CS4225/CS5425 Big Data Systems for Data Science

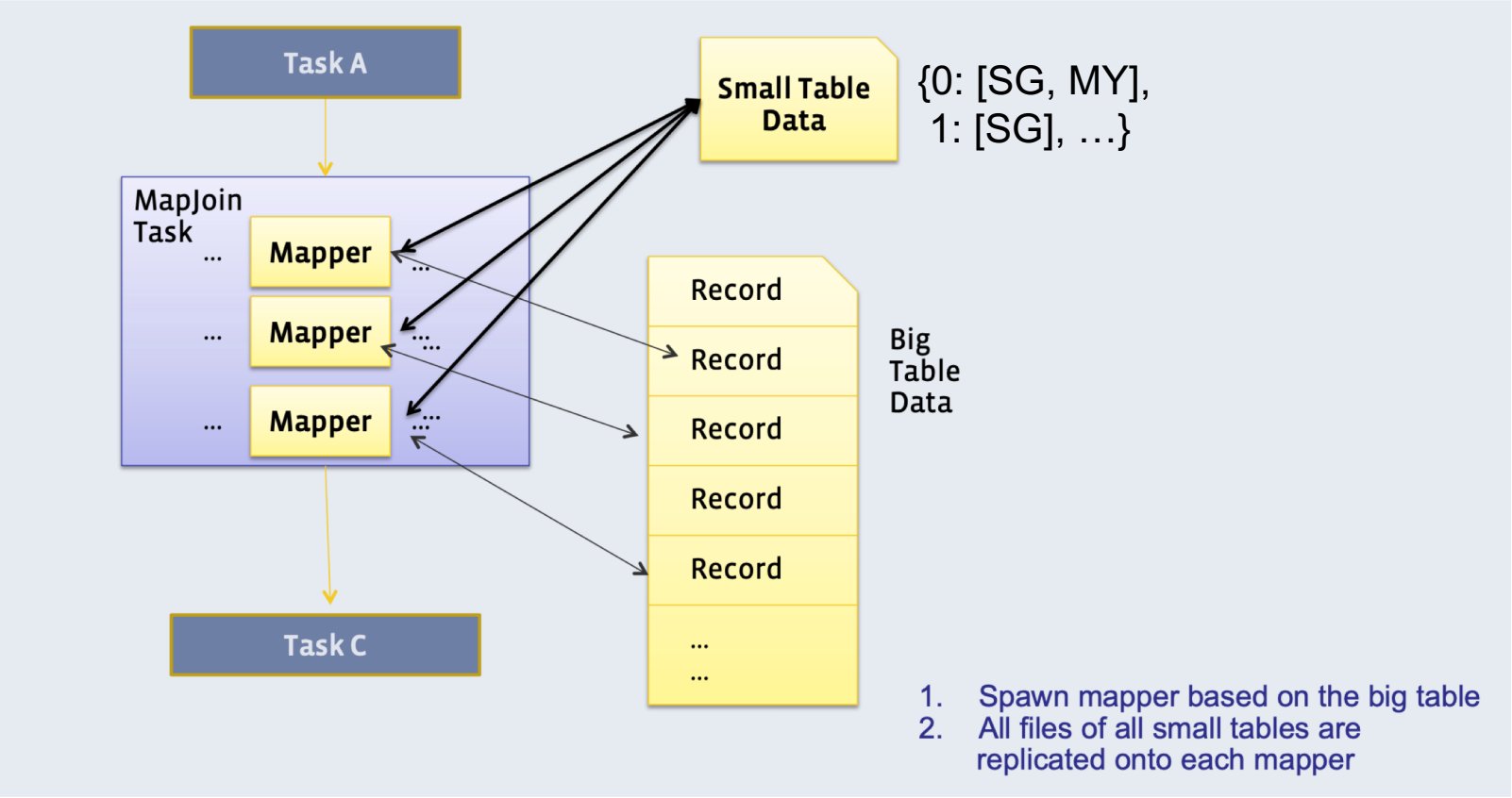
NoSQL Overview

Bryan Hooi
School of Computing
National University of Singapore
bhooi@comp.nus.edu.sg

School of Computing

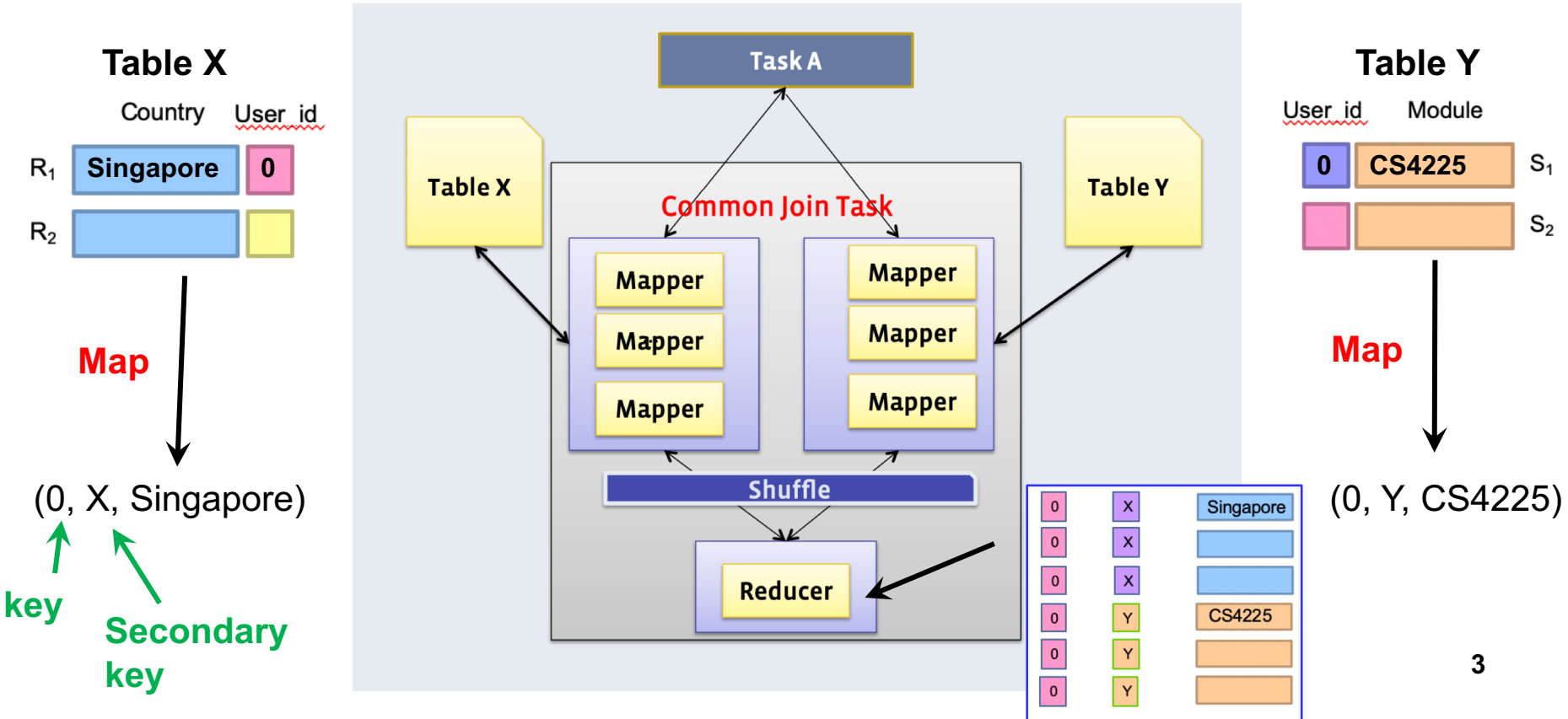
Recap: Broadcast (or Map) Join

- Requires one of the tables to fit in memory
  - All mappers store a copy of the small table, as a hash table
  - They iterate over the big table, and join the records with the small table

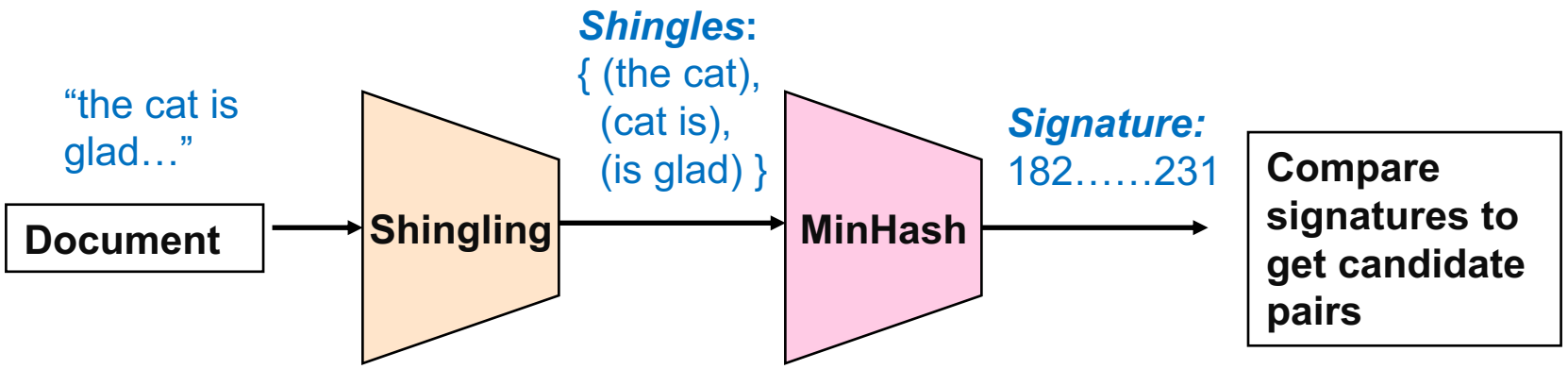


Recap: Reduce-side ('Common') Join

- Doesn't require a dataset to fit in memory, but slower than map-side join
  - Different mappers operate on each table, and emit records, with key as the variable to join by



