Introduction to NoSQL

Omid Vahdaty, Cloud Ninja

What is ACID ?

Atomic - all or nothing. Either query completed or not. (transaction)

Consistency - once transaction committed - that must conform with with the constraints of the schema.

- Requires
  - a **well defined schema** (anti pattern to NoSQL). **Great for READ intensive and AD HOC queries.**
  - **Locking of data until transaction is finished. → performance bottleneck.**
- Consistency uses cases
  - "**Strongly consistent**" : bank transactions must be ACCURATE
  - "**Eventually consistent"**: social media feed updates. (not all users must see same results at the same time.

Isolations - concurrent queries - must run in isolation, this ensuring same result.

Durable - once transaction is executed, data will preserved. E.g. power failure

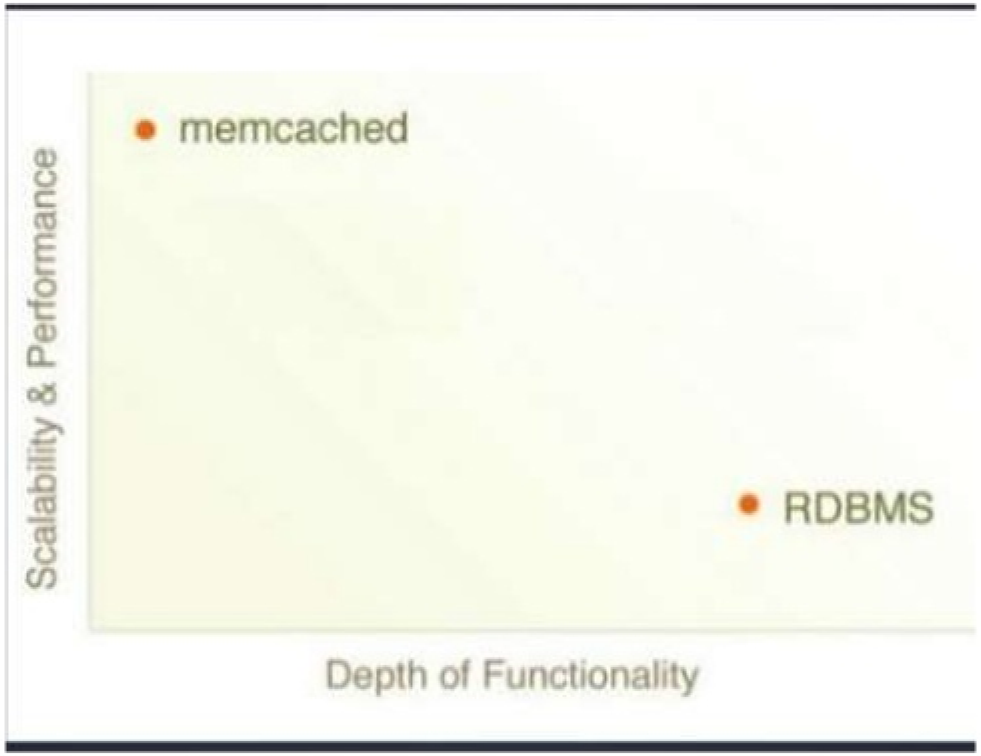
NoSql Advantages

- Advantages?
  a. Non-relational and schema-less data model
  b. Low latency and high performance
  c. Highly scalable
  d. SQL queries are not well suited for the object oriented data structures that are used in most applications now.
  e. Some application operations require multiple and/or very complex queries. In that case, data mapping and query generation complexity raises too much and becomes difficult to maintain on the application side.

