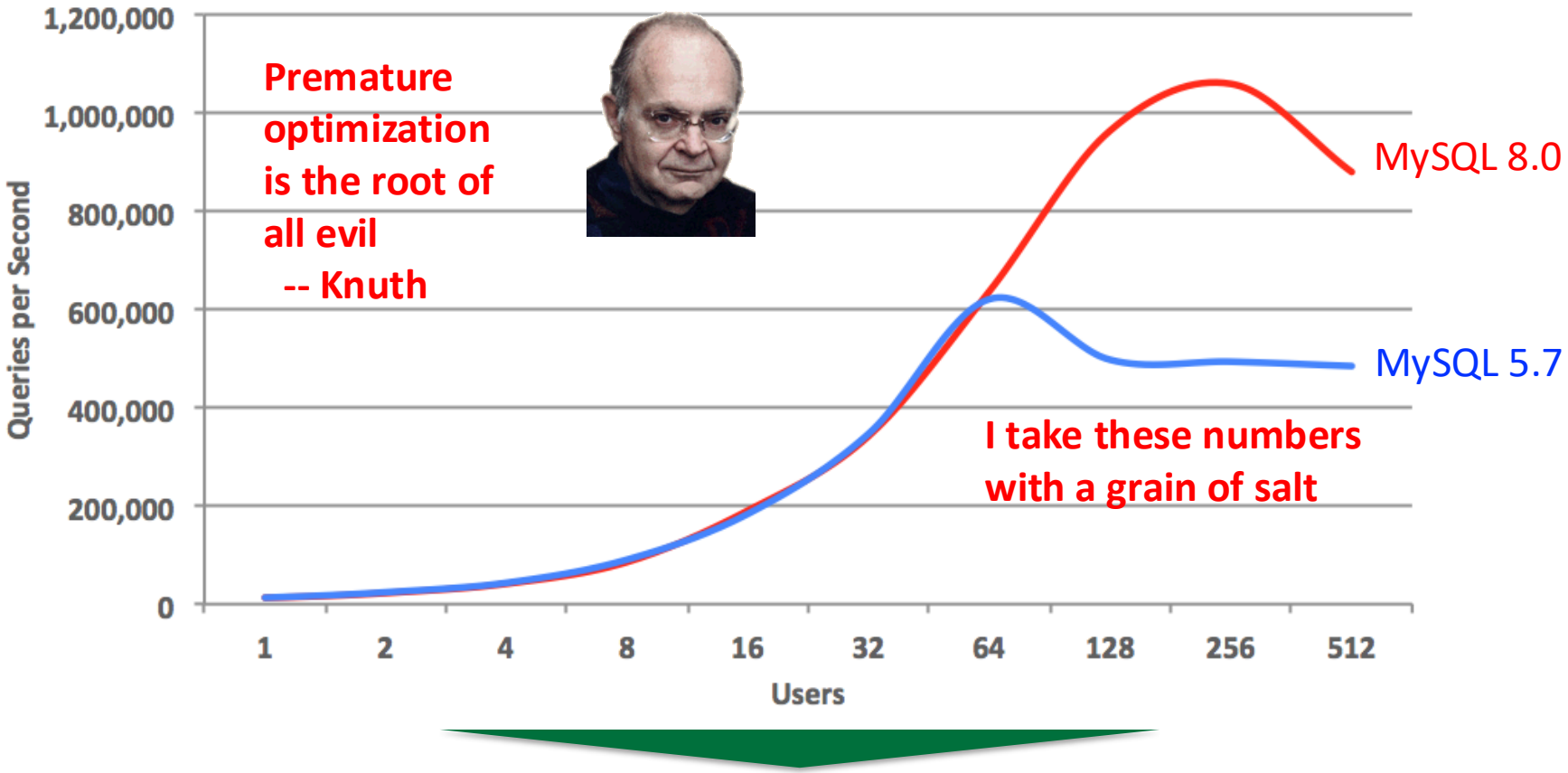
CS 61:
Database Systems

Alternatives

A single database can handle many thousands of transactions per second



Your start up that you're certain will be a smashing success in the market is unlikely to overwhelm a database running on a single reasonable server for quite some time

-- Pierson

Scale vertically – get a bigger box

Scale horizontally – get more boxes

https://www.mysql.com/why-mysql/benchmarks/

Let's estimate back of the envelope performance

Assume:

Each user interaction takes 10 queries on average due to normalization

Average of 30 user interactions/visitor/day

Database can handle 100,000 queries per second

If you exceed these numbers, you'll need some help from someone who took more than an introductory database class!

Max user interactions/second:

$$\frac{100{,}000 \text{ queries}}{\text{second}} \times \frac{1 \text{ interaction}}{10 \text{ queries}} = \frac{10{,}000 \text{ interactions}}{\text{second}}$$

Max user interactions/day:

$$\frac{10{,}000 \text{ interactions}}{\text{second}} \times \frac{60*60*24 \text{ seconds}}{\text{day}} = \frac{864\text{M interactions}}{\text{day}}$$

Max user visits/day:

$$\frac{864\text{M interactions}}{\text{day}} \times \frac{1 \text{ visitor}}{30 \text{ interactions}} = \frac{28.8\text{M visitors}}{\text{day}}^{*}$$

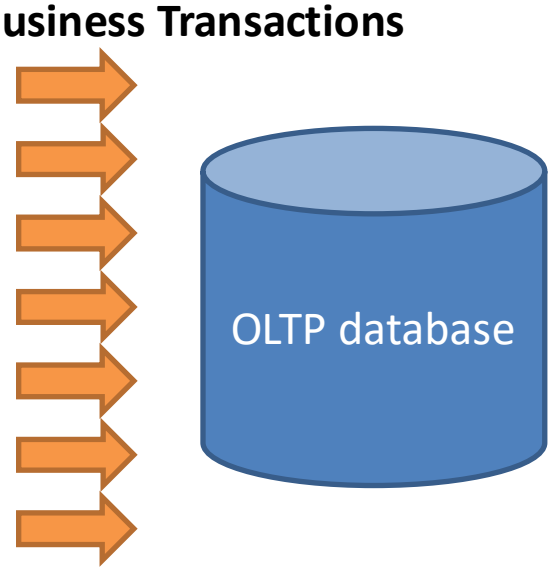
Assuming uniform demand in time

Source: percona.com

Agenda

1. Data warehouses
2. Distributed systems
3. MongoDB overview
4. MongoDB queries
5. Restaurant example
6. Course wrap up

OLTP databases are designed for operational performance



Business intelligence queries can hamper transaction performance

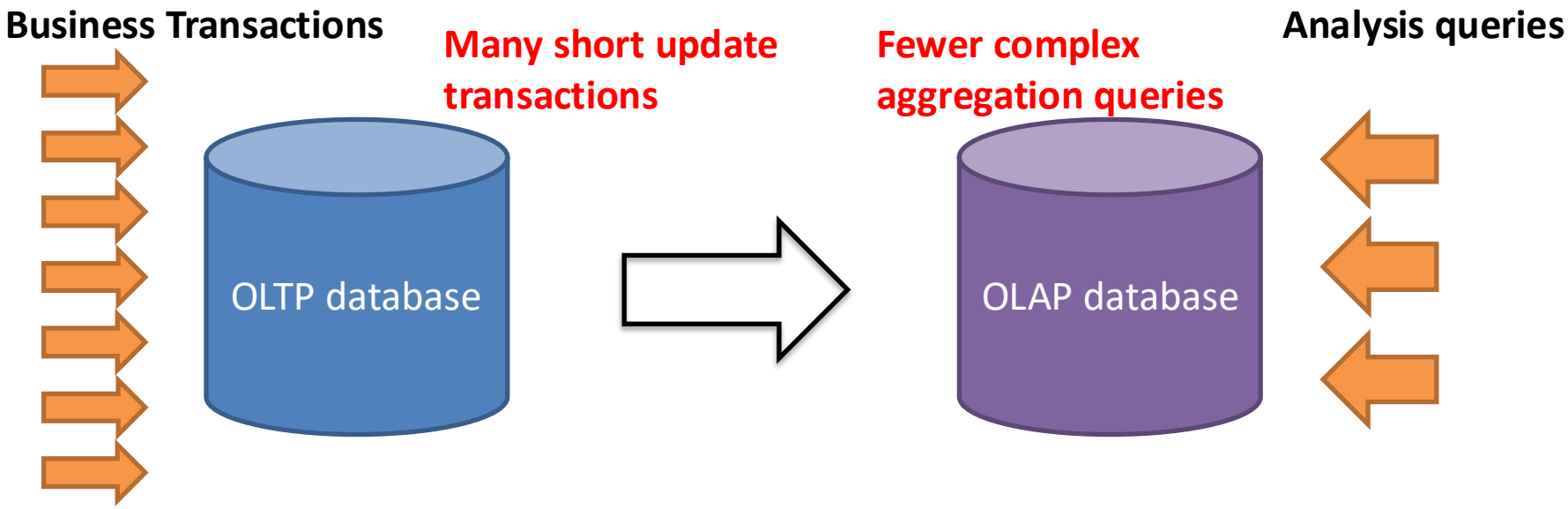
Operational data often not well suited for business analysis

Solution: create a separate database optimized for data analysis

Online Transaction Processing databases (OLTP)

- Our focus thus far
- Handles daily business operations
- Data often highly normalized
- Transactions mainly inserts or updates
- Speed is crucial!

We need tools to analyze this data for insight



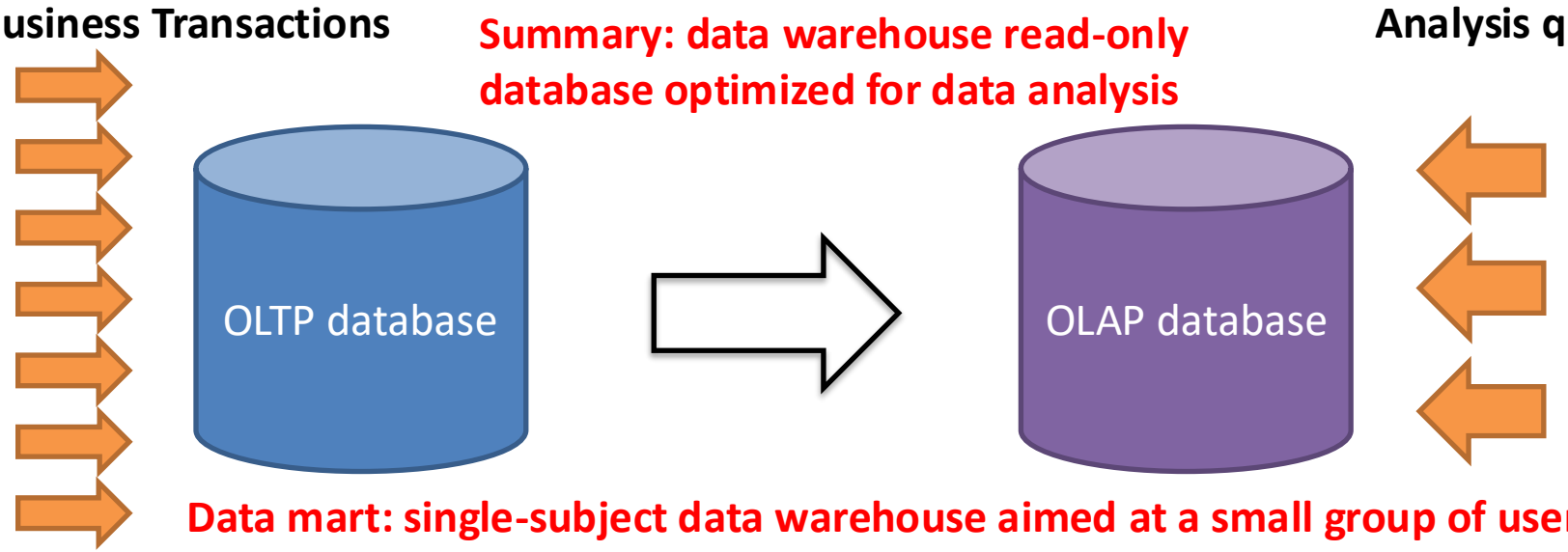
Online Transaction Processing databases (OLTP)

- Our focus thus far
- Handles daily business operations
- Data often highly normalized
- Transactions mainly inserts or updates
- Speed is crucial!

Online Analytical Processing databases (OLAP)

- Designed for analysis of "so what" of the data (get insight into data) to make decisions
- Contains summaries of data (e.g., product sales by year by region)
- Transactions mainly reads
- Speed less critical

We need tools to analyze this data for insight



Extract, Transform, Load (ETL) data from OLTP to OLAP database

- Extract data periodically from OLTP (and other sources) in a batch (how often?)
- Filter, integrate, and aggregate data (what level of aggregation?)
- Store data for easy business analysis (denormalize data! Yep, you read that right!)
- Data warehouse is an "integrated, subject-oriented, time-variant, nonvolatile" collection of data
  - Integrated – consolidate data from many sources
  - Subject-oriented – data optimized by topic such as sales, marketing, finance
  - Time-variant – represent the flow of data through time (even projected data)
  - Nonvolatile – data in warehouse not removed (or updated unless error)

Bill Inmon and Chuck Kelly, "The twelve rules of data warehouse for a client/server world", *Data Management Review*, May 1994

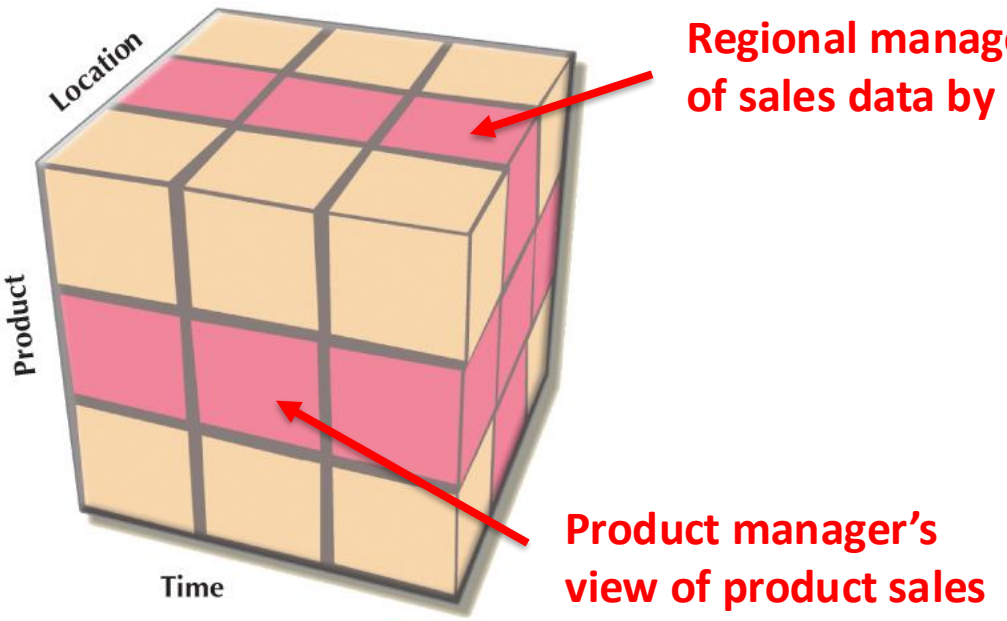
A data warehouse should conform to 12 rules

