SQL vs NoSQL

Νίκος Βορνιωτάκης

June 9, 2016

Περιεχόμενα

Layout

- Εισαγωγή
  - Έννοιες
  - Relational
- Τύποι NoSQL βάσεων
  - Document
  - Keystore
  - Column
  - Graph
- Use Case
  - Wikipedia
  - Twitter
  - Others
- Conclusion
  - Conclusions

NoSQL Databases

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”
- Sometimes called ”Not only SQL” to emphasize that they may support SQL-like query languages

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”
- Sometimes called ”Not only SQL” to emphasize that they may support SQL-like query languages

Characteristics

- Distributed (designed to scale)

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”
- Sometimes called ”Not only SQL” to emphasize that they may support SQL-like query languages

Characteristics

- Distributed (designed to scale)
- Web friendly (RESTfull, JSON aware etc)

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”
- Sometimes called ”Not only SQL” to emphasize that they may support SQL-like query languages

Characteristics

- Distributed (designed to scale)
- Web friendly (RESTfull, JSON aware etc)
- No strict schema

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”
- Sometimes called ”Not only SQL” to emphasize that they may support SQL-like query languages

Characteristics

- Distributed (designed to scale)
- Web friendly (RESTfull, JSON aware etc)
- No strict schema
- Can support BIGDATA

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”
- Sometimes called ”Not only SQL” to emphasize that they may support SQL-like query languages

Characteristics

- Distributed (designed to scale)
- Web friendly (RESTfull, JSON aware etc)
- No strict schema
- Can support BIGDATA
- Performance gains usually when many nodes are used

NoSQL

Definition

- Originally referring to ”non SQL” or ”non relational”
- Sometimes called ”Not only SQL” to emphasize that they may support SQL-like query languages

Characteristics

- Distributed (designed to scale)
- Web friendly (RESTfull, JSON aware etc)
- No strict schema
- Can support BIGDATA
- Performance gains usually when many nodes are used
- Suitable for commodity hardware

CAP theorem

Theorem

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

CAP theorem

Theorem

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- *Consistency (all nodes see the same data at the same time)*

CAP theorem

Theorem

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- *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)*

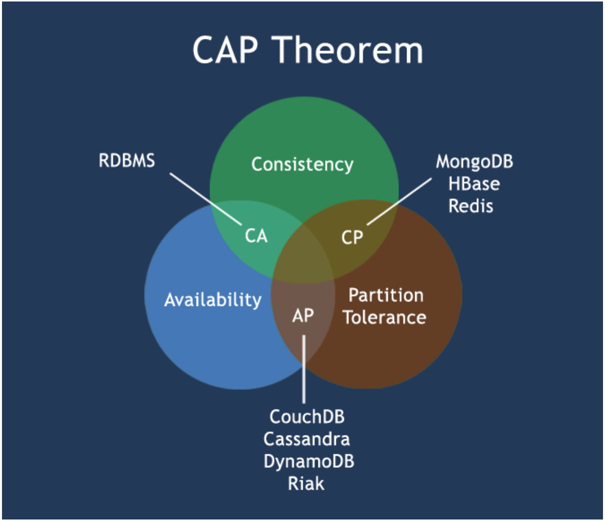
CAP theorem

Theorem

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- *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 partitioning due to network failures)*

CAP theorem


CAP Theorem
RDBMS
Consistency
MongoDB
HBase
Redis
CA
CP
Availability
AP
Partition
Tolerance
CouchDB
Cassandra
DynamoDB
Riak

ACID

Transactional guarantees

ACID

Transactional guarantees

Atomicity each transaction is ”all or nothing”: if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged

ACID

Transactional guarantees

Atomicity each transaction is ”all or nothing”: if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged

Consistency any transaction will bring the database from one valid state to another

ACID

Transactional guarantees

Atomicity each transaction is ”all or nothing”: if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged

Consistency any transaction will bring the database from one valid state to another

Isolation concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially

ACID

Transactional guarantees

Atomicity each transaction is ”all or nothing”: if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged

Consistency any transaction will bring the database from one valid state to another

Isolation concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially

Durability once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors

ACID

Transactional guarantees

Atomicity each transaction is ”all or nothing”: if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged

Consistency any transaction will bring the database from one valid state to another

Isolation concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially

Durability once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors

ACID

Transactional guarantees

RDBMSs are Transactional

RDBMSs provide ACID guarantees.

ACID

Transactional guarantees

RDBMSs are Transactional

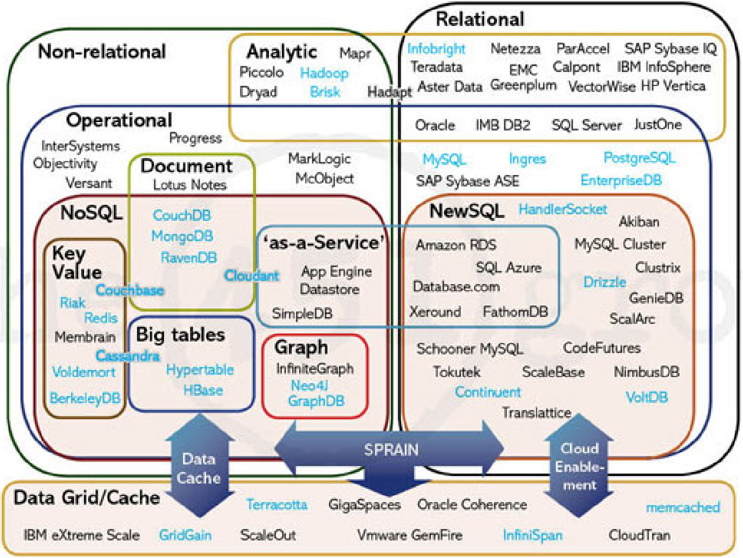
RDBMSs provide ACID guarantees.

Most NoSQL are **not** Transactional

Most NoSQL do not provide ACID.

- On a single node ACID is ”simple”
- On distributed systems to provide ACID guarantees is **realy** hard

Database Classification



NewSQL

Characteristics

- Relational

NewSQL

Characteristics

- Relational
- Distributed (designed to scale)

NewSQL

Characteristics

- Relational
- Distributed (designed to scale)
- ACID

NewSQL

Characteristics

- Relational
- Distributed (designed to scale)
- ACID
- Circumvents CAP by using distributed subsystems

NewSQL

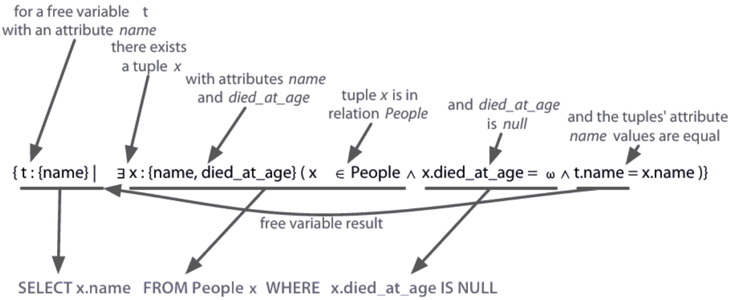
Characteristics

- Relational
- Distributed (designed to scale)
- ACID
- Circumvents CAP by using distributed subsystems
- Relies on extremely complex distributed algorithms to maintain consistency. Heavy use of hardware-assisted time synchronization using GPS clocks and atomic clocks to ensure global consistency.