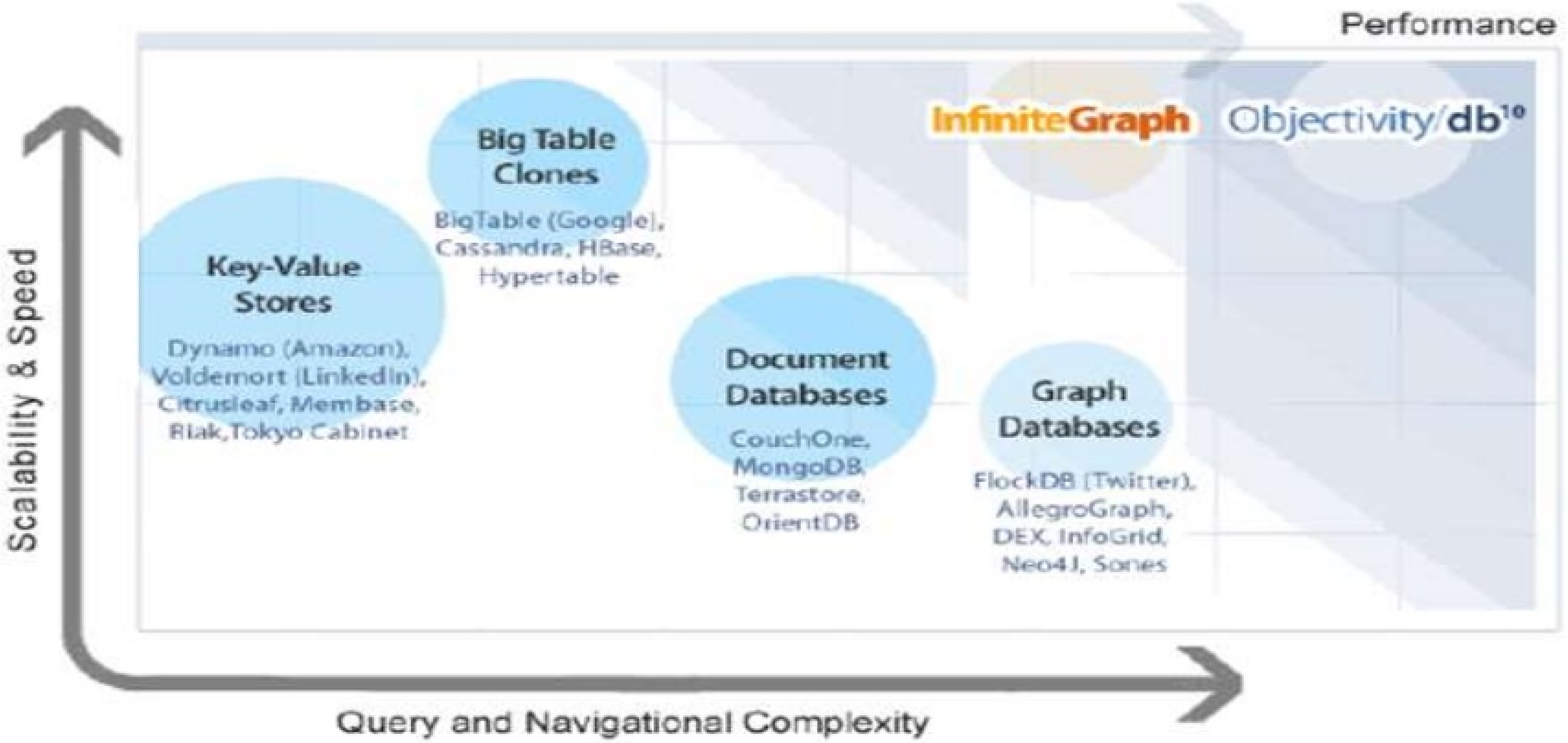
NoSQL DB types

- **Key value store.**
- **Document store** - e.g json. document stores are used for aggregate objects that have no shared complex data between them and to quickly search or filter by some object properties.
- **column family**. These are used for organizing data based on individual columns where actual data is used as a key to refer to whole data collection. This particular approach is used for very large scalable databases to greatly reduce time for searching data
- **Graph databases** map more directly to object oriented programming models and are faster for highly associative data sets and graph queries. Furthermore they typically support **ACID transaction properties in the same way as most RDBMS.**

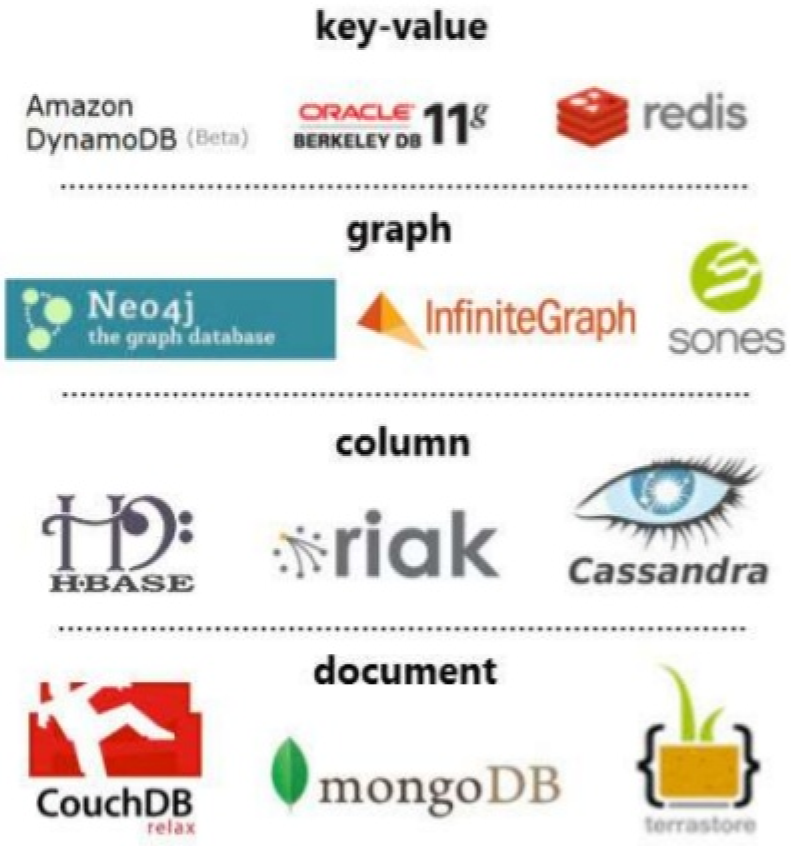
TradeOff?