12 rules for a data warehouse

Rule	Description
1	The data warehouse and operational environments are separated
2	The data warehouse is integrated (data from multiple sources)
3	The data warehouse contains historical data over a long time
4	The data warehouse data is a snapshot captured at a given point in time
5	The data warehouse is subject oriented
6	The data warehouse data is mainly read-only with periodic batch updates
7	The data warehouse is data driven; operational database is process driven
8	The data warehouse contains data with several levels of detail (current/old, summarized at various levels)
9	The data warehouse is characterized by read-only queries of very large data sets
10	The data warehouse has a system that traces data sources, transforms, and storage
11	The data warehouse's metadata is critically important
12	The data warehouse enforces optimal use of the data by end users

Bill Inmon and Chuck Kelly, "The twelve rules of data warehouse for a client/server world", *Data Management Review*, May 1994

Data warehouses are often implemented using a Star Schema

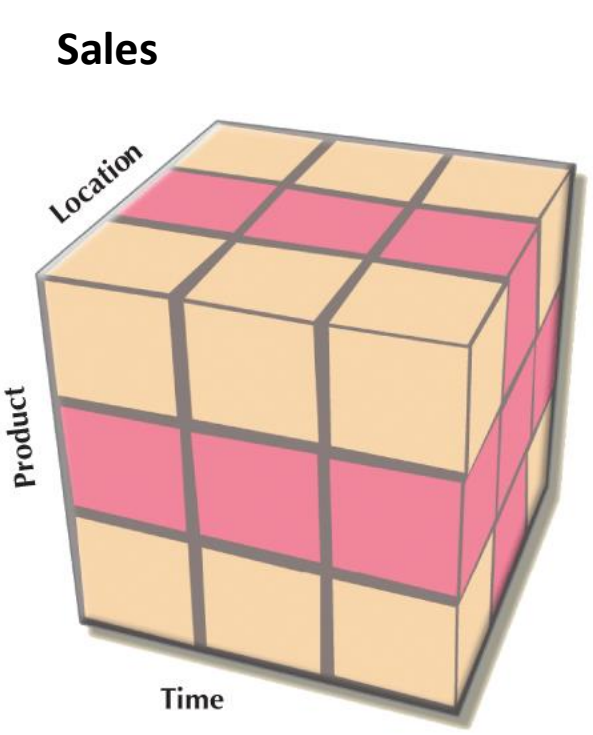


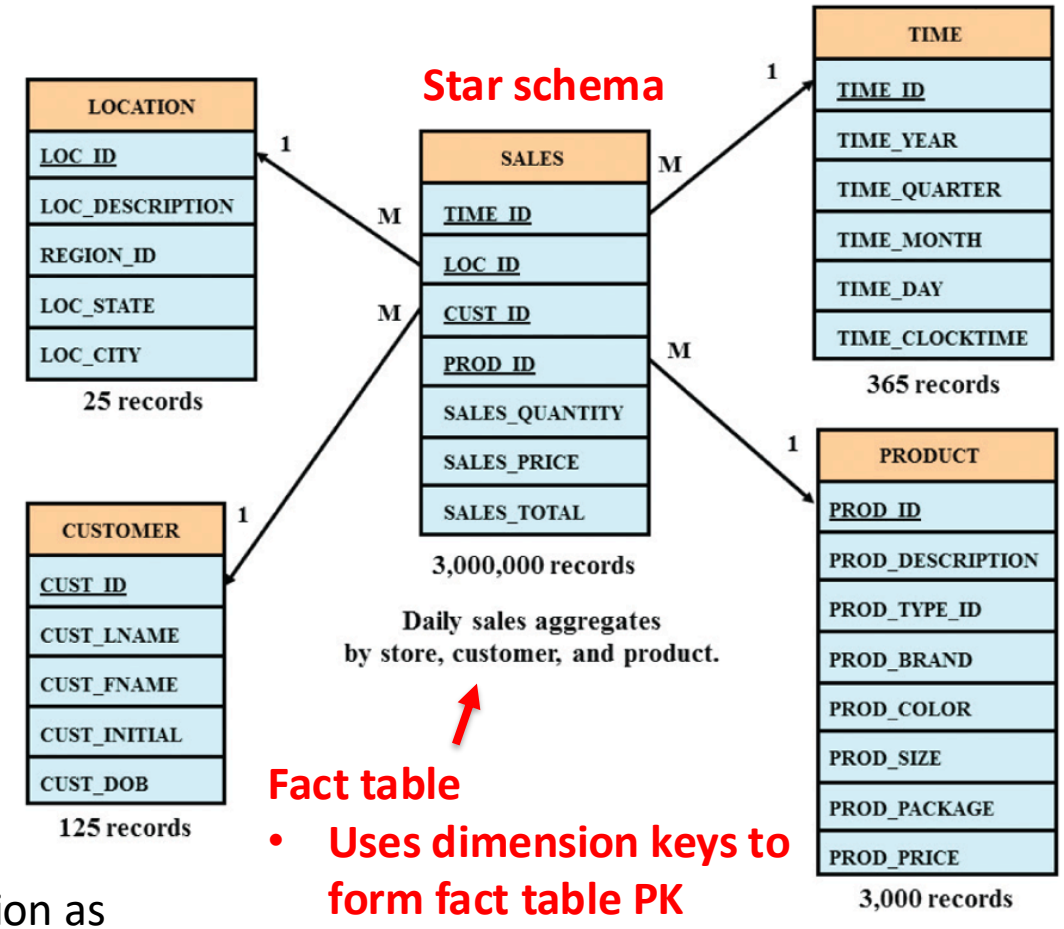
Data cube

- Create conceptual cube with dimension as sides of cube
- Each cube element contains a fact (sales $)
- Allows rapid slicing and dicing
- Uses fact and dimension tables to store data

Adapted from Coronel and Morris

Data warehouses are often implemented using a Star Schema





Data cube

- Create conceptual cube with dimension as sides of cube
- Each cube element contains a fact (sales $)
- Allows rapid slicing and dicing
- Uses fact and dimension tables to store data

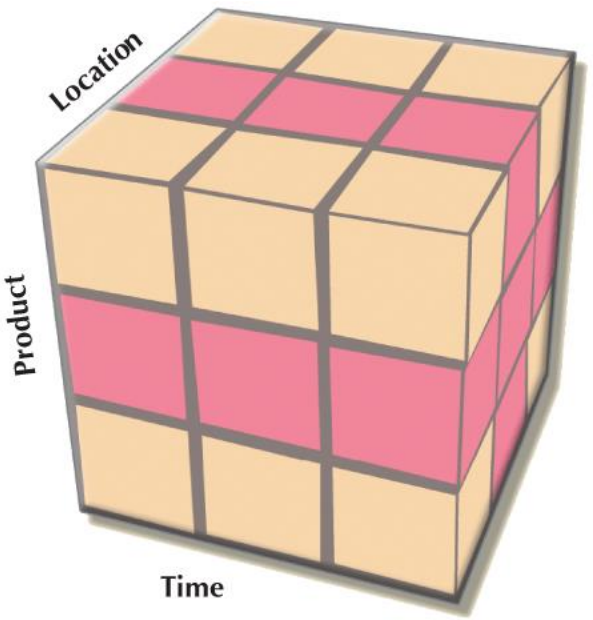
Fact table

- **Uses dimension keys to form fact table PK**
- **Denormalized data (same data stored many times)**
- **May have multiple attributes**

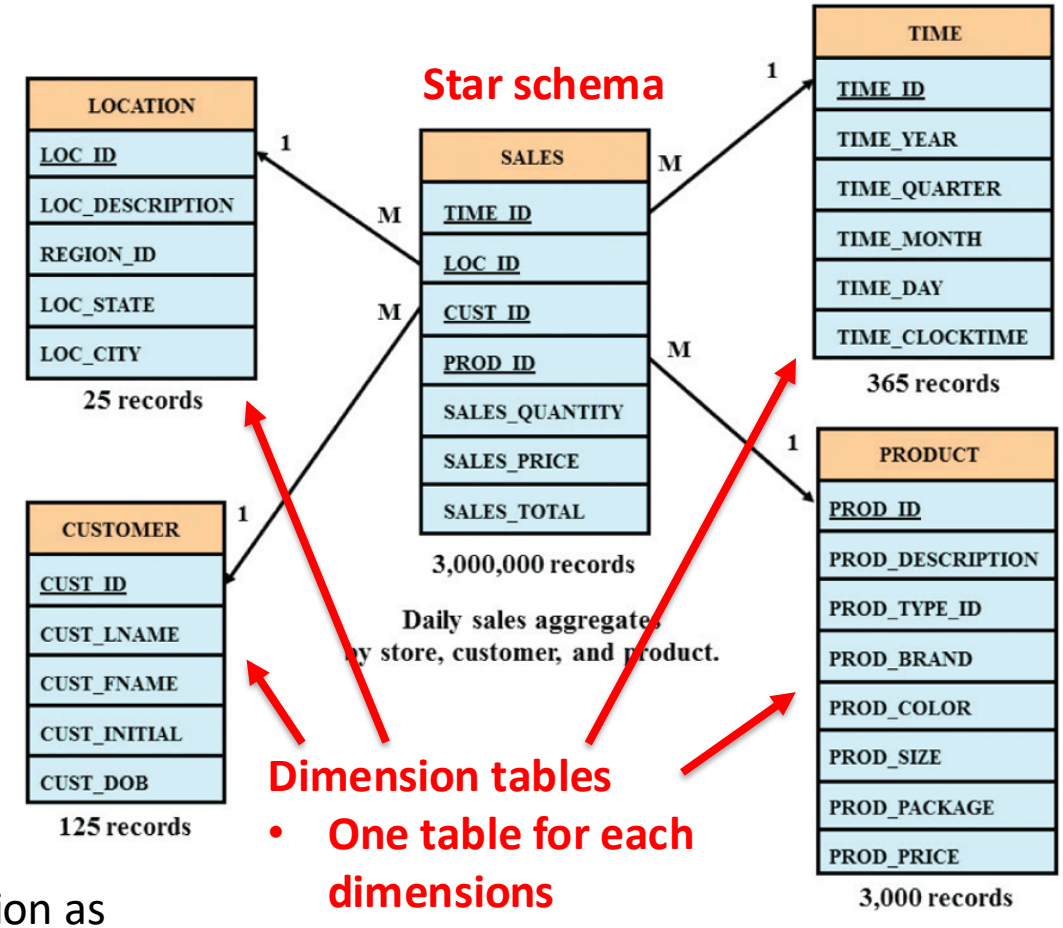
Adapted from Coronel and Morris

Data warehouses are often implemented using a Star Schema

Sales



Star schema



Data cube

- Create conceptual cube with dimension as sides of cube
- Each cube element contains a fact (sales $)
- Allows rapid slicing and dicing
- Uses fact and dimension tables to store data

Dimension tables

- One table for each dimensions
- Keys form PK on fact table
- Each table normalized with attributes for dimension

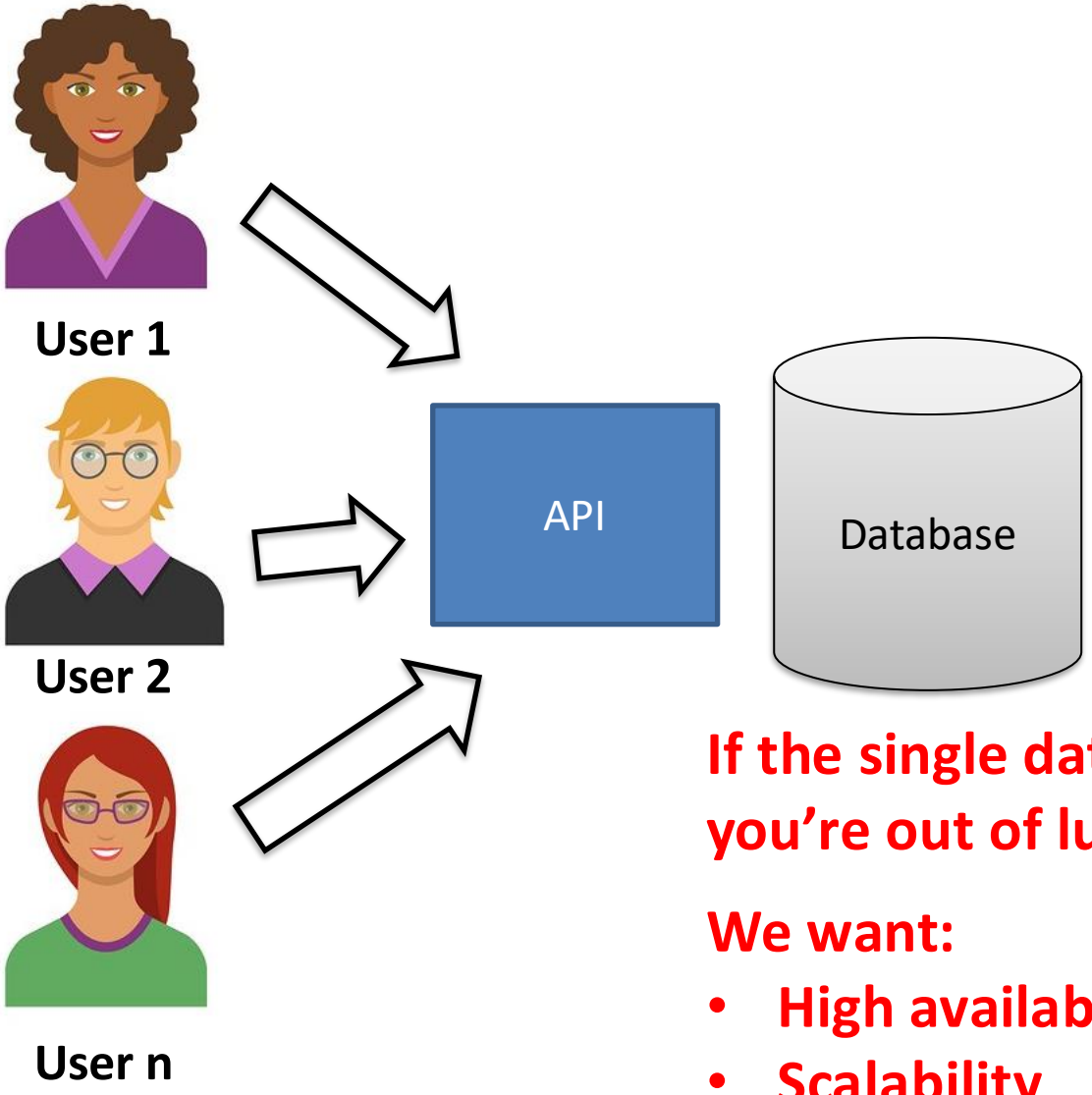
Alternative is one *large* table

Adapted from Coronel and Morris

Agenda

1. Data warehouses
2. Distributed systems
3. MongoDB overview
4. MongoDB queries
5. Restaurant example
6. Course wrap up

With one database, you've put all you eggs in one basket!