Recap: Near-Duplicate Document Search



$\Pr[h(C_1) = h(C_2)] = \textit{Jaccard-Sim}(C_1, C_2)$

Recap: Key Property of MinHash

- **Key property**: the probability that two documents have the same MinHash signature is equal to their Jaccard similarity!
- Formally: **Pr**[$h(C_1) = h(C_2)$] = *Jaccard-Sim*(C_1, C_2)
- **Proof** (not required knowledge):

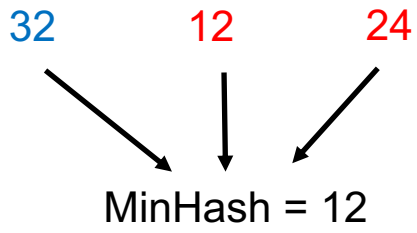
Document 1

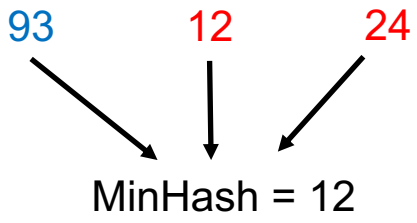
{ (the cat), (cat is), (is glad) }

Document 2

{ (no cat), (cat is), (is glad) }

Hashes:





Set of all our tuples: {(the cat), (no cat), (cat is), (is glad)}

(2 non-shared shingles) (2 shared shingles)

Jaccard sim = 2/4 = 0.5

Among these 4 tuples, each has the same probability of having the smallest hash value.
if one of the red shingles has the smallest hash, the documents will have the same MinHash.
Otherwise, they will have different MinHashes.
=> **Pr**[$h(C_1) = h(C_2)$] = (red shingles / total shingles) = (intersection / union) = Jaccard similarity

Recap: K-Means Algorithm

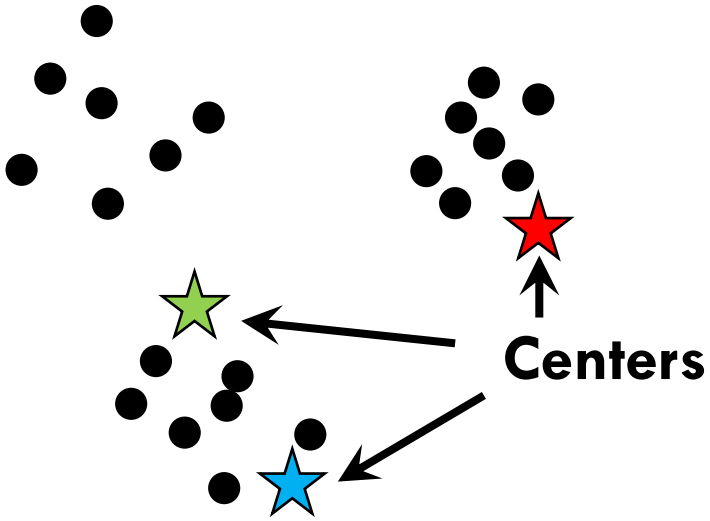
1. Initialization: Pick K random points as centers

2. Repeat:

a) Assignment: assign each point to nearest cluster

b) Update: move each cluster center to average of its assigned points

Stop if no assignments change



MapReduce Implementation v1

This MapReduce job performs a **single iteration** of k-means. It takes in the current positions of the cluster centers **c** (i.e. stars in last slide)

```
1: class Mapper
2:     method Configure()
3:         c ← LoadClusters()
4:     method Map(id i, point p)
5:         n ← NearestClusterID(clusters c, point p)
6:         p ← ExtendPoint(point p)
7:         Emit(clusterid n, point p)
1: class Reducer
2:     method Reduce(clusterid n, points [p1, p2, ...])
3:         s ← InitPointSum()
4:         for all point p ∈ points do
5:             s ← s + p
6:         m ← ComputeCentroid(point s)
7:         Emit(clusterid n, centroid m)
```

Append a "1" at the end of point p's coordinates. This allows us to simply accumulate points by summing (in reducer's line 5), in which case these "1"s keep track of the number of points in a cluster (and used for computing the centroid)

Note: each such emit() call transmits O(d) bytes of data across the network, since a point p has d dimensions
Emit() is called n times since there are n data points, further multiplied by m (# iterations)

MapReduce Implementation v2 (with in-mapper combiner)

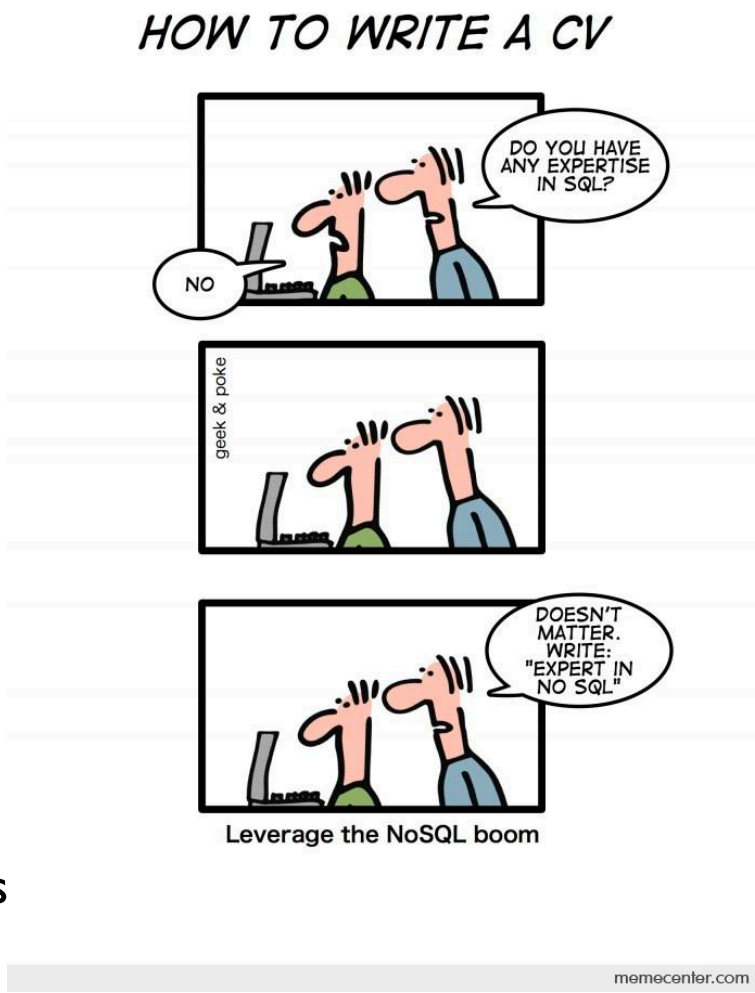
```
1: class Mapper
2:     method Configure()
3:         c ← LoadClusters()
4:         H ← InitAssociativeArray()
5:     method Map(id i, point p)
6:         n ← NearestClusterID(clusters c, point p)
7:         p ← ExtendPoint(point p)
8:         H{n} ← H{n} + p
9:     method Close()
10:        for all clusterid n ∈ H do
11:            Emit(clusterid n, point H{n})
1: class Reducer
2:     method Reduce(clusterid n, points [p1, p2, ...])
3:         s ← InitPointSum()
4:         for all point p ∈ points do
5:             s ← s + p
6:         m ← ComputeCentroid(point s)
7:         Emit(clusterid n, centroid m)
```

In each iteration, Emit() is called once for each cluster, i.e., k times in total. H{n} sums the (extended) points assigned to cluster n.

Today's Plan

1. **What is NoSQL?**
2. Major types of NoSQL systems
3. Key concepts

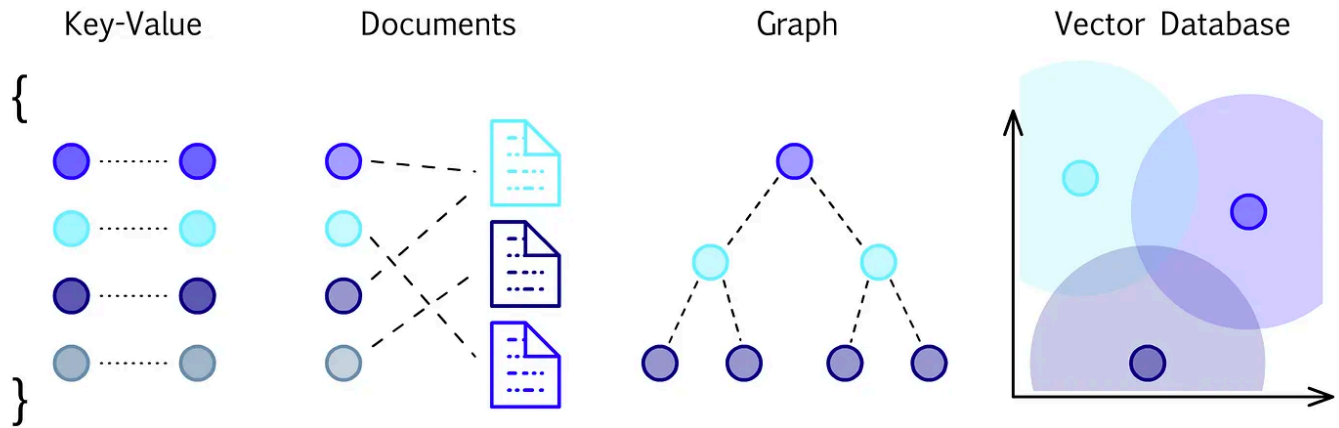
- We will see, but **no need to remember for test / HW:**
  - Historical details
  - Writing code in any NoSQL systems (e.g. MongoDB)
  - Details about any specific NoSQL systems



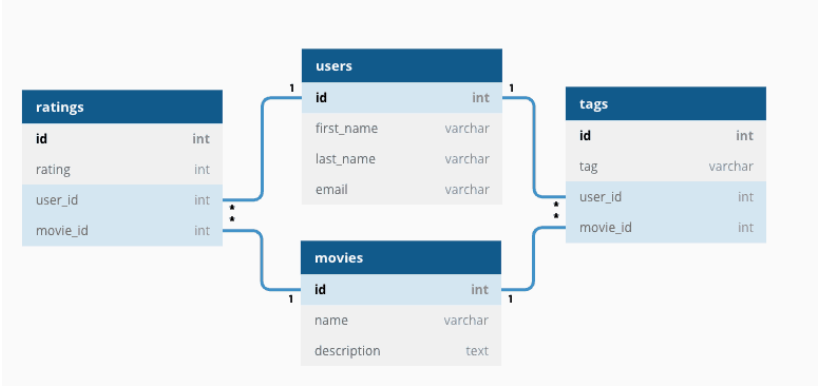
Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

What is NoSQL?

