FULL STACK WEB DEVELOPMENT PROGRAM

Lecture 8: Databases

BeaconFire

OUTLINE

- Databases
- Data Modeling
- SQL vs NoSQL
- Mongoose

DATABASES

- **Database Management System (DBMS)**: software that facilitates data storage, retrieval, modification, and deletion.
  - **Relational (RDBMS, SQL)**: Oracle, MySQL, PostgreSQL, etc.
  - **Non-relational (NoSQL)**: MongoDB, Redis, Neo4j, ElasticSearch

- **Data modeling**: the process of analyzing data requirements and defining their relationships.
  - One-to-one (1:1)
  - One-to-many (1:N)
  - Many-to-many (N:N)

DATA MODELING EXAMPLE

- Say you want to recreate Spotify with the following features:
  - Users can browse the most popular songs based on their language and genre.
  - Users can like a song.
  - Users can search for songs based on the artist name or song title.
  - Users can follow an artist.
  - Users can view & edit their own profile.

- How many entities should there be? What properties should they have?
  - **User**: userId, firstName, lastName, gender, location

DATA RELATIONSHIPS EXAMPLE

- **User**: uid, first name, last name, liked songs, followed artists, playlists
  Song: sid, name, data, duration, category, language, artist, popularity
  Playlist: pid, name, list of songs

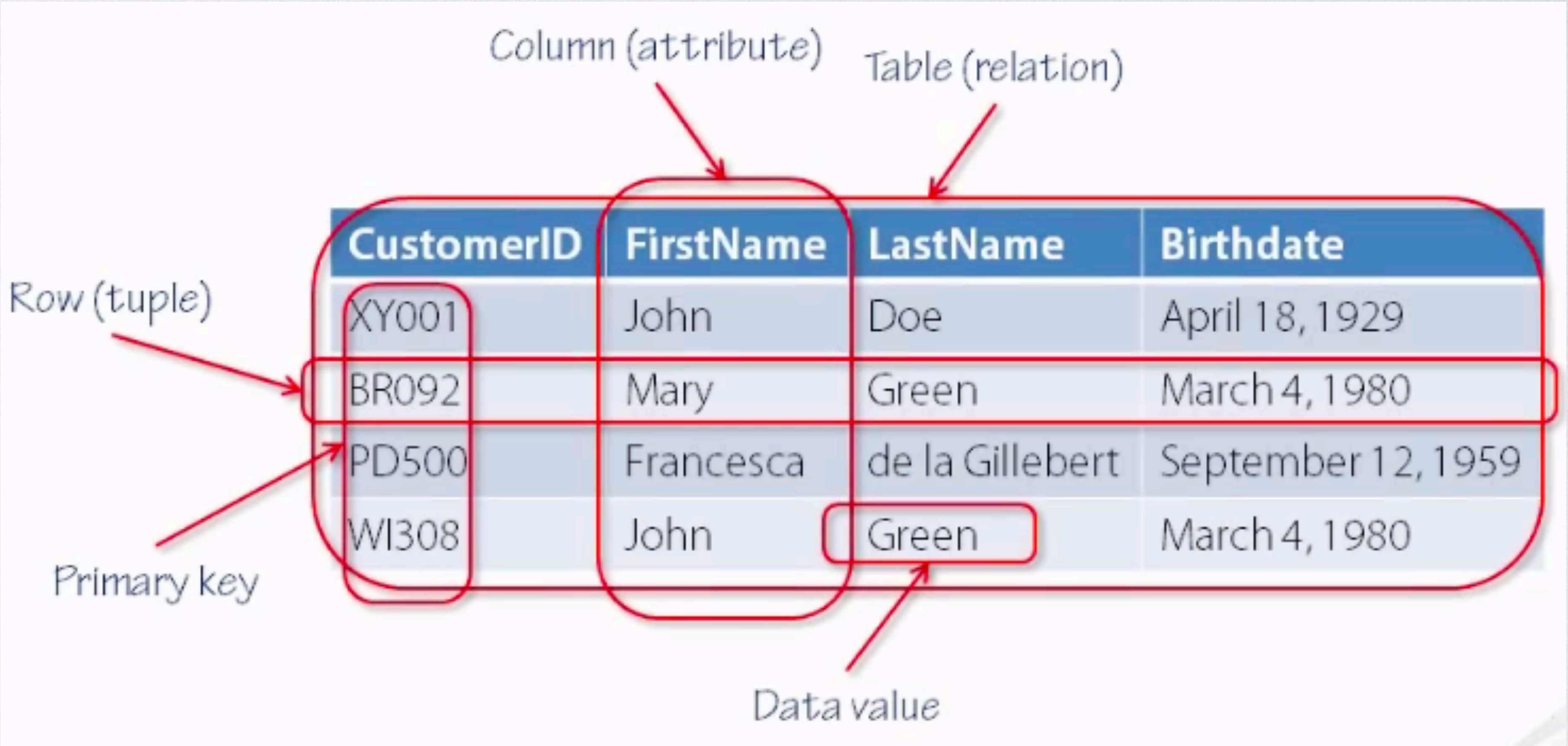
- What's the relationship between our entities? Look at it from both perspectives.
  - user & song
  - user & playlist
  - song & category
  - songs & artist
  - songs & language

