Background

- Relational databases (Banking Systems, Employee DB, School DB)
- Web-based applications caused spikes
  - Social media sites (Facebook, Twitter)
  - Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Hooking RDBMS to web-based application becomes trouble

What is NOSQL?

- Key features (advantages):
  - non-relational
  - don't require schema
  - data are replicated to multiple nodes (so, identical & fault-tolerant) and can be partitioned:
    - down nodes easily replaced
    - no single point of failure
  - horizontal scalability
  - cheap, easy to implement (open-source)
  - massive write performance
  - Fast access

What is NOSQL?

- Disadvantages:

What is NOSQL?

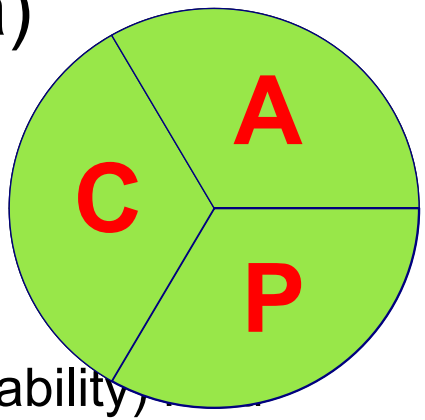
- Disadvantages:
  - Don't fully support relational features
    - no join, group by, order by operations (except within partitions)
    - no referential integrity constraints across partitions
  - No declarative query language (e.g., SQL) → more procedural/navigational
  - Relaxed ACID (see CAP theorem) → fewer guarantees
  - No easy integration with other applications that support SQL

Who is using them?

Scaling Up Vs. Scaling Out

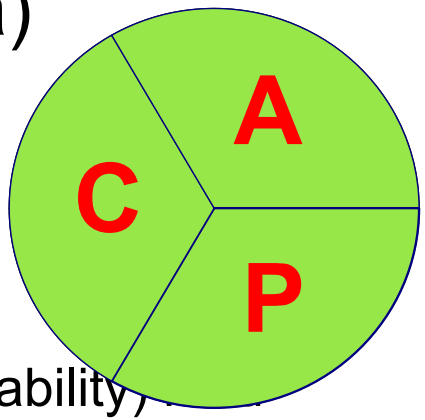
CAP Theorem

- Suppose three properties of a distributed system (sharing data)
  - **Consistency:**
    - all copies have same value
  - **Availability:**
    - reads and writes always succeed
  - **Partition-tolerance:**
    - system properties (consistency and/or availability) hold even when network failures prevent some machines from communicating with others

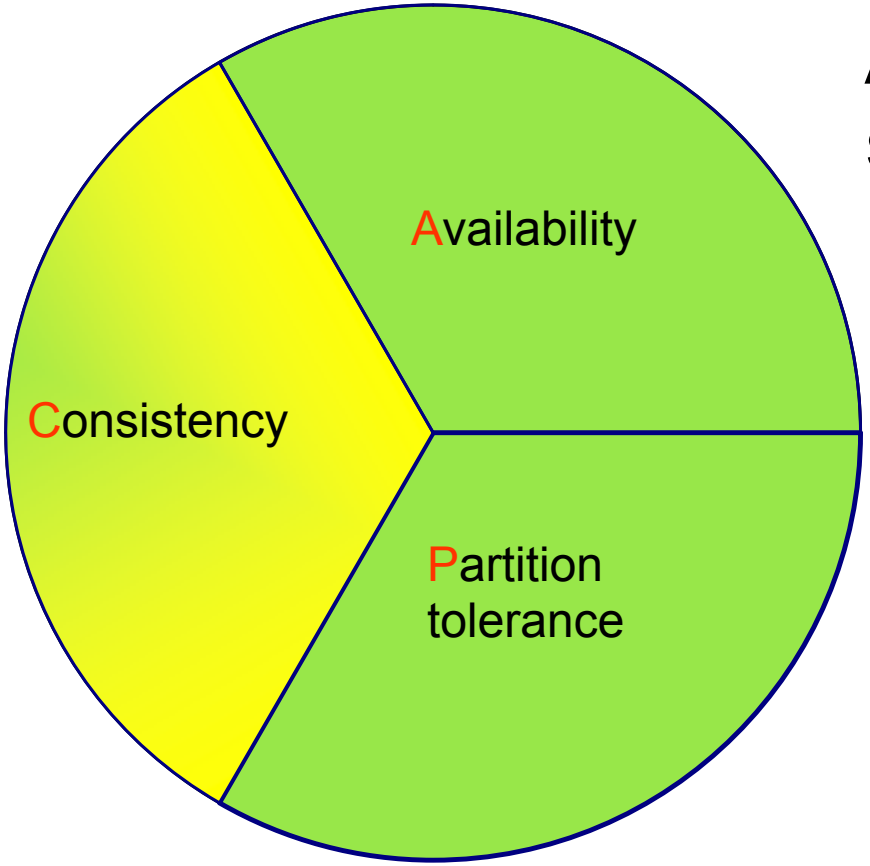


CAP Theorem

- Suppose three properties of a distributed system (sharing data)
  - **Consistency:**
    - all copies have same value
  - **Availability:**
    - reads and writes always succeed
  - **Partition-tolerance:**
    - system properties (consistency and/or availability) hold even when network failures prevent some machines from communicating with others



CAP Theorem



All client always have the same view of the data

CAP Theorem

- **Consistency**
  - 2 types of consistency:
    1. Strong consistency – ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability)
    2. Weak consistency – BASE (**B**asically **A**vailable **S**oft-state **E**ventual consistency)

CAP Theorem

- **ACID**
  - A DBMS is expected to support “ACID transactions,” processes that are:
  - **A**tomicity: either the whole process is done or none is
  - **C**onsistency: only valid data are written
  - **I**solation: one operation at a time
  - **D**urability: once committed, it stays that way
- **CAP**
  - **C**onsistency: all data on cluster has the same copies
  - **A**vailability: cluster always accepts reads and writes
  - **P**artition tolerance: guaranteed properties are maintained even when network failures prevent some machines from communicating with others

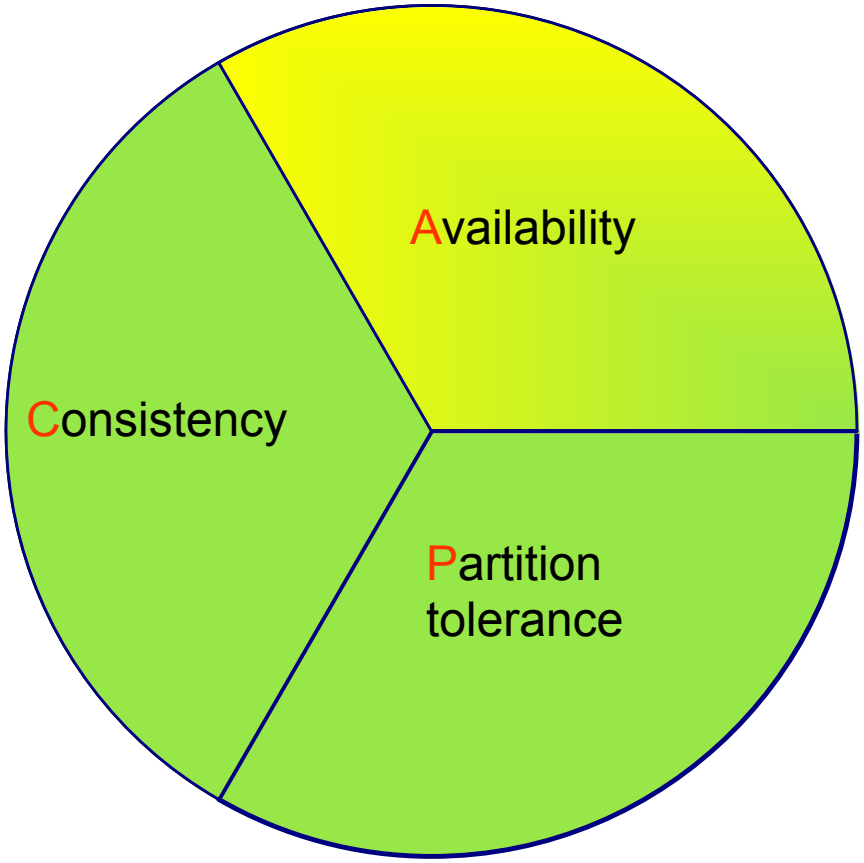
CAP Theorem

- A consistency model determines rules for visibility and apparent order of updates
- Example:
  - Row X is replicated on nodes M and N
  - Client A writes row X to node N
  - Some period of time t elapses
  - Client B reads row X from node M
  - **Does client B see the write from client A?**
  - Consistency is a continuum with tradeoffs
  - **For NOSQL, the answer would be: "maybe"**
  - CAP theorem states: *"strong consistency can't be achieved at the same time as availability and partition-tolerance"*

CAP Theorem

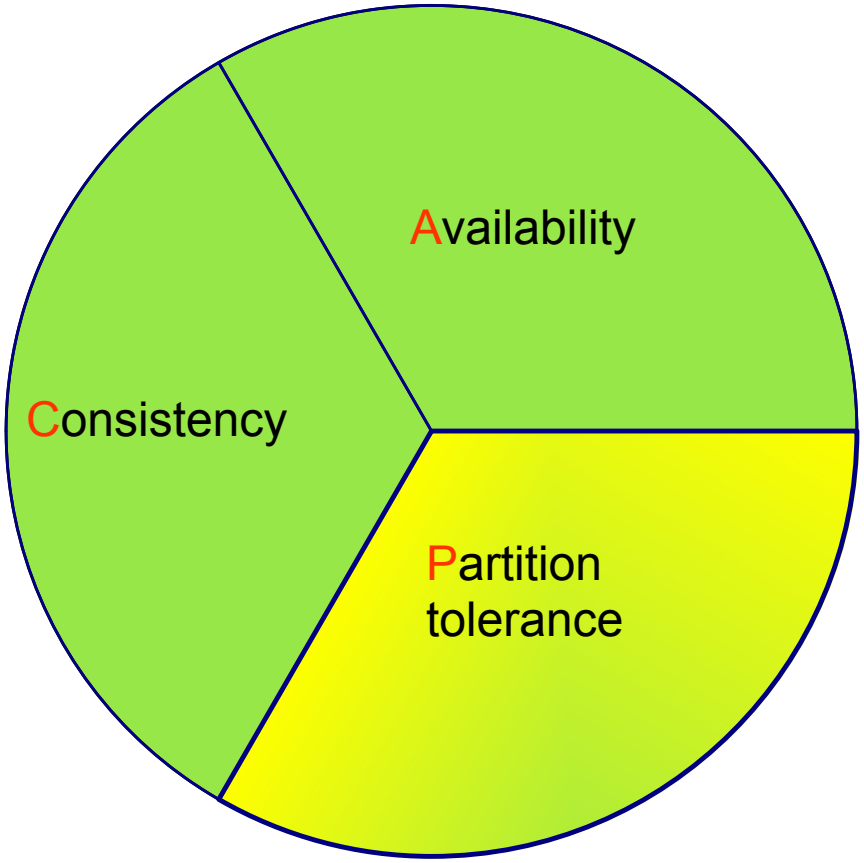
- Eventual consistency
  - When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- Cloud computing
  - ACID is hard to achieve, moreover, it is not always required, e.g. for blogs, status updates, product listings, etc.

CAP Theorem



Each client always can read and write.

CAP Theorem



A system can continue to operate in the presence of a network partitions

NOSQL categories

1. **Key-value**
   - Example: DynamoDB, Voldermort.
2. **Document-based**
   - Example: MongoDB, CouchDB
3. **Column-based**
   - Example: BigTable, Cassandra, Hbased
4. **Graph-based**
   - Example: Neo4J.

- "No-schema" is a common characteristics of most NOSQL storage systems
- Provide "flexible" data types

Large Graph Processing

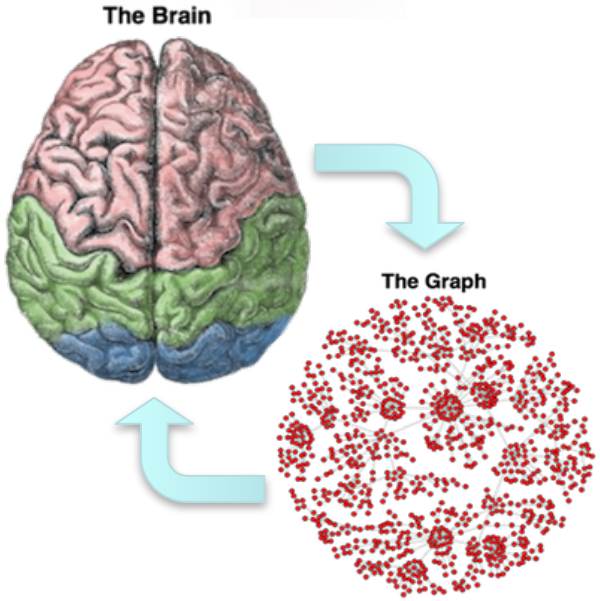
Social Graph

1.19 billion monthly active users as of September 30, 2013

Road Network Graph

400GB of Road Network Data

Brain Graph



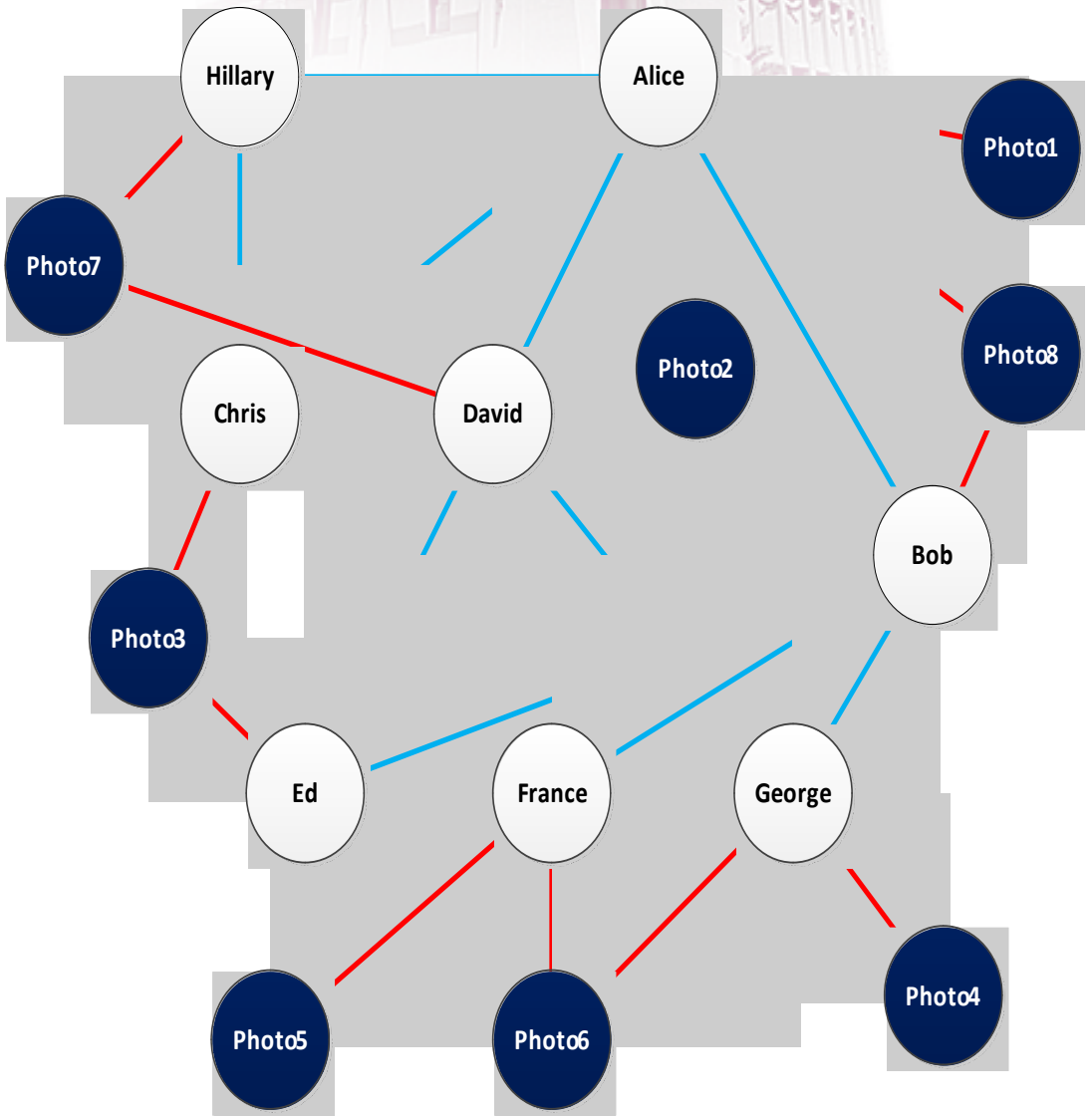
85 Billion neurons in the brain nervous system