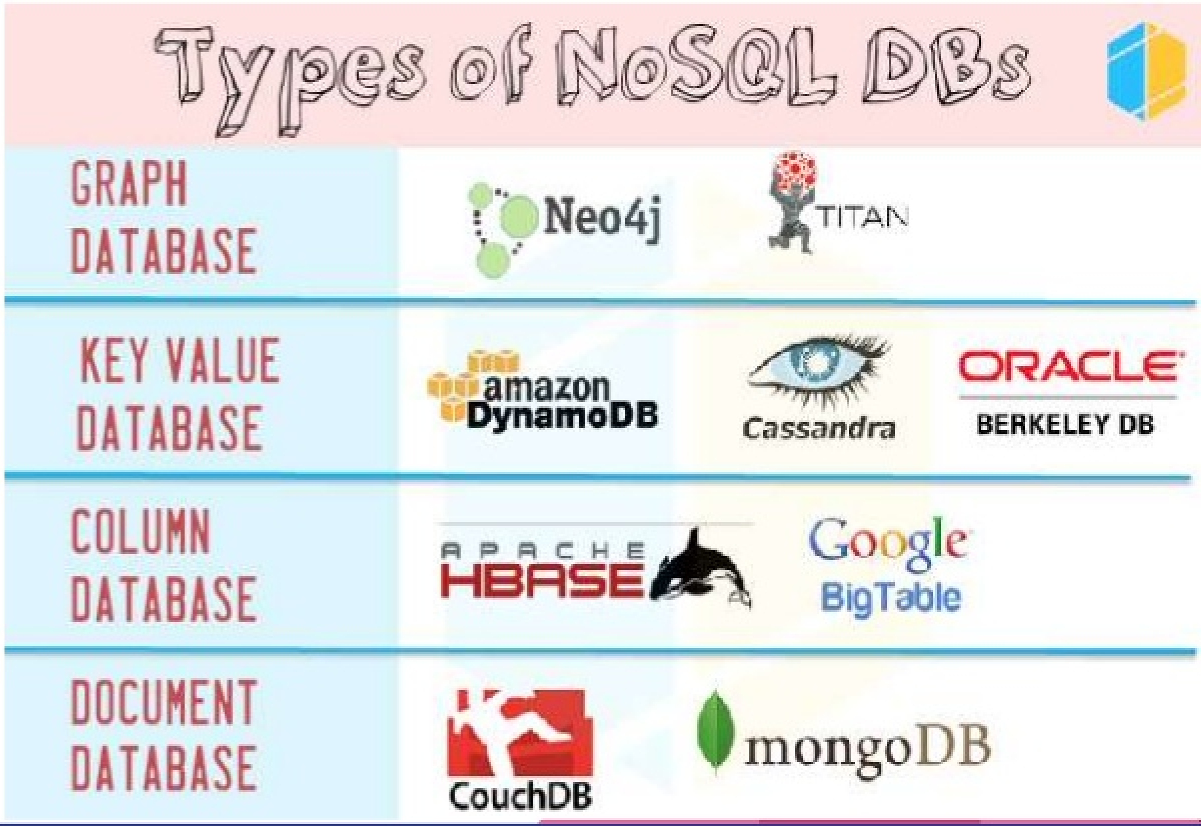

memcached
RDBMS
Scalability & Performance
Depth of Functionality

The "NOSQL" Technology Landscape


Performance
InfiniteGraph
Objectivity/db10
Big Table Clones
BigTable (Google), Cassandra, HBase, Hypertable
Key-Value Stores
Dynamo (Amazon), Voldemort (LinkedIn), Citrusleaf, Membase, Riak, Tokyo Cabinet
Document Databases
CouchOne, MongoDB, Terrastore, OrientDB
Graph Databases
FlockDB (Twitter), AllegroGraph, DEX, InfoGrid, Neo4J, Sones
Scalability & Speed
Query and Navigational Complexity

Brands?


key-value
Amazon DynamoDB (Beta)
ORACLE BERKELEY DB 11g
redis
graph
Neo4j the graph database
InfiniteGraph
sones
column
HBASE
riak
Cassandra
document
CouchDB relax
mongoDB
terrastore


Types of NoSQL DBs
GRAPH DATABASE
Neo4j
TITAN
KEY VALUE DATABASE
amazon DynamoDB
Cassandra
ORACLE BERKELEY DB
COLUMN DATABASE
APACHE HBASE
Google BigTable
DOCUMENT DATABASE
CouchDB
mongoDB

Big Data Landscape

	Key value	document	colunar	graph	OLTP	OLAP	Hadoop
OpenSource	Redis, Riak	Mongo, CouchDB	Cassandra, Hbase	Neo4J	MySQL Postgrtess		hadoop
AWS	Dynamo	DynamoDB			Aurora	Redshift	EMR
GCP	BigTable					BigQuery	
Azure	Tables	Document DB	“Hbase”		Azure SQL		HDInsights

NoSQL uses cases

- Usecases

  a. **Application requiring horizontal scaling : Entity-attribute-value = fetch by distinct value**
     i. Mobile APP with millions of users - for each user: READ/WRITE
     ii. The equivalent in OLTP is sharding - VERY COMPLEX.
  b. **User Preferences, fetch by distinct value**
     i. (structure may change dynamically)
     ii. Per user do something.
  c. **Low latency / session state**
     i. Ad tech , and large scale web servers- capturing cookie state
     ii. Gaming - game state
  d. **Thousand of requests per seconds (WRITE only)**
     i. Reality - voting
     ii. Application monitoring and IoT- logs and json.

NoSQL AntiPattern

- AntiPatterns:
  - Ad hoc Queries
  - OLAP
  - BLOB

Use cases simplified- DB family

- **Key value** = large amount of data, fast access, simple relations: online shopping carts.
- **Documents DB** = User info , web applications. Each user = document, with attributes. May differ from user to user
- **Column Store** = hbase/cassandra , billion of web pages index.
- **Graph DB** = when complex relationship matters: social networks.

When do you normalize your tables?

- eliminate redundant data, utilize space efficiently
- reduce update errors.
- in situations where we are storing immutable data such as financial transactions or a particular day's price list.

When Not to normalize?