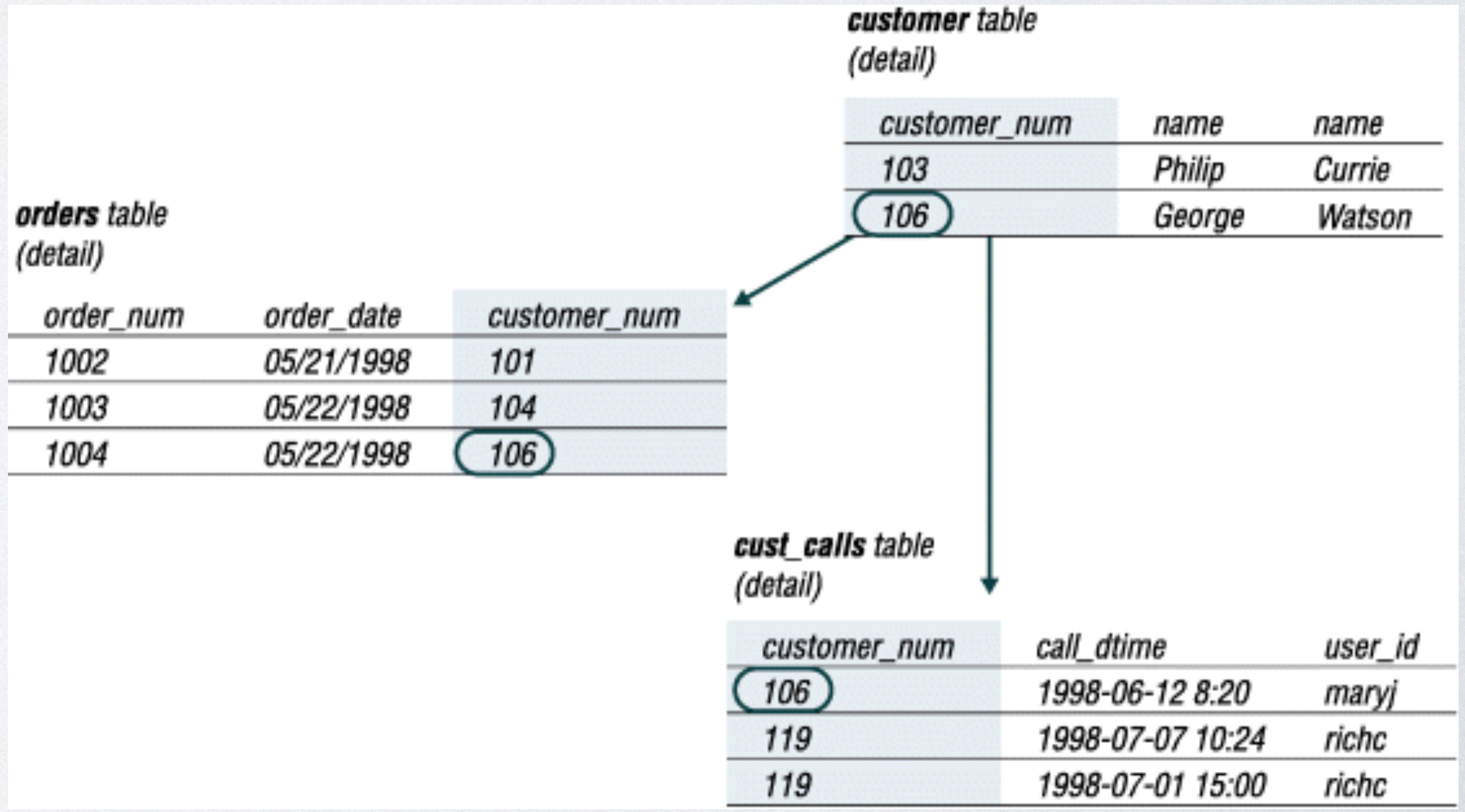
RDBMS

- **Relational model**: a standardized way to represent and query data, which involves separating data into tables (relations).
  - **Row**: an entry in the table.
  - **Column**: represents an attribute of the data.
  - **Primary key**: the column(s) that uniquely identifies a record in a table.
  - **Foreign key**: the column(s) of one table that are the primary key of another table.

- **Referential integrity**: the RDBMS ensures that all foreign keys will be valid.

- RDBMS Features: strict & pre-defined structure, normalization, transactions, ACID





NORMALIZATION

- **Normalization**: organizing datasets efficiently by splitting them to prevent data redundancy and avoid anomalies (database inconsistencies).

- Anomalies
  - **Update**: when you don't properly update all the related entries
  - **Delete**: when you delete one thing and unintentionally delete more data
  - **Insert**: when you can't add data because you don't have other data

Employee_ID	Name	Department	Student_Group
123	J. Longfellow	Accounting	Beta Alpha Psi
234	B. Rech	Marketing	Marketing Club
234	B. Rech	Marketing	Management Club
456	A. Bruchs	CIS Update 1	Technology Org.
456	A. Bruchs	CIS Update 2	Beta Alpha Psi

Employee_ID	Name	Department	Student_Group
123	J. Longfellow	Accounting	~~Beta Alpha Psi~~ Delete
234	B. Rech	Marketing	Marketing Club
234	B. Rech	Marketing	Management Club
456	A. Bruchs	CIS	Technology Org.
456	A. Bruchs	CIS	Beta Alpha Psi

SQL

- **Structured Query Language (SQL)**: the most widely used querying language for reading and writing data in RDBMS.

- **Object Relational Mapping (ORM)**: technique for querying/manipulating RDBMS using the object model.

- Given a user's first name, find all of the songs that they liked.
  select user.firstname, song.name
