CSCI-620

NoSQL

Roadmap

1. Intro to NoSQL
2. Document databases
3. Key-value databases
4. Wide column stores
5. Graph databases

NoSQL is not “no SQL”

Vs of big data

1. Volume - there is a lot of data
2. Variety - data items look different
3. Velocity - new data is arriving **fast**
4. Veracity - is the data correct?
5. Value - can we do anything with it?

$X \bowtie Y$

Joins

Media

Drawbacks of RDBMS

Flexibility

Scalability

Drawbacks of RDBMS

Volume

- Maintaining ACID is expensive and not always necessary
- Sometimes we can deal with minor inconsistencies in our results
- We also want to be able to partition our data across multiple sites

Variety

- One single fixed data model makes it harder to incorporate varying data
- Sometimes when we pull from external sources, we don’t know the schema!
- Changing a schema in a relational database can be expensive

Velocity

- Storing everything durably to a disk all the time can be prohibitively expensive
- Sometimes it’s ok if we have a low probability of losing data
- Memory is much cheaper now, and much faster than always going to disk

What is NoSQL?

- No single accepted definition
- Flexible schema
  (unlike the relational model)
- Eventual consistency (not ACID)
- Often (but not always) better at handling **really** big data tasks

Flexible schema

- Different rows may have different attributes or structure
- The database often has no understanding of the schema
- It is up to applications to maintain a consistency in the schema including any denormalization

RDBMS Scalability

- ACID is hard in a distributed environment
- Relational databases usually have a primary server and 1+ replicas
- Writes go to the primary to ensure consistency

ACID vs BASE

- ACID databases give us Atomicity, Consistency, Isolation, and Durability

- This is not universal, but many NoSQL databases choose the **BASE** approach

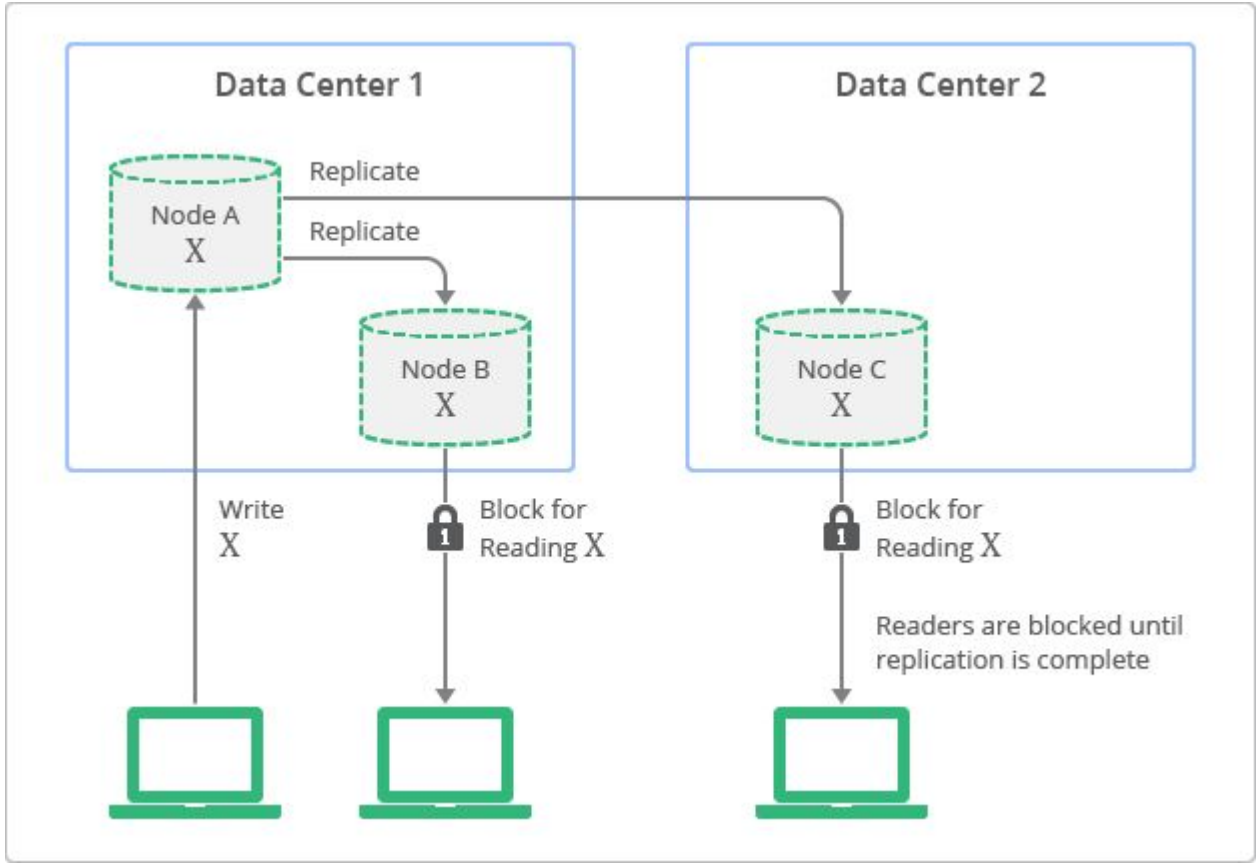
- **B**asically **A**vailable
- **S**oft state
- **E**ventual consistency

Eventual consistency

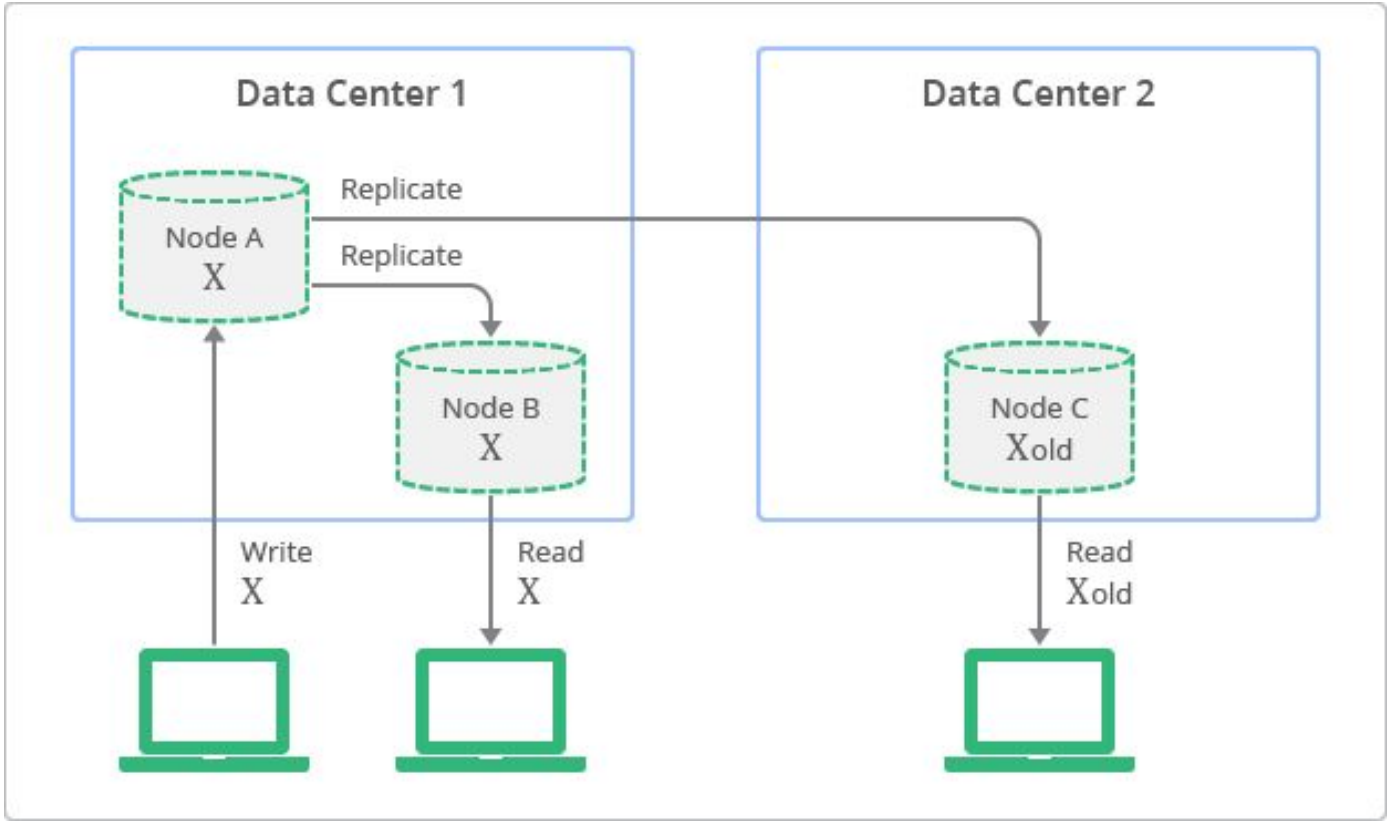
- Consistency is only guaranteed after some period of time when writes stop
- This means it is possible that queries will not see the latest data
- This is commonly implemented by storing data in memory and then lazily sending it to other machines

