# Introduction to NoSQL Databases

MSBA7024 / MACC7020

Database Design and Management

# Objectives

- Understand the shortcomings of RDBMS in today's applications
- Describe the basics of NoSQL and why it has become popular
- Describe the major properties of NoSQL databases
- Compare the pros and cons of RDBMS and NoSQL models
- Discuss and compare the different types of NoSQL models
- Understand the CAP theorem

# Background

- Relational database management systems (RDBMS)
  - mainstay of business
  - sometimes also called SQL databases
- Web-based applications caused spikes
  - explosion of social media sites (Facebook, Twitter) with large data needs
  - rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Hooking RDBMS to web-based application becomes troublesome

# Characteristics of RDBMS

- RDBMSs have been around for ages
  - MySQL is one of the most popular among them
- Data stored in tables
- Schema-based, i.e., structured tables
- Each row (data item) in a table has a primary key that is unique within that table
- Queried using SQL (Structured Query Language)
- Support joins

# Relational Database Example

**users table**

| user_id | name | zipcode | blog_url | blog_id |
|---|---|---|---|---|
| 101 | Alice | 12345 | alice.net | 1 |
| 422 | Charlie | 45783 | charlie.com | 3 |
| 555 | Bob | 99910 | bob.blogspot.com | 2 |

Primary keys

Foreign keys

**blog table**

| id | url | last_updated | num_posts |
|---|---|---|---|
| 1 | alice.net | 5/2/14 | 332 |
| 2 | bob.blogspot.com | 4/2/13 | 10003 |
| 3 | charlie.com | 6/15/14 | 7 |

**Example SQL queries**

1. SELECT zipcode
   FROM users
   WHERE name = "Bob"
2. SELECT url
   FROM blog
   WHERE id = 3
3. SELECT users.zipcode, blog.num_posts
   FROM users JOIN blog
   ON users.blog_url = blog.url

# Popular RDBMS

# Mismatch with Today's Workloads

- Data: Large and unstructured
- Lots of random reads and writes
- Sometimes write-heavy
- Foreign keys less needed
- Joins infrequent

# Needs of Today's Workloads

- Speed
- Avoid Single point of Failure (SPoF)
- Low TCO (Total cost of operation)
- Fewer system administrators
- Incremental Scalability
- Scale out, not up

# Scale out, not Scale up

- Scale up = grow your cluster capacity by replacing with more powerful machines
  - Traditional approach
  - a.k.a. vertical scaling: make a “single” machine more powerful
  - Not cost-effective
  - And you need to replace machines often
- Scale out = incrementally grow your cluster capacity by adding more COTS machines (Components Off the Shelf)
  - Cheaper
  - a.k.a. horizontal scaling
  - Over a long duration, phase in a few newer (faster) machines as you phase out a few older machines
  - Used by most companies who run datacenters and clouds today

# Scaling out RDBMS: Master/Slave

- Master/Slave
  - All writes are written to the master
  - All reads performed against the replicated slave databases
  - Critical reads may be incorrect as writes may not have been propagated down
  - Large datasets can pose problems as master needs to duplicate data to slaves

# Scaling out RDBMS: Sharding

- Sharding (Partitioning)
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards

# RDBMS

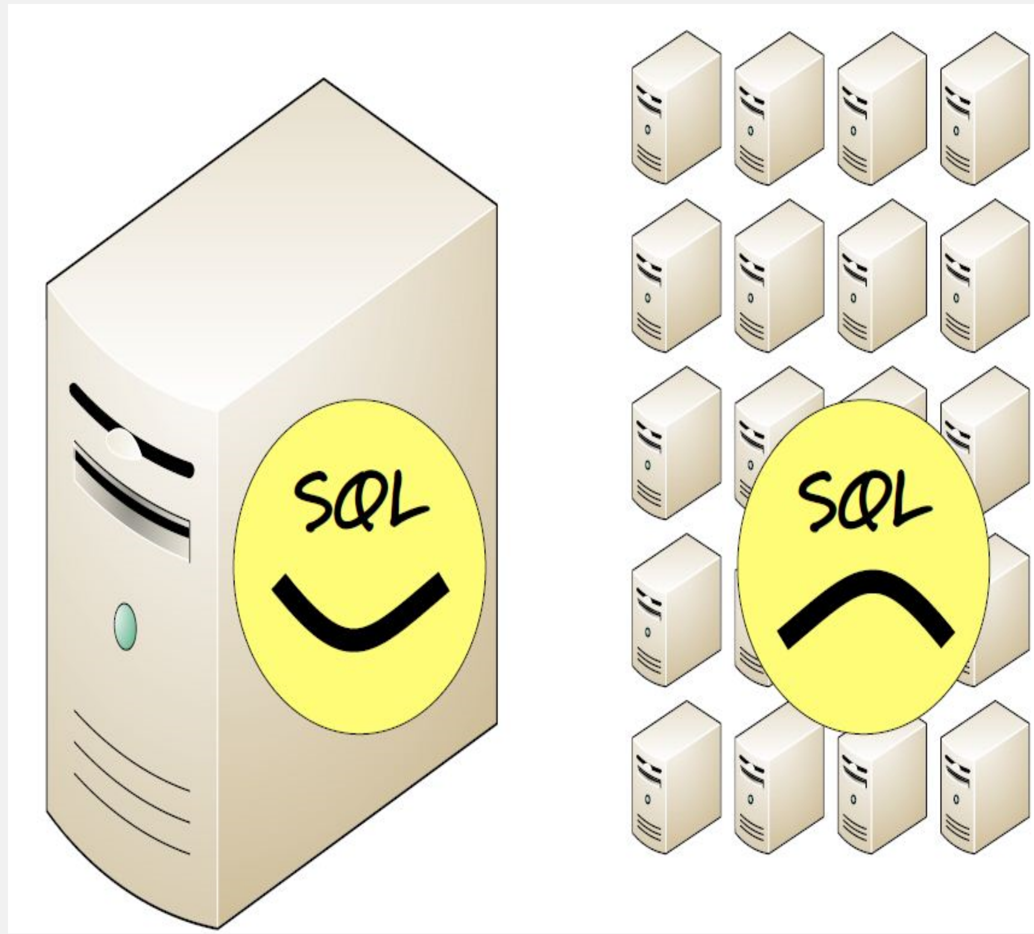
- ❑ Relational databases were not built for **distributed applications.**

**Because…**

- ❑ Joins are expensive
- ❑ Hard to scale horizontally
- ❑ Impedance mismatch occurs
- ❑ Expensive (product cost, hardware, Maintenance)

And....
It's weak in:

- ❑ Speed (performance)
- ❑ Availability
- ❑ Partition tolerance



# NoSQL Data Model

- NoSQL = "Not Only SQL"
- Necessary API operations: get(key) and put(key, value)
    - And some extended operations, e.g., "CQL" in Cassandra key-value store

- Tables
    - "Column families" in Cassandra, "Table" in HBase, "Collection" in MongoDB
    - Like RDBMS tables, but:
        - May be unstructured: May not have schemas
        - Some columns may be missing from some rows
        - Do not always support joins or have foreign keys
        - Can have index tables, just like RDBMS

# NoSQL Data Model

- Key features (advantages):
  - Non-relational
  - Do not require schema
  - Data are replicated to multiple nodes and can be partitioned:
    - down nodes easily replaced
    - no single point of failure
  - Horizontal scalable
  - Cheap, easy to implement (open-source)
  - Massive write performance
  - Fast key-value access

# NoSQL Data Model

# NoSQL Data Model

- Unstructured
- Columns Missing from some Rows
- No schema imposed
- No foreign keys, joins may not be supported