Layout

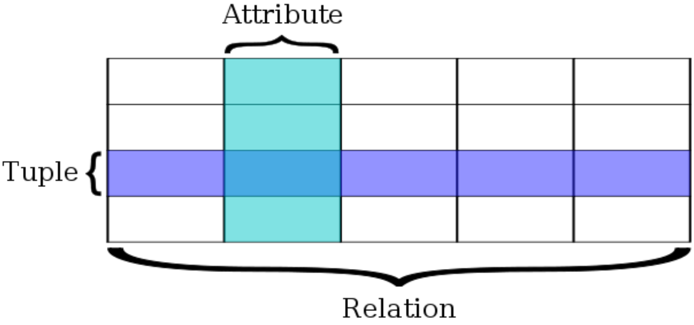
Relational model

Relational model comes straight from relational algebra



Relational model

Relation (relational algebra) = Table (SQL)



Relational model

Structured Query Language (SQL)

A language to make queries.

The current ISO SQL standard doesn’t mention the relational model or use relational terms or concepts. However, it is possible to create a database conforming to the relational model using SQL if one does not use certain SQL features.

Relational model

Structured Query Language (SQL)

A language to make queries.

The current ISO SQL standard doesn't mention the relational model or use relational terms or concepts. However, it is possible to create a database conforming to the relational model using SQL if one does not use certain SQL features.

```
SELECT Book.title AS Title,
       count(*) AS Authors
FROM  Book
JOIN  Book_author
  ON  Book.isbn = Book_author.isbn
GROUP BY Book.title;
```

Relational model

Structured Query Language (SQL)

A language to make queries.

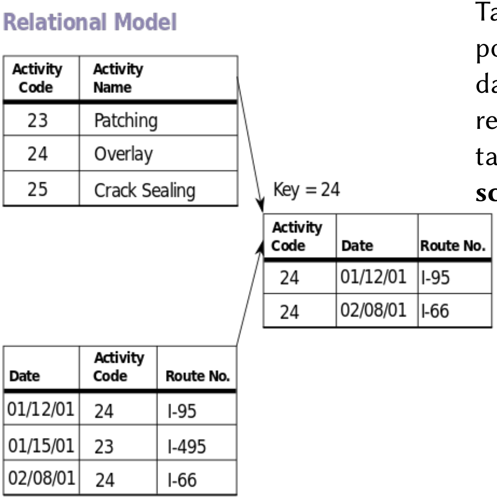
The current ISO SQL standard doesn't mention the relational model or use relational terms or concepts. However, it is possible to create a database conforming to the relational model using SQL if one does not use certain SQL features.

```
SELECT Book.title AS Title,
       count(*) AS Authors
FROM  Book
JOIN  Book_author
  ON  Book.isbn = Book_author.isbn
GROUP BY Book.title;
```

output

```
      Title             Authors
---------------------- -------
SQL Examples and Guide 4
The Joy of SQL         1
An Introduction to SQL 2
Pitfalls of SQL        1
```

Relational model


Relational Model

Activity Code	Activity Name
23	Patching
24	Overlay
25	Crack Sealing

Key = 24

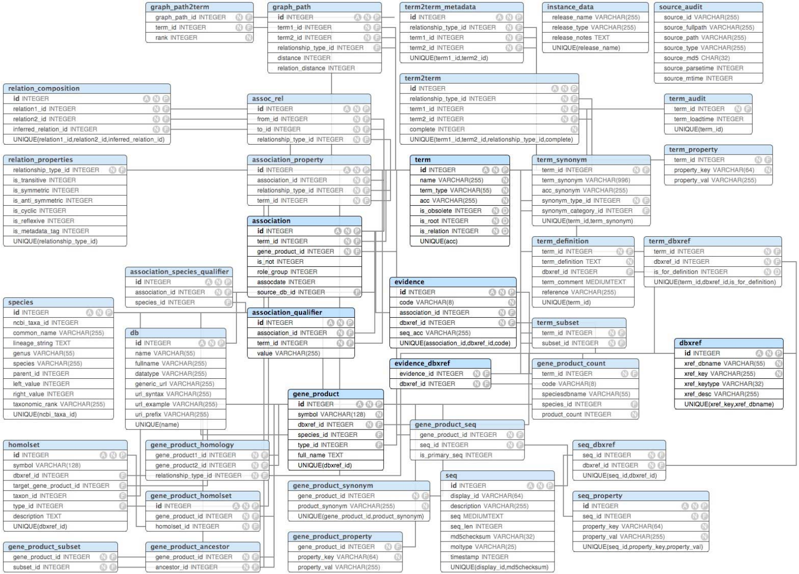
Activity Code	Date	Route No.
24	01/12/01	I-95
24	02/08/01	I-66

Date	Activity Code	Route No.
01/12/01	24	I-95
01/15/01	23	I-495
02/08/01	24	I-66

Tables are split to smallest possible not containing repeated data (normalisation) and relationships are defined among tables to create the relational **schema**...

Relational model

Relational schema



Relational model

Schema drawbacks

- Relational model is not easy to grasp
  $(R \bowtie S) \cup ((R - \pi_{r_1,r_2,\ldots,r_n}(R \bowtie S)) \times (\omega, \ldots \omega))$

Relational model

Schema drawbacks

- Relational model is not easy to grasp
  $(R \bowtie S) \cup ((R - \pi_{r_1, r_2, \ldots, r_n}(R \bowtie S)) \times (\omega, \ldots \omega))$
- It may be hard to transform application data model to a relational schema

Relational model

Schema drawbacks

- Relational model is not easy to grasp
  $(R \bowtie S) \cup ((R - \pi_{r_1, r_2, \ldots, r_n}(R \bowtie S)) \times (\omega, \ldots \omega))$
- It may be hard to transform application data model to a relational schema
- Schema changes are hard to perform

Relational model

Schema drawbacks

- Relational model is not easy to grasp
  $(R \bowtie S) \cup ((R - \pi_{r_1,r_2,\ldots,r_n}(R \bowtie S)) \times (\omega, \ldots \omega))$
- It may be hard to transform application data model to a relational schema
- Schema changes are hard to perform
- It can be difficult (or even impossible) to model some relations to relational model (eg tree structures, transitive closure etc.)