CAP theorem

- Consistency, Availability, Partition tolerance, pick two
- ACID databases are usually CP systems
- BASE databases are usually AP
- This distinction is blurry and often systems can be reconfigured to change these tradeoffs



Strong consistency



Eventual consistency

Categories of NoSQL

- Document stores
- Key-value databases
- Wide column stores
- Graph databases

Sales

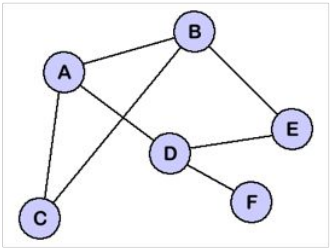
Product	Customer	Date	Sale
Beer	Thomas	2011-11-25	2 GBP
Beer	Thomas	2011-11-25	2 GBP
Vodka	Thomas	2011-11-25	10 GBP
Whiskey	Christian	2011-11-25	5 GBP
Whiskey	Christian	2011-11-25	5 GBP
Vodka	Alexei	2011-11-25	10 GBP
Vodka	Alexei	2011-11-25	10 GBP

Product

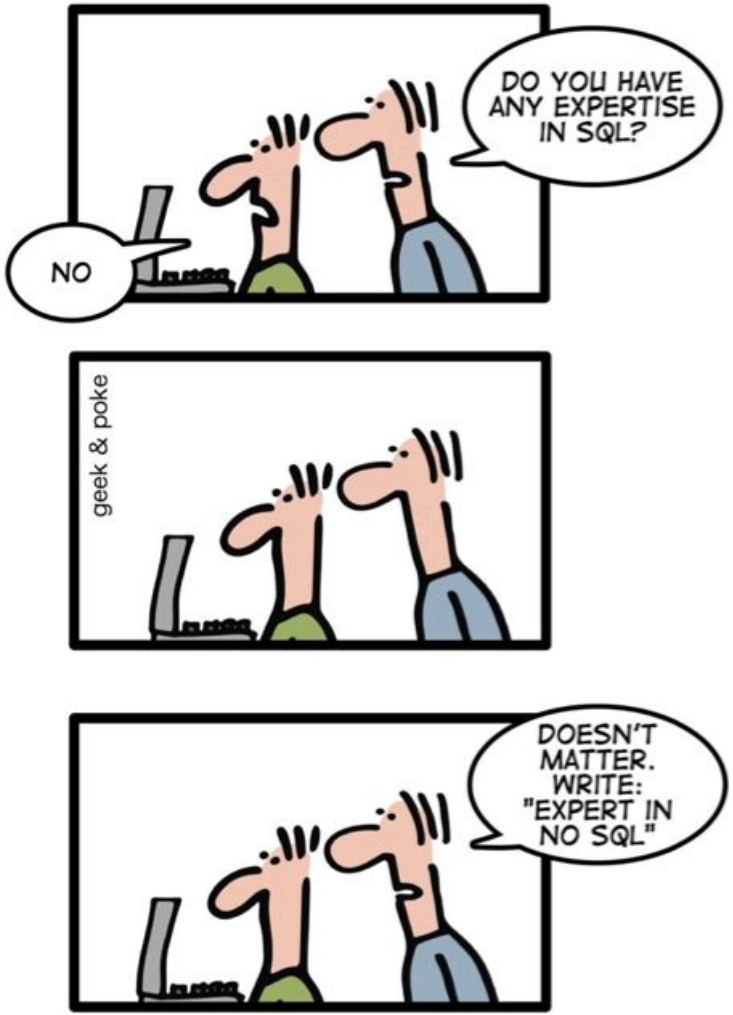
ID	Value
1	Beer
2	Beer
3	Vodka
4	Whiskey
5	Whiskey
6	Vodka
7	Vodka

Customer

ID	Customer
1	Thomas
2	Thomas
3	Thomas
4	Christian
5	Christian
6	Alexei
7	Alexei

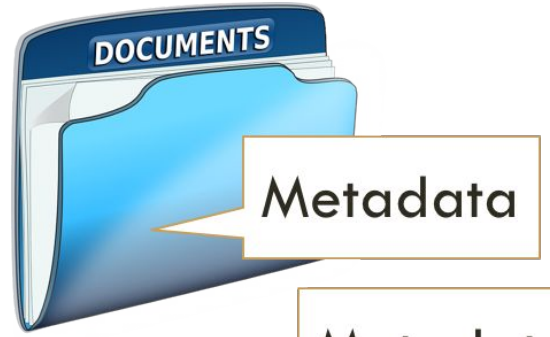


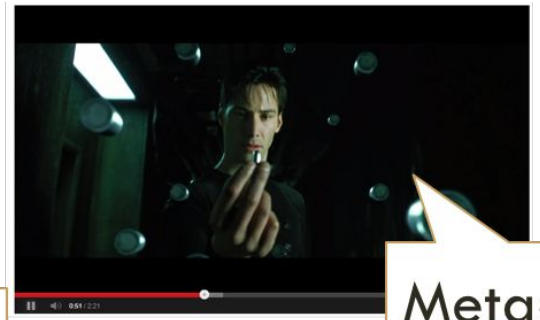
NoSQL database types


DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"

Roadmap

1. Intro to NoSQL
2. Document databases
3. Key-value databases
4. Wide column stores
5. Graph databases





Metadata

Patient		
ssn	firstName	lastName
235-14-7854	Sandra	Smith
192-48-0924	John	Moore
821-13-2108	Laura	Turner

Document databases

mongoDB

FOUNDATIONDB

RethinkDB

MarkLogic®

ArangoDB

Document databases

No standard!

```
{
  "foo": "bar",
  "baz": [1, true, 3, "quux"],
  "corge": {
    "grault": null
  }
}
```

JSON

JSON

- Taken from JavaScript
- Contains objects, strings, numbers, arrays, booleans, and null
- Objects can be arbitrarily nested

MongoDB

MongoDB hierarchy

```
db.createCollection(“bills”, [
 {option: value, …}])
```

Creating collections

```
{ “firstName”: “John”,
  “lastName”: "Smith”,
  “dob”: new ISODate(“1995-10-23”),
  “address”: {
    “street”: “21 2nd St”,
    “city”: “New York”,
    “state”: “NY”,
    “postalCode”: “10021-3100” },
  “phoneNumbers”: [
     { “type”: “home”,
       “number”: “212 555-1234”},
     { “type”: “office”,
       “number”: “646 555-4567” }
   ]
}
```

Sample document

Type	Number	Alias	Notes
Double	1	“double”	
String	2	“string”	
Object	3	“object”	
Array	4	“array”	
Binary data	5	“binData”	
Undefined	6	“undefined”	Deprecated.
ObjectId	7	“objectId”	
Boolean	8	“bool”	
Date	9	“date”	
Null	10	“null”	

BSON

Type	Number	Alias	Notes
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated.
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated.
JavaScript (with scope)	15	"javascriptWithScope"	
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit integer	18	"long"	
Decimal128	19	"decimal"	New in version 3.4.
Min key	-1	"minKey"	
Max key	127	"maxKey"	

BSON

```
db.bills.insert(
  { _id : ObjectId("5099803df3f4948bd2f98391"),
    "address" : { "zipcode" : "14534",
                  "city"    : "Pittsford",
                  "state"   : "NY" },
    "amount"  : 756.98,
    "patient" : "376-97-9845",
    "id"      : 883,
    "bDate"   : new ISODate("2010-10-01") } )
```

Need to be unique in practice!

Inserting a document

```
db.bills.insert(
  { “alias” : “BlackJack”,
    “email”  : “bj@yeah.com”} )
```

Inserting documents

Structure in practice

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

Structure validation

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

Bulk insert

```
db.bills.find({"patient": "376-97-9845"})
db.bills.find({"address.zip": "14534"})
db.bills.find({"amount": {$gte: 750}}) // >=
db.bills.find({"amount": {$gte: 750}, "address.zipcode": "14534"}) // AND
db.bills.find({ $or: [{“amount”: { $gt: 750 }}, {"address.zip": "14534"} ]})
```

Retrieving data

Comparison

For comparison of different BSON type values, see the specified BSON comparison order.