Key | Value

**users table**

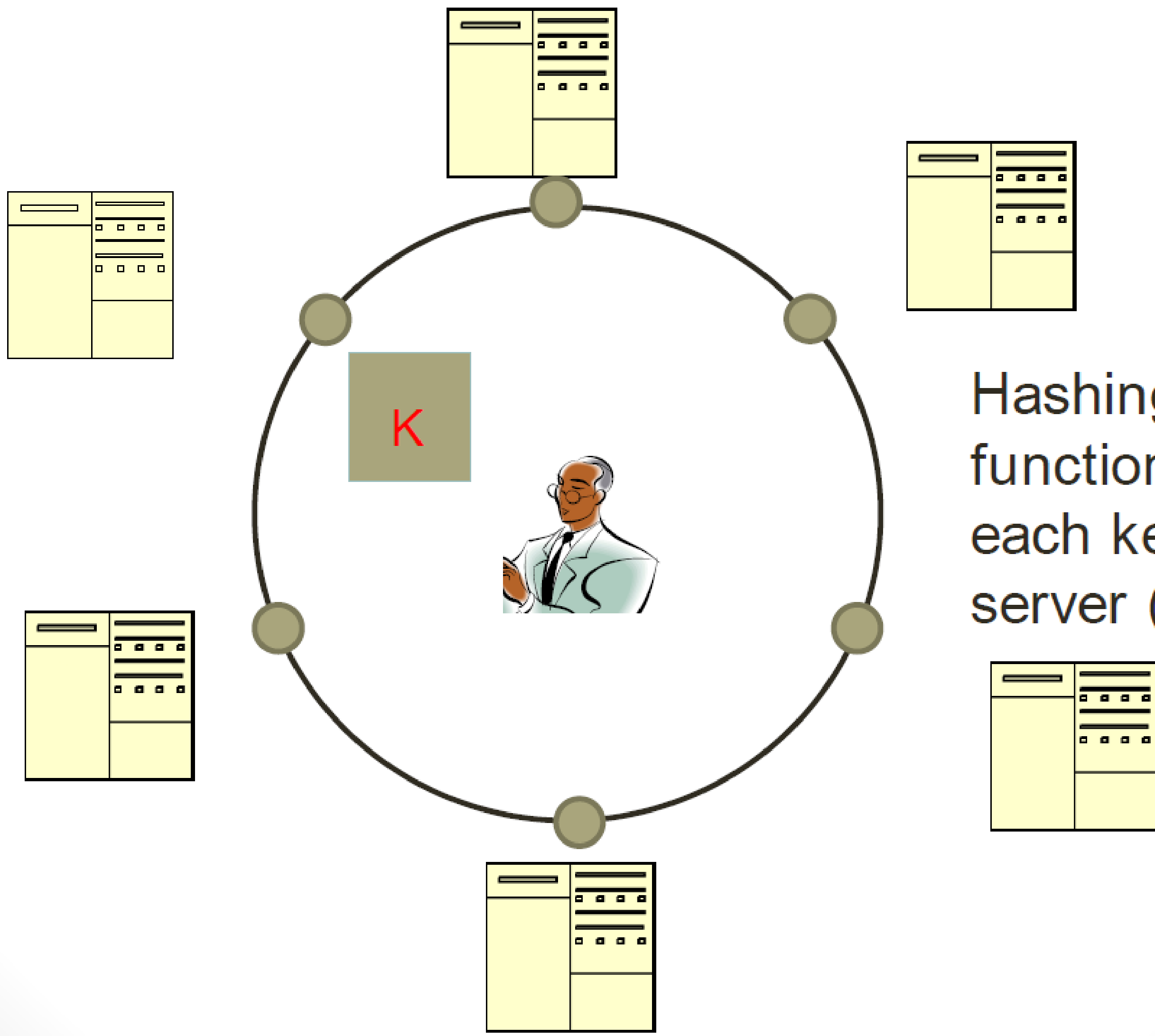
| user_id | name | zipcode | blog_url |
|---|---|---|---|
| 101 | Alice | 12345 | alice.net |
| 422 | Charlie | | charlie.com |
| 555 | | 99910 | bob.blogspot.com |

Key | Value

**blog table**

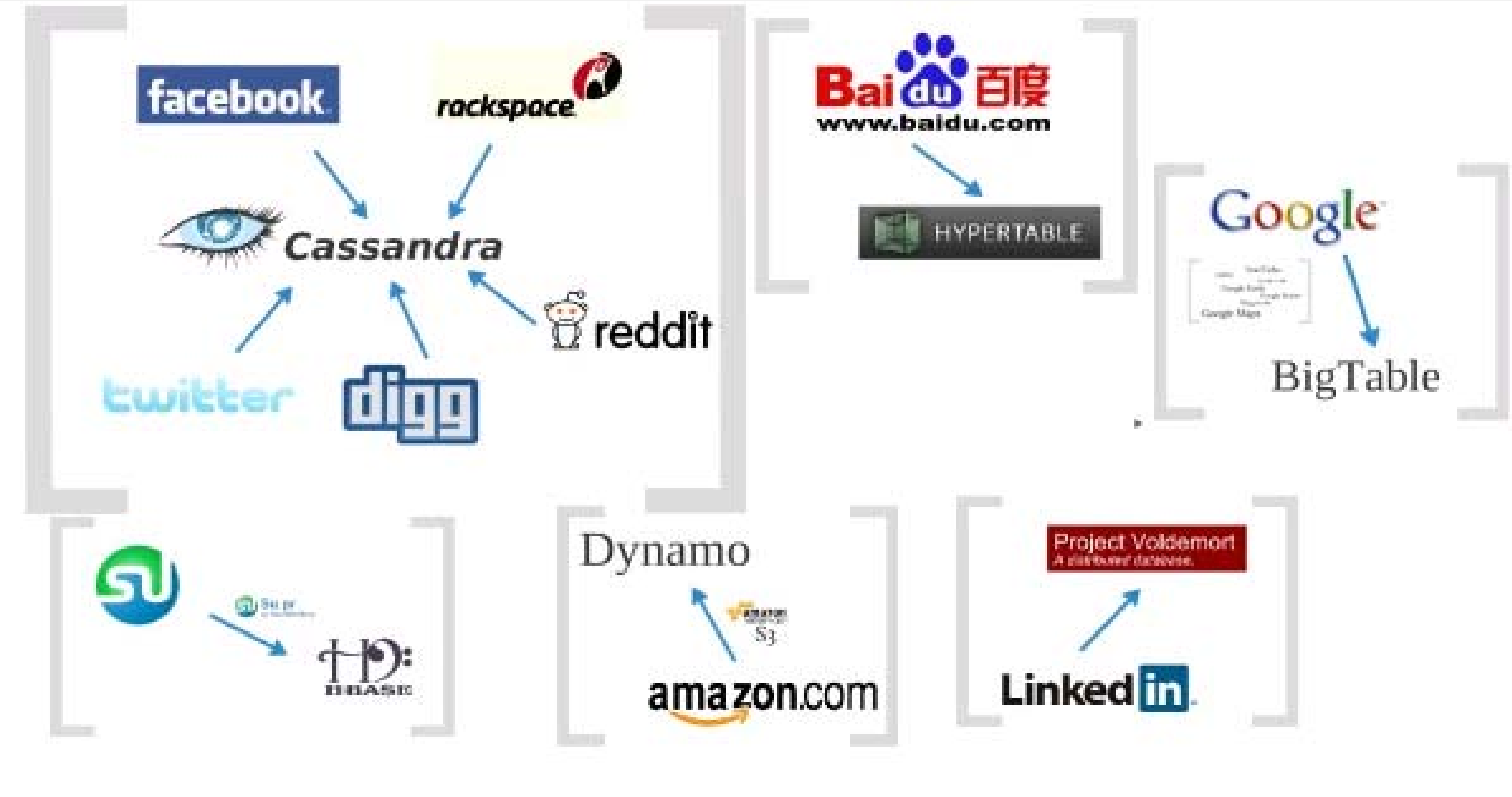
| id | url | last_updated | num_posts |
|---|---|---|---|
| 1 | alice.net | 5/2/14 | 332 |
| 2 | bob.blogspot.com | | 10003 |
| 3 | charlie.com | 6/15/14 | |

