Principles of Distributed and Parallel Database Systems: Part 2

NoSQL Database Systems

Ira A. Fulton Schools of Engineering
Arizona State University

Why NoSQL?

SQL databases are relational databases bases on tables

Change triggered by social media and cloud-based solutions

NoSQL began in 1998

Scaling up Vs. Scaling Out

Best way to provide ACID and rich query model is to have the dataset on a single machine

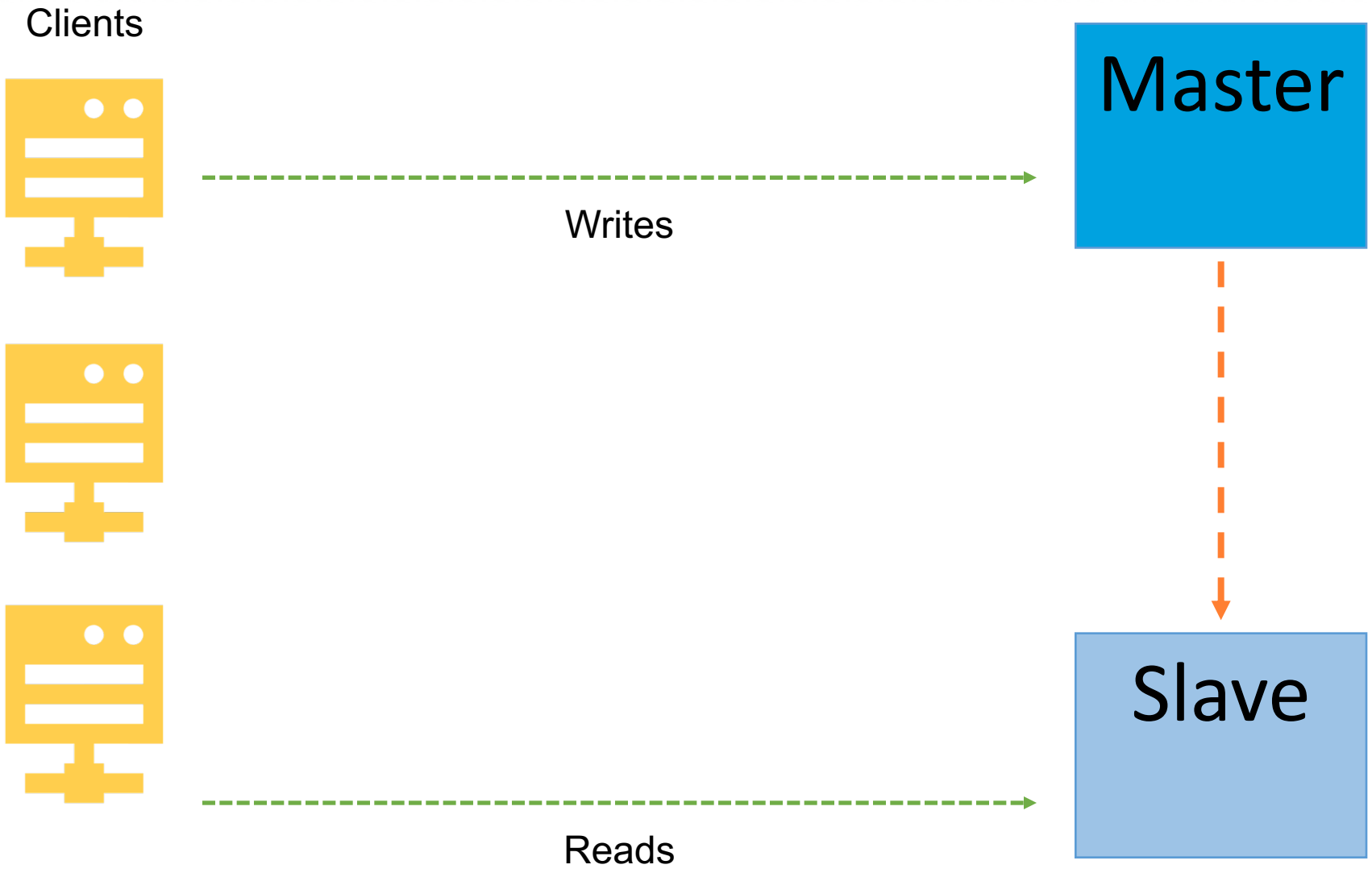
Scaling Up

- *Scaling up* (or *vertical scaling*: make a "single" machine more powerful) → dataset is just too big!

Scaling Out

- Scaling out (or *horizontal scaling*: adding more smaller/cheaper servers) is a better choice
- Different approaches for horizontal scaling (multi-node database):
  - Master/Slave
  - Sharding (partitioning)

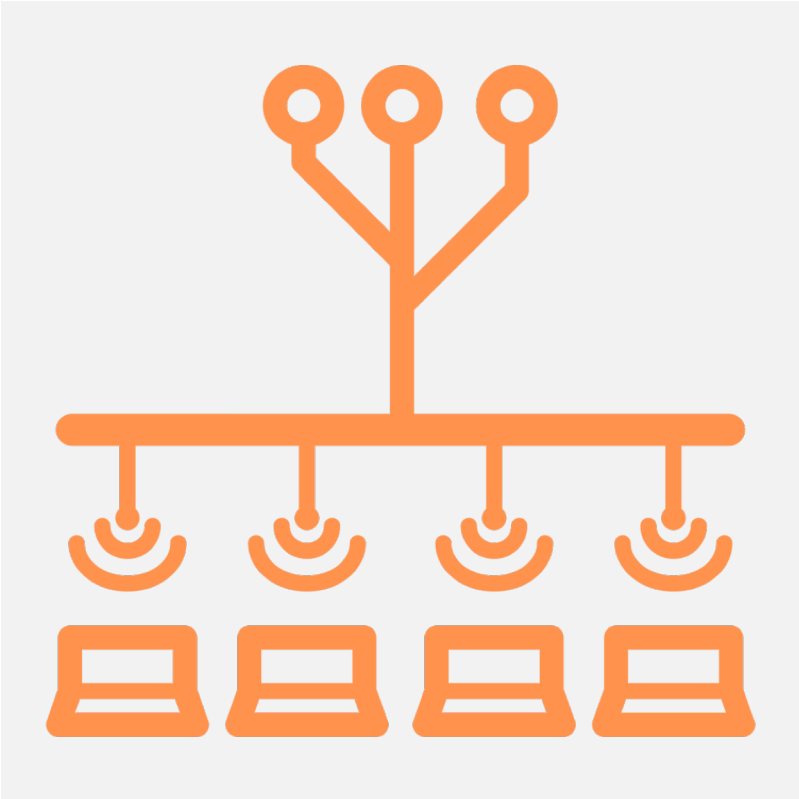
Scaling Out RDBMS: Master/Slave



Scaling out RDBMS: Sharding

Sharding (Partitioning)

- Scales well for both reads and writes
- Distributed Query Processing is challenging

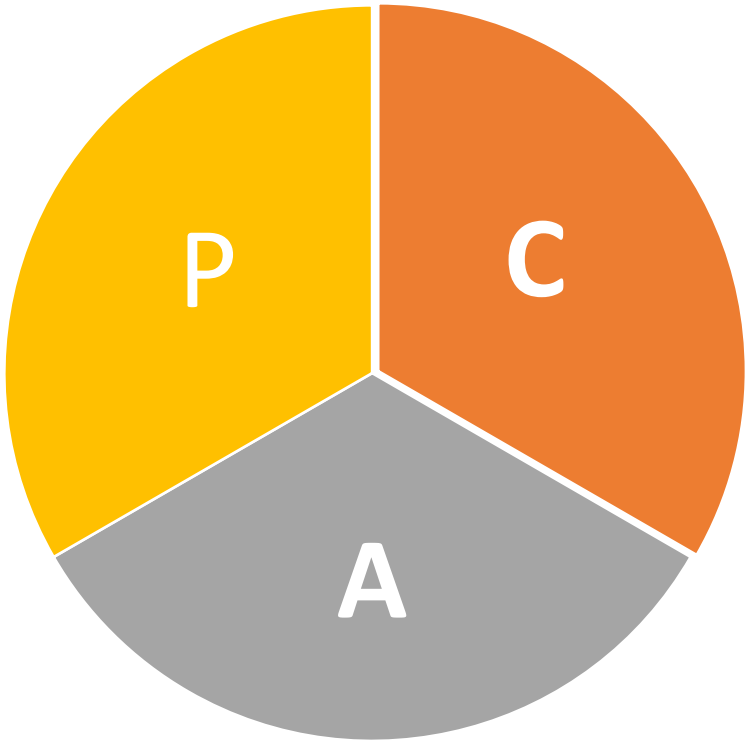


CAP Theorem

Partition-tolerance: system properties hold even through failures
Consistency: all copies have same value
Availability: reads and writes always succeed

CAP Theorem

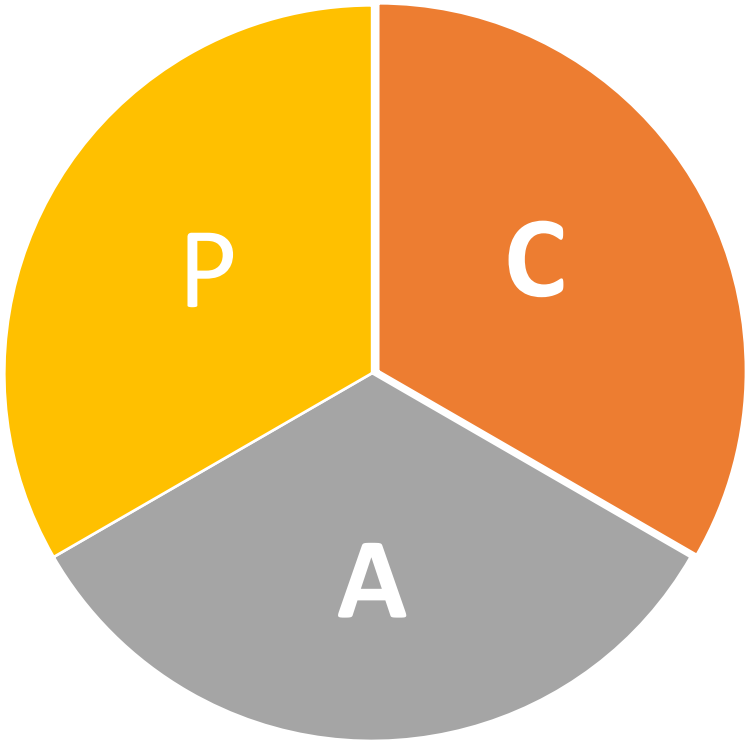
All clients always has the same view of the data



CAP Theorem: Consistency

Strong consistency – ACID (Atomicity, Consistency, Isolation, Durability)

Weak consistency – BASE
(Basically Available Soft-state Eventual consistency)



CAP Theorem

ACID

- A DBMS is expected to support "ACID transactions," processes that are:
  - **Atomicity**: either the whole process is done or none is
  - **Consistency**: only valid data are written
  - **Isolation**: one operation at a time
  - **Durability**: once committed, it stays that way

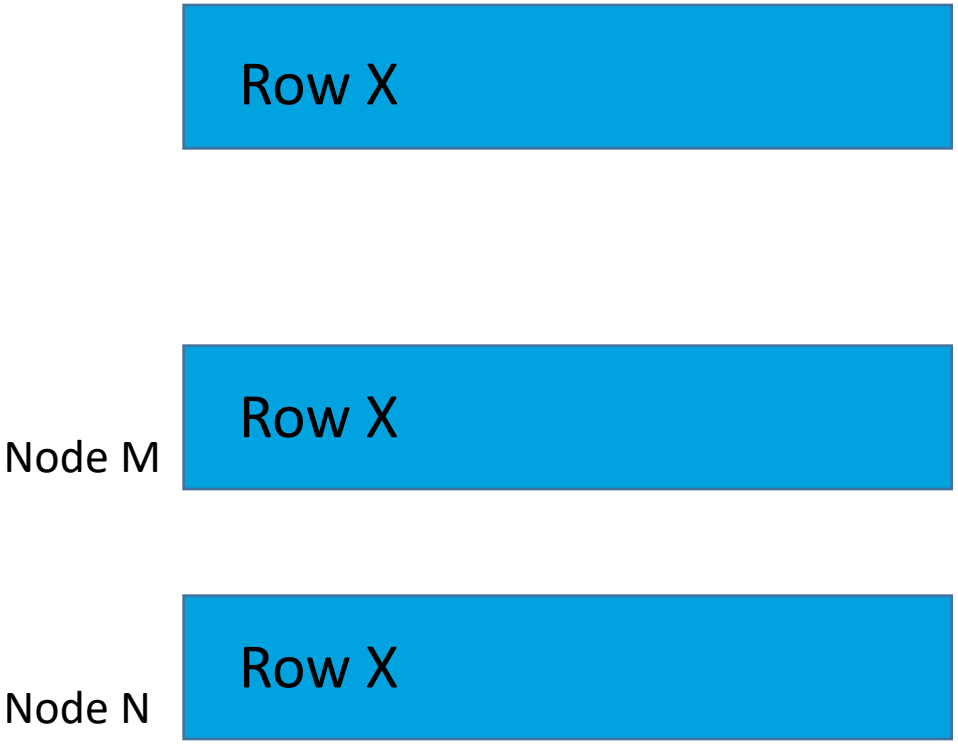
CAP

- **Consistency**: all data on cluster has the same copies
- **Availability**: cluster always accepts reads and writes
- **Partition tolerance:** guaranteed properties are maintained even when network failures prevent some machines from communicating with others