- NoSQL mainly refers to a **non-relational database**, i.e. it stores data in a format other than relational tables
- ”SQL” here refers to traditional relational database management systems (“DBMS”)
  - (This is a slight misnomer; here “SQL” is used to refer to relational databases, not the querying language. In fact, NoSQL systems can involve SQL-like querying languages in many cases.)
- NoSQL has come to stand for “Not Only SQL”, i.e. using relational and non-relational databases alongside one another, each for the tasks they are most suited for



What is NoSQL?



Traditional Relational Database Management System (DBMS)

```
>>> db.inventory.find()
[
  {
    _id: ObjectId("6138d8b1611d9dc433ad73dd"),
    item: 'canvas',
    qty: 100,
    tags: [ 'cotton' ],
    size: { h: 28, w: 35.5, uom: 'cm' }
  },
  {
    _id: ObjectId("6138d917611d9dc433ad73de"),
    item: 'table',
    qty: 5,
    price: 10
  }
]
```

mongoDB. (Document Store)

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

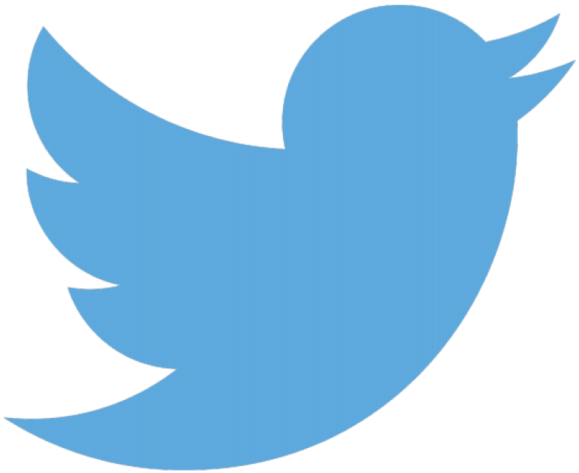
redis (Key-Value Store)

Metaphor: NoSQL = No Rules?

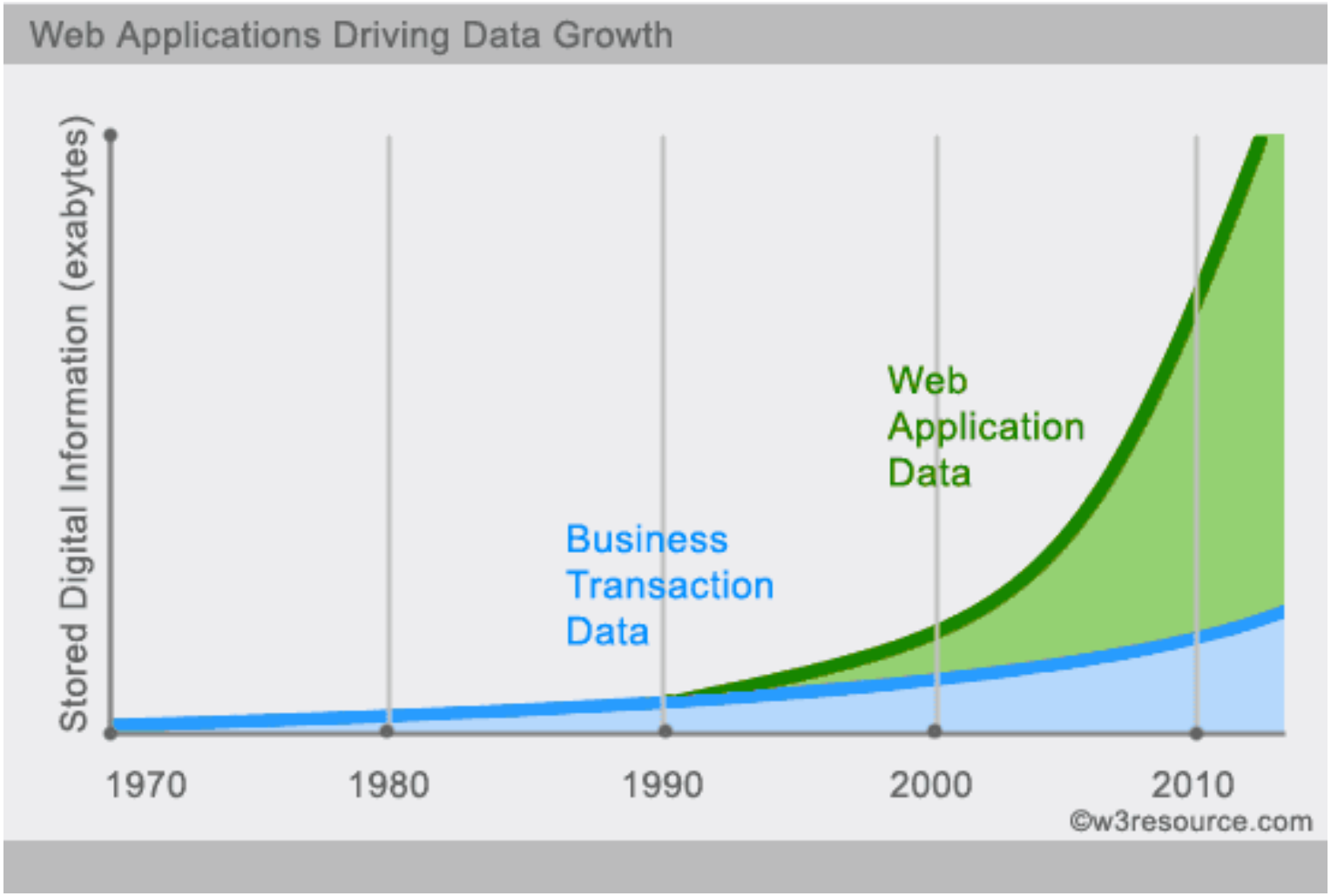
Relational Databases

Non-Relational Databases (NoSQL)

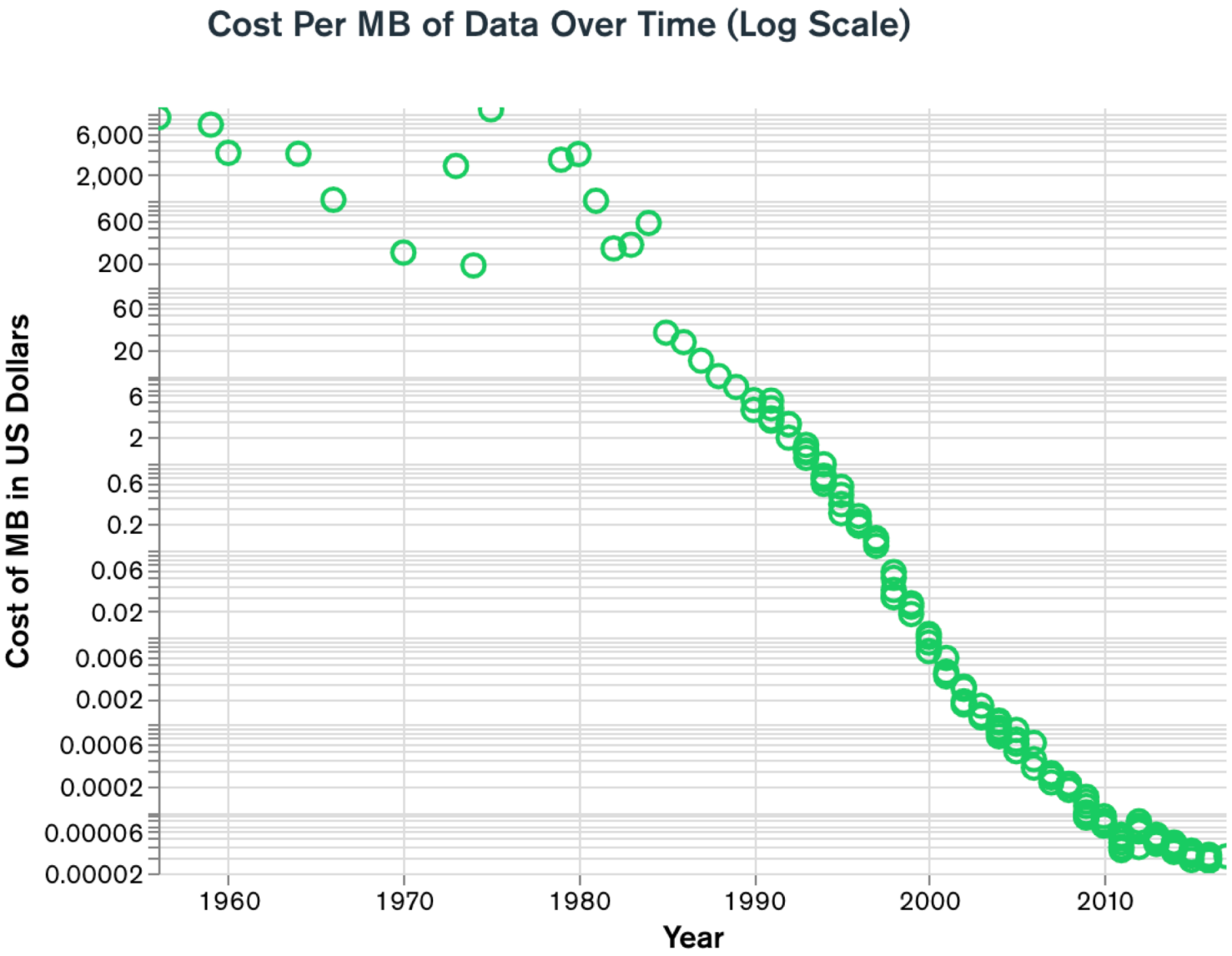
Who uses NoSQL?