# Typical NoSQL Architecture



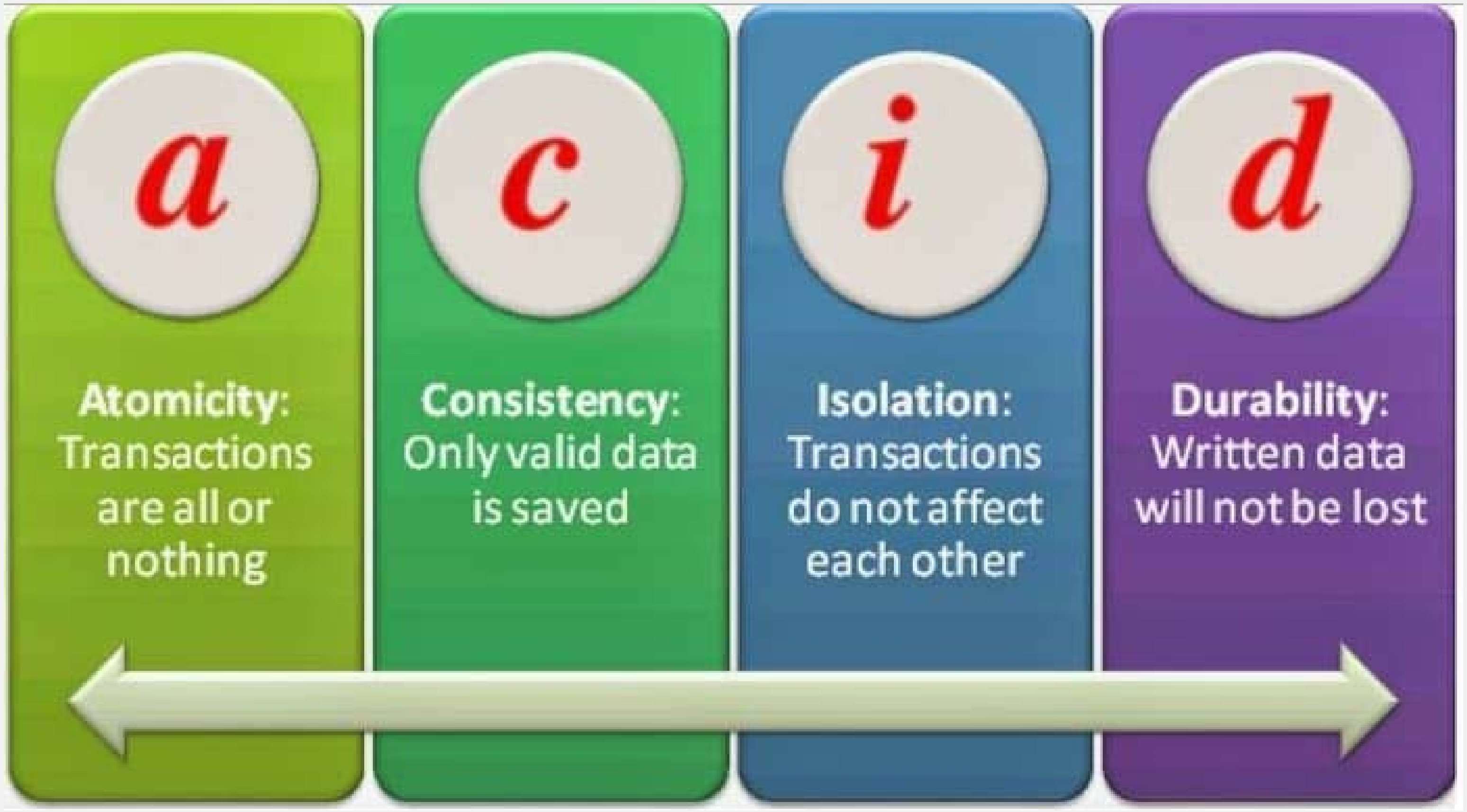

Hashing function maps each key to a server (node)

# Who are using them?



# RDBMS vs. NoSQL

- RDBMS provide ACID



# RDBMS vs. NoSQL

- NoSQL databases like MongoDB and Cassandra provide BASE
  - Basically Available Soft-state Eventual Consistency

# Benefits of NoSQL

- **Elastic Scaling**
  - RDBMS scale up – bigger load , bigger server
  - NoSQL scale out – distribute data across multiple hosts seamlessly
- **DBA Specialists**
  - RDBMS require highly trained expert to monitor DB
  - NoSQL require less management, automatic repair and simpler data models
- **Big Data**
  - Huge increase in data RDBMS: capacity and constraints of data volumes at its limits
  - NoSQL designed for big data

# Benefits of NoSQL

- **Flexible data models**
  - Change management to schema for RDBMS have to be carefully managed
  - NoSQL databases more relaxed in structure of data
    - Database schema changes do not have to be managed as one complicated change unit
    - Application already written to address an amorphous schema

- **Economics**
  - RDBMS rely on expensive proprietary servers to manage data
  - NoSQL: clusters of cheap commodity servers to manage the data and transaction volumes
  - Cost per gigabyte or transaction/second for NoSQL can be lower than the cost for a RDBMS

# Drawbacks of NoSQL

- **Support**
  - RDBMS vendors provide a high level of support to clients
    - Stellar reputation
  - NoSQL are open source projects with startups supporting them
    - Reputation not yet established

- **Maturity**
  - RDBMS mature product: means stable and dependable
    - Also means old no longer cutting edge nor interesting
  - NoSQL are still implementing their basic feature set

# Drawbacks of NoSQL

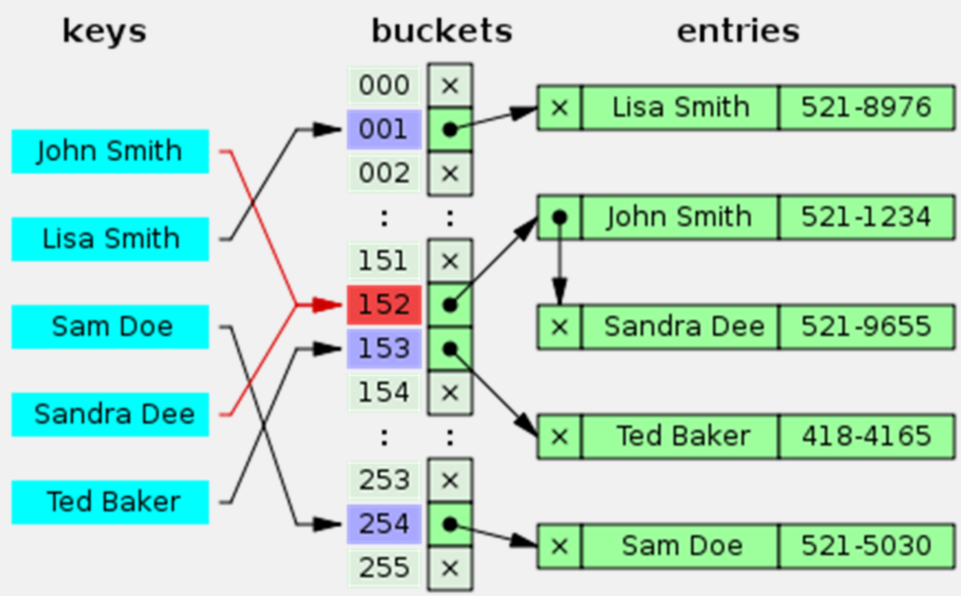
- **Administration**
  - RDBMS administrator has a well-defined role
  - NoSQL's goal: no administrator necessary; however, NoSQL still requires effort to maintain

- **Lack of Expertise**
  - Whole workforce of trained and seasoned RDBMS developers
  - Still recruiting developers to the NoSQL camp

- **Analytics and Business Intelligence**
  - RDBMS designed to support decision-making
    - Data warehouse
  - NoSQL designed to meet the needs of Web 2.0 applications—not designed for ad hoc query of the data
    - Tools are being developed to address this need
    - More flexible to include new and unstructured data