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

Query operators

Logical

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

Query operators

Element

Name	Description
`$exists`	Matches documents that have the specified field.
`$type`	Selects documents if a field is of the specified type.

Evaluation

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

Query operators

Array

Name	Description
`$all`	Matches arrays that contain all elements specified in the query.
`$elemMatch`	Selects documents if element in the array field matches all the specified `$elemMatch` conditions.
`$size`	Selects documents if the array field is a specified size.

Bitwise

Name	Description
`$bitsAllClear`	Matches numeric or binary values in which a set of bit positions *all* have a value of `0`.
`$bitsAllSet`	Matches numeric or binary values in which a set of bit positions *all* have a value of `1`.
`$bitsAnyClear`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `0`.
`$bitsAnySet`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `1`.

Query operators

MongoDB vs SQL

MongoDB does NOT support SQL, however I will include some SQL statements that are roughly equivalent. When asked to write a query in MongoDB, **always** use the MongoDB query language!

Using SQL will result in **zero** marks.

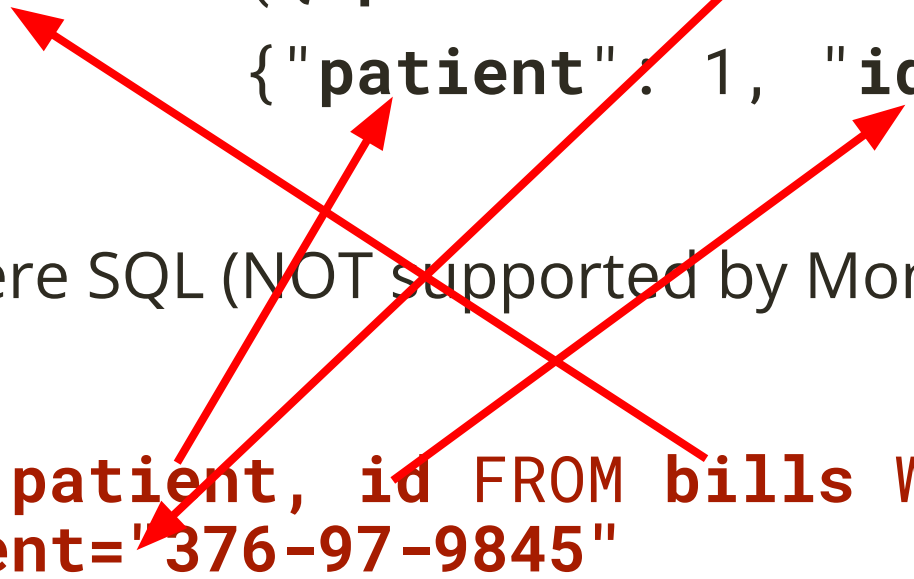
```
db.bills.find({"patient": "376-97-9845"},
              {"patient": 1, "id": 1 } )
```

If this were SQL (NOT supported by MongoDB)...

```
SELECT patient, id FROM bills WHERE
  patient="376-97-9845"
```

Projection

```
db.bills.find({"patient": "376-97-9845"},
              {"patient": 1, "id": 1 } )
```



If this were SQL (NOT supported by MongoDB)...

```
SELECT patient, id FROM bills WHERE
  patient="376-97-9845"
```

Projection

```
db.bills.find().sort(
    {"patient": 1,
     "address.zip": -1})
```

```
SELECT * FROM bills ORDER by
  patient ASC, address.zip DESC
```

Sorting

```
db.bills.update(
  {"patient": "376-97-9845"},
  {$set: {"address.zip": "14626",
          "address.city": "Greece"}})
```

```
UPDATE bills SET address.zip="14626",
  address.city="Greece" WHERE
  patient="376-97-9845"
```

Updating documents

```
db.bills.remove(
  {"patient": "376-97-9845"})
```

```
DELETE FROM WHERE patient="376-97-9845"
```

Removing documents

```
{field: {$exists: <boolean>}}
```

Not the same as **EXISTS** in SQL!

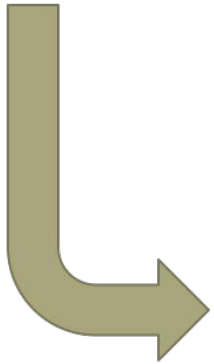
Exists

```
{ $project: { <specification(s)> } }
```

- <field>: <1 or true>, <field>:<0 or false>
  - Inclusion/exclusion of a field
- _id: <0 or false>
  - Suppression of the _id field
- <field>: <expression>
  - Add a new field or reset the value of an existing field.
- If you specify the exclusion of a field other than _id, you cannot employ any other $project specification forms.

Projection stage

```
db.bills.aggregate( [
    { $project : { “address.zipcode” : 1,
                   “amount”  : 1,
                   “patient” : 1 } } ] )
```



```
{ _id : “5099803df3f4948bd2f98391”,
  “address” : { “zipcode” : “14534” },
  “amount”  : 756.98,
  “patient”  : “376-97-9845” }
```

Projection stage

```
db.bills.aggregate( [
        { $project : { “address.zipcode” : 1,
                       “amount”  : 1,
                       “patient” : 1,
                       _id  : 0 } } ] )
```

Documents with no _ids!

Projection stage

```
{ $match: { <query> } }
```

Match stage

```
db.bills.aggregate([
  {$match : {"address.zip": "14534"}}])
```

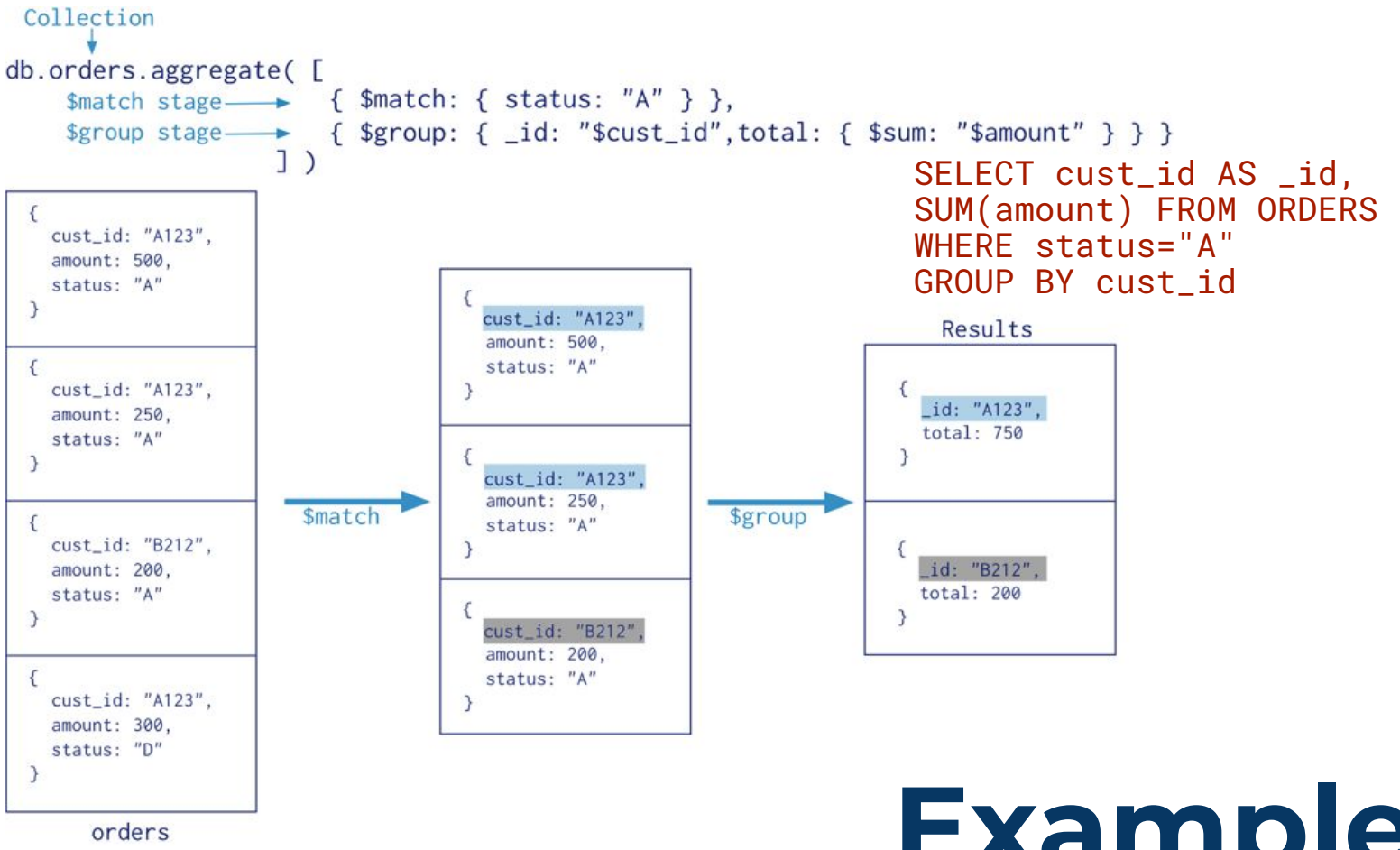
same as

```
db.bills.find({"address.zip": "14534"})
```