Cap Theorem

- A consistency model determines rules for visibility and apparent order of updates

Example



CAP Theorem

Eventual consistency

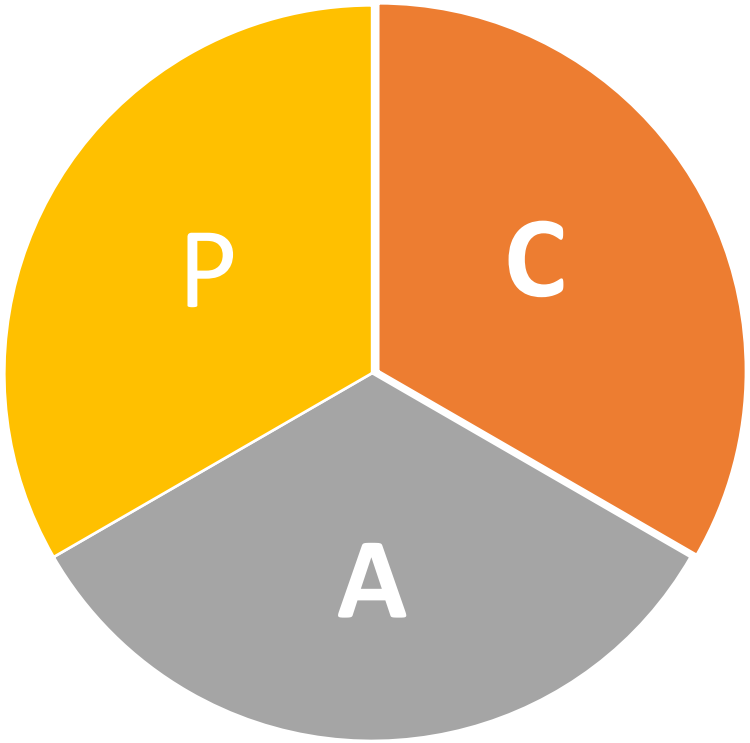
- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent

Cloud computing

- ACID is hard to achieve, moreover, it is not always required

CAP Theorem

- Each client always can read and write.
- A system can continue to operate in the presence of a network partitions



What is NoSQL?

Advantages

- Non-relational
- No require schema
- Data are replicated to multiple nodes
- Horizontal scalability
- Cheap, easy to implement
- Massive write performance
- Fast access

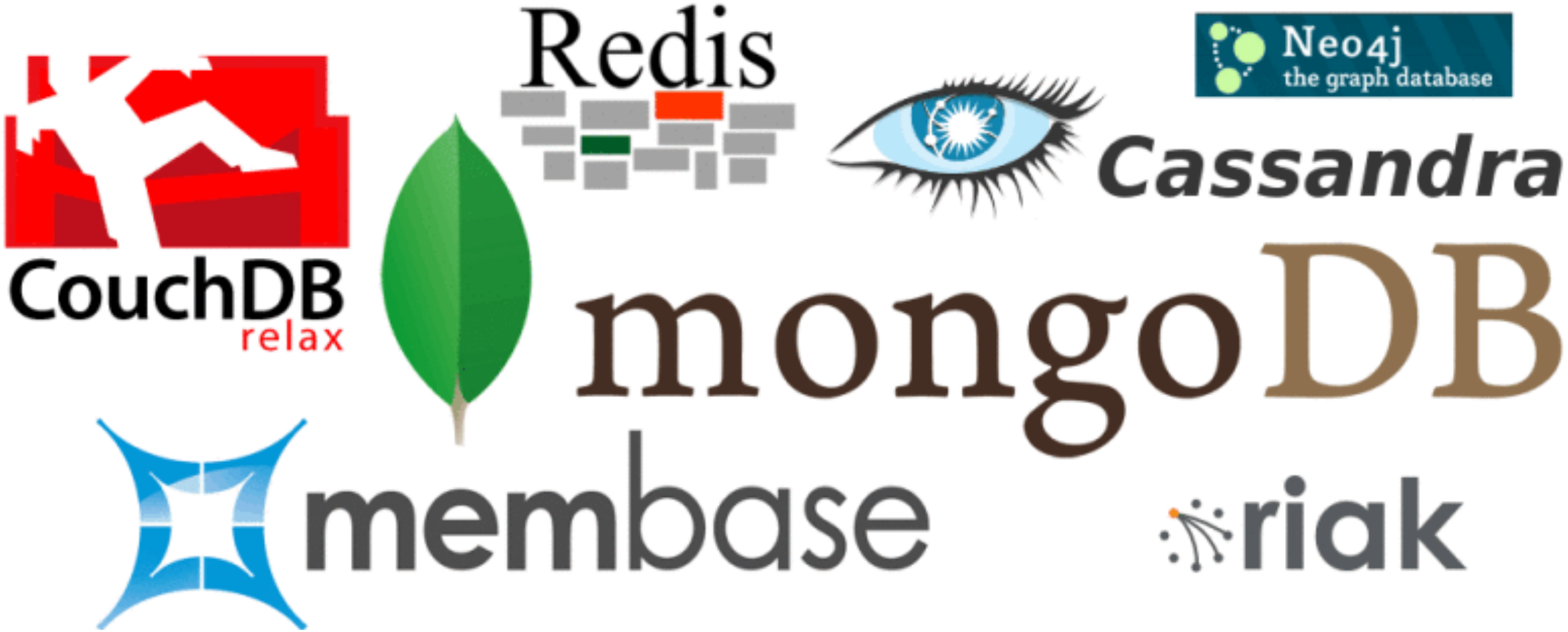
NoSQL Database Systems

What is NoSQL?

Disadvantages

- Does not fully support relational features
  - no join, group by, order by operations
  - no referential integrity constraints across partitions
- No declarative query language
- Relaxed ACID → fewer guarantees
- No easy integration with other applications that support SQL

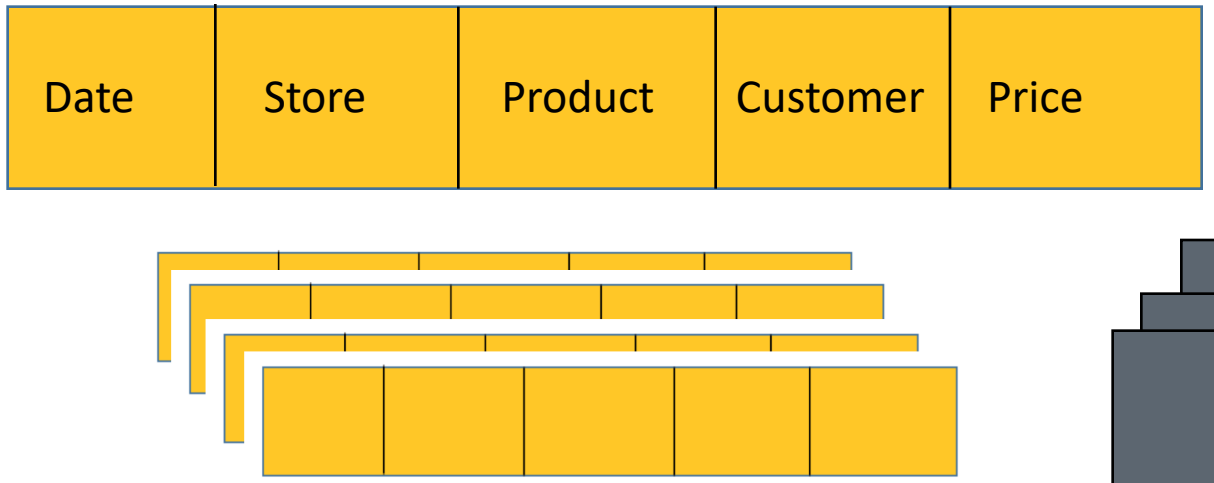
Who is using NoSQL?



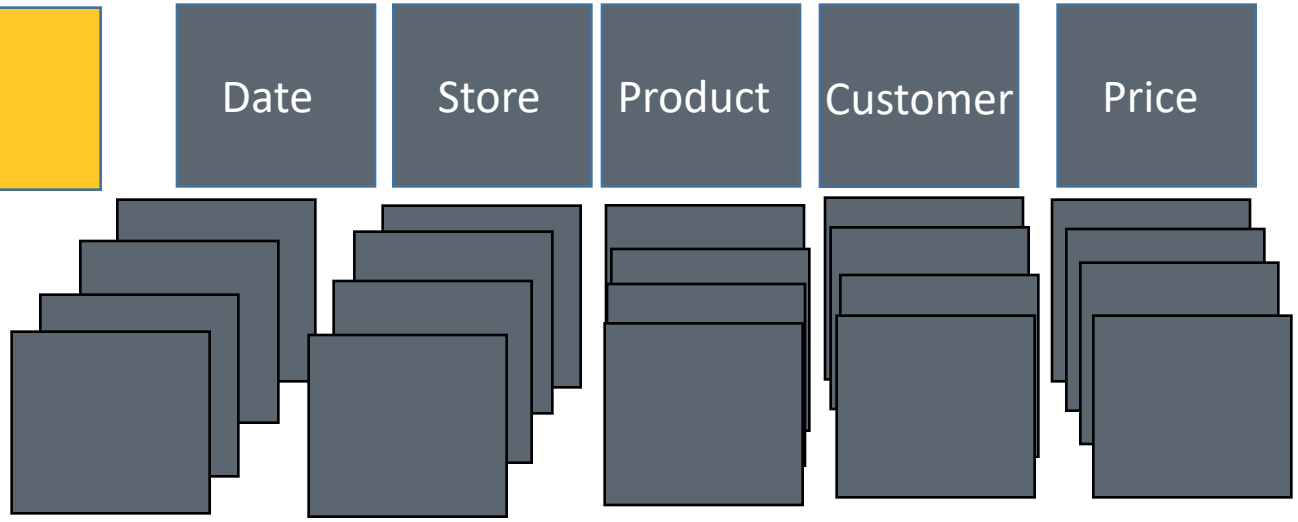
By John hoho - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=50527733

Row Store and Column Store

row-store



column-store



Row Store and Column Store

- Most of the queries does not process all the attributes of a particular relation
- For example the query
  - Select c.name and c.address
    From CUSTOMES as c
    Where c.region=Mumbai;
- Only process three attributes of the relation CUSTOMER. But the customer relation can have more than three attributes.
- Column-stores are more I/O efficient for read-only queries as they read, only those attributes which are accessed by a query.

Row Store and Column Store

Row Store

+ Easy to add/modify a record

- Might read in unnecessary data

Column Store

+ Only need to read in relevant data

- Tuple writes require multiple accesses

Why Column Stores?