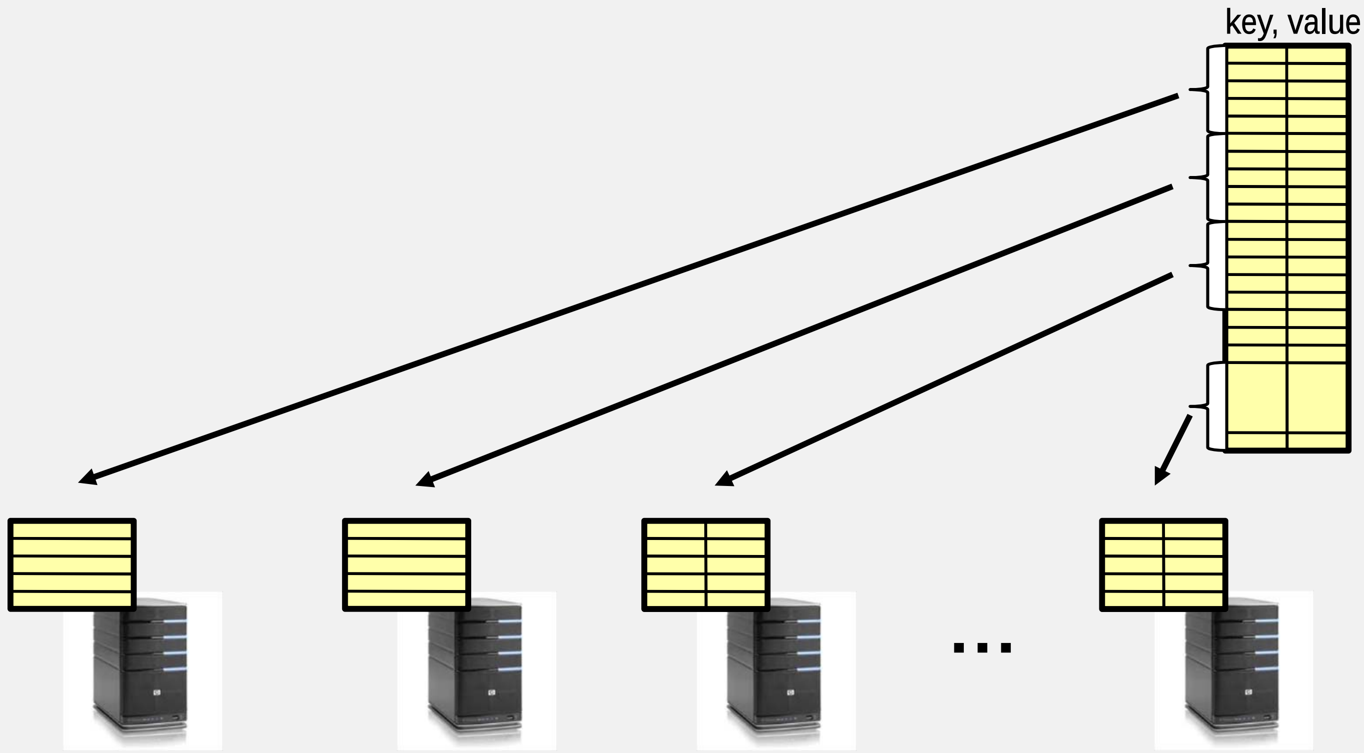
# Key-value Store

- NoSQL databases generally rely on key-value store.
- It is a dictionary data structure
  - Insert, lookup, and delete by key
  - Usually based on hash
  - But distributed



# Key-value Store

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



# Key-value Store

- Business: Key → Value
- twitter.com: tweet id → information about tweet
- amazon.com: item number → information about it
- facebook.com: user id → user profile, photos, etc.
- kayak.com: flight number → information about flight, e.g., availability
- yourbank.com: account number → account balances, transaction histories

# Key-value Store

- Basic access:
  - get(key): extract the value given a key
  - put(key, value): create or update the value given its key
  - delete(key): remove the key and its associated value
  - execute(key, operation, parameters): invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc.)

# Key-value Store

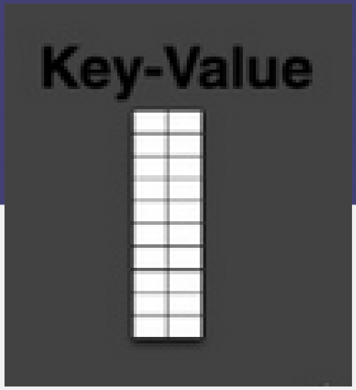
- Can handle large volumes of data, e.g., PBs
  - Can distribute data over hundreds, even thousands of machines
  - Designed to be faster with lower overhead (additional storage) than conventional DBMS.

# NoSQL categories

1. Standard key-value
   - Example: DynamoDB, Voldermort, Scalaris
2. Document-based
   - Example: MongoDB, CouchDB
3. Column-based
   - Example: BigTable, Cassandra, HBase
4. Graph-based
   - Example: Neo4J, InfoGrid
5. Vector-based
   - Example: Pinecone, Milvus, Chroma

- “No-schema” is a common characteristics of most NoSQL storage systems
- Provide “flexible” data types

# Key-value




- Focus on scaling to huge amounts of data
- Designed to handle massive load
- Based on Amazon's dynamo paper
- Data model: (global) collection of Key-value pairs
- *Dynamo ring partitioning* and *replication*
- Example: (DynamoDB)
  - *items* having one or more attributes (name, value)
  - An *attribute* can be single-valued or multi-valued like set.
  - items are combined into a *table*

# Key-value

**Pros:**

- Very fast
- Very scalable (horizontally distributed to nodes based on key)
- Simple data model
- Eventual consistency
- Fault-tolerance

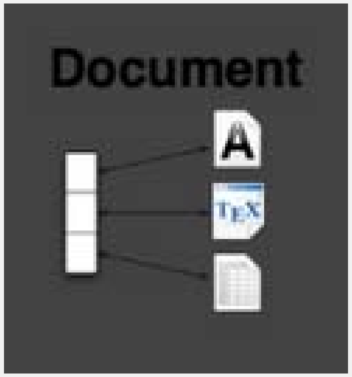
**Cons:**

- Simple key-value cannot model more complex data structure such as objects

# Key-value

| Name | Producer | Data model | Querying |
|---|---|---|---|
| | | | |
| SimpleDB | Amazon | set of couples (key, {attribute}), where attribute is a couple (name, value) | restricted SQL; select, delete, GetAttributes, and PutAttributes operations |
| Redis | Salvatore Sanfilippo | set of couples (key, value), where value is simple typed value, list, ordered (according to ranking) or unordered set, hash value | primitive operations for each value type |
| Dynamo | Amazon | like SimpleDB | simple get operation and put in a context |
| Voldemort | LinkedIn | like SimpleDB | similar to Dynamo |

# Document-based



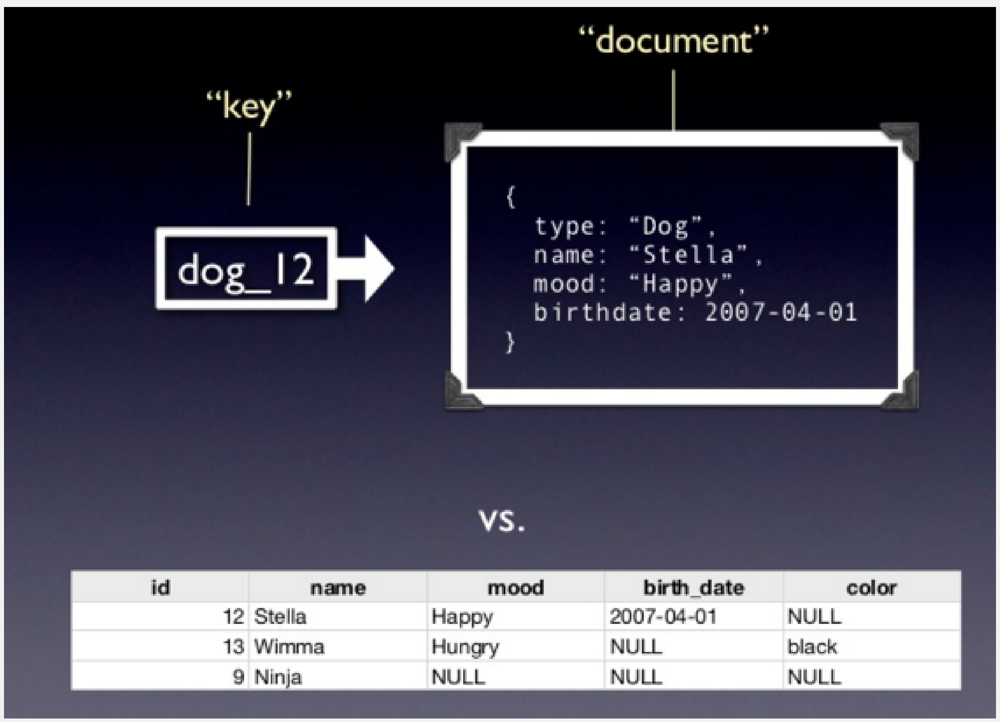

- Can model more complex objects
- Inspired by Lotus Notes
- Data model: collection of documents
- Document: JSON (**J**ava**S**cript **O**bject **N**otation)
  - A data model, key-value pairs, which supports objects, records, structs, lists, array, maps, dates, Boolean, XML, and other semi-structured formats.