Match stage

```
{ $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }
```

Group stage


Collection
db.orders.aggregate([
$match stage
{ $match: { status: "A" } },
$group stage
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
SELECT cust_id AS _id,
SUM(amount) FROM ORDERS
WHERE status="A"
GROUP BY cust_id
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

Example

`$sum`	Sum
`$avg`	Average
`$first`	First value in group
`$last`	Last value in group
`$max`	Maximum
`$min`	Minimum
`$push`	Array of all values in group
`$addToSet`	Distinct array of group values
`$stdDevPop`	Population standard deviation
`$stdDevSamp`	Sample standard deviation

Accumulators

```
db.bills.aggregate(
  [$group: {_id : null,
            avgAmount: {$avg: "$amount"}])
```

```
SELECT AVG(amount) FROM bills
```

Grouping by null

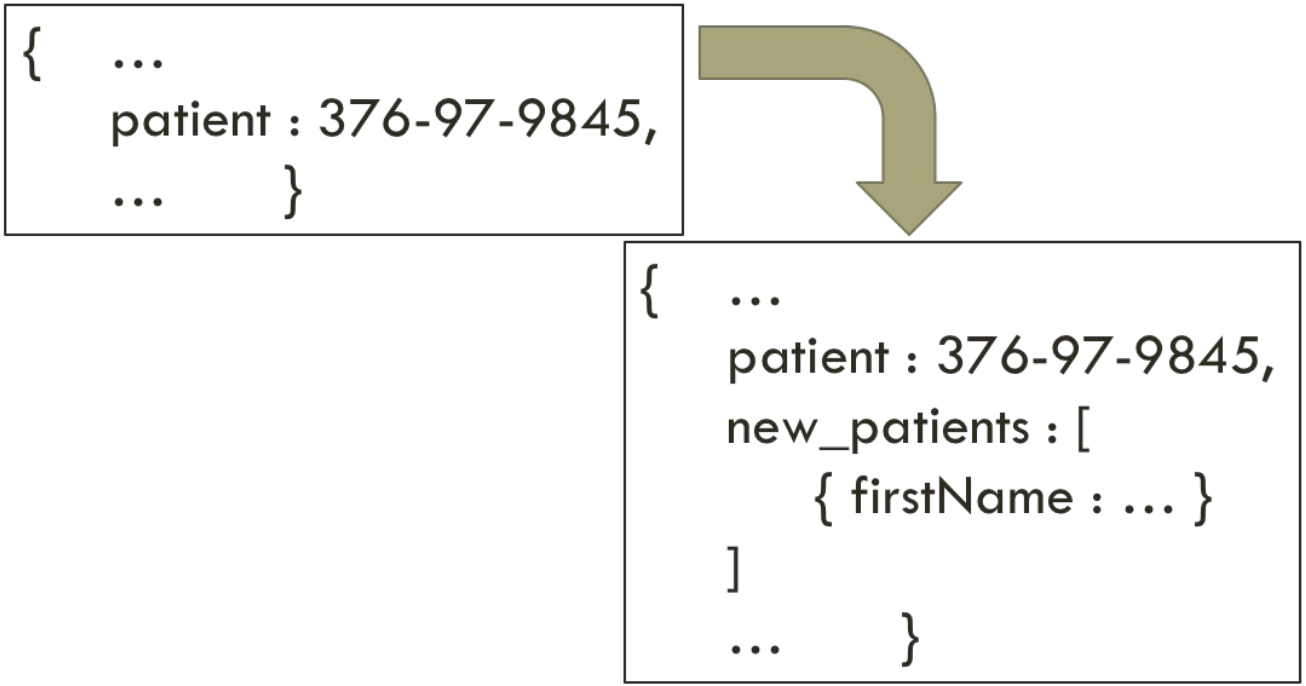
```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

Lookup stage

```
db.bills.aggregate(
    [$lookup: {from: patients,
               localField : "patient",
               foreignField : "_id",
               as: "new_patients"}])
```

```
SELECT * FROM bills JOIN patients AS new_patients
ON bills.patient = new_patients._id
```

Lookup stage



Lookup stage

```
db.patients.insert(
  { _id : "376-97-9845",
    "firstName" : "Jennifer",
    "visits"  : [758, 345],
    … } )
```

```
db.visits.insert(
  { _id : 345,
    "doctor" : "893-12-8934",
    "otherNotes"  : "fever",
    … } )
db.visits.insert(
  { _id : 758,
    "doctor" : "9094-56-9292",
    "otherNotes"  : "neck",
    … } )
```

Lookup stage

```
db.patients.aggregate(
    [$lookup: {from: visits,
               localField: "visits",
               foreignField: "_id",
               as: "visitsInfo"}])
```

```
SELECT * FROM patients JOIN visits AS
visitsInfo ON patients.visits = visitsInfo._id
```

Lookup stage

```
{ _id : “376-97-9845”,
    “firstName” : “Jennifer”,
    “visits”  : [758, 345],
    “visitsInfo” : [
      { _id : 345,
        “doctor” : “893-12-8934”,
        “otherNotes”  : “fever”,
        … },
```

```
    { _id : 758,
      “doctor” : “9094-56-9292”,
      “otherNotes”  : “neck”,
      … }
    ],
… }
```

Lookup stage

```
{ $unwind: <field path> }
```

Unwind stage

Consider an `inventory` with the following document:

```
{ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L"] }
```

The following aggregation uses the **`$unwind`** stage to output a document for each element in the `sizes` array:

```
db.inventory.aggregate( [ { $unwind : "$sizes" } ] )
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```

Documents with the same _ids!

Unwind stage

```
db.records.createIndex( { score: 1 } )
```

Indexes

Roadmap

1. Intro to NoSQL
2. Document databases
3. Key-value databases
4. Wide column stores
5. Graph databases

Key-value (KV) databases

No standard!

Key-value databases

- Updates to the value for a single key are usually atomic
- Many KV databases allow for transactions which use multiple keys
- Values have limited structure

Upsides

- Key-value databases are generally easier to run in a distributed fashion
- Queries and updates usually very fast
- Any type of data in any structure can be stored as a value

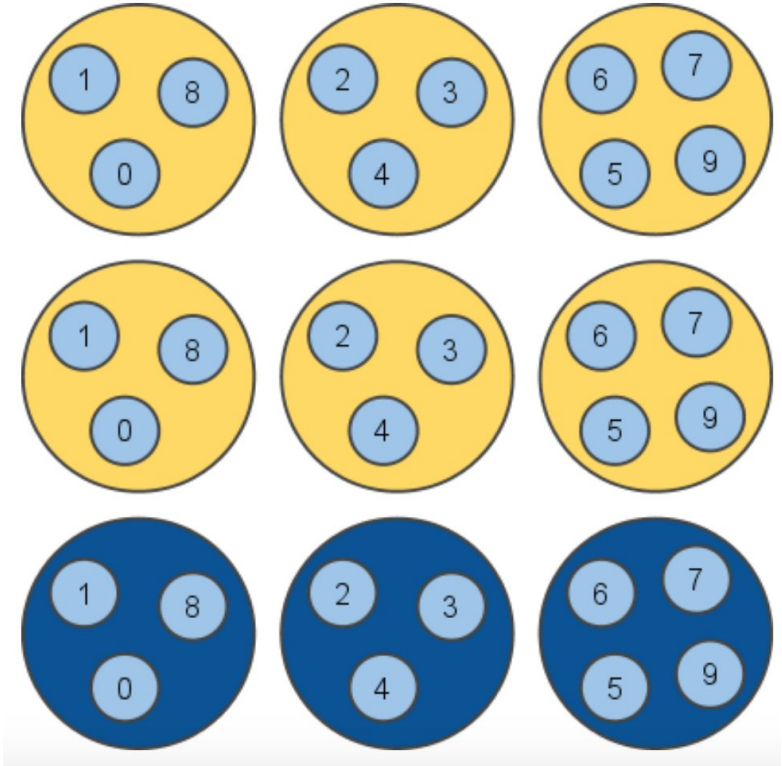
Downsides

- **Very** simple queries (usually just get a value given a key, sometimes a range)
- No referential integrity
- Limited transactional capabilities
- No schema to understand the data