- Not to normalize?
  a. Could you denormalize your schema to create flatter data structures that are easier to scale?
  b. When Multiple Joins are Needed to Produce a Commonly Accessed View
  c. decided to go with database partitioning/sharding then you will likely end up resorting to denormalization
  d. Normalization saves space, but space is cheap!
  e. Normalization simplifies updates, but reads are more common!
  f. Performance, performance, Performance

Document DB

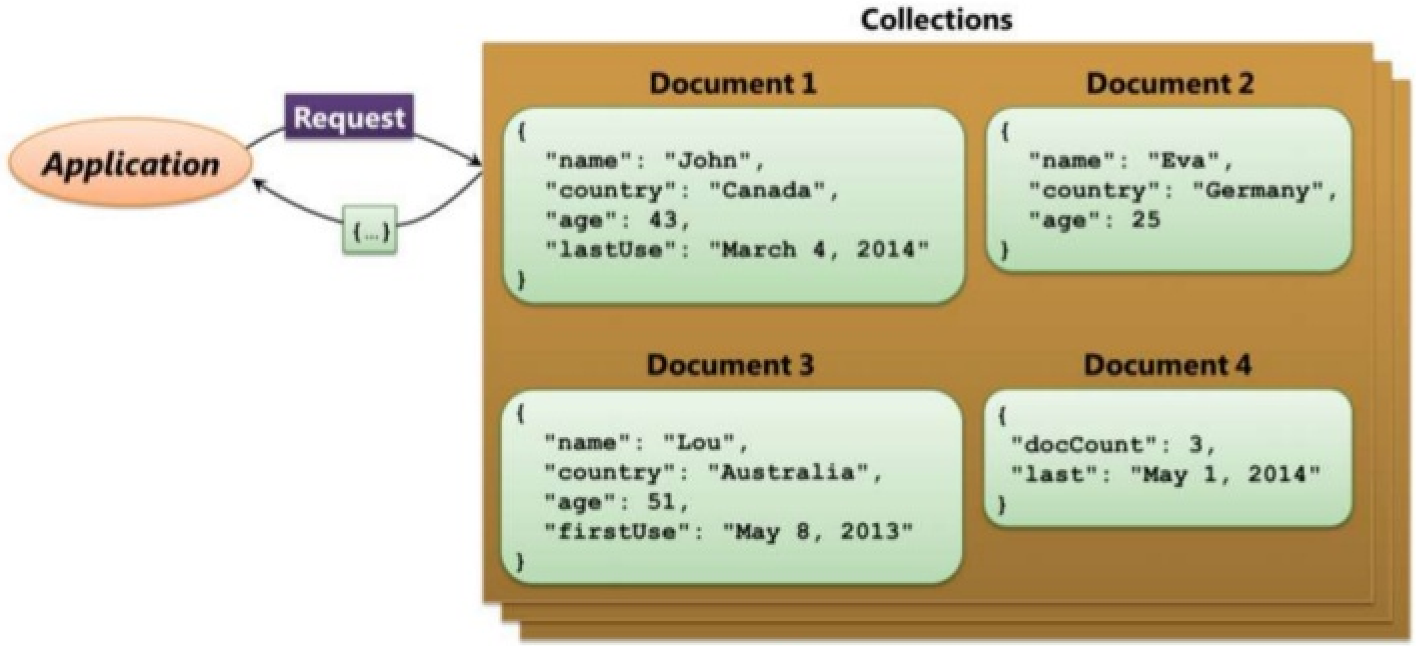
- Database = multiple collections, **schema less**

- A collection of Json files = table

- Notice that unlike a relational table, not all “rows” have a value for all “columns”

- Each json = “row”.

- **Use Case: web**



```
{
  "name": "John",
  "country": "Canada",
  "age": 43,
  "lastUse": "March 4, 2014"
}
```

```
{
  "name": "Eva",
  "country": "Germany",
  "age": 25
}
```

```
{
  "name": "Lou",
  "country": "Australia",
  "age": 51,
  "firstUse": "May 8, 2013"
}
```

```
{
 "docCount": 3,
 "last": "May 1, 2014"
}
```

Document DB:

- Schema free, not relational
- Replications support
- Simple API
- Eventually consistent
- Distributed - scale horizontally (scale out)
- Open source? :)

Document DB: Mongo DB

- Document : Json / Binary Json = Bson
- Schema free
- Performance
  - Indexes
  - No transaction - (has atomic operations)
  - Memory mapped files (delayed writing)
- Scale
  - Auto sharding
  - Replication
- Commercially supported (10gen)

Document DB: Mongo DB

- Querying
  - MapReduce
  - Aggregations
  - Run in parallel on all nodes
- GridFS
  - Ability to store large files easily
- Geospatial Indexing
  - Add location easily
  - Find nearest node
- OS:
  - mac , linux, windows

Document DB: Mongo DB

- Collection = table
- Document = row, not each json is the same (schema less)
- Embedded = think “pre joined relationship”
- References = analog to foreign key, think “relationship”
- Sample Queries
  - db.orders.find ({“state”: “bad”})
  - db.orders.find ({“state_nested.layer2”: “mostly bad”})
  - db .orders.find ({“sub_total”: {$gt : 1999}})
- Indexing:
  - db.orders.ensureIndex ({“state”:”1”})