–Social network

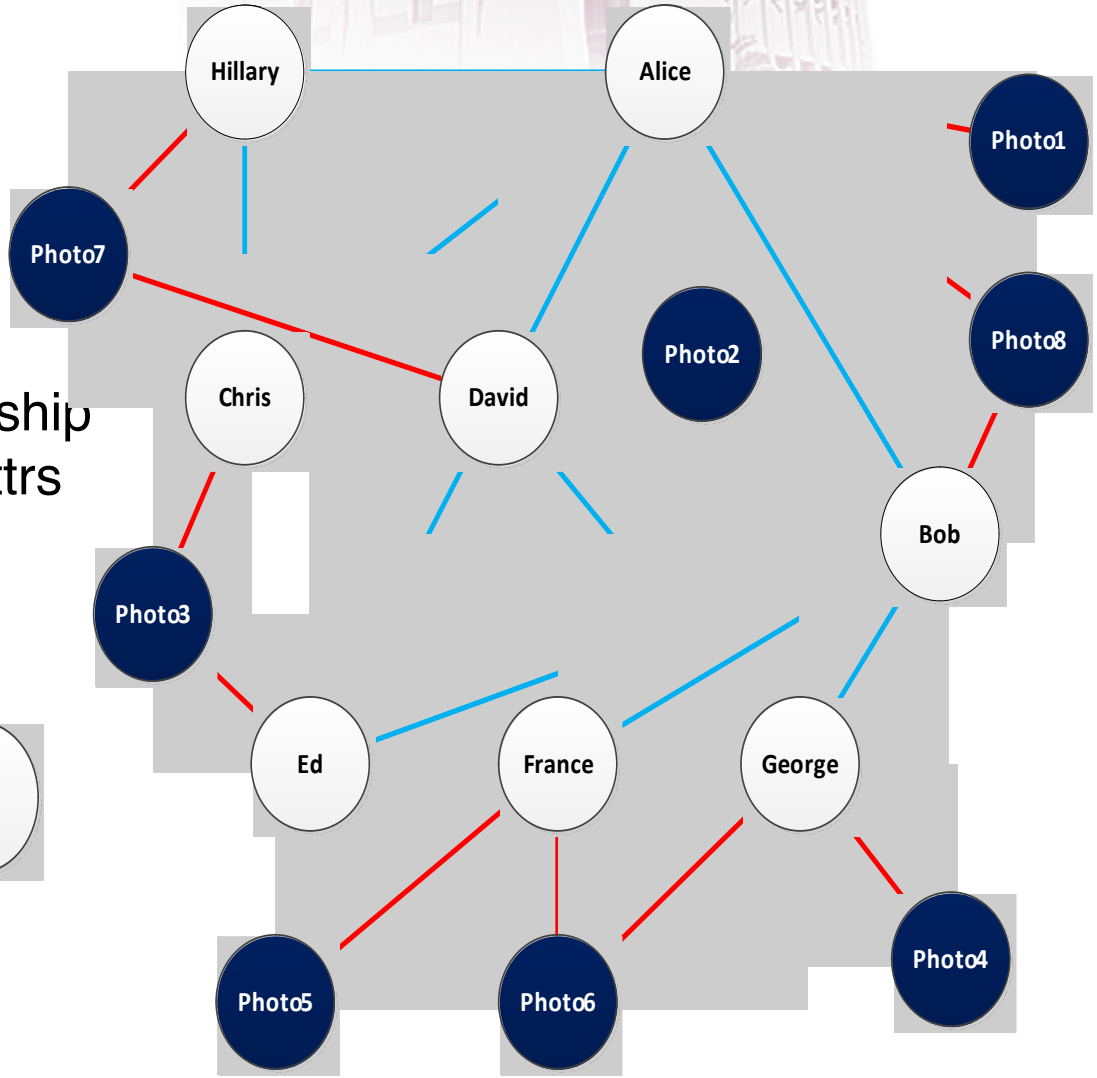
- Persons
- Friends
- Photos

–Queries

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

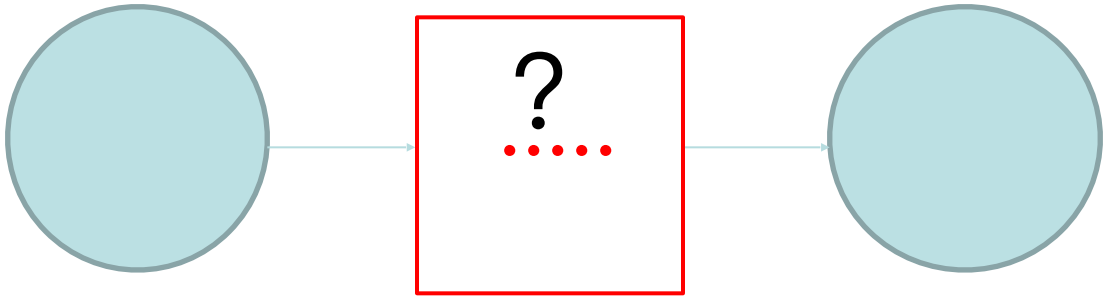


- –Attributed multi-graph
- –Node
  - Represent entities
  - ID, type, attributes
- –Edge
  - Represent binary relationship
  - Type, direction, weight, attrs

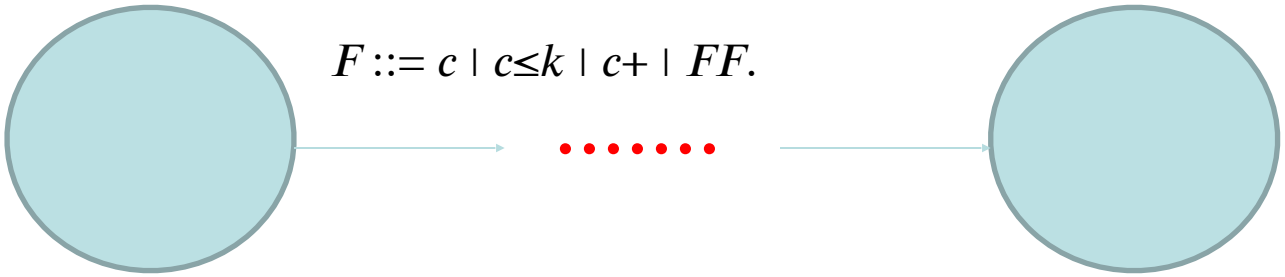


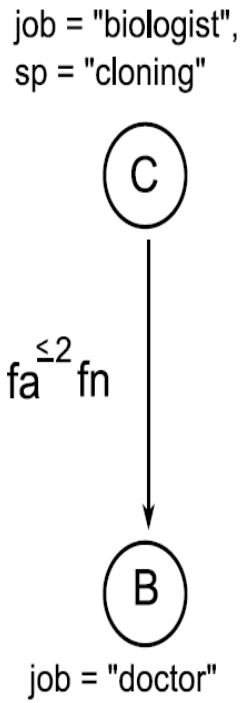
App

Graph Reachability Query



- Adding regular expressions to graph reachability





Graph Reachability Queries

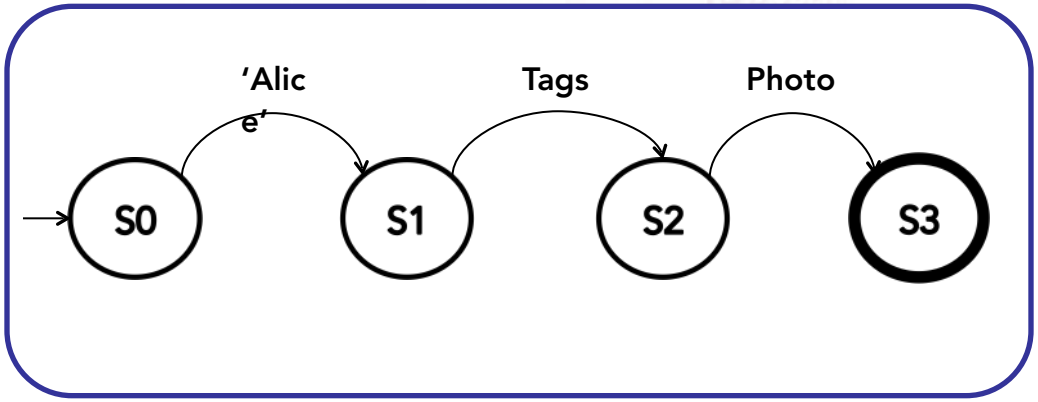
–Query is a regular expression

- Sequence of node and edge predicates

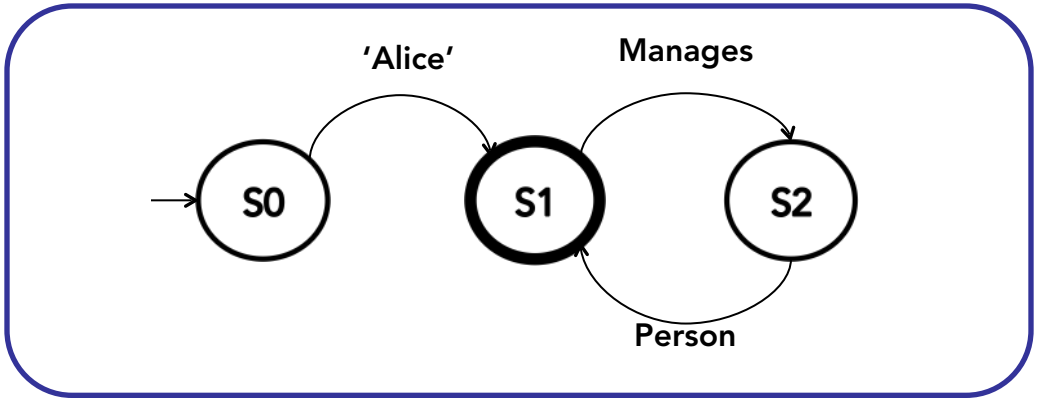
1. Hello world in reachability
   - Photo-Tags-'Alice'
   - Search for path with **node:** type=Photo, **edge:** type=Tags, **node:** id='Alice'
2. Attribute predicate
   - Photo{date.year='2012'}-Tags-'Alice'
3. Or
   - (Photo | video)-Tags-'Alice'
4. Closure for path with arbitrary length
   - 'Alice'(-Manages-Person)*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