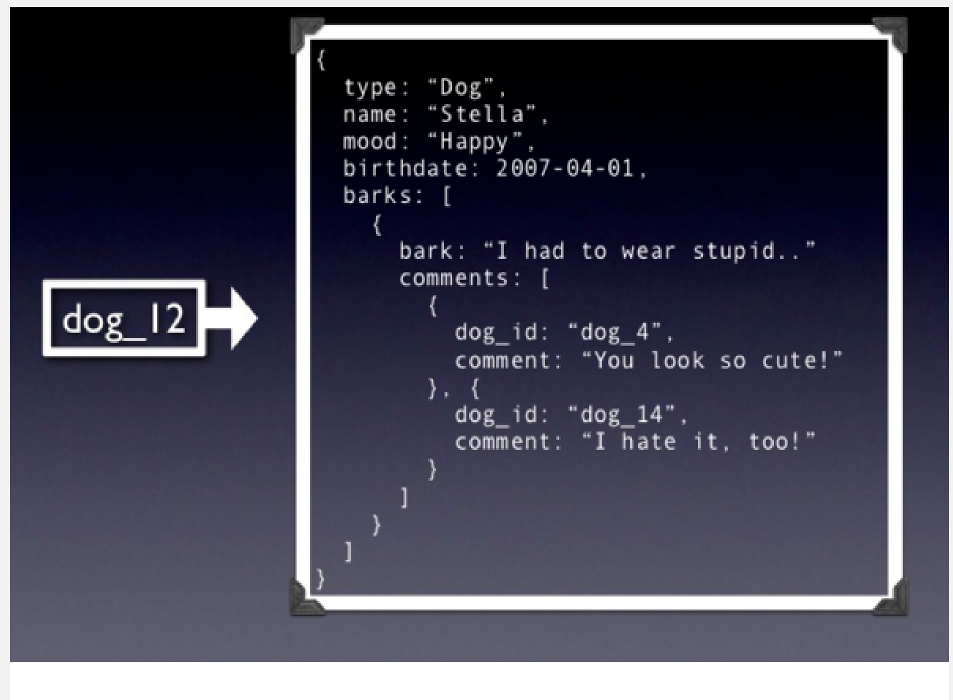
# Document-based

- Example: (MongoDB) document

```
{
Name:"Jaroslav",
Address:"Malostranske nám. 25, 118 00 Praha 1",
Grandchildren: {Claire: "7", Barbara: "6", "Magda:
"3", "Kirsten: "1", "Otis: "3", Richard: "1"},
Phones: ["123-456-7890", "234-567-8963"]
}
```

# Document-based



# Document-based

| Name | Producer | Data model | Querying |
|---|---|---|---|
| | | | |
| MongoDB | 10gen | object-structured documents stored in collections;<br>each object has a primary key called ObjectId | manipulations with objects in collections (find object or objects via simple selections and logical expressions, delete, update,) |
| Couchbase | Couchbase | document as a list of named (structured) items (JSON document) | by key and key range, views via Javascript and MapReduce |

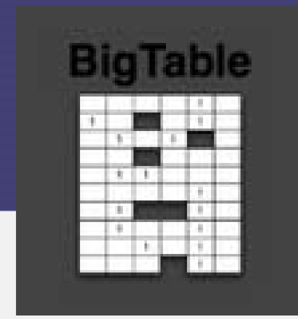
# Column-based

- Based on Google's BigTable
- Google uses the key-value paradigm to map URLs to multidimensional data, such as:
    - Timestamps/Versions
    - Rank
    - Keywords
    - Links
- No explicit ordering is needed on keys since a hash function is used

# Column-based

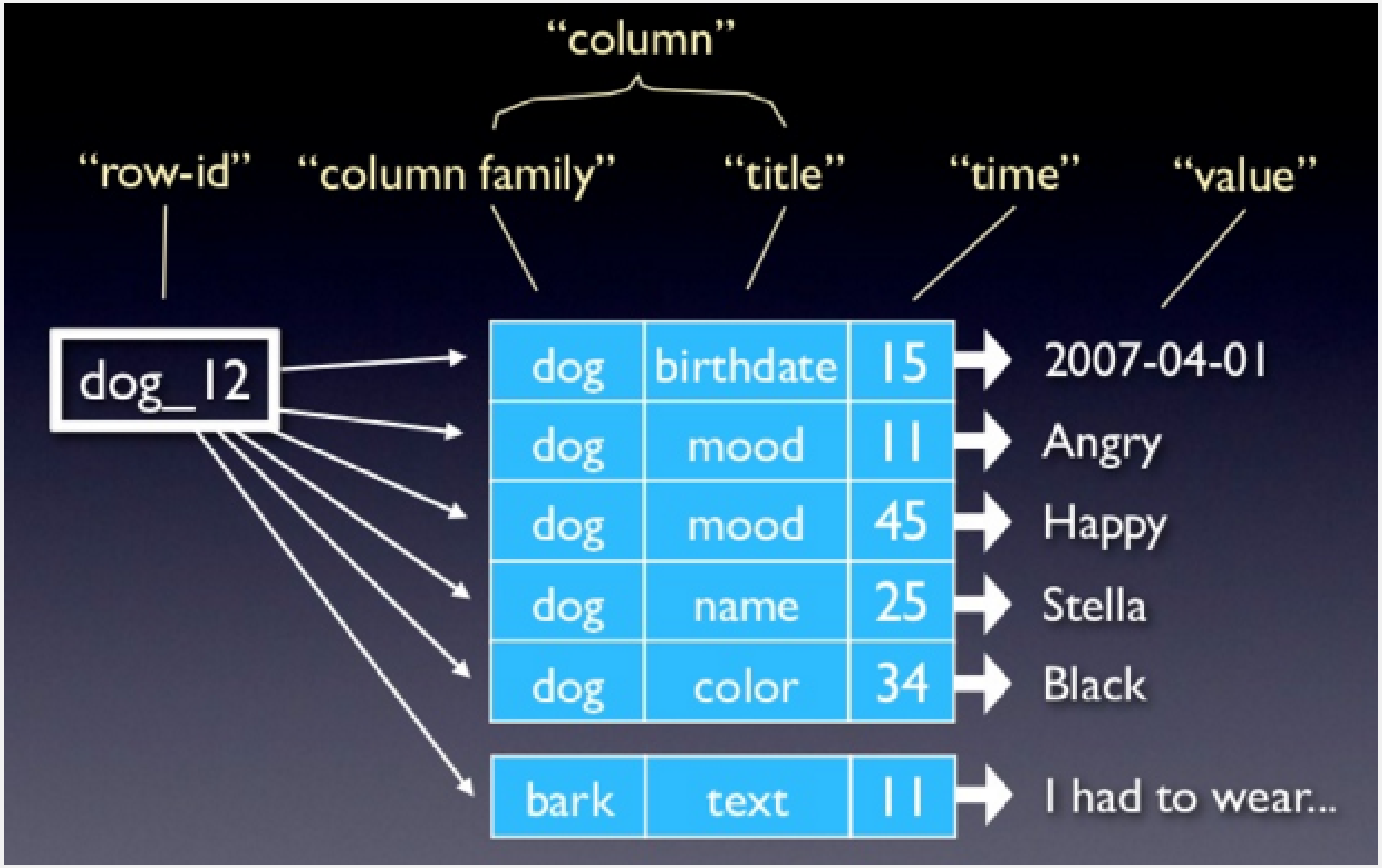
- Tables similarly to RDBMS, but handle semi-structured data
- Data model:
  - Collection of Column Families
  - Column family = (key, value) where value = set of **related** columns (standard, super)
  - indexed by *row key*, *column key* and *timestamp*
- Allow key-value pairs to be stored (and retrieved on key) as a distributed hash table
- Properties: partitioning (horizontally and/or vertically), high availability, completely transparent to application

# Column-based




- One column family can have variable numbers of columns
- Cells within a column family are sorted "physically"
- Very sparse, most cells have null values
- **Comparison:** RDBMS vs column-based NoSQL
  - Query on multiple tables
    - **RDBMS:** must fetch data from several places on disk and glue together
    - **Column-based NoSQL:** only fetch column families of those columns that are required by a query
      - all columns in a column family are stored together on the disk
      - multiple rows can be retrieved in one read operation → data locality

# Column-based



# Column-based

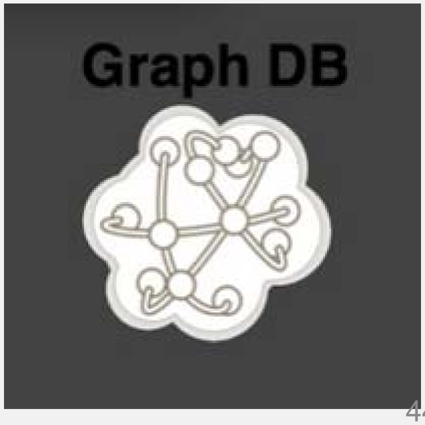
- RDBMS store an entire row together (on disk or at a server)
- Column-oriented systems typically store a column together (or a group of columns).
  - Entries within a column are indexed and easy to locate, given a key (and vice-versa)
- Why useful?
  - Range searches within a column are fast since you do not need to fetch the entire database
  - E.g., Get all the blog_ids from the blog table that were updated within the past month
    - Search in the the last_updated column, fetch corresponding blog_id column
    - Do not need to fetch the other columns