Why NoSQL: Growth in Web Applications



Why NoSQL: Volume, Velocity



A Brief Overview of NoSQL Systems

1. Horizontally scalability
2. Replicate/distribute data over many servers

Volume, Velocity

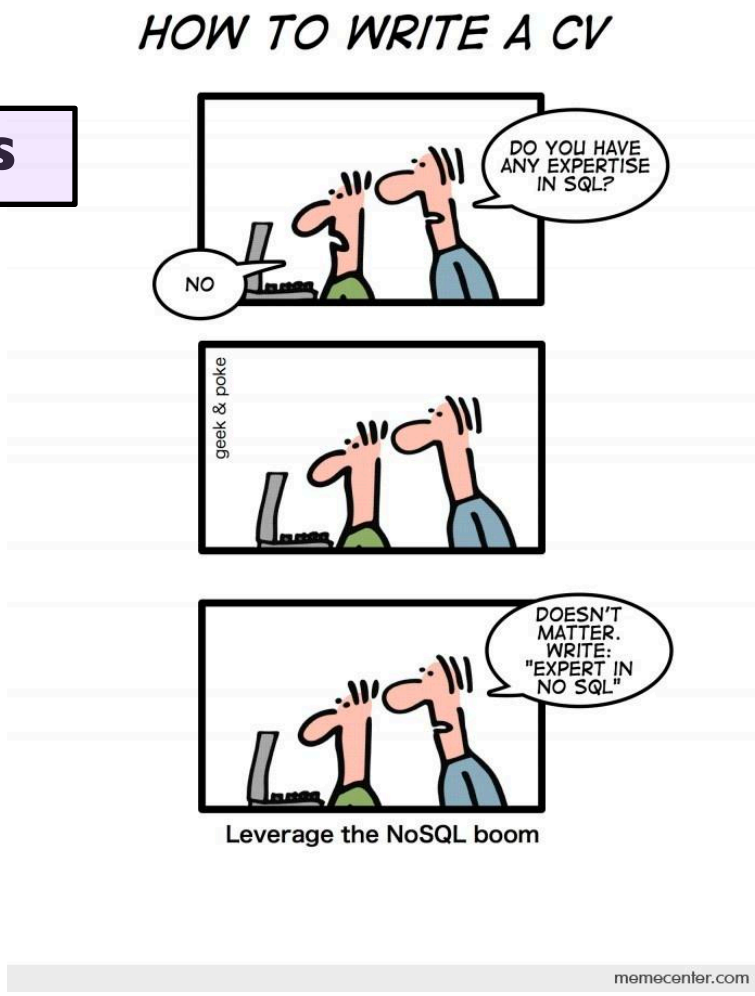
3. Simple call interface
4. Often weaker concurrency model than RDBMS
5. Efficient use of distributed indexes and RAM
6. Flexible schemas

Variety

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Today's Plan

1. What is NoSQL?
2. **Major types of NoSQL systems**
3. Key concepts



Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

(Major) Types of NoSQL databases

- Key-value stores

- Wide column databases

- Document stores

- Graph databases

- Vector databases

Key-Value Stores

Source: Wikipedia (Keychain)

Key-Value Stores: Data Model

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

MAC table

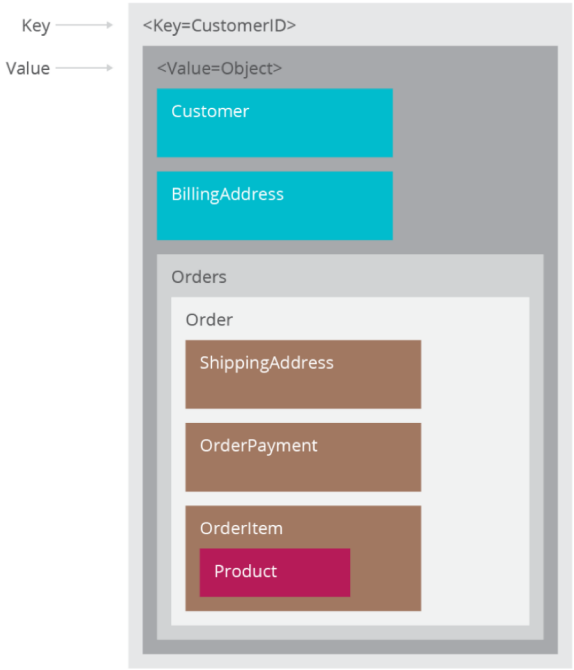
Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

- Stores associations between keys and values
  - Based on Amazon's Dynamo paper (2007)
- Keys are usually primitives and can be queried
  - For example, ints, strings, raw bytes, etc.
- Values can be primitive or complex; usually cannot be queried
  - Examples: ints, strings, lists, JSON, HTML fragments, BLOB (basic large object), etc.

DeCandia, Giuseppe, et al. "Dynamo: Amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41.6 (2007): 205-220.

Key-Value Stores: Operations

- Very simple API:
  - Get – fetch value associated with key
  - Put – set value associated with key
- Optional operations:
  - Multi-get
  - Multi-put
  - Range queries
- Suitable for:
  - Small continuous read and writes
  - Storing 'basic' information (e.g. raw chunks of bytes), or no clear schema
  - When complex queries are not required / rarely required
- Example Applications
  - Storing user sessions
  - Caches
  - User data that is often processed individually: e.g. patient medical data?
    - Only if no queries like 'How many patients who took X treatment recovered?'



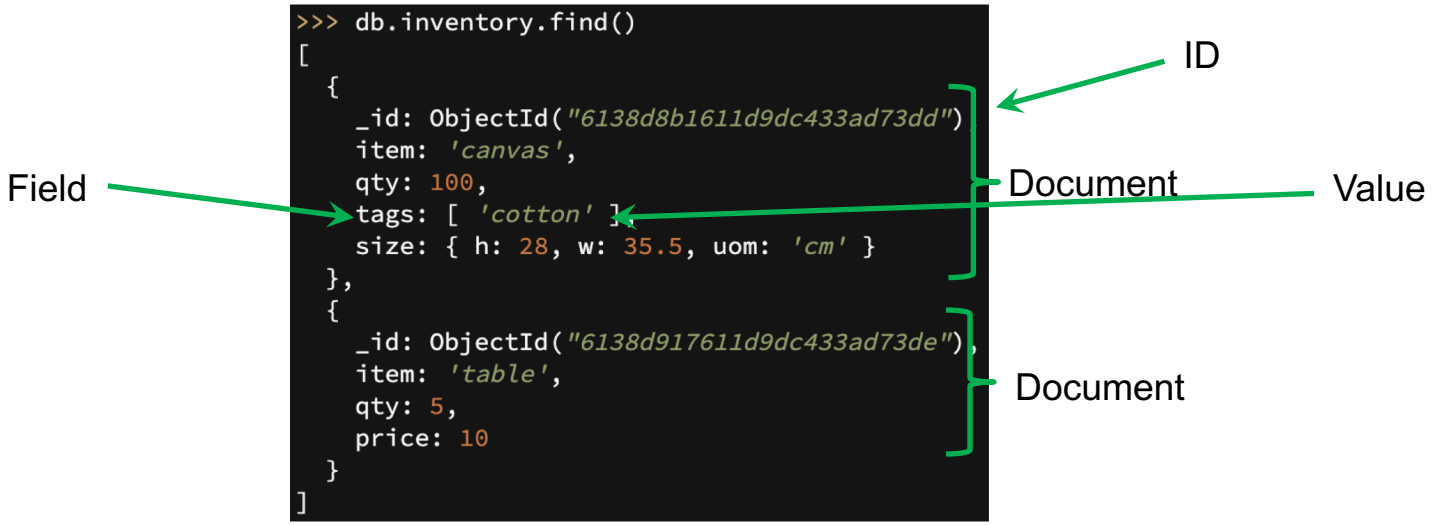
Key-Value Stores: Implementation

- Non-persistent:
  - Just a big in-memory hash table
  - Examples: Memcached, Redis
    - (But: these can also back up the data to disk periodically)
- Persistent
  - Data is stored persistently to disk
  - Examples: RocksDB, Dynamo, Riak

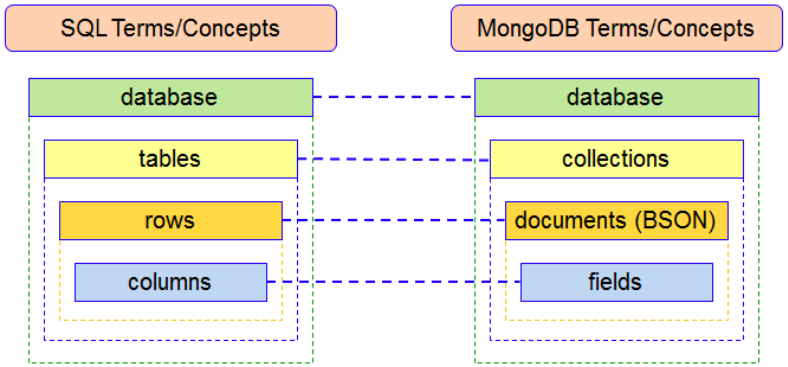
Document Stores

DeCandia, Giuseppe, et al. "Dynamo: Amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41.6 (2007): 205-220.