Compile into Algebraic Query Plan

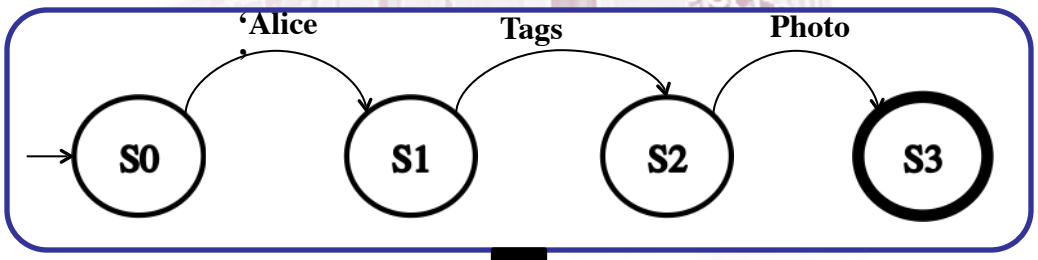
'Alice'-Tags-Photo



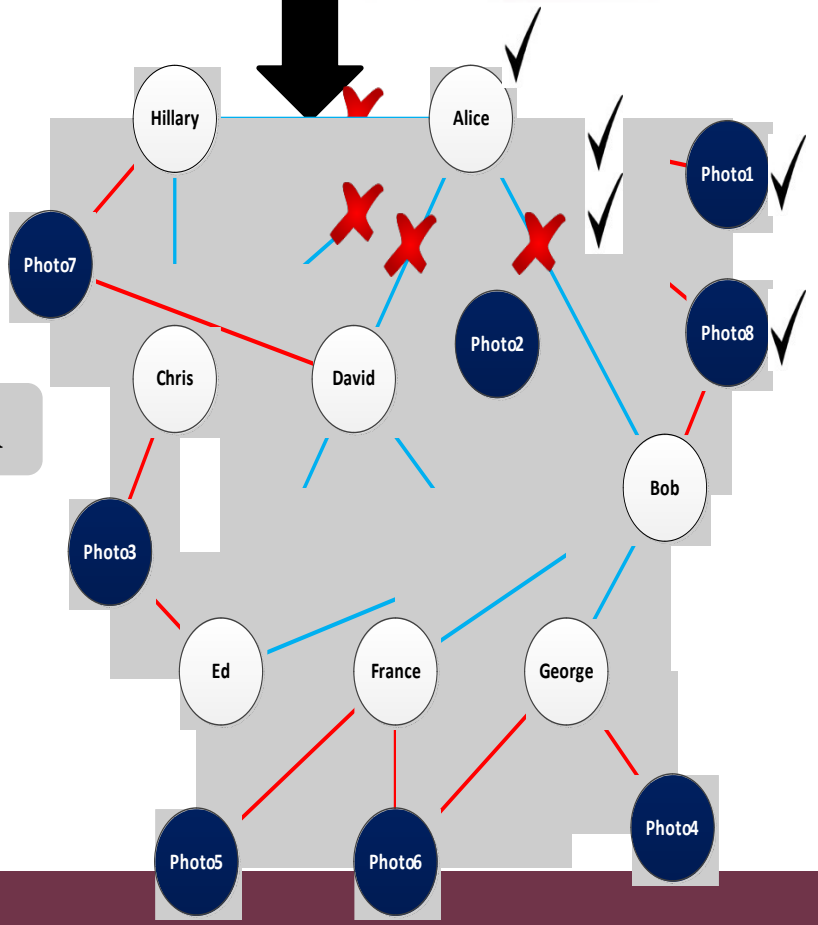
'Alice'(-Manages-Person)*



‘Alice’-Tags-Photo



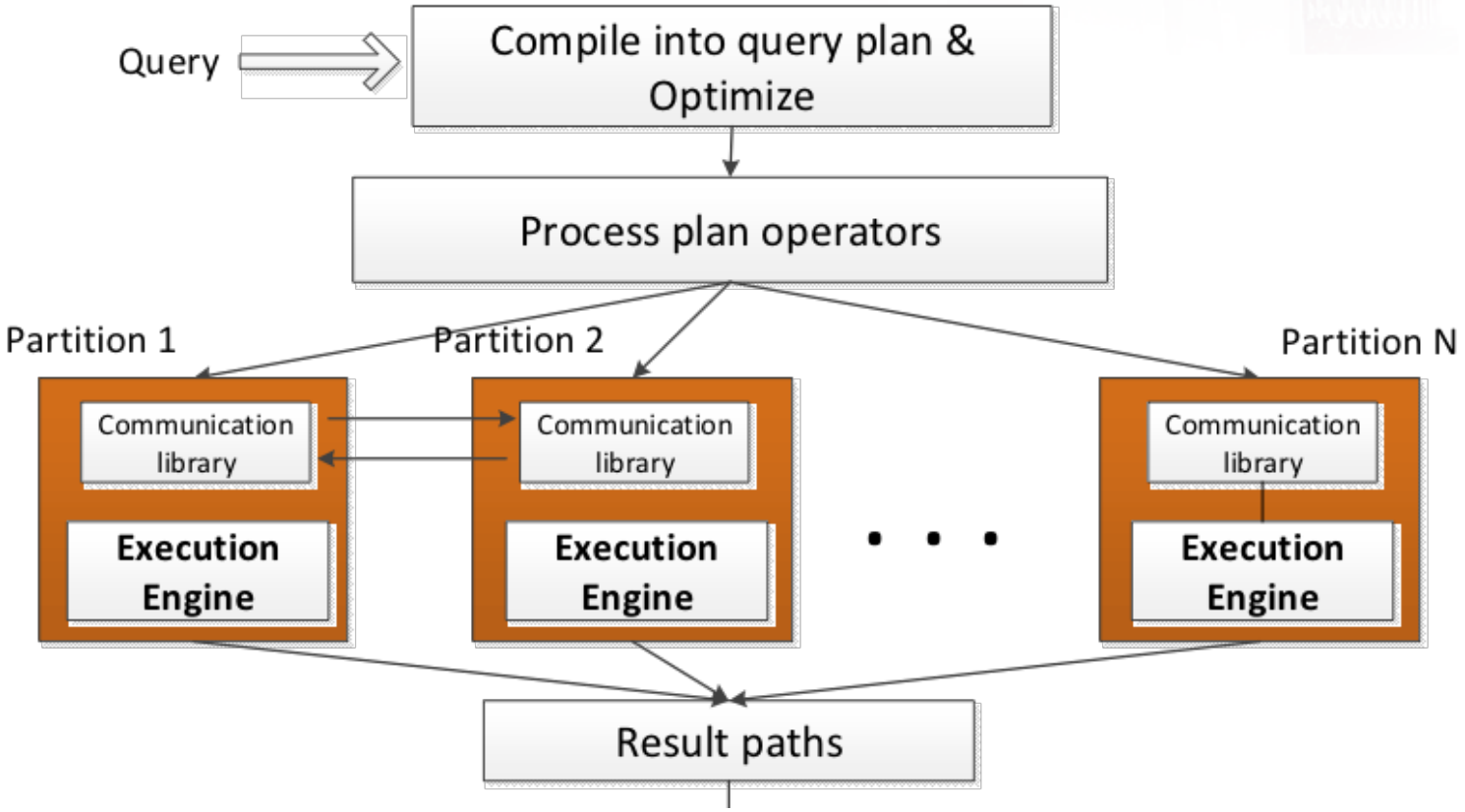
Centralized Query Execution



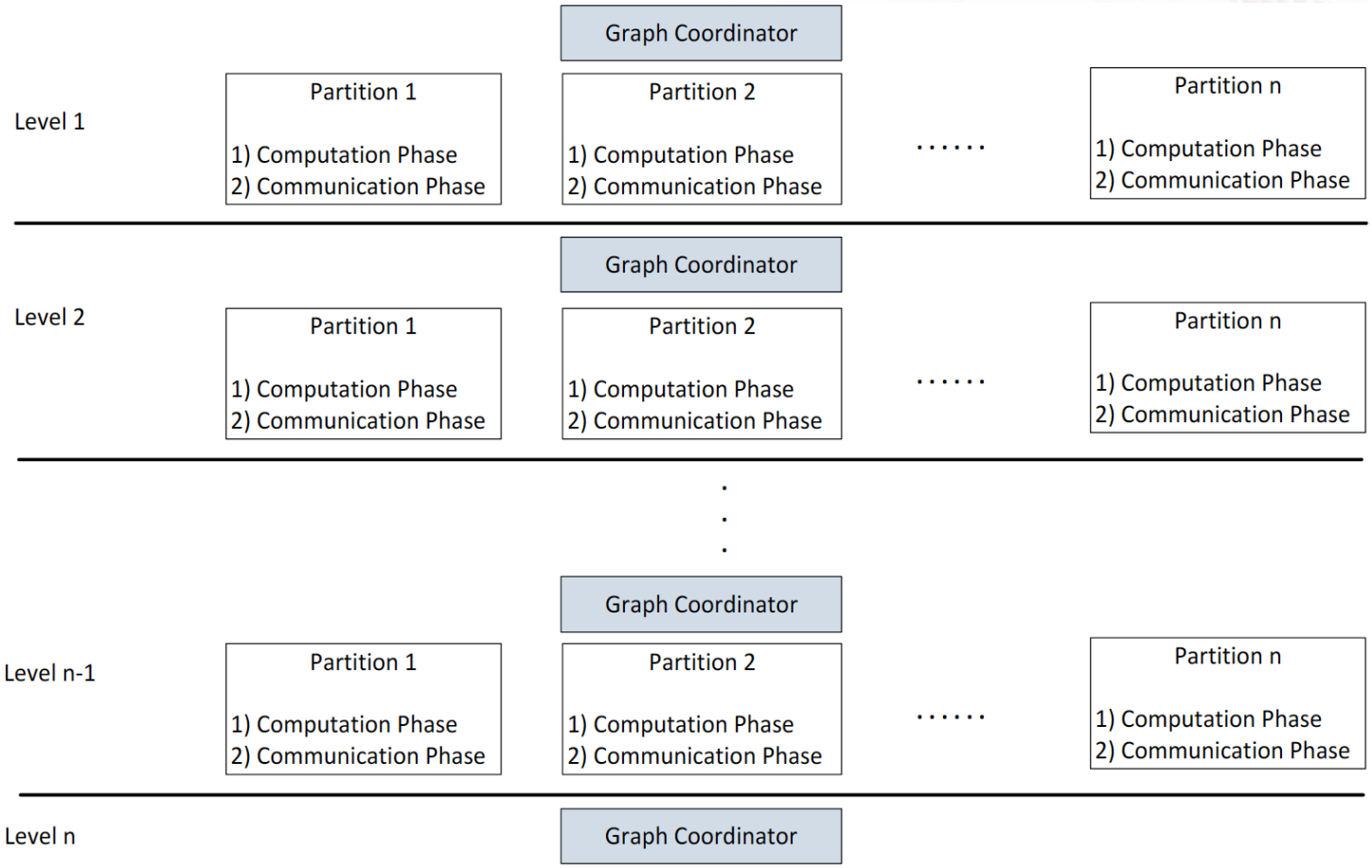
Breadth First Search

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

Distributed Query Execution

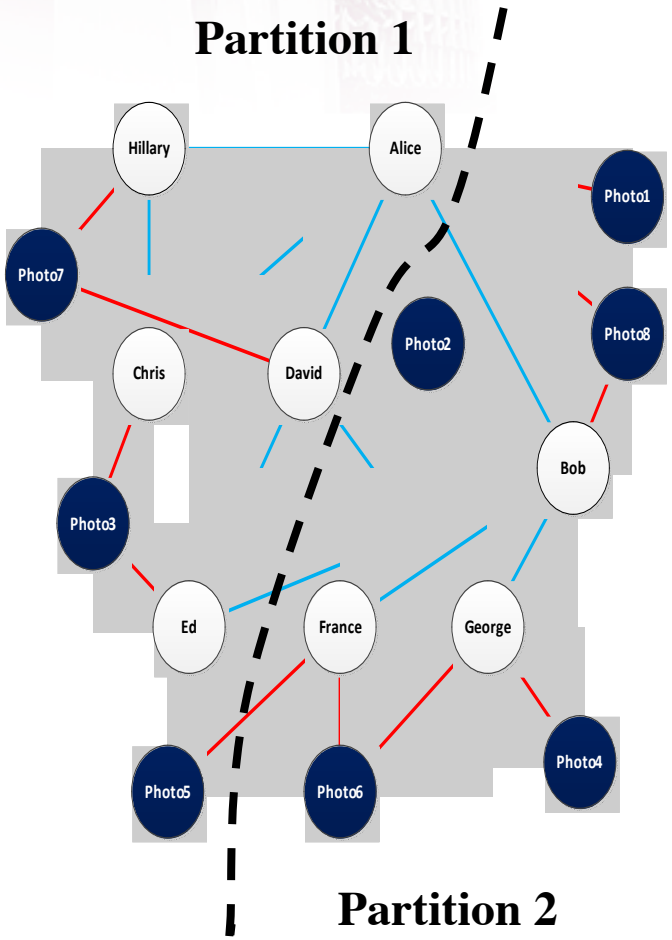


Distributed BFS Traversal

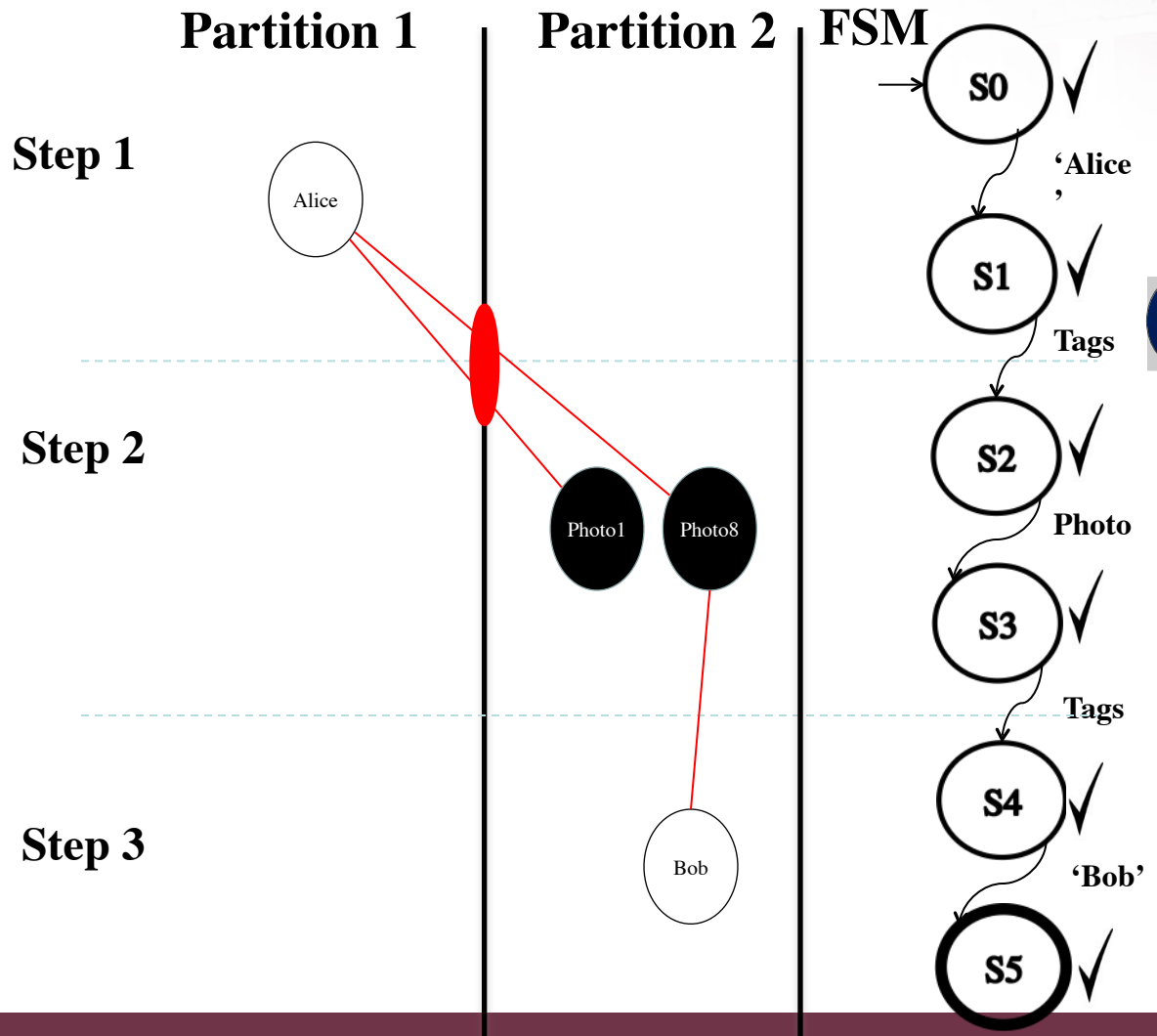


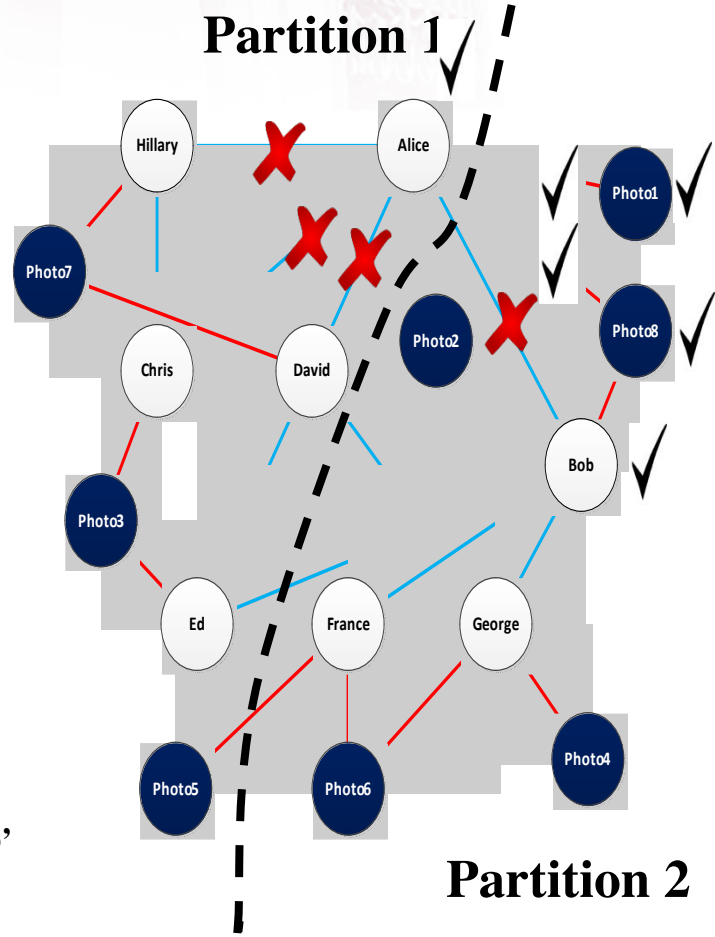
Distributed Query Execution

'Alice'-Tags-Photo-Tags-'Bob'