# Column-based

| Name | Producer | Data model | Querying |
|---|---|---|---|
| | | | |
| BigTable | Google | set of couples (key, {value}) | selection (by combination of row, column, and time stamp ranges) |
| HBase | Apache | groups of columns (a BigTable clone) | JRUBY IRB-based shell (similar to SQL) |
| Hypertable | Hypertable | like BigTable | HQL (Hypertext Query Language) |
| CASSANDRA | Apache (originally Facebook) | columns, groups of columns corresponding to a key (supercolumns) | simple selections on key, range queries, column or columns ranges |
| PNUTS | Yahoo | (hashed or ordered) tables, typed arrays, flexible schema | selection and projection from a single table (retrieve an arbitrary single record by primary key, range queries, complex predicates, ordering, top-k) |

# Graph-based

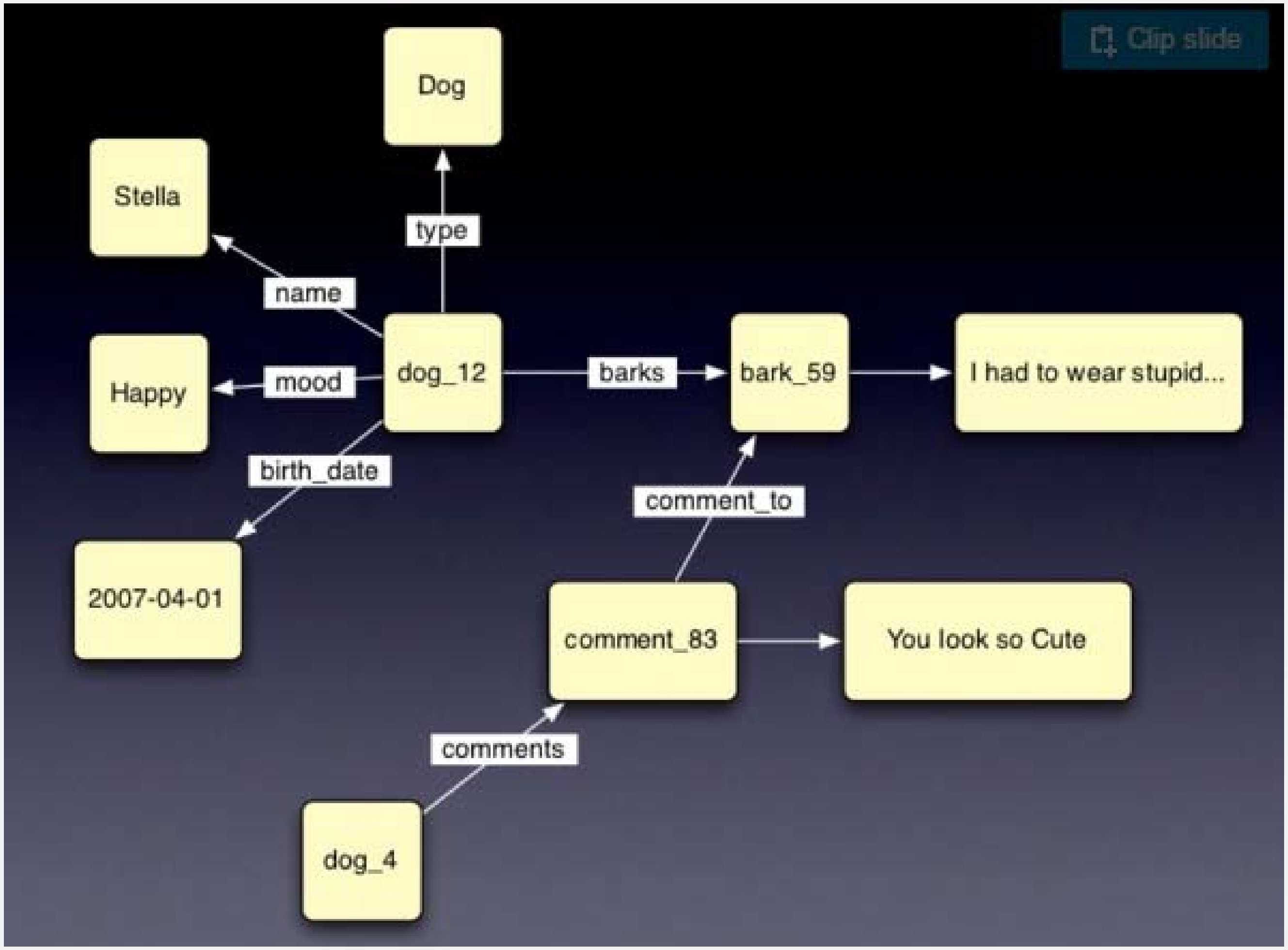
- Focus on modeling the structure of data (*interconnectivity*)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory (G=(E,V))
- Data model:
  - (Property Graph) nodes and edges
    - Nodes may have properties  (including ID)
    - Edges may have labels or roles
  - Key-value pairs on both



# Graph-based

- Good for applications where you need to traverse relationships to look for patterns such as social networks, fraud detection, and recommendation engines
- Interfaces and query languages vary
- *Single-step* vs *path expressions* vs *full recursion*
- Example:
    - Neo4j, FlockDB, Pregel, InfoGrid …