Document Stores: Document Model



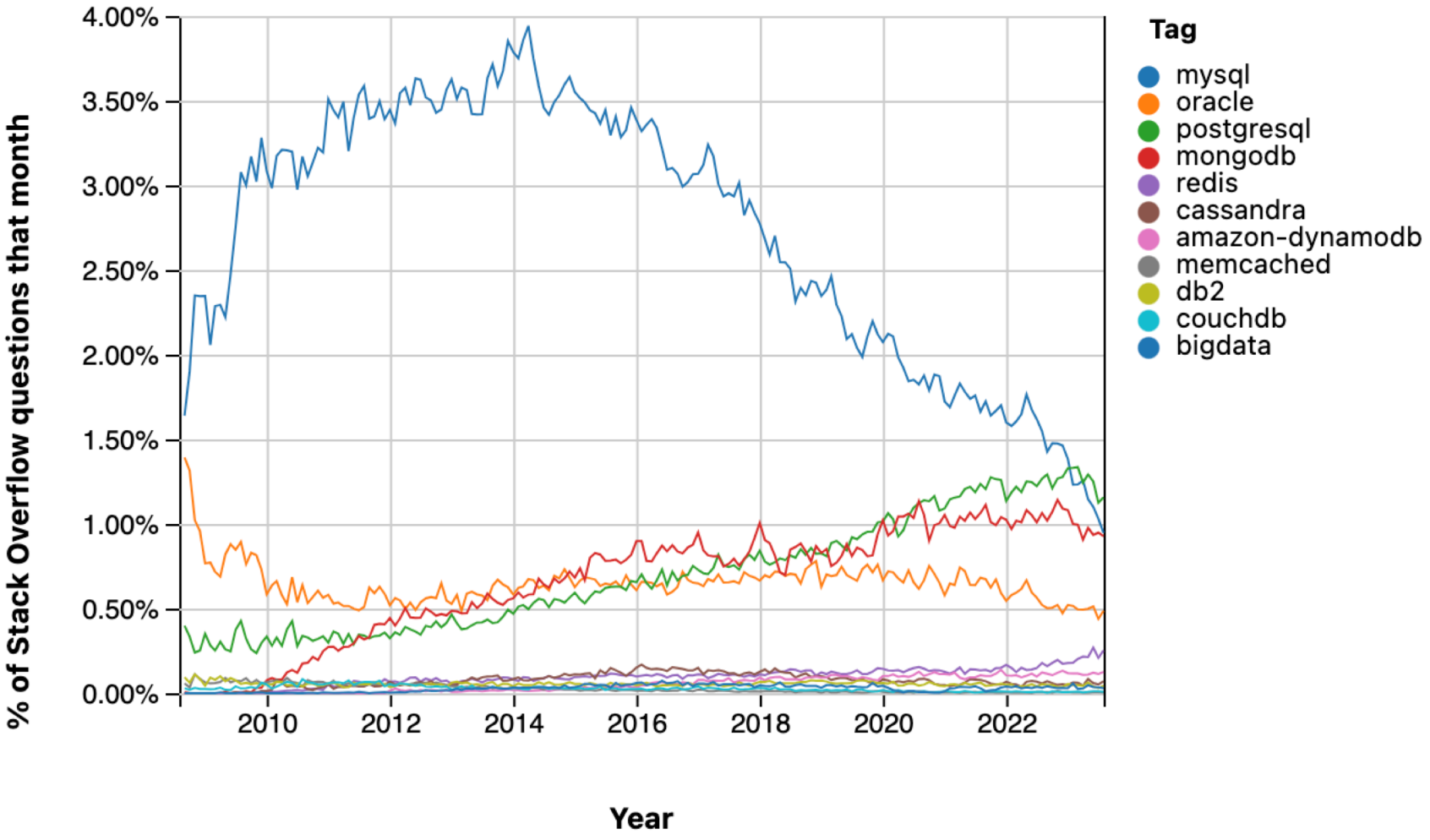
- A database can have multiple **collections**
- Collections have multiple **documents**
- A document is a JSON-like object: it has **fields and values**
  - Different documents can have different fields
  - Can be nested: i.e. JSON objects as values



Popularity of DBMS

Rank			DBMS	Database Model	Score		
Sep 2023	Aug 2023	Sep 2022			Sep 2023	Aug 2023	Sep 2022
1.	1.	1.	Oracle	Relational, Multi-model	1240.88	-1.22	+2.62
2.	2.	2.	MySQL	Relational, Multi-model	1111.49	-18.97	-100.98
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	902.22	-18.60	-24.08
4.	4.	4.	PostgreSQL	Relational, Multi-model	620.75	+0.37	+0.29
5.	5.	5.	MongoDB	Document, Multi-model	439.42	+4.93	-50.21
6.	6.	6.	Redis	Key-value, Multi-model	163.68	+0.72	-17.79
7.	7.	7.	Elasticsearch	Search engine, Multi-model	138.98	-0.94	-12.46
8.	8.	8.	IBM Db2	Relational, Multi-model	136.72	-2.52	-14.67
9.	↑ 10.	↑ 10.	SQLite	Relational	129.20	-0.72	-9.62
10.	↓ 9.	↓ 9.	Microsoft Access	Relational	128.56	-1.78	-11.47
11.	11.	↑ 13.	Snowflake	Relational	120.89	+0.27	+17.39
12.	12.	↓ 11.	Cassandra	Wide column, Multi-model	110.06	+2.67	-9.06
13.	13.	↓ 12.	MariaDB	Relational, Multi-model	100.45	+1.80	-9.70
14.	14.	14.	Splunk	Search engine	91.40	+2.42	-2.65
15.	↑ 16.	↑ 16.	Microsoft Azure SQL Database	Relational, Multi-model	82.73	+3.22	-1.69
16.	↓ 15.	↓ 15.	Amazon DynamoDB	Multi-model	80.91	-2.64	-6.51
17.	↑ 18.	↑ 20.	Databricks	Multi-model	75.18	+3.84	+19.56
18.	↓ 17.	↓ 17.	Hive	Relational	71.83	-1.52	-6.60
19.	19.	↓ 18.	Teradata	Relational, Multi-model	60.33	-0.98	-6.25
20.	20.	↑ 24.	Google BigQuery	Relational	56.46	+2.56	+6.34

Stack Overflow Trends



Source: https://insights.stackoverflow.com/

Document Stores: Querying

- Unlike (basic) key value stores, document stores allow some querying based on the content of a document
- CRUD = Create, Read, Update, Delete

CRUD: Create

In MongoDB

```
db.users.insert (
   {
     name: "sue",
     age: 26,
     status: "A"
   }
)
```

collection

field: value

field: value

field: value

document

In SQL

```
INSERT INTO users
            ( name, age, status )
VALUES      ( "sue", 26, "A" )
```

table

columns

values/row

CRUD: Read

In MongoDB

```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

In SQL

```
SELECT _id, name, address
FROM   users
WHERE  age > 18
LIMIT  5
```

← projection
← table
← select criteria
← cursor modifier

CRUD: Update

In MongoDB

```
db.users.update(                          <-- collection
    { age: { $gt: 18 } },                 <-- update criteria
    { $set: { status: "A" } },            <-- update action
    { multi: true }                       <-- update option
)
```

In SQL

```
UPDATE users              <-- table
SET    status = 'A'       <-- update action
WHERE  age > 18           <-- update criteria
```

CRUD: Delete

In MongoDB