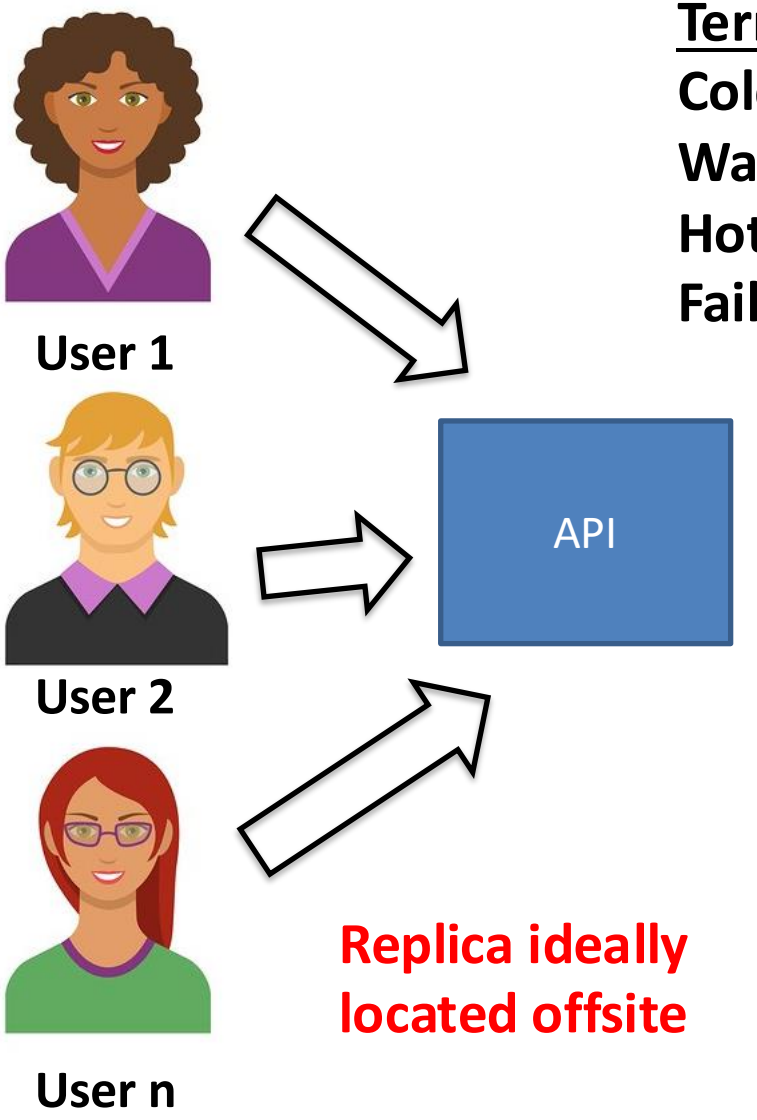


If the single database fails, you're out of luck

We want:

- **High availability**
- **Scalability**

With SANs you can have real-time, block-level replication to another database



Terms

Cold standby: replica current to a point in time

Warm standby: replica kept current

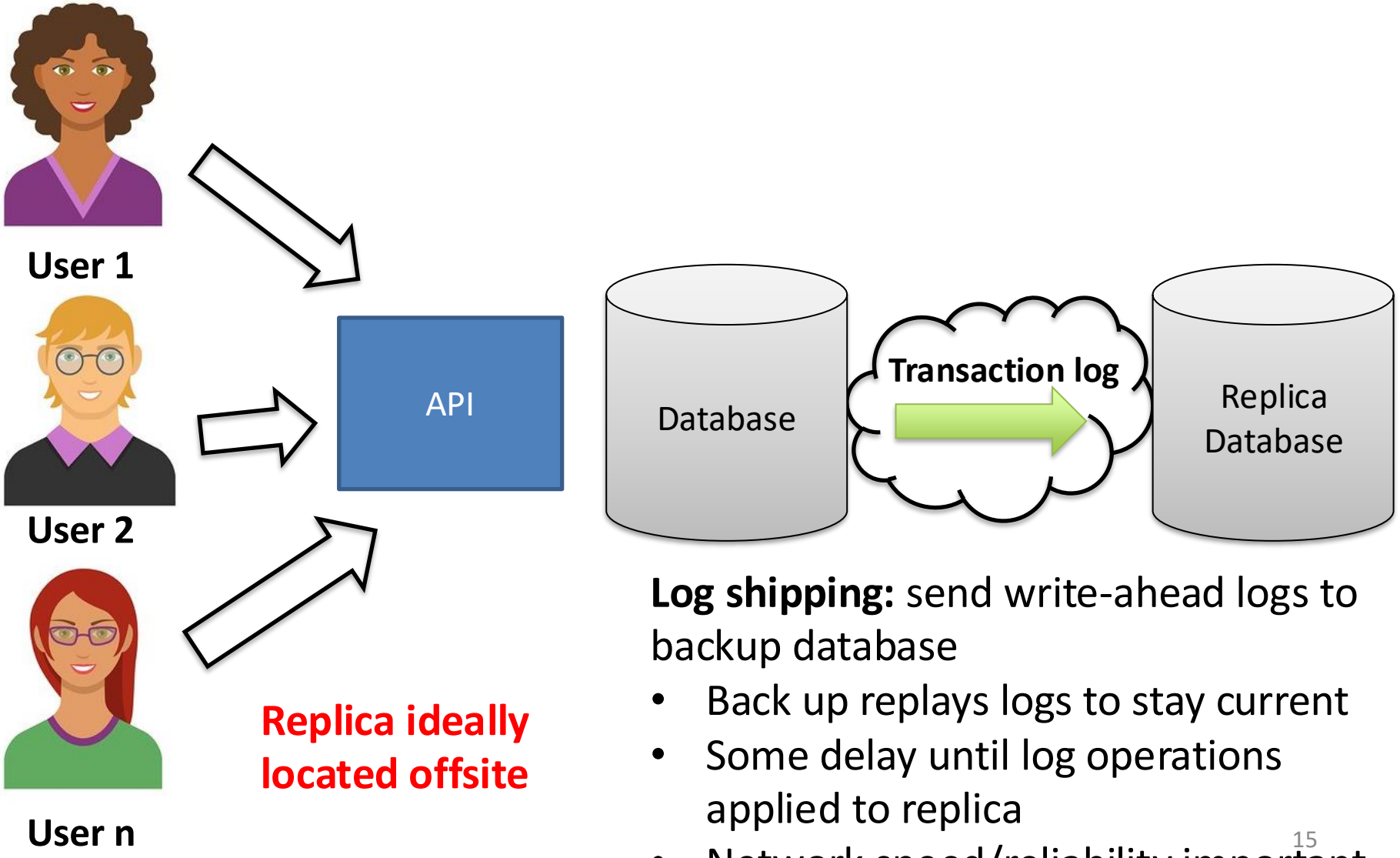
Hot standby: replica is open for read-only ops

Failover: replica takes over for primary

SAN have block-level access

- Change made to one SAN immediately replicated to another SAN
- API accesses replica database if primary fails
- Expensive, but real-time

Log shipping is another, often more cost-effective high availability solution



Log shipping: send write-ahead logs to backup database

- Back up replays logs to stay current
- Some delay until log operations applied to replica
- Network speed/reliability important

Partitioning can help with scalability when data becomes large

Partitioning

Horizontal partitioning (aka sharding) slices data by rows and spreads across multiple nodes

Vertical partitioning slices data by columns

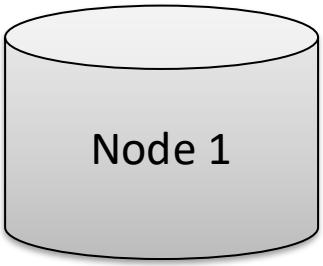
Logical data view

ID	Name	Salary
100	Alice	100,000
200	Bob	90,000
300	Charlie	85,000
…		

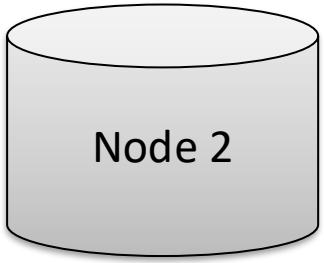
Partitioning increases capacity

- Each machine may be small, but only handles a subset of overall data
- Tradeoff: increased complexity

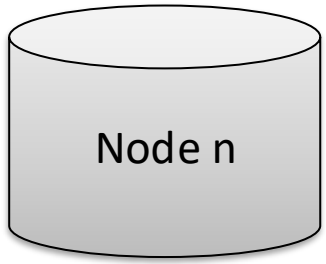
Global distributed schema keeps track of data locations



ID	Name	Salary
100	Alice	100,000
…		



ID	Name	Salary
200	Bob	90,000
…		



ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Sharding horizontally partitions data based on an attribute chosen as the shard key

Partitioning

Choose an attribute to serve as the shard key

- Shard key difficult change once database is partitioned
- Want cardinality > number of shards

Logical data view

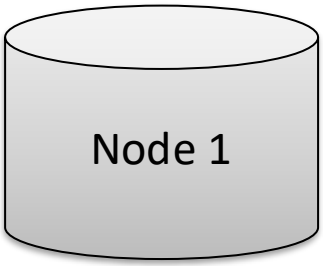
ID	Name	Salary
100	Alice	100,000
200	Bob	90,000
300	Charlie	85,000
…		

Hashing:

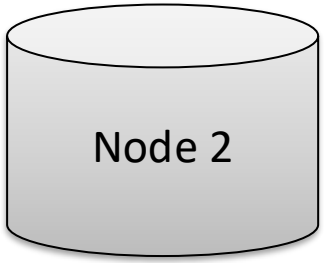
- Hash shard key to get replica number like CS10 hash table

Range partitioning:

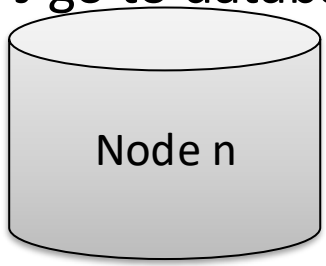
- Distribute based on a partition key (Names A-E go to database 1, F-J go to database 2, …



ID	Name	Salary
100	Alice	100,000
…		



ID	Name	Salary
200	Bob	90,000
…		



ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Choosing a shard key is the most important decision in a sharded cluster

Once chosen, the shard key is hard to change. Choose carefully!

BAD CHOICES

Monotonic IDs

All new writes hit the last shard
Last shard becomes a hotspot

Low cardinality

Few distinct values
Cannot spread across many shards

Skewed distribution

A "country" key where 80% of users are in one country

GOOD CHOICES

High cardinality

Many distinct values to spread across shards

Even distribution

Writes and reads land roughly evenly

Matches common queries

Most queries can target one shard, not scatter-gather

Claude Opus

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

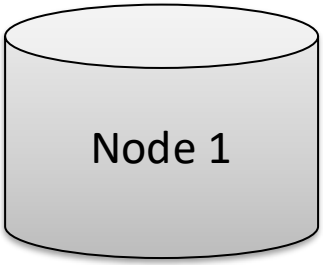
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

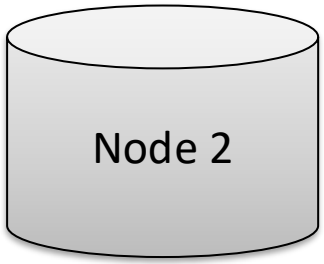
Two-phase commit: DO-UNDO-REDO protocol

- **DO:** Record before and after values in write ahead transaction log
- **UNDO:** Reverses operation using transaction log
- **REDO:** redoes an operation written by DO

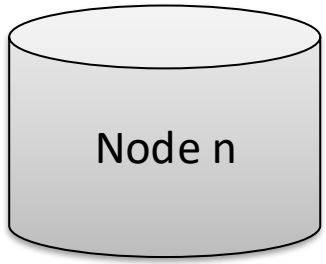
One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		



ID	Name	Salary
200	Bob	90,000
…		



ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

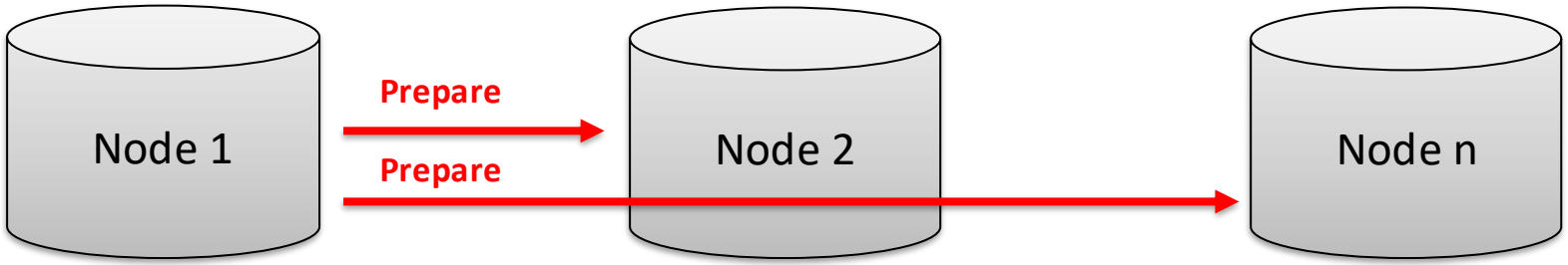
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 1: Preparation

- Coordinator send Prepare to Commit message to all subordinate nodes
- Subordinates write transaction log and send acknowledgement to coordinator
- Coordinator ensures all nodes ready to commit or aborts

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

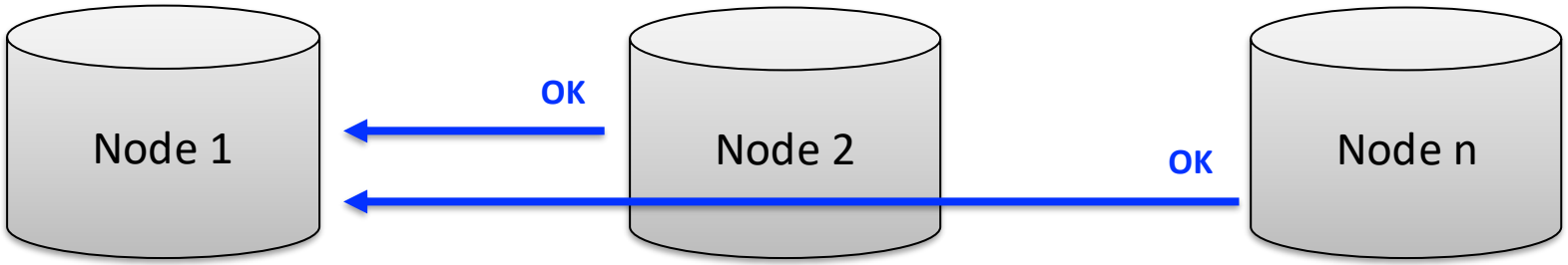
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 1: Preparation

- Coordinator send Prepare to Commit message to all subordinate nodes
- Subordinates write transaction log and send acknowledgement to coordinator
- Coordinator ensures all nodes ready to commit or aborts