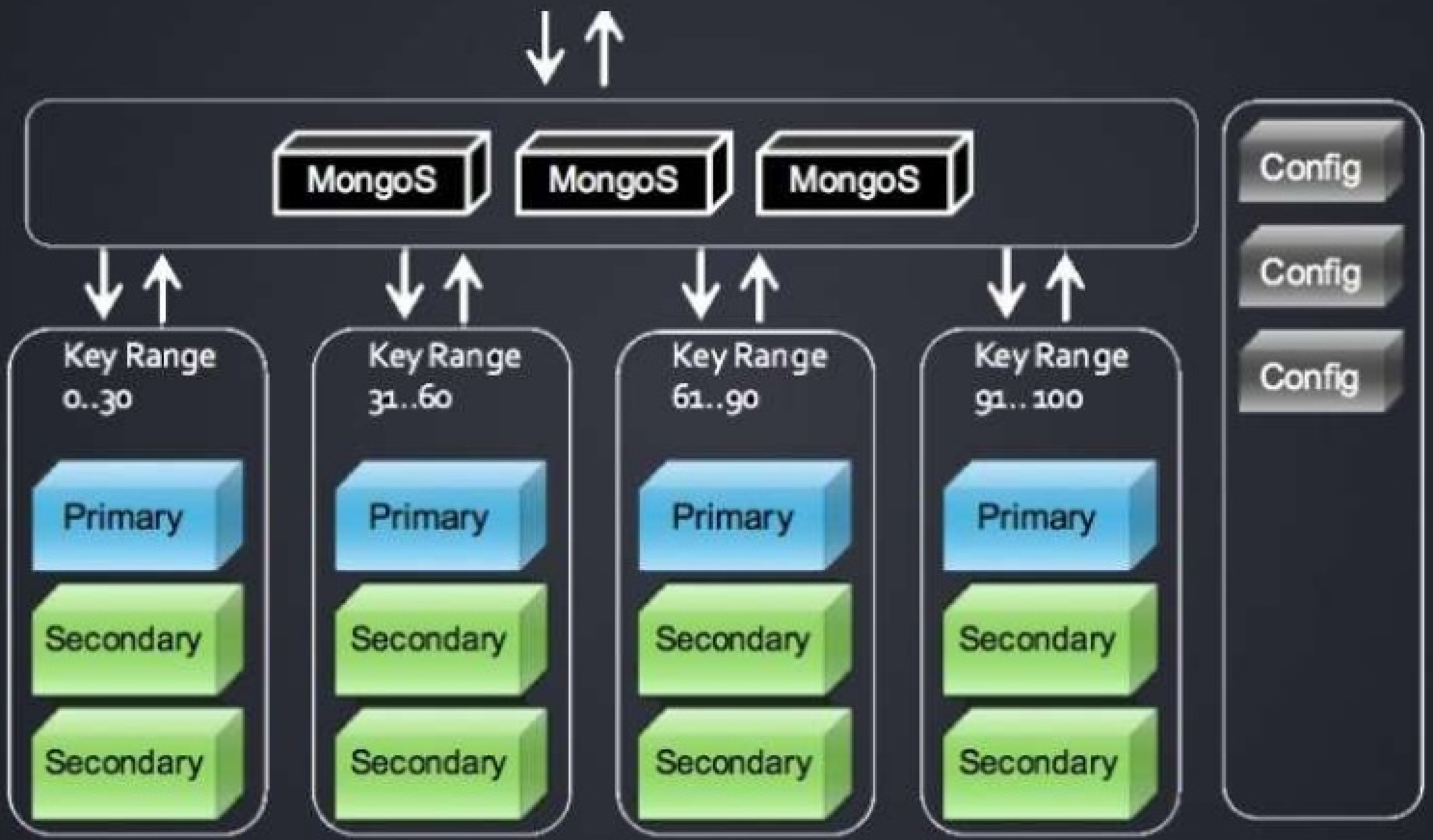
Document DB: Mongo DB

- Inserting
  - db.orders.save({“id”: “abcd” , “state”:”good” “state_nested”: {“state”: “bad” , “frequency”: ”sometimes” } })
- Updating
  - Query/modify/save
    - Query document
    - Save to variable
    - Modify
    - save(variable)
  - Atomic Operations
    - db.orders.update(....)
- Components: Server, Shell, Sharing router, Rest API

Document DB: Mongo DB

- Replication
  - master /slave
  - Master / master (limited, unsafe for concurrent updates)
  - Failover support
  - Rack aware
  - DC aware
- Sharding
  - Define a shard key
  - Auto balance
  - Failover via replicas
  - Query run in parallel

Sharding



MongoS
MongoS
MongoS
Config
Config
Config
Key Range 0..30
Key Range 31..60
Key Range 61..90
Key Range 91..100
Primary
Secondary
Secondary

AWS DynamoDB

- http://cloudacademy.com/blog/amazon-dynamodb-ten-things/
- Is a document DB and key value DB
- **Amazon DynamoDB**
    - **is a managed, NoSQL database service (multi AZ)**
    - **has Predictable Performance**
    - **is designed for massive scalability (auto partition)**
    - **Data types**
        - **Scalar**
        - **Documents: (lists/maps/json like)**
        - **Multi valued: binary. String , int**
    - **Data models: table, items , attributes →**
        - Tables are collections of Items, and Items are collections of Attributes.
        - Attributes : key value
        - Schema less, primary key

AWS DynamoDB

- Primary keys
  - Hash type key
  - Has and range type key
- Indexing
  - Local Secondary Index (LSI) --> upto 10GB, range key is mandatory
  - a Global Secondary Index (GSI). → only eventual consistency, either hash or hash+range. Key will be used for partitions.
- Partitions
  - auto by hash key
  - Logic of partitions
    - Size
    - Throughput,
    - Need to understand formula's of partions for cost...