‘Alice’-Tags-Photo-Tags-‘Bob’

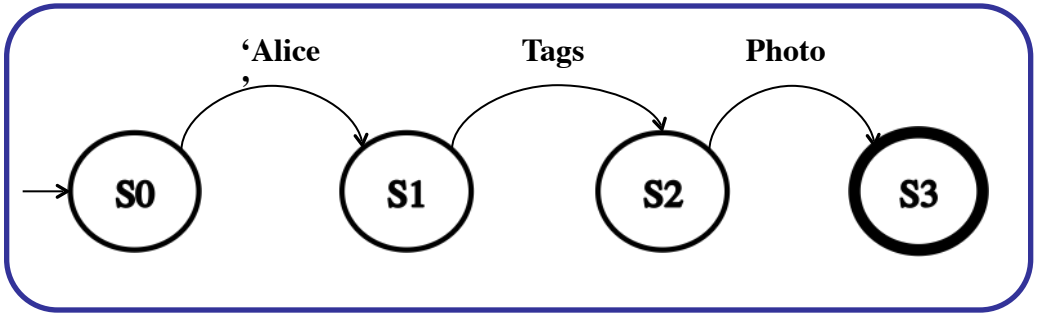




Algebraic Operators

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

‘Alice’-Tags-Photo



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: $O(n^3)$

Questions

NoSQL Database Systems

What kinds of NoSQL

- 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
  - Service must be accessible at all times
- Scalability
  - Service must scale well to handle customer growth & machine growth
- Failure Tolerance
  - With thousands of machines, failure is the default case
- Manageability
  - Must not cost a fortune to maintain

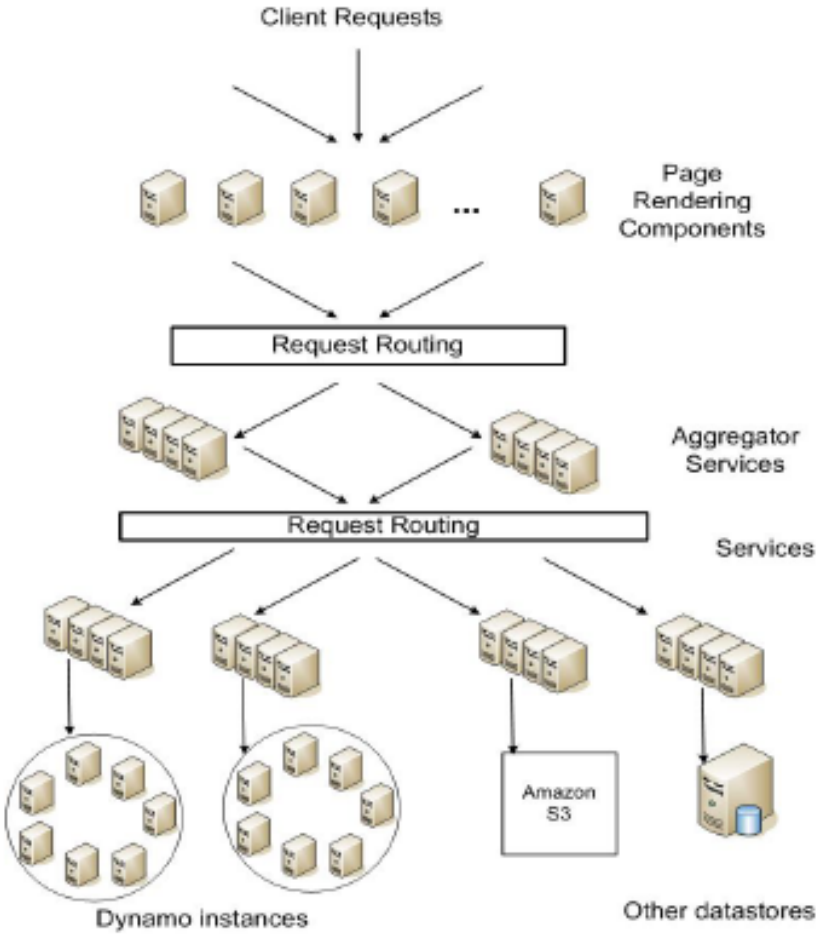
Dynamo's Assumptions

- 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 (SLAs)

- 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 (SoA)



Design Decisions

- Incremental Scalability
  - Must be able to add nodes on-demand with minimal impact
- Load Balancing

Design Decisions

- Replication
  - Must do conflict-resolution
  - Two questions:
    - 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

- Only two operations
- put (key, context, object)
  - key: primary key associated with data object
  - context: vector clocks and history (needed for merging)
  - object: data to store
- get (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

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

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

Google File System

- Large-scale distributed “filesystem”
- Master: responsible for metadata
- Chunk servers: responsible for reading and writing large chunks of data
- Chunks replicated on 3 machines, master responsible for ensuring replicas exist

Chubby

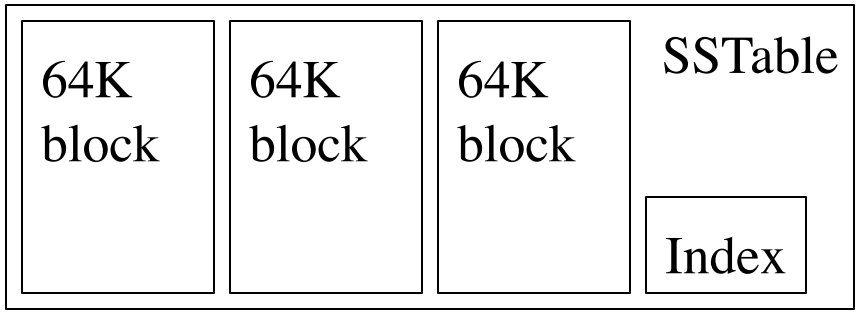
- {lock/file/name} service
- Coarse-grained locks, can store small amount of data in a lock

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

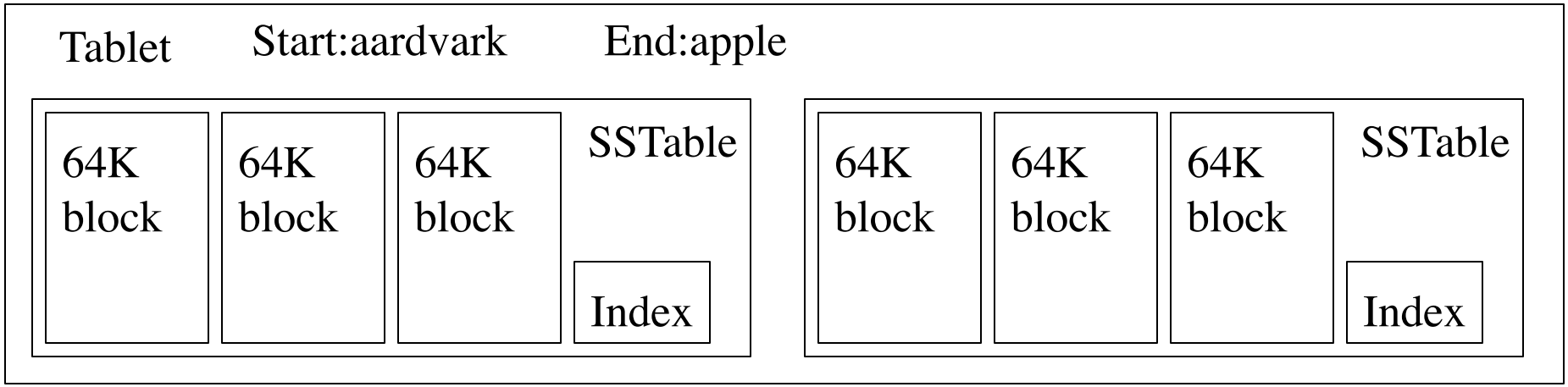
SSTable

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



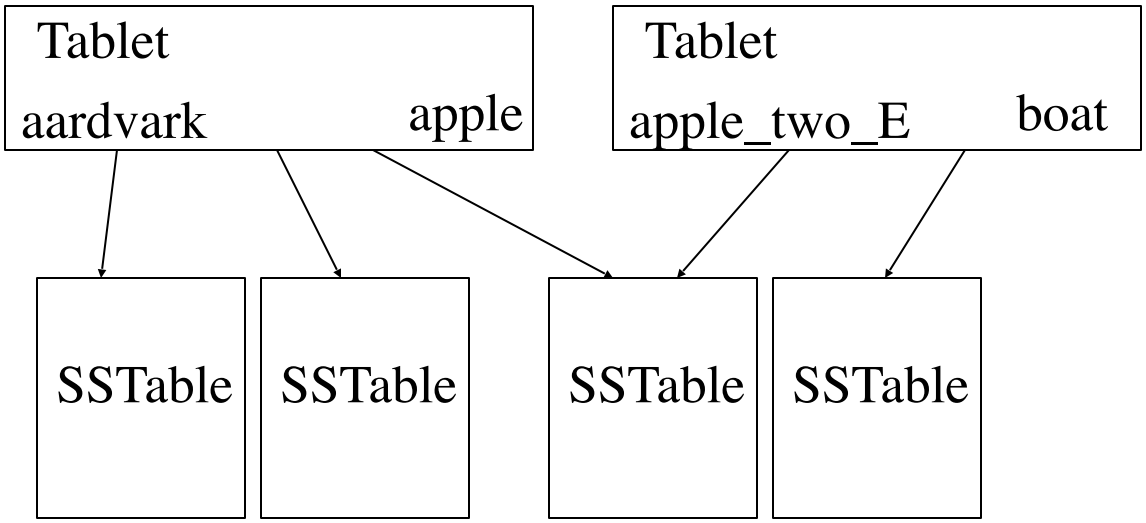
Tablet

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



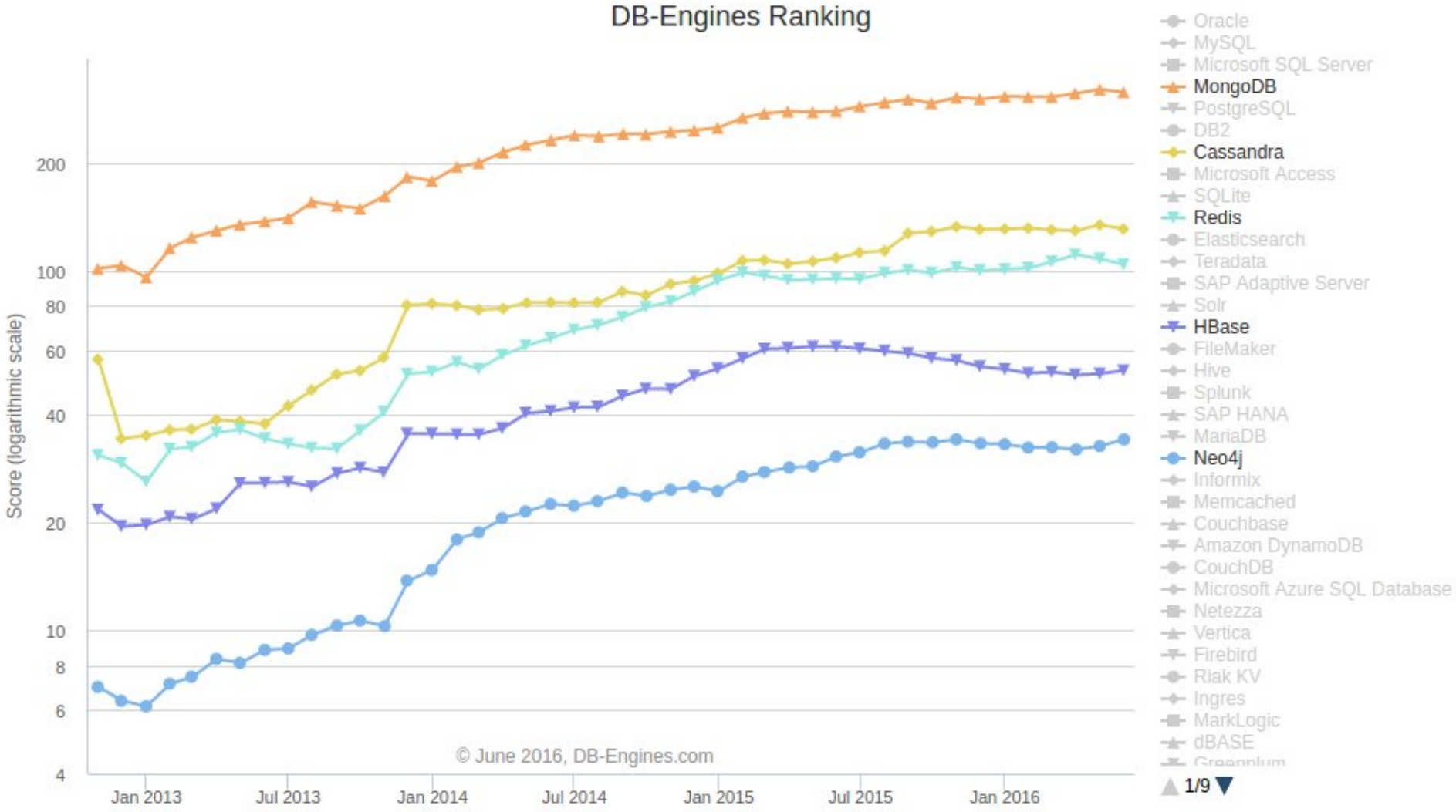
Table

- 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 at


DB-Engines Ranking
Score (logarithmic scale)
200
100
80
60
40
20
10
8
6
4
Jan 2013
Jul 2013
Jan 2014
Jul 2014
Jan 2015
Jul 2015
Jan 2016
© June 2016, DB-Engines.com
Oracle
MySQL
Microsoft SQL Server
MongoDB
PostgreSQL
DB2
Cassandra
Microsoft Access
SQLite
Redis
Elasticsearch
Teradata
SAP Adaptive Server
Solr
HBase
FileMaker
Hive
Splunk
SAP HANA
MariaDB
Neo4j
Informix
Memcached
Couchbase
Amazon DynamoDB
CouchDB
Microsoft Azure SQL Database
Netezza
Vertica
Firebird
Riak KV
Ingres
MarkLogic
dBASE
Greenplum
1/9