  from user **join** user_song on user.uid=user_song.uid
  join song on user_song.sid = song.sid
  where user.firstname = 'XXX'

- ORM: Users.join(songs, uid="").select()

user: uid, first name, last name
user_song: uid, sid
user_artist: uid, aid
playlist: pid, uid, playlist name
playlist_song: pid, sid
song: sid, name, data, duration, category, language, popularity, aid
artist: aid, name, description

TRANSACTIONS

- **Transaction**: the execution of 1+ statements as a set.
  - If successful, all the changes are committed (applied to the table).
  - If error, all changes changes are erased or rolled back.

- Transactions must be controlled to ensure data integrity. Their changes should be:
  - **Atomic**: executed as a single unit and applies all the changes or none at all.
  - **Consistent**: leaving the data in a consistent state and following all constraints.
  - **Isolated**: independent from & invisible to other transactions.
  - **Durable**: permanent after being committed.

- https://www.c-sharpcorner.com/UploadFile/84c85b/understanding-transactions-in-sql-server/

NOSQL

- **Non-relational databases (NoSQL)**: databases with storage models that are optimized for specific data requirements & scalability (distributable).
  - Does not use tables or primary/foreign keys, etc.
  - Cheaper to maintain/operate, de-normalized (redundancy), better throughput

- **Document**: manages a set of field-value pairs in a "document", usually stored in the form of binary JSON (BSON).
  - MongoDB, CouchDB

  Key-value: manages a collection of key-value pairs contained within a single object.
  - Redis, DynamoDB

  Graph: manages a collection of greatly interconnected entities.
  - Neo4j, Amazon Neptune

  Columnar: manages data in columns (conceptually similar to RDBMS).
  - Cassandra, HBase