AWS DynamoDB

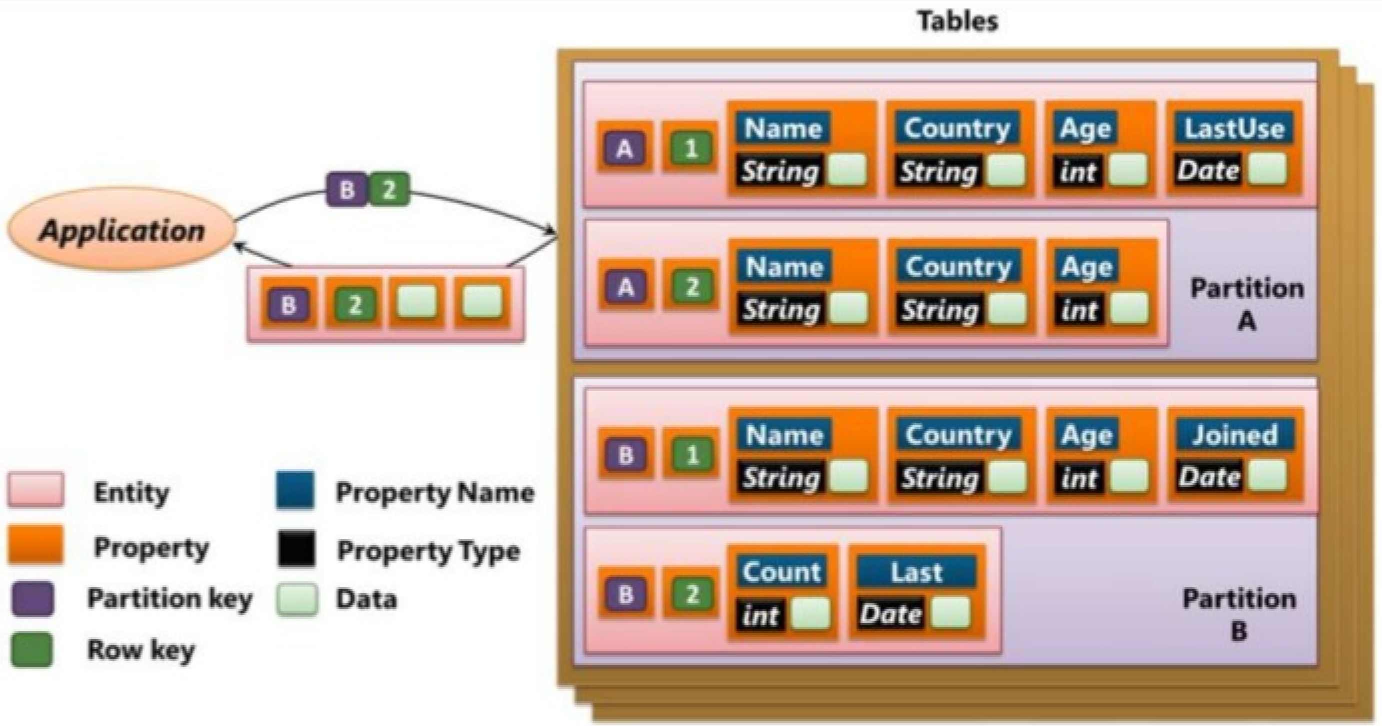
- Read about Dynamo streams
- Native integration with EMR and redshift
- Java script shell CLI for development

Key Value store

- Simple relations
- Very fast
- cheap
- Use case:

Shopping cart

Caching



Tables
Application
B 2
B 2
A 1 Name String Country String Age int LastUse Date
A 2 Name String Country String Age int
Partition A
B 1 Name String Country String Age int Joined Date
B 2 Count int Last Date
Partition B
Entity
Property
Partition key
Row key
Property Name
Property Type
Data

Key Value store

- Most common brands
  - Redis
  - Memcache
  - Riak
  - AeroSpike
  - AWS simpleDB
- Sub categories of KV db: Eventually consistent (DynamoDB), RAM (redis), ordered (memcache, InfiniDB)
- 4 common use cases
  - **Dynamic Attributes**
  - **Calculating distinct values quickly**
  - **Top N calculation**
  - **Cache / Message Queue**

Comparing redis and memcache

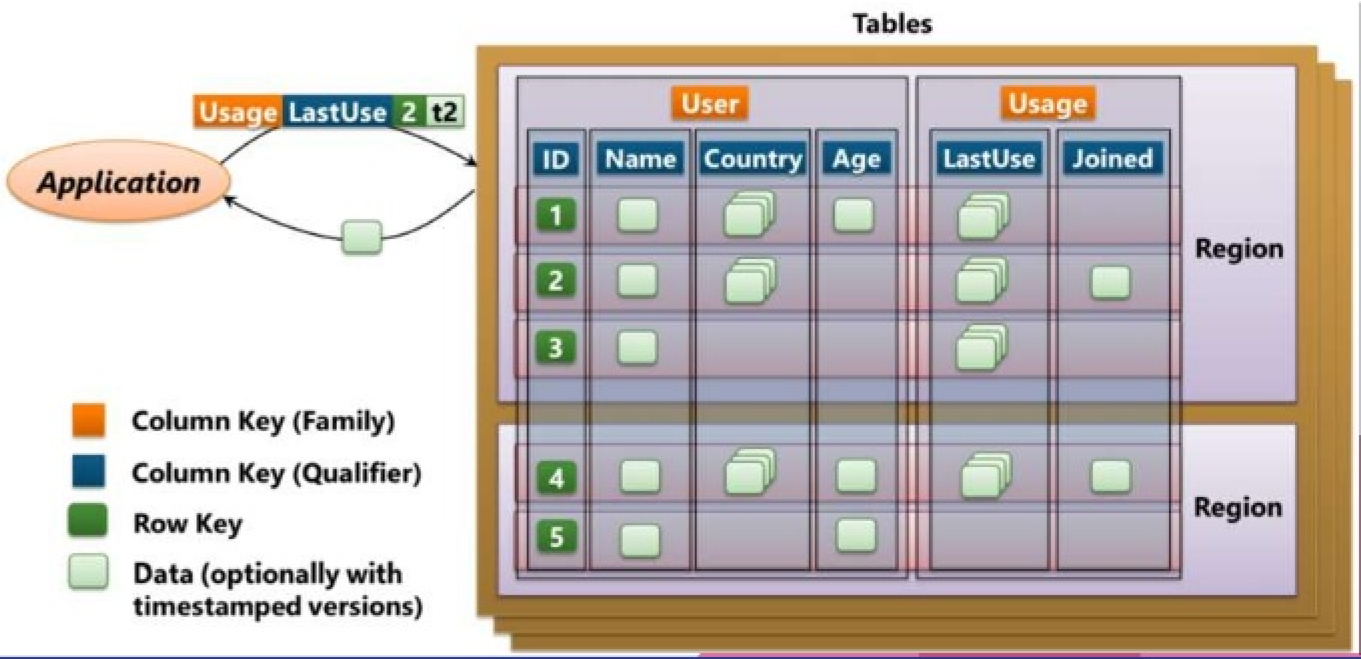
	Redis	memcache
usecase	Database and caching (better for geographic caching/ multi DC caching/ huge caching)	Primarily for caching for large websites. Simple, light, fast.
Distribution	VERY HARD, performance impact. Single threaded.	Very easy. Multi threaded.
Persistence to disk	Yes, every 2 sec.	No.volatile RAM. very costly to restart service
Overload policy	none	LRU ?
Cache consistency	none	Check and set
Data Types	Strings, ints,hashes, binary safe strings, lists, sets, sorted sets.	Strings, ints, arrays require serializations.
Length	Upto 2GB	250 bytes
replication	master/slave	Not supported