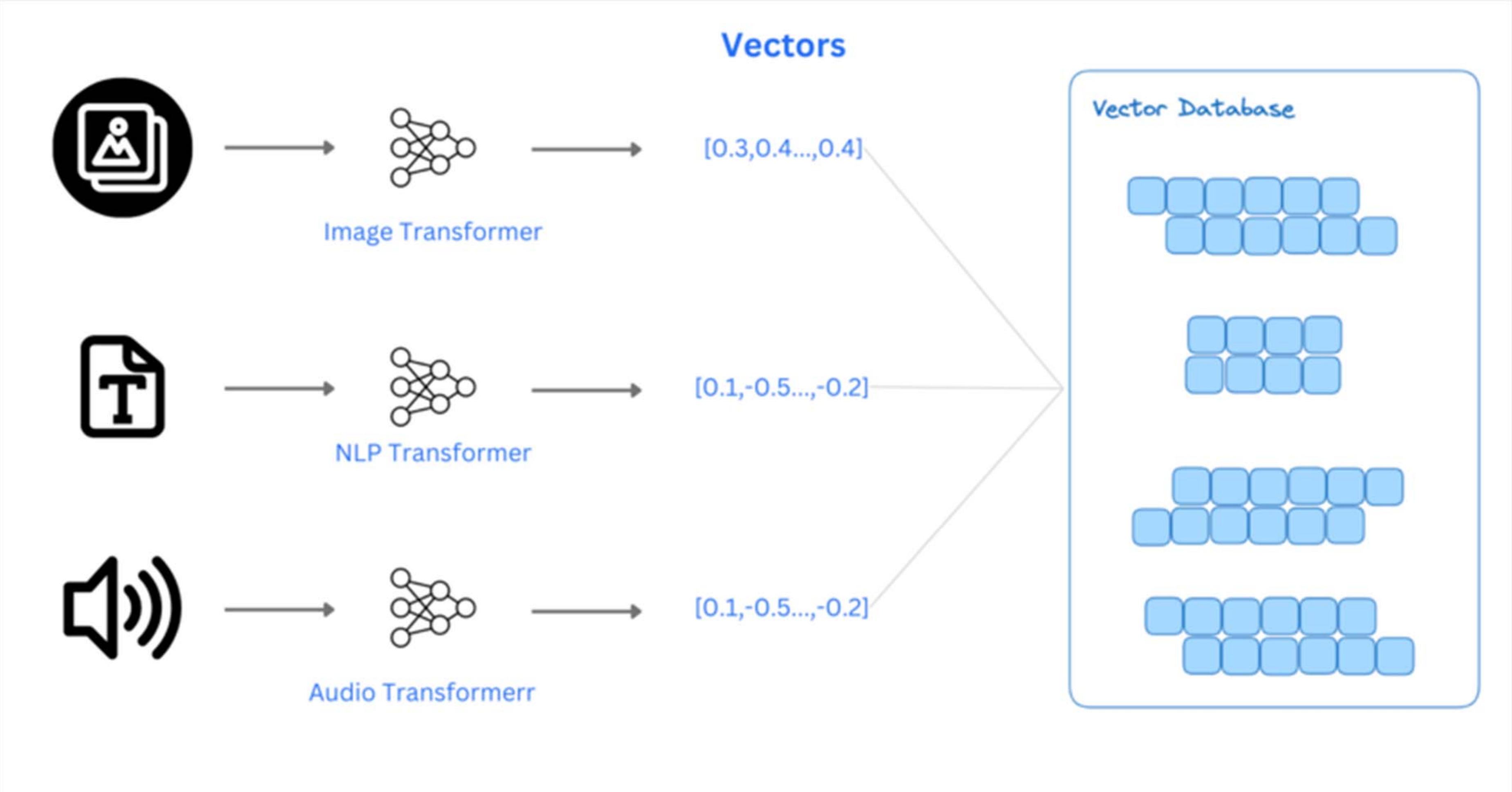
# Graph-based


Clip slide
Dog
type
Stella
name
dog_12
barks
bark_59
I had to wear stupid...
Happy
mood
birth_date
comment_to
2007-04-01
comment_83
You look so Cute
comments
dog_4

# Vector-based

- Vector databases become popular because of large language models (LLMs)
- In LLMs, documents are represented as vectors called embeddings (e.g, Word2Vec, GloVe, transformers)
- Offer optimized storage, indexing, and querying capabilities for embeddings
- Example:
  - Pinecone, Milvus, Chroma, Weaviate, Vespa, …

# Vector-based



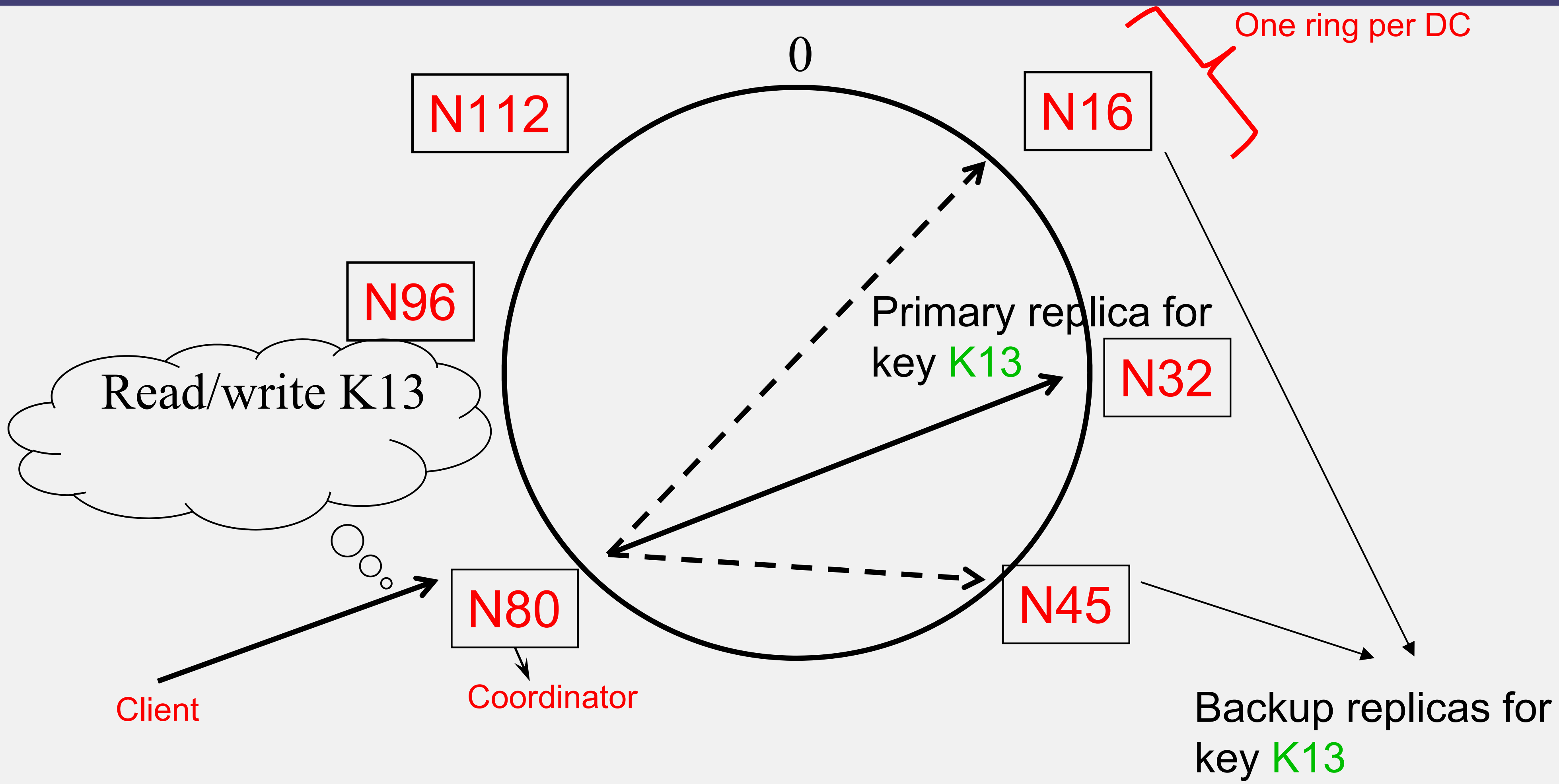
# Example: Cassandra

- A distributed column-oriented key-value store
- Intended to run in a datacenter (and also across DCs)
- Originally designed at Facebook
- Open-sourced later, today an Apache project
- Some of the companies that use Cassandra in their production clusters
  - IBM, Adobe, HP, eBay, Ericsson, Symantec
  - Twitter, Spotify
  - PBS Kids
  - Netflix: uses Cassandra to keep track of your current position in the video you are watching

# Let’s go Inside Cassandra: Key -> Server Mapping

- How do you decide which server(s) a key-value resides on?

# Key -> Server Mapping in Cassandra




Cassandra uses a Ring-based distributed hash table

# Writes

- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one coordinator node in Cassandra cluster
  - Coordinator may be per-key, or per-client, or per-query
  - Per-key Coordinator ensures writes for the key are serialized
- Coordinator uses Partitioner to send query to all replica nodes responsible for key
- When X replicas respond, coordinator returns an acknowledgement to the client
  - X? We'll see later.

# Deletes

Delete: don't delete item right away

- Add a **tombstone** to the log
- Eventually, when compaction encounters tombstone it will delete item

# Reads

Read: Similar to writes, except

- Coordinator can contact X replicas (e.g., in same rack)
  - Coordinator sends read to replicas that have responded quickest in past
  - When X replicas respond, coordinator returns the latest-timestamped value from among those X
  - (X? We'll see later.)
- Coordinator also fetches value from other replicas
  - Checks consistency in the background, initiating a **read repair** if any two values are different
  - This mechanism seeks to eventually bring all replicas up to date
- A row may be split across multiple tables

  => reads need to touch multiple tables

  => reads slower than writes (but still fast)

# Membership

- Any server in cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the cluster
- List needs to be updated automatically as servers join, leave, and fail

# Cassandra vs. RDBMS

Some statistics about Facebook Search (using Cassandra)

- On > 50 GB data
- MySQL
  - Writes 300 ms avg
  - Reads 350 ms avg
- Cassandra
  - Writes 0.12 ms avg
  - Reads 15 ms avg
- Orders of magnitude faster
- What’s the catch? What did we lose?

# Mystery of “X”: CAP Theorem

- Proposed by Eric Brewer (Berkeley)
- In a distributed system you can satisfy at most 2 out of the 3 guarantees:
  1. **Consistency**: all nodes see same data at any time, or reads return latest written value by any client
  2. **Availability**: the system allows operations all the time, and operations return quickly
  3. **Partition-tolerance**: the system continues to work in spite of network partitions (i.e., network failure)

# Why is Consistency Important?

- Consistency = all nodes see same data at any time, or reads return latest written value by any client.
  - Note: this is different from the consistency in ACID
- When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.
- When thousands of customers are looking to book a flight, all updates from any client should be accessible by other clients.