RDBMS

Most popular

Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2

RDBMS

Most popular

Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2

- Great query flexibility

RDBMS

Most popular

Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2

- Great query flexibility
- Data consistency and durability ACID

RDBMS

Most popular

Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2

- Great query flexibility
- Data consistency and durability ACID
- Tables modeled using relational algebra

RDBMS

Most popular

Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2

- Great query flexibility
- Data consistency and durability ACID
- Tables modeled using relational algebra
- Data is normalised (space efficient)

RDBMS drawbacks

RDBMS drawbacks

- Not goot at partitioning

RDBMS drawbacks

- Not goot at partitioning
- Not designed for horizontal scaling

RDBMS drawbacks

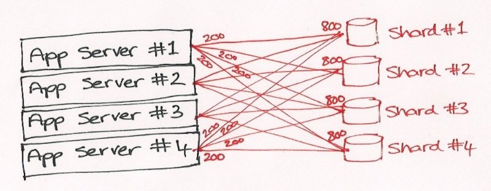
- Not goot at partitioning
- Not designed for horizontal scaling
- Your data may be hard to model in strict relational model, or just not worth it

RDBMS drawbacks

- Not goot at partitioning
- Not designed for horizontal scaling
- Your data may be hard to model in strict relational model, or just not worth it
- You need to store large data blobs

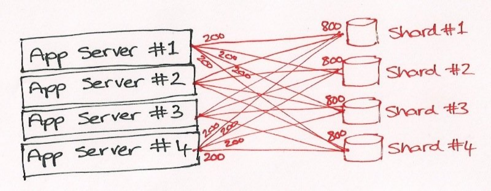
RDBMS scaling

Sharding for partitioning



RDBMS scaling

Sharding for partitioning

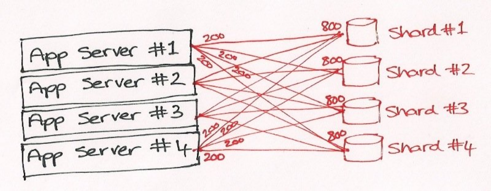


Large Datasets can be sharded for scaling

- Poor man's partitioning

RDBMS scaling

Sharding for partitioning

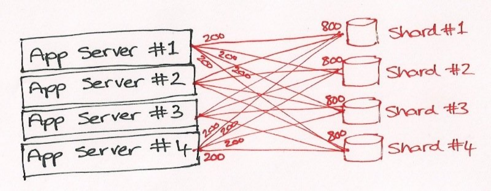


Large Datasets can be sharded for scaling

- Poor man's partitioning
- Relies on application code (bad™)

RDBMS scaling

Sharding for partitioning

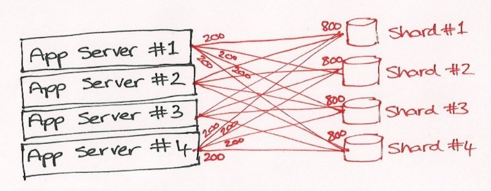


Large Datasets can be sharded for scaling

- Poor man's partitioning
- Relies on application code (bad™)
- May work for small number of tables

RDBMS scaling

Sharding for partitioning

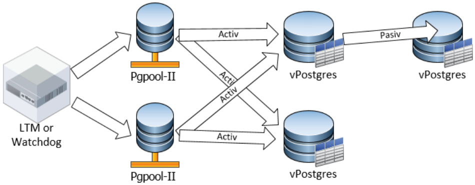


Large Datasets can be sharded for scaling

- Poor man's partitioning
- Relies on application code (bad™)
- May work for small number of tables
- Not easy to add nodes

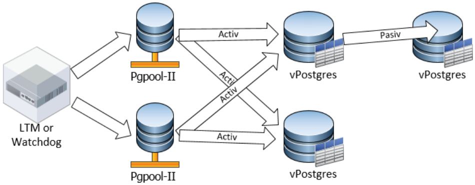
RDBMS scaling

Replication for availability



RDBMS scaling

Replication for availability

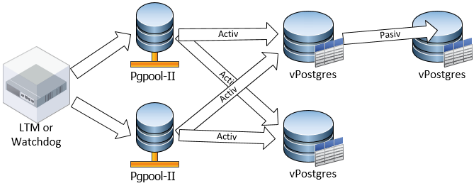


Replication can be used to improve small datasets READ/WRITE performance

- Usually requires 3rd party solutions

RDBMS scaling

Replication for availability

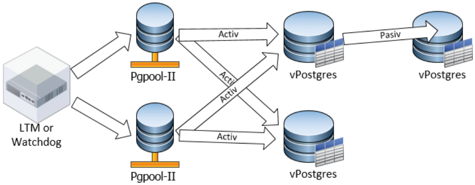


Replication can be used to improve small datasets READ/WRITE performance

- Usually requires 3rd party solutions
- Complicated setup

RDBMS scaling

Replication for availability



Replication can be used to improve small datasets READ/WRITE performance

- Usually requires 3rd party solutions
- Complicated setup
- WRITE has problems (blocking)

RDBMS scaling

Sharding and replication?

Sharding and replication for big production systems has many problems!

- Partitioning is hard, solutions rely on application code (BAD™)

RDBMS scaling

Sharding and replication?

Sharding and replication for big production systems has many problems!

- Partitioning is hard, solutions rely on application code (BAD™)
- IO **will** be a bottleneck no matter what. Multiple caching layers need to be designed.

RDBMS scaling

Sharding and replication?

Sharding and replication for big production systems has many problems!

- Partitioning is hard, solutions rely on application code (BAD™)
- IO **will** be a bottleneck no matter what. Multiple caching layers need to be designed.
- Architecture becomes really complicated. Hard to manage.

RDBMS scaling

Sharding and replication?

Sharding and replication for big production systems has many problems!

- Partitioning is hard, solutions rely on application code (BAD™)
- IO **will** be a bottleneck no matter what. Multiple caching layers need to be designed.
- Architecture becomes really complicated. Hard to manage.
- Not part of DBMS. Need to rely on heterogeneous technologies.

RDBMS scaling

Sharding and replication?

Sharding and replication for big production systems has many problems!

- Partitioning is hard, solutions rely on application code (BAD™)
- IO **will** be a bottleneck no matter what. Multiple caching layers need to be designed.
- Architecture becomes really complicated. Hard to manage.
- Not part of DBMS. Need to rely on heterogeneous technologies.

RDBMS scaling

Sharding and replication?

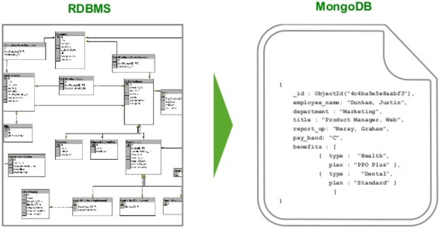
Sharding and replication for big production systems has many problems!