Comparing redis and memcache

	Redis	memcache
installation	easy	
Mem consumption	Efficient in hash data types	Efficient in simple key value, less metadata
Load of read /write	Good read write supported. Services reads & writes from DB .(slower)	**heavy read/writes as designed for large websites load ,service read writes from RAM. (not though DB)**
Performance	Slightly faster on small objects. Geo oriented.	Heavy load oriented
TTL (time to leave)	yes.	
Publish , subscribe for messaging	Yes!	

Columnar store

- **The column families in a table must be defined up front, so HBase does have a little schema**, but the columns in a particular column family are not
- Notice that unlike a relational table, not all rows have a value for all columns
- Use case: billions of records, e.g index of all websites pages. With attributes.



Columnar HBASE

- There are no data types, for instance. Every table holds nothing but bytestrings. And each cell can contain multiple time-stamped versions of a value, which is useful when applications might need to access older information

- the rows in a table are broken into regions, each of which can be stored on a different machine. This is how an HBase table scales to hold more data than will fit on a single machine

- HBase automatically partitions data across machines.

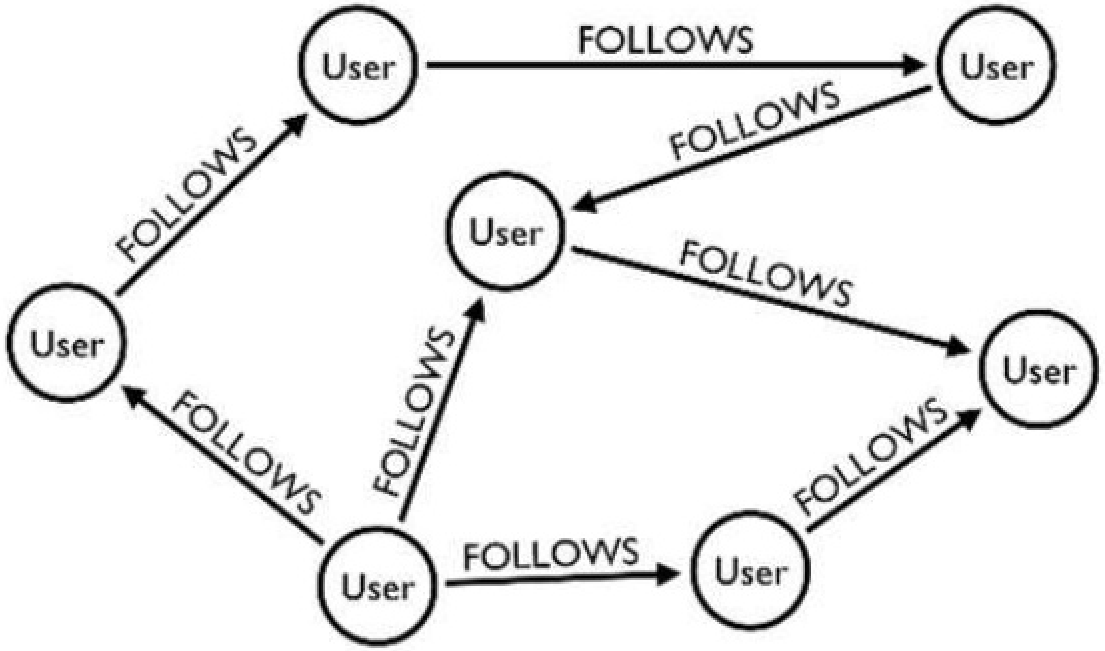
- HBase doesn't provide a query language. Instead, an application can access the value of a particular cell by providing three things: a column family, a column qualifier (i.e., a column name), and a row key. The request can also specify the timestamp for a specific version of a cell's data if desired

- because the rows are sorted by their keys, there's no need for the client to know which region contains a particular row; HBase can figure this out by itself. This approach is similar to a key/value store, and it provides fast access

- doesn't support secondary indexes;