```
db.users.remove(
    { status: "D" }
)
```

collection

remove criteria

In SQL

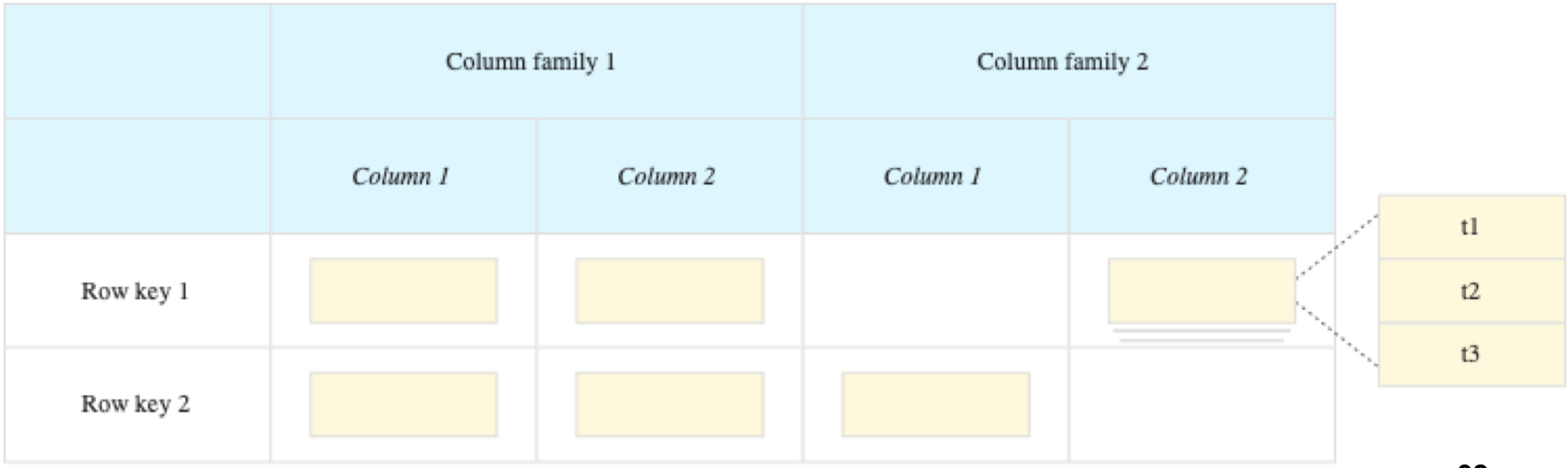
```
DELETE FROM users
WHERE  status = 'D'
```

table

delete criteria

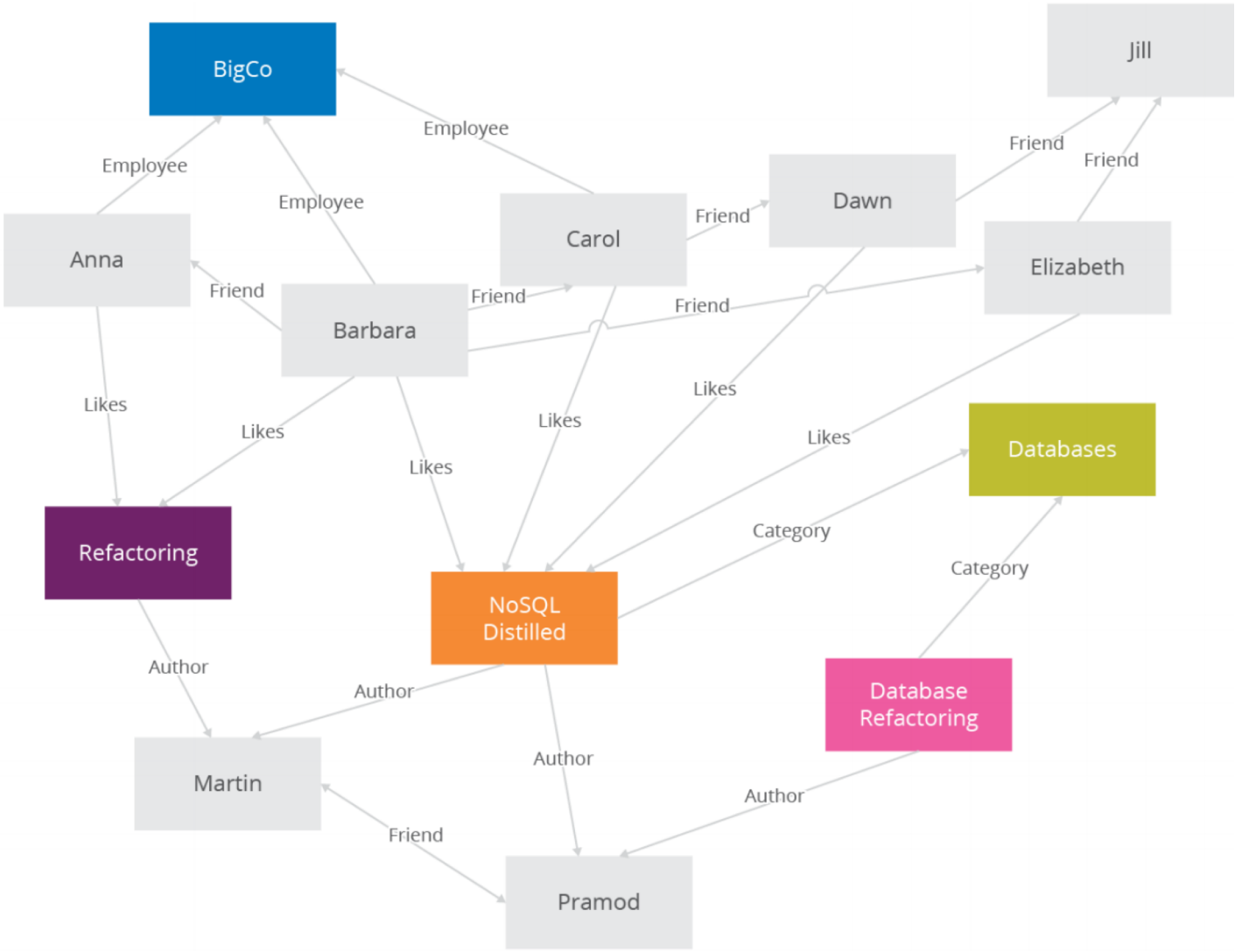
Wide Column Stores

- Rows describe entities
- Related groups of columns are grouped as **column families**
- **Sparsity**: if a column is not used for a row, it doesn't use space
- Examples: BigTable, Cassandra, HBase



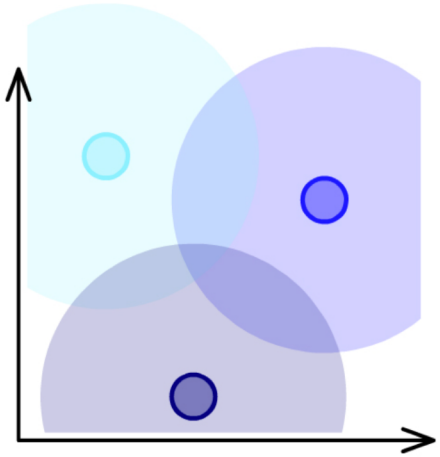
https://cloud.google.com/bigtable/docs/overview

Graph Databases



Vector Databases

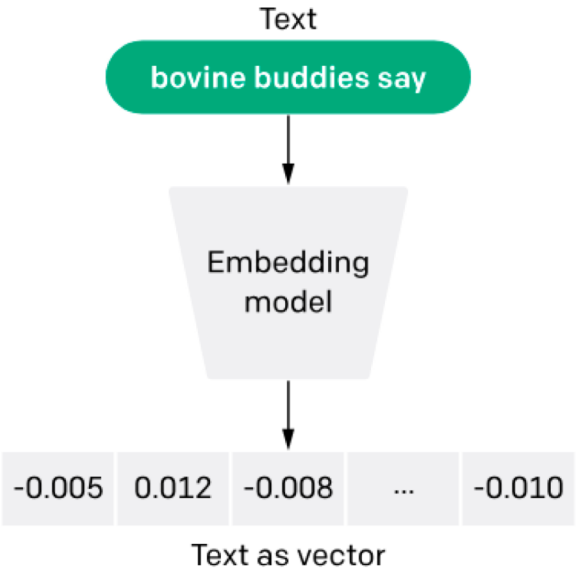
- Store **vectors** (i.e., each row represents a point in d dimensions)
  - Usually dense, numerical, and high-dimensional
- Allow fast **similarity search**, i.e., given a query, retrieve similar neighbors from the database.
  - Earlier in class we discussed similarity search using min-hashing. One of the major types of algorithms used in vector databases is **locality-sensitive hashing** (LSH), which is closely related to min-hashing.
- DB features: scalability, real-time updates, replication
- Examples: Milvus, Redis, Weaviate, MongoDB Atlas

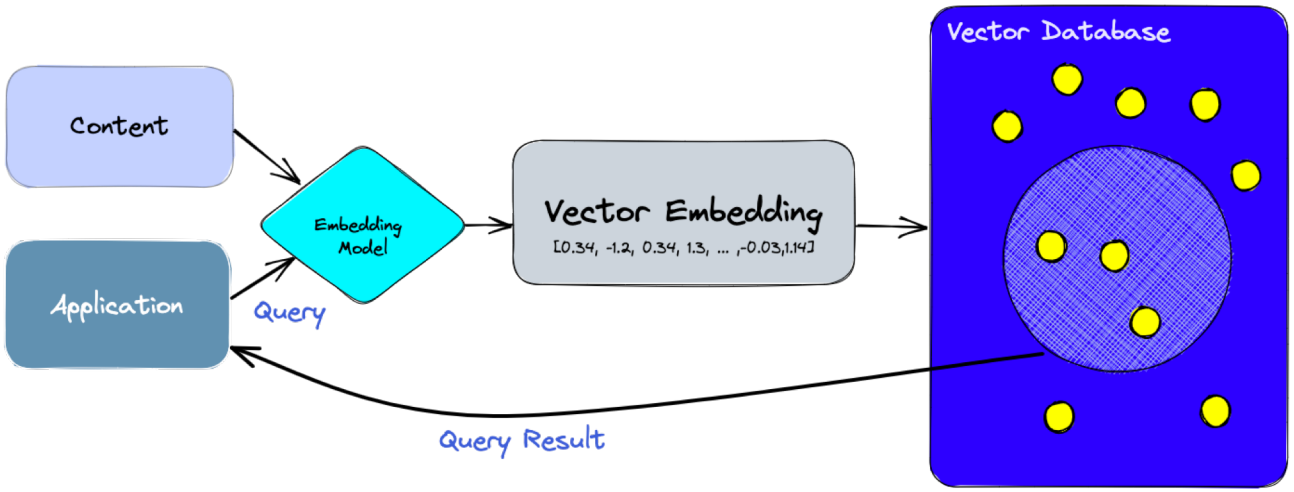