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

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

- 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

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

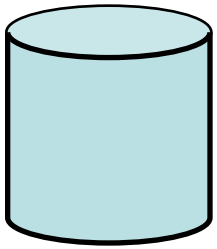
```
read location to obtain location id
Update ($push) location id to user object
```

Queries: find all locations where a user checked in:

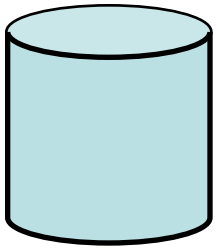
```
checkin_array = db.users.find({..},
                              {checkins:true}).checkins

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

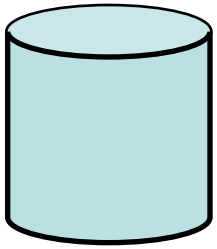
Unsharded Deployment



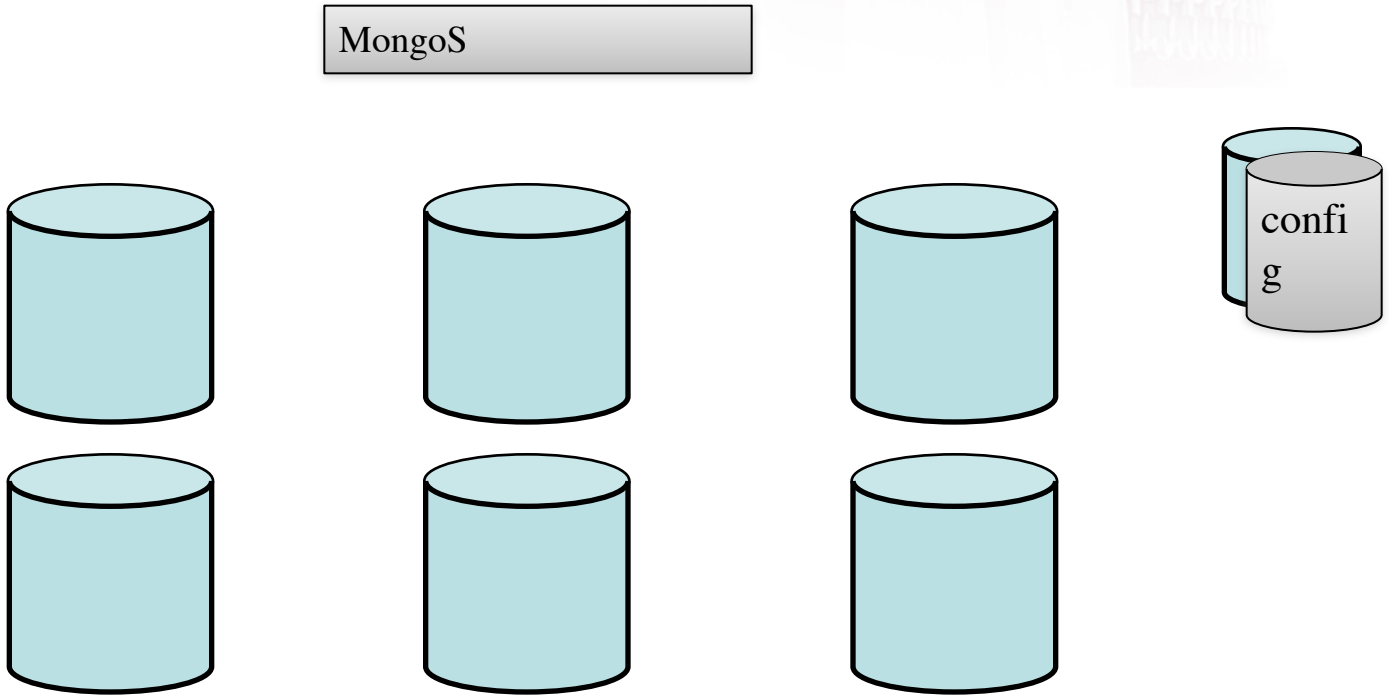
- Configure as a replica set for automated failover
- Async replication between nodes



- 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 which makes performance better** (related data is read contiguously off disk) and makes it easier to distribute data across multiple servers while preserving its locality.
- **Application logic is easier to write**. You don't have to translate between objects in your application and SQL queries, you can just turn the object model directly into a document.
- **Unstructured data can be stored easily**, since 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 am I 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 I 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 (many firms hooked in through Facebook or Twitter)
  - 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

Questions

Key-Value Storage

- Interface
  - **put**(key, value); // insert/write “value” associated with “key”
  - value = **get**(key); // get/read data associated with “key”

- Abstraction used to implement
  - A simpler and more scalable “database”
  - Content-addressable network storage (CANs)

- Can handle large volumes of data, e.g., PBs
  - Need to distribute data over hundreds, even thousands of machines
  - Designed to be faster with lower overhead (additional storage) than conventional DBMSes.

Database Attributes

Databases require 4 properties:

- **Atomicity:** When an update happens, it is “all or nothing”
- **Consistency:** The state of various tables much be consistent (relations, constraints) at all times.
- **Isolation:** Concurrent execution of transactions produces the same result as if they occurred sequentially.
- **Durability:** Once committed, the results of a transaction persist against various problems like power failure etc.

These properties ensure that data is protected even with complex updates and system failures.

CAP Theorem (Brewer, Gilbert, Lynch)

But we also have the CAP theorem for distributed systems:

Consistency: All nodes have the same view of the data

Availability: Every request receives a response of success or failure.

Partition Tolerance: System continues even with loss of messages or part of the data nodes.

The theorem states that **you cannot achieve all three at once**.

Many systems therefore strive to implement two of the three properties. Key-Value stores often do this.