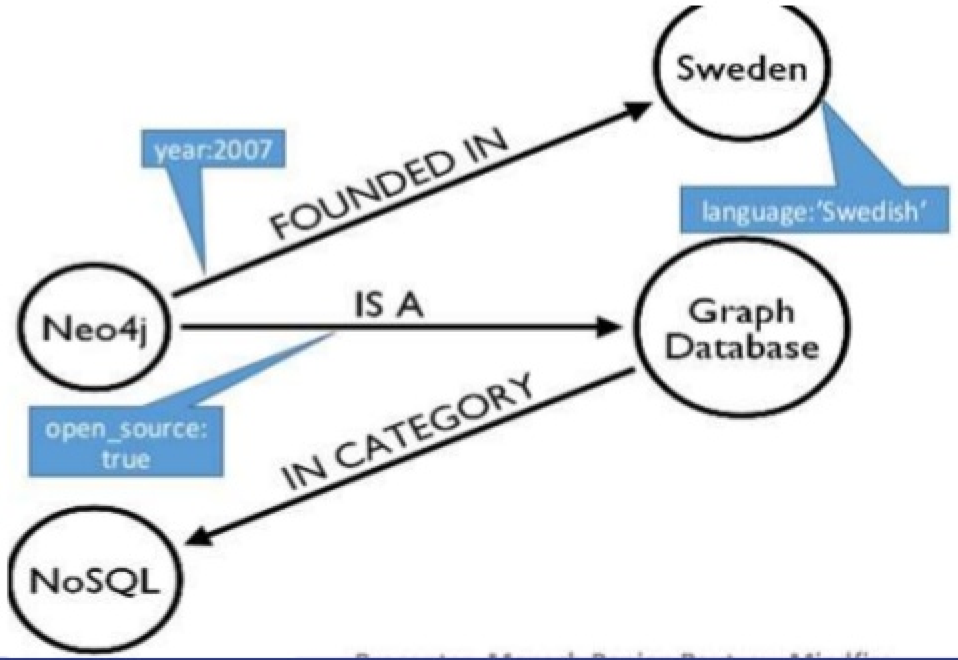
Graph DB

- Represents data as graph
- When relationships are impor
- E.g social graph
- Least common DB family

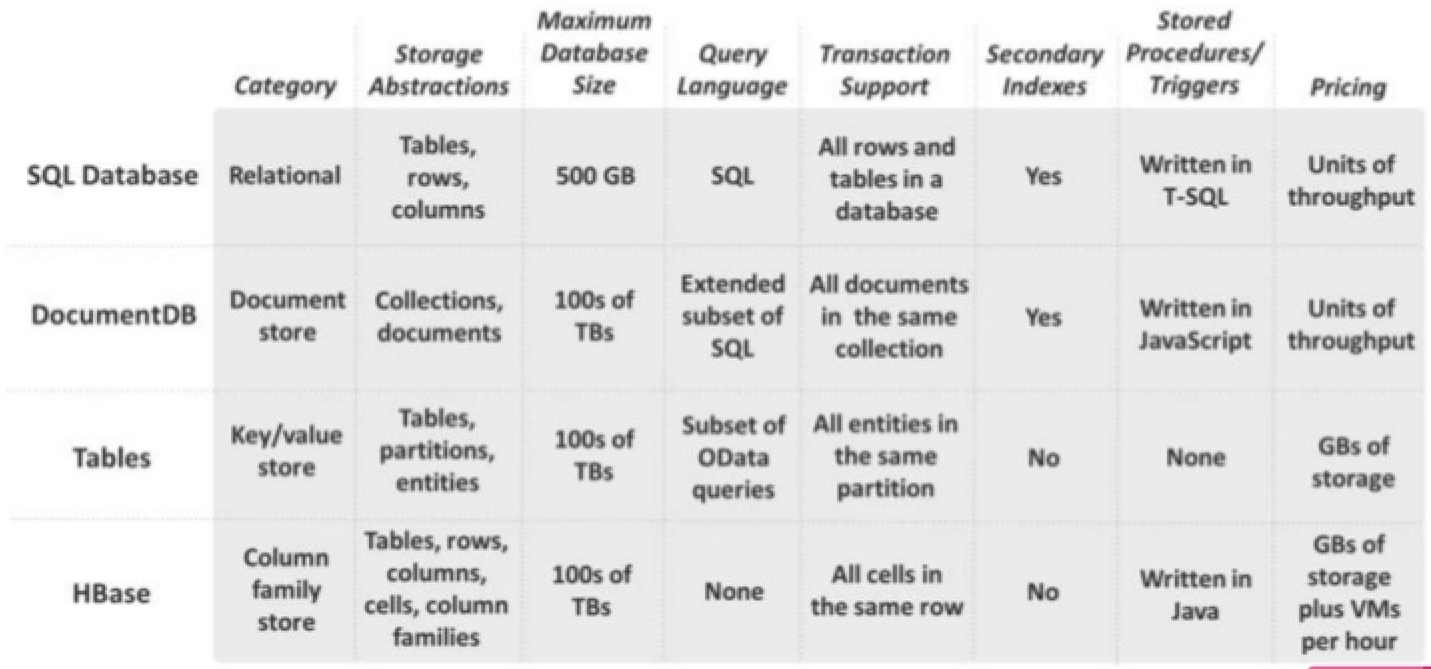


Neo4J

- ACID
- Super fasts on querying connections
- Scale Billions of nodes
- Infinite depth of connections
- 60% faster for friends of friends query
- On friends of friend …. Friend in fifth depth: SQL chokes, neo4j is X2000 faster
- Graph: nodes, relationships, properties
- Cypher Query Language: create, update, remove nodes, relationships, properties



In depth comparison of NoSql DB families



	Category	*Storage Abstractions*	*Maximum Database Size*	*Query Language*	*Transaction Support*	*Secondary Indexes*	*Stored Procedures/ Triggers*	*Pricing*
SQL Database	Relational	Tables, rows, columns	500 GB	SQL	All rows and tables in a database	Yes	Written in T-SQL	Units of throughput
DocumentDB	Document store	Collections, documents	100s of TBs	Extended subset of SQL	All documents in the same collection	Yes	Written in JavaScript	Units of throughput
Tables	Key/value store	Tables, partitions, entities	100s of TBs	Subset of OData queries	All entities in the same partition	No	None	GBs of storage
HBase	Column family store	Tables, rows, columns, cells, column families	100s of TBs	None	All cells in the same row	No	Written in Java	GBs of storage plus VMs per hour