- Partitioning is hard, solutions rely on application code (BAD™)
- IO **will** be a bottleneck no matter what. Multiple caching layers need to be designed.
- Architecture becomes really complicated. Hard to manage.
- Not part of DBMS. Need to rely on heterogeneous technologies.

NoSQL can help

Layout

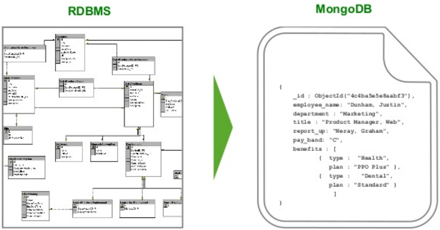
Document



Most popular

MongoDB, CouchDB

Document



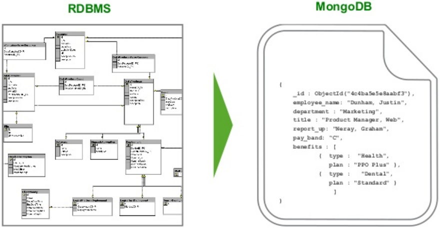
Most popular

MongoDB, CouchDB

Θετικά

- Horizontal Scaling and performance
- Hierarchical data
- Varied data (no schema)
- Web friendly (RESTfull, Javascript)
- Suitable for large data blobs

Document



Most popular

MongoDB, CouchDB

Θετικά

- Horizontal Scaling and performance
- Hierarchical data
- Varied data (no schema)
- Web friendly (RESTfull, Javascript)
- Suitable for large data blobs

Αρνητικά

- No Transactions
- Eventual consistency
- Varied data (no schema)
- Denormalised data
- Needs careful document modeling (for big scale)

Mongo Document Example

```
{
   "_id" : ObjectId("54c955492b7c8eb21818bd09"),
   "address" : {
      "street" : "2 Avenue",
      "zipcode" : "10075",
      "building" : "1480",
      "coord" : [ -73.9557413, 40.7720266 ]
   },
   "borough" : "Manhattan",
   "cuisine" : "Italian",
   "grades" : [
      {
         "date" : ISODate("2014-10-01T00:00:00Z"),
         "grade" : "A",
         "score" : 11
      },
      {
         "date" : ISODate("2014-01-16T00:00:00Z"),
         "grade" : "B",
         "score" : 17
      }
   ],
   "name" : "Vella",
   "restaurant_id" : "41704620"
}
```

Mongo INSERT

```
> db.towns.insert({
  name: "New York",
  population: 22200000,
  last_census: ISODate("2009-07-31"),
  famous_for: [ "statue of liberty", "food" ],
  mayor : {
    name : "Michael Bloomberg",
    party : "I"
  }
})
```

Mongo SELECT

```
> db.towns.find()
{
 "_id" : ObjectId("4d0ad975bb30773266f39fe3"),
 "name" : "New York",
 "population": 22200000,
 "last_census": "Fri Jul 31 2009 00:00:00 GMT-0700 (PDT)",
 "famous_for" : [ "statue of liberty", "food" ],
 "mayor" : { "name" : "Michael Bloomberg", "party" : "I" }
}
```

CouchDB SELECT *

```
# curl http://localhost:5984/music/
```


```
{
  "db_name":"music",
  "doc_count":1,
  "doc_del_count":0,
  "update_seq":4,
  "purge_seq":0,
  "compact_running":false,
  "disk_size":16473,
  "instance_start_time":"1326845777510067",
  "disk_format_version":5,
  "committed_update_seq":4
}
```

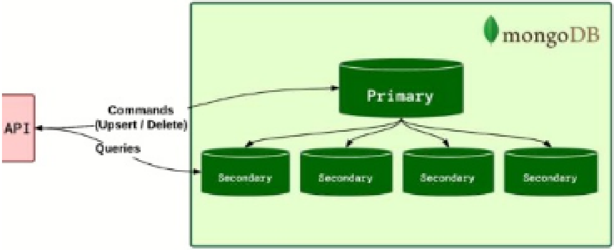
CouchDB SELECT

```
# curl http://localhost:5984/music/74c7a8d2a8548c8b97da748
{
  "_id":"74c7a8d2a8548c8b97da748f43000ac4",
  "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",
  "name":"The Beatles",
  "albums": [
    {
      "title":"Help!",
      "year":1965
    },{
      "title":"Sgt. Pepper's Lonely Hearts Club Band",
      "year":1967
    },{
      "title":"Abbey Road",
      "year":1969
    }
  ]
}
```

CouchDB INSERT

```
# curl -i -X POST "http://localhost:5984/music/" \
  -H "Content-Type: application/json" \
  -d '{ "name": "Wings" }'

HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Location: http://localhost:5984/music/74c7a8d2a8548c8b97da
Date: Wed, 18 Jan 2012 00:37:51 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate
{
  "ok":true,
  "id":"74c7a8d2a8548c8b97da748f43000f1b",
  "rev":"1-2fe1dd1911153eb9df8460747dfe75a0"
}
```

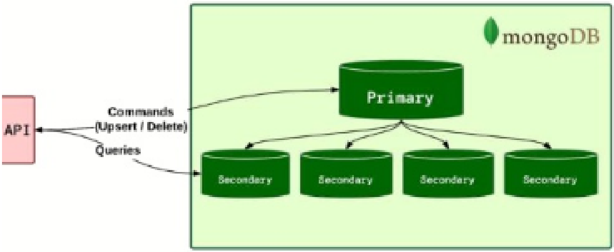
Mongo scaling with replica sets



Characteristics

- Built in the core – API agnostic

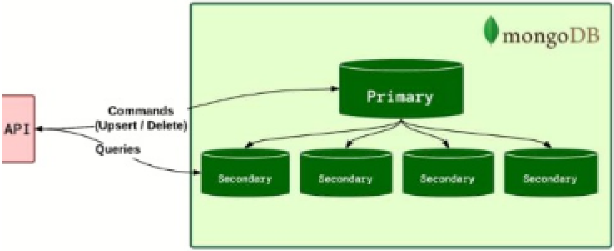
Mongo scaling with replica sets



Characteristics

- Built in the core – API agnostic
- Redundancy — auto-failover

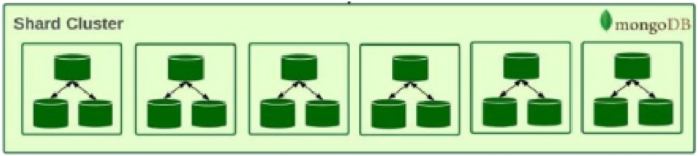
Mongo scaling with replica sets



Characteristics

- Built in the core – API agnostic
- Redundancy — auto-failover
- Easy to setup

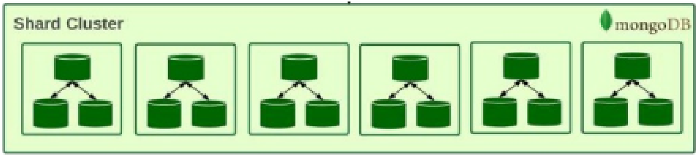
Mongo scaling with sharding



Characteristics

- Each piece of data is exclusively controlled by a single node (shard)