KV-stores and Relational Tables

KV-stores seem very simple indeed. They can be viewed as two-column (key, value) tables with a single key column.

But they can be used to implement more complicated relational tables:

State	ID	Population	Area	Senator_1
Alabama	1	4,822,023	52,419	Sessions
Alaska	2	731,449	663,267	Begich
Arizona	3	6,553,255	113,998	Boozman
Arkansas	4	2,949,131	53,178	Flake
California	5	38,041,430	163,695	Boxer
Colorado	6	5,187,582	104,094	Bennet
…	…			

Index

KV-stores and Relational Tables

The KV-version of the previous table includes one table indexed by the actual key, and others by an ID.

State	ID
Alabama	1
Alaska	2
Arizona	3
Arkansas	4
California	5
Colorado	6
…	…

ID	Population
1	4,822,023
2	731,449
3	6,553,255
4	2,949,131
5	38,041,430
6	5,187,582
…	…

ID	Area
1	52,419
2	663,267
3	113,998
4	53,178
5	163,695
6	104,094
…	…

ID	Senator_1
1	Sessions
2	Begich
3	Boozman
4	Flake
5	Boxer
6	Bennet
…	…

KV-stores and Relational Tables

You can add indices with new KV-tables:

Thus KV-tables are used for **column-based storage**, as opposed to row-based storage typical in older DBMS.

State	ID
Alabama	1
Alaska	2
Arizona	3
Arkansas	4
California	5
Colorado	6
…	…

Index

ID	Population
1	4,822,023
2	731,449
3	6,553,255
4	2,949,131
5	38,041,430
6	5,187,582
…	…

…

Senator_1	ID
Sessions	1
Begich	2
Boozman	3
Flake	4
Boxer	5
Bennet	6
…	…

Index_2

OR: the value field can contain complex data (next page):

Key-Value Storage

- Interface
  - **put**(key, value); // insert/write “value” associated with “key”
  - value = **get**(key); // get/read data associated with “key”

- Abstraction used to implement
  - A simpler and more scalable “database”
  - Content-addressable network storage (CANs)

- Can handle large volumes of data, e.g., PBs
  - Need to distribute data over hundreds, even thousands of machines
  - Designed to be faster with lower overhead (additional storage) than conventional DBMSes.

Key-Values: Examples

- Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)

- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., photos, friends, …)

- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

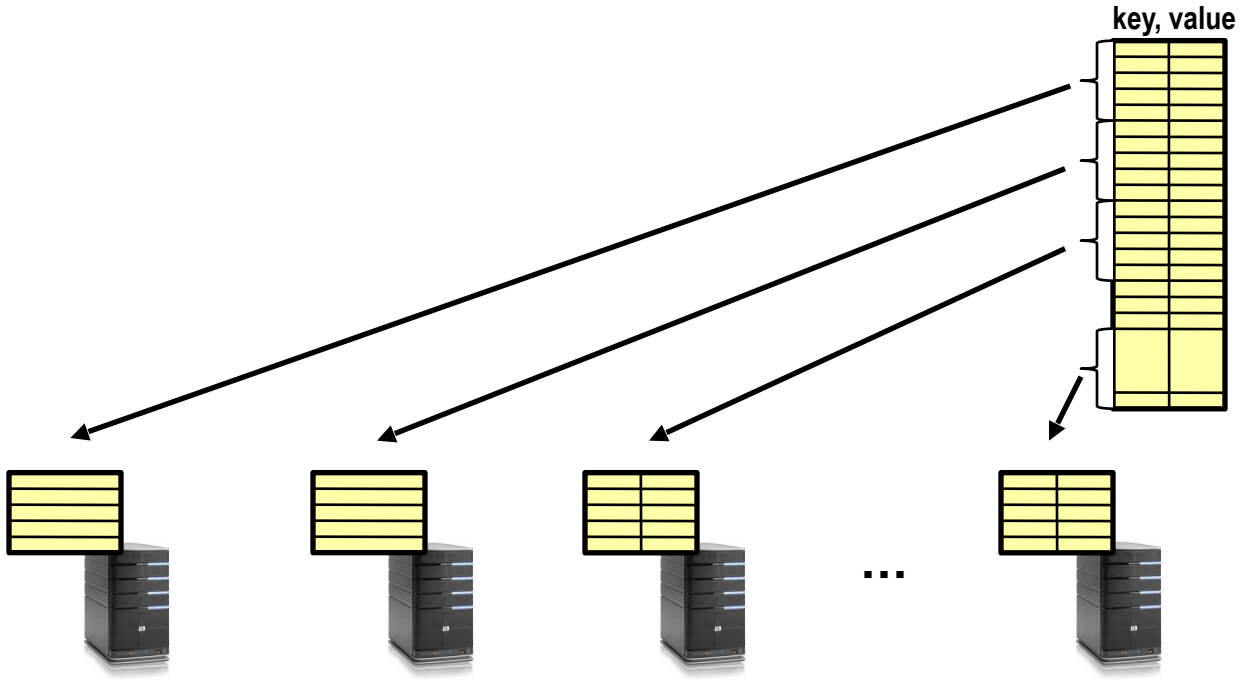
- Distributed file systems
  - Key: Block ID
  - Value: Block

System Examples

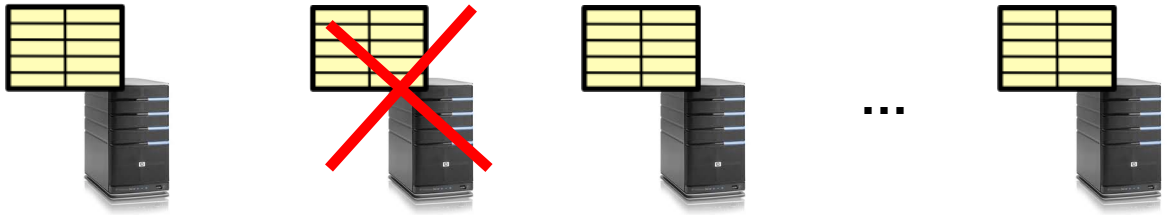
- **Google File System, Hadoop Dist. File Systems (HDFS)**
- **Amazon**
  - Dynamo: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- **BigTable/HBase/Hypertable:** distributed, scalable data storage
- **Cassandra**: “distributed data management system” (Facebook)
- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)
- **eDonkey/eMule:** peer-to-peer sharing system

Key-Value Store

- Also called a Distributed Hash Table (DHT)
- Main idea: partition set of key-values across many machines



Challenges



- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Consistency:** maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to several GB/s

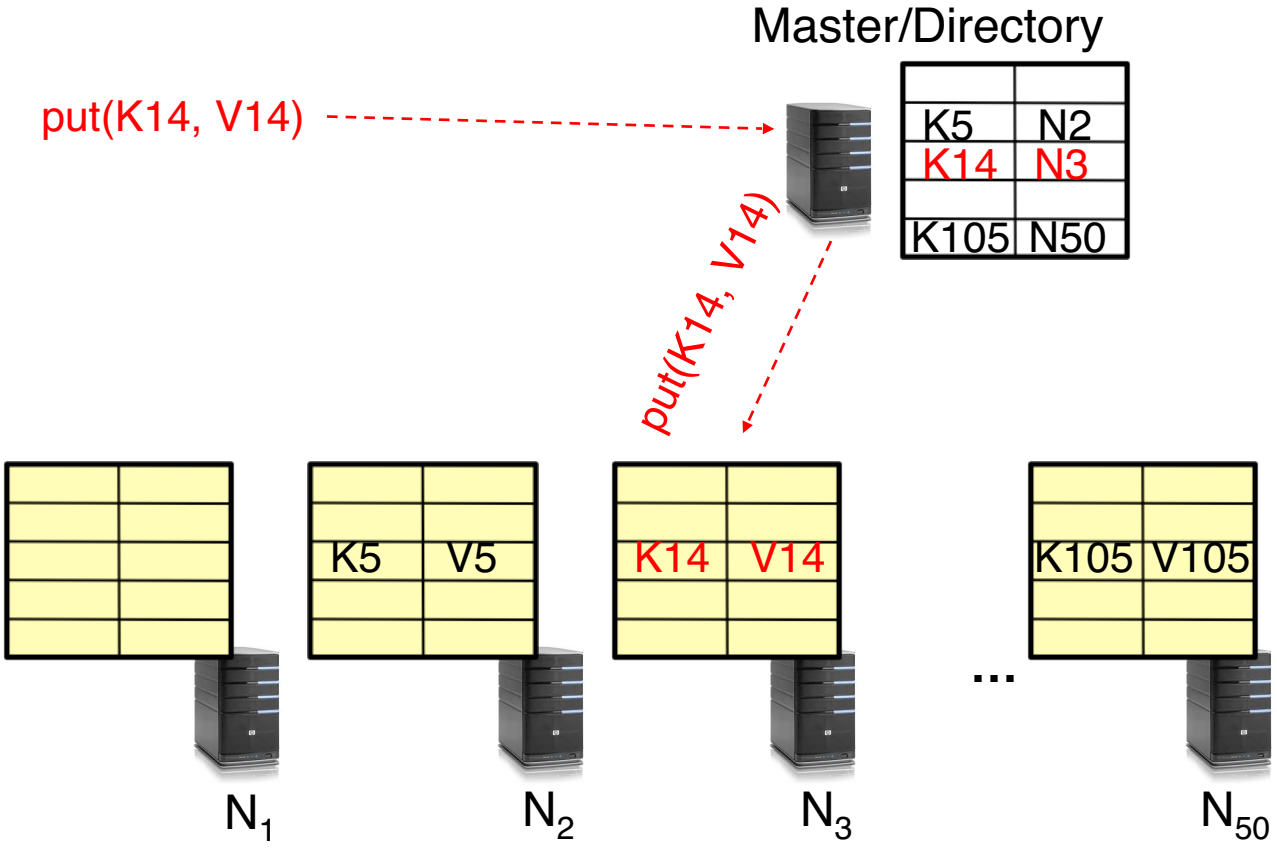
Key Questions