- Can be significantly faster than row stores for some applications
  - Fetch only required columns for a query
  - Better cache effects
  - Better compression (similar attribute values within a column)
- But can be slower for other applications
  - OLTP with many row inserts, ..
- Long war between the column store and row store camps

Column Stores – Data Model

- Standard relational logical data model
  - EMP(name, age, salary, dept)
  - DEPT(dname, floor)
- Table – collection of projections
- Projection – set of columns
- Horizontally partitioned into segments with segment identifier

Column Stores – Data Model

To answer queries, projections are joined using Storage keys and join indexes

Storage Keys

- Within a segment, every data value of every column is associated with a unique Skey
- Values from different columns with matching Skey belong to the same logical row

Join Indexes

- T1 and T2 are projections on T
- M segments in T1 and N segments in T2
- Join Index from T1 to T2 is a table of the form:
  - (s: Segment ID in T2, k: Storage key in Segment s)
  - Each row in join index matches corresponding row in T1
- Join indexes are built such that T could be efficiently reconstructed from T1 and T2

Column Stores – Data Model

Name	Age
Jill	24
Bob	25
Bill	27

Name	Salary
Bob	10K
Bill	50K
Jill	80K

SID	Key
1	2
1	3
1	1

Compression

Trades I/O for CPU

- Increased column-store opportunities:
- Higher data value locality in column stores
- Techniques such as run length encoding far more useful

Schemes

- Null Suppression
- Dictionary encoding
- Run Length encoding
- Bit-Vector encoding
- Heavyweight schemes

Query Execution - Operators

Select: Same as relational algebra but produces a a bit string

Project: Same as relational algebra

Join: Joins projections according to predicates

Aggregation: SQL like aggregates

Sort: Sort all columns of a projection

Decompress: Converts compressed column to uncompressed representation

Mask (Bitsting B, projection Cs) emit only those values whose corresponding bits are 1

Concat: Combines one or more projections sorted in the same order into a single projection

Permute: Permutes a projection according to the ordering defined by a join index

Bitstring operators: Band – Bitwise AND, Bor – Bitwise OR, Bnot – complement

Row Store vs. Column Store

- How much of the buzz around column-stores is marketing hype?
  - Do you really need to buy Sybase IQ or Vertica?
- How far will your current row-store take you?
- Can you get column-store performance from a row-store?
- Can you simulate a column-store in a row-store?

Row Store vs. Column Store

- You can obtain the performance benefits of a column-store using a row-store by making some changes to the physical structure of the row store.

- These changes can be:
  - Vertically partitioning
  - Using index-only plans
  - Using materialized views

Vertical Partitioning

Process:

- Full Vertical partitioning of each relation
  - Each column =1 Physical table
  - This can be achieved by adding integer position column to every table
- Join on Position for multi column fetch

Problems:

- “Position” - Space and disk bandwidth
- Header for every tuple – further space wastage
  - e.g. 24 bytes overhead in PostgreSQL

Vertical Partitioning

Last Name	First Name	E-mail	Phone #	Street Address

Last Name	
1	
2	
3	

First Name	
1	
2	
3	

Email	
1	
2	
3	

Index-Only Plans

Process:

- Add B+Tree index for every Table.column
- Plans never access the actual tuples on disk
- Headers are not stored, so per tuple overhead is less

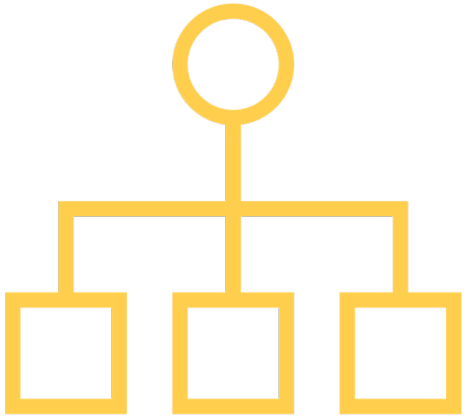
Problem:

- Separate indices may require full index scan, which is slower
- Eg:
```
SELECT AVG(salary)
FROM emp
WHERE age > 40
```
- Composite index with (age, salary) key helps.

Index-Only Plans: Example

Last Name	First Name	E-mail	Phone #	Street Address

Last Name Index



First Name Index

Materialized Views

- Process:
  - Create ‘optimal' set of MVs for given query workload
  - Objective:
    - Provide just the required data
    - Avoid overheads
    - Performs better
- Expected to perform better than other two approach

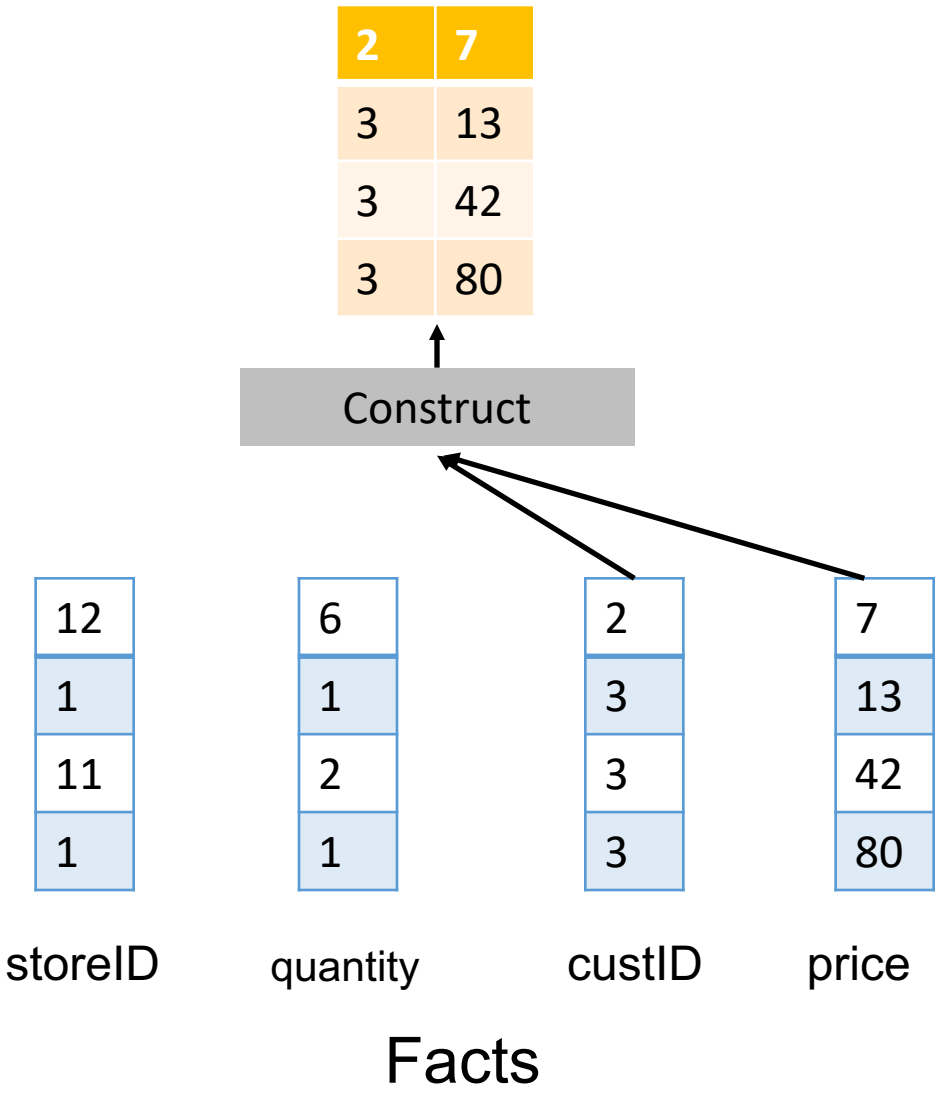
- Problems:
  - Practical only in limited situation
  - Require knowledge of query workloads in advance

Materialized View Example

Select F.custID

from Facts as F

where F.price>20



Optimizing Column Oriented Execution

- Different optimization for column oriented database
  - Compression
  - Late Materialization
  - Block Iteration
  - Invisible Join

Compression

Low information entropy (high data value locality) leads to High compression ratio

- Advantage
  - Disk Space is saved
  - Less I/O
  - CPU cost decrease if we can perform operation without decompressing
- Light weight compression schemes do better

Compression

- If data is sorted on one column that column will be super-compressible in row store
  - eg. Run length encoding

Quarter

Q1
Q1
Q1
Q1
Q1
Q1
Q1
…
Q2
Q2
Q2
Q2
…

Quarter

(Q1, 1, 300)
(Q2, 301, 350)
(Q3, 651, 5000)

Late Materialization

- Most query results entity-at-a-time not column-at-a-time
- So at some point of time multiple columns must be combined
- One simple approach is to join the columns relevant to a particular query
- But further performance can be improved using late-materialization

Late Materialization

- Delay Tuple Construction
- Might avoid constructing it altogether
- Intermediate position lists might need to be constructed

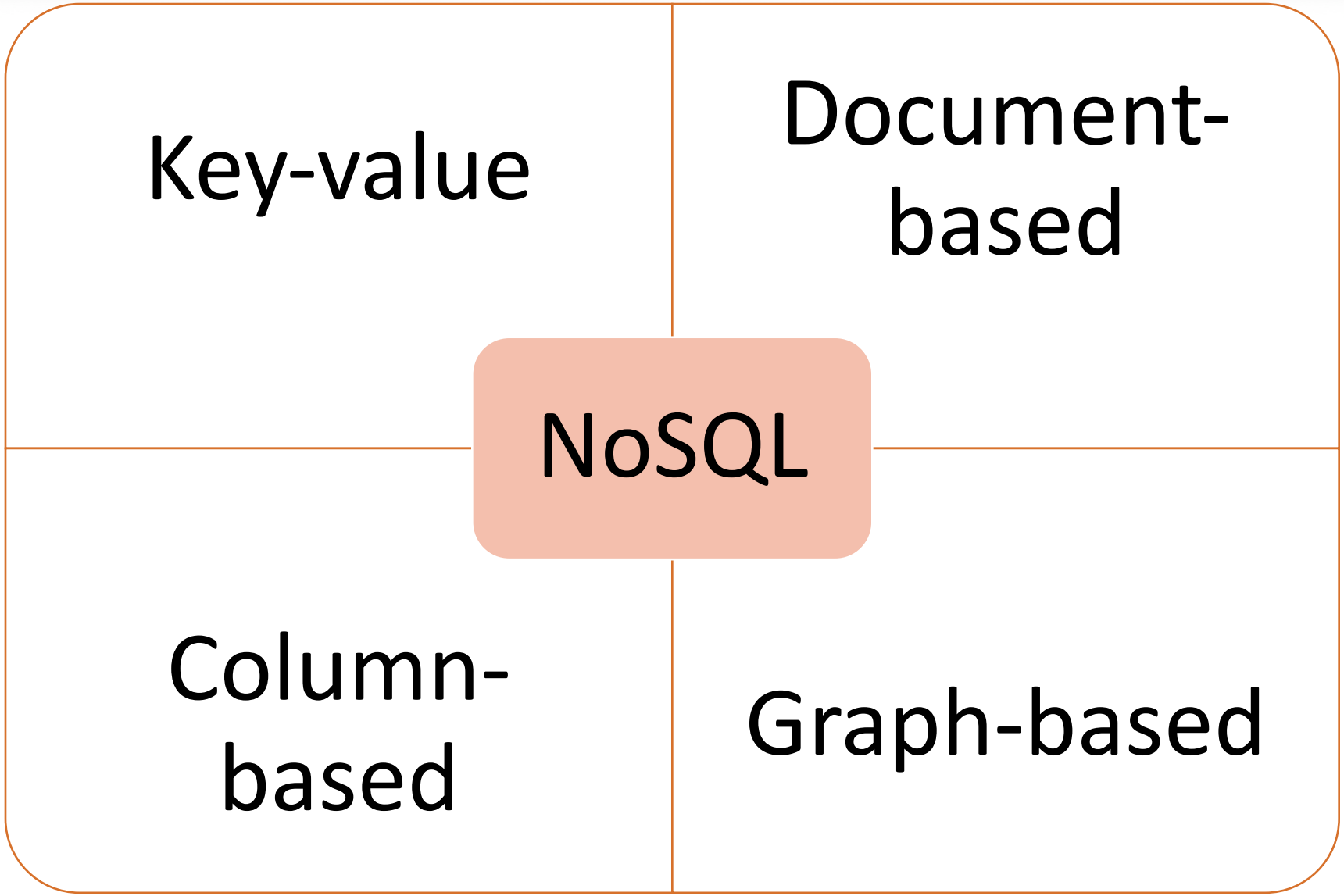
- **Eg: SELECT R.a FROM R WHERE R.c = 5 AND R.b = 10**
  - Output of each predicate is a bit string
  - Perform Bitwise AND
  - Use final position list to extract R.a

Late Materialization

Advantages

- Unnecessary construction of tuple is avoided
- Direct operation on compressed data
- Cache performance is improved (PAX)