Redis

Replicas

Primary



Scalability

Redis data model

- Basically a huge distributed hash table (DHT) with little structure to values
- All values are identified by a key which is a simple string
- If we want more structure in our keys, it has to be defined by our application
  e.g., user 3 could have the key “user:3”

Redis values

- Commonly in key-value stores, values are just an arbitrary blob of data

- Redis (and some other KV stores) allows some structure:
  - Lists
  - Sets
  - Hashes

```
SET foo bar

SADD news:1000:tags 1 2 5 77

LPUSH mylist first

HMSET user:1 username antirez birthyear 1977
```

Inserting data in Redis

```
GET foo => bar

SMEMBERS news:1000:tags => 1 2 5 77

LPOP mylist => first

HMGET user:1 username birthyear =>
                    antirez 1977
```

Retrieving data in Redis

When to use KV databases

- When you need something *really* fast
- When your data does not have a lot of structure/relationships
- For simple caching of data which is pulled from another source

Roadmap

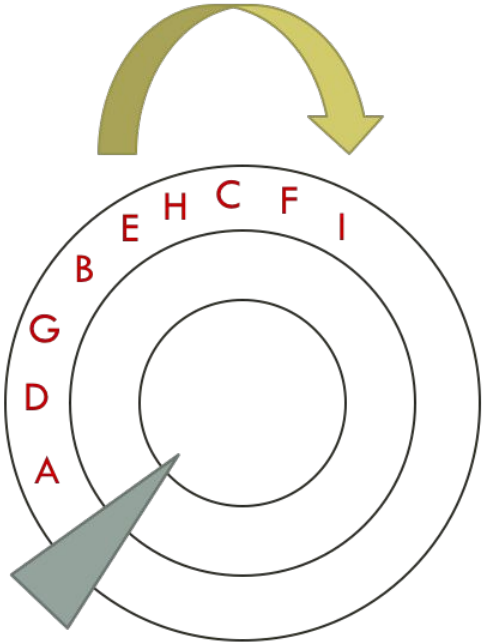
1. Intro to NoSQL
2. Document databases
3. Key-value databases
4. Wide column stores
5. Graph databases

Patient		
ssn	firstName	lastName
235-14-7854	Sandra	Smith
192-48-0924	John	Moore
821-13-2108	Laura	Turner

Wide-column stores

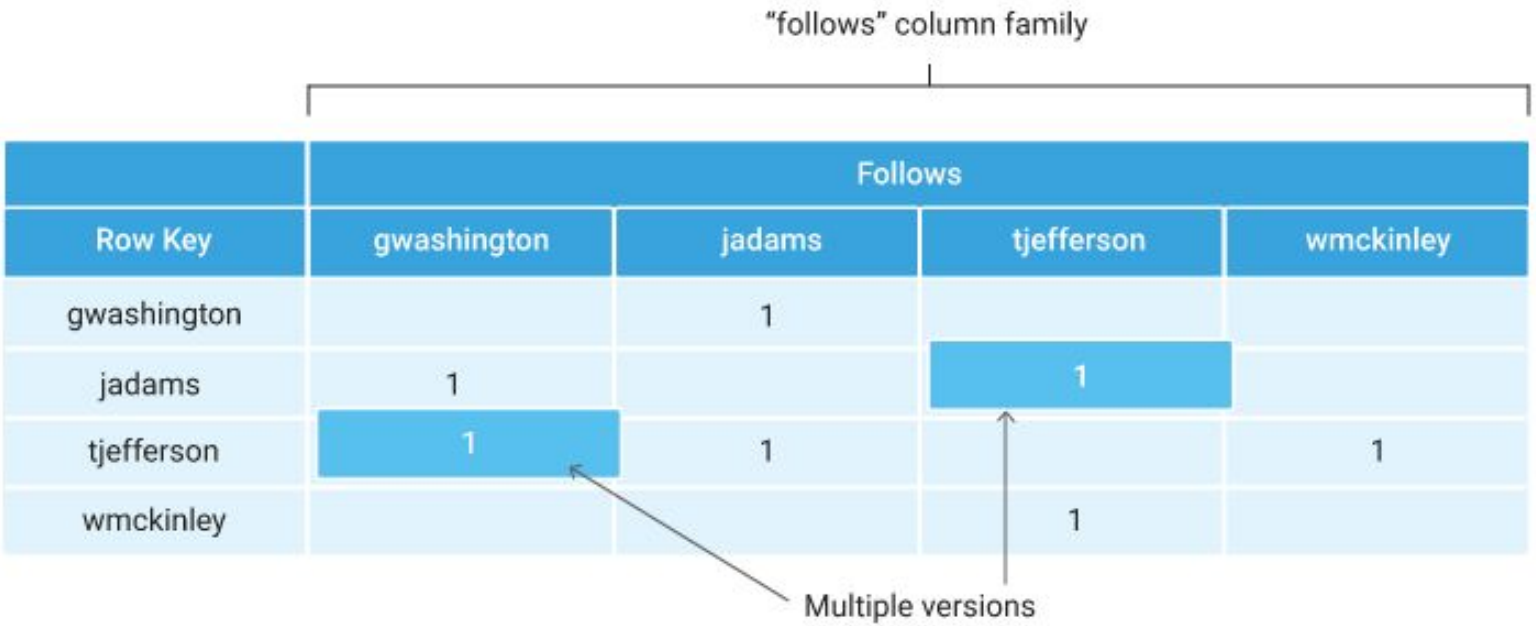
Patient		
ssn	firstName	lastName
A 235-14-7854	B Sandra	C Smith
D 192-48-0924	E John	F Moore
G 821-13-2108	H Laura	I Turner

```
SELECT ssn
FROM Patient;
```