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

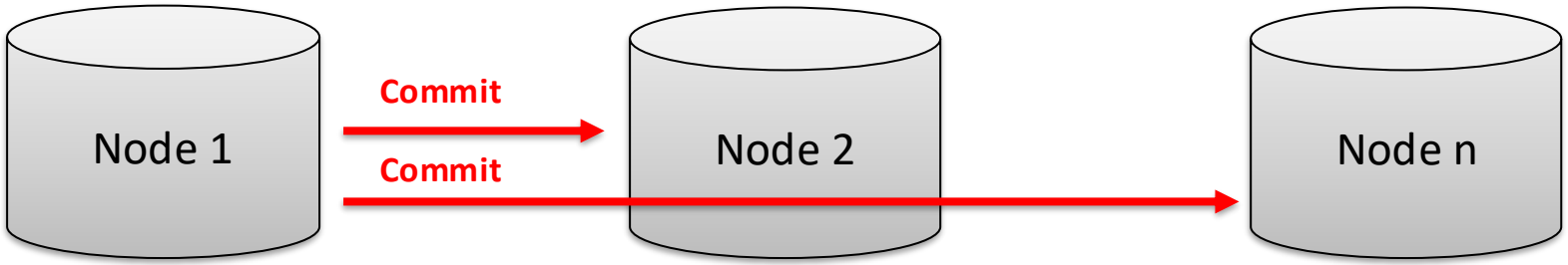
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 2: Commit

- Coordinator broadcasts commit message
- Each subordinate updates with DO
- Subordinates reply with COMMITTED or NOT COMMITTED
- If any nodes reply NOT COMMITTED, then UNDO followed by REDO

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

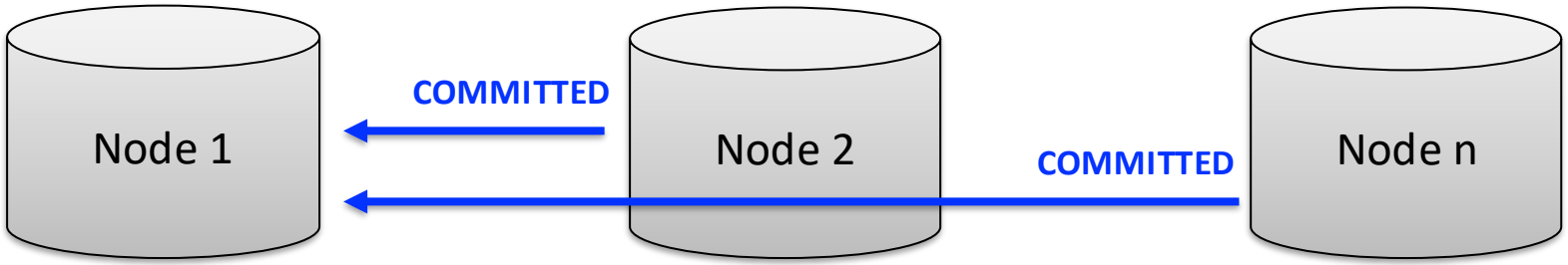
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 2: Commit

- Coordinator broadcasts commit message
- Each subordinate updates with DO
- Subordinates reply with COMMITTED or NOT COMMITTED
- If any nodes reply NOT COMMITTED, then UNDO followed by REDO

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

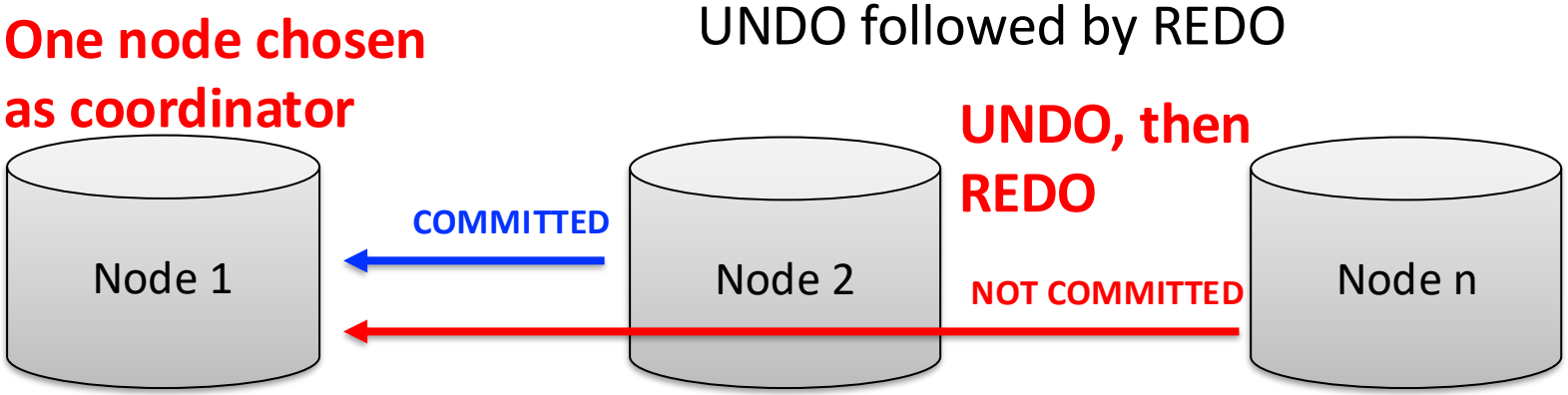
Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 2: Commit

- Coordinator broadcasts commit message
- Each subordinate updates with DO
- Subordinates reply with COMMITTED or NOT COMMITTED
- If any nodes reply NOT COMMITTED, then UNDO followed by REDO



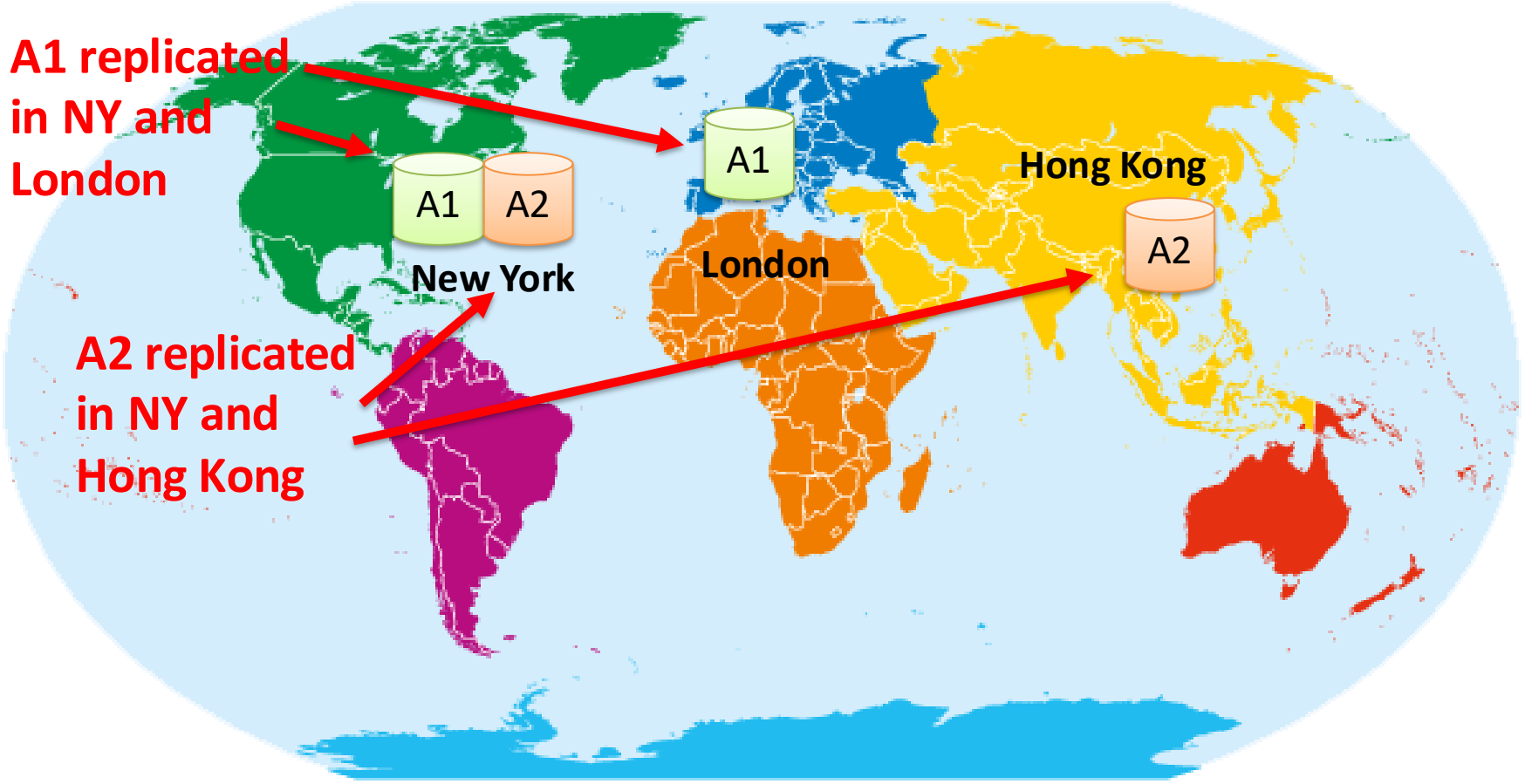
ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

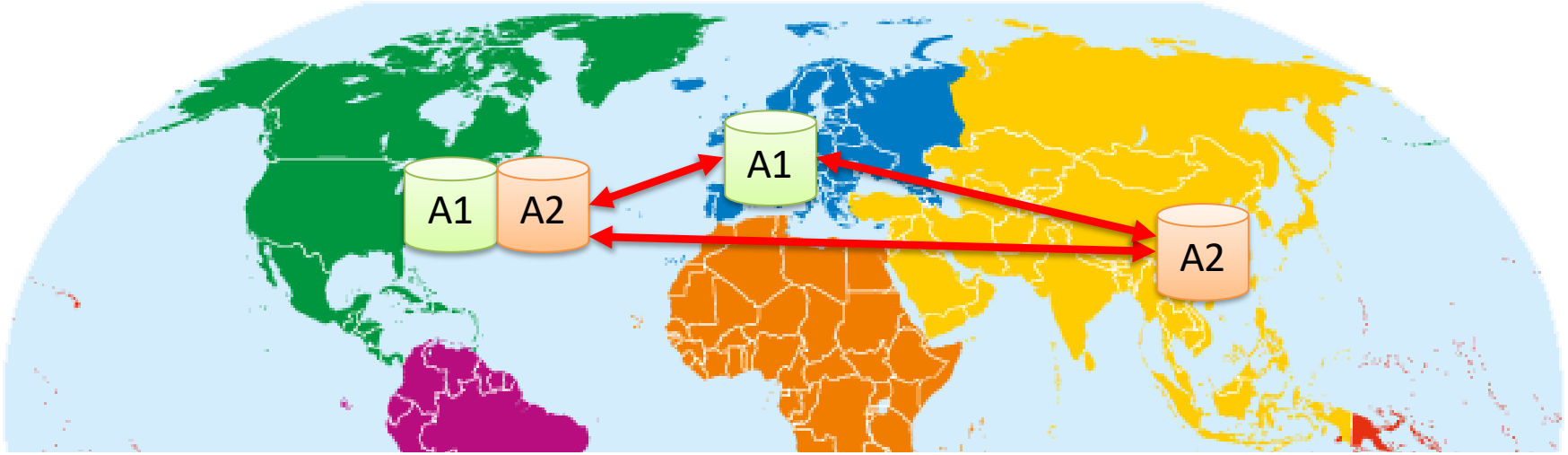
Data might be replicated to several nodes located across the globe



Data replication scenarios

- **Fully replicated:** multiple copies of each database partition at multiple sites
- **Partially replicated**: multiple copies of some database partitions at multiple sites
- **Unreplicated:** stores each database partition at a single site

All replicated nodes should have same data, but network latency raises issues

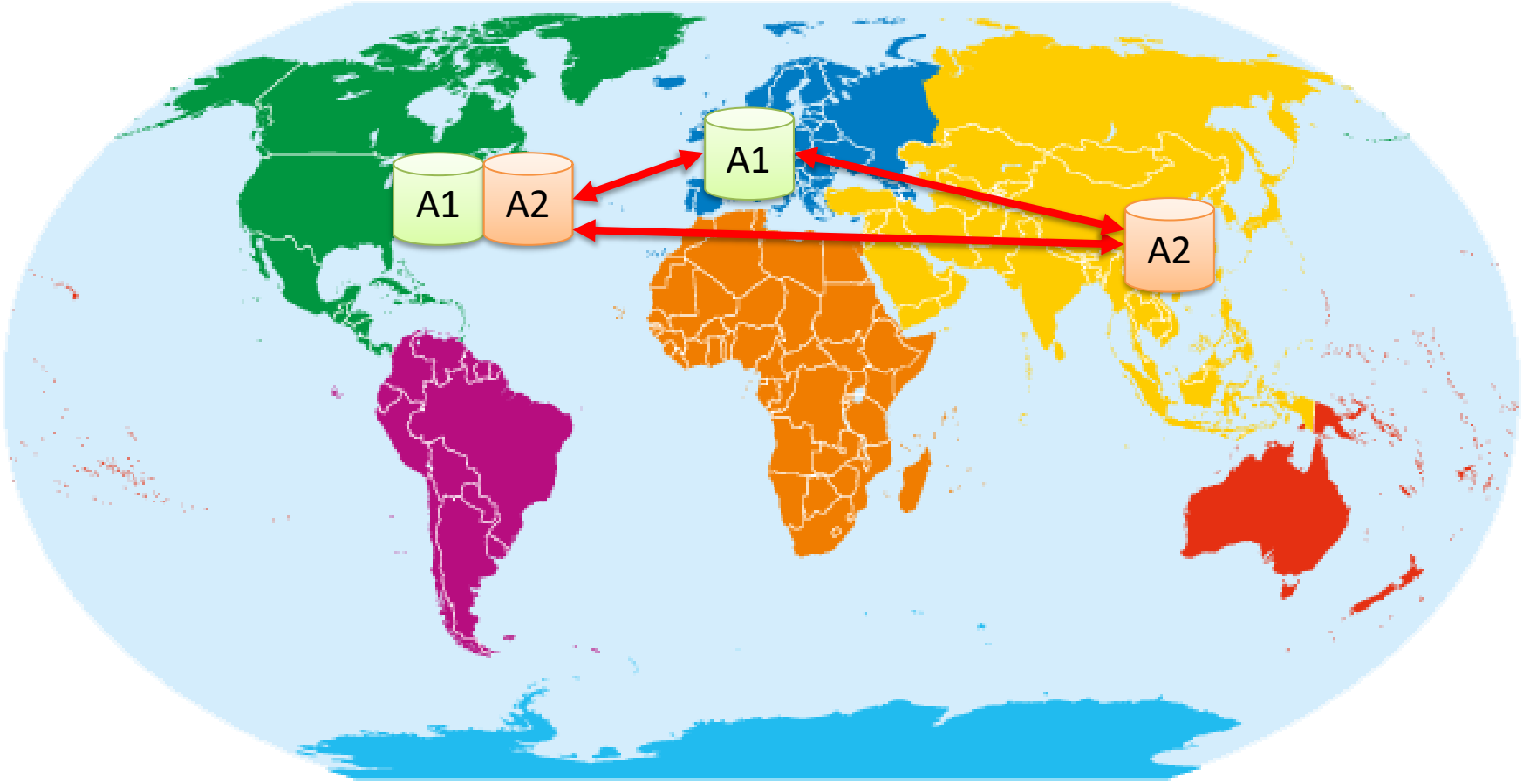


A potential problem: consistency rule says all copies must be identical when data changes are made