- put(key, value): where do you store a new (key, value) tuple?
- get(key): where is the value associated with a given “key” stored?

- And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency

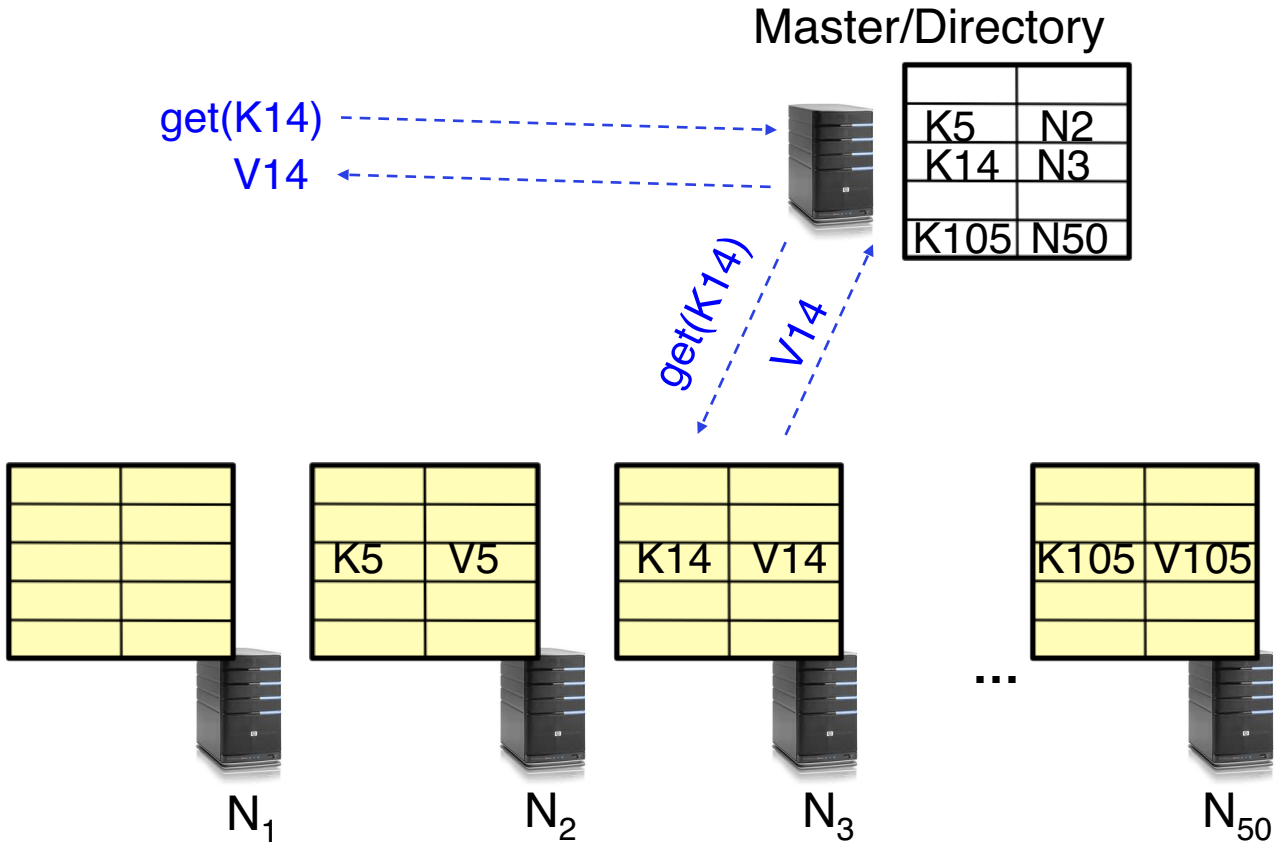
Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



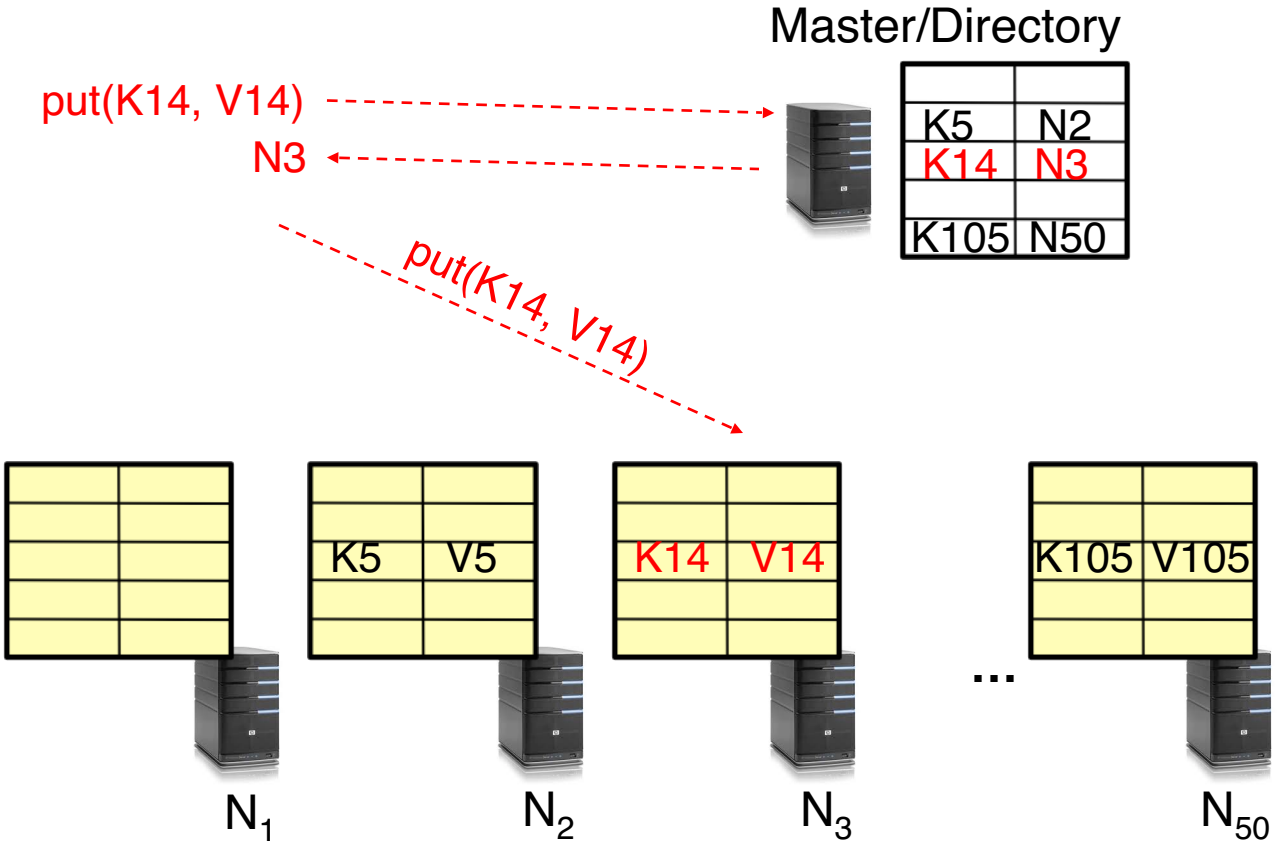
Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



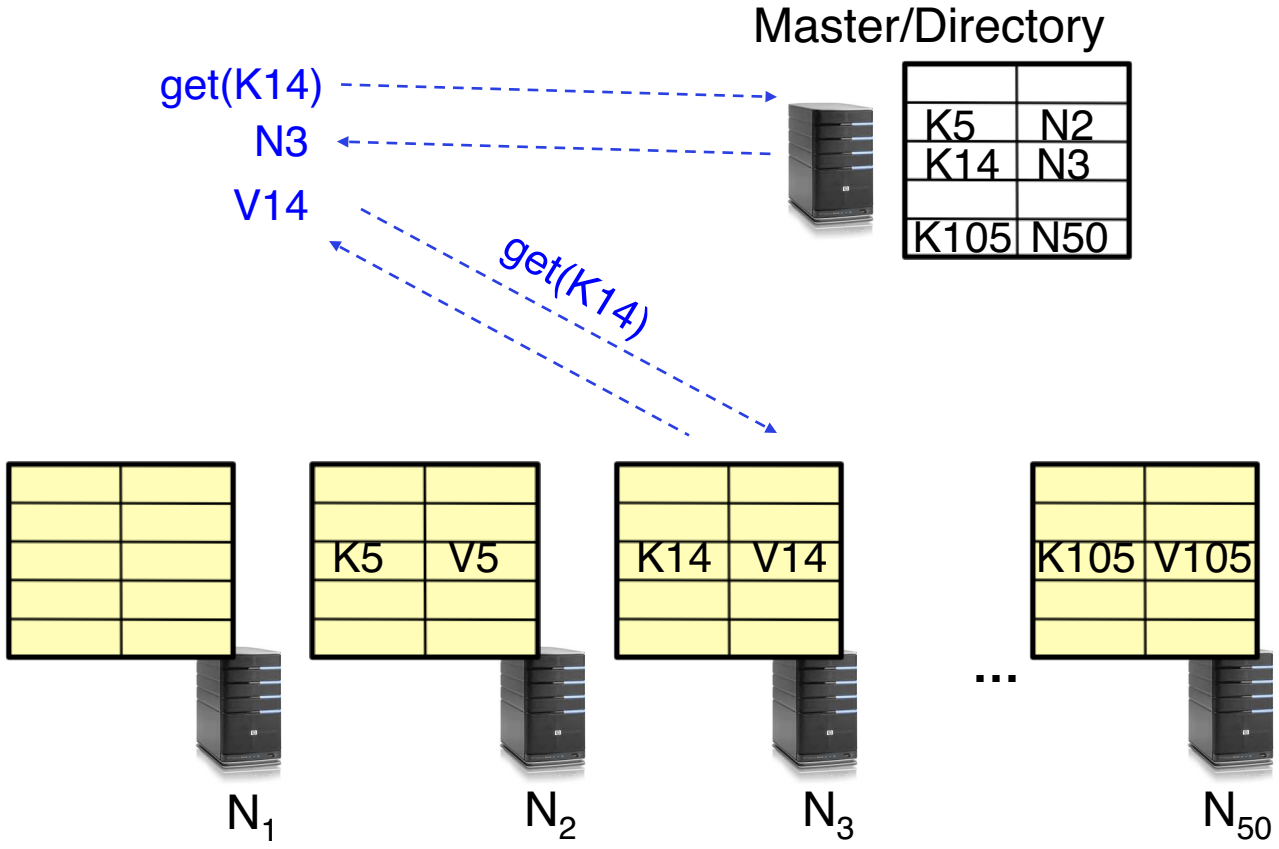
Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node

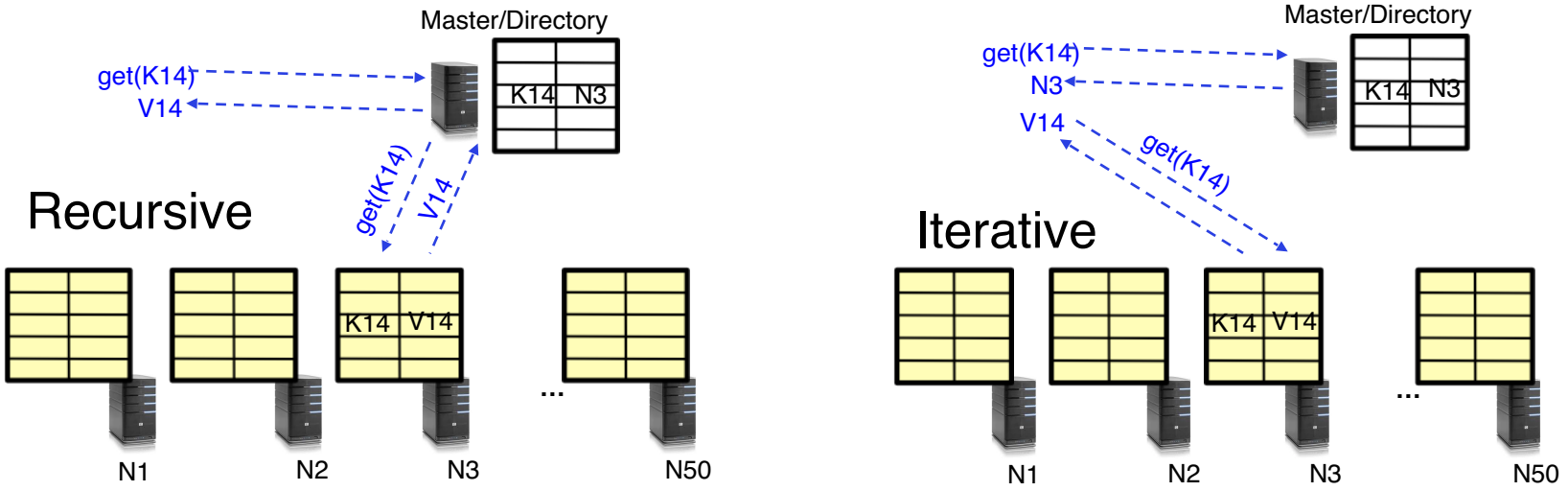


Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query**
  - Return node to requester and let requester contact node



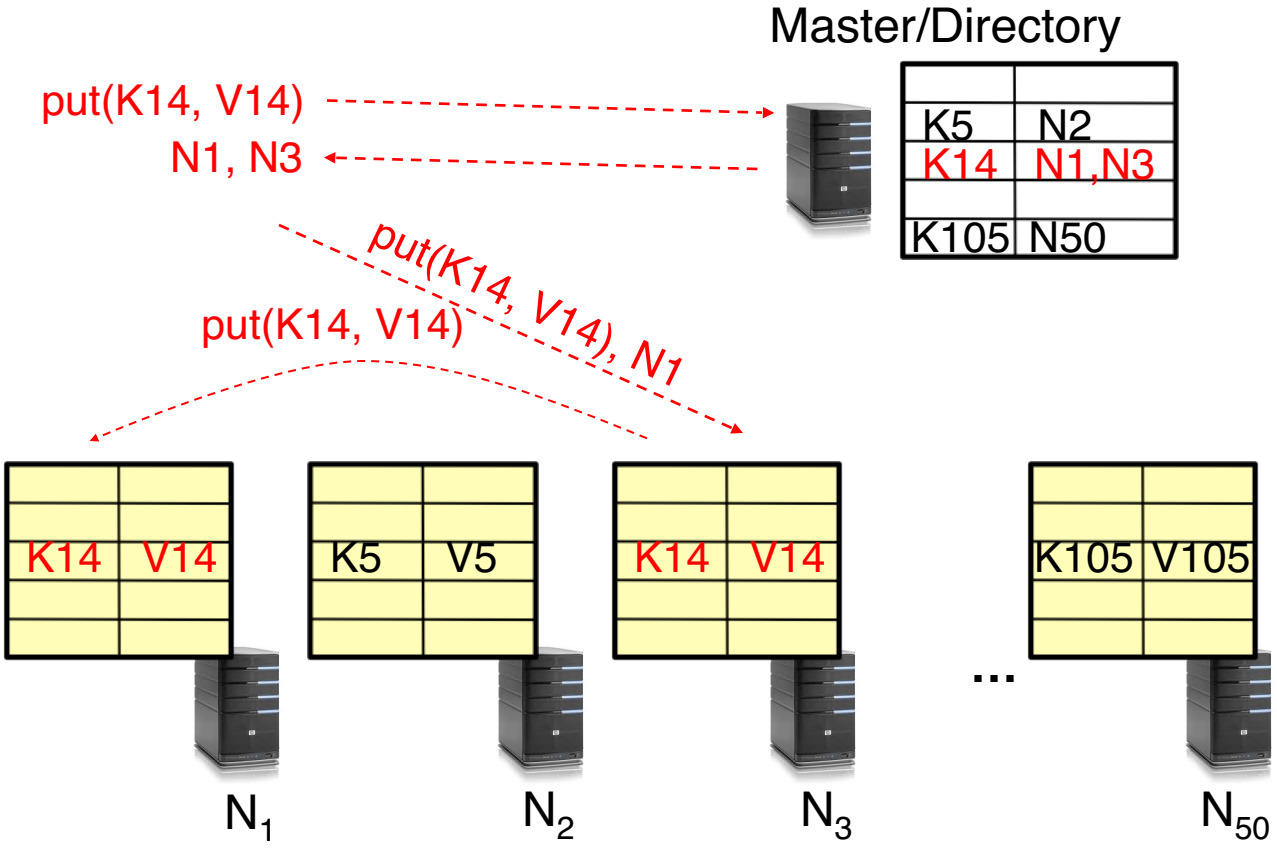
Discussion: Iterative vs. Recursive Query



- Recursive Query:
  - Advantages:
    - » Faster (latency), as typically master/directory closer to nodes
    - » Easier to maintain consistency, as master/directory can serialize puts()/gets()
  - Disadvantages: scalability bottleneck, as all "Values" go through master/directory
- Iterative Query
  - Advantages: more scalable
  - Disadvantages: slower (latency), harder to enforce data consistency

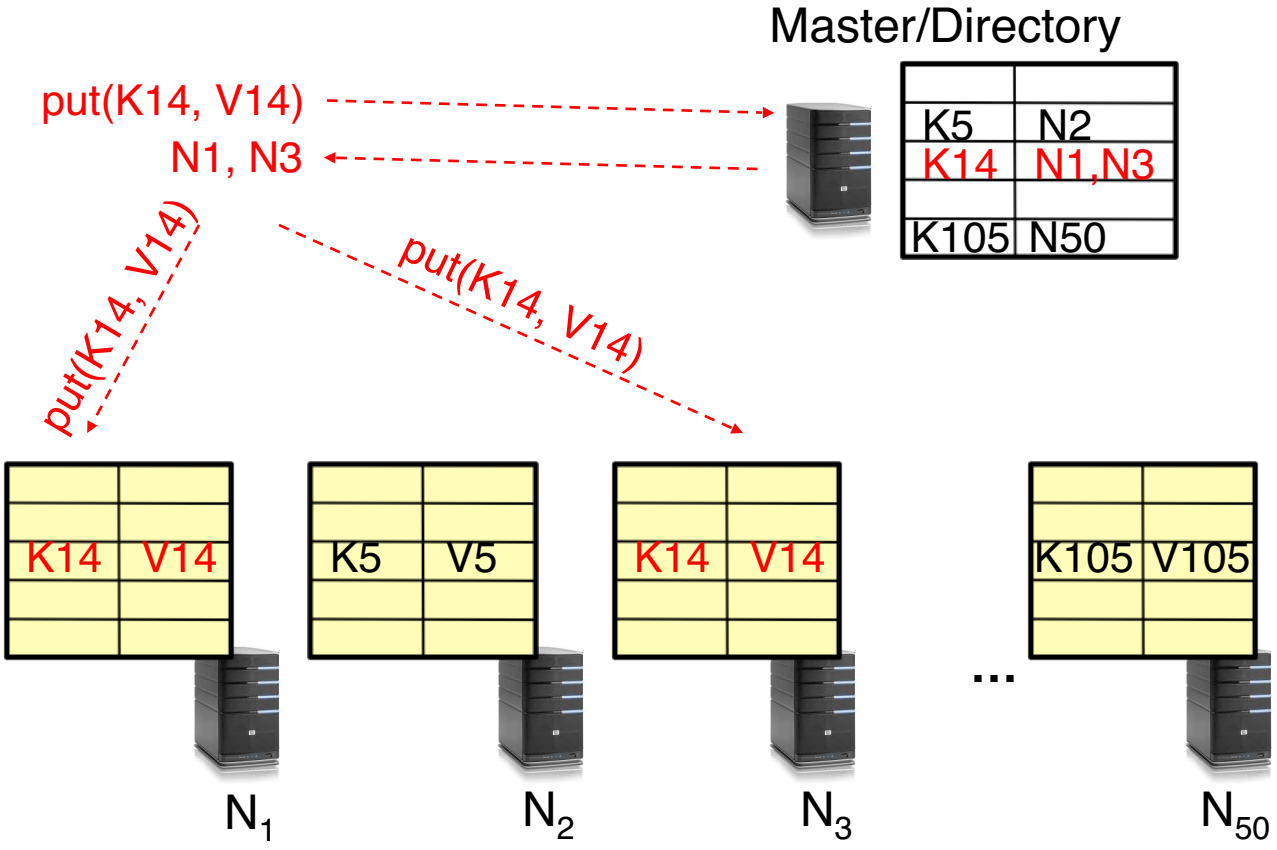
Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures (recursive version)



Fault Tolerance

- Again, we can have
  - **Recursive** replication (previous slide)
  - **Iterative** replication (this slide)



Scalability

- Storage: use more nodes
- Request Throughput:
  - Can serve requests from all nodes on which a value is stored in parallel
  - Large “values” can be broken into blocks (HDFS files are broken up this way)
  - Master can replicate a popular value on more nodes
- Master/directory scalability:
  - Replicate it
  - Partition it, so different keys are served by different masters/directories

Scalability: Load Balancing

- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available

- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node

- What happens when a node fails?
  - Need to replicate values from failed node to other nodes

Replication Challenges

- Need to make sure that a value is replicated correctly
- How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down the entire put()? Pick another node?
- In general, with multiple replicas
  - Slow puts and fast gets

Consistency

- How close does a distributed system emulate a single machine in terms of read and write semantics?

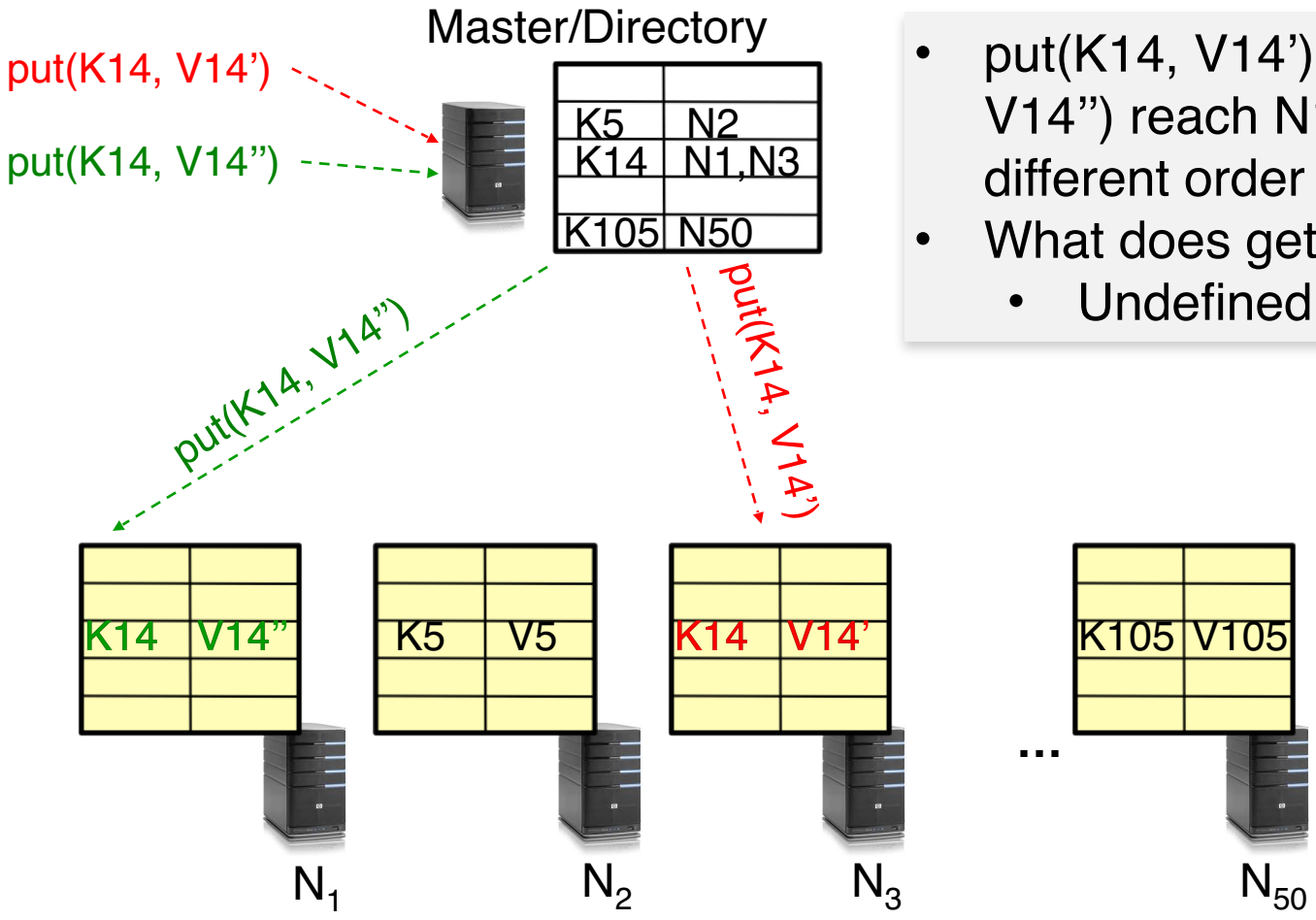
- **Q:** Assume **put(K14, V14')** and **put(K14, V14")** are concurrent, what value ends up being stored?
- **A:** assuming **put()** is atomic, then either **V14'** or **V14"**, right?

- **Q:** Assume a client calls **put(K14, V14)** and then **get(K14)**, what is the result returned by **get()**?
- **A:** It should be V14, right?

- Above semantics, not trivial to achieve in distributed systems

Concurrent Writes (Updates)

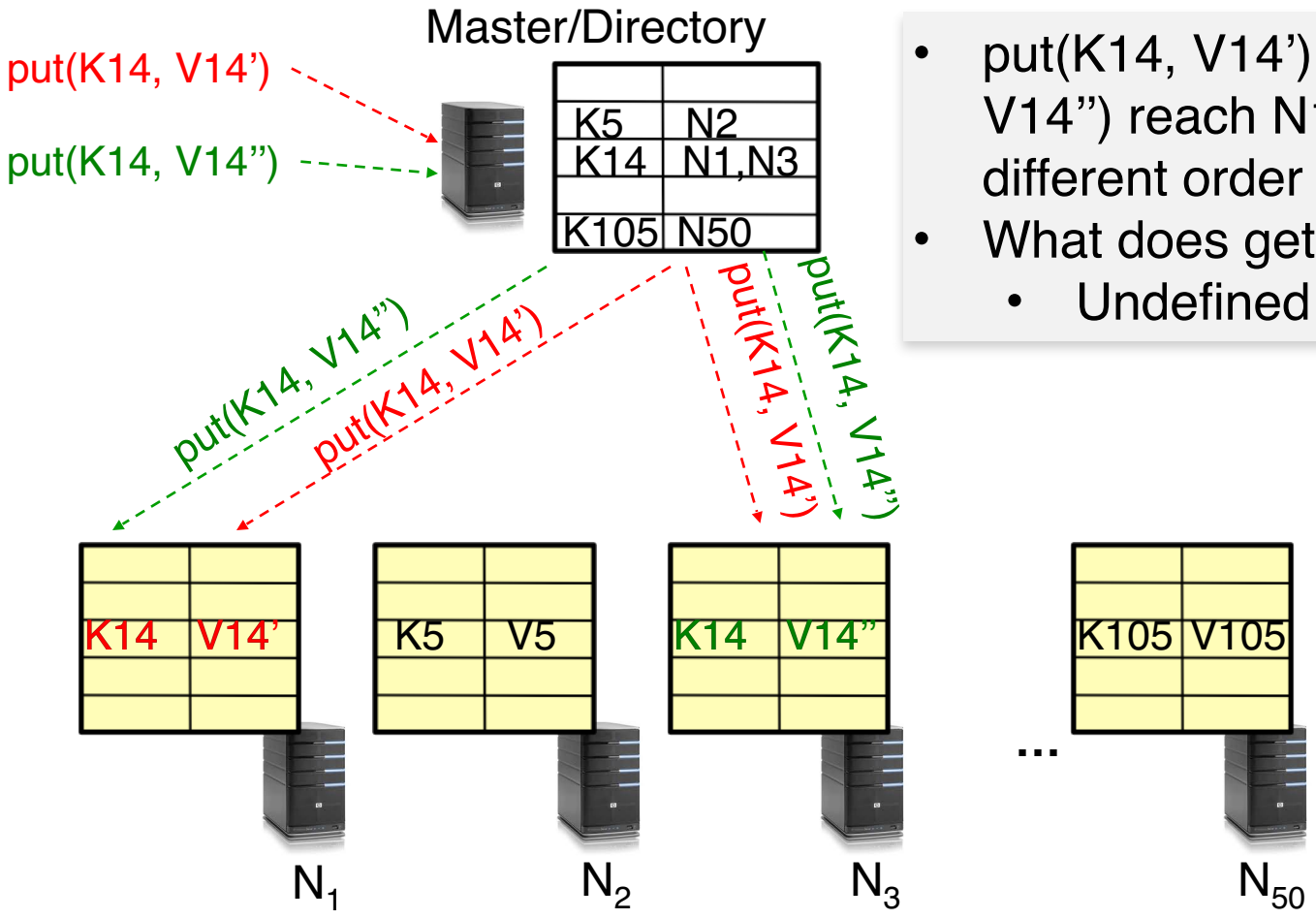
- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14") reach N1 and N3 in different order
- What does get(K14) return?
  - Undefined!