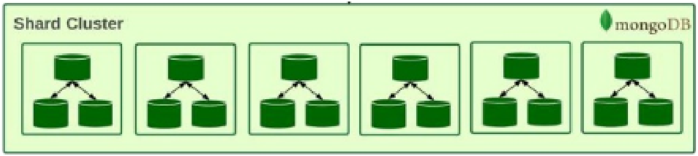
Mongo scaling with sharding



Characteristics

- Each piece of data is exclusively controlled by a single node (shard)
- Each node (shard) has exclusive control over a well defined subset of the data

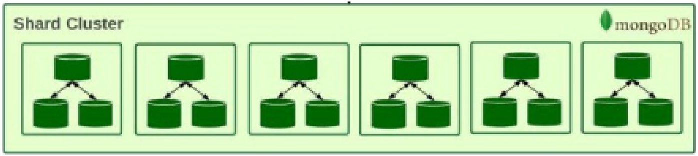
Mongo scaling with sharding



Characteristics

- Each piece of data is exclusively controlled by a single node (shard)
- Each node (shard) has exclusive control over a well defined subset of the data
- As system load changes, assignment of data to shards is **rebalanced** automatically

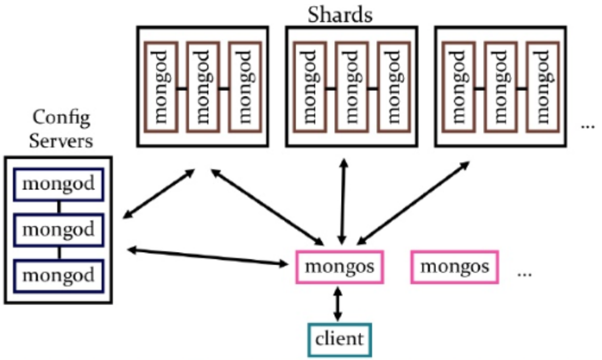
Mongo scaling with sharding



Characteristics

- Each piece of data is exclusively controlled by a single node (shard)
- Each node (shard) has exclusive control over a well defined subset of the data
- As system load changes, assignment of data to shards is **rebalanced** automatically
- Can be combined with replica set for per-shard redundancy

Mongo system architecture



Combining sharding and replication for complicate architectures.
All nodes are vanilla Mongos.

Mongo scaling caveats

scaling caveats

- Relatively easy to setup and only using one type of software. No need for third party solutions.

Mongo scaling caveats

scaling caveats

- Relatively easy to setup and only using one type of software. No need for third party solutions.
- While schema-free design is flexible and some times can model the application data model better that the relational model, at scale you need to carefully model your Documents to not end up with great performance loss.

Mongo scaling caveats

scaling caveats

- Relatively easy to setup and only using one type of software. No need for third party solutions.
- While schema-free design is flexible and some times can model the application data model better that the relational model, at scale you need to carefully model your Documents to not end up with great performance loss.
- If sharding is used, a well thought sharding key must be be used from the beginning since it is very difficult to change.

Mongo scaling caveats

scaling caveats

- Relatively easy to setup and only using one type of software. No need for third party solutions.
- While schema-free design is flexible and some times can model the application data model better that the relational model, at scale you need to carefully model your Documents to not end up with great performance loss.
- If sharding is used, a well thought sharding key must be be used from the beginning since it is very difficult to change.
- For production systems, the minimum nodes required for a scalable architecture is 9. Are your database needs that large?

Layout

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Θετικά

- Horizontal Scaling

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Θετικά

- Horizontal Scaling
- High Availability

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Θετικά

- Horizontal Scaling
- High Availability
- Extremely fast by design

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Θετικά

- Horizontal Scaling
- High Availability
- Extremely fast by design
- Usually in memory

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Θετικά

- Horizontal Scaling
- High Availability
- Extremely fast by design
- Usually in memory

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Θετικά

- Horizontal Scaling
- High Availability
- Extremely fast by design
- Usually in memory

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Αρνητικά

- Lack of schema

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Θετικά

- Horizontal Scaling
- High Availability
- Extremely fast by design
- Usually in memory

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Αρνητικά

- Lack of schema
- Lack of ACID

Keystore

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Most popular

Riak, Redis, memcached

Χαρακτηριστικά

- No Transactions
- No SQL
- No Schema

Θετικά

- Horizontal Scaling
- High Availability
- Extremely fast by design
- Usually in memory

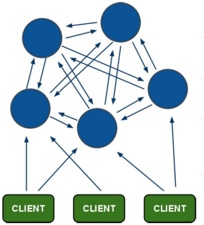
Αρνητικά

- Lack of schema
- Lack of ACID
- Lack of robust querying, only basic CRUD

Keystore

Uses

- Typical key value store
- High speed and in-memory storage suitable for caching applications
- Implementing application queues
- Can scale in a cluster



Layout

Column

Most popular

HBase, Cassandra, Hypertable

Θετικά

- Horizontal Scaling
- Suitable for big data
- Usually build-in support for versioning and compression

Column

Most popular

HBase, Cassandra, Hypertable

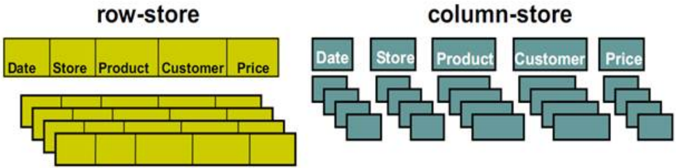
Θετικά

- Horizontal Scaling
- Suitable for big data
- Usually build-in support for versioning and compression

Αρνητικά

- schema based on how data is queried (know how data is to be used)
- Only suitable for large setups (minimum 5 nodes)

Column based characteristics



Column

- Data is stored at low level to be accessed efficiently as rows.
- Row based access needs to access much more data to get a column.
- Column-oriented storage makes sense for certain problems (eg range aggregates, analytic queries)
- Analytic queries can run orders of magnitudes quicker

Layout

Graph

Most popular

Neo4J

Graph

Most popular

Neo4J

Θετικά

- OO designs
- Modeling interconnections
- Transactional
- Many language bindings and REST
- High Availability

Graph

Most popular

Neo4J

Θετικά

- OO designs
- Modeling interconnections
- Transactional
- Many language bindings and REST
- High Availability