Read operations just query the nearest replica

- **Push replication** (focus on consistency)
  - After data update, send changes to all replicas
  - Data unavailable until changes to propagate across all copies, but data is always consistent across copies
- **Pull replication** (focus on availability)
  - Send message to all replicas, they decide when to apply change
  - Data is available, but not consistent until changes propagate

The network may be partitioned by communication breaks

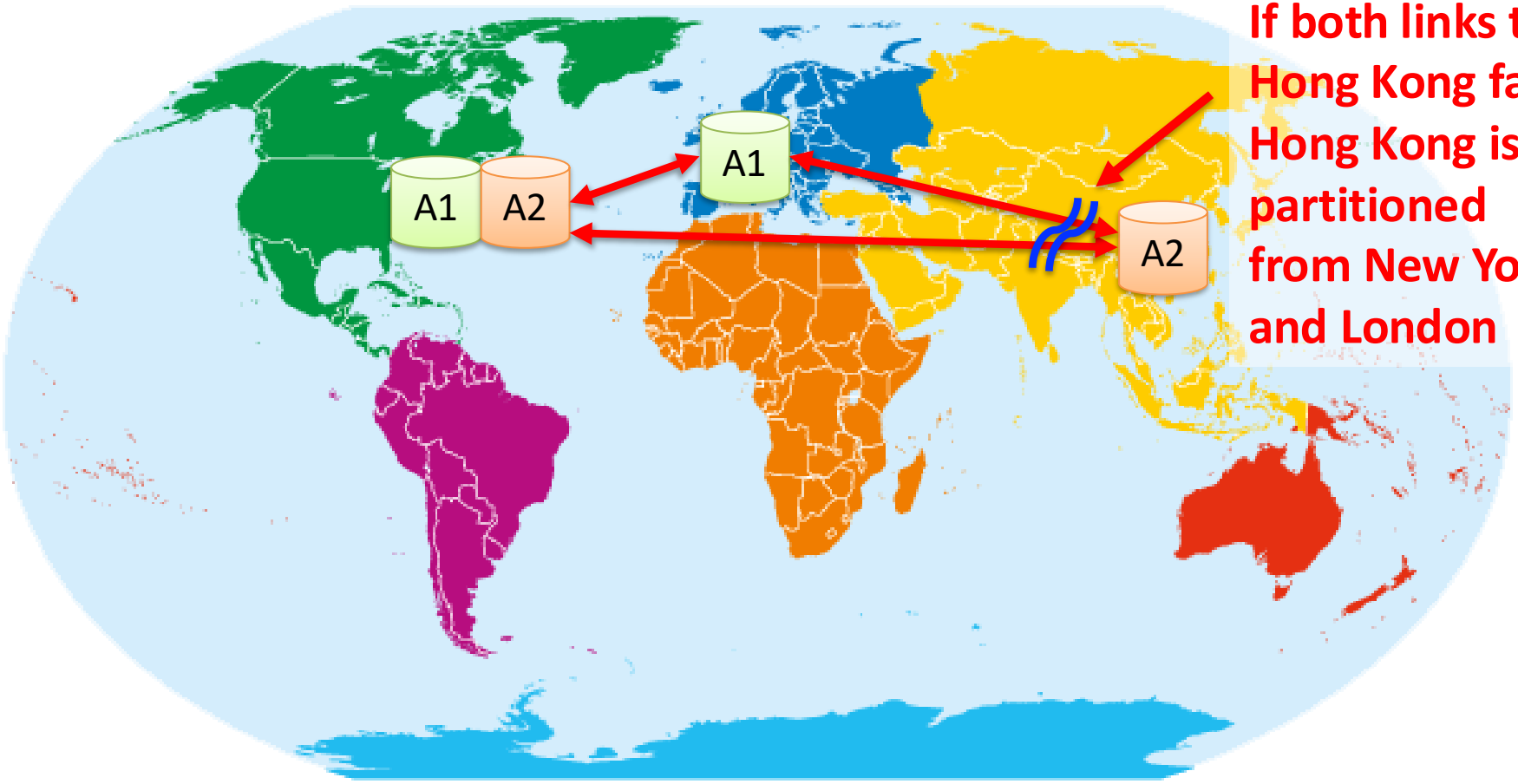


The network may have multiple communication links between each node

If one link fails, other nodes will still be reachable

Multiple link failure, however, may separate some nodes – called a network partition

The network may be partitioned by communication breaks



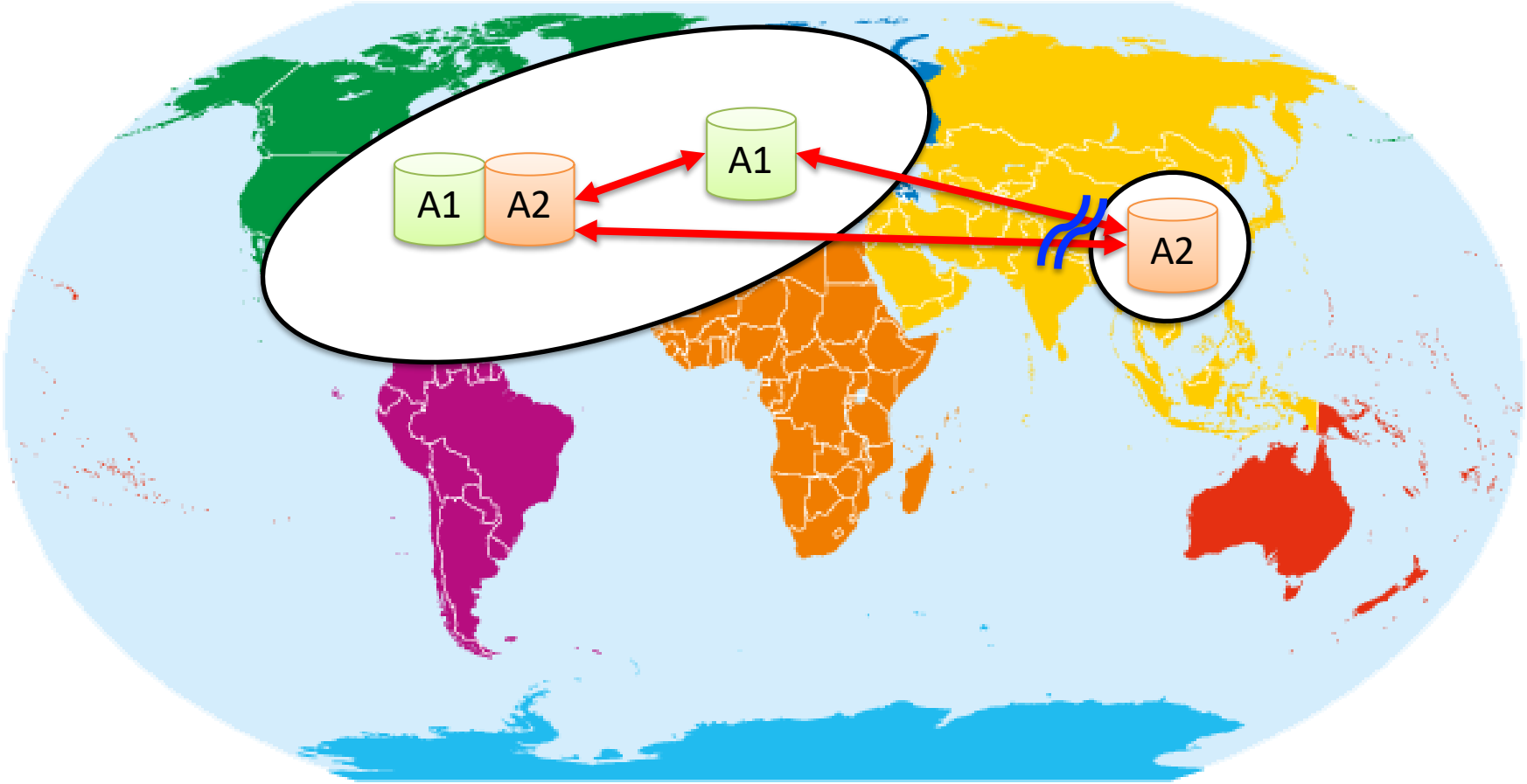
If both links to Hong Kong fail, Hong Kong is partitioned from New York and London

The network may have multiple communication links between each node

If one link fails, other nodes will still be reachable

Multiple link failure, however, may separate some nodes – called a network partition

The network may be partitioned by communication breaks



The network may have multiple communication links between each node

If one link fails, other nodes will still be reachable

Multiple link failure, however, may separate some nodes – called a network partition

It impossible to be Consistent, Available, and Partition tolerant simultaneously

CAP theorem

The CAP theorem showed that it is impossible to have three desirable properties at the same time in distributed systems

Consistency

- All nodes should return the same data at the same time
- Replicas should be immediately updated
- Network latency means this cannot happen

Availability

- A request is always fulfilled by the system
- No request is ever lost

Partition tolerant

- The system will operate even if nodes fail
- Operations that are lost due to node failure are picked up by other nodes
- The system will only fail if all nodes fail

Trade-off between consistency and availability

BASE rather than ACID: Basically Available, Soft state, Eventually consistent (BASE)

Data changes are not immediate but propagate slowly through the system until all replicas are eventually consistent

Eric A. Brewer, "Towards robust distributed systems," *Principles of Distributed Computing*, ACM, July 2000.

Real distributed systems make different tradeoffs between available and consistent

System	Choice	Why
Bank ledger	**Consistent/Partition tolerant (CP)**	Refuse the write rather than let balances diverge
Shopping cart	**Available/Partition tolerant (AP)**	Merging two carts is fine; losing one is worse
DNS	**Available/Partition tolerant (AP)**	Stale-but-available wins; propagation delay is the design
Distributed lock	**Consistent/Partition tolerant (CP)**	Two clients holding the same lock is a correctness disaster
Social feed	**Available/Partition tolerant (AP)**	Showing replies before parent comments looks broken

Claude Opus

Agenda

1. Data warehouses
2. Distributed systems
3. MongoDB overview
4. MongoDB queries
5. Restaurant example
6. Course wrap up

Relational databases have historically been the “safe bet” for the enterprise

Nobody ever got fired for buying

*

-- Tech industry saying

* Or promoted either…
-- Pierson

SQL databases are a solid choice for many database scenarios

SQL databases scale vertically well (get a bigger box), historically did not scale horizontally well (get lots of boxes)

NoSQL (“Not Only SQL”) databases are generally designed for scaling horizontally (sharding)

NoSQL shines with unstructured data

Tech industry saying: https://www.ibm.com/ibm/history/ibm100/us/en/icons/personalcomputer/words/

But when you do outgrow one server, the relational model fights back

Where it gets hard at scale

Schema rigidity

- Every new field is a migration; ALTER TABLE on a billion rows is a project, not a task

Object–relational mismatch

- Your code has nested objects; your database has flat rows; someone has to translate

Horizontal scaling is painful

- Sharding a relational DB while preserving joins and ACID is genuinely hard

Joins get expensive

- Five-table joins across billions of rows do not always have a fast plan

Semi-structured data

- JSON from APIs, sensor data, event logs — variable shapes do not fit rigid columns

None of this means relational is bad

It means the relational model is a tool, but not the only tool

Claude Opus

MongoDB is a NoSQL database designed for high availability and scalability

MongoDB is a document database designed for scalability and flexibility

- MongoDB is “schemaless”
  - Stores data in JSON-like documents
  - Fields can vary from document to document
  - Data structure can change over time
- MongoDB is a distributed database at its core
  - High availability (replication)
  - Horizontal scaling (sharding)
  - Geographic distribution built in
- MongoDB is free to use
  - Cloud-based version at MongoDB Atlas (https://www.mongodb.com/cloud/atlas)

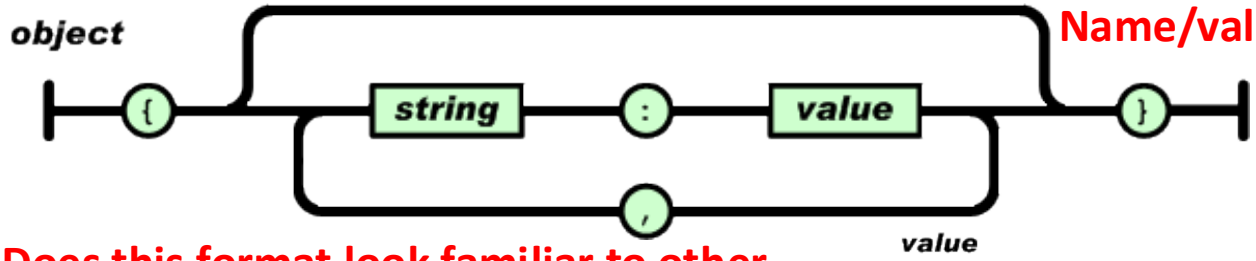
Adapted from Mongo documentation

MongoDB uses a variant of JSON to store documents; simple but powerful!