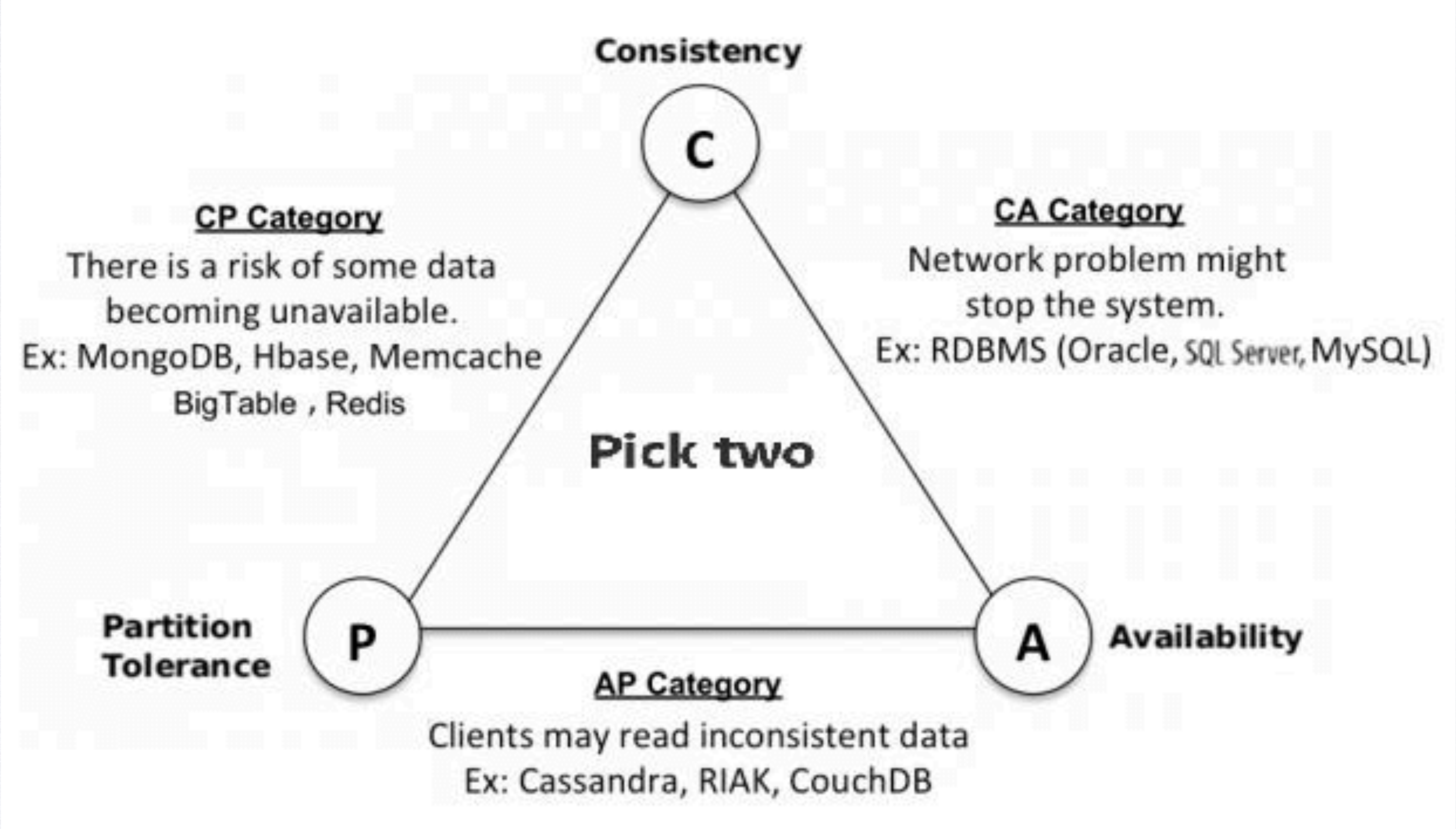
CAP

- **Consistency**: all clients see the same, most updated data at any time.
  Availability: any client that sends a request will receive a response.
  Partition Tolerance: (required) the system still works even when some connections are lost.