Concurrent Writes (Updates)

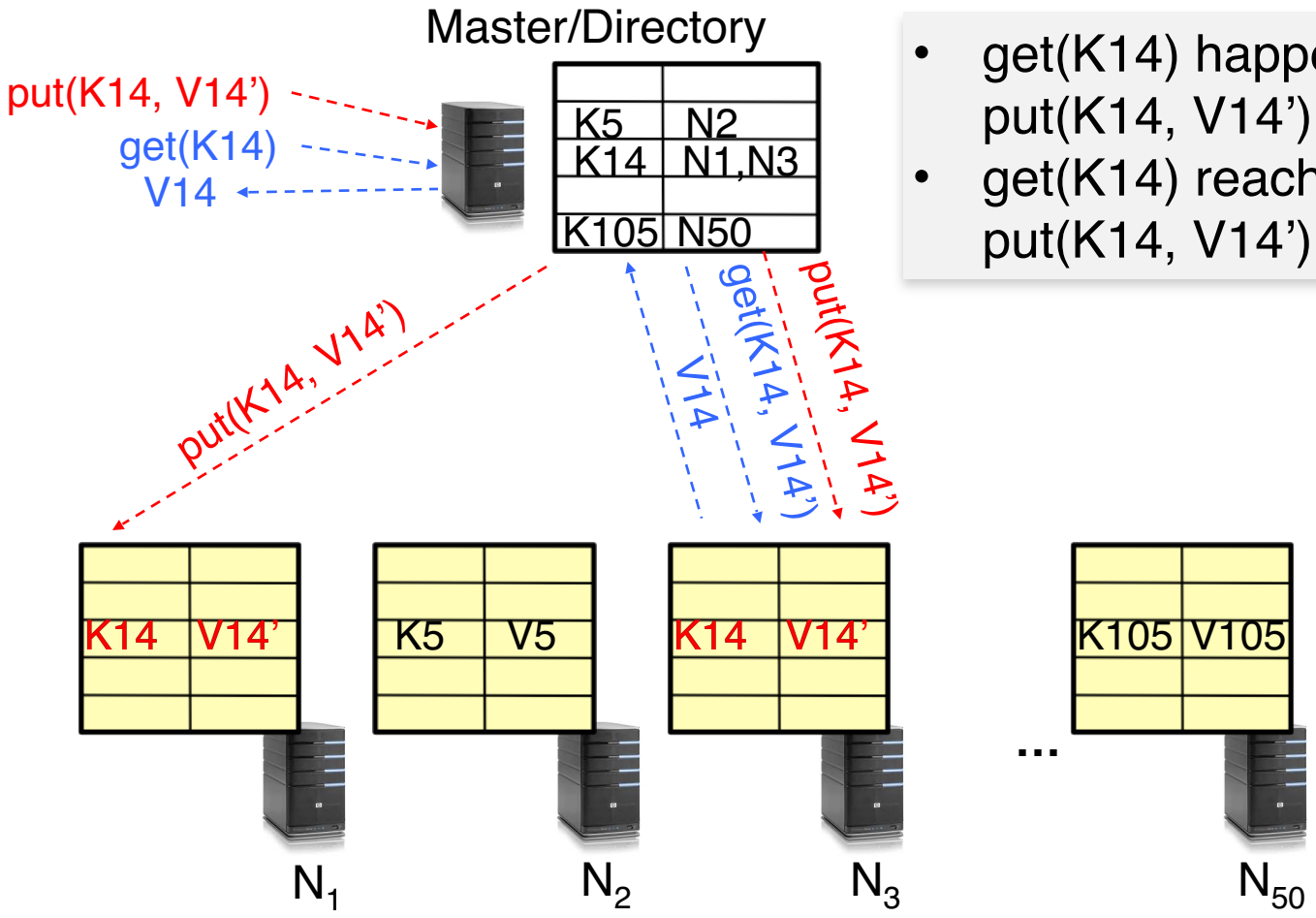
- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14") reach N1 and N3 in different order
- What does get(K14) return?
  - Undefined!

Read after Write

- Read not guaranteed to return value of latest write
  - Can happen if Master processes requests in different threads



- get(K14) happens right after put(K14, V14')
- get(K14) reaches N3 before put(K14, V14')!

Consistency (cont'd)

- Large variety of consistency models:
  - Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
    - » Think "one updated at a time"
    - » Transactions (we talked about it already!)
  - Eventual consistency: given enough time all updates will propagate through the system
    - » One of the weakest forms of consistency; used by many systems in practice
  - And many others: causal consistency, sequential consistency, strong consistency, …

Strong Consistency

- Assume Master serializes all operations
- Challenge: master becomes a bottleneck
  – Not addressed here
- Still want to improve performance of reads/writes → quorum consensus

Quorum Consensus

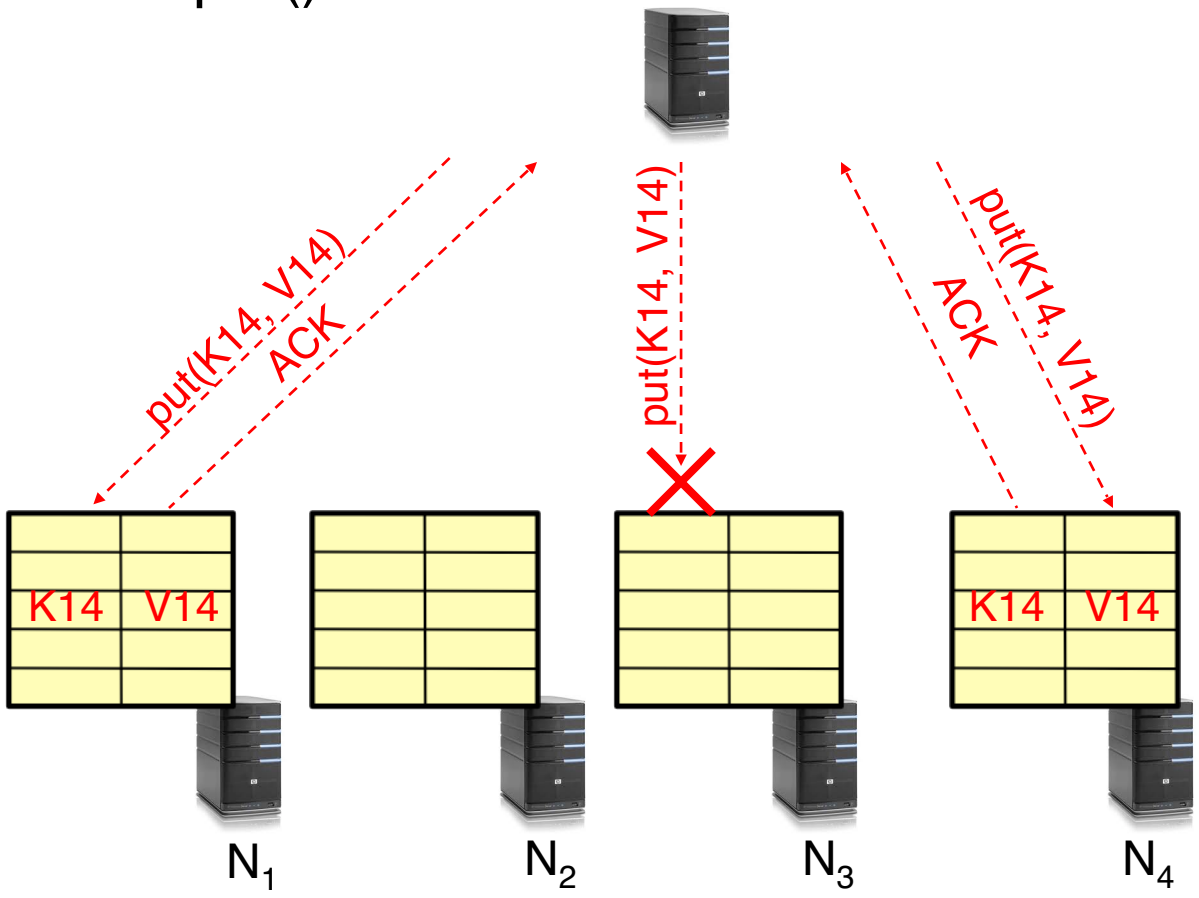
- Improve **put()** and **get()** operation performance

- Define a replica set of size N
- **put()** waits for acks from at least W replicas
- **get()** waits for responses from at least R replicas
- W+R > N

- Why does it work?
  - There is at least one node that contains the update

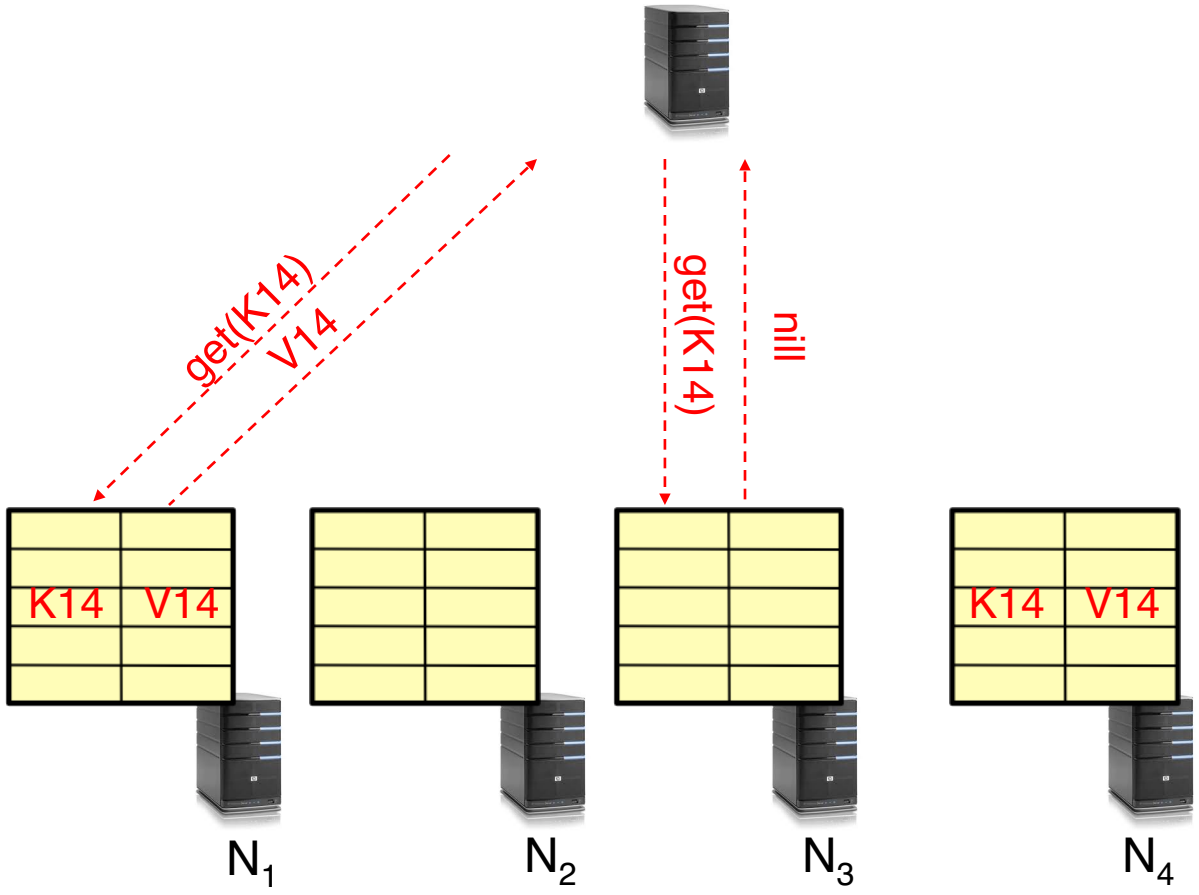
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N3, N4}
- Assume put() on N3 fails



Quorum Consensus Example

- Now, issuing get() to any two nodes out of three will return the answer



Summary: Key-Value Store

- Very large scale storage systems

- Two operations
  - put(key, value)
  - value = get(key)

- Challenges
  - Fault Tolerance → replication
  - Scalability → serve get()'s in parallel; replicate/cache hot tuples
  - Consistency → quorum consensus to improve put/get performance

Quiz 15.1: Key-Value Store

- Q1: True _ False _ Distributed Key-Value stores should always be Consistent, Available and Partition-Tolerant (CAP)
- Q2: True _ False _ On a single node, a key-value store can be implemented by a hash-table
- Q3: True _ False _ A Master can be a bottleneck point for a key-value store
- Q4: True _ False _ Iterative PUTs achieve lower throughput than recursive PUTs on a loaded system
- Q5: True _ False _ With quorum consensus, we can improve read performance at expense of write performance

Quiz 15.1: Key-Value Store

- Q1: True _ False _x_ Distributed Key-Value stores should always be Consistent, Available and Partition-Tolerant (CAP)
- Q2: True _x_ False _ On a single node, a key-value store can be implemented by a hash-table
- Q3: True _x_ False _ A Master can be a bottleneck point for a key-value store
- Q4: True _ False _x_ Iterative PUTs achieve lower throughput than recursive PUTs on a loaded system
- Q5: True _x_ False _ With quorum consensus, we can improve read performance at expense of write performance