NoSQL Categories



Large Graph Processing

Social Graph

- Facebook has 2.13 billion monthly users as of February 2018

Road Network Graph

- 400 GB of Road Network Data

Brain Graph

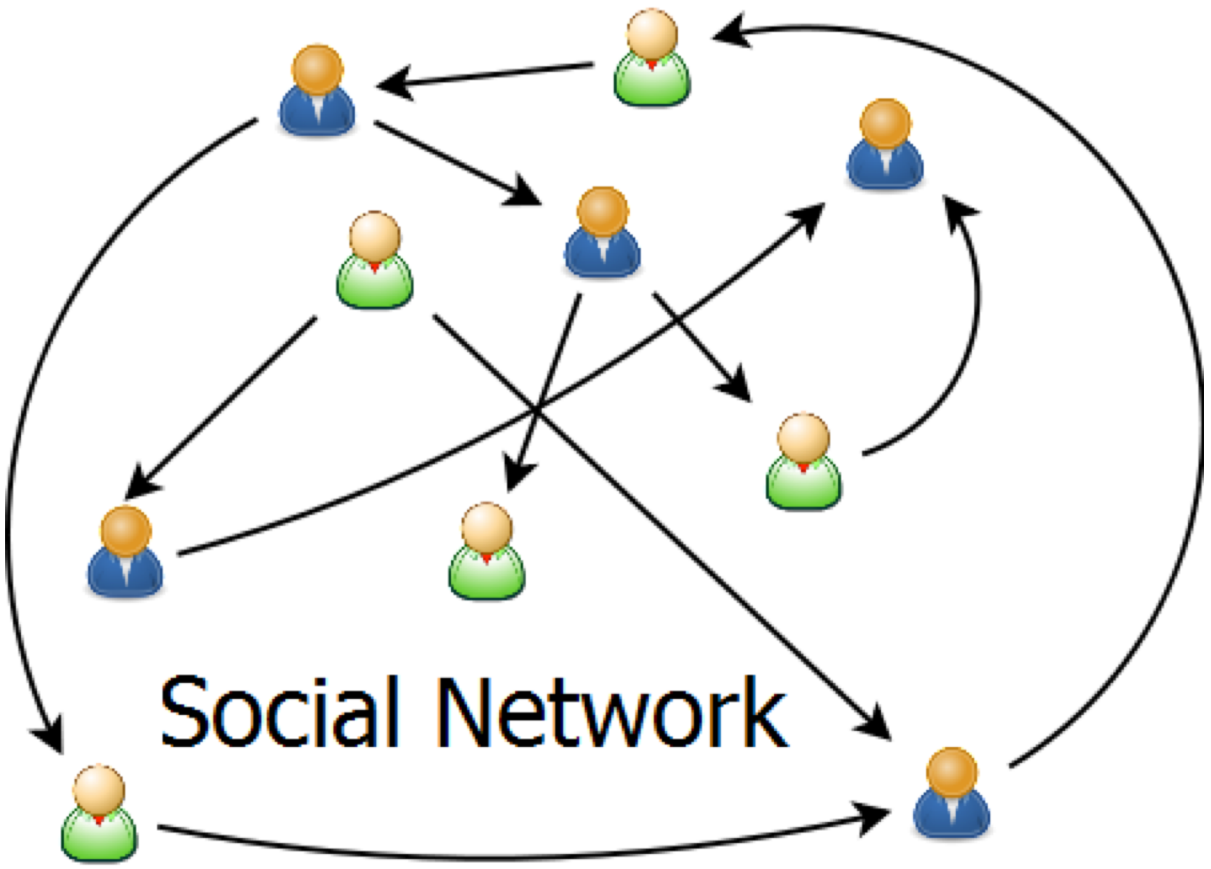
- 85 Billion neurons in the brain nervous system

Social Network

Social network

- Persons
- Friends
- Photos

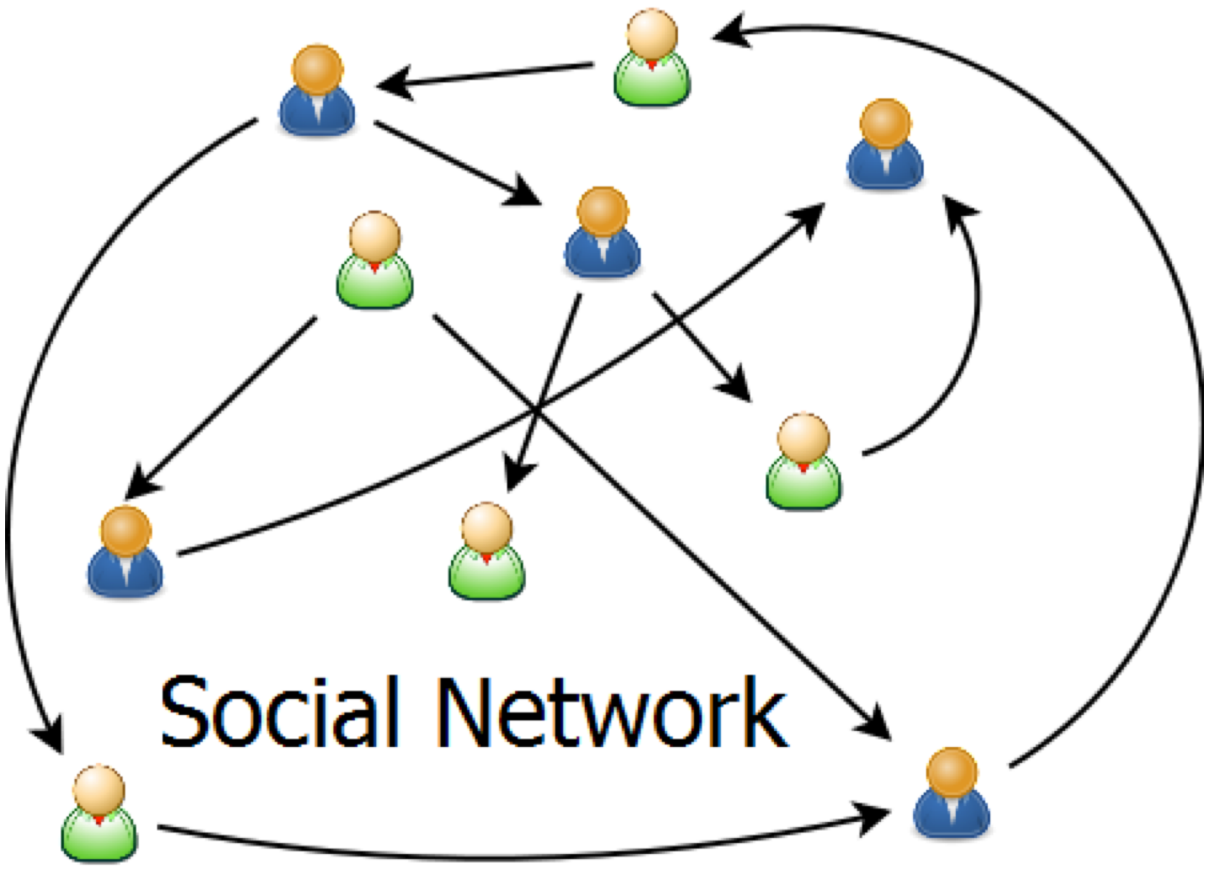
- **Queries**
  - Find Alice's friends
  - How Alice & Ed are connected
  - Find Alice's photos with friends



By Zigomitros Athanasios - Thor4bp [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons

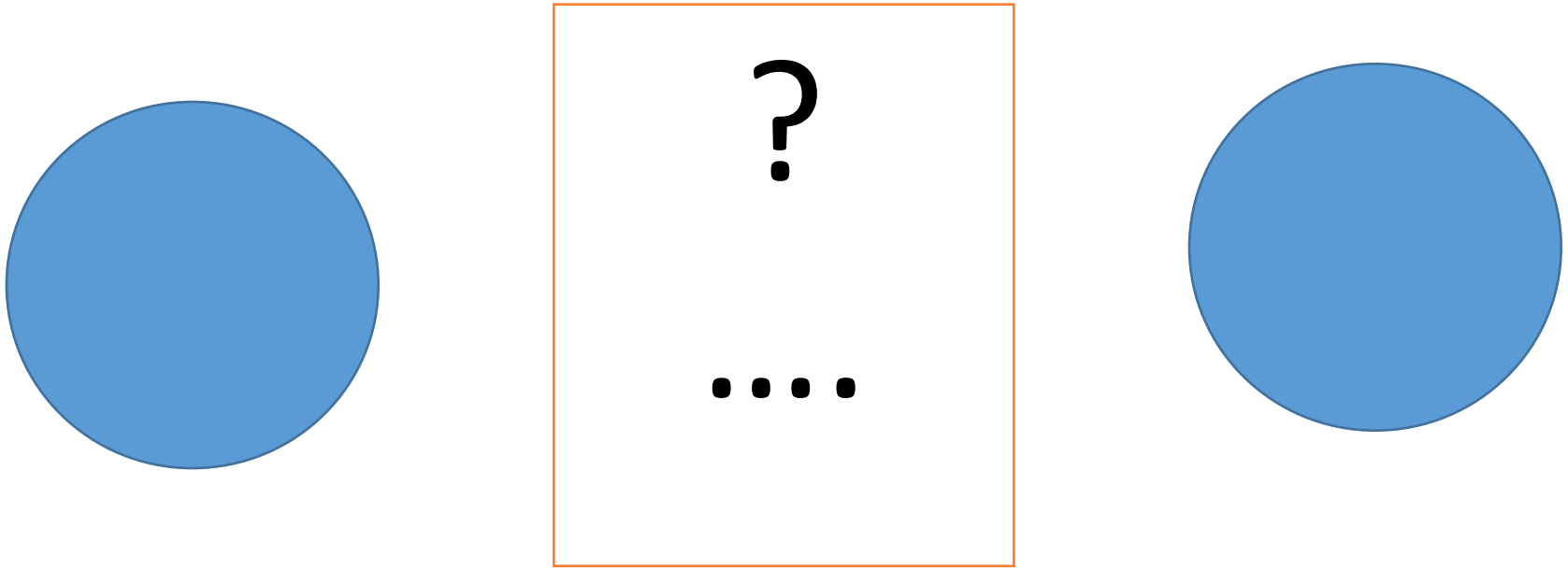
Social Network

- **Attributed multi-graph**
- **Node**
  - Represent entities
  - ID, type, attributes
- **Edge**
  - Represent binary relationship
  - Type, direction, weight, attrs



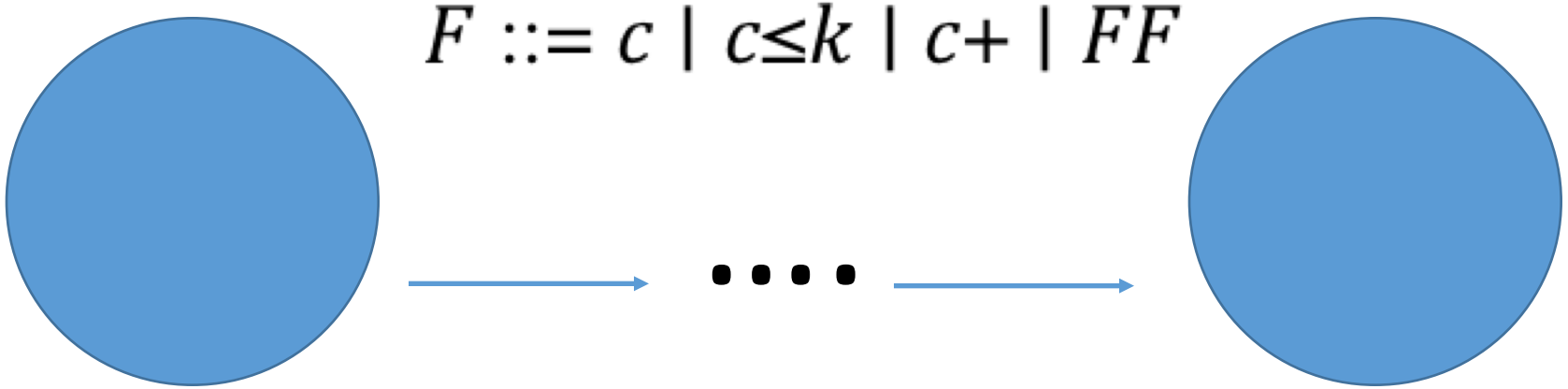
By Zigomitros Athanasios - Thor4bp [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons

Graph Reachability Query



Graph Reachability Query

$$F ::= c \mid c{\leq}k \mid c{+} \mid FF$$



Graph Reachability Queries

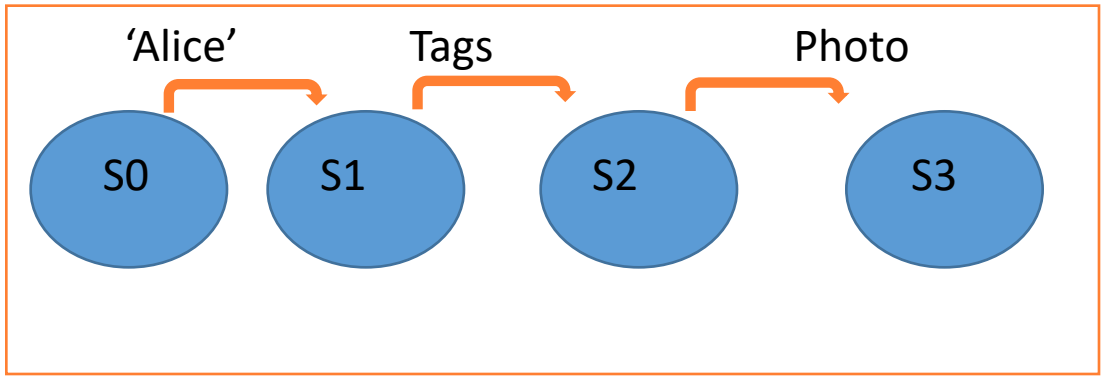
Query is a regular expression

- Sequence of node and edge predicates

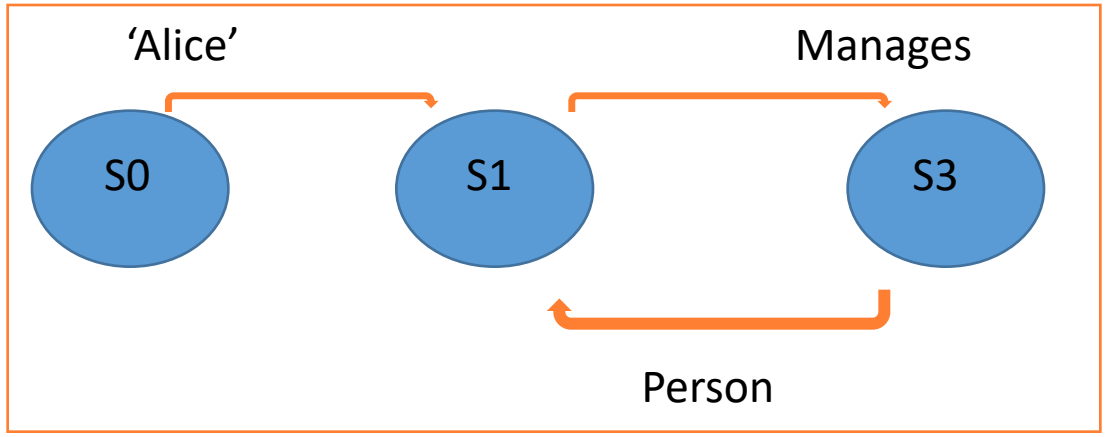
1. **Hello world in reachability**
   - Photo-Tags-'Alice'
   - Search for path with node: type=Photo, edge: type=Tags, node: id='Alice'
2. **Attribute predicate**
   - Photo{date.year='2012'}-Tags-'Alice OR
   - (Photo | video)-Tags-'Alice'
3. **Closure for path with arbitrary length**
   - 'Alice'(-Manages-Person)*
   - Kleene star to find Alice's org chart