JSON (JavaScript Object Notation) has two high-level structures

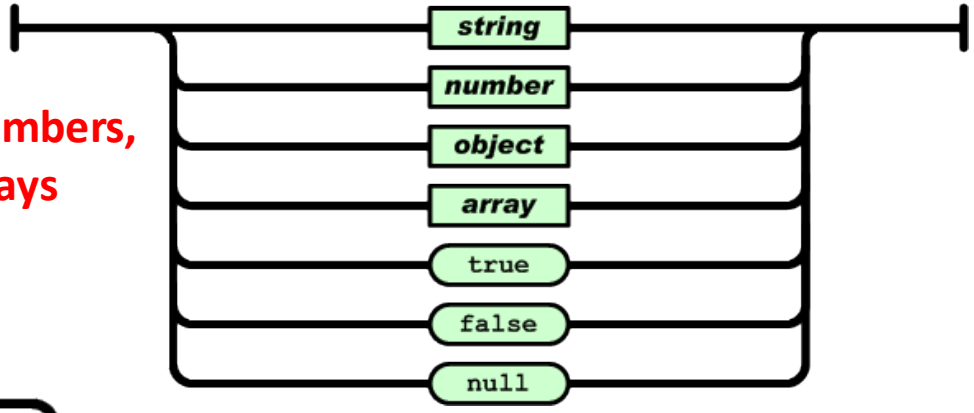
1. **Objects: collection of name/value pairs**

Objects are unordered name/value pairs
Begin with { and end with }
Name/value pairs separated by commas

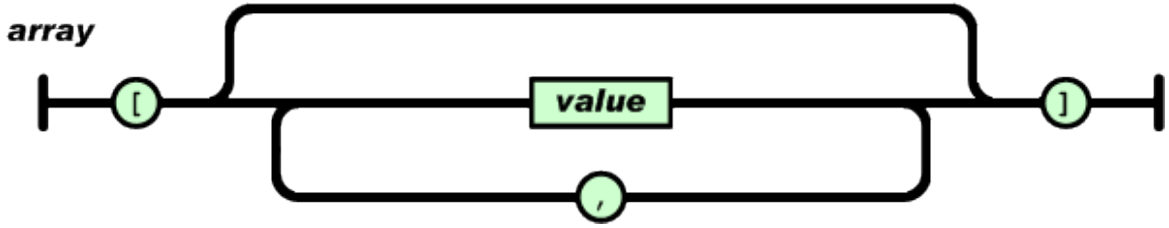


Does this format look familiar to other structures we've seen in CS10?
Finite Automata

- Values can be strings, numbers, booleans, objects, or arrays
- Very powerful "nesting"



2. **Arrays: ordered list of values**



Arrays are ordered
Begin with [and end with]
Items separated by commas

Source: www.json.org

Document databases store JSON-like objects instead of rows

Hierarchy

Database
Logical container (like a SQL schema)

Collection
Group of related documents (~ a table)

Document
The unit you read and write (a JSON-like object)

Field
Key/value pair inside a document; can nest

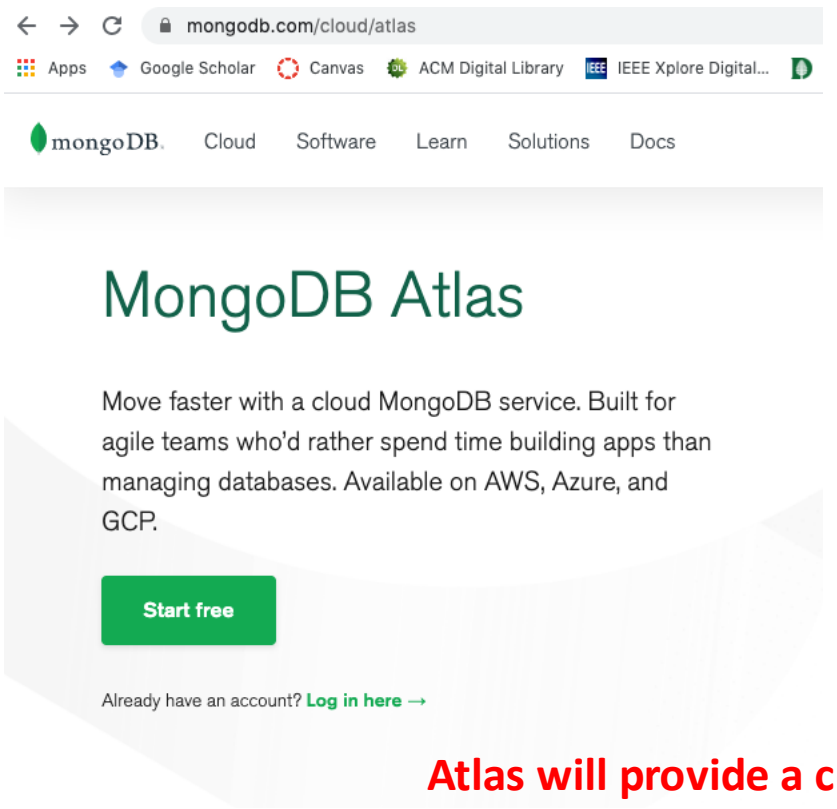
A document

```
{
  _id: ObjectId("65a1..."),
  user: "alex",
  joined: ISODate("2024-03-12"),
  prefs: {
    theme: "dark",
    locale: "en-US"
  },
  tags: ["beta", "annual"],
  loginCount: 47
}
```

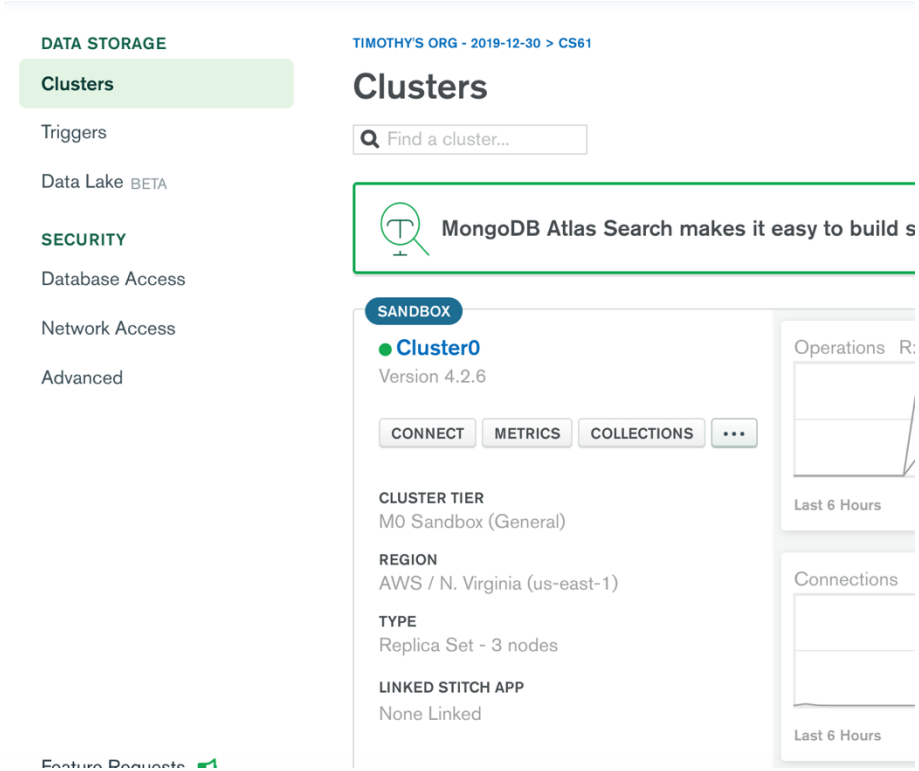
Fields can vary from document to document — no schema enforced by default
Tradeoff: flexibility now, but the database will not catch bad data for you

You can get free access to a Mongo installation in the cloud using Atlas

Create free account at
https://www.mongodb.com/cloud/atlas



Compass is GUI like MySQL Workbench



Atlas will provide a connection string
Can install Compass or command line shell to issue commands

MongoDB documents are a form of JSON and allow document embedding

Blog-style data structure

```
{
      _id: ObjectId(7df78ad8902c)
      title: 'MongoDB Overview',
      by: 'tutorials point',
      tags: ['mongodb', 'database', 'NoSQL'],
      comments: [
            {
                  user:'user1',
                  message: This is user1 comment',
                  dateCreated: new Date(2011,1,20,2,15)
            },
            {
                  user:'user2',
                  message: This is user2  comment',
                  dateCreated: new Date(2011,1,25,7,45)
            }
      ]
}
```

Fields are name/value pairs

You can provide your own _id

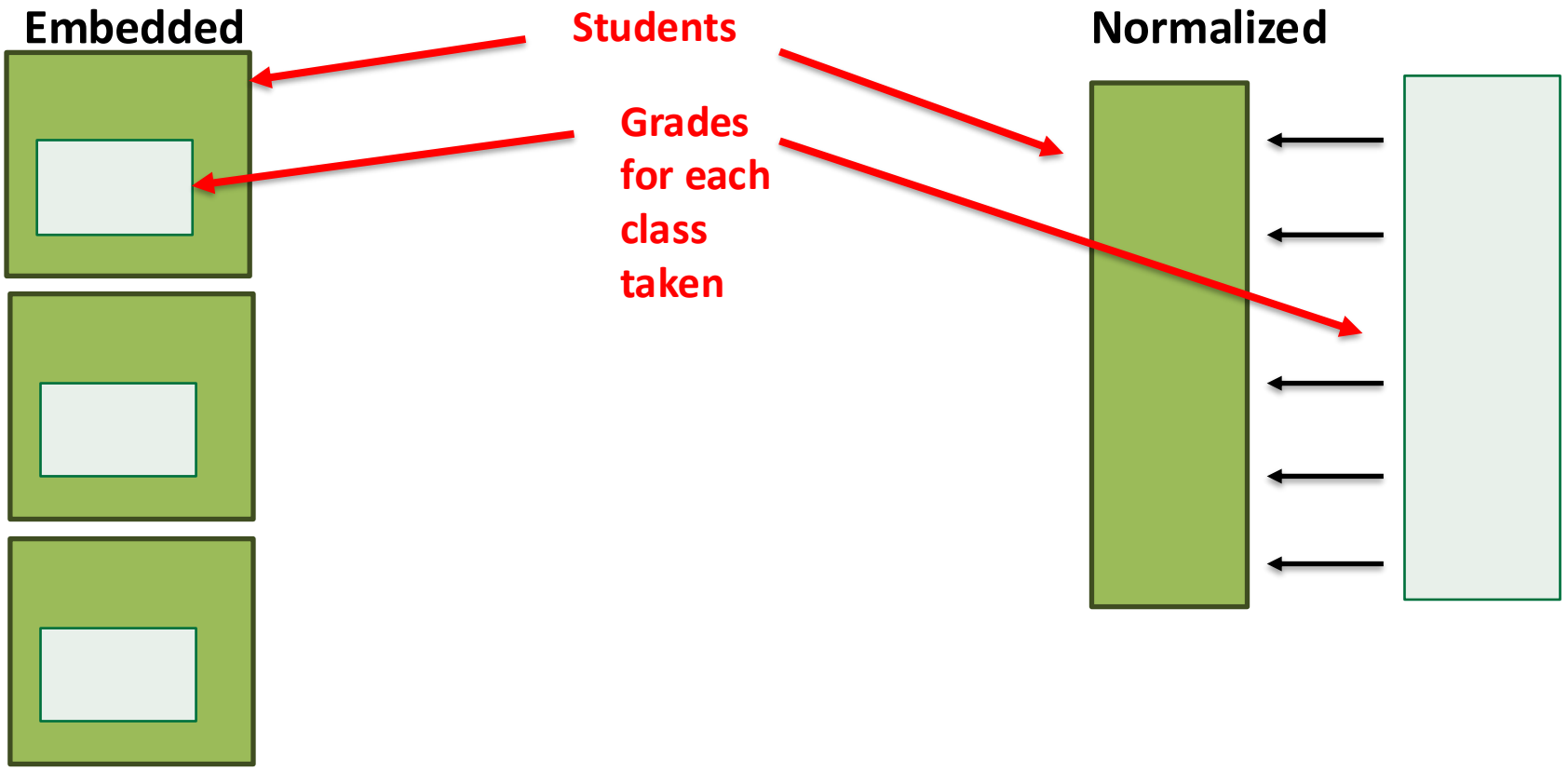
If you do not provide one, MongoDB will generate a unique 12-byte value

Here *comments* is a JSON array of objects embedded inside of a document

Source: https://www.tutorialspoint.com/mongodb/index.htm

The big schema design question is whether to embed documents or normalize

Embedded vs normalized



Each document in a collection has embedded documents

Each entity stands alone
Query second collection for details on primary collection

Embedded data model moves all fields into one document

```
> db.Students.find().pretty()
{
        "_id" : ObjectId("ABC"),
        "name" : "Alice",
        "year" : 27,
        "GPA" : 3.5,
        "grades" : [
                {
                "class" : "CS1",
                "grade" : "A"
                },
                {
                "class" : "CS10",
                "grade" : "A-"
                }
        ]
}
```

Also known as a denormalized data model

Allows applications to store related pieces of information in the same database record

Improves read performance

Result is fewer database queries and updates (no joins needed)

Writes to documents are atomic

Use when:

- Have a “contains” or “has” relationship between entities
- 1:M relationship when M ≲ 1000 and when many side will always appear with one side (not stand alone)
- Document must be < 16 MB in size

Normalized data model references other documents, like a relational database

Normalized data model

Students collection

> db.Students1.find().pretty()

```
        "_id" : ObjectId("ABC"),
        "name" : "Alice",
        "year" : 27
}
```

Like normalized tables in RDBMS

Grades collection

> db.Grades.find().pretty()

```
{
        "_id" : ObjectId("123"),
        "student_id" : ObjectId("ABC"),
        "class" : "CS1",
        "grade" : "A"
}
```

```
{
        "_id" : ObjectId("124"),
        "student_id" : ObjectId("ABC"),
        "class" : "CS10",
        "grade" : "A-"
}
```

Referential integrity is not enforced

Use when:

- Embedding would result in duplication of data, but would not improve read performance enough to outweigh duplication
- To represent complex M:N relationships
- To model large hierarchical datasets

Writes are not atomic across collections, but MongoDB has transactions

Source: https://docs.mongodb.com/manual/core/data-model-design/

1:1 relationships often suggest using embedded documents

1:1 relationships

Normalized

```
// patron collection
{
  _id: "joe",
  name: "Joe Bookreader"
}
```

```
// address collection
{
  patron_id: "joe", //patron
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
```

Embedded

```
{
  _id: "joe",
  name: "Joe Bookreader",
  address: {
        street: "123 Fake Street",
        city: "Faketon",
        state: "NH",
        zip: "12345"
      }
}
```

- **Embed address into patron document**
- **Now one database read gets both patron and address info vs. two reads for normalized approach**
- **Embedding is the preferred approach**

Source: https://docs.mongodb.com/manual/core/data-model-design/

1:1 relationship counter-example is the subset problem, use normalized approach

1:1 relationship subset problem

```
{
  "_id": 1,
  "title": "The Arrival of a Train",
  "year": 1896,
"plot": "A train is seen pulling into a station"
"fullplot": "A group of people are standing in a straight line along…
"type": "movie",
  "directors": [ "Auguste Lumière", "Louis Lumière" ],
  "imdb": {
    "rating": 7.3, "votes": 5043, "id": 12
  },
  "countries": [ "France" ],
  "genres": [ "Documentary", "Short" ],
  "tomatoes": {
    "viewer": {
      "rating": 3.7, "numReviews": 59
    }
}
```

Easily store multiple values in array

Would require multiple tables and JOINs in RDBMS

If you normally only need summary data about a movie, then having plot and fullplot means more disk block reads

Create separate collection for movie details

Leave summary fields in main collection

Only read details when needed

Source: https://docs.mongodb.com/manual/core/data-model-design/

1:M relationships: embed documents if number of embedded document is small

1:M embedded relationships

Normalized

// patron collection

```
{  _id: "joe",
   name: "Joe Bookreader” }
```

// address collection

```
{  patron_id: "joe", //patron
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345” }

{  patron_id: "joe", //patron
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: "12345 }
```

Embedded

Max document size is 16MB

```
{   "_id": "joe",
    "name": "Joe Bookreader",
    "addresses": [
            { "street": "123 Fake Street",
              "city": "Faketon",
              "state": "MA",
              "zip": "12345” },

            { "street": "1 Some Other Street",
              "city": "Boston",
              "state": "MA",
              "zip": "12345” }
      ]
}
```

- **All addresses read in with one read of document**
- **No need for a JOIN operation to get addresses**
- **Subset problem applies here too**
- **Use if address does not need to stand alone**

Source: https://docs.mongodb.com/manual/core/data-model-design/

1:M relationships: use normalized references to avoid duplication

1:M normalized relationships

//books collection

```
{  title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf"],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher: { name: "O'Reilly Media",
                founded: 1980, location: "CA" }
}

{ title: "50 Tips and Tricks for MongoDB ",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher: { name: "O'Reilly Media",
                founded: 1980, location: "CA" }
}
```

Source: https://docs.mongodb.com/manual/core/data-model-design/

//publisher collection

```
{  _id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA" }
```

//books collection

```
{ _id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf"],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher_id: "oreilly" }
{  _id: 234567890,
   title: "50 Tips and Tricks for MongoDB ",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher_id: "oreilly"}
```

M:N relationships can be easily implemented with two-way referencing

M:N

```
db.person.findOne()
{ _id: ObjectId("ABC"), name: "Alice",
tasks [ // Alice is assigned three tasks
        ObjectId("123"), //write lesson plan below
        ObjectId("124"), //another task
        ObjectId("125") //Alice's third task
        ]
}

db.tasks.findOne()
{ _id: ObjectID("123"), description: "Write lesson plan", due_date: ISODate("2014-04-01"),
assigned: [ObjectId("ABC") // Reference to Alice
            ObjectId("DEF") //Reference to another person assigned to this task
            ]
}
```

Two collections
One *person* is assigned many tasks
One *task* is assigned to many people

Create array of references
- **Person to task**
- **Task to person**

Advantage:
- Easy to find who is assigned to tasks, and which tasks a person is assigned

Disadvantage:
- If person added to, or removed from task, must update two tables

Adapted from https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1

Sometimes it is useful to denormalize

M:N

```
db.person.findOne()
{ _id: ObjectId("ABC"), name: "Alice",
tasks [ // Alice is assigned three tasks
        ObjectId("123"), //write lesson plan below
        ObjectId("124"), //another task
        ObjectId("125") //still another task
        ]
}

db.tasks.findOne()
{ _id: ObjectID("123"), description: "Write lesson plan", due_date: ISODate("2014-04-01"),
assigned: [{person _id: ObjectId("ABC"), name: "Alice"}, // now have Alice's name
            {person_id: ObjectId("DEF"), name: "Bob"} //also have Bob's name
           ]
}
```

Two collections
One *person* is assigned many tasks
One *task* is assigned to many people

Denormalize to include person's name in tasks collection of assigned people

Now do not need to look up the names of people assigned to tasks

Use denormaliztion if many more reads than writes

Do not denormalize something that changes frequently!