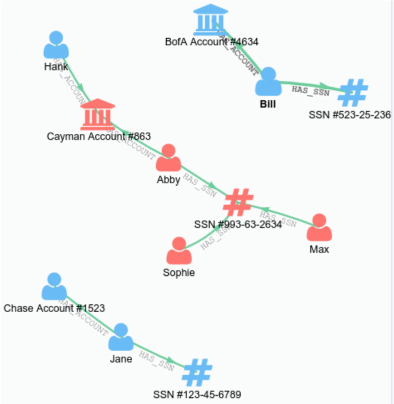
Αρνητικά

- Bad at partitioning (bad scale out)

Neo4J INSERT

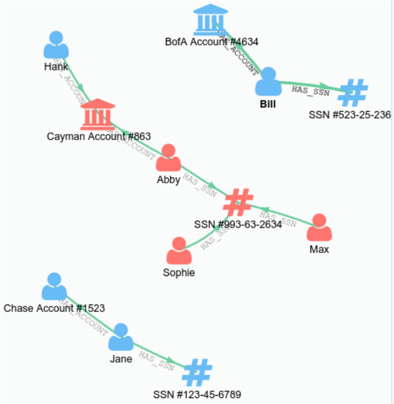
```
driver = GraphDatabase.driver("bolt://localhost", auth=basic_auth("neo4j", "test"))
session = driver.session()

# Insert data
insert_query = '''
CREATE (hank:Person {name:"Hank"}),
(abby:Person {name:"Abby"}),
(max:Person {name:"Max"}),
(sophie:Person {name: "Sophie"}),
(jane:Person {name: "Jane"}),
(bill:Person {name: "Bill"}),
(ssn993632634:SSN {number: 993632634}),
(ssn123456789:SSN {number: 123456789}),
(ssn523252364:SSN {number: 523252364}),
(chase:Account {bank: "Chase", number: 1523}),
(bofa:Account {bank: "Bank of America", number: 4634}),
(cayman:Account {bank: "Cayman", number: 863}),
(bill)-[:HAS_SSN]->(ssn523252364),
(bill)-[:HAS_ACCOUNT]->(bofa),
(jane)-[:HAS_SSN]->(ssn123456789),
(jane)-[:HAS_ACCOUNT]->(chase),
(hank)-[:HAS_ACCOUNT]->(cayman),
(abby)-[:HAS_ACCOUNT]->(cayman),
(abby)-[:HAS_SSN]->(ssn993632634),
(sophie)-[:HAS_SSN]->(ssn993632634),
(max)-[:HAS_SSN]->(ssn993632634)
'''

session.run(insert_query)
```

Transitive Closure



Given suspicions about Hank, find related information to investigate.

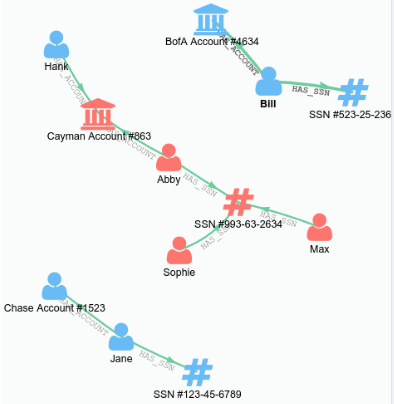
Transitive Closure



Given suspicions about Hank, find related information to investigate.

```
MATCH
  (n:Person)-[*]-(o)
WHERE
  n.name = "Hank"
RETURN
  o
```

Transitive Closure



Given suspicions about Hank, find related information to investigate.

```
MATCH
   (n:Person)-[*]-(o)
WHERE
   n.name = "Hank"
RETURN
   o
```

Hank shares an account with Abby. Abby shares a SSN with Sophie and Max. Given that we suspect Hank may be involved in fraudulent activity, we can flag the Cayman account, Sophie, and Abby as possible fraudulent entities.

Neo4J Transitive Closure

```python
query = '''
MATCH (n:Person)-[*]-(o)
WHERE n.name = {name}
RETURN DISTINCT o AS other
'''

results = session.run(query, parameters={"name": "Hank"})
for record in results:
    print(record["other"])
```

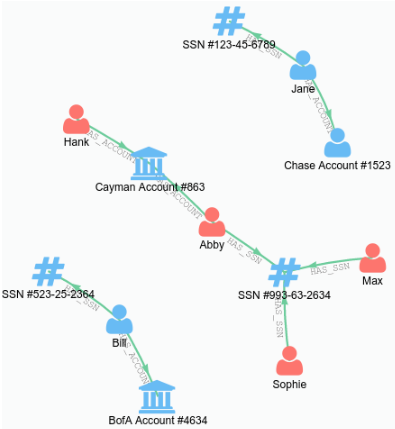
Neo4J Transitive Closure

```
query = '''
MATCH (n:Person)-[*]-(o)
WHERE n.name = {name}
RETURN DISTINCT o AS other
'''

results = session.run(query, parameters={"name": "Hank"})
for record in results:
    print(record["other"])
```

output:

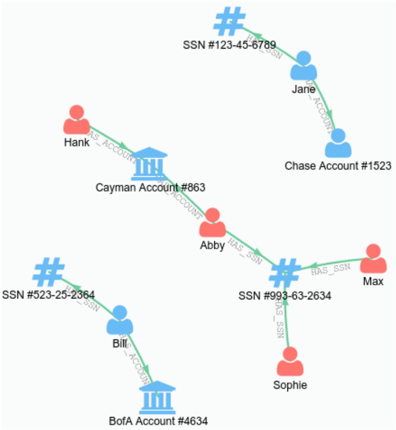
```
<Node id=95 labels=set([u'Account']) properties={u'number': 863, u'bank': u'Cayman'}>
<Node id=85 labels=set([u'Person']) properties={u'name': u'Abby'}>
<Node id=90 labels=set([u'SSN']) properties={u'number': 993632634}>
<Node id=87 labels=set([u'Person']) properties={u'name': u'Sophie'}>
<Node id=86 labels=set([u'Person']) properties={u'name': u'Max'}>
```

Investigation Targeting



Fraud rings often share fraudulent identifying information. Any person with connections to more than two entities in the graph are suspicious. Find large cliques to investigate further.