- **Brewer's CAP Theorem**: during network failure, distributed systems & data stores can only guarantee two, not all of CAP.
  - CP: MongoDB, Redis
  - AP: CouchDB, Cassandra

- **Eventual consistency**: some time after an update the data in multiple databases will be consistent.
  - ex: order the last few items in stock

- https://www.scylladb.com/glossary/cap-theorem

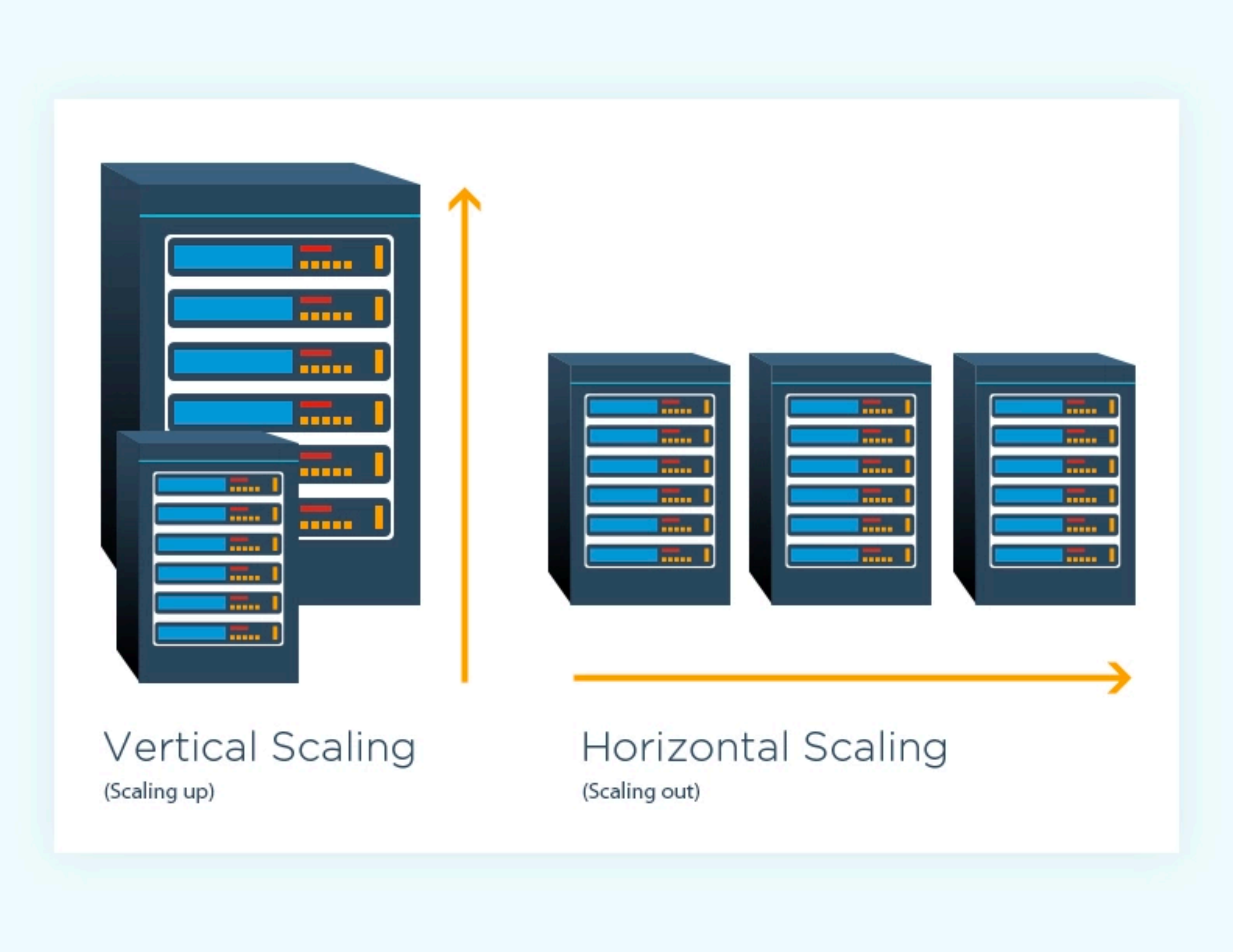


SCALABILITY

- When do we need to scale? Size of data, query rate

- **Vertical scaling**: upgrade a server with better CPU, more RAM, or more storage space (SSD).
  - Pros: Maintains application architecture
  - Cons: Expensive, technological limitations, cloud-based restrictions

- **Horizontal scaling**: distribute the dataset and query load over multiple servers that handle a subset of the overall work.
  - Pros: Lower cost
  - Cons: Changes application architecture, complexity & maintenance



SCALABILITY (2)

- **Replication**: make copies of a master database and host them on separate servers, resulting in "slave" databases.
  - Read data from any slave, but only write to master
  - **Fault tolerance**: ability of the system to continue operating even during partial failure