```
>>> collection = Collection(name='my_collection', data=None, schema=schema)
>>> index_params = {
        'index_type': 'IVF_FLAT',
        'params': {'nlist': 1024},
        "metric_type": 'L2'}
>>> collection.create_index('embedding', index_params)
>>> search_param = {
        'data': vector,
        'anns_field': 'embedding',
        'param': {'metric_type': 'L2', 'params': {'nprobe': 16}},
        'limit': 10,
        'expr': 'id_field > 0'
    }
>>> results = collection.search(**search_param)
```

Vector Databases in AI / ML

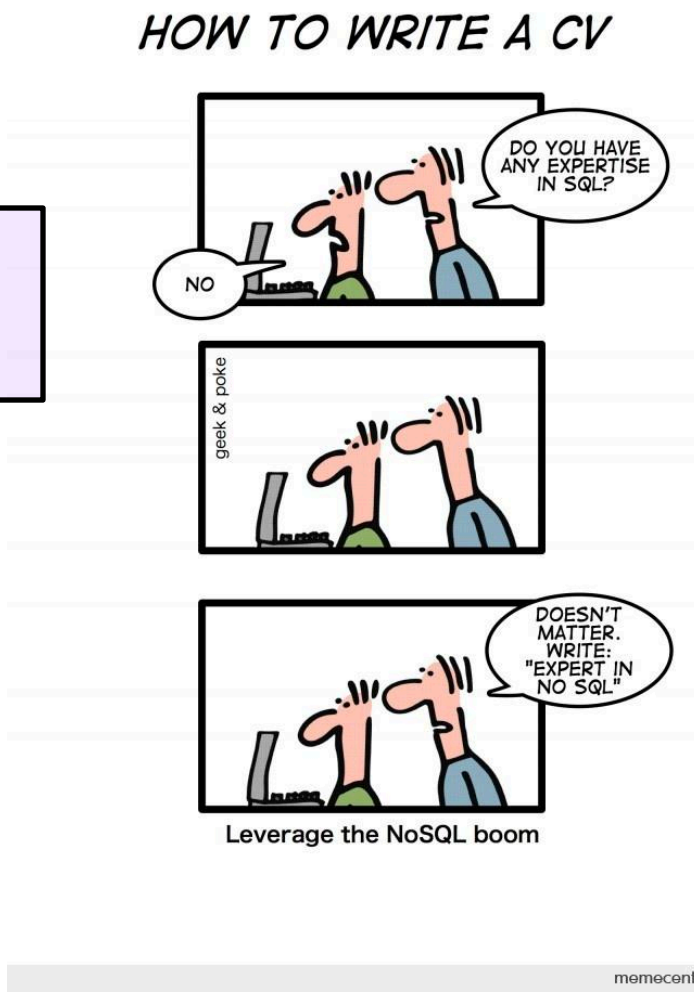
- Vector DBs have become a hot topic recently due to AI / ML
  - Large language models are often used to convert some text into vectors (or **embeddings**) useful for applications like search, recommendation, clustering
  - Similarly, vision models convert images into embeddings
- To quickly search for similar embeddings (e.g., for search / recommendation), efficient and scalable vector DBs are needed





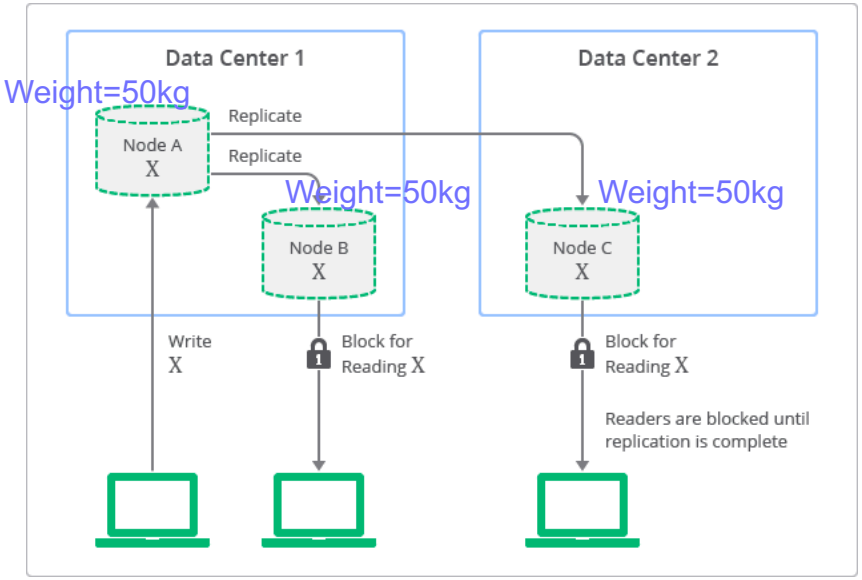
Today's Plan

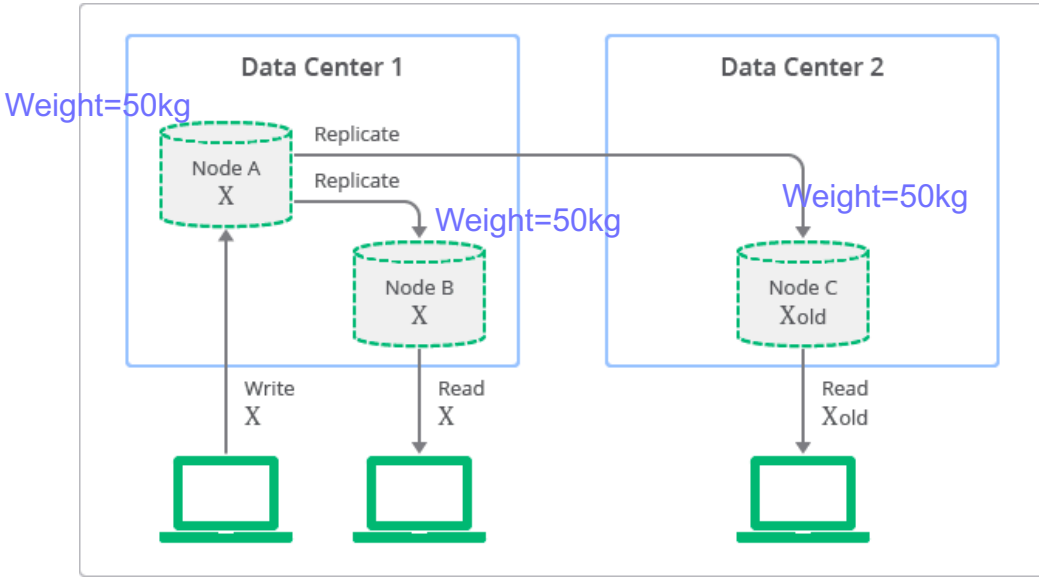
1. What is NoSQL?
2. Major types of NoSQL systems
3. **Key concepts**
   - **Eventual Consistency**
   - Duplication



Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Strong vs Eventual Consistency

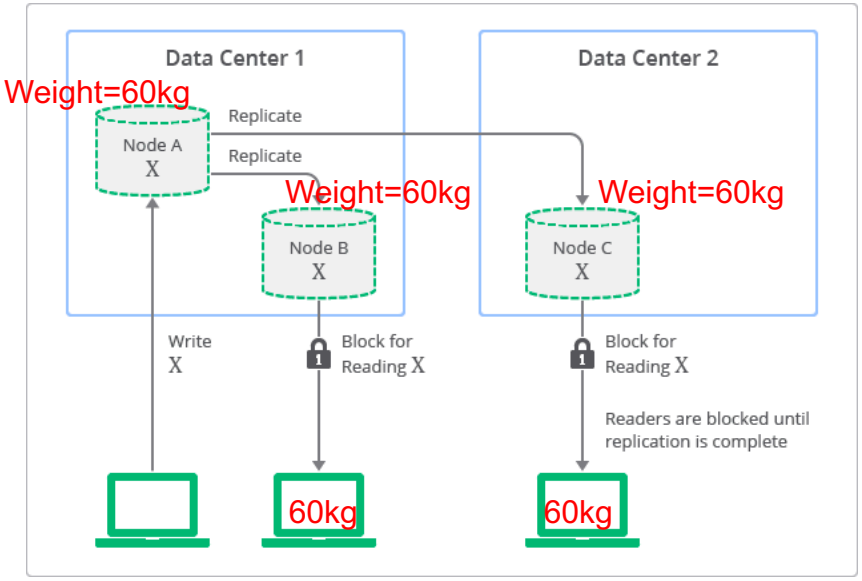




Strong consistency: any reads immediately after an update must give the same result on all observers

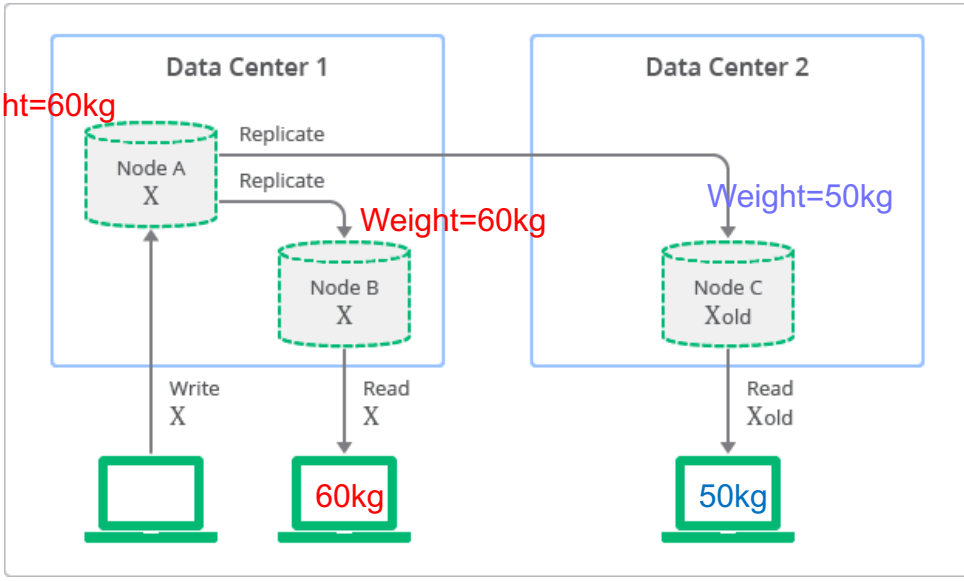
Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value

Strong vs Eventual Consistency



All readers read new value (60kg)

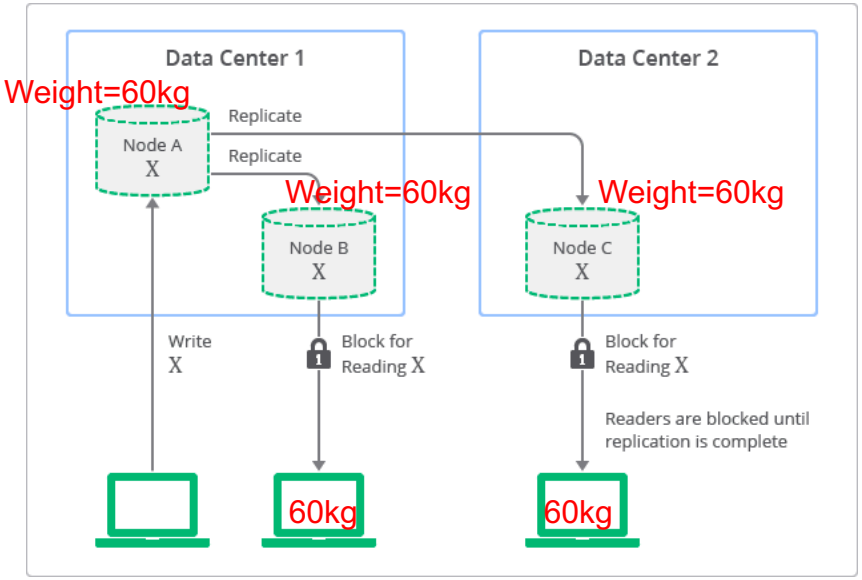
Strong consistency: any reads immediately after an update must give the same result on all observers



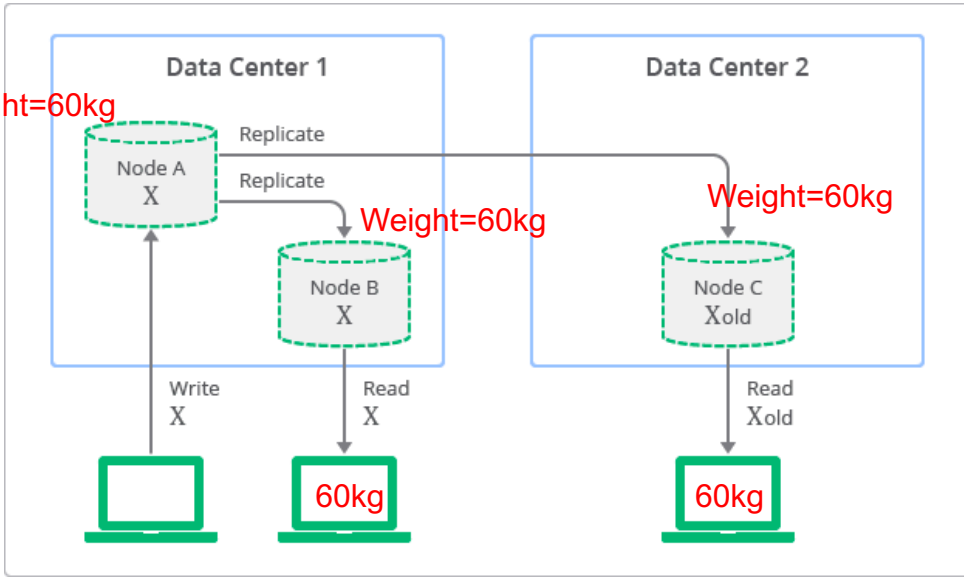
Readers may read old value (50kg)

Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value

Strong vs Eventual Consistency



All readers read new value (60kg)



Eventually, the update is propagated to Node C; after which, readers will read the correct value

Strong consistency: any reads immediately after an update must give the same result on all observers

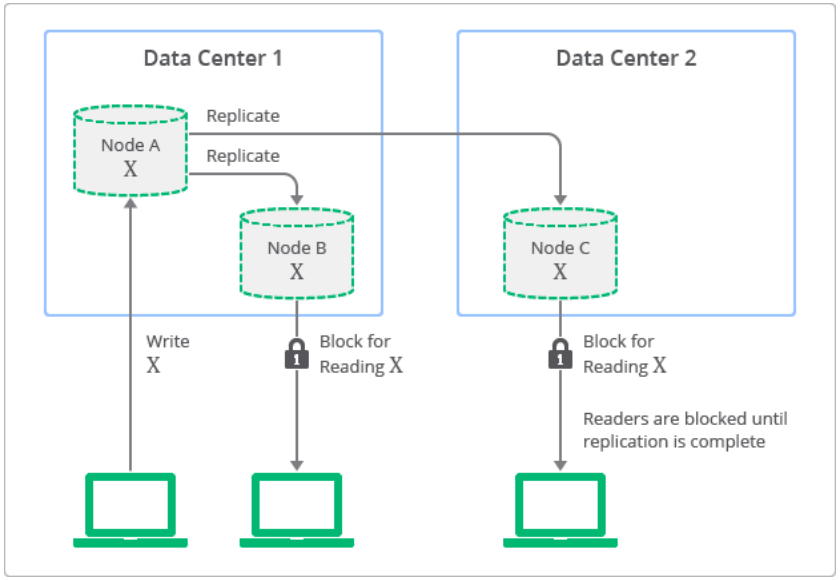
Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value

Eventual Consistency

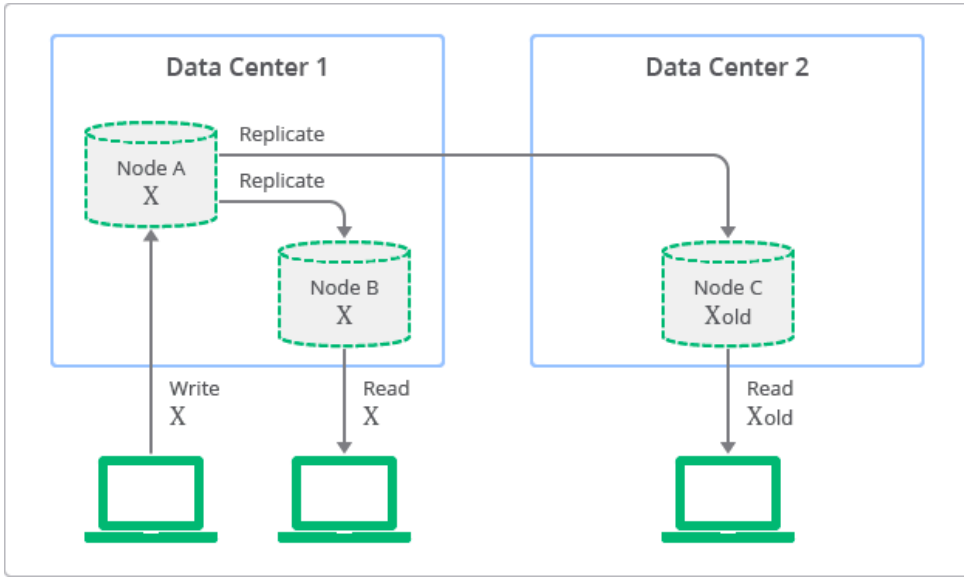
- **ACID vs BASE:** Relational DBMS provide stronger (ACID) guarantees, but many NoSQL system relax this to weaker "BASE" approach:
  - **Basically Available**: basic reading and writing operations are available most of the time
  - **Soft State**: without guarantees, we only have some probability of knowing the state at any time
  - **Eventually Consistent**: Contrast to **"strong consistency"**:

Strong vs Eventual Consistency

Strong consistency



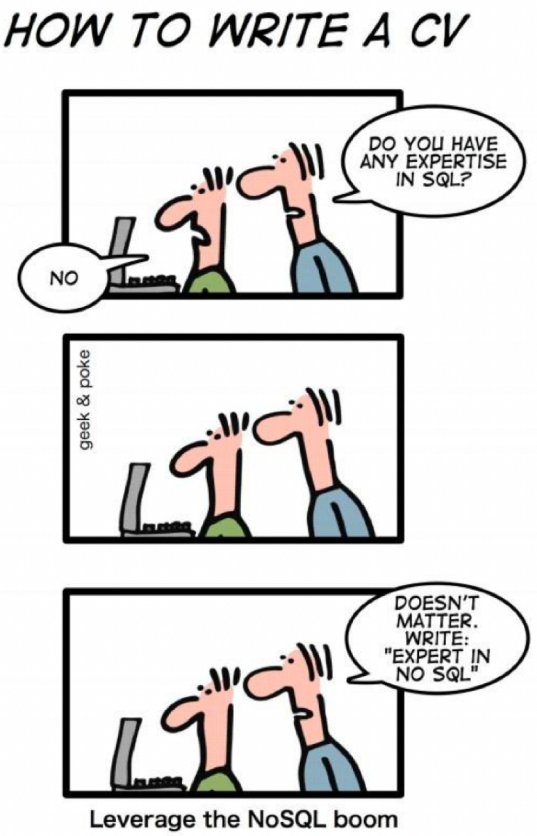
Eventual consistency



- **Implications**: eventual consistency offers better **availability** at the cost of a much weaker consistency guarantee. This may be acceptable for some applications (e.g. statistical queries, tweets, social network feed, …) but not for others (e.g. financial transactions)
  - Note that while NoSQL systems allow for weaker consistency guarantees, many more recent systems / versions are often configurable, i.e. can be configured for multiple different consistency levels (including strong) – *'tunable consistency'*

Today's Plan

1. What is NoSQL?
2. Major types of NoSQL systems
3. **Key concepts**
   - Eventual Consistency
   - **Duplication**



Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

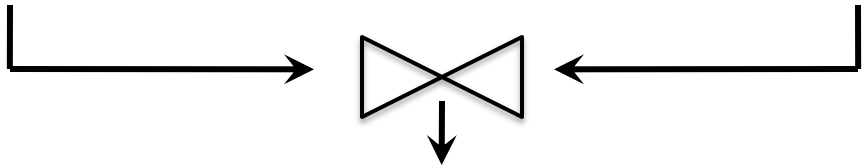
Recall: Joins in RDBMS

Users

user_id	name
100	Alice
101	Bob

Posts

post_id	post	user_id
1	Hello world	100
2	Bye world	101



post_id	post	user_id	name
1	Hello world	100	Alice
2	Bye world	101	Bob

What if we want to display posts with usernames?

Users