Compile into Algebraic Query Plan

‘Alice’-Tags-Photo

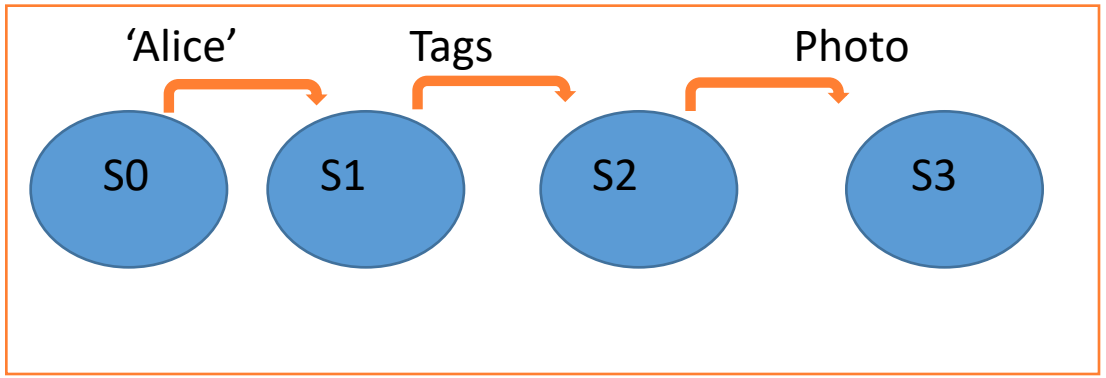


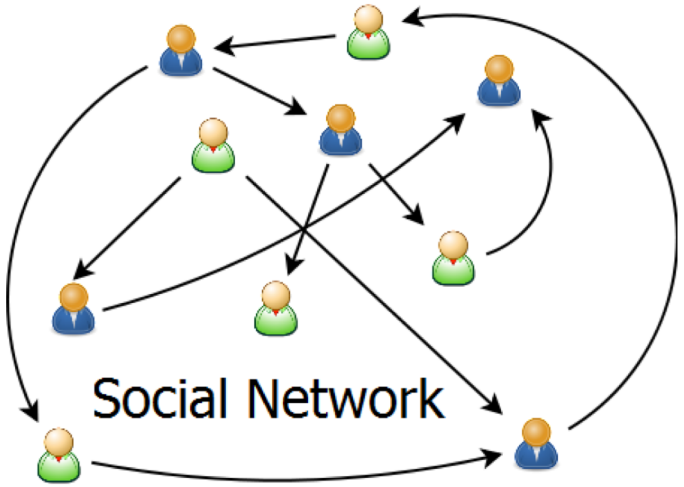
‘Alice’ (-Manages-Person)*



Centralized Query Execution: Breadth First Search

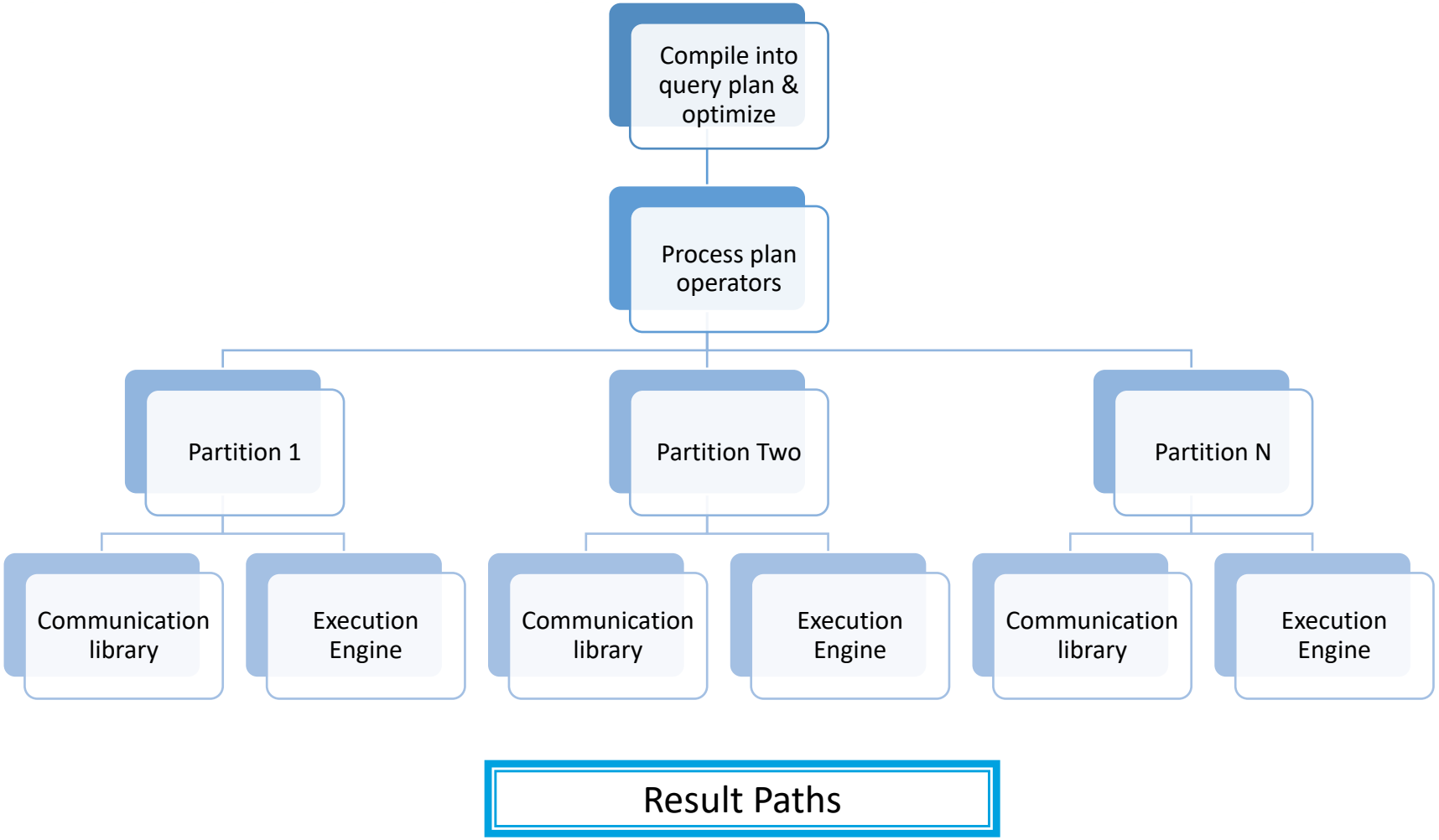
‘Alice’-Tags-Photo



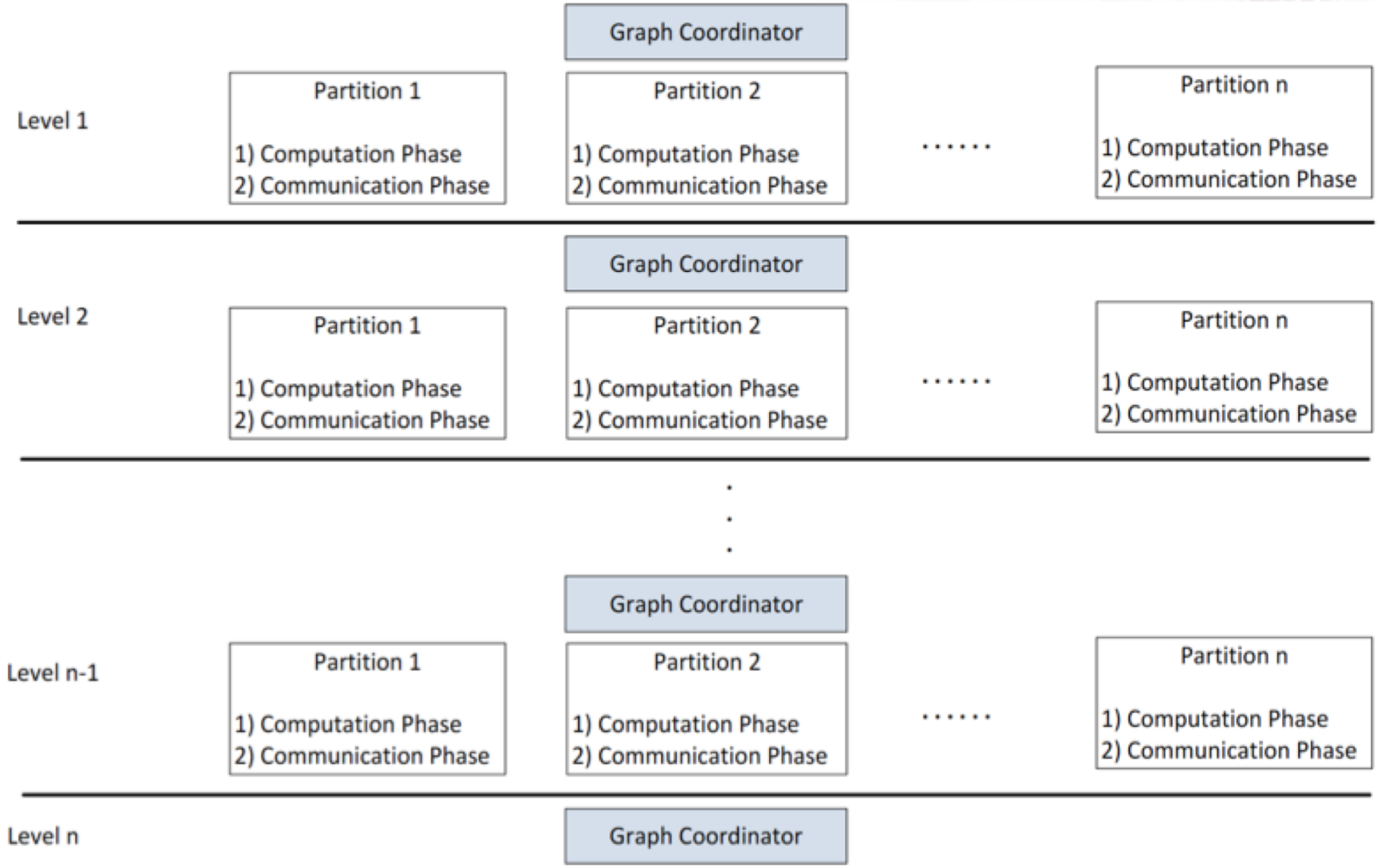


Answer Paths:
‘Alice’-Tags-Photo1
‘Alice’-Tags-Photo8

Distributed Query Execution



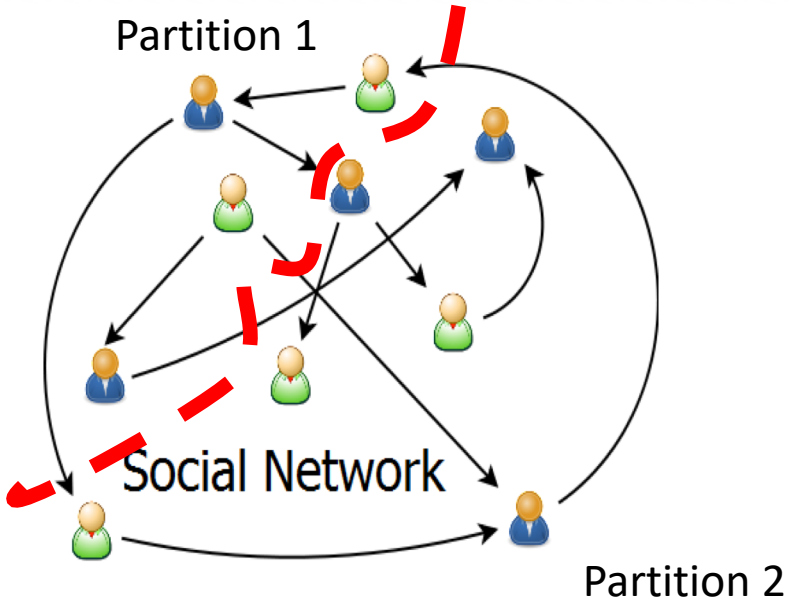
Distributed BFS Traversal



Distributed Query Execution

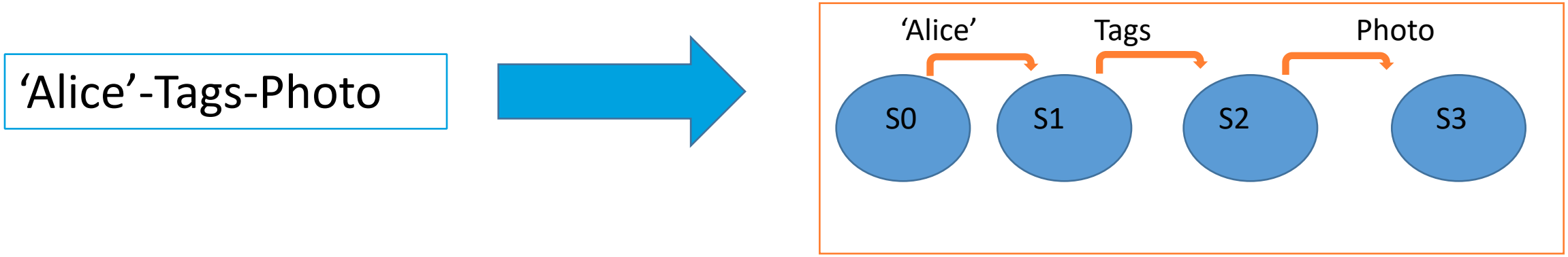
‘Alice’-Tags-Photo- Tags- ‘Bob’

Partition 1	Partition 2	FSM
Step 1 Alice		S0 S1
Step 2	Photo 1 Photo 8	S2 S3
Step 3	Bob	S4 S5



Algebraic Operators

1. **Select**
   - Find set of starting nodes
2. **Traverse**
   - Traverse graph to construct paths
3. **Join**
   - Construct longer paths



Plan Enumeration for Query Optimization

- **Query:** ‘Mike’-Tags-Photo-Tags-Person-FriendOf-‘Mike’
- **Example plans**
  1. Left to right
     - ‘Mike’-Tags-Photo-Tags-Person-FriendOf-‘Mike’
  2. Right to left
     - ‘Mike’-FriendOf-Person-Tags-Photo-Tags-‘Mike’
  3. Split then join
     - (‘Mike’-FriendOf-Person) ⋈ (Person-Tags-Photo-Tags-‘Mike’)
  4. Split then join
     - (‘Mike’-FriendOf-Person-Tags-Photo) ⋈ (Photo-Tags-‘Mike’)
  5. …

Enumeration Algorithm

Query: $Q[1, n] = N_1\ E_1\ N_2\ E_2 \ldots\ldots N_{n-1}\ E_{n-1}\ N_n$

Selectivity of query Q[i,j] : Sel(Q[i,j])

Minimum cost of query Q[i,j] : F(Q[i,j])

```
F(Q[i,j]) = min{
    SequentialCost_LR(Q[i,j]),
    SequentialCost_RL(Q[i,j]),
    min_{i<k<j}  (F(Q[i,k]) + F(Q[k,j]) + Sel(Q[i,k])*Sel(Q[k,j]))
    }
```

Base step: $F(Q_i) = F(N_i)$ = Cost of matching predicate N_i

Apply dynamic programming

Store intermediate results of all F(Q[i,j]) pairs

Complexity: $O(n^3)$

NoSQL Database System

NoSQL solutions fall into two major areas:

- **Key/Value or ‘the big hash table’.**
  - Dynamo
- **Schema-less which comes in multiple flavors, column-based, document-based or graph-based.**
  - Cassandra (column-based)
  - CouchDB (document-based)
  - Neo4J (graph-based)

Requirements

Availability

Scalability

Fault Tolerance

Manageability

Dynamo's Assumption

Query Model:

- Simple R/W operations to data with unique IDs
- No operation span multiple records
- Data stored as binary objects of small size

ACID Properties

- Weaker (eventual) consistency

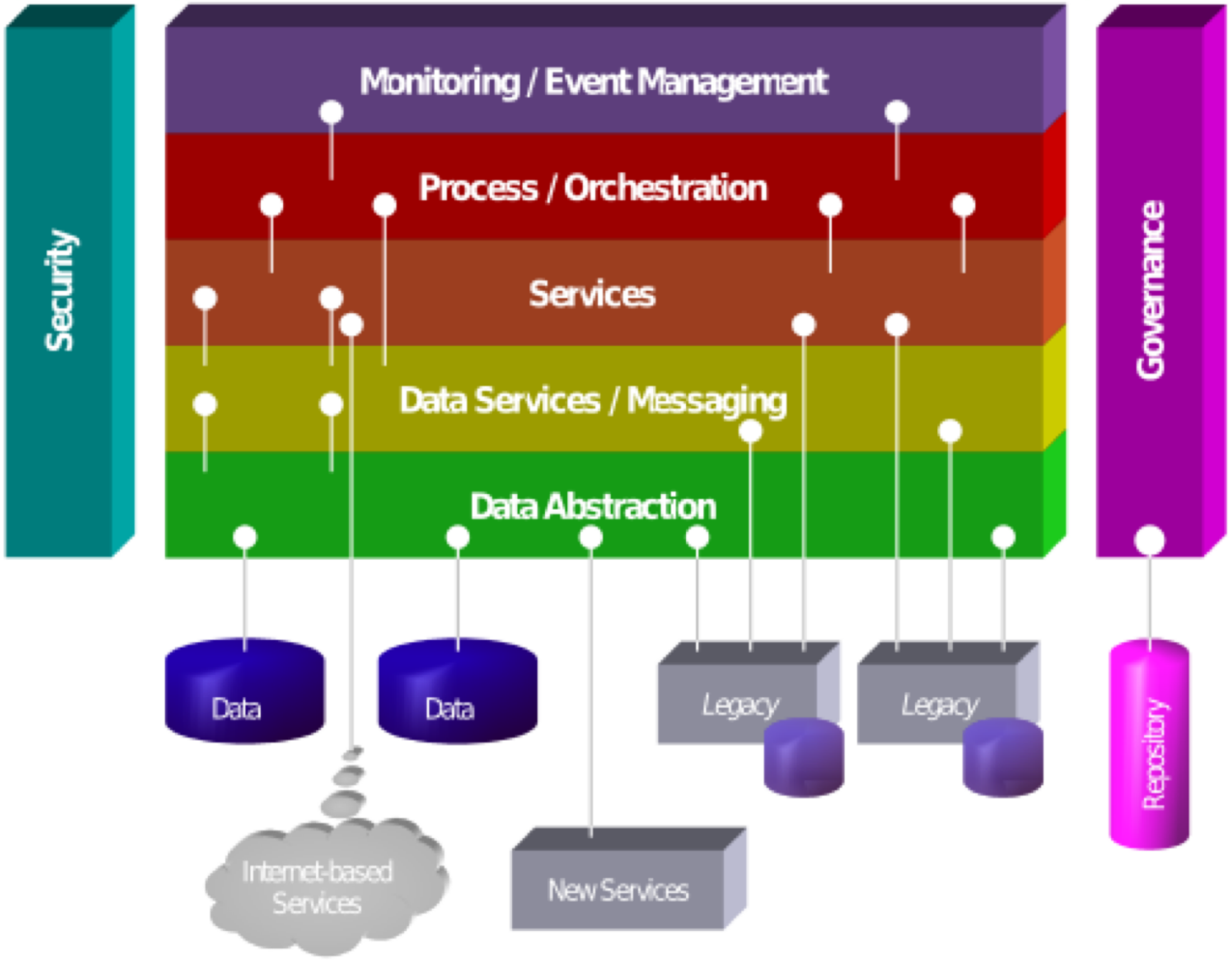
Efficiency

- Optimize for the 99.9th percentile

Service Level Agreements

- Cloud-computing and virtual hosting contracts include SLAs
- Most are described in terms of mean, median, and variance of response times
  - Suffers from outliers

Service-Oriented Architecture



By Loïc Corbasson, created with en:OOo Draw (ODG source file available on request) [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)], via Wikimedia Commons

Design Decisions

Incremental Scalability

- Must be able to add nodes on-demand with minimal impact

Load Balancing

Design Decisions: Replication

When?

- Solve on write to reduce read complexity
- Solve on read and reduce write complexity
  - Dynamo is an “always writeable” data store
  - Fine for shopping carts and such services

Who ?