Hard disk

Wide-column stores

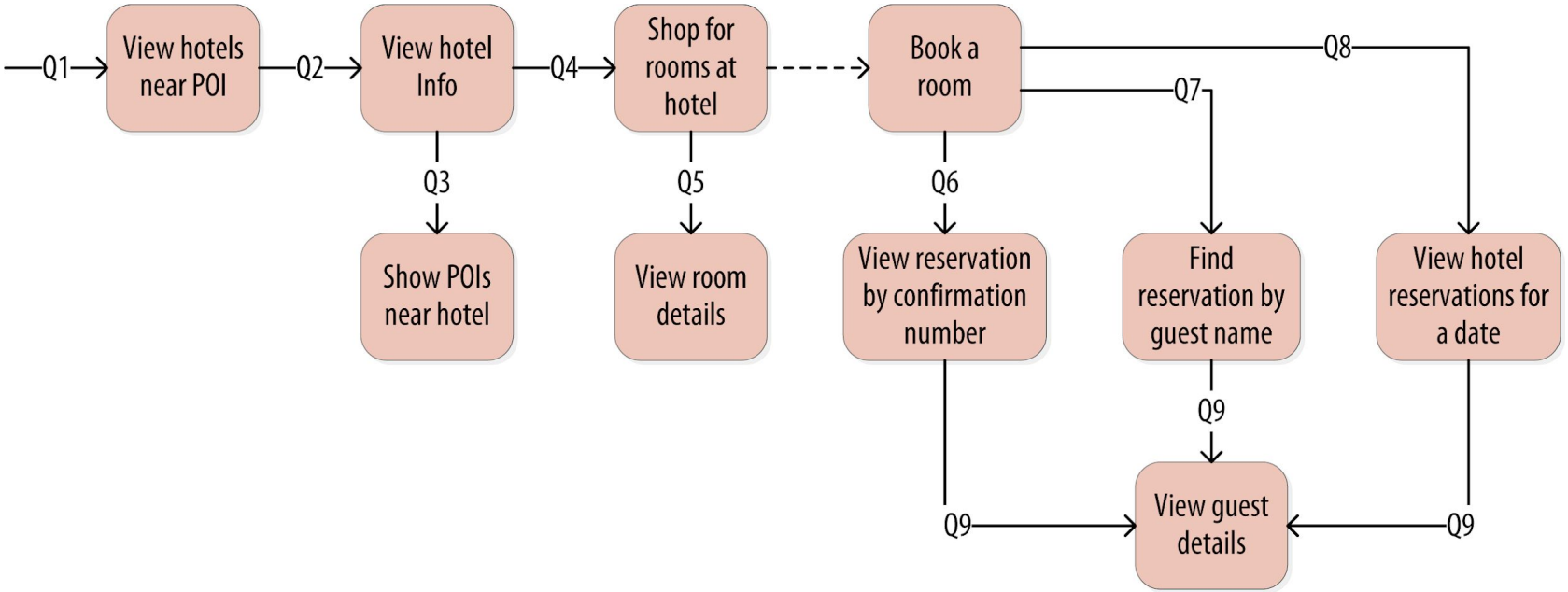


Wide-column stores

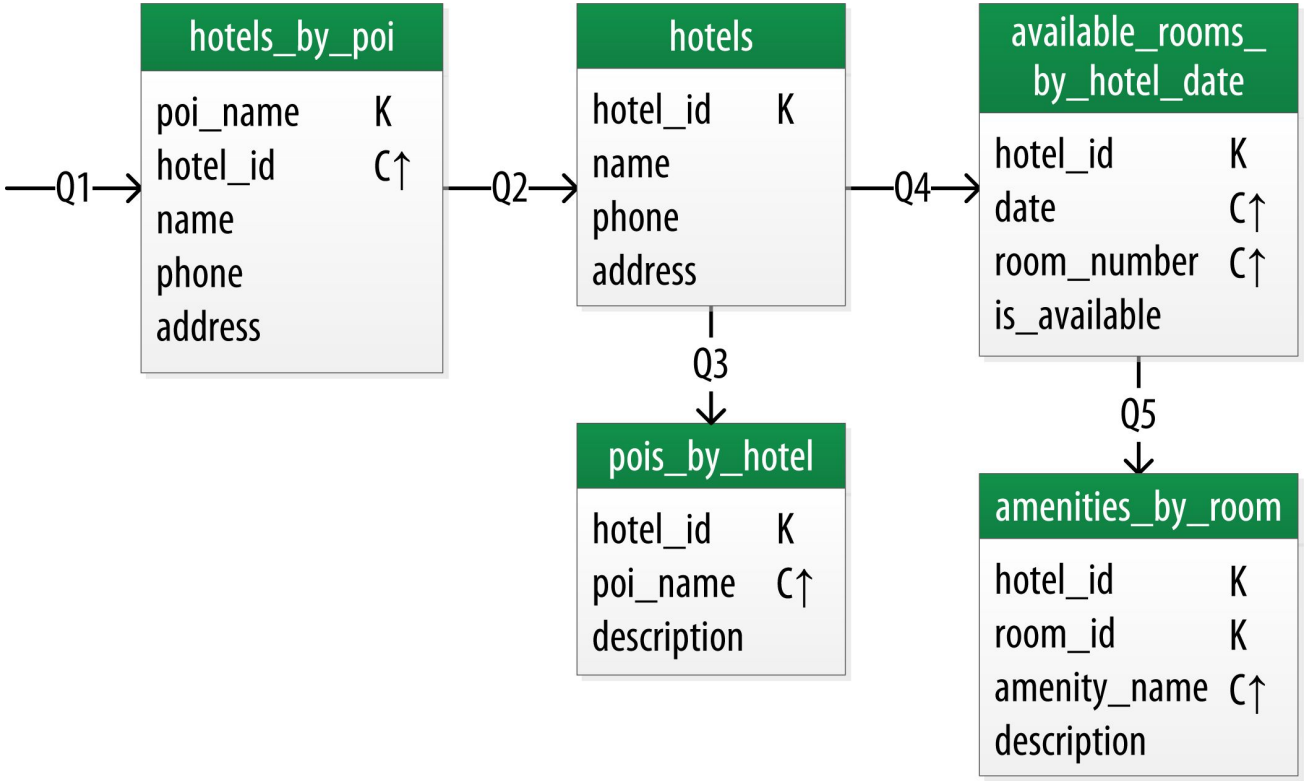
No standard!

cassandra

Cassandra



Logical model



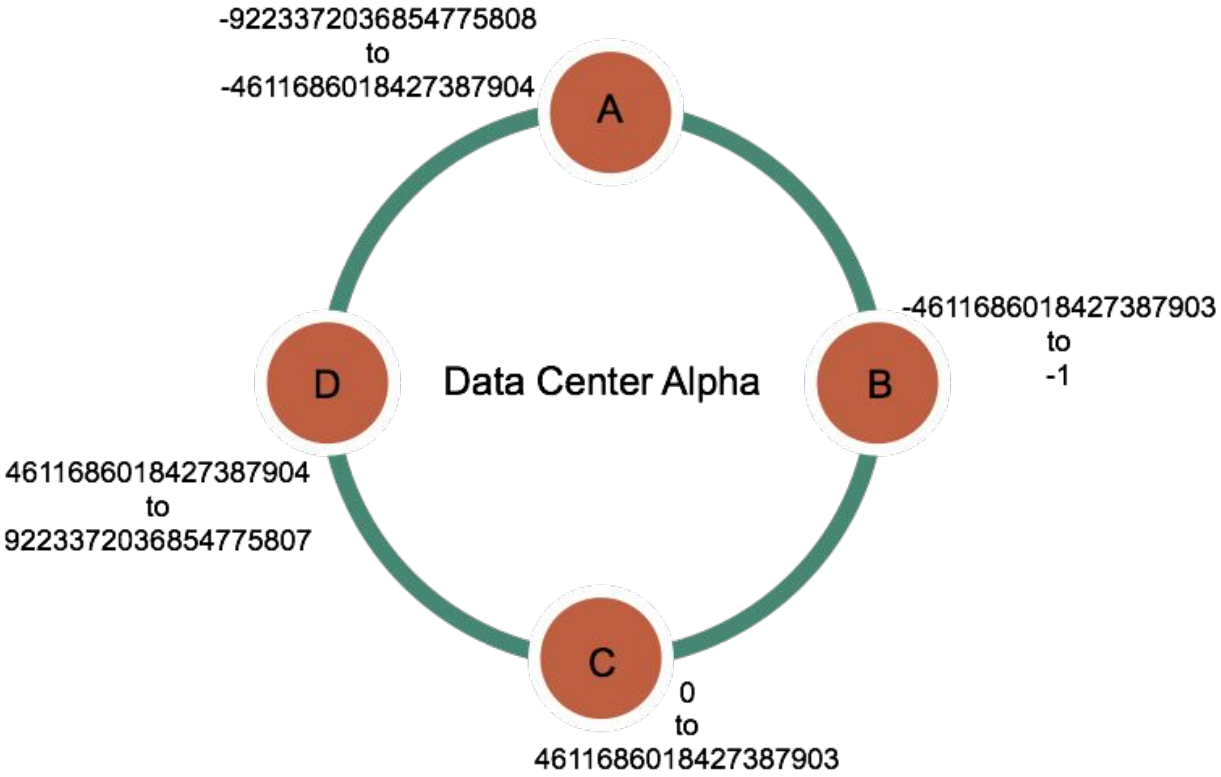
Logical model

```
SELECT name, phone, address
FROM hotels
WHERE
    hotel_id = “127” OR
    hotel_id = “349”;
```

Querying using CQL

```
SELECT name, phone, address
FROM hotels
WHERE
    phone = “12345678910”;
```

Querying using CQL



Scalability

Consistent hashing

- To find what data a node should hold, hash values are organized in a ring
- Each node is responsible for a portion of the ring
- When nodes are added or removed, this minimizes data movement

Upsides

- Very fast for writes
- Some structure to data and queries are somewhat familiar
- Highly available since there is no single point of failure