- **Partitioning**: splitting datasets into smaller subsets, all on the same server.
  - **Horizontal (rows)**: divide the number of rows into different tables (same shape/structure)
  - **Vertical (columns)**: divide the number of columns into different tables (different shape/structure)

- **Sharding**: spreading a dataset across multiple databases that are distributed across multiple servers, none of which communicate with each other.
  - Strategies: key, range, dictionary, hierarchy, entity groups

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

DATABASE COMPARISON

	Relational	Non-relational
Schemas	Static, Pre-defined	Dynamic
Focus	Consistency Business transaction requirements	Scalability & Availability
Querying Language	SQL	Depends
Performance	Multiple entities	Single entity
Scalability	vertical horizontal/vertical partitioning	horizontal sharding
Properties	ACID, joins	CAP

MONGODB

- **MongoDB**:
  - document-based data model naturally supports JSON
  - Ad-hoc queries for optimized, real-time analytics
  - Indexing appropriately for better query executions
  - Replication for better data availability and stability
  - Sharding
  - Load balancing

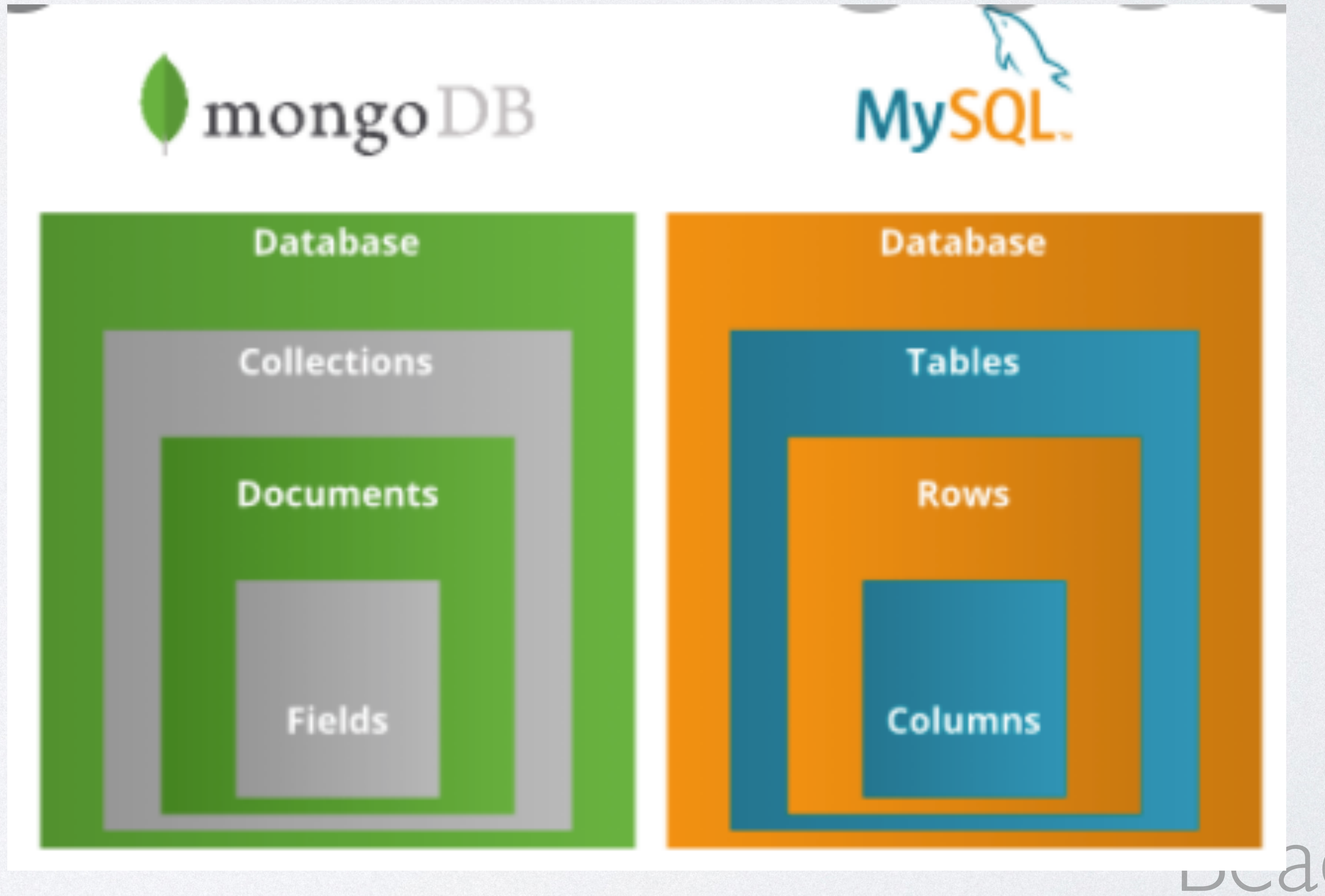
- **Object Document Mapping (ODM)**: technique for querying/manipulating document-based databases using the object model.
  - Mongoose