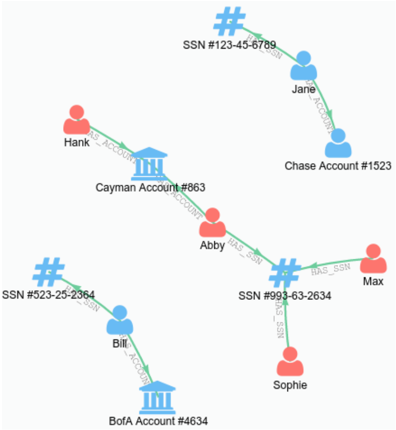
Investigation Targeting



Fraud rings often share fraudulent identifying information. Any person with connections to more than two entities in the graph are suspicious. Find large cliques to investigate further.

```
MATCH
  (n:Person)-[*]-(o)
WITH
  n,
  count(DISTINCT o) AS size
WHERE
  size > 2
RETURN
  n
```

Investigation Targeting



Fraud rings often share fraudulent identifying information. Any person with connections to more than two entities in the graph are suspicious. Find large cliques to investigate further.

```
MATCH
   (n:Person)-[*]-(o)
WITH
   n,
  count(DISTINCT o) AS size
WHERE
   size > 2
RETURN
   n
```

Sophie, Max and Abby all share a SSN, which is suspicious. Hank is also suspicious because he is sharing an account with Abby.

Neo4J Investigation Targeting

```
targeting_query = """
MATCH (n:Person)-[*]-(o)
WITH n, count(DISTINCT o) AS size
WHERE size > 2
RETURN n
"""

results = session.run(targeting_query)
for record in results:
    print(record["n"])
```

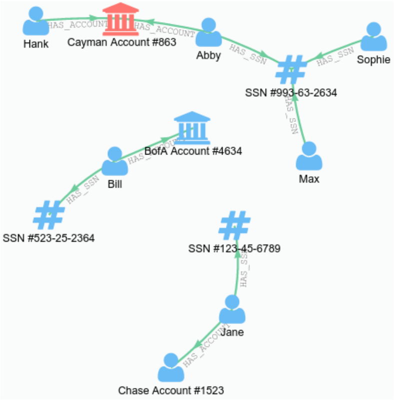
Neo4J Investigation Targeting

```
targeting_query = """
MATCH (n:Person)-[*]-(o)
WITH n, count(DISTINCT o) AS size
WHERE size > 2
RETURN n
"""

results = session.run(targeting_query)
for record in results:
    print(record["n"])
```

output:

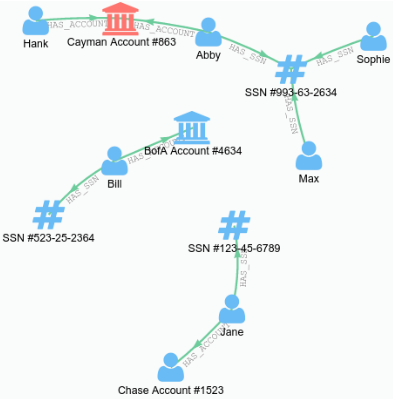
```
<Node id=86 labels=set([u'Person']) properties={u'name': u'Max'}>
<Node id=85 labels=set([u'Person']) properties={u'name': u'Abby'}>
<Node id=87 labels=set([u'Person']) properties={u'name': u'Sophie'}>
<Node id=84 labels=set([u'Person']) properties={u'name': u'Hank'}>
```

Fast Insights



Given that we've identified SSN 993-63-2634 as suspcious, find all associated accounts.

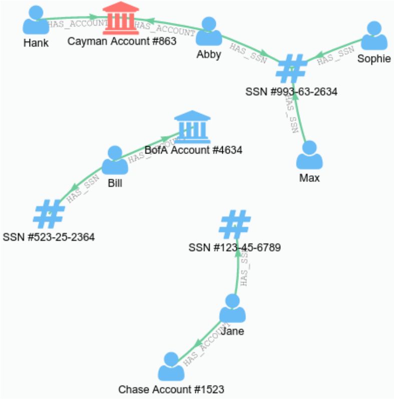
Fast Insights



Given that we've identified SSN 993-63-2634 as suspcious, find all associated accounts.

```
MATCH
  (ssn:SSN)<-[:HAS_SSN]-(:Person)-
  [:HAS_ACCOUNT]->(acct:Account)
WHERE
  ssn.number = 993632634
RETURN
  acct
```

Fast Insights



Given that we've identified SSN 993-63-2634 as suspcious, find all associated accounts.

```
MATCH
  (ssn:SSN)<-[:HAS_SSN]-(:Person)-
  [:HAS_ACCOUNT]->(acct:Account)
WHERE
  ssn.number = 993632634
RETURN
  acct
```

We see that the Cayman account #863 is the only account where a Person using this SSN owns the account.

Neo4J Fast Insights

```
query = """
MATCH (ssn:SSN)<-[:HAS_SSN]-(:Person)-[:HAS_ACCOUNT]->(acct
WHERE ssn.number = {flagged_ssn}
RETURN acct
"""

results = session.run(query, parameters={"flagged_ssn": 99
for record in results:
    print(record["acct"])
```

Neo4J Fast Insights

```
query = """
MATCH (ssn:SSN)<-[:HAS_SSN]-(:Person)-[:HAS_ACCOUNT]->(acc
WHERE ssn.number = {flagged_ssn}
RETURN acct
"""

results = session.run(query, parameters={"flagged_ssn": 99
for record in results:
    print(record["acct"])
```

output:

```
<Node id=95 labels=set([u'Account']) properties={u'number': 863, u'bank': u'Cayman'}>
```

Layout

Wikipedia

Μέγεθος

Design Desisions

- 99-percent availability
- Geographic distribution
- Relaxed policy on security and data Transactions

Wikipedia

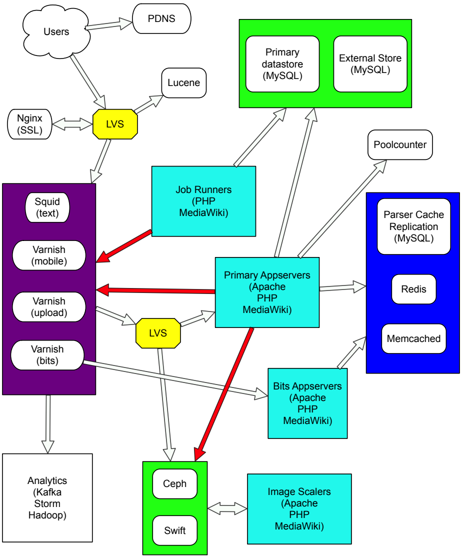
Μέγεθος

Design Desisions

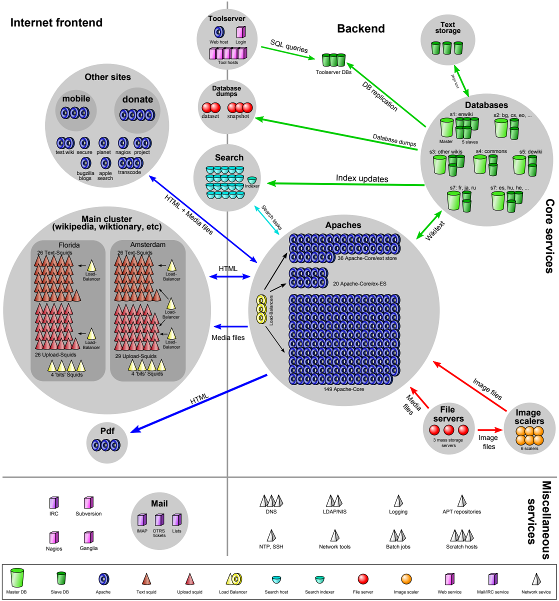
- 99-percent availability
- Geographic distribution
- Relaxed policy on security and data Transactions