# Why is Availability Important?

- Availability = Reads/writes complete reliably and quickly.
- Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue.
- At Amazon, each added millisecond of latency implies a $6M yearly loss.
- SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients.

# Why is Partition-Tolerance Important?

- Partitions can happen across datacenters when the Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working
- Partitions can also occur within a datacenter, e.g., a rack switch outage
- Still desire system to continue functioning normally under this scenario

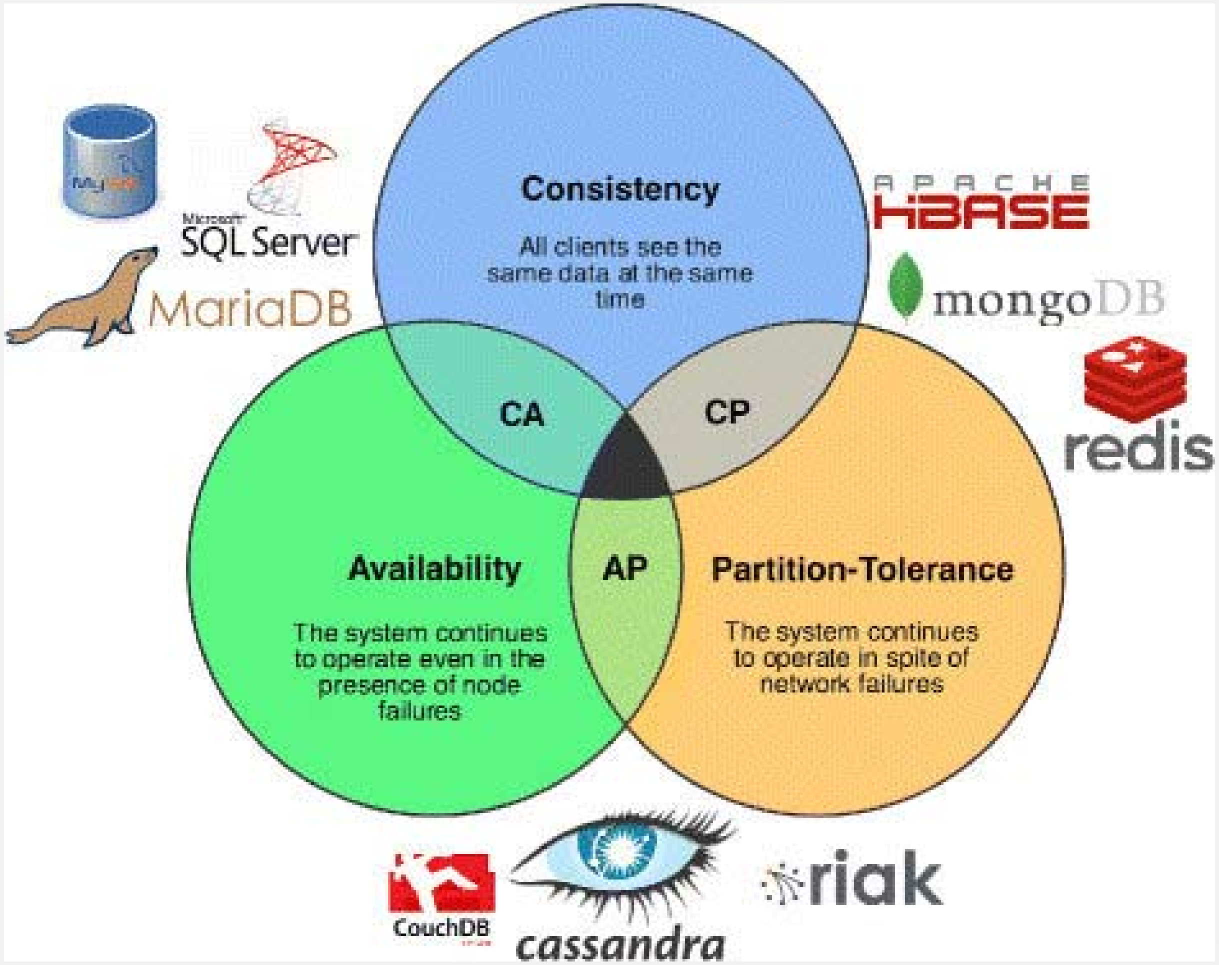
# CAP Theorem Fallout

- Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability

- Cassandra
  - Eventual (weak) consistency, Availability, Partition-tolerance
- Traditional RDBMSs
  - Strong consistency over availability under a partition

# CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C, A, and P.
- When partition-tolerance is important, you have to choose between consistency and availability



# Eventual Consistency

- If all writes stop (to a key), then all its values (replicas) will converge eventually.
- If writes continue, then system always tries to keep converging.
  - Moving “wave” of updated values lagging behind the latest values sent by clients, but always trying to catch up.
- May still return stale values to clients (e.g., if many back-to-back writes).
- But works well when there a few periods of low writes– system converges quickly.

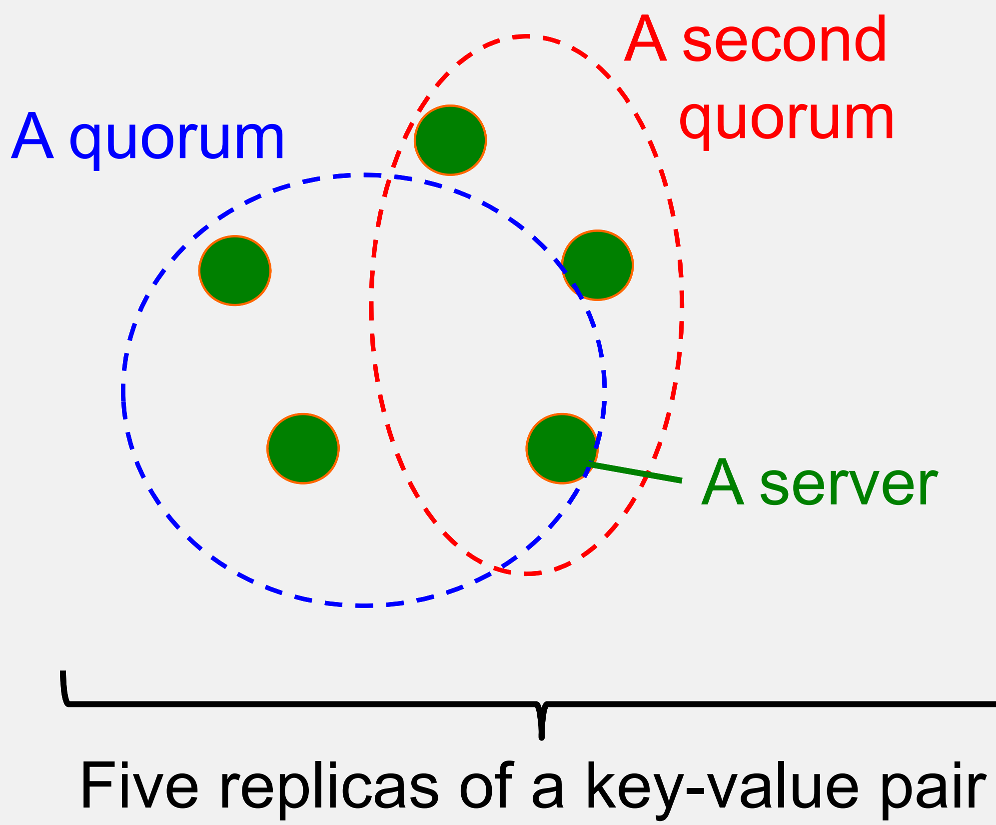
# Back to Cassandra: Mystery of X

- Cassandra has consistency levels
- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator caches write and replies quickly to client
  - ALL: all replicas
    - Ensures strong consistency, but slowest
  - ONE: at least one replica
    - Faster than ALL, but cannot tolerate a failure (e.g., a write operation is only confirmed by one replica)
  - QUORUM: quorum across all replicas in all datacenters (DCs)

# Quorums

In a nutshell:

- Quorum = majority
  - > 50%
- Then any two quorums intersect
  - e.g., Client 1 does a write in red quorum
  - Then client 2 does a read in blue quorum
  - At least one server in blue quorum returns latest write
- Quorum is faster than ALL, but still ensures strong consistency



# Consistency Spectrum

# Spectrum Ends: Eventual Consistency

- Cassandra offers Eventual Consistency
  - If writes to a key stop, all replicas of key will converge