- https://www.mongodb.com/docs/atlas/getting-started/

COLLECTIONS

- **Document**: object of field:value pairs that represent the entity.
  - BSON

- **Collection**: the group of all documents for a specific entity.
  - Like tables in RDBMS
  - Data that is accessed together should be stored together



CONNECTING TO MONGODB

- **Connection URL**: the set of instructions that the Node.js driver uses to connect to a MongoDB deployment.
  - **client.close()**: close the client when you finish, or the process hangs

- **Connection pool**: a cache of authenticated database connections maintained by your driver.
  - Controls the rate and number of connections

BeaconFire

DOTENV

- **dotenv**: an npm library for configuring environment variables, which shouldn't be uploaded publicly.
  - Store key-value pairs in **.env** file
  - credentials, API keys, environment type

```
MONGO_URL="mongodb+srv://test:test@[redacted]/demo-db?retryWrites=true&
w=majority"
PORT="3000"
```

MONGODB QUERIES

- https://docs.mongodb.com/manual/crud/
  http://mongodb.github.io/node-mongodb-native/3.6/reference/ecmascriptnext/crud/

- **C**reate
  - insert(), insertOne(), insertMany()
  Read
  - find(), findOne(), findOneAndUpdate()
  Update
  - update(), updateOne(), updateMany(), replaceOne()
  Delete
  - deleteOne(), deleteMany()

MONGOOSE

- **Mongoose**: a MongoDB ODM that is designed to work in an asynchronous environment.
  - Supports promises & callbacks
  - Buffers queries until server is connected to MongoDB
  - Data validation

- https://mongoosejs.com/docs/index.html

SCHEMA

- **Schema**: an abstract representation of a MongoDB collection that defines the shape and types of the documents in that collection.

- **SchemaTypes**: String, Number, Date, Buffer, Boolean, Mixed, ObjectId, Array, Decimal128, Map

- Validation and error messages
  - Numbers: min and max validator
  - String: enum, match, minLength, maxLength
  - Custom validators: declared by passing a validation function

- https://mongoosejs.com/docs/guide.html

BeaconFire

MODEL

- **Model**: objects created based on Schema definition
  - Instances of a model are documents.
  - **mongoose.model**(modelName, schema, collectionName)

- https://mongoosejs.com/docs/queries.html

MODEL RELATIONSHIPS

- **Embedded**: documents of one model contain nested documents of another model.
  Reference: documents of one model contain references (based on document IDs) to other documents of another model.
- **One-to-one (1:1)**: user and profile
- **One-to-many (1:N)**: user and posts
- **Many-to-many (N:N)**: user and forums
  - Better to use references
- https://www.techighness.com/post/no-sql-data-modeling-1-to-1-1-to-many-many-to-many/

REF AND POPULATE

- **Reference**: the model name that a schema property refers to.
  - query**.populate()**: substitute the reference with data from that collection

- https://mongoosejs.com/docs/populate.html
  https://www.mongodb.com/docs/manual/reference/operator/update/

```
const Schema = mongoose.Schema;
const refType = Schema.Types.ObjectId;

// Define your schemas
const UserSchema = new Schema({
  username: { type: String, required: true },
  email: { type: String, required: true },
  profile: { type: refType, ref: "Profile" }
});

const ProfileSchema = new Schema({
  picture: String,
  karma: Number,
  birthday: Date
});
```

RELATIONSHIP COMPARISON

	Embedded	Reference
Normalization	De-normalized	Normalized
Access	One query	Subqueries
Use-cases	- Document is retrieved together with parent - Mostly unique data	- Rarely retrieved with parent - Static, but parent is updated - Document size limit

ANY QUESTIONS?