Advantage:

- No need to lookup people's name when finding tasks

Disadvantage:

- If Alice's name changes, must update person collection and *all* entries in task collection

Adapted from https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1

Embedding vs. normalization rules of thumb

Advice from William Zola, MongoDB Lead Technical Support Engineer

1. Favor embedding unless there is a compelling reason not to
2. The need to access an object on its own is a compelling reason not to embed
3. Arrays should not grow unbounded:
   - If there are more than a couple hundred documents on the many side, don't embed them
   - If there are more than a few thousand on the many side, don't use an array of ObjectId references
   - High-cardinality arrays are a compelling reason not to embed
4. Don't be afraid of application-level joins: if you index correctly and use the projection specifier, then application-level joins are barely more expensive than server-side joins in a relational database
5. Consider the read/write ratio when denormalizing: a field that is mostly read and only seldomly updated is a good candidate for denormalization
6. Your design should depend on how your application accesses data!

Adapted from https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-3

When MongoDB fits and when it does not

Good fit

- ✓ **Evolving schemas**
  Rapidly changing product, many optional fields
- ✓ **Document-shaped data**
  Hierarchical by nature
- ✓ **Horizontal scale**
  Built-in sharding; easy to grow
- ✓ **Geographic distribution**
  Replica sets across regions
- ✓ **Working set fits memory**
  Hot data should fit; Mongo loves RAM

Bad fit

- ✗ **Heavy multi-table joins**
  Relational engines do this far better
- ✗ **Strong cross-entity ACID**
  Possible but slow; discouraged on hot paths
- ✗ **Ad-hoc analytics**
  OLAP engines and warehouses outperform here
- ✗ **Strict data integrity**
  Foreign keys, check constraints — not its strength
- ✗ **Small, simple data**
  MySQL on a single box is hard to beat

SQL and NoSQL worlds are converging

2016

"Pick a side: SQL or NoSQL"

2026

Pick the right tool. They borrow freely

Evidence of the convergence

Postgres — First-class JSONB, GIN indexes, document-style queries.

MongoDB — Multi-document ACID transactions, joins via $lookup, schema validation.

Cloud DBs — Spanner, CockroachDB, FaunaDB: SQL surface with distributed guarantees.

The interesting battles now are about operations and ecosystems, not the data model

Claude Opus

Key take away points

1. **NoSQL is not anti-SQL**
   It is a family of design choices for problems the relational model did not ace

2. **There is no free lunch**
   Every distributed system trades consistency, availability, latency, or simplicity

3. **Pick by workload**
   Match data shape, query patterns, and consistency needs — not the hype cycle

4. **MongoDB's edge is documents**
   Flexible schema, hierarchical data, horizontal scale
   Embed or reference deliberately

5. **Boundaries are blurring**
   MySQL and Postgres do JSON
   MongoDB does ACID
   Battles now are about operations and ecosystems

Claude Opus

Agenda

1. Data warehouses
2. Distributed systems
3. MongoDB overview
4. MongoDB queries
5. Restaurant example
6. Course wrap up

MongoDB CRUD on a collection is similar to SQL CRUD on a table

CRUD operations

CRUD	SQL	MongoDB
Create	INSERT	insertOne/insertMany
Read	SELECT	find({name:value})
Update	UPDATE	updateOne/updateMany
Delete	DELETE	deleteOne/deleteMany

One: operate on first document

Many: operate on all matching

The CRUD operations in MongoDB have different names but function similarly on a collection as SQL commands on a table

Embedded and linking documents, however, is different

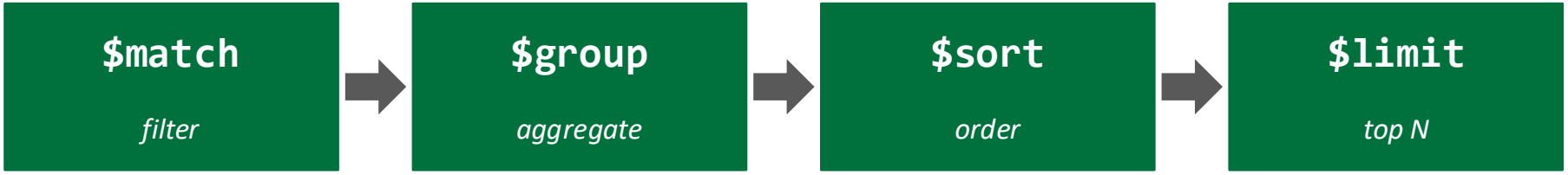
Basic CRUD operations on documents

INSERT	`db.users.insertOne({ name: "Alex", age: 30 })`
FIND	`db.users.find({ age: { $gte: 18 } })`
UPDATE	`db.users.updateOne({ name: "Alex" }, { $inc: { age: 1 } })`
DELETE	`db.users.deleteMany({ active: false })`

Query operators ($gte, $in, $or, $regex…) are prefixed with $
Update operators ($set, $inc, $push…)

The aggregation pipeline runs documents through transformation stages

Documents flow through stages; each stage transforms its input and feeds the next




```
db.orders.aggregate([
  { $match: { status: "shipped" } },
  { $group: { _id: "$customer", total: { $sum: "$amount" } } },
  { $sort:  { total: -1 } },
  { $limit: 10 }
])

// Top 10 customers by shipped order revenue descending
```


Claude Opus

Indexes work the same way as in SQL — same idea, more flavors

MongoDB indexes are B-trees. Same principles, more index types.

Single field

On one key. The classic.

```
db.users.createIndex({ email: 1 })
```

Compound

Multiple keys, in order. Prefix rule applies.

```
db.orders.createIndex({ customer: 1, date: -1 })
```

Multikey

On array fields. One entry per element.

```
db.posts.createIndex({ tags: 1 })
```

Text

Inverted index for full-text search.

```
db.articles.createIndex({ body: "text" })
```

Geospatial

2dsphere indexes for lat/lng queries.

```
db.places.createIndex({ loc: "2dsphere" })
```

Use explain() to inspect the chosen plan — just as you would in SQL

See databases with “show dbs”

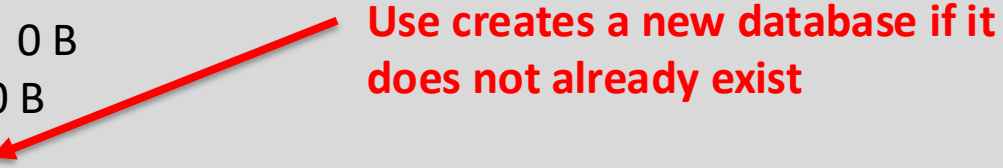
```
> show dbs
sample_mflix  111.86 MiB
admin          0 B
local         0 B
```

List databases

Note: no cs61 database

Make a new collection simply by saying "use"

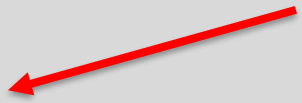
```
> show dbs
sample_mflix  111.86 MiB
admin          0 B
local          0 B
> use cs61
switched to db cs61
```



Use creates a new database if it does not already exist

Make a new collection simply by saying "use"

```
> show dbs
sample_mflix  111.86 MiB
admin          0 B
local          0 B
> use cs61
switched to db cs61
> show dbs
sample_mflix  111.86 MiB
admin          0 B
local          0 B
```



cs61 does not appear yet
It has no data

Add data into the collection with insert command

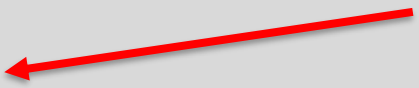
```
> show dbs
sample_mflix  111.86 MiB
admin          0 B
local          0 B
> use cs61
switched to db cs61
> show dbs
sample_mflix  111.86 MiB
admin          0 B
local          0 B
> db.Students.insert({"name": "Alice", "year":27, "GPA": 3.5, "Grades": [] })
{
acknowledged: true,
insertedIds: {
'0': ObjectId('6a1b1737b80721196e26cb61')
}
```

db.<collection name> adds a collection
Creates collection and insert adds data
Note: no schema!
Can add any fields we would like!

We did not provide a unique id, so Mongo created one for us

Once the database contains data, it appears in “show dbs”

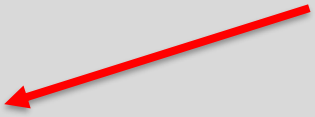
```
> db.Students.insert({"name": "Alice", "year":27, "GPA": 3.5, "Grades": [] })
{
acknowledged: true,
insertedIds: {
'0': ObjectId('6a1b1737b80721196e26cb61')
}
> show dbs
cs61          8.00 KiB
sample_mflix  111.86 MiB
admin          0 B
local          0 B
```



cs61 database now appears with show dbs (It now has data)

List collections in a database with “show collections”

```
> db.Students.insert({"name": "Alice", "year":27, "GPA": 3.5, "Grades": [] })
{
acknowledged: true,
insertedIds: {
'0': ObjectId('6a1b1737b80721196e26cb61')
}
> show dbs
cs61          8.00 KiB
sample_mflix  111.86 MiB
admin          0 B
local          0 B
> show collections
Students
```



Show collections lists the collections in this database (see current database with just “db”)

List data in a collection with “find”

```
> db.Students.find().pretty()
{
  _id: ObjectId('6a1b1737b80721196e26cb61'),
  name: 'Alice',
  year: 27,
  GPA: 3.5,
  Grades: []
}
```

db.<collection name>.find() is like SELECT FROM <table name>

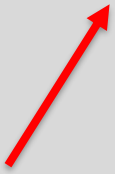
Note: Mongo created a unique id for us

Alice’s GPA is 3.5

Update with updateOne

```
> db.Students.updateOne({name:"Alice"}, {$set:{GPA:3.7}})
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Find Alice



Update Alice's GPA to 3.7

Update with updateOne

```
> db.Students.updateOne({name:"Alice"}, {$set:{GPA:3.7}})
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
> db.Students.find().pretty()
{
_id: ObjectId('6a1b1e4cb80721196e26cb62'),
name: 'Alice',
year: 27,
GPA: 3.7,
Grades: []
}
```

Find Alice

Update Alice's GPA to 3.7

Confirmed, GPA is now 3.7

Update with updateOne

```
> db.Students.updateOne({name:"Alice"}, {$set:{GPA:3.7}})
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
> db.Students.find().pretty()
{
_id: ObjectId('6a1b1e4cb80721196e26cb62'),
name: 'Alice',
year: 27,
GPA: 3.7,
Grades: []
}
```

Find Alice

Update Alice's GPA to 3.7

Confirmed, GPA is now 3.7

Design decision: embed grades (rather than normalize)

Add to embedded array with $push

```
> db.Students.updateOne({name:"Alice"},{$push:{Grades:{class:"CS61",grade:"A"}}})
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Find Alice

Push new Grade object into Alice's array of grades

Add to embedded array with $push

```
> db.Students.updateOne({name:"Alice"},{$push:{Grades:{class:"CS61",grade:"A"}}})
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
> db.Students.find().pretty()
{
 _id: ObjectId('6a1b1e4cb80721196e26cb62'),
 name: 'Alice',
 year: 27,
 GPA: 3.7,
 Grades: [
  {
   class: 'CS61',
   grade: 'A'
  }
 ]
```

Find Alice

Push new Grade object into Alice's array of grades

Embedded array now contains an entry for CS61 with grade A

Push grades for other classes similarly

Add new student Bob

```
> db.Students.insert({"name": "Bob", "year":28, "Grades": [] })
{
acknowledged: true,
insertedIds: {
'0': ObjectId('6a1b2167b80721196e26cb63')
}
}
```

No GPA provided
No need to conform to a schema!
Grades is an empty JSON array

Add new student Bob

```
> db.Students.find().pretty()
{
  _id: ObjectId('6a1b1e4cb80721196e26cb62'),
  name: 'Alice',
  year: 27,
  GPA: 3.7,
  Grades: [
    {
      class: 'CS61',
      grade: 'A'
    }
  ]
}
{
  _id: ObjectId('6a1b2167b80721196e26cb63'),
  name: 'Bob',
  year: 28,
Grades: []
}
```

Students Alice and Bob are in the collection

No GPA or grades yet for Bob

Find all students who got an A in CS61 using “find”

```
> db.Students.find({"Grades.class": "CS61", "Grades.grade":"A"})
{
  _id: ObjectId('6a1b1e4cb80721196e26cb62'),
  name: 'Alice',
  year: 27,
  GPA: 3.7,
  Grades: [
    {
      class: 'CS61',
      grade: 'A'
    }
  ]
}
```

Only Alice (Bob doesn’t have any grades yet)

Find all students who got an A in CS61 using “find” (now just with names)

```
> db.Students.find({"Grades.class": "CS61", "Grades.grade":"A"},{"name":1})
{
  _id: ObjectId('6a1b1e4cb80721196e26cb62'),
  name: 'Alice'
}
```

Only return name (and id)
Same as a project

Find all students who got an A in CS61 using “find” (now just with names)

```
> db.Students.find({"Grades.class": "CS61", "Grades.grade":"A"},{"name":1})
{
  _id: ObjectId('6a1b1e4cb80721196e26cb62'),
  name: 'Alice’
}
> db.Students.find({"Grades.class": "CS61", "Grades.grade":"A"},{"name":1, "_id":0})
{
  name: 'Alice'
}
```

Only return name (and id)
Same as a project

Do not return id
Use 0

Create second collection for courses

```
> db.Courses.insertOne({"name":"CS61", "longName": "CS61: Database systems"})
{
acknowledged: true,
insertedIds: {
'0': ObjectId('6a1da392caed5d0c591ee2b4')
}
}

> db.Courses.find().pretty()
{
  _id: ObjectId('6a1da392caed5d0c591ee2b4'),
  name: 'CS61',
  longName: 'CS61: Database systems'
}
```

Course name

Mongo does not provide constraint checking

```
> db.Courses.insertOne({"name":"CS61", "longName": "CS61: Database systems"})
{
acknowledged: true,
insertedId: ObjectId('6a1da401caed5d0c591ee2b5')
}
```

No constraint checks
Adding the same class again is not prevented
Each entry gets its own _id

Mongo does not provide constraint checking

```
> db.Courses.insertOne({"name":"CS61", "longName": "CS61: Database systems"})
{
acknowledged: true,
insertedId: ObjectId('6a1da401caed5d0c591ee2b5')
}

> db.Courses.find().pretty()
{
  _id: ObjectId('6a1da392caed5d0c591ee2b4'),
  name: 'CS61',
  longName: 'CS61: Database systems'
}
{
  _id: ObjectId('6a1da401caed5d0c591ee2b5'),
  name: 'CS61',
  longName: 'CS61: Database systems'
}
```

No constraint checks
Adding the same class again is not prevented
Each entry gets its own _id

Now we have two entries for CS61!