Downsides

- Limited queries (no joins!)
- No referential integrity
- Eventual consistency

Denormalization

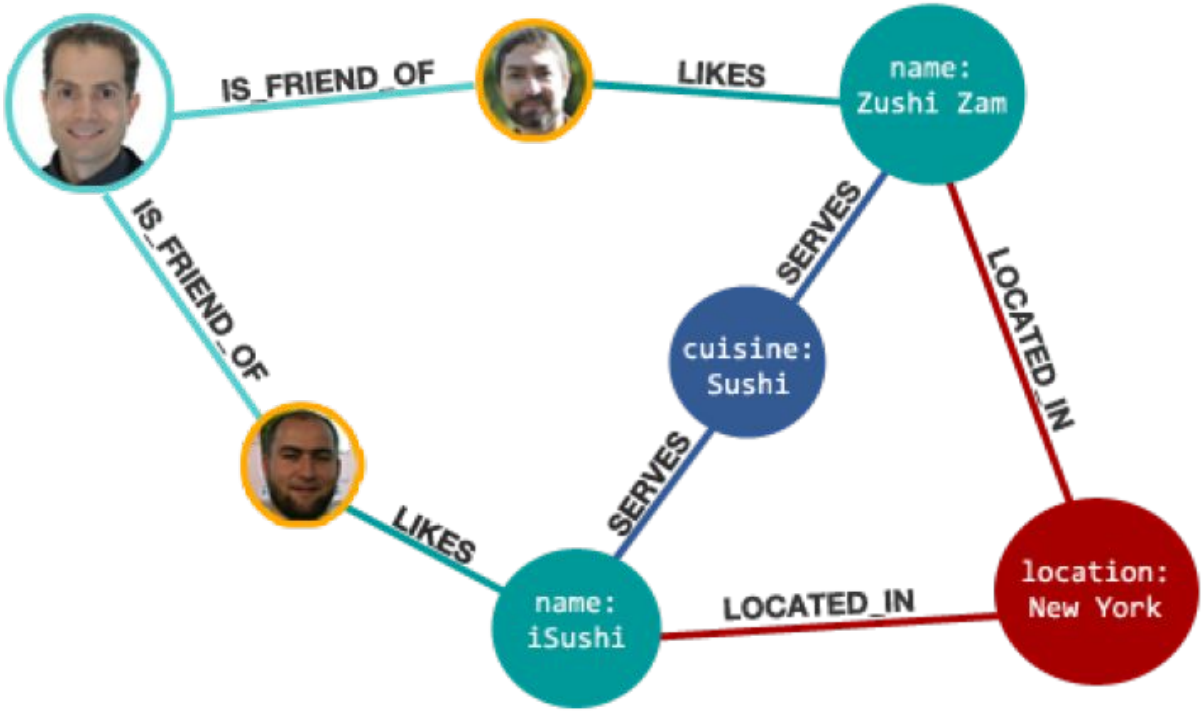
- Since the queries that can be issued are very simple, denormalization is often required
- A common approach is to create a column family for each type of query and pre-compute results
- Some systems support materialized views and indexes

When to use wide column stores

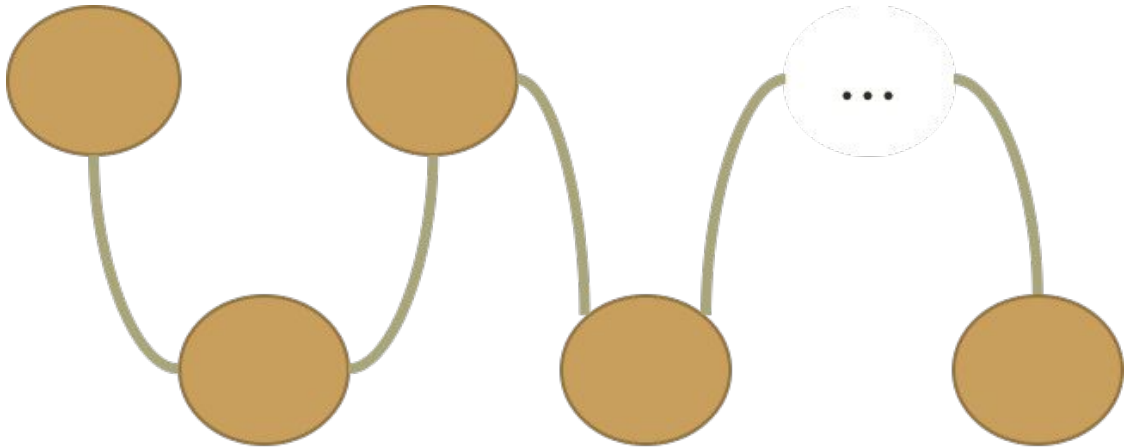
- Especially great for time series data (e.g. sensor measurements, logs)
- Simple data with predictable structure
- When you are writing a lot of new data but rarely updating old data
- If you don’t need transactions or complex queries

Roadmap

1. Intro to NoSQL
2. Document databases
3. Key-value databases
4. Wide column stores
5. Graph databases



Graph databases



Unbounded queries

Graph databases

Friend	
person	contact
John	Matthew
Matthew	Jennifer
Jennifer	Paul

```
SELECT person, contact
FROM Friend
WH
```

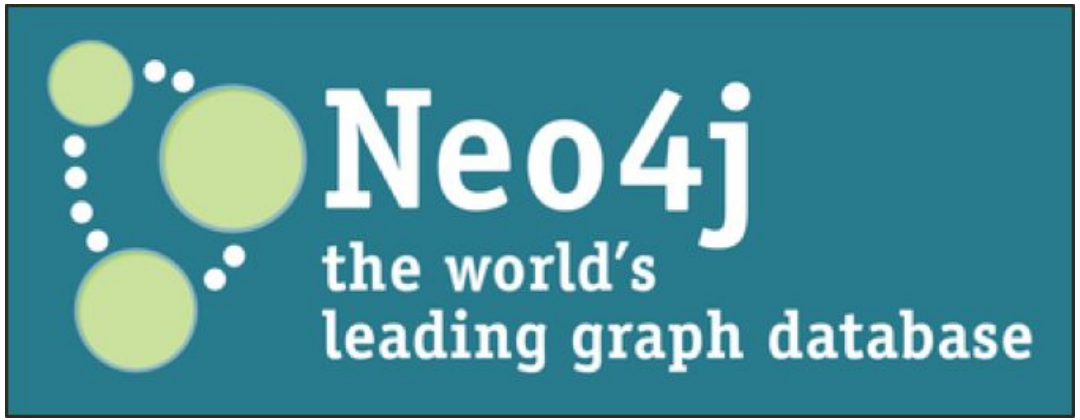
```
SELECT f1.person, f2.contact
FROM Friend AS f1  JOIN
     Friend AS f2 ON
          f1.contact = f2.person
WHERE
     f1.person =“John” AND
          f2.contact = “Paul”;
```

Unbounded SQL queries

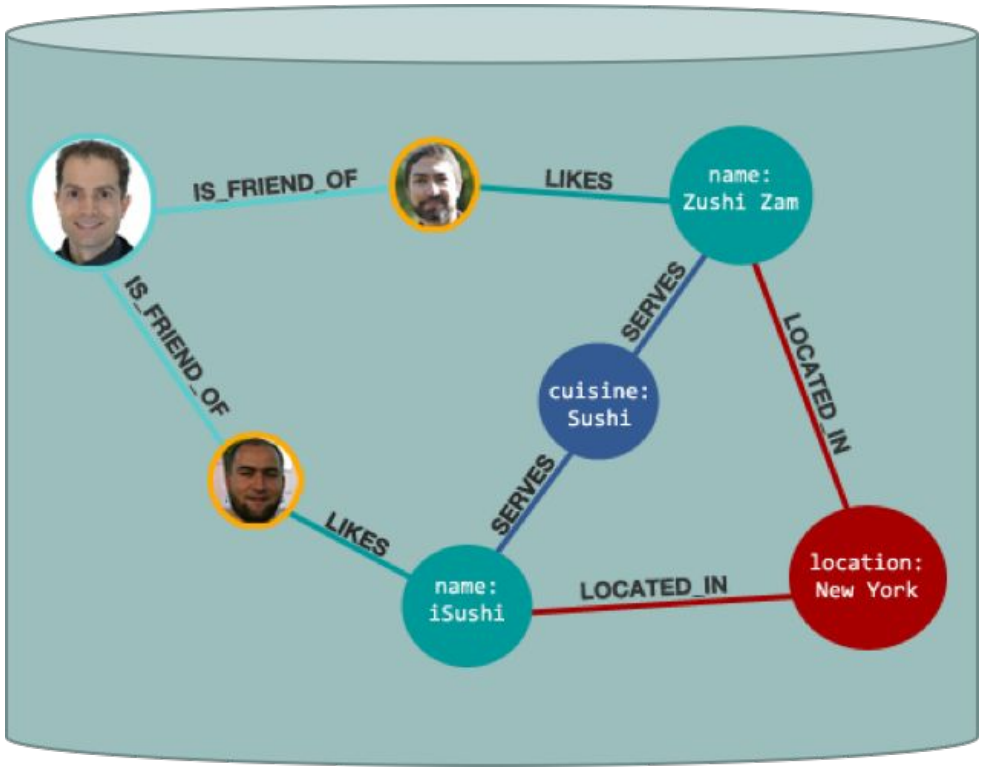
Friend	
person	contact
John	Matthew
Matthew	Jennifer
Jennifer	Paul

```
WITH RECURSIVE friends(name)
AS (VALUES('John') UNION
SELECT contact from friend JOIN
friends ON person=name)
SELECT * from friends OFFSET 1;
```