```
[
  {
    "user_id": 100,
    "name": "Alice"
  },
  {
    "user_id": 101,
    "name": "Bob"
  },
]
```

Posts

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101
  },
]
```

Posts (+ username)

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100,
    "name": "Alice",
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101,
    "name: "Bob"
  },
]
```

- Answer 1: some NoSQL databases do support joins (e.g. later versions of MongoDB)
- Answer 2: Duplication (i.e. 'denormalization')
  - 'Storage is cheap: why not just duplicate data to improve efficiency?'
  - Tables are designed around the queries we expect to receive. This is beneficial especially when we mostly need to process a fixed type of queries
  - Leads to a new problem: what if user changes their name? (this needs to be propagated to multiple tables)

Conclusion: Pros & Cons of NoSQL Systems

Pros

+ **Flexible / dynamic schema**: suitable for less well-structured data

+ **Horizontal scalability**: we will discuss this more next week

+ **High performance and availability**: due to their relaxed consistency model and fast reads / writes

Cons

- **No declarative query language**: query logic (e.g. joins) may have to be handled on the application side, which can add additional programming

- **Weaker consistency guarantees**: application may receive stale data that may need to be handled on the application side

Conclusion: no one size fits all. Depends on needs of application: whether denormalization is suitable; complexity of queries (joins vs simple read/writes); importance of consistency (e.g. financial transactions vs tweets); data volume / need for availability.

- Binary view is oversimplified: NoSQL databases often still have SQL-like query languages (e.g. Cassandra) and tunable consistency levels.
- Other alternative solutions: "NewSQL", distributed SQL (e.g. Presto), PostgreSQL JSON, …
- Need to understand tradeoffs to make an informed decision!

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Acknowledgements

- CS4225 slides by He Bingsheng
- Penn State CMPSC 431W Database Management Systems (Fall 2015) – McGrawHill, Wang-Chien Lee, Yu-San Lin
- mongodb.com, redis.com, openai.com, pinecone.io
- Otter video from https://www.reddit.com/r/singapore/comments/mezgg3/otters_on_the_grass_at_botanic_gardens_today/