Each has a unique id

You can delete items by _id

```
> db.Courses.find().pretty()
{
  _id: ObjectId('6a1da392caed5d0c591ee2b4'),
  name: 'CS61',
  longName: 'CS61: Database systems’
}
{
  _id: ObjectId('6a1da401caed5d0c591ee2b5'),
  name: 'CS61',
  longName: 'CS61: Database systems'
}

> db.Courses.deleteOne({_id:ObjectId("6a1da401caed5d0c591ee2b5")})
{
acknowledged: true,
deletedCount: 1
}
```

Delete second entry by _id
Note: use ObjectId(“<id>”)

Lookup items in another table with $lookup

```
> db.Students.aggregate([{$lookup: {from: "Courses",localField: "Grades.class",foreignField: "name",as: "course_details"}}])
{
_id: ObjectId('6a1b1e4cb80721196e26cb62'),
name: 'Alice',
year: 27,
GPA: 3.7,
Grades: [
{
class: 'CS61',
grade: 'A'
}
],
course_details: [
{
_id: ObjectId('6a1da392caed5d0c591ee2b4'),
name: 'CS61',
longName: 'CS61: Database systems'
}
]
```

Get matching data from Courses collection

Agenda

1. Data warehouses
2. Distributed systems
3. MongoDB overview
4. MongoDB queries
5. Restaurant example
6. Course wrap up

First connect to server and select a database to use

Get connection to database

Get connection string from Atlas web site (or use local host)

```
vpn-two-factor-general-230-148-96:~ tim$ mongo "mongodb+srv://cluster0-rq8e6.mongodb.net/test" --username user
MongoDB shell version v4.2.0
Enter password:
connecting to: mongodb://cluster0-shard-00-01-rq8e6.mongodb.net:27017,cluster0-shard-00-02-rq8e6.mongodb.net:27017, <snip>
MongoDB server version: 4.2.6
```

Atlas provides a connection string
Provide password when prompted

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> show dbs
admin  0.000GB
cs61   0.000GB
local  3.875GB
```

***show dbs* lists the databases (schemas) available on this server**

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db
test
```

***db* shows the currently selected database (test is default)**

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> use cs61
switched to db cs61
```

***use dbName* switches to *dbName* database**

Database is created if does not already exist, but will not show up in *show dbs* until at least one document is inserted

dropDatabase deletes a database

db.dropDatabase()

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.dropDatabase()
{
"dropped" : "cs61",
"ok" : 1,
"$clusterTime" : {
"clusterTime" : Timestamp(1589633625, 2),
"signature" : {
"hash" : BinData(0,"GjGGdJxmLCiQnjphPBpPqucfiWE="),
"keyId" : NumberLong("6826973194142875651")
}
},
"operationTime" : Timestamp(1589633625, 2)
}
```

dropDatabase() deletes the currently selected database (here cs61)

Database *cs61* is deleted

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> show dbs
admin  0.000GB
local  3.871GB

MongoDB Enterprise Cluster0-shard-0:PRIMARY> db
cs61
```

Currently select database is still *cs61*, but it is empty

createCollections makes a collection, like CREATE TABLE

db.createCollection(“CollectionName”)

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.createCollection("Restaurants")
{
"ok" : 1,
"$clusterTime" : {
"clusterTime" : Timestamp(1589633855, 1),
"signature" : {
"hash" : BinData(0,"4aXY1b0hyVq4XI7esxFogn5IwEM="),
"keyId" : NumberLong("6826973194142875651")
}
},
"operationTime" : Timestamp(1589633855, 1)
}
```

***createCollections* makes a new collection**
Could also have just said *use Restaurants*

***show collections* lists the collections in the database**

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> show collections
Restaurants
```

Insert documents into a collection with *insert*, list documents with *find*

db.Collection.insert({document)

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.insert({name:"Tim's
Tasty Treats",boro:"Manhattan"})
{
"acknowledged" : true,
"insertedId" : ObjectId("5ebfe42a6ead6c5a3b84c554")
}
```

***find* is like SELECT**

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find()
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan"
}

MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find().pretty()
{
"_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"),
"name" : "Tim's Tasty Treats",
"boro" : "Manhattan"
}
```

Adding *pretty* to *find* prints the document with a nice format

Note: _id not in the insert, so MongoDB created one
If insert provides _id, that value is used

Insert several documents at the same time by using a JSON array

db.Collection.insert([{document},{document}…])

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.insert([{name:"Tim's Untasty Treats",boro:"Queens"},{name: "Brooklyn's Best Restaurant",boro:"Brooklyn"},{name:"Bronx diner",boro:"Bronx"}])
BulkWriteResult({
"writeErrors" : [ ],
"writeConcernErrors" : [ ],
"nInserted" : 3,
"nUpserted" : 0,
"nMatched" : 0,
"nModified" : 0,
"nRemoved" : 0,
"upserted" : [ ]
})
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find()
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c557"), "name" : "Brooklyn's Best Restaurant", "boro" : "Brooklyn" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c558"), "name" : "Bronx diner", "boro" : "Bronx" }
```

Insert multiple document as a JSON array of Objects

Three new documents inserted

upsert is a document that overwrites an existing document

Also *insertOne* and *insertMany* commands available

find returns documents matching the provided criteria

db.Collection.find({criteria})

***findOne* returns only one document**

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find()
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c557"), "name" : "Brooklyn's Best Restaurant", "boro" :
"Brooklyn" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c558"), "name" : "Bronx diner", "boro" : "Bronx" }

MongoDB Enterprise Cluster0-shard-0:PRIMARY>
db.Restaurants.find({boro:"Manhattan"}).pretty()
{
"_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"),
"name" : "Tim's Tasty Treats",
"boro" : "Manhattan"
}
```

Find restaurants in Manhattan
Only returns one document in this case

Use *$or, $and, $nor, $not* for multiple criteria
Must put criteria in an array

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find({$or:
[{boro:"Manhattan"},{boro:"Queens"}]})
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
```

find can also project fields and limit the number of documents returned

db.Collection.find({criteria},{field:1, field:0})

1 is returned, 0 is not returned
If project criteria not provided, assume 0 (except for _id)

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find({},{"name":1})
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c557"), "name" : "Brooklyn's Best Restaurant" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c558"), "name" : "Bronx diner", "boro" : "Bronx" }

MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find({},{"name":1,_id:0})
{ "name" : "Tim's Tasty Treats" }
{ "name" : "Tim's Untasty Treats" }
{ "name" : "Brooklyn's Best Restaurant" }
```

Skip _id by setting it to 0

Return only two results

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find({}).limit(2)
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan"}
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
```

Add *sort* to sort results

db.Collection.find({criteria},{field:1, field:0}).sort({key:1, key:-1})

1 sorted in ascending order
-1 sorted in descending order

Ascending sort on boro

MongoDB Enterprise Cluster0-shard-0:PRIMARY> **db.Restaurants.find({}).sort({boro:1})**

```
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c557"), "name" : "Brooklyn's Best Restaurant", "boro" :
"Brooklyn" }
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan”}
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
```

Descending sort on boro

MongoDB Enterprise Cluster0-shard-0:PRIMARY> **db.Restaurants.find({}).sort({boro:-1})**

```
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan"}
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c557"), "name" : "Brooklyn's Best Restaurant", "boro" :
"Brooklyn" }
```

update can add new fields or update existing field values

db.Collection.update({criteria},{updated data})

MongoDB Enterprise Cluster0-shard-0:PRIMARY> **Add new field for score**

db.Restaurants.update({_id:ObjectId("5ebfe42a6ead6c5a3b84c554")},{$set:{score:5}})

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

MongoDB Enterprise Cluster0-shard-0:PRIMARY> **db.Restaurants.find({boro:"Manhattan"})**

{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan", **"score" : 5** }

MongoDB Enterprise Cluster0-shard-0:PRIMARY> **Can also update field values**

db.Restaurants.update({_id:ObjectId("5ebfe42a6ead6c5a3b84c554")},{$set:{score:6}})

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

MongoDB Enterprise Cluster0-shard-0:PRIMARY> **db.Restaurants.find({boro:"Manhattan"})**

{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan", **"score" : 6** }

Use *remove* to delete documents

Format: db.Collection.remove({criteria})

Add score to Bronx Diner

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.update({_id:ObjectId("5ebfe8c46ead6c5a3b84c558")},{$set:{score:12}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find()
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan", "score" : 6 }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c557"), "name" : "Brooklyn's Best Restaurant", "boro" : "Brooklyn" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c558"), "name" : "Bronx diner", "boro" : "Bronx", "score" : 12 }
```

Remove all documents with score greater than 6

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.remove({score: {$gt: 6}})
WriteResult({ "nRemoved" : 1 })
```

Can use $eq, $lt, $gt, $lte, $gte, $ne, $in, and $nin

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.find()
{ "_id" : ObjectId("5ebfe42a6ead6c5a3b84c554"), "name" : "Tim's Tasty Treats", "boro" : "Manhattan", "score" : 6 }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c556"), "name" : "Tim's Untasty Treats", "boro" : "Queens" }
{ "_id" : ObjectId("5ebfe8c46ead6c5a3b84c557"), "name" : "Brooklyn's Best Restaurant", "boro" : "Brooklyn" }
```

Use db.Collection.remove({}) to remove all documents, like TRUNCATE

Use *createIndex* to add an index on a field for faster lookup like a RDMS

db.Collection.createIndex({field:1})

1 ascending, -1 descending

```
MongoDB Enterprise Cluster0-shard-0:PRIMARY> db.Restaurants.createIndex({boro:1})
{
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
"ok" : 1,
"$clusterTime" : {
"clusterTime" : Timestamp(1589640477, 2),
"signature" : {
"hash" : BinData(0,"xI/A6/Ux5me+MXwxK0Uj+UASLwE="),
"keyId" : NumberLong("6826973194142875651")
}
},
"operationTime" : Timestamp(1589640477, 2)
}
```

Practice

1. **Design a MongoDB database for restaurant inspections from our running example**
   - Each restaurant has:
     - Name, Boro, Cuisine
   - Each restaurant can be inspected many times, each inspection has
     - Date, Score, zero or more violations
   - Each violation has
     - Code (e.g., 06N), Description (e.g., “Improperly cleaned food surface”)
2. **Insert several restaurants into your database**
   - Name: Morris Park Bake Shop, boro: Bronx, Cuisine: Bakery
     - Inspected on 5/5/2025, score 10, violations 06N, 08B
     - Inspected on 4/4/2024, score 12, violations 12C
   - Name: Tim’s Tasty Treats, boro: Manhattan, Cuisine: Fruits/Vegetables
     - Inspected on 3/3/2023, score 5, violations none
     - Inspected on 2/2/2022, score 7, violations 08B
3. **Query your database**
   - Find all bakeries
   - Find just the names of all restaurants inspected after 5/1/2023
   - Find all inspections that had a violation code of 08B

Agenda

1. Data warehouses
2. Distributed systems
3. MongoDB overview
4. MongoDB queries
5. Restaurant example
6. Course wrap up

Course summary

I hope you've taken away four things from this course:

1. How to query existing databases for insight
   - SQL: SELECT * FROM table
2. How to design your own databases
   - ER models, normalization
   - Security/APIs
3. What happens under the hood
   - Persistence
   - Query optimization
   - Transactions/concurrency
4. Alternatives

Pierson's conjecture

Every non-trivial application has a database component

Good luck

- Let me know if I can be helpful in the future
- Say “Hi”
- Look me up on Linked In if you’d like
- Good luck to you all, it’s been my pleasure

Add new items to array using $push

```
> db.Students.find().pretty()
{
        "_id" : ObjectId(”ABC"),
        "name" : "Alice",
        "year" : 27,
        "GPA" : 3.5,
        "grades" : [
                {
                "class" : "CS1",
                "grade" : "A"
                },
                {
                "class" : "CS10",
                "grade" : "A-"
                }
        ]
}
```

Add new grade for Alice

```
db.Students.update(
        {name:"Alice"},
        {$push:
                {
                        grades:{
                                class:"CS61",
                                grade:"A”
                        }
                }
        }
)
```

Find document to update

Add entry to grades array using $push

Add new items to array using $push

```
> db.Students.find().pretty()
{
        "_id" : ObjectId(”ABC"),
        "name" : "Alice",
        "year" : 27,
        "GPA" : 3.5,
        "grades" : [
                {
                "class" : "CS1",
                "grade" : "A"
                },
                {
                "class" : "CS10",
                "grade" : "A-"
                }
                {
                “class” : “CS61”,
                “grade” : “A”
                }
        ]
}
```

Add new grade for Alice

```
db.Students.update(
        {name:"Alice"},
        {$push:
                {
                        grades:{
                                class:"CS61",
                                grade:"A”
                        }
                }
        }
)
```

Find document to update

Add entry to grades array using $push

Grade for CS61 added

Use $pull to remove item from array

Access embedded document using “dot notation”

```
> db.Students.find().pretty()
{
        "_id" : ObjectId(”ABC"),
        "name" : "Alice",
        "year" : 27,
        "GPA" : 3.5,
        "grades" : [
                {
                "class" : "CS1",
                "grade" : "A"
                },
                {
                "class" : "CS10",
                "grade" : "A-"
                }
                {
                “class” : “CS61”,
                “grade” : “A”
                }
        ]
}
```

Find students who got an A in CS61

```
db.Students.find(
        {
                "grades.class":"CS61",
                "grades.grade":"A”
        }
)

// Returns Alice document
```

Reference fields in grades array with dot notation