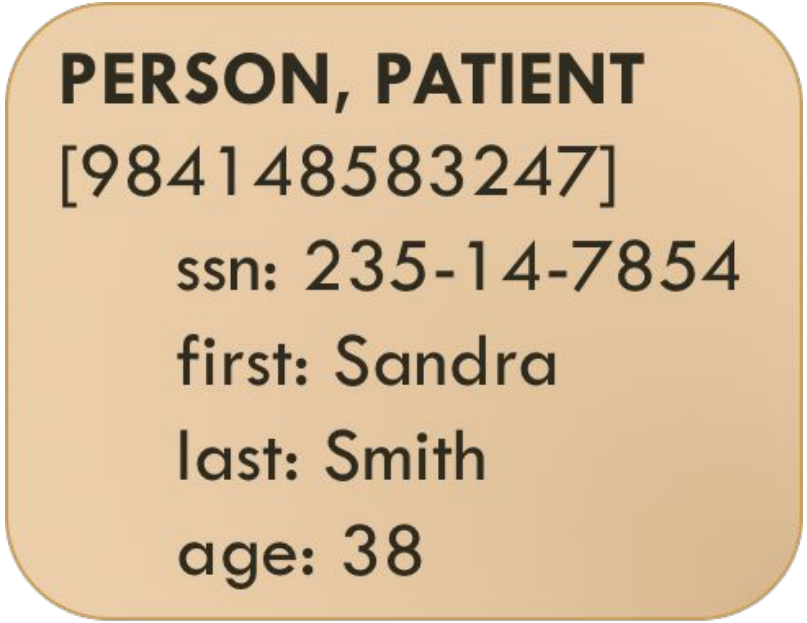
Unbounded SQL queries



Neo4j



Databases and graphs


PERSON, PATIENT
[98414858324 7]
ssn: 235-14-7854
first: Sandra
last: Smith
age: 38

Nodes

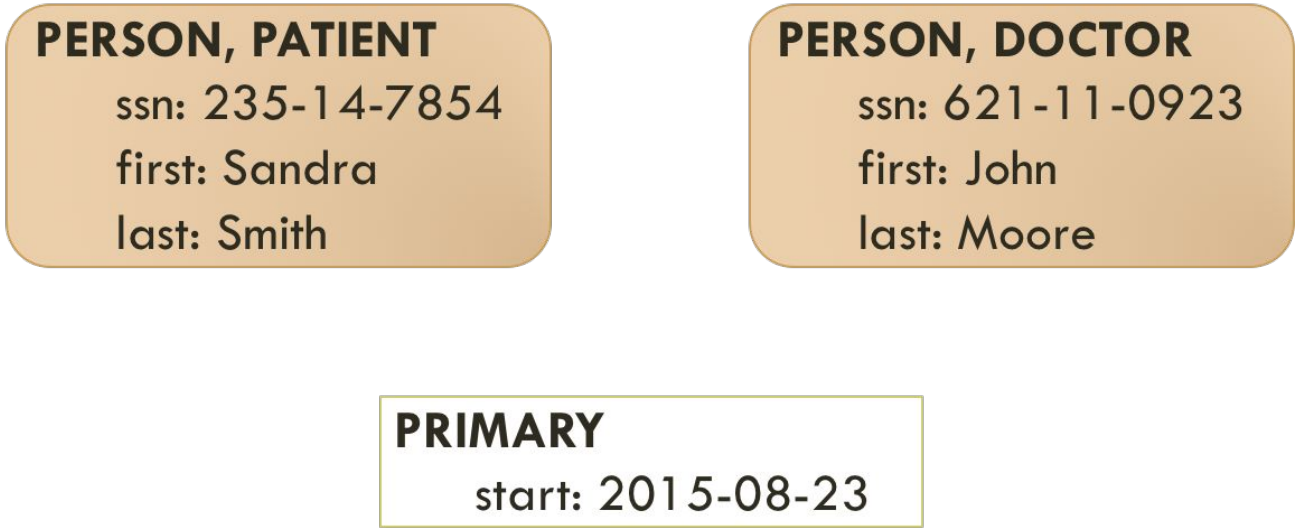
```
CREATE (n:PERSON {first: 'Sandra'})
```

```
Map<String, Object> attributesSandra =
        new HashMap<>();
attributesSandra.put("first", "Sandra");
long sandraNode = inserter.createNode(
        attributesSandra, Label.label("PERSON"));
```

Creating a node

```
inserter.createNode(1L,
        attributesSandra, Label.label("PERSON"),
                Label.label(“PATIENT"));
```

Internal node IDs



Relationships

```
CREATE (p)-[:PRIMARY {start: '2015-08-23'}]->(d)
```

```
Map<String, Object> att =
        new HashMap<>();
att.put("start", "2015-08-23");
inserter.createRelationship(sandra, john,
        RelationshipType.withName("PRIMARY"), att);
```

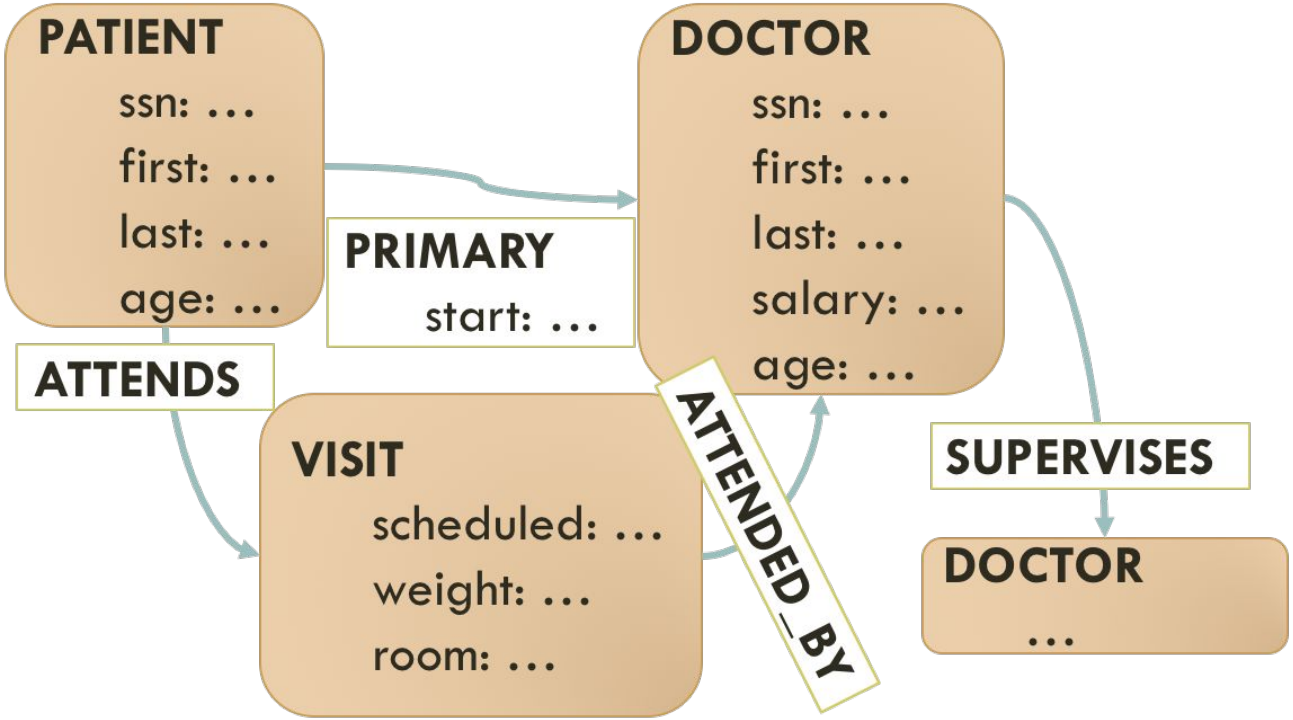
Creating edges

Logical model

```
CREATE CONSTRAINT ON (p:PATIENT)
    ASSERT p.ssn IS UNIQUE;

CREATE CONSTRAINT ON (p:PATIENT)
    ASSERT EXISTS(p.ssn);
```

Logical model


PATIENT
ssn: ...
first: ...
last: ...
age: ...
PRIMARY
start: ...
DOCTOR
ssn: ...
first: ...
last: ...
salary: ...
age: ...
ATTENDS
VISIT
scheduled: ...
weight: ...
room: ...
ATTENDED_BY
SUPERVISES
DOCTOR
...

Sample data

```
MATCH (p:PATIENT)
WHERE p.age > 30
RETURN p.last
ORDER BY p.last ASC
```

Single node queries

```
MATCH (p:PATIENT) --> (d:Doctor)
WHERE p.age > 30 AND
        d.salary > $100K
RETURN (p)
```

Queries with edges

```
MATCH (p:PATIENT) -[:PRIMARY]- (d:Doctor)
WHERE p.age > 30 AND
        d.salary > $100K
RETURN (p)
```

Queries with edges

```
MATCH (d:DOCTOR)
WHERE d.age > 30
RETURN AVG(d.salary)
```

Aggregation