
AK+bar	Jimmy Bar:1
AK+baz	Ricky Baz:1
AK+troy	Troy:1
foo+bar	Jimmy Bar:1
foo+AK	AK:1

The alternative design: Benefits

- row key: *follower + followed* userid (i.e. *AK* follows *foo*);
- column family name has been shortened to f .

Getting a list of *followed* users becomes a short Scan

(i.e. scan all keys starting with "AK") instead of a Get operation .

- Little performance impact of that as Gets are internally implemented as Scans of length 1 .

- Unfollowing, simple row delete operation
- (i.e. if *AK* unfollows *bar* simply delete row *"AK+bar")*

Answering the question "Does A follow B?"

simple get row operations, respectively,

- No need to iterate through the entire list of users in the row in the earlier table designs . Significantly cheaper way of answering that question, espe- cially when list of followed users is large .

followed user's name

follower + followed userid

	f
AK+foo	James Foo:1
AK+bar	Jimmy Bar:1
AK+baz	Ricky Baz:1
AK+troy	Troy:1
foo+bar	Jimmy Bar:1
foo+AK	AK:1

References

[1] Introduction to Hbase Schema Design, Amandeep Khurana

[2] Fay Chang, et. al., "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (OSDI '06), USENIX, 2006, pp . 205–218 .

[3] "Graph databases", Ian Robinson et al., O'Reilly

[4]http://docs.mongodb.org/manual/core/sharding-introduction/

Popular DBs

Technology	Main Use	Other
Oracle	Relational	Supports json structures
Mysql	Relational	Supports json structures
Microsoft SQL Server	Relational	Supports json structures
PostgreSQL	Relational	Supports json structures
MongoDB	NoSQL/Document storage	Search Engine
Elastic Search	Search Engine	NoSQL/Document storage
Couchbase	Document storage	Key balue
Cassandra	Column storage	Key value
Hbase	Column storage	Key value
Redis	Key value	Document,Search Engine, Time Series
Dynamo	Key value	Document
Prometheus	Time series	
Neo4j	Graph	
Hive	NoSQL over other datastorage	HiveQL also found in Spark
Titan	Huge GRaphs	
Amazon DynamoDB	Document	Key value
Impala	Relational	Document
Solr	Search engine	