- 50,000 http requests per second
- 80,000 SQL queries per second
- 7 million registered users
- 18 million page objects in the English version
- 250 million page links
- 220 million revisions
- 1.5 terabytes of compressed data

Wikipedia (Architecture)



Wikipedia (Server structure)



Layout

Twitter

Multiple databases

Twitter

Multiple databases

MySQL primary storage of Tweets and Users (custom fork)

Twitter

Multiple databases

MySQL primary storage of Tweets and Users (custom fork)

FlockDB in-house graph database, store social graph information (following, etc)

Twitter

Multiple databases

MySQL primary storage of Tweets and Users (custom fork)

FlockDB in-house graph database, store social graph information (following, etc)

Memcached ”heavily modified” fork of Memcached 1.4.4

Twitter

Multiple databases

MySQL primary storage of Tweets and Users (custom fork)

FlockDB in-house graph database, store social graph information (following, etc)

Memcached ”heavily modified” fork of Memcached 1.4.4

Cassandra services like Snowflake for quickly generating unique identifiers

Twitter

Multiple databases

MySQL primary storage of Tweets and Users (custom fork)

FlockDB in-house graph database, store social graph information (following, etc)

Memcached ”heavily modified” fork of Memcached 1.4.4

Cassandra services like Snowflake for quickly generating unique identifiers

Gizzard in-house scala framework for building custom distributed databases (with arbitrary storage technology)

Twitter

Multiple databases

MySQL	primary storage of Tweets and Users (custom fork)
FlockDB	in-house graph database, store social graph information (following, etc)
Memcached	”heavily modified” fork of Memcached 1.4.4
Cassandra	services like Snowflake for quickly generating unique identifiers
Gizzard	in-house scala framework for building custom distributed databases (with arbitrary storage technology)
Apache Lucene	search index

Twitter

Multiple databases

MySQL primary storage of Tweets and Users (custom fork)

FlockDB in-house graph database, store social graph information (following, etc)

Memcached ”heavily modified” fork of Memcached 1.4.4

Cassandra services like Snowflake for quickly generating unique identifiers

Gizzard in-house scala framework for building custom distributed databases (with arbitrary storage technology)

Apache Lucene search index

HBase and Hadoop analytics running over data every day

Twitter

Multiple databases

MySQL	primary storage of Tweets and Users (custom fork)
FlockDB	in-house graph database, store social graph information (following, etc)
Memcached	”heavily modified” fork of Memcached 1.4.4
Cassandra	services like Snowflake for quickly generating unique identifiers
Gizzard	in-house scala framework for building custom distributed databases (with arbitrary storage technology)
Apache Lucene	search index
HBase and Hadoop	analytics running over data every day
Redis	experimental timeline storage technology

Layout

Facebook

Size

- >200M active users
- >100M users log on to Facebook at least once each day
- >30M users update their statuses at least once each day

Facebook

Size

- >200M active users
- >100M users log on to Facebook at least once each day
- >30M users update their statuses at least once each day

Databases

MySQL primary storage (4000 shards, 9000 mencached)

Facebook

Size

- >200M active users
- >100M users log on to Facebook at least once each day
- >30M users update their statuses at least once each day

Databases

MySQL primary storage (4000 shards, 9000 mencached)

Hive simple summarization jobs, business intelligence, machine learning etc.

Facebook

Size

- >200M active users
- >100M users log on to Facebook at least once each day
- >30M users update their statuses at least once each day

Databases

MySQL primary storage (4000 shards, 9000 mencached)

Hive simple summarization jobs, business intelligence, machine learning etc.

Cassandra Currently used for Facebook's private messaging.

Facebook

Size

- >200M active users
- >100M users log on to Facebook at least once each day
- >30M users update their statuses at least once each day

Databases

MySQL primary storage (4000 shards, 9000 mencached)

Hive simple summarization jobs, business intelligence, machine learning etc.

Cassandra Currently used for Facebook's private messaging.

Layout

What to use

Question:

Which **one** DB technology should I use?

What to use

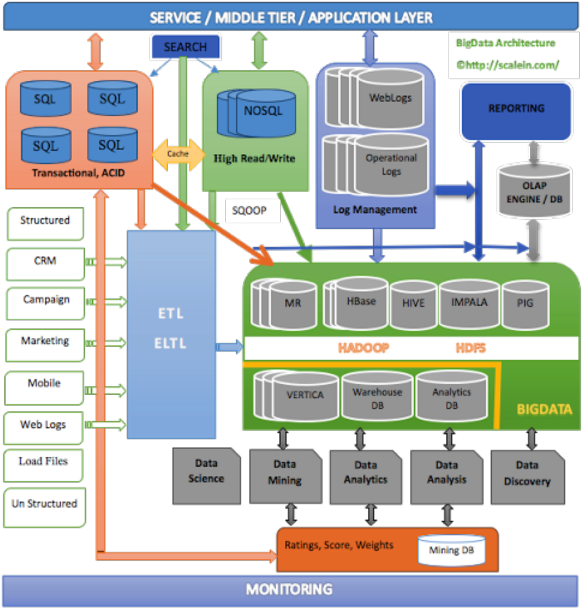
Question:

Which **one** DB technology should I use?

Answer:

Why use **one**? Depending on your scale you may need to use all of them!

Typical BigData Architecture



Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage
- Web friendly

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage
- Web friendly
- Data analysis

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage
- Web friendly
- Data analysis
- Διαθεσιμότητα (9s)

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage
- Web friendly
- Data analysis
- Διαθεσιμότητα (9s)
- Requests per second (concurent users)

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage
- Web friendly
- Data analysis
- Διαθεσιμότητα (9s)
- Requests per second (concurent users)
- Read vs Write percentage

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage
- Web friendly
- Data analysis
- Διαθεσιμότητα (9s)
- Requests per second (concurent users)
- Read vs Write percentage
- Policy on security, transactions (consistency, corruption)

Conclusions

Design requirements

- Query types (flexibility, relations, hierarchies)
- Data form (schema)
- Data size (for sharding etc.)
- Data inter–linkage
- Web friendly
- Data analysis
- Διαθεσιμότητα (9s)
- Requests per second (concurent users)
- Read vs Write percentage
- Policy on security, transactions (consistency, corruption)
- Geographic distribution

Thank you!