- Data store
- User application

Dynamo's System Interface

put (key, context, object)

- key: primary key associated with data object
- context: vector clocks and history (needed for merging)
- object: data to store

got (key)

Data Partitioning & Replication

Use consistent hashing

- Each node gets an ID from the space of keys
- Nodes are arranged in a ring
- Data stored on the first node clockwise of the current placement of the data key

Replication

- Preference lists of N nodes following the associated node

Data Versioning

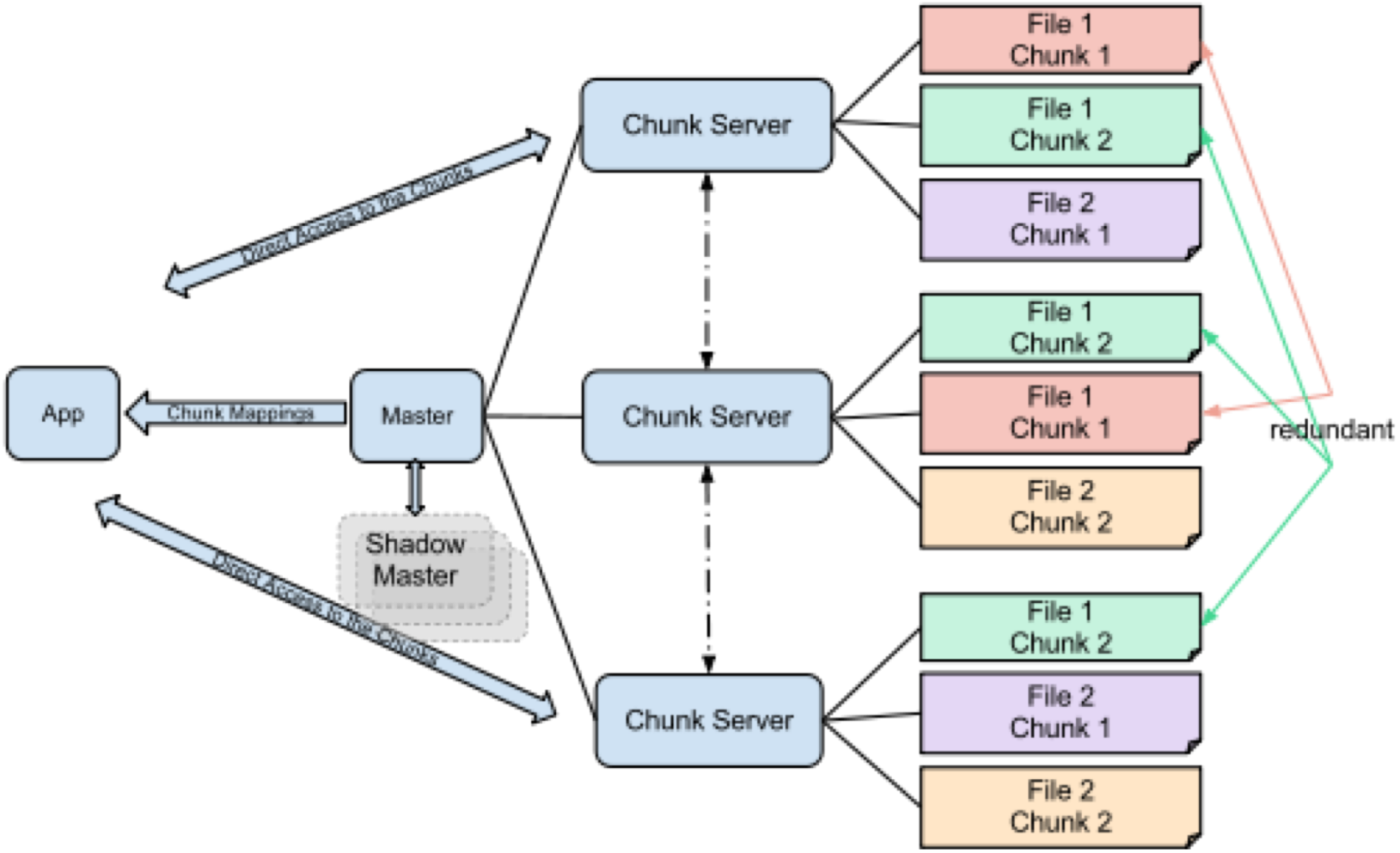
- **Updates generate a new timestamp**
- **Eventual consistency**
  - Multiple versions of the same object might co-exist
- **Syntactic Reconciliation**
  - System might be able to resolve conflicts automatically
- **Semantic Reconciliation**
  - Conflict resolution pushed to application

Google Big Table: Building Blocks

- **BigTable:** build a more application-friendly storage service using these parts

- **Scheduler (Google WorkQueue)**
- **Google Filesystem**
- **Chubby Lock service**
- **Two other pieces helpful but not required**
  - MapReduce (despite what the Internet says)

Google File System



By Saqibali [CC0], via Wikimedia Commons

Chubby

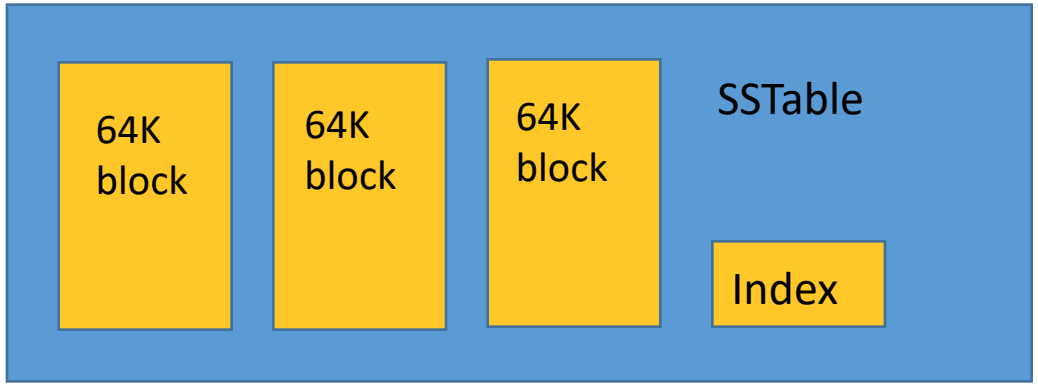
- {lock/file/name} service
- Coarse-grained locks, can store small amount of data in a lock

Data Model: A Big Map

- **<Row, Column, Timestamp> triple for key - lookup, insert, and delete API**
- **Arbitrary “columns” on a row-by-row basis**
  - Column family:qualifier. Family is heavyweight, qualifier lightweight
  - Column-oriented physical store- rows are sparse!
- **Does not support a relational model**
  - No table-wide integrity constraints
  - No multirow transactions

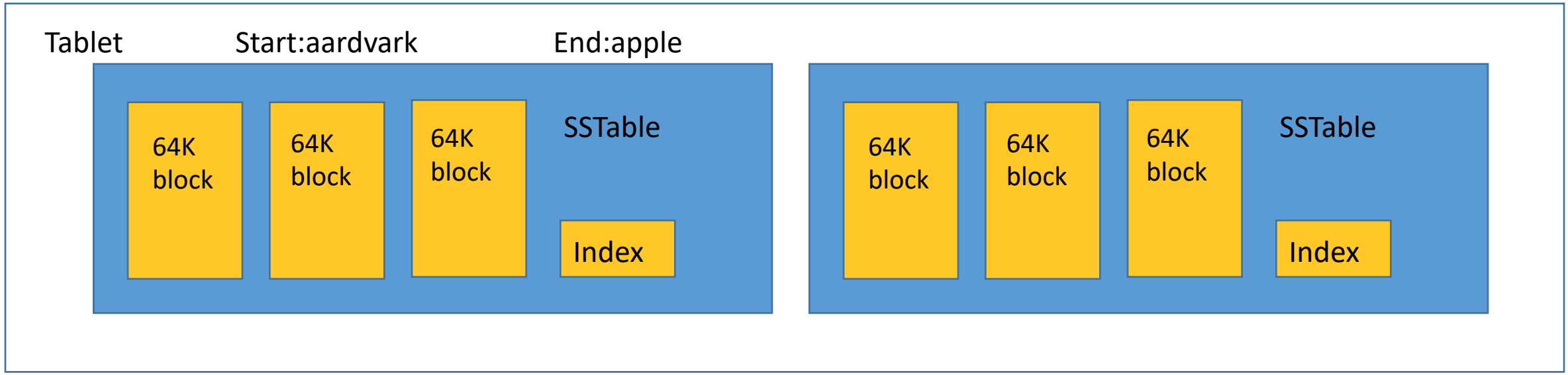
SSTable

- **Immutable, sorted file of key-value pairs**
- **Chunks of data plus an index**
  - Index is of block ranges, not values



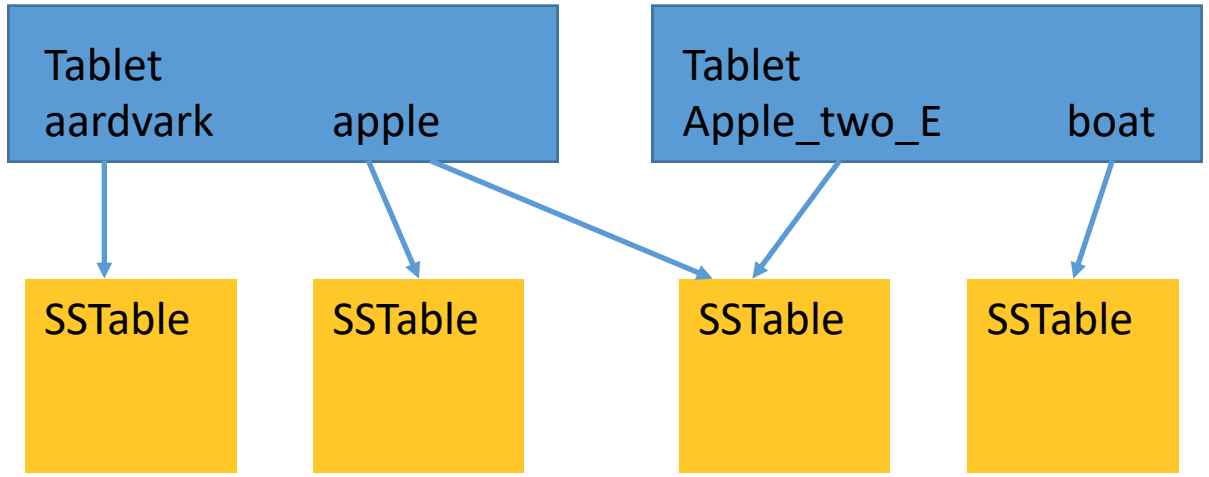
Tablet

- Contains some range of rows of the table
- Built out of multiple SSTables



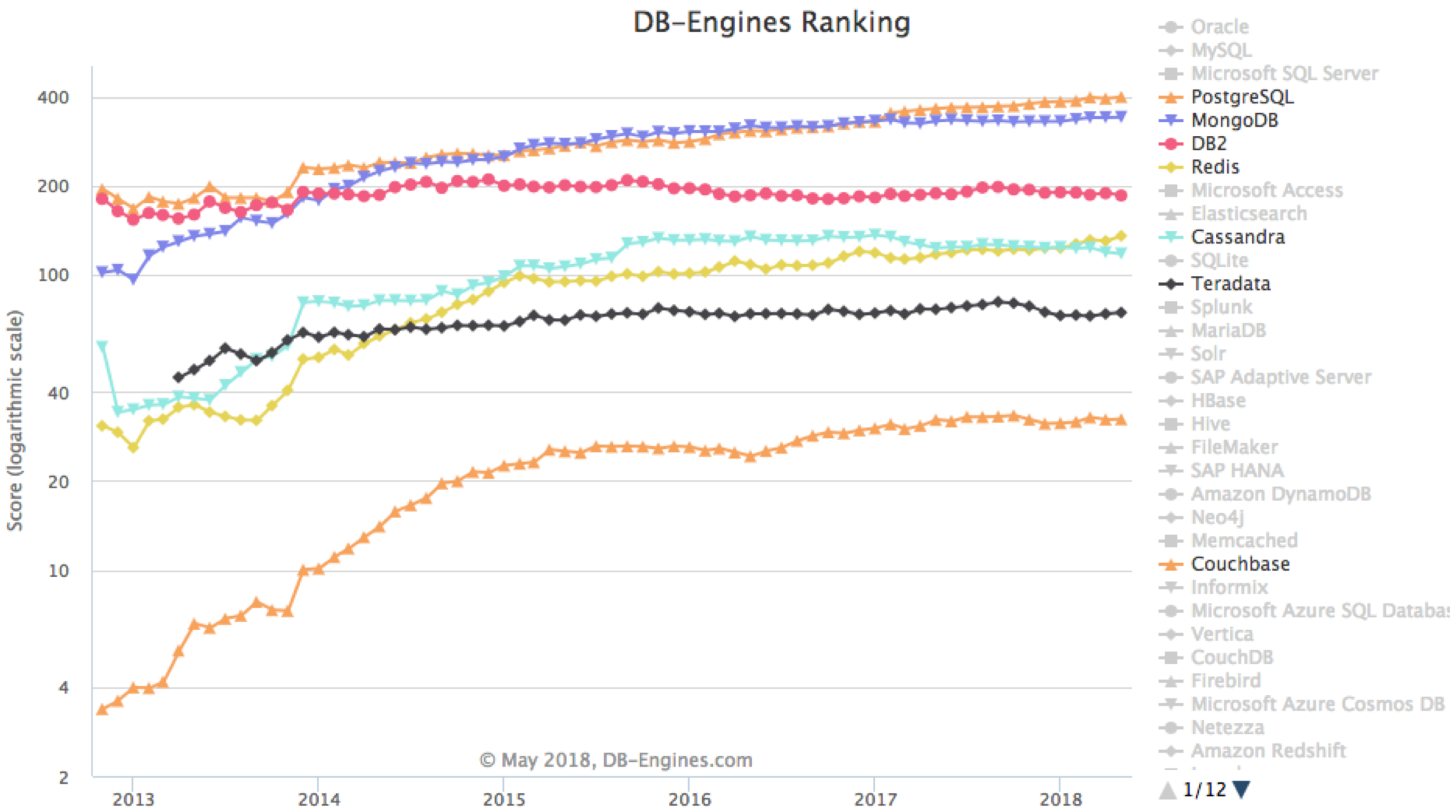
Table

- Multiple tablets make up the table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap



Servers

- **Tablet servers manage tablets, multiple tablets per server.**
  - Each tablet lives at only one server
  - Tablet server splits tablets that get too big
- **Master responsible for load balancing and fault tolerance**
  - GFS replicates data. Prefer to start tablet server on same machine that the data is already


DB-Engines Ranking
Score (logarithmic scale)
400
200
100
40
20
10
4
2
2013
2014
2015
2016
2017
2018
© May 2018, DB-Engines.com
Oracle
MySQL
Microsoft SQL Server
PostgreSQL
MongoDB
DB2
Redis
Microsoft Access
Elasticsearch
Cassandra
SQLite
Teradata
Splunk
MariaDB
Solr
SAP Adaptive Server
HBase
Hive
FileMaker
SAP HANA
Amazon DynamoDB
Neo4j
Memcached
Couchbase
Informix
Microsoft Azure SQL Databa
Vertica
CouchDB
Firebird
Microsoft Azure Cosmos DB
Netezza
Amazon Redshift
1/12

Document Databases: Requirements

Locations

- Need to store locations (Offices, Restaurants etc)
  - Want to be able to store name, address and tags
  - Maybe User Generated Content, i.e. tips / small notes ?
- Want to be able to find other locations nearby

Checkins

- User should be able to ‘check in’ to a location
- Want to be able to generate statistics

JSON Sample Doc

```
{ _id : ObjectId("4c4ba5c0672c685e5e8aabf3"),
    author : "roger",
    date : "Sat Jul 24 2010 19:47:11 GMT-0700 (PDT)",
    text : ”MongoSF",
    tags : [ ”San Francisco", ”MongoDB" ] }
```

Notes:

- _id is unique, but can be anything you’d like

BSON

- **JSON has powerful, but limited set of datatypes**
  - Mongo extends datypes with Date, Int types, Id, …
- **MongoDB stores data in BSON**
- **BSON is a binary representation of JSON**
  - Optimized for performance and navigational abilities
  - Also compression

  - See bsonspec.org

Locations v1

```
location1= {
  name: "10gen East Coast”,
        address: ”134 5th Avenue 3rd Floor”,
        city: "New York”,
        zip: "10011”
  }
```

Places v1

```
location1= {
  name: "10gen East Coast”,
        address: ”134 5th Avenue 3rd Floor”,
        city: "New York”,
        zip: "10011”
  }
```

```
db.locations.find({zip:”10011”}).limit(10)
```

Places v2

```
location1 = {
  name: "10gen East Coast”,
        address: "17 West 18th Street 8th Floor”,
        city: "New York”,
        zip: "10011”,

  tags: [“business”, “mongodb”]
  }
```

Places v2

```
location1 = {
  name: "10gen East Coast”,
        address: "17 West 18th Street 8th Floor”,
        city: "New York”,
        zip: "10011”,

  tags: [“business”, “mongodb”]
  }
db.locations.find({zip:”10011”, tags:”business”})
```

Places v3

```
location1 = {
  name: "10gen East Coast”,
        address: "17 West 18th Street 8th Floor”,
        city: "New York”,
        zip: "10011”,

  tags: [“business”, “mongodb”],

  latlong: [40.0,72.0]
  }
```

Places v3

```
location1 = {
  name: "10gen East Coast”,
        address: "17 West 18th Street 8th Floor”,
        city: "New York”,
        zip: "10011”,

  tags: [“business”, “cool place”],

  latlong: [40.0,72.0]
  }
```

```
db.locations.ensureIndex({latlong:”2d”})
```

Places v3

```
location1 = {
  name: "10gen HQ”,
        address: "17 West 18th Street 8th Floor”,
        city: "New York”,
        zip: "10011”,

  tags: [“business”, “cool place”],

  latlong: [40.0,72.0]
  }
```

```
db.locations.ensureIndex({latlong:”2d”})
db.locations.find({latlong:{$near:[40,70]}})
```

Places v4

```
location1 = {
  name: "10gen HQ”,
        address: "17 West 18th Street 8th Floor”,
        city: "New York”,
        zip: "10011”,
        latlong: [40.0,72.0],

  tags: [“business”, “cool place”],

        tips: [
        {user:"nosh", time:6/26/2010, tip:"stop by for office hours on Wednesdays from 4-6pm"},
    {.....},
        ]
  }
```

Querying your Places

Creating your indexes

db.locations.ensureIndex({tags:1})

db.locations.ensureIndex({name:1})

db.locations.ensureIndex({latlong:”2d”})

Finding places:

db.locations.find({latlong:{$near:[40,70]}})

With regular expressions:

db.locations.find({name: /^*typeaheadstring*/)

By tag:

db.locations.find({tags: “business”})

Inserting and Updating Locations

Initial data load:

```
db.locations.insert(place1)
```

Using update to Add tips:

```
db.locations.update({name:"10gen HQ"},
  {$push :{tips:
                {user:"nosh", time:6/26/2010,
                 tip:"stop by for office hours on
                 Wednesdays from 4-6"}}}}
```

Requirements

Checkins

- User should be able to ‘check in’ to a location
- Want to be able to generate statistics

Users

```
user1 = {
  name: “nosh”
  email: “nosh@10gen.com”,
  .
  .
  .
  checkins: [{ location: “10gen HQ”,
               ts: 9/20/2010 10:12:00,
               …},
               …
            ]
  }
```

Simple Stats

```
db.users.find({ ‘checkins.location’ : “10gen HQ”)
```

```
db.checkins.find({ ‘checkins.location’ : “10gen HQ”})
                    .sort({ts:-1}).limit(10)
```

```
db.checkins.find({ ‘checkins.location’ : “10gen HQ”,
                    ts: {$gt: midnight}}).count()
```

User Check in

Check-in = 2 ops

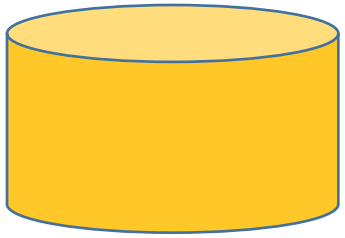
- read location to obtain location id
- Update ($push) location id to user object

Queries: find all locations where a user checked in:

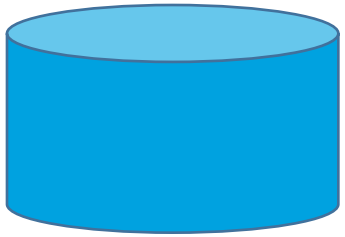
```
checkin_array = db.users.find({..},
                              {checkins:true}).checkins

db.location.find({_id:{$in: checkin_array}})
```

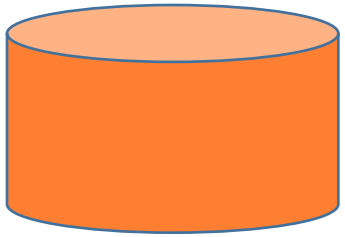
Unsharded Deployment



Configure as a replica set for automated failover

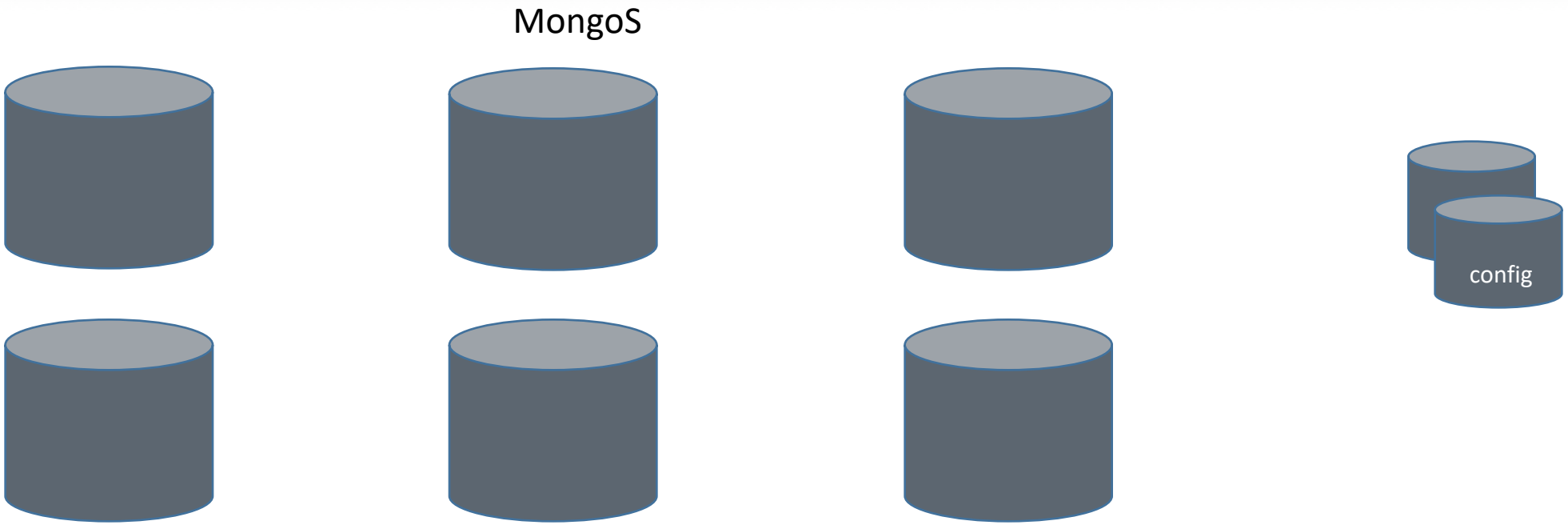


Async replication between nodes



Add more secondaries to scale reads

Sharded Deployment



- Autosharding distributes data among two or more replica sets
- Mongo Config Server(s) handles distribution & balancing
- Transparent to applications

DocumentDB Advantages

Documents are independent units

- makes performance better (related data is read contiguously off disk)
- makes it easier to distribute data across multiple servers while preserving its locality

Application logic is easier to write.

- don't have to translate between objects in your application and SQL queries
- just turn the object model directly into a document

Unstructured data can be stored easily

- a document contains whatever keys and values the application logic requires

Key/Value

Pros

very fast

very scalable

simple model

able to distribute horizontally

Cons

many data structures (objects) can't be easily modeled as key value pairs

Schema-Less

Pros

Schema-less data model is richer than key/value pairs

- eventual consistency
- many are distributed
- still provide excellent performance and scalability

Cons

typically no ACID transactions or joins

What Are You Giving Up?

- joins
- group by
- order by
- ACID transactions
- SQL is sometimes frustrating but still powerful query language
- easy integration with other applications that support SQL

Where Would You Use It?

- For most of us, we work in corporate IT and a LinkedIn or Twitter is not in our future
- Where would I use a NoSQL database?
- Do you have somewhere a large set of uncontrolled, unstructured, data that you are trying to fit into a RDBMS?
  - Log Analysis
  - Social Networking Feeds
  - External feeds from partners (EAI)
  - Data that is not easily analyzed in a RDBMS such as time-based data
  - Large data feeds that need to be massaged before entry into an RDBMS